

libflame

The

Complete

Reference

Field G. Van Zee

The University of Texas at Austin

Copyright © 2009 by Field G. Van Zee.

10 9 8 7 6 5 4 3 2 1

All rights reserved. No part of this book may be reproduced, stored, or transmitted in any manner without the written permission of the publisher. For information, contact either of the authors.

No warranties, express or implied, are made by the publisher, authors, and their employers that the programs contained in this volume are free of error. They should not be relied on as the sole basis to solve a problem whose incorrect solution could result in injury to person or property. If the programs are employed in such a manner, it is at the user's own risk and the publisher, authors, and their employers disclaim all liability for such misuse.

Trademarked names may be used in this book without the inclusion of a trademark symbol. These names are used in an editorial context only; no infringement of trademark is intended.

Library of Congress Cataloging-in-Publication Data not yet available

Draft, November 2008

This "Draft Edition" allows this material to be used while we sort out through what mechanism we will publish the book.

Contents

1. Introduction	1
1.1. What's provided	1
1.2. What's not provided	6
1.3. Acknowledgments	8
2. Setup for GNU/Linux and UNIX	9
2.1. Before obtaining <code>libflame</code>	9
2.1.1. System software requirements	9
2.1.2. System hardware support	10
2.1.3. License	10
2.1.4. Source code	10
2.1.5. Tracking source code revisions	11
2.1.6. If you have problems	12
2.2. Obtaining <code>libflame</code>	12
2.3. Preparation	12
2.4. Configuration	12
2.4.1. <code>configure</code> options	14
2.4.2. Running <code>configure</code>	17
2.5. Compiling	19
2.5.1. Parallel <code>make</code>	20
2.6. Installation	20
2.7. Linking against <code>libflame</code>	22
2.7.1. Linking with <code>liblapack2flame</code>	24
2.7.2. A word on LAPACK routines	25
3. Setup for Microsoft Windows	27
3.1. Before obtaining <code>libflame</code>	27
3.1.1. System software requirements	27
3.1.2. System hardware support	28
3.1.3. License	28
3.1.4. Source code	28
3.1.5. Tracking source code revisions	29
3.1.6. If you have problems	29
3.2. Obtaining <code>libflame</code>	29
3.3. Preparation	29
3.4. Configuration	31
3.4.1. IronPython	32
3.4.2. Running <code>configure.cmd</code>	33
3.5. Compiling	34
3.6. Installation	35
3.7. Linking against <code>libflame</code>	37

4. Using libflame	39
4.1. FLAME/C examples	39
4.2. FLASH examples	42
4.3. SuperMatrix examples	42
5. User-level Application Programming Interfaces	47
5.1. Conventions	47
5.1.1. General terms	47
5.1.2. Notation	48
5.1.3. Objects	51
5.2. FLAME/C Basics	52
5.2.1. Initialization and finalization	52
5.2.2. Object creation and destruction	52
5.2.3. General query functions	53
5.2.4. Interfacing with conventional matrix arrays	56
5.3. Managing Views	61
5.3.1. Vertical partitioning	61
5.3.2. Horizontal partitioning	62
5.3.3. Bidirectional partitioning	63
5.3.4. Merging views	65
5.4. FLASH	66
5.4.1. Motivation	66
5.4.2. Concepts	67
5.4.3. Interoperability with FLAME/C	68
5.4.4. Object creation and destruction	69
5.4.5. Interfacing with flat matrix objects	72
5.4.6. Interfacing with conventional matrix arrays	78
5.4.7. Object query functions	81
5.5. SuperMatrix	82
5.5.1. Overview	82
5.5.2. API	83
5.5.3. Integration with FLASH front-ends	86
5.6. Utility functions	86
5.6.1. Advanced query functions	86
5.6.2. Assignment/Update functions	89
5.6.3. Math-related functions	90
5.6.4. Miscellaneous functions	99
5.7. Front-ends	100
5.7.1. BLAS operations	101
5.7.1.1. Level-1 BLAS	101
5.7.1.2. Level-2 BLAS	115
5.7.1.3. Level-3 BLAS	127
5.7.2. LAPACK operations	136
5.7.3. Utility functions	150
5.8. External wrappers	159
5.8.1. BLAS operations	159
5.8.1.1. Level-1 BLAS	159
5.8.1.2. Level-2 BLAS	168
5.8.1.3. Level-3 BLAS	175
5.8.2. LAPACK operations	182
5.8.3. LAPACK-related utility functions	191
5.9. LAPACK compatibility (<code>liblapack2flame</code>)	192
5.9.1. Supported routines	192

6. Developer Application Programming Interfaces	195
6.1. Locks	195
6.1.1. API	195
6.2. Memory management	197
6.3. Object creation	198
6.4. SuperMatrix	198
6.5. Control trees	199
6.5.1. Motivation	199
6.5.2. The solution in <code>libflame</code>	199
6.5.3. Structure fields	200
6.5.4. Control tree API	202
6.5.4.1. Blocksize structures	202
6.5.4.2. Level-3 BLAS operations	206
6.5.4.3. LAPACK-level operations	214
6.5.4.4. Miscellaneous operations	223
6.5.5. Default control trees	224
6.5.6. Operation front-ends	224
6.5.7. Internal back-ends	224
6.5.7.1. Level-3 BLAS operations	224
6.5.7.2. LAPACK operations	229
6.5.8. Algorithmic variants	231
6.6. Parameter and error checking	233
6.6.1. Linear algebra parameters	233
6.6.2. Datatypes	235
6.6.3. Element types	238
6.6.4. Object dimensions	239
6.6.5. UNIX file I/O	242
6.6.6. Operation-specific errors	244
6.6.7. Other system errors	245
6.6.8. Misc. errors	246
A. FLAME Project Related Publications	249
A.1. Books	249
A.2. Dissertations	249
A.3. Journal Articles	249
A.4. Conference Papers	250
A.5. FLAME Working Notes	251
A.6. Other Technical Reports	254
B. License	255
B.1. GNU Lesser General Public License	255

List of Contributors

A large number of people have contributed, and continue to contribute, to the FLAME project. For a complete list, please visit

<http://www.cs.utexas.edu/users/flame/>

Below we list the people who have contributed directly to the knowledge and understanding that is summarized in this text.

Paolo Bientinesi
The University of Texas at Austin

Ernie Chan
The University of Texas at Austin

John A. Gunnels
IBM T.J. Watson Research Center

Kazushige Goto
The University of Texas at Austin

Tze Meng Low
The University of Texas at Austin

Margaret E. Myers
The University of Texas at Austin

Enrique S. Quintana-Ortí
Universidad Jaume I

Gregorio Quintana-Ortí
Universidad Jaume I

Robert A. van de Geijn
The University of Texas at Austin

Chapter 1

Introduction

In past years, the FLAME project, a collaborative effort between The University of Texas at Austin and Universidad Jaime I de Castellon, developed a unique methodology, notation, and set of APIs for deriving and representing linear algebra libraries. In an effort to better promote the techniques characteristic to the FLAME project, we have implemented a functional prototype library that demonstrates findings and insights from the last decade of research. We call this library *libflame*.¹

The primary purpose of `libflame` is to provide the scientific and numerical computing communities with a modern, high-performance dense linear algebra library that is extensible, easy to use, and available under an open source license. Its developers have published numerous papers and working notes over the last decade documenting the challenges and motivations that led to the APIs and implementations present within the `libflame` library. Most of these publications listed in Appendix A. Seasoned users within scientific and numerical computing circles will quickly recognize the general set of functionality targeted by `libflame`. In short, in `libflame` we wish to provide not only a framework for developing dense linear algebra solutions, but also a ready-made library that is, by almost any metric, easier to use and offers competitive (and in many cases superior) real-world performance when compared to the more traditional LAPACK and BLAS libraries [?, ?, ?, ?, ?].

1.1 What’s provided

The FLAME project is excited to offer potential users numerous reasons to adopt `libflame` into their software solutions.

A solution based on fundamental computer science. The FLAME project advocates a new approach to developing linear algebra libraries. It starts with a more stylized notation for expressing loop-based linear algebra algorithms [?, ?, ?, ?]. This notation closely resembles how matrix algorithms are naturally illustrated with pictures. (See Figure 1.1 and Figure 1.2 (left).) The notation facilitates rigorous formal derivation of algorithms [?, ?, ?], which guarantees that the resulting algorithms are correct.

Object-based abstractions and API. The BLAS, LAPACK, and ScaLAPACK [?] projects place backward compatibility as a high priority, which hinders progress towards adopting modern software engineering principles such as object abstraction. `libflame` is built around opaque structures that hide implementation details of matrices, such as leading dimensions, and exports object-based programming interfaces to operate upon these structures [?]. Likewise, FLAME algorithms are expressed (and coded) in terms of smaller operations on sub-partitions of the matrix operands. This abstraction facilitates programming without array or loop indices, which allows the user to avoid painful index-related programming errors altogether. Figure 1.2 compares the coding styles of `libflame` and LAPACK, highlighting the inherent elegance of FLAME

¹Henceforth, we will typeset the name of the library in a fixed-width font, just as we typeset the names of executable programs and scripts.

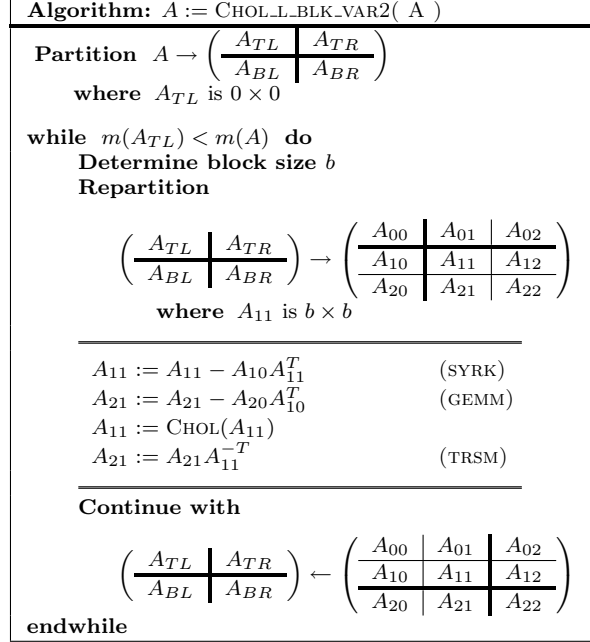


Figure 1.1: Blocked Cholesky factorization (variant 2) expressed as a FLAME algorithm. Subproblems annotated as SYRK, GEMM, and TRSM correspond to Level-3 BLAS operations.

code and its striking resemblance to the corresponding FLAME algorithm shown in Figure 1.1. This similarity is quite intentional, as it preserves the clarity of the original algorithm as it would be illustrated on a white-board or in a publication.

Educational value. Aside from the potential to introduce students to formal algorithm derivation, FLAME serves as an excellent vehicle for teaching linear algebra algorithms in a classroom setting. The clean abstractions afforded by the API also make FLAME ideally suited for instruction of high-performance linear algebra courses at the undergraduate and graduate level. Robert van de Geijn routinely uses FLAME in his linear algebra and numerical analysis courses. Historically, the BLAS/LAPACK style of coding has been used in these pedagogical settings. However, we believe that coding in that manner obscures the algorithms; students often get bogged down debugging the frustrating errors that often result from indexing directly into arrays that represent the matrices.

A complete dense linear algebra framework. Like LAPACK, `libflame` provides ready-made implementations of common linear algebra operations. The implementations found in `libflame` mirror many of those found in the BLAS and LAPACK packages. However, `libflame` differs from LAPACK in two important ways: First, it provides families of algorithms for each operation so that the best can be chosen for a given circumstance [?]. Second, it provides a framework for building complete custom linear algebra codes. We believe this makes it a more useful environment as it allows the user to quickly chose and/or prototype a linear algebra solution to fit the needs of the application.

High performance. In our publications and performance graphs, we do our best to dispel the myth that user- and programmer-friendly linear algebra codes cannot yield high performance. Our FLAME implementations of operations such as Cholesky factorization and triangular matrix inversion often outperform the corresponding implementations currently available in LAPACK [?]. Figure 1.3 shows an example of the performance increase made possible by using `libflame` for a Cholesky factorization, when compared to LAPACK. Many instances of the `libflame` performance advantage result from the fact that LAPACK provides only one variant (algorithm) of most operations, while `libflame` provides many variants. This allows the

libflame	LAPACK
<pre> FLA_Error FLA_Chol_1_blk_var2(FLA_Obj A, dim_t nb_alg) { FLA_Obj ATL, ATR, A00, A01, A02, ABL, ABR, A10, A11, A12, A20, A21, A22; dim_t b; int value; FLA_Part_2x2(A, &ATL, &ATR, &ABL, &ABR, 0, 0, FLA_TL); while (FLA_Obj_length(ATL) < FLA_Obj_length(A)) { b = min(FLA_Obj_length(ABR), nb_alg); FLA_Repart_2x2_to_3x3(ATL, /**/ ATR, &A00, /**/ &A01, &A02, /* ***** */ /* ***** */ &A10, /**/ &A11, &A12, ABL, /**/ ABR, &A20, /**/ &A21, &A22, b, b, FLA_BR); /* ----- */ FLA_Syrk(FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_MINUS_ONE, A10, FLA_ONE, A11); FLA_Gemm(FLA_NO_TRANSPOSE, FLA_TRANSPOSE, FLA_MINUS_ONE, A20, A10, FLA_ONE, A21); value = FLA_Chol_unb_external(FLA_LOWER_TRIANGULAR, A11); if (value != FLA_SUCCESS) return (FLA_Obj_length(A00) + value); FLA_Trsm(FLA_RIGHT, FLA_LOWER_TRIANGULAR, FLA_TRANSPOSE, FLA_NONUNIT_DIAG, FLA_ONE, A11, A21); /* ----- */ FLA_Cont_with_3x3_to_2x2(&ATL, /**/ &ATR, A00, A01, /**/ A02, /* ***** */ /* ***** */ &ABL, /**/ &ABR, A20, A21, /**/ A22, FLA_TL); } return value; } </pre>	<pre> SUBROUTINE DPOTRF(UPLO, N, A, LDA, INFO) CHARACTER UPLO INTEGER INFO, LDA, N DOUBLE PRECISION A(LDA, *) DOUBLE PRECISION ONE PARAMETER (ONE = 1.0D+0) LOGICAL UPPER INTEGER J, JB, NB LOGICAL LSAME INTEGER ILAENV EXTERNAL LSAME, ILAENV EXTERNAL DGEMM, DPOTF2, DSYRK, DTRSM, XERBLA INTRINSIC MAX, MIN INFO = 0 UPPER = LSAME(UPLO, 'U') IF(.NOT.UPPER .AND. .NOT.LSAME(UPLO, 'L')) THEN INFO = -1 ELSE IF(N.LT.0) THEN INFO = -2 ELSE IF(LDA.LT.MAX(1, N)) THEN INFO = -4 END IF IF(INFO.NE.0) THEN CALL XERBLA('DPOTRF', -INFO) RETURN END IF INFO = 0 UPPER = LSAME(UPLO, 'U') IF(N.EQ.0) \$ RETURN NB = ILAENV(1, 'DPOTRF', UPLO, N, -1, -1, -1) IF(NB.LE.1 .OR. NB.GE.N) THEN CALL DPOTF2(UPLO, N, A, LDA, INFO) ELSE IF(UPPER) THEN ***** Upper triangular case omitted for purposes of fair comparison. ELSE DO 20 J = 1, N, NB JB = MIN(NB, N-J+1) CALL DSYRK('Lower', 'No transpose', JB, J-1, -ONE, A(J, 1), LDA, ONE, A(J, J), LDA) CALL DPOTF2('Lower', JB, A(J, J), LDA, INFO) IF(INFO.NE.0) GO TO 30 IF(J+JB.LE.N) THEN CALL DGEMM('No transpose', 'Transpose', N-J-JB+1, JB, J-1, -ONE, A(J+JB, 1), LDA, A(J, 1), LDA, ONE, A(J+JB, J), LDA) CALL DTRSM('Right', 'Lower', 'Transpose', 'Non-unit', N-J-JB+1, JB, ONE, A(J, J), LDA, A(J+JB, J), LDA) END IF 20 CONTINUE END IF GO TO 40 30 CONTINUE INFO = INFO + J - 1 40 CONTINUE RETURN END </pre>

Figure 1.2: The algorithm shown in Figure 1.1 implemented with FLAME/C code (left) and Fortran-77 code (right). The FLAME/C code represents the style of coding found in libflame while the Fortran-77 code was obtained from LAPACK.

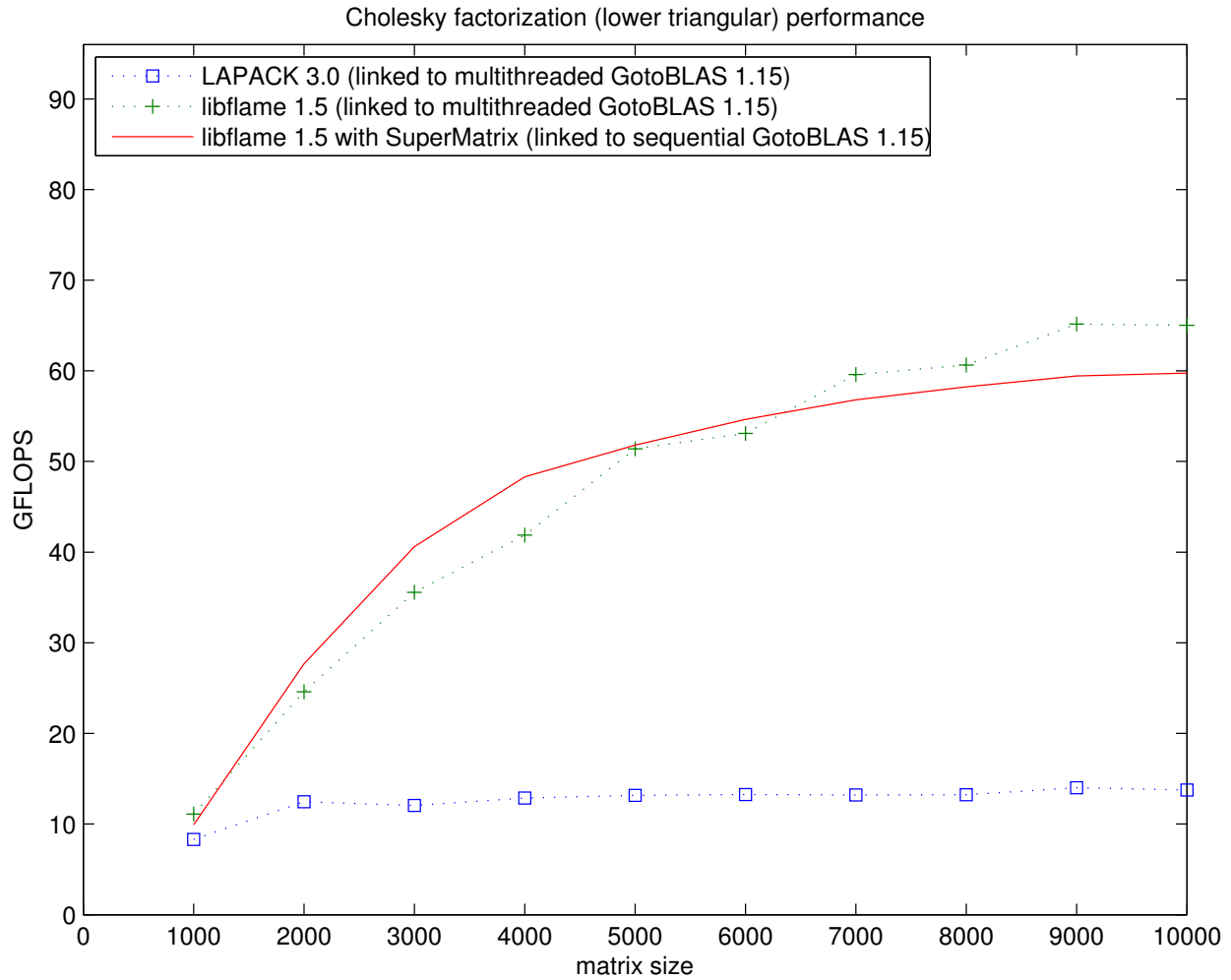


Figure 1.3: Performance of Cholesky factorization implementations measured on a 16 core Itanium2 system. Theoretical peak system performance is 96 GFLOPS. libflame uses variant 3 while LAPACK uses variant 2. For non-SuperMatrix experiments, GotoBLAS was configured to provide multithreaded parallelism for level-3 BLAS operations. For SuperMatrix experiments, GotoBLAS parallelism was disabled.

user and/or library developer to choose which algorithmic variant is most appropriate for a given situation. Currently, **libflame** relies only on the presence of a core set of highly optimized unblocked routines to perform the small sub-problems found in FLAME algorithm codes.

Dependency-aware multithreaded parallelism. Until recently, the most common method of getting shared-memory parallelism from LAPACK routines by simply linking to multithreaded BLAS. This low-level solution requires no changes to LAPACK code but also suffers from sharp limitations in terms of efficiency and scalability for small- and medium-sized matrix problems. The fundamental bottleneck to introducing parallelism directly within many algorithms is the web of data dependencies that inevitably exists between sub-problems. The **libflame** project has developed a runtime system, SuperMatrix, to detect and analyze dependencies found within FLAME algorithms-by-blocks (algorithms whose sub-problems operate only on block operands) [?, ?, ?, ?]. Once dependencies are known, the system schedules sub-operations to independent threads of execution. This system is completely abstracted from the algorithm that is being parallelized and requires virtually no change to the algorithm code, but at the same time exposes abundant high-level parallelism. We have observed that this method provides increased performance for a range of small- and medium-sized problems, as shown in Figure 1.3. The most recent version of LAPACK does not offer any similar mechanism.²

Support for hierarchical storage-by-blocks. Storing matrices by blocks, a concept advocated years ago by Fred Gustavson of IBM, often yields performance gains through improved spatial locality [?, ?, ?]. Instead of representing matrices as a single linear array of data with a prescribed leading dimension as legacy libraries require (for column- or row-major order), the storage scheme is encoded into the matrix object. Here, internal elements refer recursively to child objects that represent sub-matrices. Currently, **libflame** provides a subset of the conventional API that supports hierarchical matrices, allowing users to create and manage such matrix objects as well as convert between storage-by-blocks and conventional “flat” storage schemes [?, ?].

Advanced build system. From its early revisions, **libflame** distributions have been bundled with a robust build system, featuring automatic makefile creation and a configuration script conforming to GNU standards (allowing the user to run the `./configure; make; make install` sequence common to many open source software projects). Without any user input, the configure script searches for and chooses compilers based on a pre-defined preference order for each architecture. The user may request specific compilers via the configure interface, or enable other non-default features of **libflame** such as custom memory alignment, multithreading (via POSIX threads or OpenMP), compiler options (debugging symbols, warnings, optimizations), and memory leak detection. The reference BLAS and LAPACK libraries provide no configuration support and require the user to manually modify a makefile with appropriate references to compilers and compiler options depending on the host architecture.

Windows support. While **libflame** was originally developed for GNU/Linux and UNIX environments, we have in the course of its development had the opportunity to port the library to Microsoft Windows. The Windows port features a separate build system implemented with Python and **nmake**, the Microsoft analogue to the **make** utility found in UNIX-like environments. As of this writing, the port is still very new and therefore should be considered experimental. However, we feel **libflame** for Windows is very close to useable for many in our audience, particularly those who consider themselves experts. We invite interested users to try the software and, of course, we welcome feedback to help improve our Windows support, and **libflame** in general.

Backwards compatibility with LAPACK. We understand that you may have already invested a lot of time in your current dense linear algebra application. That is why we provide a set of compatibility routines that map conventional LAPACK invocations to their corresponding implementations within **libflame**. By

²Some of the lead developers of LAPACK have independently investigated these ideas as part of a spin-off project, PLASMA. However, the project has not yet made their code available for general use [?, ?].

simply linking to `libflame` as explained below, you can take advantage of the performance benefits offered by FLAME with virtually no changes to your application.³

1.2 What's not provided

While we feel that `libflame` is a powerful and effective tool, it is not for everyone. In this section we list reasons you may want to avoid using `libflame`.

Distributed memory parallelism. `libflame` does not currently offer distributed memory parallelism. Some of the FLAME project members once maintained a library called PLAPACK [?, ?], which provided a framework for implementing dense linear algebra operations in a parallel distributed memory environment. However, this library is no longer supported by the FLAME group. We have begun preliminary work on rewriting PLAPACK to incorporate many of the things we've learned while developing the FLAME methodology. But until we can finish this rewrite of PLAPACK, `libflame` will not support parallel distributed memory computing.

Out-of-core computation. `libflame` does not currently support out-of-core computation. However, the FLAME group has published research based on results from a prototype extension to `libflame` [?]. While this prototype extension is not distributed with `libflame`, we believe that FLASH with its hierarchical storage format will provide us with a relatively straightforward path to incorporating out-of-core functionality in the future. Our colleagues at Universidad Jaime I de Castellon, however, have more recent expertise in this area. Those interested in out-of-core functionality should contact them directly.

Sparse matrix functionality. Algorithms implemented in `libflame` do not take advantage of sparseness that may be present within a matrix, nor does it take advantage of any special structure beyond the traditional dense matrix forms (triangular, symmetric, Hermitian), nor does it support special storage formats that avoid storing the zero elements present in sparse matrices. Users looking to operate with sparse matrices, especially those that are large, should look into more specialized software packages that leverage the properties inherent to your application.

Banded matrix support Many routines within the BLAS and LAPACK are specially written to take advantage of banded matrices. As with sparse matrices, these routines expect that the matrix arguments be stored according to a special storage scheme that takes advantage of the sparsity of the matrix. Unfortunately, `libflame` does not offer any storage scheme targeting banded matrices, and thus does not include any routines that leverage such storage within the computation. Though, our colleagues in Spain have reported on work using banded matrices in algorithms-by-blocks [?].

Traditional coding style. We are quite proud of `libflame` and its interfaces, which we believe are much easier to use than those of the BLAS and LAPACK. However, it's entirely possible that switching to the `libflame` API is not feasible for you or your organization. For example, if you are a Fortran programmer, you may not have the patience or the liberty to write and use C wrappers to the `libflame` routines. Or, your project may need to remain written in Fortran for reasons beyond your control. Whatever the case, we understand and appreciate that coding style imposed by `libflame` may be too different for some users and applications.

Interactive/Interpreted programming Some people require a degree of interactivity in their scientific computing environment. Good examples of linear algebra tools that supports interpreted programming are The MathWorks' MATLAB and National Instruments' LabVIEW MathScript. Programming with MATLAB or LabVIEW MathScript is a great way to prototype new ideas and flesh out your algorithms before moving

³ Currently, any operation called through the `liblapack2flame` compatibility layer will execute sequentially. In order to invoke our parallelized implementations, you must use native FLAME interfaces.

them to a high-performance environment. While `libflame` provides many features and benefits, an interpreted programming environment is not one of them. If you require this feature, we encourage you to look at MATLAB as well as other free alternatives, such as Octave and Sage.

Licensing conflict. `libflame` is provided as free software to the general public under the GNU Lesser General Public License, version 2.1, given in Appendix B. However, some organizations prohibit their employees from using (or even looking at) code that is released under GNU licenses. If this applies to you, then you should look for software that is either freely-available⁴, or released under a license which is less restrictive than the LGPL.

Whatever the reason, we acknowledge that it may not be practical or even possible to incorporate `libflame` into your software solution. We hope `libflame` fits your needs, but if it does not then we would like to refer you to other software packages that you may want to consider:

- **BLAS.** The official reference implementation of the BLAS is available through the `netlib` software repository [?]. It implements basic matrix-matrix operations such as general matrix multiply as well as several less computationally intensive operations involving one or more vector operands. The BLAS is freely-available software.
- **LAPACK.** Like the BLAS, the official reference implementation of LAPACK is available through the `netlib` software repository [?]. This library implements many more sophisticated dense linear algebra operations, such as factorizations, linear system solvers, and eigensolvers. LAPACK is freely-available software.
- **ScaLAPACK.** ScaLAPACK was designed by the creators of the BLAS and LAPACK libraries to implement dense linear algebra operations for parallel distributed memory environments. Its API is similar to that of LAPACK and targets mostly Fortran-77 applications, though it may also be accessed from programs written in C. ScaLAPACK is freely available software and available through the `netlib` software repository [?].
- **PETSc.** PETSc, written and maintained by The University of Chicago, provides parallel solvers for PDEs, and other related tools, with bindings for C, C++, Fortran, and Python [?]. PETSc is available from the University of Chicago under a custom GNU-like license.
- **MATLAB.** The MathWorks' flagship product, MATLAB, is a scientific and numerical programming environment featuring a rich library of linear algebra, signal processing, and visualization functions [?]. MATLAB is licensed as a commercial product.
- **LabVIEW.** National Instruments offers a commercial solution, LabVIEW MathScript, which is a component of LabVIEW, that provides an interactive programming environment compatible with MATLAB [?].
- **Octave.** GNU Octave is a free alternative to MATLAB, providing high-level interpreted language functionality for scientific and numerical applications. GNU Octave is distributed under the GNU General Public License [?].
- **Sage.** Sage, like Octave, is free software that provides much of the functionality of MATLAB, but also targets users of Magma, Maple, and Mathematica. Sage is distributed under the GNU General Public License [?].

We thank you for your interest in `libflame` and the FLAME project.

⁴Some relevant software packages are considered to be public domain, while others are released under a BSD-like license. Some may even be public domain with regard to only certain components. To err on the side of safety, in cases where the license is not clear, we refer to these packages collectively as "freely-available", particularly when the authors choose to use this terminology. Please refer to the homepages of each software package for precise licensing information.

1.3 Acknowledgments

The `libflame` library was made possible thanks to innovative contributions from some of the top researchers in the field of dense linear algebra, including many active members of the FLAME project. I am flattered and grateful that, despite the fact that the library represents the hard work of all of these individuals, my colleagues encouraged me to publish this document without them as coauthors. Their contributions are well-documented in the many journal papers, conference proceedings, and working notes published over the last decade. Citations for many of these publications may be found in [Appendix A](#).

Over the years, the FLAME project and the `libflame` library effort have been generously funded by the National Science Foundation grants CCF-0702714, CCF-0540926, CCF-0342369, ACI-0305163, and ACI-0203685. In addition, Microsoft, NEC Systems (America), Inc., and National Instruments have provided significant support. An equipment donation from Hewlett-Packard has also been invaluable.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

Chapter 2

Setup for GNU/Linux and UNIX

This chapter discusses how to obtain, configure, compile, and install `libflame` in GNU/Linux and UNIX-like environments.

2.1 Before obtaining `libflame`

We encourage new users to read this section before proceeding to download the `libflame` source code.

2.1.1 System software requirements

Before you attempt to build `libflame`, be sure you have the following software tools:

- **GNU/Linux or UNIX.** `libflame` should compile under a wide variety of GNU/Linux distributions¹ and also on any of the mainstream flavors of UNIX, provided that a somewhat sane development environment is present.
- **GNU tools.** We strongly recommend the availability of a GNU development environment. If a full GNU environment is not present, then at the very least we absolutely require that reasonably recent versions of GNU `make` (3.79 or later) and GNU `bash` (2.0 or later) are installed and specified in the user's `PATH` shell environment variable.²
- **A working BLAS library.** Users must link against an implementation of the BLAS in order to use `libflame`. Currently, `libflame` functions make extensive use of BLAS routines such as `dgemm()` and `dsyrk()` to perform subproblems that inherently occur within almost all linear algebra algorithms. `libflame` also provides access to BLAS routines by way of wrappers that map object-based APIs to the traditional Fortran-77 routine interface. Any library that adheres to the BLAS interface should work fine. However, we strongly encourage the use of Kazushige Goto's GotoBLAS [?, ?, ?]. GotoBLAS provides excellent performance on a wide variety of mainstream architectures. Other BLAS libraries, such as ESSL (IBM), MKL (Intel), ACML (AMD), and netlib's BLAS, have also been successfully tested with `libflame`. Of course, performance will vary depending on which library is used.

The following items are not required in order to build `libflame`, but may still be useful to certain users, depending on how the library is configured.

- **A working LAPACK library.** The user is free to use a third-party implementation of LAPACK in conjunction with `libflame`. Why would the user need an LAPACK implementation? Isn't `libflame`

¹`libflame` has been known to compile successfully under cygwin. However, cygwin is not an environment in which we routinely test our software. If this is your preferred environment, we welcome you to give it a try, even if we will not be able to provide support.

²On some UNIX systems, such as AIX and Solaris, GNU `make` may be named `gmake` while the older UNIX/BSD implementation retains the name `make`. In these environments, the user must be sure to invoke `gmake`, as the `libflame` build system utilizes functionality that is present only in GNU `make`.

supposed to replace the functionality in LAPACK? Yes, `libflame` is designed to be a complete alternative to LAPACK. However, many `libflame` operations still invoke unblocked LAPACK routines for the small subproblems that occur within blocked algorithms. `libflame` implementations of these unblocked algorithms exist, but most are not optimized. Therefore, for now, `libflame` requires that certain LAPACK routines be present at link-time. Fortunately, the `libflame` developers do not require you to provide your own LAPACK implementation. By default, `libflame` includes basic netlib implementations of all routines necessary for successful linking. See Section 2.4.1 for more information on disabling this feature.

- **An OpenMP-aware C compiler.** `libflame` supports parallelism for several operations via the SuperMatrix runtime scheduling system. SuperMatrix requires either a C compiler that supports OpenMP (1.0 or later), or a build environment that supports POSIX threads. As of this writing, the GNU C compiler does not support OpenMP. Therefore, the user must either ensure that `libflame` is configured to use a commercial OpenMP-aware compiler, or configure `libflame` so that SuperMatrix uses POSIX threads.³

2.1.2 System hardware support

Over time, `libflame` has been tested on a wide swath of modern architectures, including but not limited to:

- x86 (Pentium, Athlon, Celeron, Duron, older Xeon series)
- x86_64 (Opteron, Athlon64, recent Xeon, Core2 series)
- ia64 (Itanium series)
- PowerPC/POWER series

Support by an architecture is primarily determined by the presence of an appropriate compiler. At configure-time, the `configure` script will attempt to find an appropriate compiler for a given architecture according to a predetermined search order for that architecture. For example, The first C compiler searched for on an Itanium2 system is Intel's `icc`. If `icc` is not found, then the search continues for GNU `gcc`. If neither `icc` nor `gcc` is present, then the script checks for a generic compiler named `cc`. Table 2.1 summarizes the search order of C and Fortran compilers for some of the more common architectures supported by `libflame`. Here, the architecture is identified by the canonical build system type, which is a string of three dash-separated substrings, identifying the CPU type, vendor, and operating system of the system which is performing the build. The build system type is determined by the helper shell script `config.guess` and output by `configure` at configure-time.

It is also possible for the user to specify the C and Fortran compilers explicitly at configure-time. For more information on this and related topics, refer to Section 2.4.1.

2.1.3 License

`libflame` is intellectual property of The University of Texas. Unless you or your organization has made other arrangements, `libflame` is provided as free software under version 2.1 of the GNU Lesser General Public License (LGPL). Please refer to Appendix B for the full text of this license.

2.1.4 Source code

The `libflame` source code is available in two forms:

- **Nightly snapshots.** We encourage users to download and use the latest nightly snapshot. These packages contain full copies of the `libflame` source tree, including all the relevant build system scripts and makefiles as well as items of interest to developers of `libflame`. As their name suggests, these

³Whether there is an advantage in using OpenMP over POSIX threads will depend on the specific OpenMP and POSIX implementations. However, preliminary evidence suggests that configuring SuperMatrix to derive its parallelism from OpenMP results in slightly higher and slightly more consistent performance.

Build system type	Compiler/Tool	Search order
i386-*- i586-*- i686-*-	C	gcc icc cc
	Fortran	gfortran g77 ifort f77 f95
x86_64-*-	C	gcc icc pathcc cc
	Fortran	gfortran g77 ifort pathf95 f77 f95
ia64-*-	C	icc gcc cc
	Fortran	ifort gfortran g77 f77 f95
powerpc*-ibm-aix*	C	xlc
	Fortran	xlf
powerpc64-*-linux-gnu	C	gcc xlc
	Fortran	gfortran g77 xlf
All others	C	gcc cc
	Fortran	gfortran g77 f77 f95

Table 2.1: The list of compilers that are searched for as a function of build system type, which consists three strings, separated by dashes, identifying the build system CPU type, vendor, and operating system, where ‘*’ will match any substring. The actual build system string is determined by the helper shell script `config.guess` and output by `configure` at configure-time. Note that the search for the appropriate system type is performed from top to bottom. Once a matching string is found, the search for each compiler/tool is performed from left to right.

releases represent the state of the `libflame` repository each night (early morning, actually). Expect these nightly snapshots to incorporate the latest interfaces, code improvements, and bug fixes.

- **Milestone releases.** The `libflame` team also makes previous milestone releases available to users. These releases are similar to the nightly snapshots, except that they are also associated with an incremented version number and an update of the `CHANGELOG`.

Though it may seem like the milestone releases would be more stable and the nightly snapshots more prone to bugs, we have actually found the opposite to be true. Milestone releases are fine immediately after they are released, but they quickly grow out-of-date. We check in updates to the library often, sometimes several in one day. However, these updates are not applied to older releases. If an error exists in an older milestone release and the fix has already been applied by `libflame` developers, then the user must obtain a more recent nightly snapshot to obtain the corrected code.⁴ It is for this reason that we *strongly* encourage users to use nightly snapshots over milestones.

2.1.5 Tracking source code revisions

Each copy of `libflame` is named according to its subversion⁵ *revision* number. These revision numbers are positive integers which uniquely identify various states of the `libflame` source tree and are usually preceded by a lowercase “r”. By contrast, milestone *version* numbers, such as “2.0”, are somewhat arbitrary labels that refer to long contiguous revision intervals. As an example, version 1.0 was associated with revisions r1307 through r1753. Revision numbers are incremented automatically every time a developer commits a change or set of changes to the `libflame` source tree. Version numbers, however, identify milestone releases and increase rather infrequently. Usually, milestones are released (and the version number bumped) only when `libflame` developers decide that enough features and bug fixes have been added to be considered newsworthy to its target audience.

⁴The user may also find bug fixes by downloading a more recent milestone release. However, since milestone releases are quite infrequent, roughly one per year, obtaining a more recent milestone release is oftentimes not an option.

⁵We use the subversion version control system to manage changes and synchronize updates among developers.

2.1.6 If you have problems

If you encounter trouble while trying to build and install `libflame`, if you think you've found a bug, or if you have a question not answered in this document, we invite you to email your question to `flame@cs.utexas.edu`. A `libflame` developer will try to get back in touch with you as soon as possible.

2.2 Obtaining libflame

The source code for `libflame` may be obtained through the FLAME project website:

```
http://www.cs.utexas.edu/users/flame/libflame/
```

This webpage also contains important information related to configuring, compiling, installing, and linking against `libflame`. Most of the information provided there is repeated and expanded upon in this chapter.

2.3 Preparation

Download the `.tar.gz` package from the website, and then `un-tar` and `un-gzip` the source code. Here, we assume that we've downloaded revision `r3021` from the nightly snapshots directory.

```
> tar xzf libflame-r3021.tar.gz
> ls
libflame-r3021  libflame-r3021.tar.gz
```

Change into the `libflame-r3021` directory:

```
> cd libflame-r3021
```

The top-level directory of the source tree should look something like this:

```
> ls
AUTHORS      Doxyfile  Makefile   build      docs      src      windows
CHANGELOG    INSTALL  README    configure  revision  test
CONTRIBUTORS LICENSE  bootstrap  configure.ac  run-conf  tmp
```

Table 2.2 describes each file present here. In addition, the figure lists files that are created and overwritten only upon running `configure`.

2.4 Configuration

The first step in building `libflame` is to configure the build system by running the `configure` script. `libflame` may be configured many different ways depending on which options are passed into `configure`. These options and their syntax are always available by running `configure` with the `--help` option:

```
> ./configure --help
```

Be aware that `./configure --help` lists several options that are ignored by `libflame`.⁶ The options that are supported are listed explicitly and described in the next subsection.

⁶This is due to boilerplate content that `autoconf` inserts into the `configure` script regardless of whether it is desired.

File	Type	Description
AUTHORS	peristent	Credits for authorship of various sub-components of libflame .
CHANGELOG	peristent	A list of major changes in each major milestone release version.
CONTRIBUTORS	peristent	Credits for co-authors of working notes, conference papers, and journal articles that have influenced the development of libflame .
Doxyfile	peristent	The configuration file for running doxygen , which we use to automatically generate a map of the source tree.
INSTALL	peristent	Generic instructions for configuring, compiling, and installing the software package, courtesy of the Free Software Foundation.
LICENSE	peristent	The file specifying the license under which the software is made available. As of this writing, libflame is available as free software under version 2.1 of the GNU Lesser General Public License (LGPL).
Makefile	peristent	The top-level makefile for compiling libflame . This makefile uses the GNU include directive to recursively include the makefile fragments that are generated at configure-time. Therefore, it is inoperable until configure has been run.
README	peristent	A short set of release notes directing the user to the libflame web page and the libflame reference manual for more detailed information concerning installation and usage.
bootstrap	peristent	A shell script used by developers to regenerate the configure script.
build	peristent	This directory contains auxiliary build system files and shell scripts. These files are probably only of interest to developers of libflame , and so most users may safely ignore this directory.
config	build	A directory containing intermediate architecture-specific build files.
config.log	build	Logs information as it is gathered and processed by configure .
config.status	build	A helper script invoked by configure .
config.sys_type	build	This file is used to communicate the canonical build system type between configure and config.status helper script. The reasons this file is needed relate to variable scoping quirks and are beyond the scope of this description.
configure	peristent	The script used to configure libflame for compiling and installation. configure accepts many options, which may be queried by running ./configure --help .
configure.ac	peristent	An input file to autoconf that specifies how to build the configure script based on a sequence of m4 macros. This file is only of interest to libflame developers.
docs	peristent	A directory containing documentation related to libflame . The LaTeX source to the libflame reference manual resides here.
lib	build	A directory containing the libraries created after compilation.
obj	build	A directory containing the object files created during compilation.
revision	build/persistent	A file containing the subversion revision number of the source code.
run-conf	peristent	A directory containing a wrapper script to configure that the user may use to help them specify multiple options. The script is strictly a convenience; some users will opt to instead invoke configure directly.
src	peristent	The root directory of all source code that goes into building libflame .
tmp	peristent	A directory containing miscellaneous developer odds-and-ends.
windows	peristent	The directory containing the Windows build system. See Chapter 3 for detailed instructions on how to configure, compile, install, and link against a Windows build of libflame .

Table 2.2: A list of the files and directories the user can expect to find in the top-level **libflame** directory along with descriptions. Files marked “persistent” should always exist while files marked “build” are build products created by the build system. This latter group of files may be safely removed by invoking the **make** target **distclean**.

2.4.1 configure options

The command line options supported by the `configure` script may be broken down into standard options, which most `configure` scripts respond to, and `libflame`-specific options, which refer to functionality unique to `libflame`.

The standard command line options are:

`--prefix=prefixdir`

The *prefixdir* directory specifies the install prefix directory (ie: the root directory at which all `libflame` build products will be installed). If the directory does not exist, it is created. This value defaults to `$HOME/flame`.

`--help, -h`

Display a summary of all valid options to `configure`. (Note that this will display more options than `libflame` actually uses. Only those options described in this section are used internally by the build system.)

`--help=short`

Display a summary of only those options that are specific to `libflame`.

`--version, -V`

Display `libflame` and `autoconf` version information.

`--silent, --quiet, -q`

Silent mode. Do not print “checking...” messages during configuration.

All command line options specific to `libflame` fall into two categories: those which describe a particular *feature* to enable or disable, and those which instruct `configure` to set up the build for use with a particular *tool*.

Command line options which denote features take the form `--disable-FEATURE` or `--enable-FEATURE`, where *FEATURE* is a short string that describes the feature being enabled or disabled. Enabling some options requires that an argument be specified. In these cases, the syntax takes the form of `--enable-FEATURE=ARG`, where *ARG* is an argument specific to the feature being enabled.

Command line options which request the usage of certain tools are similar to feature options, except that tool options always take an argument. These options take the form `--with-TOOL=TOOLNAME`, where *TOOL* and *NAME* are short strings that identify the class of tool and the actual tool name, respectively.

The supported command line feature options are:

`--enable-verbose-make-output`

Enable verbose output as `make` compiles source files and archives them into libraries. By default, `configure` instructs `make` to suppress the actual commands sent to the compilers (and to `ar`) and instead print out more concise progress messages. This option is useful to developers and advanced users who suspect that `make` may not be invoking the compilers correctly. *Disabled by default.*

`--enable-max-arg-list-hack`

Enable a workaround for environments where the amount of memory allocated to storing command line argument lists is too small for `ar` to archive all of the library’s object files with one command. This usually is not an issue, but on some systems the user may get an “Argument list too long” error message. In those situations, the user should enable this option. Note: `make` may not be run in parallel to build `libflame` when this option is enabled! Doing so will result in undefined behavior from `ar`. *Disabled by default.*

`--enable-builtin-lapack-routines`

Build and include into `libflame` blocked and unblocked LAPACK routines for all operations supported within `libflame`. When this option is disabled, LAPACK is required at link-time. Note that FLAME implementations of LAPACK operations (such as Cholesky, LU, and QR factorizations) only invoke LAPACK code for their unblocked subproblems, though `libflame` also includes object-based wrappers to external blocked routines to allow easy performance comparison to reference implementations. Enabling this option is useful when a user is setting up `libflame` for the first time and does not want to build LAPACK from source and has no intention of using a third-party library, such as Intel's MKL, to provide basic LAPACK functionality. *Enabled by default.*

--enable-non-critical-code

Enable code that provides non-critical functionality. This code has been identified as unnecessary when total library size is of concern. *Enabled by default.*

--enable-blas3-front-end-cntl-trees

Enable code that uses control trees⁷ to select a reasonable variant and blocksize when level-3 BLAS front-ends are invoked. When disabled, the front-ends invoke their corresponding external implementations. Note that control trees are always used for LAPACK-level operations. *Enabled by default.*

--enable-multithreading=*model*

Enable multithreading support. Valid values for *model* are `pthread` and `openmp`. Multithreading must be enabled to access the shared memory parallelized implementations provided by SuperMatrix. *Disabled by default.*

--enable-supermatrix

Enable SuperMatrix, a dependency-aware task scheduling and parallel execution system. Note that multithreading support must also be enabled, via **--enable-multithreading**, in order to activate parallelized implementations. If SuperMatrix is enabled but multithreading is not, then SuperMatrix-aware routines will operate sequentially in a verbose “simulation” mode. *Disabled by default.*

--enable-memory-alignment=*N*

Enable code that aligns dynamically allocated memory regions at *N*-byte boundaries. Specifically, this option configures `libflame` to use `posix.memalign()` instead of `malloc()` for all internal memory allocation. Note: *N* must be a power of two and multiple of `sizeof(void*)`, which is usually 4 on 32-bit architectures and 8 on 64-bit architectures. *Disabled by default.*

--enable-ldim-alignment

If memory alignment is requested, enable code that will increase, if necessary, the leading dimension of `libflame` objects so that each matrix column begins at an aligned address. *Disabled by default.*

--enable-optimizations

Employ traditional compiler optimizations when compiling C and Fortran source code. *Enabled by default.*

--enable-warnings

Use the appropriate flag(s) to request warnings when compiling C and Fortran source code. *Enabled by default.*

--enable-debug

⁷ Control trees are internal constructs designed to reduce code redundancy within `libflame`. They allow developers to specify parameters such as blocksize, algorithmic variant, and parallel execution without changing the code that defines the algorithm in question. They are described in detail in Chapter 6.

Use the appropriate debug flag (usually `-g`) when compiling C and Fortran source code. *Disabled by default.*

--enable-profiling

Use the appropriate profiling flag (usually `-pg`) when compiling C source code. *Disabled by default.*

--enable-internal-error-checking=level

Enable various internal runtime checks of function parameters and object properties to prevent functions from executing with unexpected values. Valid values for *level* are `full`, `minimal`, and `none`. *Enabled by default to full.*

--enable-memory-counter

Enable code that keeps track of the balance between calls to `FLA_malloc()` and `FLA_free()`. Upon calling `FLA_Finalize()`, the counter value is output to standard error. *Disabled by default.*

--enable-goto-interfaces

Enable code that interfaces with internal/low-level functionality within GotoBLAS, such as those symbols that may be queried for architecture-dependent blocksize values. When this option is disabled, reasonable static values are used instead. Note that in order to use `libflame` with a BLAS library other than GotoBLAS, the user must disable this option. *Enabled by default.*

--enable-cblas-interface

Enable code that interfaces `libflame`'s external wrapper routines to the BLAS via the CBLAS interface rather than the traditional Fortran-77 interface. *Disabled by default.*

--enable-default-m-blocksize=mb

--enable-default-k-blocksize=kb

--enable-default-n-blocksize=nb

Enable user-defined blocksizes in the *m*, *k*, and *n* dimensions. These options may be used to define the blocksizes that will be returned from blocksize query functions when GotoBLAS interfaces are disabled. Note that these options have no effect when GotoBLAS interfaces are enabled. *Disabled by default.*

--enable-portable-timer

Define the `FLA_Clock()` timer function using `gettimeofday()` even if a more optimized inline assembly implementation is available for the build architecture. *By default, an architecture-specific implementation is used if it exists; otherwise, the more portable gettimeofday()-based code is used.*

A few command line feature options are supported by `configure` but refer to features that are experimental and/or not yet completely implemented. Unless you are know what you are doing, you should avoid using these options:

--enable-windows-build

Enable code that is needed for a Windows-friendly build of `libflame`. This entails disabling all code specific to Linux/UNIX. (Note: this option is actually never used in practice because the Windows build of `libflame` does not use `configure` to begin with.) *Disabled by default.*

--enable-supermatrix-visualization

Enable code that timestamps wall clock information into SuperMatrix task objects. This information may subsequently be recovered to reconstruct and study the runtime execution. *Disabled by default.*

The supported command line tool options are:

`--with-cc=cc`

Search for and use a C compiler named *cc*. If *cc* is not found, then use the first compiler found from the default search list for the detected build architecture.

`--with-f77=f77`

Search for and use a Fortran-77 compiler named *f77*. If *f77* is not found, then use the first compiler found from the default search list for the detected build architecture.

`--with-ar=ar`

Search for and use a library archiver named *ar*. If *ar* is not found, then use the first library archiver found from the default search list for the detected build architecture. Note: the library archiver search list usually consists only of *ar*.

`--with-ranlib=ranlib`

Search for and use a library archive indexer named *ranlib*. If *ranlib* is not found, then use the first library archiver found from the default search list for the detected build architecture. Note: the library archiver search list usually consists only of *ranlib*.

In addition to specifying tools via command line options, the user may alternately make the same requests via environment variables. Environment variables, if they are set, *always* override their corresponding command line options. **configure** also supports a few related environment variables which do not have an analogous command line option.

Table 2.3 lists the supported environment variables and their corresponding tool options, if one exists.

2.4.2 Running configure

The simplest way to run **configure** is to invoke it explicitly on the command line, followed by any of the various options described in the previous subsection.

```
> ./configure --enable-supermatrix --enable-multithreading=pthreads --disable-internal-error-checking
```

Alternatively, the user may invoke **configure** indirectly through a convenient wrapper script, **run-configure.sh**. This script contains an invocation of **configure** along with nearly all of the default **configure** options. To specify non-default options, the user can simply edit the script and then invoke it from the top-level directory, just as he would for **configure**.

```
> ./run-conf/run-configure.sh
```

The benefit of using **run-configure.sh** is twofold. First, the user has a clear and concise way of reviewing the options passed into **configure**. This information is automatically output to **config.log**; however, in order to recover this information the user must sift through many lines of logging output, which tends to be more cumbersome. Second, the user can easily re-configure **libflame** with slightly different options by simply editing **run-configure.sh** and then re-running the script.

The primary purpose of running **configure** is to provide **make** with some of the information it needs in order to begin compiling **libflame**. As **configure** searches for and checks various parts of the build environment, it echoes its progress to standard output. The following is an example of a snippet of such output:

Variable	Command line option	Description
CC	<code>--with-cc=cc</code>	Use <i>CC</i> as the C compiler.
CFLAGS	<i>none</i>	The command line flags to use with the C compiler. Note: Do not set this variable unless you know what you are doing! It overrides C compiler flags that are set internally by libflame feature options.
F77	<code>--with-f77=f77</code>	Use <i>F77</i> as the Fortran-77 compiler.
FFLAGS	<i>none</i>	The command line flags to use with the Fortran-77 compiler. Note: Do not set this variable unless you know what you are doing! It overrides Fortran-77 compiler flags that are set internally by libflame feature options.
AR	<code>--with-ar=ar</code>	Use <i>AR</i> to create and fill static library archives.
RANLIB	<code>--with-ranlib=ranlib</code>	Use <i>RANLIB</i> to generate the index to the static library archive. Note: In modern environments, the functionality of <i>ranlib</i> has been superceded by <i>ar</i> ; GNU <i>ranlib</i> is equivalent to running <i>ar -s</i> .
FIND	<i>none</i>	The <i>find</i> utility is needed by the <i>clean</i> targets defined in the Makefile.
XARGS	<i>none</i>	<i>xargs</i> , like <i>find</i> , is needed by the <i>clean</i> targets defined in the Makefile.

Table 2.3: A list of the environment variables supported by *configure*. Those environment variables which have corresponding command line options are listed with entries in the middle column. Note: Environment variables *always* override their corresponding command line option, if one exists, and provided that it is passed in at configure-time.

```
> ./run-conf/run-configure.sh
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking whether user requested enabling SPU parallelism for the Cell processor... no
checking for GNU make... make
checking for GNU bash... bash
checking whether user requested a specific C compiler... no
configure: CC environment variable is set to icc, which will override --with-cc option
and default search list for C compiler.
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
```

configure has another purpose, though: to create makefile fragments for each directory in the source tree. The user can see this second half of the *configure* process with output that looks something like:

```
gen-make-frag.sh: creating makefile fragment in src/base/flamec
gen-make-frag.sh: creating makefile fragment in src/base/flamec/check
gen-make-frag.sh: creating makefile fragment in src/base/flamec/check/base
gen-make-frag.sh: creating makefile fragment in src/base/flamec/check/base/main
gen-make-frag.sh: creating makefile fragment in src/base/flamec/check/base/util
gen-make-frag.sh: creating makefile fragment in src/base/flamec/check/blas
gen-make-frag.sh: creating makefile fragment in src/base/flamec/check/blas/1
gen-make-frag.sh: creating makefile fragment in src/base/flamec/check/blas/2
gen-make-frag.sh: creating makefile fragment in src/base/flamec/check/blas/3
gen-make-frag.sh: creating makefile fragment in src/base/flamec/check/lapack
gen-make-frag.sh: creating makefile fragment in src/base/flamec/check/lapack/util
```

These makefile fragments are included recursively by the top-level `Makefile` and give `make` access to the source files which reside throughout the source tree.⁸ The makefile fragments are all named `.fragment.mk`, and thus they are hidden from normal directory viewing.

Once `configure` has completed, it invokes a secondary script, `post-configure.sh`, to print out a summary of the configuration process. Please review this summary and confirm that `libflame` has been configured as intended.

There is one section of the configuration summary that you should pay special attention to. At the end of the summary, there will be output that looks like:

```
NOTE: Based on the Fortran compiler recognized above (ifort), configure
thinks that the following linker flags will be necessary to successfully link
a program to the Fortran intrinsic and run-time libraries. You may also need
to pass these flags to the linker in order to link against the libflame
libraries.

LDFLAGS = -L/opt/intel/fortran/compiler/lib -L/usr/lib/gcc/x86_64-linux-gnu/3.4.6/
-L/usr/lib/gcc/x86_64-linux-gnu/3.4.6/../../../../lib64 -lifport -lifcore -limf
-lsvml -lm -lipgo -lirc -lirc_s -ldl
```

The purpose of this note is to inform the user of linker flags⁹ that may be needed in order to successfully link your application against `libflame`. Sometimes, these flags are not necessary, but it is safer to always use them. Please see Section 2.7 for further instructions on using these flags at link-time. In the meantime, there is no need to copy and save these flags to a separate file. You may view the flags detected by the previous run of `configure` at any time by opening the `post-configure.sh` script in your favorite file editor or viewer. The `post-configure.sh` script resides in the subdirectory of `config` that is identified by the build system string detected by `configure`.

```
> ls -l config/x86_64-unknown-linux-gnu/post-configure.sh
-rwxr--r-- 1 field flame 6800 Aug 29 16:23 config/x86_64-unknown-linux-gnu/post-configure.sh
```

2.5 Compiling

After `configure` has run, the user may proceed to building the library. The simplest way to do this is to just run `make`:

```
> make
```

This is actually shorthand for `make all`. That is, it tells `make` to invoke the `all` target, which in turn invokes the `libs` target. Invoking the `libs` target compiles and archives the library. Table 2.4 lists the most useful `make` targets defined in the `libflame` `Makefile`.

As `make` performs individual compiles individual source files into object files, it will output progress information. By default, this appears as:

```
Compiling src/base/flamec/main/FLA_Blocksize.c
Compiling src/base/flamec/main/FLA_Check.c
Compiling src/base/flamec/main/FLA_Error.c
Compiling src/base/flamec/main/FLA_File.c
```

⁸ The idea behind generating recursively-includable makefile fragments at configure-time is that these fragments will often change when files and directories are added, moved, or deleted by `libflame` developers, and thus it is much more convenient for them to be generated automatically than to be stored and maintained within the source code repository.

⁹ The flags shown were detected when `libflame` was configured to use Intel compilers in an `x86_64-unknown-linux-gnu` build environment that happens to provide both Intel and GNU compilers. Oftentimes, `post-configure.sh` will display link flags that appear to accomodate linking with two different compiler packages. In our experience, we've found that these extraneous flags do not interfere with the compiler at link-time.

```
Compiling src/base/flamec/main/FLA_Init.c
Compiling src/base/flamec/main/FLA_Lock.c
Compiling src/base/flamec/main/FLA_Memory.c
Compiling src/base/flamec/main/FLA_Misc.c
Compiling src/base/flamec/main/FLA_Obj.c
```

If `libflame` was configured with `--enable-verbose-make-output`, then the output will show the actual compiler commands being executed.

2.5.1 Parallel make

`libflame` has been known to take a while to build, especially on systems with slow processors and/or slow compilers. If you are performing the build on an SMP or multicore system, then you may parallelize the compilation by using the `-j n` option to `make`. This option tells `make` to perform up to n tasks in parallel. In the following example, we request that `make` avail itself to four-way parallelism.

```
> make -j4
```

The n argument should be set to a reasonable value, such as the number of cores or processors on the system. Be aware that this may not necessarily speed up the build process if the build system has an I/O bottleneck, such as a slow network-mounted filesystem.

2.6 Installation

After `make` has successfully completed, the `libflame` library archives reside in a subdirectory of the `lib` directory. The exact subdirectory name depends on the build system type.

```
> ls -l lib/x86_64-unknown-linux-gnu/
total 17680
-rw-r--r-- 1 field flame 17226100 Aug 29 17:00 libflame.a
-rw-r--r-- 1 field flame 846052 Aug 29 17:00 liblapack2flame.a
```

In this example, `libflame` was built for an x86_64 system with a build system type of `x86_64-unknown-linux-gnu`, and so the library archives reside in the directory `lib/x86_64-unknown-linux-gnu`.

At this point, you need to use the `install` target to move the libraries and header files to a more permanent location.

```
> make install
Installing libflame-x86_64-r3021.a into /home/field/flame/lib/
Installing liblapack2flame-x86_64-r3021.a into /home/field/flame/lib/
Installing C header files into /home/field/flame/include-x86_64-r3021
```

Here, we can see the library and header files were moved into the default subdirectories of the user's home directory. Notice that the libraries and include directory are renamed to reflect the build architecture and the revision number.

```
> ls -l $HOME/flame/
total 8
drwxr-xr-x 2 field flame 4096 Aug 29 17:07 include-x86_64-r3021
drwxr-xr-x 5 field flame 4096 Aug 29 17:39 lib
> ls -l $HOME/flame/lib/
total 17680
-rw-r--r-- 1 field flame 17226100 Aug 29 17:39 libflame-x86_64-r3021.a
-rw-r--r-- 1 field flame 846052 Aug 29 17:39 liblapack2flame-x86_64-r3021.a
```

Target	Function
<code>all</code>	Invoke the <code>libs</code> target.
<code>check</code>	Verify that <code>configure</code> has been run.
<code>libs</code>	Invoke the <code>check</code> target and then build the <code>libflame</code> and <code>liblapack2flame</code> library archives.
<code>install</code>	Invoke the <code>libs</code> target and then copy the library archives and header files to their respective <code>lib</code> and <code>include</code> subdirectories of the install prefix directory, which is <code>\$HOME/flame</code> by default.
<code>install-symlinks</code>	Create symbolic links (with <code>ln -s</code>) to the library files and <code>include</code> subdirectory so that the symbolic links do not contain the architecture or version strings. This is target convenient when you are building <code>libflame</code> for only one architecture (let's assume the <code>x86_64</code>) and would rather refer to <code>libflame.a</code> than <code>libflame_x86_64-v2.0.a</code> in your application's <code>Makefile</code> .
<code>install-symlinks-with-arch</code>	Similar to <code>install-symlinks</code> , except that the resulting symbolic links will only omit the version string (leaving the architecture string). Continuing the example above, this allows you to refer to the library as <code>libflame_x86_64.a</code> in your application's <code>Makefile</code> . This target should be used when you are building <code>libflame</code> for multiple architectures and would like to install all copies to the same install prefix directory.
<code>clean</code>	Remove all object files and previously-built library archives from the <code>obj</code> and <code>lib</code> directories for the current build system only. (Recall that the current build system is written to <code>config.sys.type</code> .) Build products built for other systems will not be touched.
<code>cleanmk</code>	Remove all makefile fragments from the source tree.
<code>distclean</code>	Invoke <code>clean</code> , <code>cleanmk</code> , and then remove all other intermediate build files and directories that are created either by <code>configure</code> or one of the autotools such as <code>autoconf</code> .
<code>send-thanks</code>	Use <code>mail</code> to send the <code>libflame</code> developers a short thank-you message.

Table 2.4: A list of “phony” `make` targets defined in the `libflame Makefile`. Note that not all targets guarantee that action will take place. Most targets will not fire if `make` determines that the target is already up-to-date. For example, invoking the `clean` target will not remove any object files if they do not exist.

Now that the libraries have been installed, you are almost ready to use them. However, we offer you one last convenience. Presumably, you will edit your application's `Makefile` to refer to one or both of the `libflame` archive files. If you only plan on building `libflame` for one architecture, then you may use the `install-symlinks` target to install symbolic links to the library archives and the include directory.

```
> make install-symlinks
Installing symlink libflame.a into /home/field/flame/lib/
Installing symlink liblapack2flame.a into /home/field/flame/lib/
Installing symlink include into /home/field/flame/
```

After executing `install-symlinks`, the install directories should contain:

```
> ls -l $HOME/flame/
total 8
lrwxrwxrwx 1 field flame 19 Aug 29 17:08 include -> include-x86_64-r3021
drwxr-xr-x 2 field flame 4096 Aug 29 17:07 include-x86_64-r3021
drwxr-xr-x 2 field flame 4096 Aug 29 17:39 lib
> ls -l $HOME/flame/lib/
total 17680
lrwxrwxrwx 1 field flame 22 Aug 29 17:08 libflame.a -> libflame-x86_64-r3021.a
-rw-r--r-- 1 field flame 17226100 Aug 29 17:07 libflame-x86_64-r3021.a
lrwxrwxrwx 1 field flame 29 Aug 29 17:08 liblapack2flame.a -> liblapack2flame-x86_64-r3021.a
-rw-r--r-- 1 field flame 846052 Aug 29 17:07 liblapack2flame-x86_64-r3021.a
```

However, if you plan on building and installing multiple instances of `libflame` and you wish to install them to the same directory, we offer the `install-symlinks-with-arch` target. Executing this target creates symbolic links similar to those of `install-symlinks`, except that only the revision string is omitted. Keeping the architecture string provides enough differentiation to allow multiple installations of `libflame` to coexist in the same install directory.

```
> make install-symlinks-with-arch
Installing symlink libflame-x86_64.a into /home/field/flame/lib/
Installing symlink liblapack2flame-x86_64.a into /home/field/flame/lib/
Installing symlink include-x86_64 into /home/field/flame/
```

After executing the `install` and `install-symlinks-with-arch` targets, we are left with an install directory that looks like:

```
> ls -l $HOME/flame
total 8
lrwxrwxrwx 1 field flame 19 Aug 29 17:08 include-x86_64 -> include-x86_64-r3021
drwxr-xr-x 2 field flame 4096 Aug 29 17:07 include-x86_64-r3021
drwxr-xr-x 2 field flame 4096 Aug 29 17:39 lib
> ls -l $HOME/flame/lib
total 17680
-rw-r--r-- 1 field flame 17226100 Aug 29 17:07 libflame-x86_64-r3021.a
lrwxrwxrwx 1 field flame 22 Aug 29 17:08 libflame-x86_64.a -> libflame-x86_64-r3021.a
-rw-r--r-- 1 field flame 846052 Aug 29 17:07 liblapack2flame-x86_64-r3021.a
lrwxrwxrwx 1 field flame 29 Aug 29 17:08 liblapack2flame-x86_64.a -> liblapack2flame-x86_64-r3021.a
```

Of course, you could execute both the `install-symlinks` and `install-symlinks-with-arch` targets and get both kinds of symbolic links.

2.7 Linking against libflame

Since you are building `libflame`, you probably wish to use it in your application. This section will show you how to link `libflame` with your existing application.

Let's assume that you've installed `libflame` to the default location in `$HOME/flame`. Let's also assume that you invoked the `install-symlinks` target, giving you shorthand symbolic links to both `libflame` and the directory containing header files.

In general, you should make the following changes to your application build process:

- **Add the `libflame` header directory to the include path of your compiler.** Usually, this is done by with the `-I` compiler option. For example, if you configured `libflame` to use `$HOME/flame` as the install prefix, then you would add `-I$HOME/flame/include` to the command line when invoking the compiler. Strictly speaking, this is only necessary when compiling source code files that use `libflame` symbols or APIs, but it is generally safe to use when compiling all of your application's source code.
- **Add `libflame` to the link command that links your application.** If you only wish to use the native `libflame` API, then you only need to add `libflame.a` to your link command. However, note that `libflame.a` *must* appear in front of the LAPACK and BLAS libraries. This is because the linker only searches for symbols in the “current” archive and those that appear further down in the link command. Placing `libflame` after LAPACK or the BLAS will result in undefined symbol errors at link-time.
- **Use the recommended linker flags detected by `configure`.** This topic was previously alluded to toward the end of Section 2.4.2. It is often the case that you must add various linker flags to the link command in order to properly link your application with `libflame`. This is usually the result of the compilers embedding certain low-level functions into the object code. These functions may only be resolved at link-time if the library in which they are defined is also provided to the linker. The list of linker flags that you will need is displayed when `configure` finished and exits. After `configure` is run, you may also find these linker flags in the `post-configure.sh` script, as described near the end of Section 2.4.2.

Now let's give a concrete example of these changes. Suppose you've been building your application with a `Makefile` that looks something like:

```
SRC_PATH    := .
OBJ_PATH    := .
INC_PATH    := .

LIB_HOME    := $(HOME)
BLAS_LIB    := $(LIB_HOME)/lib/libblas.a
LAPACK_LIB  := $(LIB_HOME)/lib/liblapack.a

CC          := icc
LINKER      := $(CC)
CFLAGS      := -g -O2 -Wall -I$(INC_PATH)
LDFLAGS     := -lm

MYAPP_OBJS  := main.o file.o util.o proc.o
MYAPP_BIN   := my_app

$(OBJ_PATH)/%.o: $(SRC_PATH)/%.c
    $(CC) $(CFLAGS) -c $< -o $@

$(MYAPP): $(MYAPP_OBJS)
    $(LINKER) $(MYAPP_OBJS) $(LDFLAGS) $(LAPACK_LIB) $(BLAS_LIB) -o $(MYAPP_BIN)

clean:
    rm -f $(MYAPP_OBJS) $(MYAPP_BIN)
```

To link against `libflame`, you should change your `Makefile` as follows:

```
SRC_PATH    := .
OBJ_PATH    := .
INC_PATH    := .
```

```

LIB_HOME    := $(HOME)
BLAS_LIB    := $(LIB_HOME)/lib/libblas.a
LAPACK_LIB   := $(LIB_HOME)/lib/liblapack.a

FLAME_HOME  := $(HOME)
FLAME_INC   := $(FLAME_HOME)/flame/include
FLAME_LIB   := $(FLAME_HOME)/flame/lib/libflame.a

CC          := icc
LINKER      := $(CC)
CFLAGS      := -g -O2 -Wall -I$(INC_PATH) -I$(FLAME_INC)
LDLFLAGS    := -L/opt/intel/fc/em64t/10.0.026/lib
LDLFLAGS    += -L/usr/lib/gcc/x86_64-pc-linux-gnu/3.4.6/
LDLFLAGS    += -L/usr/lib/gcc/x86_64-pc-linux-gnu/3.4.6/../../../../lib64
LDLFLAGS    += -lifport -lifcore -limf -lsvml -lm -lipgo -lirc -lirc_s -ldl

MYAPP_OBJS  := main.o file.o util.o proc.o
MYAPP_BIN   := my_app

$(OBJ_PATH)/%.o: $(SRC_PATH)/%.c
    $(CC) $(CFLAGS) -c $< -o $@

$(MYAPP): $(MYAPP_OBJS)
    $(LINKER) $(MYAPP_OBJS) $(LDLFLAGS) $(FLAME_LIB) $(LAPACK_LIB) $(BLAS_LIB) -o $(MYAPP_BIN)

clean:
    rm -f $(MYAPP_OBJS) $(MYAPP_BIN)

```

The changes appear in red.

First, we define the locations of `libflame` and the `libflame` header directory.

Second, we include the location of the `libflame` headers to the compilers' command line options so that the C compiler will be able to perform type checking against `libflame` declarations and prototypes.

Third, we add the linker flags to the `LDLFLAGS` variable so that the linker can find any auxiliary system libraries that might be needed in order to link your application with the object code present in `libflame`.

Finally, we add the `libflame` library to the link command, making sure to insert it before the LAPACK and BLAS libraries.

2.7.1 Linking with liblapack2flame

The previous section demonstrated how to modify a hypothetical makefile to link a pre-existing application to `libflame`. However, some users have applications which use LAPACK interfaces and wish to use `libflame` without changing their application code. This may be accomplished through a companion library, `liblapack2flame`. For more information about `liblapack2flame`, refer to Section 5.9. In this section, we will show how to extend those makefile modifications to use `liblapack2flame`.

To link against `liblapack2flame`, you should change your `Makefile` as follows:

```

BLAS_LIB    := $(HOME)/lib/libblas.a
LAPACK_LIB   := $(HOME)/lib/liblapack.a
FLAME_LIB   := $(HOME)/flame/lib/libflame.a
L2F_LIB     := $(HOME)/flame/lib/liblapack2flame.a
FLAME_INC   := $(HOME)/flame/include

CC          := icc
FC          := ifort
LINKER      := $(CC)
CFLAGS      := -g -O3 -Wall -I$(FLAME_INC)
FFLAGS      := $(CFLAGS)

LDLFLAGS    := -L/opt/intel/fc/em64t/10.0.026/lib
LDLFLAGS    += -L/usr/lib/gcc/x86_64-pc-linux-gnu/3.4.6/
LDLFLAGS    += -L/usr/lib/gcc/x86_64-pc-linux-gnu/3.4.6/../../../../lib64

```



```

LDFLAGS      += -lifport -lifcore -limf -lsvml -lm -lipgo -lirc -lirc.s -ldl

MYAPP_OBJS := main.o file.o util.o proc.o
MYAPP_BIN  := my_app

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@
%.o: %.f
    $(FC) $(FFLAGS) -c $< -o $@

$(MYAPP): $(MYAPP_OBJS)
    $(LINKER) $(MYAPP_OBJS) $(LDFLAGS) $(L2F_LIB) $(FLAME_LIB) $(LAPACK_LIB) $(BLAS_LIB) -o $(MYAPP_BIN)

clean:
    rm -f $(MYAPP_OBJS) $(MYAPP_BIN)

```

As before, the changes appear in red.

The changes here are identical to those shown in Section 2.7, except for two lines. First, a new makefile variable, `L2F_LIB`, is defined to be the path to the actual `liblapack2flame` library archive. Second, this variable is used in the link command. It is important that `liblapack2flame` occur in the link command *before* `LAPACK` and `libflame`, since the interface routines within `liblapack2flame` depend upon routines and functions that occur in each library.

2.7.2 A word on LAPACK routines

By default, `libflame` is configured to include a core subset of the netlib implementations of LAPACK. Placing `libflame` ahead of LAPACK in the link command means that these routines will be linked into your executable. As a result, routines with identical interfaces that reside in the LAPACK library will be ignored. However, any other routines that are not part of the core set needed by `libflame` will be linked in from LAPACK.

On the other hand, if `libflame` was configured with the `--disable-builtin-lapack-routines` option, then *all* LAPACK routines, both those needed by `libflame` and those which might be called directly from your application, will be obtained from the LAPACK library provided at link-time.

Chapter 3

Setup for Microsoft Windows

This chapter discusses how to obtain, configure, compile, and install `libflame` under Microsoft Windows.

3.1 Before obtaining `libflame`

We encourage new users to read this section before proceeding to download the `libflame` source code.

3.1.1 System software requirements

Before you attempt to build `libflame`, be sure you have the following software tools:

- **Microsoft Windows XP or later.** At this time we have only tested `libflame` under Windows XP. We have not yet been able to test the software under Windows Vista, though we suspect it may compile and run just fine.
- **A C/C++ compiler.** Most of `libflame` is written in C, and therefore building `libflame` on Windows requires a C (or C++) compiler. The build system may be configured to use either the Intel C/C++ compiler or the Microsoft C/C++ compiler. However, another compiler can be substituted by tweaking the definitions file included into the main makefile.
- **A Fortran compiler.** A Fortran compiler is needed in order to compile certain supporting routines that serve as placeholders until `libflame` is further developed. For now, the user must also have a Fortran compiler installed in the build environment. The Intel Fortran compiler is referenced by default, though a different compiler may be used provided that the appropriate makefile definitions are adjusted properly.
- **nmake.** `libflame` for Windows requires the Microsoft Program Maintenance Utility `nmake`. `nmake` is a command line tool similar to GNU `make` that allows developers to use makefiles to specify how programs and libraries should be built. This utility is included with the Microsoft Visual Studio development suite.
- **Python.** Certain helper scripts within the Windows build system are written in Python, and therefore the user must have Python installed in the build environment in order to run the build `libflame`. We recommend a recent version, though version 2.6 or later should work fine.
- **A working BLAS library.** Users must link against an implementation of the BLAS in order to use `libflame`. Currently, `libflame` functions make extensive use of BLAS routines such as `dgemm()` and `dsyrk()` to perform subproblems that inherently occur within almost all linear algebra algorithms. `libflame` also provides access to BLAS routines by way of wrappers that map object-based APIs to the traditional Fortran-77 routine interface. Any library that adheres to the BLAS interface should work fine. However, we strongly encourage the use of Kazushige Goto's GotoBLAS [?, ?, ?]. GotoBLAS provides excellent performance on a wide variety of mainstream architectures. Other BLAS libraries,

such as ESSL (IBM), MKL (Intel), ACML (AMD), and netlib's BLAS, have also been successfully tested with `libflame`. Of course, performance will vary depending on which library is used.

The following items are not required in order to build `libflame`, but may still be useful to certain users, depending on how the library is configured.

- **A working LAPACK library.** The user is free to use a third-party implementation of LAPACK in conjunction with `libflame`. Why would the user need an LAPACK implementation? Isn't `libflame` supposed to replace the functionality in LAPACK? Yes, `libflame` is designed to be a complete alternative to LAPACK. However, many `libflame` operations still invoke unblocked LAPACK routines for the small subproblems that occur within blocked algorithms. `libflame` implementations of these unblocked algorithms exist, but most are not optimized. Therefore, for now, `libflame` requires that certain LAPACK routines be present at link-time. Fortunately, the `libflame` developers do not require you to provide your own LAPACK implementation. By default, `libflame` includes basic netlib implementations of all routines necessary for successful linking. See Section 3.4 for more information on disabling this feature.
- **An OpenMP-aware C compiler.** `libflame` supports parallelism for several operations via the SuperMatrix runtime scheduling system. SuperMatrix requires either a C compiler that supports OpenMP (1.0 or later), or a build environment that supports POSIX threads. POSIX threads support is not shipped with Microsoft Windows. However, as of this writing, both the Microsoft and Intel C/C++ compilers support OpenMP. Therefore, the user must either ensure that `libflame` is configured to use an OpenMP-aware compiler.

3.1.2 System hardware support

Since `libflame` for Windows is still very new, we have not had the time or opportunity to test it on many hardware architectures. We suspect it should compile and run fine on any of the modern Intel architectures, including traditional 32-bit x86 architectures as well as newer 64-bit em64t systems. Other architectures, such as ia64 systems, may work, but they are untested as of this writing.

3.1.3 License

`libflame` is intellectual property of The University of Texas. Unless you or your organization has made other arrangements, `libflame` is provided as free software under version 2.1 of the GNU Lesser General Public License (LGPL). Please refer to Appendix B for the full text of this license.

3.1.4 Source code

The `libflame` source code is available in two forms:

- **Nightly snapshots.** We encourage users to download and use the latest nightly snapshot. These packages contain full copies of the `libflame` source tree, including all the relevant build system scripts and makefiles as well as items of interest to developers of `libflame`. As their name suggests, these releases represent the state of the `libflame` repository each night (early morning, actually). Expect these nightly snapshots to incorporate the latest interfaces, code improvements, and bug fixes.
- **Milestone releases.** The `libflame` team also makes previous milestone releases available to users. These releases are similar to the nightly snapshots, except that they are also associated with an incremented version number and an update of the `CHANGELOG`.

Though it may seem like the milestone releases would be more stable and the nightly snapshots more prone to bugs, we have actually found the opposite to be true. Milestone releases are fine immediately after they are released, but they quickly grow out-of-date. We check in updates to the library often, sometimes several in one day. However, these updates are not applied to older releases. If an error exists in an older milestone release and the fix has already been applied by `libflame` developers, then the user must obtain a more

recent nightly snapshot to obtain the corrected code.¹ It is for this reason that we *strongly* encourage users to use nightly snapshots over milestones.

3.1.5 Tracking source code revisions

Each copy of **libflame** is named according to its subversion² *revision* number. These revision numbers are positive integers which uniquely identify various states of the **libflame** source tree and are usually preceded by a lowercase “r”. By contrast, milestone *version* numbers, such as “2.0”, are somewhat arbitrary labels that refer to long contiguous revision intervals. As an example, version 1.0 was associated with revisions r1307 through r1753. Revision numbers are incremented automatically every time a developer commits a change or set of changes to the **libflame** source tree. Version numbers, however, identify milestone releases and increase rather infrequently. Usually, milestones are released (and the version number bumped) only when **libflame** developers decide that enough features and bug fixes have been added to be considered newsworthy to its target audience.

3.1.6 If you have problems

If you encounter trouble while trying to build and install **libflame**, if you think you’ve found a bug, or if you have a question not answered in this document, we invite you to email your question to flame@cs.utexas.edu. A **libflame** developer will try to get back in touch with you as soon as possible.

3.2 Obtaining libflame

The source code for **libflame** may be obtained through the FLAME project website:

<http://www.cs.utexas.edu/users/flame/libflame/>

This webpage also contains information related to configuring, compiling, installing, and linking against **libflame** under GNU/Linux and UNIX environments. Most of the information provided there is repeated and expanded upon in Chapter 2.

3.3 Preparation

Download the .zip package from the website, and then unzip the the source code. Here, we assume that we’ve downloaded revision r3021 from the nightly snapshots directory and unzipped the package to a directory by the same name, minus the extension.

```
C:\field\temp> dir
Volume in drive C is OS
Volume Serial Number is 941B-95F5

Directory of C:\field\temp

01/07/2009  02:19 PM    <DIR>          .
01/07/2009  02:19 PM    <DIR>          ..
01/07/2009  02:18 PM    <DIR>          libflame-r3021
01/07/2009  02:04 PM             7,838,643 libflame-r3021.zip
               1 File(s)             7,838,643 bytes
               3 Dir(s)  89,927,610,368 bytes free
```

Change into the **libflame-r3021** directory:

¹The user may also find bug fixes by downloading a more recent milestone release. However, since milestone releases are quite infrequent, roughly one per year, obtaining a more recent milestone release is oftentimes not an option.

²We use the subversion version control system to manage changes and synchronize updates among developers.

```
C:\field\temp> cd libflame-r3021
```

The top-level directory of the source tree should look something like this:

```
C:\field\temp\libflame-r3021> dir
Volume in drive C is OS
Volume Serial Number is 941B-95F5

Directory of C:\field\temp\libflame-r3021

01/07/2009  02:18 PM    <DIR>          .
01/07/2009  02:18 PM    <DIR>          ..
01/07/2009  02:18 PM                138 .svnignore
01/07/2009  02:18 PM                934 AUTHORS
01/07/2009  02:18 PM                91 bootstrap
01/07/2009  02:18 PM    <DIR>          build
01/07/2009  02:18 PM                3,875 CHANGELOG
01/07/2009  02:18 PM            296,219 configure
01/07/2009  02:18 PM            14,479 configure.ac
01/07/2009  02:18 PM            2,329 CONTRIBUTORS
01/07/2009  02:18 PM    <DIR>          docs
01/07/2009  02:18 PM            50,456 Doxyfile
01/07/2009  02:18 PM            9,478 INSTALL
01/07/2009  02:18 PM           26,420 LICENSE
01/07/2009  02:18 PM           18,303 Makefile
01/07/2009  02:18 PM            1,216 README
01/07/2009  02:18 PM                5 revision
01/07/2009  02:18 PM    <DIR>          run-conf
01/07/2009  02:18 PM    <DIR>          src
01/07/2009  02:17 PM    <DIR>          test
01/07/2009  02:18 PM    <DIR>          tmp
01/07/2009  02:18 PM    <DIR>          windows
                13 File(s)          423,943 bytes
                9 Dir(s)  89,931,284,480 bytes free
```

This is the top-level directory for the default GNU/Linux and UNIX builds.³ However, since we are building `libflame` for Windows, we should focus on the `windows` subdirectory.

```
C:\field\temp\libflame-r3021> cd windows

C:\field\temp\libflame-r3021\windows> dir
Volume in drive C is OS
Volume Serial Number is 941B-95F5

Directory of C:\field\temp\libflame-r3021\windows

01/07/2009  02:18 PM    <DIR>          .
01/07/2009  02:18 PM    <DIR>          ..
01/07/2009  02:18 PM    <DIR>          build
01/07/2009  02:18 PM                2,493 configure.cmd
01/07/2009  02:18 PM            5,630 gendll.cmd
01/07/2009  02:18 PM            211 libpaths.txt
01/07/2009  02:18 PM            8,766 Makefile
01/07/2009  02:18 PM                5 revision
                5 File(s)          17,105 bytes
                3 Dir(s)  89,931,259,904 bytes free
```

Table 3.1 describes each file present here. In addition, the figure lists files that are created and overwritten only upon running `configure.cmd`.

³Table 2.2 describes the files present in the top-level GNU/Linux and UNIX build directory.

File	Type	Description
Makefile	peristent	The top-level makefile for compiling libflame under Microsoft Windows. This makefile is written for Microsoft's Program Maintenance Utility, nmake . It may only be run after configure.cmd is run.
build	peristent	This directory contains auxiliary build system files and scripts. These files are probably only of interest to developers of libflame , and so most users may safely ignore this directory.
config	build	A directory containing intermediate build files whose contents depend on how libflame was configured.
configure.cmd	peristent	The script used to prepare the Windows build environment for compiling libflame . configure.cmd has multiple required arguments, which are explained when configure.cmd is run with no arguments (or the wrong number of arguments).
dll	build	A directory containing the dynamic libraries created after compilation.
gendll.cmd	peristent	The script used to generate a dynamically-linked library and associated files from a list of object files. It is meant to be invoked by nmake and so normal users should never need to invoke it manually.
include	build	A temporary directory containing copies of the source header files gathered from the top-level source directory tree.
lib	build	A directory containing the static libraries created after compilation.
nmake-cc.log	build	A file capturing the standard output of the C compiler.
nmake-fc.log	build	A file capturing the standard output of the Fortran compiler.
nmake-copy.log	build	A file capturing the standard output of the copy command line utility.
libpaths.txt	peristent	A list of library paths that may be needed by gendll.cmd when building a dynamically-linked library. Since the file is actually a list of arguments to the compiler, the contents must be formatted properly. The format is detailed in Section 3.4.2.
obj	build	A directory containing the object files created during compilation.
revision	build/persistent	A file containing the subversion revision number of the source code.
src	build	A temporary directory containing copies of the source code files gathered from the top-level source directory tree.

Table 3.1: A list of the files and directories the user can expect to find in the **windows** build directory along with descriptions. Files marked “peristent” should always exist while files marked “build” are build products created by the build system. This latter group of files may be safely removed by invoking the **nmake** target **distclean**.

3.4 Configuration

The first step in building **libflame** for Windows is to set the configuration options.

The bulk of the configuration options are specified in the file **build\FLA_config.h**.⁴ The options correspond to C preprocessor macros. If a macro is commented out, the feature is disabled, otherwise it is enabled. Each macro is also preceded with a comment containing a brief description of its corresponding feature. Full documentation for each feature macro in **build\FLA_config.h** may be found in Section 2.4.1.

Those users who plan on building a dynamically-linked library must specify a path to a BLAS library. This is necessary so the linker can resolve all BLAS symbol references within **libflame** at the time the library is built. To specify your BLAS library, edit the **BLAS_LIBPATH** variable in the **build\defs.mk** file:

```
BLAS_LIBPATH = c:\field\lib\blas\blas_WIN32.lib
```

This variable is not referenced if the user builds only a static library.

There are two additional configuration options that must be set in the **build\defs.mk**:

⁴Unlike in the GNU/Linux build system, the user must set these options manually. We apologize for the inconvenience.

Argument	Accepted Values	Consequence
<i>architecture string</i>	<i>any string</i>	This string is inserted into the filename of the final library. It has no effect on how libflame is built.
<i>build type</i>	debug	Enables debugging symbols and disables all compiler optimizations.
	release	Disables debugging symbols and enables maximum compiler optimizations.
<i>C compiler string</i>	icl	Compile C source code with the Intel C/C++ compiler, icl .
	cl	Compile C source code with the Microsoft C/C++ compiler, cl .

Table 3.2: The arguments expected by `configure.cmd`.

- **Verboseness.** **libflame** for Windows may be compiled in verbose mode, in which actual commands are echoed to the command line instead of the more concise output that the user sees by default. In order to compile in verbose mode, the variable **VERBOSE** must be defined. Thus, you may enable verbose mode by uncommenting the following line:

```
# VERBOSE = 1
```

This feature is disabled by default.

- **Built-in LAPACK routines.** The Windows port of **libflame**, just like the GNU/Linux and UNIX version, contains code that is implemented in terms of certain LAPACK routines. If the user does not need to use his own LAPACK implementation, **libflame** may be configured to build and include all requisite LAPACK routines necessary for successful linking (for both static and dynamic libraries). The option is specified by defining the variable **ENABLE_BUILTIN_LAPACK**:

```
ENABLE_BUILTIN_LAPACK = 1
```

By default, the feature is enabled. Commenting out the above line will cause the build system to *not* include any LAPACK routines in the final library. If the user intends to build and use **libflame** as a static library, some implementation of LAPACK must be provided when the application is linked. However, if the user builds **libflame** as a dynamically-linked library, he must specify a LAPACK library when **libflame** is built so that the linker may resolve all symbol references. To specify a LAPACK library, the user should modify the **LAPACK_LIBPATH** variable:

```
LAPACK_LIBPATH = c:\field\lib\lapack\lapack_WIN32.lib
```

Note that this variable is only referenced when building **libflame** as a dynamically-linked library, and only if built-in LAPACK routines are disabled.

3.4.1 IronPython

Users of IronPython will need to manually change `configure.cmd` in order for the script to run correctly. If you are relying on IronPython as your Python implementation, edit the `configure.cmd` file and change the following lines:

```
set GEN_CHECK_REV_FILE=.\build\gen-check-rev-file.py
set GATHER_SRC=.\build\gather-src-for-windows.py
set GEN_CONFIG_FILE=.\build\gen-config-file.py
```


to:

```
set GEN_CHECK_REV_FILE=ipy .\build\gen-check-rev-file.py
set GATHER_SRC=ipy .\build\gather-src-for-windows.py
set GEN_CONFIG_FILE=ipy .\build\gen-config-file.py
```

Also, be sure that the `PATH` environment variable is set to contain the path to your IronPython installation.

3.4.2 Running `configure.cmd`

Once all configuration options are set, the user may run the `configure.cmd` script. The `configure.cmd` script takes three mandatory arguments, which are described in Table 3.2. Usage information can also be found by running `configure.cmd` with no arguments.

The output from running `configure.cmd` should look something like:

```
C:\field\temp\libflame-r3021\windows> configure.cmd x86 debug icl
configure.cmd: Checking/updating revision file.
gen-check-rev-file.py: Found export. Checking for revision file...
gen-check-rev-file.py: Revision file found containing revision string "3021". Export is valid snapshot!
configure.cmd: Gathering source files into local flat directories.
configure.cmd: Creating configure definitions file.
configure.cmd: Configuration and setup complete! You may now run nmake.
```

The `configure.cmd` script first checks whether the revision file needs updating.⁵ Then, a helper script gathers the source code from the primary source tree and places copies within a “flat” directory structure inside of a new `src` subdirectory. Header files are copied into a new `include` subdirectory. Finally, a `config.mk` makefile fragment is generated with various important definitions which will be included by the main `nmake` makefile.

After `configure.cmd` has run, the user must decide what kind of library he wishes to generate. If he wants only a static library, he may proceed by running `nmake lib`. However, if he wants a dynamically-linked library, he must edit the contents of the `libpaths.txt` file. This file should be modified to include the full paths to any directories that may be needed when the linker is creating the dynamically-linked library. The file format is simple; each line is a line passed to the compiler when it is invoked as a linker. Simply modify the existing lines, and/or add additional lines if you have more library directories to pass to the linker. The following is an example of the contents of `libpaths.txt`.

```
/link /LIBPATH:"C:\Program Files\Microsoft SDKs\Windows\v6.0A\Lib"
/link /LIBPATH:"C:\Program Files\Microsoft Visual Studio 9.0\VC\lib"
/link /LIBPATH:"C:\Program Files\Intel\Compiler\11.0\066\fortran\lib\ia32"
```

These paths are typically necessary so that system libraries may be located by the compiler at link-time. If, when attempting to build a dynamically-linked copy of `libflame`, you encounter errors that indicate the compiler could not find certain libraries, try locating the libraries manually and then add those directory paths to the `libpaths.txt` file.

Once the `libpaths.txt` file is modified (if applicable), the user must execute any compiler environment scripts that may be necessary in order to run the compiler from the command line. For example, the Intel C/C++ compiler typically includes a script named `iclvars_ia32.bat` which allows the user to use the `icl` compiler command from the Windows shell prompt. Note that this step, wherein the user executes any applicable environment scripts, must be performed before executing `nmake`.

⁵If the user is working with a checked-out working copy from the `libflame` subversion repository, the script will update the file with the latest revision based on the revision specified within the `.svn\entries` file in the top-level `windows` directory.

Target	Function
<code>all</code>	Invoke the <code>lib</code> and <code>dll</code> targets.
<code>lib</code>	Build <code>libflame</code> as a static library.
<code>dll</code>	Build <code>libflame</code> as a dynamically-linked library.
<code>install</code>	Invoke the <code>install-lib</code> , <code>install-dll</code> , and <code>install-headers</code> targets.
<code>install-lib</code>	Invoke the <code>lib</code> target and then copy the library file to the <code>lib</code> subdirectory of the <code>libflame</code> install path specified in the <code>Makefile</code> .
<code>install-dll</code>	Invoke the <code>dll</code> target and then copy the library files to the <code>dll</code> subdirectory of the <code>libflame</code> install path specified in the <code>Makefile</code> .
<code>install-headers</code>	Copy the <code>libflame</code> header files to the install path specified in <code>Makefile</code> .
<code>help</code>	Output help and usage information.
<code>clean</code>	Invoke <code>clean-log</code> and <code>clean-build</code> targets.
<code>clean-log</code>	Remove any log files present.
<code>clean-config</code>	Remove all products of <code>configure.cmd</code> . Namely, remove the <code>config</code> , <code>include</code> , and <code>src</code> directories. Note that invoking the <code>clean-config</code> target will require the user to run <code>configure.cmd</code> again before being able to run any other <code>nmake</code> target.
<code>clean-build</code>	Remove all products of the compilation portion of the build process. Namely, remove the <code>obj</code> , <code>lib</code> , and <code>dll</code> directories.
<code>distclean</code>	Invoke <code>clean-log</code> , <code>clean-config</code> , and <code>clean-build</code> targets.

Table 3.3: A list of useful `nmake` targets defined in the `Makefile` for building `libflame` for Windows. Note that not all targets guarantee that action will take place. Most targets will not fire if `nmake` determines that the target is already up-to-date. For example, invoking the `clean` target will not remove any object files if they do not exist.

3.5 Compiling

After running `configure.cmd`, modifying `libpaths.txt`, and ensuring the compilers are operational from the command line, you may run `nmake`. Running `nmake` with no target specified causes the `all` target to be invoked implicitly. The `all` target causes both static and dynamic libraries to be built.

```
C:\field\temp\libflame-r3021\windows> nmake
```

Alternatively, the user may invoke the `lib` and/or `dll` targets individually. Invoking the `lib` target compiles the source code into object files and then archives them into a static library. Invoking the `dll` target compiles the source code and creates a dynamically-linked library without first creating the static library. Table 3.3 lists the most useful `nmake` targets defined in the `Makefile` that resides in the `windows` directory.

As `nmake` performs individual compiles individual source files into object files, it will output progress information. By default, this appears as:

```
C:\field\temp\libflame-r3021\windows> nmake

Microsoft (R) Program Maintenance Utility Version 9.00.21022.08
Copyright (C) Microsoft Corporation. All rights reserved.

nmake: Creating .\obj\flamec\x86\debug directory
nmake: Compiling .\src\flamec\cgelq2_ut.f
nmake: Compiling .\src\flamec\cgelqf_ut.f
nmake: Compiling .\src\flamec\cgeqr2_ut.f
nmake: Compiling .\src\flamec\cgeqrf_ut.f
nmake: Compiling .\src\flamec\cgetn2.f
nmake: Compiling .\src\flamec\cgetnf.f
nmake: Compiling .\src\flamec\clarfb_ut.f
nmake: Compiling .\src\flamec\clarfg_ut.f
nmake: Compiling .\src\flamec\clarft_ut.f
```

Link type	Component files	Purpose
static	libflame_x86_r2670.lib	The static library containing the libflame object files. Link to this file when statically linking your application to libflame.
dynamic	libflame_x86_r2670.dll	The dynamic library containing the libflame object code. This file is loaded into memory by the operating system at run-time the first time a dependent program or library references libflame symbols.
	libflame_x86_r2670.lib	The import library. This file contains information such as the dynamic library filename and which symbols are available within the dynamic library. The import library is used by the linker at link-time to resolve all function calls referenced by the application being built. If you plan to use a dynamic library build of libflame, reference this file when linking your application.
	libflame_x86_r2670.exp	The export file. This file is necessary only when building other dynamic libraries that depend on a dynamic library build of libflame.

Table 3.4: The files generated when building revision r2670 of libflame as either a static or dynamic library. The filenames reflect using “x86” as the architecture string when running `configure.cmd`.

```
nmake: Compiling .\src\flamec\dgelq2_ut.f
nmake: Compiling .\src\flamec\dgelqf_ut.f
nmake: Compiling .\src\flamec\dgeqr2_ut.f
nmake: Compiling .\src\flamec\dgeqrf_ut.f
nmake: Compiling .\src\flamec\dgetn2.f
nmake: Compiling .\src\flamec\dgetnf.f
```

When compilation is complete, the library will be archived. The output for static libraries will appear as:

```
nmake: Creating .\lib\x86\debug directory
nmake: Creating static library .\lib\x86\debug\libflame-x86-r3021.lib
```

And for dynamically-linked libraries as:

```
nmake: Creating .\dll\x86\debug directory
nmake: Creating dynamic library .\dll\x86\debug\libflame-x86-r3021.dll
Creating library libflame-x86-r3021.lib and object libflame-x86-r3021.exp
```

As you can see, the “x86” architecture string and “r3021” revision strings were inserted into the final library names. libflame is still under heavy development and undergoes frequent changes, and so the revision string is helpful for obvious reasons. Recall that the architecture string is completely arbitrary and has no effect on how the library gets built. However, it should be set to something reasonable to help you remember which environment was used to compile libflame.

The purpose of each file produced for static and dynamic builds of libflame is described in Table 3.4. The filenames in this table correspond to those that would result from building revision r3021 with the architecture string “x86”.

3.6 Installation

When a static library is built, the library file resides in a subdirectory of the `lib` directory, depending on the architecture and build type strings given to `configure.cmd`.

```
C:\field\temp\libflame-r3021\windows> dir lib\x86\debug
Volume in drive C is OS
Volume Serial Number is 941B-95F5

Directory of C:\field\temp\libflame-r3021\windows\lib\x86\debug

01/27/2009  03:18 PM    <DIR>          .
01/27/2009  03:18 PM    <DIR>          ..
01/27/2009  03:18 PM               4,557,904 libflame-x86-r3021.lib
                1 File(s)              4,557,904 bytes
                2 Dir(s)  85,190,336,512 bytes free
```

The dynamic library files reside in a similar directory named `dll`:

```
C:\field\temp\libflame-r3021\windows> dir dll\x86\debug
Volume in drive C is OS
Volume Serial Number is 941B-95F5

Directory of C:\field\temp\libflame-r3021\windows\dll\x86\debug

01/27/2009  03:19 PM    <DIR>          .
01/27/2009  03:19 PM    <DIR>          ..
01/27/2009  03:19 PM               2,642,944 libflame-x86-r3021.dll
01/27/2009  03:19 PM               264,384 libflame-x86-r3021.exp
01/27/2009  03:19 PM               460,424 libflame-x86-r3021.lib
                3 File(s)              3,367,752 bytes
                2 Dir(s)  86,630,637,568 bytes free
```

Once the static and/or dynamic libraries have been built, the library files may be copied out manually for use by the application developer. Alternatively, the user may specify an installation directory in the `build\defs.mk` file by setting the following variables:⁶

```
INSTALL_PREFIX = c:\field\lib
```

After these variables are set, one of the `nmake` install targets may be invoked. Invoking the `install` target, for example, results in both static and dynamic libraries being built, if they were not already, and then copied to their respective destination directories, along with the `libflame` header files:

```
C:\field\temp\libflame-r3021\windows> nmake install

Microsoft (R) Program Maintenance Utility Version 9.00.21022.08
Copyright (C) Microsoft Corporation. All rights reserved.

nmake: Installing .\lib\x86\release\libflame-x86-r3021.lib to c:\field\lib\libflame\lib
nmake: Installing .\dll\x86\release\libflame-x86-r3021.dll to c:\field\lib\libflame\dll
nmake: Installing .\dll\x86\release\libflame-x86-r3021.lib to c:\field\lib\libflame\dll
nmake: Installing .\dll\x86\release\libflame-x86-r3021.exp to c:\field\lib\libflame\dll
nmake: Installing libflame header files to c:\field\lib\libflame\include-x86-r3021
```

The user may also invoke the `install-lib` and/or `install-dll` targets separately to install only one library type or the other.

⁶ Of course, if the user is going to invoke an `install` target, he should first verify that he has permission to access and write to the directory specified in `build\defs.mk`. Otherwise, the file copy will fail.

3.7 Linking against libflame

This section will show you how to link a Windows build of `libflame` with your existing application. Let's assume that you've installed `libflame` to `c:\field\lib\libflame`. Let's also assume that you are building your application from the command line.⁷

In general, you should make the following changes to your application build process:

- **Add the `libflame` header directory to the include path of your compiler.** Usually, this is done by with the `/I` compiler option. For example, if you configured `libflame` r3021 with the “x86” build label, and specified that `configure.cmd` use `c:\field\lib\libflame` as the install prefix, then you would add `/Ic:\field\lib\libflame\include-x86-r3021` to the command line when invoking the compiler. Strictly speaking, this is only necessary when compiling source code files that use `libflame` symbols or APIs, but it is generally safe to use when compiling all of your application's source code.
- **Add `libflame` to the link command that links your application.** To link against `libflame`, you need to add `libflame_x86_r3021.lib` to your link command.

Now let's give a concrete example of these changes. Suppose you've been building your application with an `nmake` Makefile that looks something like:

```
SRC_PATH    = .
OBJ_PATH    = .
INC_PATH    = .

LIB_HOME    = c:\field\lib
BLAS_LIB    = $(LIB_HOME)\libblas.lib
LAPACK_LIB  = $(LIB_HOME)\liblapack.lib

CC          = cl.exe
LINKER      = link.exe
CFLAGS      = /nologo /O2 /I$(INC_PATH)
LDFLAGS     = /nologo

MYAPP_OBJS  = main.obj file.obj util.obj proc.obj
MYAPP_BIN   = my_app.exe

$(SRC_PATH).c$(OBJ_PATH).obj:
    $(CC) $(CFLAGS) /c $< /Fo$@

$(MYAPP): $(MYAPP_OBJS)
    $(LINKER) $(MYAPP_OBJS) $(LDFLAGS) $(LAPACK_LIB) $(BLAS_LIB) /out:$(MYAPP_BIN)

clean:
    del /F /Q $(MYAPP_OBJS) $(MYAPP_BIN)
    del /F /Q $(MYAPP_BIN)*.manifest
```

To link against `libflame`, you should change your Makefile as follows:

```
SRC_PATH    = .
OBJ_PATH    = .
INC_PATH    = .

LIB_HOME    = c:\field\lib
BLAS_LIB    = $(LIB_HOME)\libblas.a
LAPACK_LIB  = $(LIB_HOME)\liblapack.a

FLAME_HOME  = c:\field\lib\libflame
```

⁷ We acknowledge that most users will probably be using an integrated development environment to develop their programs. However, just as `libflame` only supports building from the command line, we will only demonstrate how to link against the library using `nmake` and leave it up to the motivated user to learn how to link against `libflame` from within whatever IDE he wishes.

```

FLAME_INC    = $(FLAME_HOME)\include-x86-r3021
FLAME_LIB    = $(FLAME_HOME)\dll\libflame-x86-r3021.lib

CC           = cl.exe
LINKER       = link.exe
CFLAGS       = /nologo /O2 /I$(INC_PATH) /I$(FLAME_INC)
LDFLAGS      = /nologo

MYAPP_OBJS   = main.obj file.obj util.obj proc.obj
MYAPP_BIN    = my_app.exe

$(SRC_PATH).c$(OBJ_PATH).obj:
    $(CC) $(CFLAGS) /c $< /Fo$@

$(MYAPP): $(MYAPP_OBJS)
    $(LINKER) $(MYAPP_OBJS) $(LDFLAGS) $(FLAME_LIB) $(LAPACK_LIB) $(BLAS_LIB) /out:$(MYAPP_BIN)

clean:
    del /F /Q $(MYAPP_BIN) $(MYAPP_OBJS)
    del /F /Q $(MYAPP_BIN).manifest

```

The changes appear in red.

First, we define the locations of `libflame` and the `libflame` header directory.

Second, we include the location of the `libflame` headers to the compilers' command line options so that the C compiler will be able to perform type checking against `libflame` declarations and prototypes.

Finally, we add the `libflame` library to the link command, making sure to insert it before the LAPACK and BLAS libraries.

Note that we are linking against a dynamically-linked build of `libflame`, which had its internal symbols resolved against the BLAS and LAPACK when the library was built. Thus, we only continue to link against the BLAS and LAPACK libraries under the assumption that the application still makes use of these libraries directly.

Chapter 4

Using libflame

This chapter contains code examples that illustrate how to use `libflame` in your application.

4.1 FLAME/C examples

Let us begin by illustrating a small program that uses LAPACK. Figure 4.1 contains a C language program that reads some input values, creates and fills a matrix, performs a Cholesky factorization on the matrix, and then frees the memory associated with the matrix buffer.

```
int main( void )
{
    double* buffer;
    int     m, ldim;
    int     info;
    char     uplo = 'L';

    // Get the matrix size and leading dimension.
    get_matrix_info( &m, &ldim );

    // Create and initialize an m x m matrix with leading dimension ldim.
    create_matrix( &buffer, m, ldim );
    initialize_matrix( buffer, m, ldim );

    // Compute the Cholesky factorization of the matrix, reading from and
    // updating the lower triangle.
    dpotrf_( &uplo, &m, buffer, &ldim, &info );

    // Free the matrix buffer.
    free_matrix( buffer );

    return 0;
}
```

Figure 4.1: A simple program that calls `dpotrf()` from LAPACK.

The program is trivial in that it does not do anything with the factored matrix before exiting. Furthermore, the corresponding code found in most real-world programs would most likely exist within a loop of some sort. However, we are keeping things simple here to better illustrate the usage of `libflame` functions.

Now suppose we wish to modify the previous program to use the FLAME/C API within `libflame`. There are two general methods.

- Create a `libflame` object without a buffer and then attach the conventional column-major matrix buffer to the bufferless `libflame` object. This method almost always requires the fewest number of

```

#include "FLAME.h"

int main( void )
{
    double* buffer;
    int     m, ldim;
    FLA_Obj A;

    // Initialize libflame.
    FLA_Init();

    // Get the matrix size and leading dimension.
    get_matrix_info( &m, &ldim );

    // Create and initialize an m x m matrix with leading dimension ldim.
    create_matrix( &buffer, m, ldim );
    initialize_matrix( buffer, m, ldim );

    // Create an m x m double-precision libflame object without a buffer,
    // and then attach the matrix buffer to the object.
    FLA_Obj_create_without_buffer( FLA_DOUBLE, m, m, &A );
    FLA_Obj_attach_buffer( buffer, ldim, &A );

    // Compute the Cholesky factorization, storing to the lower triangle.
    FLA_Chol( FLA_LOWER_TRIANGULAR, A );

    // Free the object without freeing the matrix buffer.
    FLA_Obj_free_without_buffer( &A );

    // Free the matrix buffer.
    free_matrix( buffer );

    // Finalize libflame.
    FLA_Finalize();

    return 0;
}

```

Figure 4.2: The program from Figure 4.1 modified to use `libflame` objects. This example code illustrates the minimal amount of work to use FLAME/C APIs in a program that was originally designed to use the BLAS or LAPACK.

code changes in the application.

- Modify the application such that the matrix is created natively along with the `libflame` object. This will require the user to interface the application to the matrix data within the object using various query routines. This method often involves more work because many applications are written to access matrix buffers directly without any abstractions. There are two different strategies for implementing this method, and depending on the nature of the application, one strategy may be more appropriate than the other:
 - The matrix may be created and fully initialized, and then copied into a `libflame` object.
 - The matrix may be created and initialized piecemeal, perhaps one block at a time.

Regardless of whether the matrix is initialized in full or one submatrix at a time, the user may use `FLA_Copy_submatrix_to_global()` to copy the data from a conventional column-major matrix arrays to `libflame` objects.

The program in Figure 4.2 uses the first method to integrate `libflame`. Note that changes from the original example are tracked in red. We start by inserting a `#include` directive for the `libflame` header


```

#include "FLAME.h"

int main( void )
{
    double* buffer;
    int     m, ldim;
    FLA_Obj A;

    // Initialize libflame.
    FLA_Init();

    // Get the matrix size and leading dimension.
    get_matrix_info( &m, &ldim );

    // Create and initialize an m x m matrix with leading dimension ldim.
    create_matrix( &buffer, m, ldim );
    initialize_matrix( buffer, m, ldim );

    // Create an m x m double-precision libflame object.
    FLA_Obj_create( FLA_DOUBLE, m, m, &A );

    // Copy the contents of the conventional matrix into a libflame object.
    FLA_Copy_submatrix_to_global( FLA_NO_TRANSPOSE, m, m, buffer, ldim, 0, 0, A );

    // Compute the Cholesky factorization, storing to the lower triangle.
    FLA_Chol( FLA_LOWER_TRIANGULAR, A );

    // Free the object.
    FLA_Obj_free( &A );

    // Free the matrix buffer.
    free_matrix( buffer );

    // Finalize libflame.
    FLA_Finalize();

    return 0;
}

```

Figure 4.3: The program from Figure 4.1 modified to use `libflame` objects natively. This code does not attach the conventional matrix buffer to a bufferless object and instead copies the matrix contents into the object using `FLA_Copy_submatrix_to_global()`. Note that the matrix is copied all at once, and thus here we assume that original matrix is fully initialized in `initialize_matrix()`

file, `FLAME.h`. Before calling any other `libflame` functions, we must first invoke `FLA_Init()`. Next, we replace the invocation to `dpotrf()` with four lines of `libflame` code. First, an $m \times m$ object `A` of datatype `FLA_DOUBLE` is created without a buffer. Then the matrix buffer `buffer` is attached to the `libflame` object, assuming a leading dimension of `ldim`. The Cholesky factorization is invoked on `A` with `FLA_Chol()`. And finally, the matrix object is released with `FLA_Obj_free_without_buffer()`. The library is finalized with a call to `FLA_Finalize()`.

The second method requires somewhat more extensive modifications to the original program. In Figure 4.3, we revise and extend the previous example. This program initializes the matrix as before, but then creates a `libflame` object natively (with an internal buffer), and then copies the contents of the conventional matrix into the `libflame` object all at once.

Finally, Figure 4.4 shows what a program might look like if it were to use a native `libflame` object but only copy over the data one block at a time. Here, we place `FLA_Copy_submatrix_to_global()` in a loop that copies a single submatrix per iteration. The body of `acquire_submatrix()` is not shown, but we envision this function to temporarily create or otherwise acquire the $b_m \times b_n$ matrix block whose top-left element is the (i, j) element of the overall matrix. Also note that we now are inputting a block size `b` instead of a

leading dimension.

Note that `FLA_Copy_submatrix_to_global()` may also be used to copy over one row or column at a time. Copying single rows or columns are just special cases of copying rectangular blocks.

4.2 FLASH examples

Now let us discuss how we might convert the `libflame` programs in Section 4.1 to use the FLASH API. Please see Section 5.4 for a full discussion of FLASH, including the motivation behind hierarchical objects and a summary of related terminology.

In the previous section, we reviewed a code (Figure 4.2) that uses `libflame` functions with an existing matrix buffer. Figure 4.5 shows what this code would look like if we wished to use hierarchical objects. Note that the changes from the corresponding FLAME/C code are highlighted in red. The application-specific code changes are limited to inputting a blocksize value to use in the creation of the hierarchical object `A`. All of the `libflame` function names are the same as in Figure 4.2 except that the prefix has changed from `FLA_` to `FLASH_`. Additionally, all of the function type signatures are the same, except for the invocation to `FLASH_Obj_create_without_buffer()`. This function takes two additional arguments: a depth, and an array of blocksizes.¹ The depth and the blocksize array together determine the details of the object hierarchy. Also note that since a conventional matrix buffer is being attached, the hierarchical object `A` will refer to submatrices that are not contiguous in memory.

In similar fashion, we have modified the code in Figure 4.3 to use hierarchical objects, as shown in Figure 4.6. The changes in this code are similar to those discussed for the previous example. Note that while `FLA_Copy_submatrix_to_global()` accepts a transposition argument, `FLASH_Copy_submatrix_to_global()` does not, and thus we had to remove this argument from the invocation of the latter function.

In Figure 4.7, we show the code from Figure 4.4 modified to use hierarchical objects. Once again, most of the differences are limited to changing the function prefixes. The one other change deserves additional attention, though, which is the use of the blocksize `b` in the object creation. In the previous code, the blocksize was used only to determine the sizes of the submatrices that were individually acquired and copied into the `A`. This code still uses the blocksize in this manner. However, it also uses the same value to establish the size of the submatrix blocks in the hierarchical object. It should be emphasized that `FLASH_Copy_submatrix_to_global()` allows the user to copy submatrices into the object that are different in size than the sizes of the underlying leaf-level blocks. That is, the function is capable of handling copies that span multiple block boundaries.

The key insight we hope to have impressed on our readers from these simple examples is that the FLASH API (1) provides an easy interface for creating and manipulating hierarchical objects, and (2) is strikingly similar to the original FLAME/C API wherever possible.

4.3 SuperMatrix examples

¹Since the depth is 1 in this example, we choose to simply pass the address of the integer `b` rather than create a separate single-element array.

```

#include "FLAME.h"

int main( void )
{
    int    m;
    int    b, i, j;
    FLA_Obj A;

    // Initialize libflame.
    FLA_Init();

    // Get the matrix size and block size.
    get_matrix_info( &m, &b );

    // Create an m x m double-precision libflame object.
    FLA_Obj_create( FLA_DOUBLE, m, m, &A );

    // Acquire the conventional matrix one block at a time and copy these
    // blocks into the appropriate location within the libflame object.
    for( j = 0; j < m; j += b )
    {
        for( i = 0; i < m; i += b )
        {
            double* ij_ptr;
            int    b_m, b_n, ldim;

            // Compute the block dimensions, in case they are blocks along the lower and/or
            // right edges of the overall matrix.
            b_m = ( m - i < b ? m - i : b );
            b_n = ( m - j < b ? m - j : b );

            // Get a pointer to the b_m x b_n block that starts at element (i,j), and also
            // get the leading dimension, in case it happens to be different from m.
            acquire_submatrix( i, j, b_m, b_n, &ij_ptr, &ldim );

            // Copy the current block into the correct location within the libflame object.
            FLA_Copy_submatrix_to_global( FLA_NO_TRANSPOSE, b_m, b_n, ij_ptr, ldim, i, j, A );
        }
    }

    // Compute the Cholesky factorization, storing to the lower triangle.
    FLA_Chol( FLA_LOWER_TRIANGULAR, A );

    // Free the object.
    FLA_Obj_free( &A );

    // Finalize libflame.
    FLA_Finalize();

    return 0;
}

```

Figure 4.4: The program from Figure 4.1 modified to use FLAME/C in a way that initializes a libflame object incrementally, one block at a time.

```
#include "FLAME.h"

int main( void )
{
    double* buffer;
    int     m, ldim, b;
    FLA_Obj A;

    // Initialize libflame.
    FLA_Init();

    // Get the matrix size and leading dimension.
    get_matrix_info( &m, &ldim, &b );

    // Create and initialize an m x m matrix with leading dimension ldim.
    create_matrix( &buffer, m, ldim );
    initialize_matrix( buffer, m, ldim );

    // Create an m x m double-precision hierarchical object without a buffer,
    // of depth 1 and blocksize b, and then attach the matrix buffer to the object.
    FLASH_Obj_create_without_buffer( FLA_DOUBLE, m, m, 1, &b, &A );
    FLASH_Obj_attach_buffer( buffer, ldim, &A );

    // Compute the Cholesky factorization, storing to the lower triangle.
    FLASH_Chol( FLA_LOWER_TRIANGULAR, A );

    // Free the object without freeing the matrix buffer.
    FLASH_Obj_free_without_buffer( &A );

    // Free the matrix buffer.
    free_matrix( buffer );

    // Finalize libflame.
    FLA_Finalize();

    return 0;
}
```

Figure 4.5: The program from Figure 4.2 modified to use the FLASH API.

```
#include "FLAME.h"

int main( void )
{
    double* buffer;
    int     m, ldim, b;
    FLA_Obj A;

    // Initialize libflame.
    FLA_Init();

    // Get the matrix size and leading dimension.
    get_matrix_info( &m, &ldim, &b );

    // Create and initialize an m x m matrix with leading dimension ldim.
    create_matrix( &buffer, m, ldim );
    initialize_matrix( buffer, m, ldim );

    // Create an m x m double-precision libflame object.
    FLASH_Obj_create( FLA_DOUBLE, m, m, 1, &b, &A );

    // Copy the contents of the conventional matrix into a libflame object.
    FLASH_Copy_submatrix_to_global( m, m, buffer, ldim, 0, 0, A );

    // Compute the Cholesky factorization, storing to the lower triangle.
    FLASH_Chol( FLA_LOWER_TRIANGULAR, A );

    // Free the object.
    FLASH_Obj_free( &A );

    // Free the matrix buffer.
    free_matrix( buffer );

    // Finalize libflame.
    FLA_Finalize();

    return 0;
}
```

Figure 4.6: The program from Figure 4.3 modified to use the FLASH API.

```

#include "FLAME.h"

int main( void )
{
    int      m;
    int      b, i, j;
    FLA_Obj A;

    // Initialize libflame.
    FLA_Init();

    // Get the matrix size and block size.
    get_matrix_info( &m, &b );

    // Create an m x m double-precision libflame object.
    FLASH_Obj_create( FLA_DOUBLE, m, m, 1, &b, &A );

    // Acquire the conventional matrix one block at a time and copy these
    // blocks into the appropriate location within the libflame object.
    for( j = 0; j < m; j += b )
    {
        for( i = 0; i < m; i += b )
        {
            double* ij_ptr;
            int      b_m, b_n, ldim;

            // Compute the block dimensions, in case they are blocks along the lower and/or
            // right edges of the overall matrix.
            b_m = ( m - i < b ? m - i : b );
            b_n = ( m - j < b ? m - j : b );

            // Get a pointer to the b_m x b_n block that starts at element (i,j), and also
            // get the leading dimension, in case it happens to be different from m.
            acquire_submatrix( i, j, b_m, b_n, &ij_ptr, &ldim );

            // Copy the current block into the correct location within the libflame object.
            FLASH_Copy_submatrix_to_global( b_m, b_n, ij_ptr, ldim, i, j, A );
        }
    }

    // Compute the Cholesky factorization, storing to the lower triangle.
    FLASH_Chol( FLA_LOWER_TRIANGULAR, A );

    // Free the object.
    FLASH_Obj_free( &A );

    // Finalize libflame.
    FLA_Finalize();

    return 0;
}

```

Figure 4.7: The program from Figure 4.4 modified to use the FLASH API.

Chapter 5

User-level Application Programming Interfaces

This chapter documents the user-level application programming interfaces (APIs) provided by `libflame`.

5.1 Conventions

Before describing the `libflame` APIs, let us take a moment to introduce and discuss some of the terminology that we use when discussing the interfaces. Besides introducing terms, we will, when appropriate, mention any implicit assumptions we make.

5.1.1 General terms

- *Matrix v. object.* Throughout this document we refer to both objects and matrices. There are many instances when the two words are used interchangeably. However, in other cases, the distinction is intentional. In these cases, an object refers to the data structure that represents the matrix (or vector or scalar) in question while a matrix refers to a mathematical entity. However, since we, as `libflame` developers and users, are only concerned with matrices as they are represented in computational environments, we often attribute object-like qualities to matrices, such as datatype and leading dimension.
- *Real matrix.* A real matrix is one that contains only real numbers.
- *Complex matrix.* A complex matrix is one that contains complex numbers. That is, every element in the matrix consists of a real and imaginary component.
- *General matrix.* A general matrix is one for which we make no special assumptions. That is, we do not assume any special structure concerning the upper or lower triangles, or the diagonal. General matrices are sometimes referred to as “full” matrices because algorithms that operate upon them must assume that each entry is non-zero.
- *Symmetric matrix.* A symmetric matrix is a square matrix whose (i, j) entry is equal to its (j, i) . In `libflame`, only the upper or lower triangle of a symmetric matrix is referenced.¹
- *Hermitian matrix.* A Hermitian matrix is a square complex matrix whose (i, j) entry is equal to the conjugate of its (j, i) . As such, the diagonal of a Hermitian matrix is always real. In `libflame`, only the upper or lower triangle of a Hermitian matrix is stored or referenced.¹

¹ Symmetric, Hermitian, and triangular matrices stored conventionally use the same amount of storage space as a general matrix with identical dimensions. That is, the conventional matrix storage scheme used by `libflame` does not attempt to save space by omitting the redundant (symmetric), conjugated (Hermitian), or zero (triangular) entries in the opposite triangle. The user is free to initialize the opposite triangle of the matrix, even if none of the computational routines will access it.

- *Triangular matrix.* A matrix is lower triangular if all non-zero entries appear on or below the diagonal, with entries above the diagonal equal to zero. Likewise, a matrix is upper triangular if all non-zero entries appear on or above the diagonal, with entries below the diagonal equal to zero. Triangular matrices are by definition square. In `libflame`, only the upper or lower triangle of a triangular matrix, whichever contains the non-zero entries, is stored or referenced.¹
- *Trapezoidal matrix.* A trapezoidal matrix is the rectangular analog of a triangular matrix. The name “trapezoidal” describes the shape of the area of the matrix containing non-zero entries. Specifically, a matrix is lower trapezoidal if $m > n$ and all non-zero entries appear on or below the diagonal, with entries above the diagonal equal to zero. Likewise, a matrix is upper trapezoidal if $m < n$ and all non-zero entries appear on or above the diagonal, with entries below the diagonal equal to zero.

5.1.2 Notation

- *Matrices, vectors, and scalars.* Throughout this text, we distinguish between matrices, vectors, and scalars in the following manner. Matrices are denoted by uppercase letters (examples: A , B , C). Vectors are denoted by lowercase letters (examples: v , x , y). Scalars are denoted by lowercase Greek letters (examples: α , β , ρ).

It is worth pointing out that a reference to a matrix A does not preclude A from being a vector or scalar in certain instances. Similarly, a reference to a vector x does not preclude x from being a 1×1 scalar. Thus, our choice of name reflects the most liberal assumptions we can make about the linear algebra entity in question.

Whether an entity is referred to as a matrix, vector, or scalar carries implications with respect to its dimensions. Matrices are $m \times n$ for $m, n \geq 0$ while vectors may either be $m \times 1$ or $1 \times n$ for $m, n \geq 0$.² Scalars, however, are always 1×1 .

- *Conjugation and conjugate transposition.* Within this document, we denote the complex conjugate transpose, or Hermitian transpose, of a matrix A as A^H . Similarly, we denote the conjugate of matrix A as \bar{A} .
- *BLAS and LAPACK routine notation.* Most operations implemented within the BLAS and LAPACK come in four separate implementations, one for each of the four floating-point numerical datatypes. These datatypes are usually encoded by the first letter of the routine name. For example, `dgemm()` implements the general matrix-matrix multiply (GEMM) operation for real matrices stored in double-precision floating-point format. Some level-1 routines stray slightly from this convention to handle situations where the datatypes of two arguments are expected to be different. The `zdscl()` routine implements a vector-scaling operation where a double-precision complex vector is scaled by a double-precision real scalar. In order to more easily refer to related families of routines, we use the following notation:
 - `?`: Used as a placeholder for the letter that identifies the datatype expected by the routine: (`s`, `d`, `c`, or `z`). Example: `?gemm()` refers to the four level-3 BLAS routines that implement the GEMM operation: `sgemm()`, `dgemm()`, `cgemm()`, and `zgemm()`.
 - `*`: Used as a placeholder for the letter or letters that identify the datatypes expected by the routine. The `*` character is used for only a handful of level-1 operations that require more than one letter to encode all datatype instances of the routine. Example: `*scal()` refers to the six level-1 BLAS routines that implement the SCAL operation: `sscal()`, `dscal()`, `cscal()`, `csscal()`, `zscal()`, and `zdscl()`.
- *Routine name qualifiers.* In the course of developing `libflame`, we found ourselves implementing extended variations of several BLAS operations. In order to distinguish these similar but distinct operations from their original counterparts, we use the following letters to encode the specific manner in which the operation was extended:

²We allow matrices and vectors with zero dimensions to facilitate matrix partitioning, which is a fundamental concept present in all FLAME algorithms[?].

Type	Typical parameter name	Permitted values	Of interest to...
FLA_Bool	<i>return value</i>	TRUE FALSE	all users
FLA_Datatype	datatype	FLA_INT FLA_FLOAT FLA_DOUBLE FLA_COMPLEX FLA_DOUBLE_COMPLEX FLA_CONSTANT	all users
FLA_Elemtypes	elemtypes	FLA_SCALAR FLA_MATRIX	advanced users and developers
FLA_Matrix_type	matrix_type	FLA_FLAT FLA_HIER	advanced users and developers
FLA_Side	side	FLA_LEFT FLA_RIGHT	all users
FLA_Uplo	uplo	FLA_LOWER_TRIANGULAR FLA_UPPER_TRIANGULAR	all users
FLA_Trans	trans	FLA_NO_TRANSPOSE FLA_TRANSPOSE FLA_CONJ_NO_TRANSPOSE FLA_CONJ_TRANSPOSE	all users
FLA_Conj	conj	FLA_NO_CONJUGATE FLA_CONJUGATE	all users
FLA_Diag	diag	FLA_NONUNIT_DIAG FLA_UNIT_DIAG FLA_ZERO_DIAG	all users
FLA_Quadrant	quadrant	FLA_TL FLA_TR FLA_BL FLA_BR	all users
FLA_Direct	direct	FLA_FORWARD FLA_BACKWARD	all users
FLA_Store	storev	FLA_ROWWISE FLA_COLUMNWISE	all users
FLA_Pivot_type	ptype	FLA_NATIVE_PIVOTS FLA_LAPACK_PIVOTS	all users
FLA_Error	<i>return value</i>	FLA_SUCCESS FLA_FAILURE ...	all users

Table 5.1: Table of libflame types and permitted values.

- **r**: Includes an uplo argument.
- **t**: Includes a trans argument.
- **c**: Includes a conjugation argument.
- **s**: Utilizes additional scalars.
- **x**: Accumulates to a different matrix or vector object.

So, for example, the `libflame` routine `FLA_Gemvc_external()` implements the same GEMV operation implemented by `FLA_Gemv_external()`, except that it allows the user to optionally conjugate the x vector argument. Likewise, the routine `FLA_Trmvsx_external()` implements an operation similar to the TRMV operation implemented in `FLA_Trmv_external()`, except that it allows the user to use additional scalars and accumulate the result into a separate vector rather than overwrite the contents of one of the original input arguments.

- *Constraints.* Some interface descriptions contain a section describing constraints placed on the implementation. These constraints may be imposed by the operation (e.g. “The length of vector x must be equal to the length of vector y .”) or by the interface (e.g. “The datatype of A must not be `FLA_CONSTANT`.”) These constraints correspond to internal safety checks performed by `libflame`. If one of these checks fails, then the implementation invokes `abort()`.³

Some things that would otherwise qualify as an operation constraint are not listed explicitly as constraints, but rather implied by the operation description (e.g. That x is defined as a vector.) These implicit constraints often still correspond to safety checks.

- *Types.* Table 5.1 lists all constant types and valid type values defined by `libflame`.
- *API descriptions.* The API descriptions in this document may contain various combinations of the following sections:
 - **Purpose.** Provides a general overview of the function, and/or a description of the mathematical operation that the function implements.
 - **Notes.** Describes additional information of a general nature.
 - **Int. Notes.** Describes additional information concerning the function interface.
 - **Imp. Notes.** Describes additional information concerning the function’s implementation within `libflame`.
 - **Dev. Notes.** This section is usually a note to developers, often a reminder of needed attention to a function that needs improvement.
 - **More Info.** Usually this section appears in documentation for a function that is very similar to another function, and points the reader elsewhere for further details of the operation being implemented.
 - **Returns.** A brief characterization of the type and value returned by the function.
 - **Caveats.** Contains warnings to the user on the function’s usage.
 - **Constraints.** A list of constraints on the function, including constraints imposed by the operation specification and its implementation within `libflame`. These constraints almost always correspond to checks that are performed at runtime.
 - **Arguments.** A list of function parameters with brief descriptions.

³The `libflame` developers understand that this behavior is overkill. Some might argue in favor of handling fatal errors through return values. We do not believe that offloading the burden of error checking to the user is the right answer. However, `libflame` may in the future offer a query routine that allows the application to query whether the library has encountered an error.

5.1.3 Objects

- *Numerical datatype.* The numerical datatype, or just datatype, of a matrix is a constant stored in the matrix object that determines the both the floating-point precision and the domain of the elements within the matrix. The constants `FLA_FLOAT` and `FLA_DOUBLE` identify matrix objects created to store single precision real and double precision real values, respectively. Likewise, `FLA_COMPLEX` and `FLA_DOUBLE_COMPLEX` identify matrix objects created to store single precision complex and double precision complex values, respectively. We also include `FLA_INT` in the category of numerical datatypes; however, we exclude `FLA_INT` when referring to *floating-point* numerical datatypes, or more simply, floating-point datatypes.
- *Leading dimension.* The leading dimension of the matrix object refers to the distance in memory that separates the beginning of one column from the beginning of the next column. Oftentimes, the leading dimension is equal to the m dimension, since our matrices in `libflame` are stored in column-major order. However, it is also quite common for a matrix object to refer to a submatrix of a larger matrix, in which case the leading dimension will be greater than the m dimension of the submatrix.
- *Row vectors v. column vectors.* A row vector is a vector with an m dimension of one, while a column vector is a vector with an n dimension of one. Given a column-major storage scheme, column vectors are contiguous in memory. The elements of a row vector, however, are almost never stored contiguously. Row vectors are often part of larger matrices, and in such cases their elements are separated by a leading dimension in memory. Vectors should be assumed to be column vectors unless otherwise qualified.
- *Indices.* The interfaces in `libflame` largely circumvent indices altogether. However, in some cases, indices are unavoidable. Furthermore, we use indices when describing some of the mathematical operations implemented in `libflame`. Unless otherwise indicated, the user should assume that all indices start with zero.
- *Conformal dimensions.* Various API descriptions use the term “conformal” to describe a requirement on the dimensions of two matrices. Matrices A and B are said to have conformal dimensions if A and B are both $m \times n$.
- *Storage.* Unless otherwise indicated, all matrices are stored conventionally in column-major order. We sometimes refer to these matrices as “flat” matrices.⁴
- *Transposition.* Many routines in `libflame` allow the user to optionally transpose one or more arguments as part of the operation. For example, the GEMM operation allows the user to transpose matrix A , or matrix B , or both. It is worth mentioning that this kind of transposition does not actually change the contents of matrices A or B . In these situations, the transposition is performed as part of the algorithm. In very few cases does the computation actually transpose the contents of a matrix, and these exceptions should be clear from the interface description.
- *Global scalar constants.* Many functions within `libflame` require the user to provide a 1×1 object to serve as a scaling factor in the operation in question. The GEMM operation, for example, has two of these scalars, α and β . For convenience, `libflame` defines the following global objects to represent commonly used scalars: `FLA_MINUS_ONE`, `FLA_ZERO`, `FLA_ONE`, `FLA_TWO`. These global scalar may be used wherever an operation reads, but does not write to or update, a scalar object. We’ve placed safeguards in most `libflame` functions that would prevent the user from changing these global scalar objects. Still, the user should consider them to be constant and should never attempt to update or overwrite them.

⁴In Section 5.4 especially, we use this term to contrast against “hierarchical” matrices, which is an alternate means of constructing and storing matrices.

5.2 FLAME/C Basics

5.2.1 Initialization and finalization

```
void FLA_Init( void );
```

Purpose: Initialize the library.

Notes: This function must be invoked before any other `libflame` functions.

```
void FLA_Finalize( void );
```

Purpose: Release all internal library resources. After `FLA_Finalize()` returns, `libflame` functions should not be used until `FLA_Init()` is called again.

Notes: This function should be invoked when your application is finished using `libflame`.

```
FLA_Bool FLA_Initialized( void );
```

Purpose: Check if the library is initialized.

Returns: A boolean value: `TRUE` if `libflame` is currently initialized; `FALSE` otherwise.

5.2.2 Object creation and destruction

```
FLA_Error FLA_Obj_create( FLA_Datatype datatype, dim_t m, dim_t n, FLA_Obj* obj );
```

Purpose: Create a new object from an uninitialized `FLA_Obj` structure. Upon returning, `obj` points to a valid heap-allocated $m \times n$ object whose elements are of numerical type `datatype`.

Returns: `FLA_SUCCESS`

Arguments:

- `datatype` – A constant corresponding to the numerical datatype requested.
- `m` – The number of rows to be created in new object.
- `n` – The number of columns to be created in the new object.
- `obj`
 - (on entry) – A pointer to an uninitialized `FLA_Obj`.
 - (on exit) – A pointer to a new `FLA_Obj` parameterized by `m`, `n`, and `datatype`.

```
FLA_Error FLA_Obj_create_conf_to( FLA_Trans trans, FLA_Obj obj_cur, FLA_Obj* obj_new );
```

Purpose: Create a new object `obj_new` with the same datatype and dimensions as an existing object `obj_cur`. The user may optionally create `obj_new` with the m and n dimensions transposed by specifying `FLA_TRANSPOSE` for the `trans` argument.

Notes: This function does not initialize the contents of `obj_new`.

Constraints:

- `trans` may not be `FLA_CONJ_TRANSPOSE` or `FLA_CONJ_NO_TRANSPOSE`.

Returns: `FLA_SUCCESS`

Arguments:

- | | | |
|----------------------|---|--|
| <code>trans</code> | – | Indicates whether to create the object pointed to by <code>obj_new</code> with transposed dimensions. |
| <code>obj_cur</code> | – | An existing <code>FLA_Obj</code> . |
| <code>obj_new</code> | | |
| (on entry) | – | A pointer to an uninitialized <code>FLA_Obj</code> . |
| (on exit) | – | A pointer to a new <code>FLA_Obj</code> parameterized by the datatype and dimensions of <code>obj_cur</code> . |

```
FLA_Error FLA_Obj_free( FLA_Obj* obj );
```

Purpose: Release all resources allocated to an object. `FLA_Obj_free()` must only be used with objects that were allocated with `FLA_Obj_create()` or `FLA_Obj_create_conf_to()`. Upon returning, `obj` points to a structure which is, for all intents and purposes, uninitialized.

Notes: If the object was created with `FLA_Obj_create_without_buffer()`, you should free the object with `FLA_Obj_free_without_buffer()`.

Returns: `FLA_SUCCESS`

Arguments:

- | | | |
|------------------|---|--|
| <code>obj</code> | | |
| (on entry) | – | A pointer to a valid <code>FLA_Obj</code> . |
| (on exit) | – | A pointer to an uninitialized <code>FLA_Obj</code> . |

5.2.3 General query functions

```
FLA_Datatype FLA_Obj_datatype( FLA_Obj obj );
```

Purpose: Query the numerical datatype of an object.

Returns: One of `{FLA_INT, FLA_FLOAT, FLA_DOUBLE, FLA_COMPLEX, FLA_DOUBLE_COMPLEX, FLA_CONSTANT}`.

Arguments:

- | | | |
|------------------|---|---------------------------|
| <code>obj</code> | – | An <code>FLA_Obj</code> . |
|------------------|---|---------------------------|

```
dim_t FLA_Obj_length( FLA_Obj obj );
```

Purpose: Query the number of rows in an object.

Returns: An unsigned integer value of type `dim_t`.

Arguments:
obj – An `FLA_Obj`.

```
dim_t FLA_Obj_width( FLA_Obj obj );
```

Purpose: Query the number of columns in an object.

Returns: An unsigned integer value of type `dim_t`.

Arguments:
obj – An `FLA_Obj`.

```
dim_t FLA_Obj_min_dim( FLA_Obj obj );
```

Purpose: Query the smaller of the object's length and width dimensions.

Returns: An unsigned integer value of type `dim_t`.

Arguments:
obj – An `FLA_Obj`.

```
dim_t FLA_Obj_max_dim( FLA_Obj obj );
```

Purpose: Query the larger of the object's length and width dimensions.

Returns: An unsigned integer value of type `dim_t`.

Arguments:
obj – An `FLA_Obj`.

```
dim_t FLA_Obj_vector_dim( FLA_Obj obj );
```

Purpose: If `obj` is a column or row vector, then return the number of elements in the vector. Otherwise, return to object's length.

Returns: An unsigned integer value of type `dim_t`.

Arguments:
obj – An `FLA_Obj`.

```
dim_t FLA_Obj_vector_inc( FLA_Obj obj );
```

Purpose: If `obj` is a column or row vector, then return the stride, or increment, that separates elements of the vector in memory. Otherwise, return 1.

Returns: An unsigned integer value of type `dim_t`.

Arguments:
obj – An `FLA_Obj`.

```
FLA_Datatype FLA_Obj_elemtype( FLA_Obj obj );
```

Purpose: Query the type of the elements contained within an object.

Notes: An object of element type `FLA_SCALAR` is also referred to as a “flat” object. By contrast, an object of element type `FLA_MATRIX` is considered hierarchical with a depth of at least one. More information on hierarchical matrices may be found in Section 5.4.

Returns: One of `{FLA_SCALAR, FLA_MATRIX}`.

Caveats: This is primarily a developer routine and should only be used by people who know what they are doing.

Arguments:

<code>obj</code>	–	An <code>FLA_Obj</code> .
------------------	---	---------------------------

```
size_t FLA_Obj_datatype_size( FLA_Datatype datatype );
```

Purpose: Query the size, in bytes, of an `FLA_Datatype` value.

Returns: An unsigned integer value of type `size_t`.

Caveats: This is primarily a developer routine and should only be used by people who know what they are doing.

Arguments:

<code>datatype</code>	–	An <code>FLA_Datatype</code> value.
-----------------------	---	-------------------------------------

```
size_t FLA_Obj_elem_size( FLA_Obj obj );
```

Purpose: Query the size, in bytes, of the elements within an `FLA_Obj`.

Returns: An unsigned integer value of type `size_t`.

Caveats: This is primarily a developer routine and should only be used by people who know what they are doing.

Arguments:

<code>obj</code>	–	An <code>FLA_Obj</code> .
------------------	---	---------------------------

```
FLA_Error FLA_Obj_show( char* header, char* format, FLA_Obj obj, char* footer );
```

Purpose: Display the numerical values inside an `FLA_Obj`.

Returns: `FLA_SUCCESS`

Arguments:

<code>obj</code>	–	An <code>FLA_Obj</code> .
<code>header</code>	–	A pointer to a string to precede the output of <code>obj</code> .
<code>format</code>	–	A pointer to a <code>printf()</code> -style format string.
<code>footer</code>	–	A pointer to a string to proceed the output of <code>obj</code> .

5.2.4 Interfacing with conventional matrix arrays

```
FLA_Error FLA_Obj_create_without_buffer( FLA_Datatype datatype, dim_t m, dim_t n,
                                         FLA_Obj* obj );
```

Purpose: Create a new object, except without any internal numerical data buffer. Before using the object, the user must attach a valid buffer with `FLA_Obj_attach_buffer()`.

Notes: The object's datatype will have already been set when `FLA_Obj_create_without_buffer()` returns, and therefore the user must take care to create the object with the datatype corresponding to the numerical values contained in the buffer he plans on attaching.

Returns: FLA_SUCCESS

Arguments:

- `datatype` – A constant corresponding to the numerical datatype requested.
- `m` – The number of rows to be created in new object.
- `n` – The number of columns to be created in the new object.
- `obj`
 - (on entry) – A pointer to an uninitialized `FLA_Obj`.
 - (on exit) – A pointer to a new, bufferless `FLA_Obj` parameterized by `m`, `n`, and `datatype`.

```
FLA_Error FLA_Obj_free_without_buffer( FLA_Obj* obj );
```

Purpose: Release all resources allocated to an object that was created without a data buffer. `FLA_Obj_free_without_buffer()` must only be used with objects that were allocated with `FLA_Obj_create_without_buffer()`. Upon returning, `obj` points to a structure which is, for all intents and purposes, uninitialized.

Notes: If the object was created with `FLA_Obj_create()` or `FLA_Obj_create_conf_to()`, you should free the object with `FLA_Obj_free()`.

Returns: FLA_SUCCESS

Arguments:

- `obj`
 - (on entry) – A pointer to a valid `FLA_Obj`.
 - (on exit) – A pointer to an uninitialized `FLA_Obj`.


```
FLA_Error FLA_Obj_attach_buffer( void* buffer, dim_t ldim, FLA_Obj* obj );
```

Purpose: Attach a user-allocated region of memory to an object that was created with `FLA_Obj_create_without_buffer()`. This routine is useful when the user, either by preference or necessity, wishes to allocate and/or initialize memory for linear algebra objects before encapsulating the data within an object structure. Note that it is important that the user submit the correct leading dimension `ldim`, which, combined with the `m` and `n` dimensions submitted when the object was created, will determine what region of memory is accessible. A leading dimension which is inadvertently set too large may result in memory accesses outside of the intended region during subsequent computation, which will likely cause undefined behavior.

Notes: When you are finished using an `FLA_Obj` with an attached buffer, you should free it with `FLA_Obj_free_without_buffer()`. However, you are still responsible for freeing the memory pointed to by `buffer` using `free()` or whatever memory deallocation function your system provides.

Returns: `FLA_SUCCESS`

Arguments:

- `buffer` – A valid region of memory allocated by the user. Typically, the address to this memory is obtained dynamically through a system function such as `malloc()`, but the memory may also be statically allocated.
- `ldim` – The leading dimension of the matrix stored conventionally in `buffer`.
- `obj`
 (on entry) – A pointer to a valid `FLA_Obj` that was created without a buffer.
 (on exit) – A pointer to a valid `FLA_Obj` that encapsulates the data in `buffer`.

```
void* FLA_Obj_buffer( FLA_Obj obj );
```

Purpose: Query the starting address of an object's underlying numerical data buffer. This function supports sub-object "views" in that it returns the address corresponding to the sub-object's current position within the overall object.

Notes: Since the address returned by `FLA_Obj_buffer()` is of type `void*`, the user must typecast it to one of the five numerical datatypes supported by the library (`int`, `float`, `double`, `complex`, `double complex`). The correct typecast may be determined with `FLA_Obj_datatype()`.

Returns: A pointer of type `void*`.

Arguments:

- `obj` – An `FLA_Obj`.

```
dim_t FLA_Obj_ldim( FLA_Obj obj );
```

Purpose: Query the leading dimension associated with the object's underlying element data buffer. The leading dimension is the number of elements that separates matrix element (r, c) from element $(r, c + 1)$.

Notes: `libflame` uses column-major storage. Thus, the leading dimension can be thought of as the number of elements allocated per column to the underlying base object into which `obj` is a view.

Returns: An unsigned integer value of type `dim_t`.

Arguments:

- `obj` – An `FLA_Obj`.

```
FLA_Error FLA_Copy_submatrix_to_global( FLA_Trans trans, dim_t m, dim_t n, void* A,
                                       dim_t ldim, dim_t i, dim_t j, FLA_Obj B );
```

Purpose: Copy the contents of an $m \times n$ conventional column-major matrix A with leading dimension $ldim$ into the submatrix B_{ij} whose top-left element is the (i, j) entry of B . The **trans** argument may be used to optionally transpose the matrix during the copy.

Notes: The user should ensure that the numerical datatype used in A is the same as the datatype used when B was created.

Constraints:

- If **trans** equals `FLA_NO_TRANSPOSE`, then B must be at least $i + m \times j + n$; otherwise, if **trans** equals `FLA_TRANSPOSE`, then B must be at least $i + n \times j + m$. Also, $ldim$ must be greater than or equal to m .
- **trans** may not be `FLA_CONJ_TRANSPOSE` or `FLA_CONJ_NO_TRANSPOSE`.
- The datatype of B may not be `FLA_CONSTANT`.

Returns: `FLA_SUCCESS`

Arguments:

trans	–	Indicates whether to transpose the matrix A during the copy.
m	–	The number of rows to copy from A to B_{ij} .
n	–	The number of columns to copy from A to B_{ij} .
A	–	A pointer to the first element in A .
ldim	–	The leading dimension of A .
i	–	The row offset in B of the submatrix B_{ij} .
j	–	The column offset in B of the submatrix B_{ij} .
B	–	An <code>FLA_Obj</code> representing matrix B .

```
FLA_Error FLA_Copy_global_to_submatrix( FLA_Trans trans, dim_t i, dim_t j, FLA_Obj A,
                                       dim_t m, dim_t n, void* B, dim_t ldim );
```

Purpose: Copy the contents of an $m \times n$ submatrix A_{ij} whose top-left element is the (i, j) entry of A into a conventional column-major matrix B with leading dimension $ldim$. The **trans** argument may be used to optionally transpose the submatrix during the copy.

Notes: The user should be aware of the numerical datatype of A and then access B accordingly.

Constraints:

- If **trans** equals `FLA_NO_TRANSPOSE`, then A must be at least $i + m \times j + n$; otherwise, if **trans** equals `FLA_TRANSPOSE`, then A must be at least $i + n \times j + m$. Also, $ldim$ must be greater than or equal to m .
- **trans** may not be `FLA_CONJ_TRANSPOSE` or `FLA_CONJ_NO_TRANSPOSE`.
- The datatype of A may not be `FLA_CONSTANT`.

Returns: `FLA_SUCCESS`

Arguments:

trans	–	Indicates whether to transpose the submatrix A_{ij} during the copy.
i	–	The row offset in A of the submatrix A_{ij} .
j	–	The column offset in A of the submatrix A_{ij} .
A	–	An <code>FLA_Obj</code> representing matrix A .
m	–	The number of rows to copy from A_{ij} to B .
n	–	The number of columns to copy from A_{ij} to B .
B	–	A pointer to the first element in B .
ldim	–	The leading dimension of B .

```
FLA_Error FLA_Axy_submatrix_to_global( FLA_Trans trans, FLA_Obj alpha,
                                       dim_t m, dim_t n, void* X, dim_t ldim,
                                       dim_t i, dim_t j, FLA_Obj Y );
```

Purpose: Perform one of the following operations:

$$\begin{aligned} Y_{ij} &:= Y_{ij} + \alpha X \\ Y_{ij} &:= Y_{ij} + \alpha X^T \end{aligned}$$

where α is a scalar, X is an $m \times n$ conventional column-major matrix, and Y_{ij} is the submatrix whose top-left element is the (i, j) entry of Y . The **trans** argument may be used to optionally transpose X during the operation.

Notes: The user should ensure that the numerical datatype used in X is the same as the datatype used when Y was created.

Constraints:

- If **trans** equals **FLA_NO_TRANSPOSE**, then Y must be at least $i + m \times j + n$; otherwise, if **trans** equals **FLA_TRANSPOSE**, then Y must be at least $i + n \times j + m$. Also, **ldim** must be greater than or equal to m .
- **trans** may not be **FLA_CONJ_TRANSPOSE** or **FLA_CONJ_NO_TRANSPOSE**.
- The datatype of Y may not be **FLA_CONSTANT**.

Returns: **FLA_SUCCESS**

Arguments:

trans	–	Indicates whether to transpose the matrix X during the operation.
alpha	–	An FLA_Obj representing scalar α .
m	–	The number of rows in X and Y_{ij} referenced by the operation.
n	–	The number of columns in X and Y_{ij} referenced by the operation.
X	–	A pointer to the first element in X .
ldim	–	The leading dimension of X .
i	–	The row offset in Y of the submatrix Y_{ij} .
j	–	The column offset in Y of the submatrix Y_{ij} .
Y	–	An FLA_Obj representing Y .

```
FLA_Error FLA_Axpy_global_to_submatrix( FLA_Trans trans, FLA_Obj alpha,
                                         dim_t i, dim_t j, FLA_Obj X,
                                         dim_t m, dim_t n, void* Y, dim_t ldim );
```

Purpose: Perform one of the following operations:

$$\begin{aligned} Y &:= Y + \alpha X_{ij} \\ Y &:= Y + \alpha X_{ij}^T \end{aligned}$$

where α is a scalar, X_{ij} is the submatrix whose top-left element is the (i, j) entry of X , and Y is an $m \times n$ conventional column-major matrix. The **trans** argument may be used to optionally transpose X_{ij} during the operation.

Notes: The user should be aware of the numerical datatype of X and then access Y accordingly.

Constraints:

- If **trans** equals `FLA_NO_TRANSPOSE`, then X must be at least $i + m \times j + n$; otherwise, if **trans** equals `FLA_TRANSPOSE`, then X must be at least $i + n \times j + m$. Also, **ldim** must be greater than or equal to m .
- **trans** may not be `FLA_CONJ_TRANSPOSE` or `FLA_CONJ_NO_TRANSPOSE`.
- The datatype of X may not be `FLA_CONSTANT`.

Returns: `FLA_SUCCESS`

Arguments:

trans	–	Indicates whether to transpose the matrix Y during the operation.
alpha	–	An <code>FLA_Obj</code> representing scalar α .
i	–	The row offset in X of the submatrix X_{ij} .
j	–	The column offset in X of the submatrix X_{ij} .
X	–	An <code>FLA_Obj</code> representing X .
m	–	The number of rows in X_{ij} and Y referenced by the operation.
n	–	The number of columns in X_{ij} and Y referenced by the operation.
Y	–	A pointer to the first element in Y .
ldim	–	The leading dimension of Y .

5.3 Managing Views

5.3.1 Vertical partitioning

```
FLA_Error FLA_Part_2x1( FLA_Obj A,  FLA_Obj* AT,
                      FLA_Obj* AB,
                      dim_t  mb,  FLA_Side side );
```

Purpose: Partition a matrix A into a top and bottom side where the side indicated by `side` has `mb` rows.

Returns: FLA_SUCCESS

Arguments:

- `A` – An FLA_Obj.
- `AT`
 - (on entry) – A pointer to an uninitialized FLA_Obj.
 - (on exit) – A pointer to an FLA_Obj view into the top side of A .
- `AB`
 - (on entry) – A pointer to an uninitialized FLA_Obj.
 - (on exit) – A pointer to an FLA_Obj view into the bottom side of A .
- `mb` – The number of rows to extract.
- `side` – The side to which to extract `mb` rows.

```
FLA_Error FLA_Repart_2x1_to_3x1( FLA_Obj AT,  FLA_Obj* A0,
                                FLA_Obj* A1,
                                FLA_Obj AB,  FLA_Obj* A2,
                                dim_t  mb,  FLA_Side side );
```

Purpose: Repartition a 2×1 partitioning of matrix A into a 3×1 partitioning where `mb` rows are split from the side indicated by `side`.

Returns: FLA_SUCCESS

Arguments:

- `AT, AB` – FLA_Obj structures that were partitioned via FLA_Part_2x1().
- `A0...A2`
 - (on entry) – Pointers to uninitialized FLA_Obj structures.
 - (on exit) – Pointers to FLA_Obj views into AT and AB .
- `mb` – The number of rows to extract.
- `side` – The side from which to extract `mb` rows.

```
FLA_Error FLA_Cont_with_3x1_to_2x1( FLA_Obj* AT,  FLA_Obj A0,
                                   FLA_Obj A1,
                                   FLA_Obj* AB,  FLA_Obj A2,
                                   FLA_Side side );
```

Purpose: Update the 2×1 partitioning of matrix A by moving the boundaries so that A_1 is shifted to the side indicated by `side`.

Returns: FLA_SUCCESS

Arguments:

- AT, AB
 - (on entry) – Pointers to FLA_Obj structures that were partitioned via FLA_Part_2x1() that do not yet reflect the repartitioning.
 - (on exit) – Pointers to FLA_Obj structures that were partitioned via FLA_Part_2x1() that reflect the new matrix boundaries.
- A0...A2 – FLA_Obj structures that were repartitioned via FLA_Part_2x1_to_3x1().
- side – The side to which to shift the `mb` rows of A_1 .

5.3.2 Horizontal partitioning

```
FLA_Error FLA_Part_1x2( FLA_Obj A,  FLA_Obj* AL, FLA_Obj* AR,
                       dim_t  nb, FLA_Side side );
```

Purpose: Partition a matrix A into a left and right side where the side indicated by `side` has `nb` columns.

Returns: FLA_SUCCESS

Arguments:

- A – An FLA_Obj.
- AL
 - (on entry) – A pointer to an uninitialized FLA_Obj.
 - (on exit) – A pointer to an FLA_Obj view into the left side of A .
- AR
 - (on entry) – A pointer to an uninitialized FLA_Obj.
 - (on exit) – A pointer to an FLA_Obj view into the right side of A .
- nb – The number of columns to extract.
- side – The side to which to extract `nb` columns.

```
FLA_Error FLA_Repart_1x2_to_1x3( FLA_Obj AL,          FLA_Obj AR,
                                FLA_Obj* A0, FLA_Obj* A1, FLA_Obj* A2,
                                dim_t  nb, FLA_Side side );
```

Purpose: Repartition a 1×2 partitioning of matrix A into a 1×3 partitioning where `nb` columns are split from the side indicated by `side`.

Returns: FLA_SUCCESS

Arguments:

- AL, AR – FLA_Obj structures that were partitioned via FLA_Part_1x2().
- A0...A2
 - (on entry) – Pointers to uninitialized FLA_Obj structures.
 - (on exit) – Pointers to FLA_Obj views into AL and AR .
- nb – The number of columns to extract.
- side – The side from which to extract `nb` columns.

```
FLA_Error FLA_Part_1x2( FLA_Obj* AL,          FLA_Obj* AR,
                       FLA_Obj  A0, FLA_Obj  A1, FLA_Obj  A2,
                       FLA_Side side );
```

Purpose: Update the 1×2 partitioning of matrix A by moving the boundaries so that A_1 is shifted to the side indicated by `side`.

Returns: FLA_SUCCESS

Arguments:

- AL, AR
 - (on entry) – Pointers to FLA_Obj structures that were partitioned via FLA_Part_1x2() that do not yet reflect the repartitioning.
 - (on exit) – Pointers to FLA_Obj structures that were partitioned via FLA_Part_1x2() that reflect the new matrix boundaries.
- A0...A2 – FLA_Obj structures that were repartitioned via FLA_Part_1x2_to_1x3().
- side – The side to which to shift the `nb` columns of A1.

5.3.3 Bidirectional partitioning

```
FLA_Error FLA_Part_2x2( FLA_Obj A,  FLA_Obj* ATL, FLA_Obj* ATR,
                       FLA_Obj* ABL, FLA_Obj* ABR,
                       dim_t  mb, dim_t  nb, FLA_Quadrant quadrant );
```

Purpose: Partition a matrix A into a four quadrants where the quadrant indicated by `quadrant` is $mb \times nb$.

Returns: FLA_SUCCESS

Arguments:

- A – An FLA_Obj.
- ATL...ABR
 - (on entry) – Pointers to uninitialized FLA_Obj structures.
 - (on exit) – Pointers to FLA_Obj views into the four quadrants of A.
- mb – The number of rows to extract.
- nb – The number of columns to extract.
- quadrant – The quadrant to which to extract `mb` rows and `nb` columns.

```

FLA_Error FLA_Repart_2x2_to_3x3(
    FLA_Obj ATL, FLA_Obj ATR,  FLA_Obj* A00, FLA_Obj* A01, FLA_Obj* A02,
                                FLA_Obj* A10, FLA_Obj* A11, FLA_Obj* A12,
    FLA_Obj ABL, FLA_Obj ABR,  FLA_Obj* A20, FLA_Obj* A21, FLA_Obj* A22,
    dim_t  mb,  dim_t  nb,  FLA_Quadrant quadrant );

```

Purpose: Repartition a 2×2 partitioning of matrix A into a 3×3 partitioning where $mb \times nb$ submatrix A_{11} is split from the quadrant indicated by `quadrant`.

Returns: FLA_SUCCESS

Arguments:

- ATL...ABR – FLA_Obj structures that were partitioned via FLA_Part_2x2().
- A00...A22
 - (on entry) – Pointers to uninitialized FLA_Obj structures.
 - (on exit) – Pointers to FLA_Obj views into ATL, ATR, ABL, and ABR.
- mb – The number of rows to extract.
- nb – The number of columns to extract.
- quadrant – The quadrant from which to shift the `mb` rows and `nb` columns of A_{11} .

```

FLA_Error FLA_Cont_with_3x3_to_2x2(
    FLA_Obj* ATL, FLA_Obj* ATR,  FLA_Obj A00, FLA_Obj A01, FLA_Obj A02,
                                FLA_Obj A10, FLA_Obj A11, FLA_Obj A12,
    FLA_Obj* ABL, FLA_Obj* ABR,  FLA_Obj A20, FLA_Obj A21, FLA_Obj A22,
    FLA_Quadrant quadrant );

```

Purpose: Update the 2×2 partitioning of matrix A by moving the boundaries so that A_{11} is shifted to the quadrant indicated by `quadrant`.

Returns: FLA_SUCCESS

Arguments:

- ATL...ABR
 - (on entry) – Pointers to FLA_Obj structures that were partitioned via FLA_Part_2x2() that do not yet reflect the repartitioning.
 - (on exit) – Pointers to FLA_Obj structures that were partitioned via FLA_Part_2x2() that reflect the new matrix boundaries.
- A00...A22 – FLA_Obj structures that were repartitioned via FLA_Part_2x2_to_3x3().
- quadrant – The quadrant to which to shift the `mb` rows and `nb` columns of A_{11} .

5.3.4 Merging views

```
FLA_Error FLA_Merge_2x1( FLA_Obj AT,
                        FLA_Obj AB,  FLA_Obj* A );
```

Purpose: Merge a 2×1 set of adjacent matrix views into a single view.

Constraints:

- AT and AB must be views into the same object.
- AT and AB must be vertically adjacent and vertically aligned.
- AT and AB must have an equal number of columns.

Returns: FLA_SUCCESS

Arguments:

AT, AB	–	Valid FLA_Obj views eligible for merging.
A		
(on entry)	–	A pointer to an uninitialized FLA_Obj.
(on exit)	–	A pointer to an FLA_Obj view that represents the merging of AL and AR.

```
FLA_Error FLA_Merge_1x2( FLA_Obj AL, FLA_Obj AR,  FLA_Obj* A );
```

Purpose: Merge a 1×2 set of adjacent matrix views into a single view.

Constraints:

- AL and AR must be views into the same object.
- AL and AR must be horizontally adjacent and horizontally aligned.
- AL and AR must have an equal number of rows.

Returns: FLA_SUCCESS

Arguments:

AL, AR	–	Valid FLA_Obj views eligible for merging.
A		
(on entry)	–	A pointer to an uninitialized FLA_Obj.
(on exit)	–	A pointer to an FLA_Obj view that represents the merging of AT and AB.

```
FLA_Error FLA_Merge_2x2( FLA_Obj ATL, FLA_Obj ATR,
                        FLA_Obj ABL, FLA_Obj ABR,  FLA_Obj* A );
```

Purpose: Merge a 2×2 set of adjacent matrix views into a single view.

Constraints:

- ATL, ATR, ABL, and ABR must be views into the same object.
- The number of rows in ATL and ABL must equal that of ATR and ABR, respectively.
- The number of columns in ATL and ATR must equal that of ABL and ABR, respectively.
- ATL and ATR must be vertically adjacent and vertically aligned to ABL and ABR, respectively.
- ATL and ABL must be horizontally adjacent and horizontally aligned to ATR and ABR, respectively.

Returns: FLA_SUCCESS

Arguments:

- | | | |
|------------|---|---|
| ATL...ABR | – | Valid FLA_Obj views to be merged. |
| A | | |
| (on entry) | – | A pointer to an uninitialized FLA_Obj. |
| (on exit) | – | A pointer to an FLA_Obj view that represents the merging of ATL, ABL, ATR, and ABR. |

5.4 FLASH

5.4.1 Motivation

Traditionally, dense matrices are stored in column-major order. That is, matrices are stored as a sequence of columns, with the elements of the j th column stored contiguously, beginning at memory location $l_{dim,j}$, where l_{dim} is the leading dimension of the matrix. This particular storage scheme works fine for matrices small enough to fit in the processor's level-2 cache [?, ?]. However, for larger matrices, the larger leading dimensions result in attenuated performance. The cause is primarily due to lack of spacial locality across columns and increased TLB misses from accessing a larger region of memory [?].

Alternative data storage schemes have been explored thoroughly. In particular, storage-by-blocks has shown promise as a storage scheme capable of delivering higher performance. The idea, in principle, is straightforward: instead of storing the entire matrix column-major order, store individual blocks of the matrix contiguously.⁵ When paired with an algorithm that performs its computation on individual blocks, this storage scheme can reduce cache and TLB misses and result in better performance.

However, at the time of this writing, storage-by-blocks is not widely used. The most likely reason stems from the difficulty of indexing directly into the submatrices. Storage-by-blocks tends to require complicated indexing expressions, which further obfuscates the algorithm as expressed in its implementation. This inability to easily index into the matrix makes it difficult to even initialize the matrix, let alone implement an algorithm that operate upon it. Thus, the unpleasantness of storage-by-blocks is felt by both the library implementor and the user alike.

The FLAME project presents a solution to this problem in [?]. As an extension to `libflame`, the FLASH API provides a set of interfaces that allows a user to create, initialize, and compute with matrices stored by blocks. More generally, FLASH provides an interfaces for managing hierarchical matrices, which, when set to contain one level of hierarchy, allows us to easily implement storage-by-blocks. For now, FLASH only supports one level of hierarchy, but in principle multiple levels have potential applications for out-of-core computation and sparse matrix storage. The FLAME project intends to investigate these possibilities in future research.

⁵Presumably, each of these individual blocks would be stored in column-major order, but row-major order is also possible. Actually, the exact storage scheme of the blocks is not important, as long as they are stored in a manner that is compatible with the computational kernels that will operate upon the blocks.

5.4.2 Concepts

This section is devoted to introducing and defining various concepts that will reoccur throughout our descriptions of the FLASH API.

- *Conventional object.* Conventional objects, also known as “flat” objects, are those which are created using the traditional FLAME/C API. In `libflame`, flat objects store their numerical data contiguously, in column-major.
- *Hierarchy.* The hierarchy of a matrix refers to the internal tree-like structure of object that represents and stores the matrix.
- *Hierarchical object.* Hierarchical objects, also referred to as objects “stored by blocks”, are those which are created using the FLASH API. Hierarchical objects contain a matrix hierarchy.
- *Block.* A block is a submatrix numerical data which is typically a part of a larger hierarchical matrix. In `libflame`, individual blocks use the same storage scheme as flat objects.
- *Node.* Since matrix hierarchies resemble trees, we sometimes use “node” as a synonym to refer to objects within a matrix hierarchy.
- *Element.* Elements are the immediate constituent members of a matrix object. The nature of an object’s elements is determined by the element type, which may be either `FLA_SCALAR` or `FLA_MATRIX`. The former identifies a matrix object which contains numerical data while the later refers to a matrix object whose elements are themselves references to other submatrix objects.
- *Leaf object.* The leaf object is an object in a matrix hierarchy that encapsulates a submatrix whose elements contains actual numerical data (ie: an object which encapsulates a block). Leaf objects always have an element type of `FLA_SCALAR`.
- *Non-leaf object.* A non-leaf object is an object in a matrix hierarchy that encapsulates a submatrix whose elements contains references to other objects. Non-leaf objects always have an element type of `FLA_MATRIX`. In `libflame`, non-leaf objects store their elements in column-major order.
- *Child object.* Child objects are those objects referred to by the elements contained within a non-leaf object. Child objects may contain additional levels of hierarchy (if they are of element type `FLA_MATRIX`) or they may encapsulate numerical data (if they are of element type `FLA_SCALAR`). Only non-leaf objects may have child objects.
- *Root object.* The root object of a matrix hierarchy corresponds to the top-level structure that is visible to the user. When a root object is also a leaf object, then the matrix has no hierarchy and thus is effectively equivalent to a matrix object stored conventionally in column-major order.
- *Depth.* The depth of a matrix hierarchy is defined as the distance from the root object to any leaf object⁶. A depth of zero means the object has no hierarchy.
- *Level.* A level in a hierarchy refers to all objects that are some constant distance from the root. Level 0 refers to the root object, level 1 refers to the children of the root object, and so on.
- *Element length.* The element length, also referred to as simply “the length”, of an object refers to the number of element rows within the object, where these elements may be contiguous blocks or references to deeper portions of the matrix hierarchy.
- *Element width.* The element width, also referred to as simply “the width”, of an object refers to the number of element columns within the object. The semantics are otherwise identical to that of element length.

⁶Currently, the FLASH API assumes that all leaf objects are equidistant from the root. This may change in a future revision.

- *Scalar length.* The scalar length of a hierarchical object refers to the number of rows in the matrix that the object represents. We distinguish between this from the element length of the object, which refers to the number of rows of elements in the object *at that level* in the hierarchy. Put another way, the scalar length is a property of the matrix as a mathematical entity, while the element length is a property of an individual node within the hierarchy that represents the matrix. As such, the user is typically only concerned with the scalar length of an object, while developers of `libflame` must routinely query both the scalar length and element length of hierarchical objects.
- *Scalar width.* The scalar width of a hierarchical object refers to the number of columns in the matrix that the object represents. The semantics are otherwise identical to those of scalar length.
- *Blocksize.* The blocksize is a property of a non-leaf object, and refers to the element dimensions of its child objects. Specifically, it refers to the element length and width of the child objects, *not* the element length and width. The blocksize(s) used by a hierarchical object are set when the object is created and may not be subsequently changed.
- *Hierarchical conformality.* Two objects A and B are hierarchically conformal when the following conditions are satisfied:
 - The depth of A is equal to the depth of B .
 - For every level in the hierarchies of both objects, the element length and/or width of A equals the corresponding dimension of B . Whether only the element lengths are equal, or only the element widths are equal, or that they are both equal, depends on the context. In a matrix-matrix multiply operation $C = C + AB$, hierarchical conformality requires, for every level, that: the element length of A must equal the element length of C ; the element width of A equal the element length of B ; and the element width of B equal the element width of C . Alternately, in the context of the triangular matrix multiply operation $B := LB$, where L is a lower triangular matrix, hierarchical conformality only requires the element length (which equals the element width because L is square) of L equal the element length of B .

Almost all FLASH functions that involve two matrix arguments require that the matrices be hierarchically conformal.

5.4.3 Interoperability with FLAME/C

The FLASH API is an extension to the base FLAME/C interfaces. That is, from the perspective of the library developer, FLASH employs much of the internal machinery present in the FLAME/C framework. However, objects that are created as hierarchical objects via any of the FLASH object creation routines should *not* be used with any of the base FLAME/C interfaces except by developers and other experts who know what they are doing. The FLASH API includes a basic but complete set of routines for creating, destroying, querying, and managing hierarchical objects. The API also provides computational routines that support the matrices stored by blocks. As a general rule of thumb, once a hierarchical object has been created the user should only use that object with routines that begin with the `FLASH_` prefix.

The FLASH API, as written, should accept flat matrix objects without any problems. When a flat matrix is passed into a FLASH routine, the underlying implementation simply invokes the appropriate code for a flat matrix object.

The remaining subsections, 5.4.4 through 5.4.7, document the core set of APIs provided by FLASH. The computational routines are documented alongside their conventional FLAME/C brethren in Section 5.7.

5.4.4 Object creation and destruction

```
void FLASH_Obj_create( FLA_Datatype datatype, dim_t m, dim_t n, dim_t depth,
                      dim_t* b_mn, FLA_Obj* H );
```

Purpose: Create a new hierarchical object from an uninitialized `FLA_Obj` structure. Upon returning, H points to a valid heap-allocated object that refers to a $m \times n$ matrix of numerical datatype `datatype`. Furthermore, H will have a hierarchical depth of `depth` and the value in `b_mn[i]` will specify the square block sizes for the $i + 1$ th level of the hierarchy. Only the first `depth` values of `blocksizes` will be referenced.

Notes: If `depth` > 0 , the matrix will be hierarchical. In this case, the dimensions of the root matrix are not explicitly specified and instead are determined by the block sizes at each hierarchical level combined with the dimensions of the overall hierarchical matrix. If `depth` $= 0$, the matrix will be flat and have no hierarchy, in which case the dimensions of the root matrix are the same as the dimensions of the overall matrix.

Constraints:

- Neither m nor n may be zero.
- `datatype` may not be `FLA_CONSTANT`.
- The pointer arguments `b_mn` and H must not be `NULL`.
- Each of the first `depth` values in `b_mn` must be greater than zero.

Arguments:

<code>datatype</code>	–	A constant corresponding to the numerical datatype requested.
<code>m</code>	–	The number of rows to be created in new object.
<code>n</code>	–	The number of columns to be created in the new object.
<code>depth</code>	–	The number of levels to create in the hierarchy of H .
<code>b_mn</code>	–	A pointer to an array of <code>depth</code> values to be used as block sizes in creating the matrix hierarchy of H .
H		
	(on entry) –	A pointer to an uninitialized <code>FLA_Obj</code> .
	(on exit) –	A pointer to a new hierarchical <code>FLA_Obj</code> parameterized by <code>datatype</code> , m , n , <code>depth</code> , and <code>b_mn</code> .

Supports flat objects? Yes.

```
void FLASH_Obj_create_ext( FLA_Datatype datatype, dim_t m, dim_t n, dim_t depth,
                          dim_t* b_m, dim_t* b_n, FLA_Obj* H );
```

Purpose: Create a new hierarchical object from an uninitialized `FLA_Obj` structure. Upon returning, `H` points to a valid heap-allocated object that refers to a $m \times n$ matrix of numerical datatype `datatype`. Furthermore, `H` will have a hierarchical depth of `depth` and the values in `b_m[i]` and `b_n[i]` will specify the blocksizes in the row and column dimension, respectively, for the $i + 1$ th level of the hierarchy. Only the first `depth` values of `b_m` and `b_n` will be referenced.

Notes: If `depth > 0`, the matrix will be hierarchical. In this case, the dimensions of the root matrix are not explicitly specified and instead are determined by the row and column blocksizes at each hierarchical level combined with the dimensions of the overall hierarchical matrix. If `depth = 0`, the matrix will be flat and have no hierarchy, in which case the dimensions of the root matrix are the same as the dimensions of the overall matrix.

Constraints:

- Neither m nor n may be zero.
- `datatype` may not be `FLA_CONSTANT`.
- The pointer arguments `b_m`, `b_n`, and `H` must not be `NULL`.
- Each of the first `depth` values in `b_m` and `b_n` must be greater than zero.

Arguments:

- | | |
|-----------------------|---|
| <code>datatype</code> | – A constant corresponding to the numerical datatype requested. |
| <code>m</code> | – The number of rows to be created in new object. |
| <code>n</code> | – The number of columns to be created in the new object. |
| <code>depth</code> | – The number of levels to create in the hierarchy of <code>H</code> . |
| <code>b_m</code> | – A pointer to an array of <code>depth</code> values to be used as the row dimensions of the blocksizes needed when creating the matrix hierarchy of <code>H</code> . |
| <code>b_n</code> | – A pointer to an array of <code>depth</code> values to be used as the column dimensions of the blocksizes needed when creating the matrix hierarchy of <code>H</code> . |
| <code>H</code> | <div style="margin-left: 20px;"> <div>(on entry) – A pointer to an uninitialized <code>FLA_Obj</code>.</div> <div>(on exit) – A pointer to a new hierarchical <code>FLA_Obj</code> parameterized by <code>datatype</code>, <code>m</code>, <code>n</code>, <code>depth</code>, <code>b_m</code>, and <code>b_n</code>.</div> </div> |

Supports flat objects? Yes.

```
void FLASH_Obj_create_conf_to( FLA_Trans trans, FLA_Obj H_cur, FLA_Obj* H_new );
```

Purpose: Create a new hierarchical object with the same datatype, dimensions, depth, and block-sizes as an existing hierarchical object. The user may optionally create the object pointed to by `H_new` with the m and n dimensions transposed by specifying `FLA_TRANSPOSE` for the `trans` argument.

Notes: This function does not initialize the contents of `H_new`.

Constraints:

- `trans` may not be `FLA_CONJ_TRANSPOSE` or `FLA_CONJ_NO_TRANSPOSE`.

Arguments:

- | | | |
|--------------------|---|---|
| <code>trans</code> | – | Indicates whether to create the object pointed to by <code>H_new</code> with transposed dimensions. |
| <code>H_cur</code> | – | An existing hierarchical <code>FLA_Obj</code> . |
| <code>H_new</code> | | |
| (on entry) | – | A pointer to an uninitialized <code>FLA_Obj</code> . |
| (on exit) | – | A pointer to a new hierarchical <code>FLA_Obj</code> parameterized by the datatype, dimensions, depth, and blocksizes of <code>H_cur</code> . |

Supports flat objects? Yes.

```
void FLASH_Obj_free( FLA_Obj* H );
```

Purpose: Release all resources allocated to a hierarchical object. `FLASH_Obj_free()` must only be used with objects that were allocated with `FLASH_Obj_create()`, `FLASH_Obj_create_conf_to()`, `FLASH_Obj_create_hier_conf_to_flat()`, or `FLASH_Obj_create_hier_copy_of_flat()`. Upon returning, `H` points to a structure which is, for all intents and purposes, uninitialized.

Notes: If the object was created with `FLASH_Obj_create_without_buffer()`, you should free the object with `FLASH_Obj_free_without_buffer()`.

Arguments:

- | | | |
|----------------|---|--|
| <code>H</code> | | |
| (on entry) | – | A pointer to a valid hierarchical <code>FLA_Obj</code> . |
| (on exit) | – | A pointer to an uninitialized <code>FLA_Obj</code> . |

Supports flat objects? Yes.

5.4.5 Interfacing with flat matrix objects

```
void FLASH_Obj_create_hier_conf_to_flat( FLA_Trans trans, FLA_Obj F, dim_t depth,
                                         dim_t* blocksizes, FLA_Obj* H );
```

Purpose: Create a new hierarchical object H with the same datatype and dimensions as an existing flat object F . The function will create H with a matrix hierarchy specified by the **depth** and **blocksizes** arguments. The user may optionally create H with the m and n dimensions transposed by specifying **FLA_TRANSPOSE** for the **trans** argument.

Notes: This function does not initialize the contents of H .

Constraints:

- **trans** may not be **FLA_CONJ_TRANSPOSE** or **FLA_CONJ_NO_TRANSPOSE**.
- The pointer arguments **blocksizes** and **H** must not be **NULL**.
- Each of the first **depth** values in **blocksizes** must be greater than zero.

Arguments:

- | | |
|-------------------|---|
| trans | – Indicates whether to create the object pointed to by H with transposed dimensions. |
| F | – An existing flat FLA_Obj representing matrix F . |
| depth | – The number of levels to create in the hierarchy of H . |
| blocksizes | – A pointer to an array of depth values to be used as blocksizes in creating the matrix hierarchy of H . |
| H | |
| (on entry) | – A pointer to an uninitialized FLA_Obj . |
| (on exit) | – A pointer to a new hierarchical FLA_Obj parameterized by the datatype and dimensions of F , depth , and blocksizes . |

Supports flat objects? Yes.


```
void FLASH_Obj_create_hier_conf_to_flat_ext( FLA_Trans trans, FLA_Obj F, dim_t depth,
                                             dim_t* b_m, dim_t* b_n, FLA_Obj* H );
```

Purpose: Create a new hierarchical object H with the same datatype and dimensions as an existing flat object F . The function will create H with a matrix hierarchy specified by the **depth**, **b_m**, and **b_n** arguments. The user may optionally create H with the m and n dimensions transposed by specifying **FLA_TRANSPOSE** for the **trans** argument.

Notes: This function does not initialize the contents of H .

Constraints:

- **trans** may not be **FLA_CONJ_TRANSPOSE** or **FLA_CONJ_NO_TRANSPOSE**.
- The pointer arguments **b_m**, **b_n**, and **H** must not be **NULL**.
- Each of the first **depth** values in **b_m** and **b_n** must be greater than zero.

Arguments:

- | | | |
|--------------|--------------|---|
| trans | – | Indicates whether to create the object pointed to by H with transposed dimensions. |
| F | – | An existing flat FLA_Obj representing matrix F . |
| depth | – | The number of levels to create in the hierarchy of H . |
| b_m | – | A pointer to an array of depth values to be used as the row dimensions of the blocksizes needed when creating the matrix hierarchy of H . |
| b_n | – | A pointer to an array of depth values to be used as the column dimensions of the blocksizes needed when creating the matrix hierarchy of H . |
| H | | |
| | (on entry) – | A pointer to an uninitialized FLA_Obj . |
| | (on exit) – | A pointer to a new hierarchical FLA_Obj parameterized by the datatype and dimensions of F , depth , b_m , and b_n . |

Supports flat objects? Yes.

```
void FLASH_Obj_create_hier_copy_of_flat( FLA_Obj F, dim_t depth,
                                          dim_t* blocksizes, FLA_Obj* H );
```

Purpose: Create a new hierarchical object H with the same datatype and dimensions as an existing flat object F and then copy the numerical contents of F to H . The function will create H with a matrix hierarchy specified by the **depth** and **blocksizes** arguments.

Constraints:

- The pointer arguments **blocksizes** and **H** must not be **NULL**.
- Each of the first **depth** values in **blocksizes** must be greater than zero.

Arguments:

- | | | |
|-------------------|--------------|--|
| F | – | An existing flat FLA_Obj representing matrix F . |
| depth | – | The number of levels to create in the hierarchy of H . |
| blocksizes | – | A pointer to an array of depth values to be used as blocksizes in creating the matrix hierarchy of H . |
| H | | |
| | (on entry) – | A pointer to an uninitialized FLA_Obj . |
| | (on exit) – | A pointer to a new hierarchical FLA_Obj parameterized by the datatype and dimensions of F , depth , and blocksizes , and which contains the contents of the flat matrix F . |

Supports flat objects? Yes.

```
void FLASH_Obj_create_hier_copy_of_flat_ext( FLA_Obj F, dim_t depth,
                                              dim_t* b_m, dim_t* b_n, FLA_Obj* H );
```

Purpose: Create a new hierarchical object H with the same datatype and dimensions as an existing flat object F and then copy the numerical contents of F to H . The function will create H with a matrix hierarchy specified by the `depth`, `b_m`, and `b_n` arguments.

Constraints:

- The pointer arguments `b_m`, `b_n`, and `H` must not be NULL.
- Each of the first `depth` values in `b_m` and `b_n` must be greater than zero.

Arguments:

- | | | |
|--------------------|---|--|
| <code>F</code> | – | An existing flat <code>FLA_Obj</code> representing matrix F . |
| <code>depth</code> | – | The number of levels to create in the hierarchy of H . |
| <code>b_m</code> | – | A pointer to an array of <code>depth</code> values to be used as the row dimensions of the blocksizes needed when creating the matrix hierarchy of H . |
| <code>b_n</code> | – | A pointer to an array of <code>depth</code> values to be used as the column dimensions of the blocksizes needed when creating the matrix hierarchy of H . |
| <code>H</code> | | |
| (on entry) | – | A pointer to an uninitialized <code>FLA_Obj</code> . |
| (on exit) | – | A pointer to a new hierarchical <code>FLA_Obj</code> parameterized by the datatype and dimensions of F , <code>depth</code> , <code>b_m</code> , and <code>b_n</code> , and which contains the contents of the flat matrix F . |

Supports flat objects? Yes.

```
void FLASH_Obj_create_flat_conf_to_hier( FLA_Trans trans, FLA_Obj H, FLA_Obj* F );
```

Purpose: Create a new flat object F with the same datatype and dimensions as an existing flat object H . The user may optionally create F with the m and n dimensions transposed by specifying `FLA_TRANSPOSE` for the `trans` argument.

Notes: This function does not initialize the contents of F .

Constraints:

- `trans` may not be `FLA_CONJ_TRANSPOSE` or `FLA_CONJ_NO_TRANSPOSE`.
- The pointer argument `F` must not be NULL.

Arguments:

- | | | |
|--------------------|---|--|
| <code>trans</code> | – | Indicates whether to create the object pointed to by F with transposed dimensions. |
| <code>H</code> | – | An existing hierarchical <code>FLA_Obj</code> representing matrix H . |
| <code>F</code> | | |
| (on entry) | – | A pointer to an uninitialized <code>FLA_Obj</code> . |
| (on exit) | – | A pointer to a new flat <code>FLA_Obj</code> parameterized by the datatype and dimensions of F . |

Supports flat objects? Yes.

```
void FLASH_Obj_create_flat_copy_of_hier( FLA_Obj H, FLA_Obj* F );
```

Purpose: Create a new flat object F with the same datatype and dimensions as an existing hierarchical object H and then copy the numerical contents of F to H .

Constraints:

- The pointer argument F must not be NULL.

Arguments:

- | | | |
|-----|------------|--|
| H | – | An existing hierarchical FLA_Obj representing matrix H . |
| F | | |
| | (on entry) | – A pointer to an uninitialized FLA_Obj. |
| | (on exit) | – A pointer to a new flat FLA_Obj parameterized by the datatype and dimensions of F , and which contains the contents of the hierarchical matrix F . |

Supports flat objects? Yes.

```
void FLASH_Copy_submatrix_to_global( dim_t m, dim_t n, void* F, dim_t ldim,
                                     dim_t i, dim_t j, FLA_Obj H );
```

Purpose: Copy the contents of an conventional column-major matrix F with leading dimension $ldim$ into the submatrix H_{ij} whose top-left element is the (i, j) entry of hierarchical matrix H , where both F and H_{ij} are $m \times n$.

Notes: The user should ensure that the numerical datatype used in F is the same as the datatype used when H was created.

Constraints:

- The numerical datatype of H must not be FLA_CONSTANT.
- H must be at least $i + m \times j + n$. Also, $ldim$ must be greater than or equal to m .
- The pointer argument F must not be NULL.

Arguments:

- | | | |
|--------|---|--|
| m | – | The number of rows to copy from F to H_{ij} . |
| n | – | The number of columns to copy from F to H_{ij} . |
| F | – | A pointer to the first element in conventional column-major matrix F . |
| $ldim$ | – | The leading dimension of F . |
| i | – | The row offset in H of the submatrix H_{ij} . |
| j | – | The column offset in H of the submatrix H_{ij} . |
| H | – | A hierarchical FLA_Obj representing matrix H . |

Supports flat objects? Yes.

```
void FLASH_Copy_global_to_submatrix( dim_t i, dim_t j, FLA_Obj H,
                                     dim_t m, dim_t n, void* F, dim_t ldim );
```

Purpose: Copy the contents of the submatrix H_{ij} whose top-left element is the (i, j) entry of hierarchical matrix H into an conventional column-major matrix F with leading dimension $ldim$, where both H_{ij} and F are $m \times n$.

Notes: The user should be aware of the numerical datatype of H and then access F accordingly.

Constraints:

- The numerical datatype of H must not be `FLA_CONSTANT`.
- H must be at least $i + m \times j + n$. Also, $ldim$ must be greater than or equal to m .
- The pointer argument F must not be `NULL`.

Arguments:

- | | |
|--------|--|
| i | – The row offset in H of the submatrix H_{ij} . |
| j | – The column offset in H of the submatrix H_{ij} . |
| H | – A hierarchical <code>FLA_Obj</code> representing matrix H . |
| m | – The number of rows to copy from H_{ij} to F . |
| n | – The number of columns to copy from H_{ij} to F . |
| F | – A pointer to the first element in conventional column-major matrix F . |
| $ldim$ | – The leading dimension of F . |

Supports flat objects? Yes.

```
void FLASH_Copy_subobject_to_global( FLA_Obj F, dim_t i, dim_t j, FLA_Obj H );
```

Purpose: Copy the contents of a flat matrix F into the submatrix H_{ij} whose top-left element is the (i, j) entry of hierarchical matrix H , where both F and H_{ij} are $m \times n$.

Constraints:

- The numerical datatypes of F and H must be identical and must not be `FLA_CONSTANT`.
- H must be at least $i + m \times j + n$.

Arguments:

- | | |
|-----|---|
| F | – A flat <code>FLA_Obj</code> representing matrix F . |
| i | – The row offset in H of the submatrix H_{ij} . |
| j | – The column offset in H of the submatrix H_{ij} . |
| H | – A hierarchical <code>FLA_Obj</code> representing matrix H . |

Supports flat objects? Yes.

```
void FLASH_Copy_global_to_subobject( dim_t i, dim_t j, FLA_Obj H, FLA_Obj F );
```

Purpose: Copy the contents of the submatrix H_{ij} whose top-left element is the (i, j) entry of hierarchical matrix H into a flat matrix F , where both H_{ij} and F are $m \times n$.

Constraints:

- The numerical datatypes of F and H must be identical and must not be `FLA_CONSTANT`.
- H must be at least $i + m \times j + n$.

Arguments:

- | | | |
|---|---|---|
| i | – | The row offset in H of the submatrix H_{ij} . |
| j | – | The column offset in H of the submatrix H_{ij} . |
| H | – | A hierarchical <code>FLA_Obj</code> representing matrix H . |
| F | – | A flat <code>FLA_Obj</code> representing matrix F . |

Supports flat objects? Yes.

```
void FLASH_Obj_hierarchify( FLA_Obj F, FLA_Obj H );
```

Purpose: Copy the contents of a flat matrix F into a hierarchical matrix H , where both H and F are $m \times n$.

Imp. Notes: This function is currently implemented as:

```
FLASH_Copy_subobject_to_global( F, 0, 0, H );
```

Constraints:

- The numerical datatypes of F and H must be identical and must not be `FLA_CONSTANT`.
- H must be at least $m \times n$.

Arguments:

- | | | |
|---|---|---|
| F | – | A flat <code>FLA_Obj</code> representing matrix F . |
| H | – | A hierarchical <code>FLA_Obj</code> representing matrix H . |

Supports flat objects? Yes.

```
void FLASH_Obj_flatten( FLA_Obj H, FLA_Obj F );
```

Purpose: Copy the contents of a hierarchical matrix H into a flat matrix F , where both H and F are $m \times n$.

Imp. Notes: This function is currently implemented as:

```
FLASH_Copy_global_to_subobject( 0, 0, F, H );
```

Constraints:

- The numerical datatypes of F and H must be identical and must not be `FLA_CONSTANT`.
- H must be at least $m \times n$.

Arguments:

- | | | |
|---|---|---|
| H | – | A hierarchical <code>FLA_Obj</code> representing matrix H . |
| F | – | A flat <code>FLA_Obj</code> representing matrix F . |

Supports flat objects? Yes.

5.4.6 Interfacing with conventional matrix arrays

```
void FLASH_Obj_create_without_buffer( FLA_Datatype datatype, dim_t m, dim_t n,
                                     dim_t depth, dim_t* blocksizes, FLA_Obj* H );
```

Purpose: Create a new hierarchical object from an uninitialized `FLA_Obj` structure, just as with `FLASH_Obj_create()`, except without any internal numerical data buffer. Before using the object, the user must attach a valid buffer with `FLASH_Obj_attach_buffer()`.

Constraints:

- Neither m nor n may be zero.
- `datatype` may not be `FLA_CONSTANT`.
- The pointer arguments `blocksizes` and `H` must not be `NULL`.
- Each of the first `depth` values in `blocksizes` must be greater than zero.

Arguments:

- | | | |
|-------------------------|---|--|
| <code>datatype</code> | – | A constant corresponding to the numerical datatype requested. |
| <code>m</code> | – | The number of rows to be created in new object. |
| <code>n</code> | – | The number of columns to be created in the new object. |
| <code>depth</code> | – | The number of levels of hierarchy in the object that represents matrix H . |
| <code>blocksizes</code> | – | A pointer to an array of <code>depth</code> values to be used as blocksizes in creating the matrix hierarchy of H . |
| <code>H</code> | | |
| (on entry) | – | A pointer to an uninitialized <code>FLA_Obj</code> . |
| (on exit) | – | A pointer to a new, bufferless hierarchical <code>FLA_Obj</code> parameterized by <code>m</code> , <code>n</code> , <code>depth</code> , <code>blocksizes</code> , and <code>datatype</code> . |

Supports flat objects? Yes.

```
void FLASH_Obj_create_without_buffer_ext( FLA_Datatype datatype, dim_t m, dim_t n,
                                         dim_t depth, dim_t* b_m, dim_t* b_n,
                                         FLA_Obj* H );
```

Purpose: Create a new hierarchical object from an uninitialized `FLA_Obj` structure, just as with `FLASH_Obj_create_ext()`, except without any internal numerical data buffer. Before using the object, the user must attach a valid buffer with `FLASH_Obj_attach_buffer()`.

Constraints:

- Neither m nor n may be zero.
- `datatype` may not be `FLA_CONSTANT`.
- The pointer arguments `b_m`, `b_n`, and `H` must not be `NULL`.
- Each of the first `depth` values in `b_m` and `b_n` must be greater than zero.

Arguments:

- | | | |
|-----------------------|--------------|--|
| <code>datatype</code> | – | A constant corresponding to the numerical datatype requested. |
| <code>m</code> | – | The number of rows to be created in new object. |
| <code>n</code> | – | The number of columns to be created in the new object. |
| <code>depth</code> | – | The number of levels of hierarchy in the object that represents matrix H . |
| <code>b_m</code> | – | A pointer to an array of <code>depth</code> values to be used as the row dimensions of the blocksizes needed when creating the matrix hierarchy of H . |
| <code>b_n</code> | – | A pointer to an array of <code>depth</code> values to be used as the column dimensions of the blocksizes needed when creating the matrix hierarchy of H . |
| <code>H</code> | | |
| | (on entry) – | A pointer to an uninitialized <code>FLA_Obj</code> . |
| | (on exit) – | A pointer to a new, bufferless hierarchical <code>FLA_Obj</code> parameterized by <code>m</code> , <code>n</code> , <code>depth</code> , <code>b_m</code> , <code>b_n</code> , and <code>datatype</code> . |

Supports flat objects? Yes.

```
void FLASH_Obj_free_without_buffer( FLA_Obj* H );
```

Purpose: Release all resources allocated to a hierarchical object that was created without a data buffer. `FLASH_Obj_free_without_buffer()` should be used only with objects that were allocated `FLASH_Obj_create_without_buffer()`. Upon returning, `obj` points to a structure which is, for all intents and purposes, uninitialized.

Notes: If the object was created with `FLASH_Obj_create()` or `FLASH_Obj_create_conf_to()`, you should free the object with `FLASH_Obj_free()`.

Arguments:

- | | | |
|----------------|--------------|--|
| <code>H</code> | | |
| | (on entry) – | A pointer to a valid hierarchical <code>FLA_Obj</code> . |
| | (on exit) – | A pointer to an uninitialized <code>FLA_Obj</code> . |

Supports flat objects? Yes.

```
void FLASH_Obj_attach_buffer( void* buffer, dim_t ldim, FLA_Obj* H );
```

Purpose: Attach a user-allocated region of memory to a hierarchical object that was created with `FLASH_Obj_create_without_buffer()`. This routine is useful when the user, either by preference or necessity, wishes to allocate and/or initialize memory for linear algebra objects before encapsulating the data within a hierarchical object structure. Note that it is important that the user submit the correct leading dimension `ldim`, which, combined with the `m` and `n` dimensions submitted when the object was created, will determine what region of memory is accessible. A leading dimension which is inadvertently set too large may result in memory accesses outside of the intended region during subsequent computation, which will likely cause undefined behavior.

Notes: When you are finished using a hierarchical `FLA_Obj` with an attached buffer, you should free it with `FLASH_Obj_free_without_buffer()`. However, you are still responsible for freeing the memory pointed to by `buffer` using `free()` or whatever memory deallocation function your system provides.

Caveats: This routine is not an ideal way to retrofit hierarchical storage into your application. The problem is that a “native” hierarchical object, one which was created with its own data buffer, will contain leaf objects that refer to blocks that are contiguous in memory, which provides performance benefits in the way of spacial locality. If a user creates a hierarchical object without a buffer and then attaches an existing matrix stored conventionally, the memory referred to by individual leaf objects will not be contiguous due to the large leading dimension of the conventional matrix. Therefore, we highly encourage users to create hierarchical matrices one of two other ways:

- Use `FLASH_Obj_create()` and then initialize the matrix elements incrementally, one submatrix at a time, with `FLASH_Copy_subobject_to_global()` or `FLASH_Copy_submatrix_to_global()`.
- Use `FLASH_Obj_create_hier_copy_of_flat()` to create a hierarchical object and initialize it with the contents of an existing flat object.

Arguments:

- | | | |
|---------------------|---|--|
| <code>buffer</code> | – | A valid region of memory allocated by the user. Typically, the address to this memory is obtained dynamically through a system function such as <code>malloc()</code> , but the memory may also be statically allocated. |
| <code>ldim</code> | – | The leading dimension of the matrix stored conventionally in <code>buffer</code> . |
| <code>H</code> | | |
| (on entry) | – | A pointer to a valid hierarchical <code>FLA_Obj</code> that was created without a buffer. |
| (on exit) | – | A pointer to a valid hierarchical <code>FLA_Obj</code> that encapsulates the data in <code>buffer</code> . |

Supports flat objects? Yes.

5.4.7 Object query functions

```
FLA_Datatype FLASH_Obj_datatype( FLA_Obj H );
```

Purpose: Query the numerical datatype of H . This corresponds to the numerical datatype of the data stored at the leaves of the matrix heirarchy.

Returns: A constant of type `FLA_Datatype`.

Arguments:

H — An `FLA_Obj` representing matrix H .

Supports flat objects? Yes.

```
dim_t FLASH_Obj_scalar_length( FLA_Obj H );
```

Purpose: Query the scalar length of H . That is, query the number of rows in the matrix represented by the object H .

Notes: Using `FLASH_Obj_scalar_length()` on a flat matrix will always return the correct value. However, using `FLA_Obj_length()` on a hierarchical matrix will return the number of rows of child objects within the the top level of the hierarchy of H . The user should be aware of the difference, as the latter situation is usually only of interest to developers.

Returns: An unsigned integer value of type `dim_t` representing the number of rows in H .

Arguments:

H — An `FLA_Obj` representing matrix H .

Supports flat objects? Yes.

```
dim_t FLASH_Obj_scalar_width( FLA_Obj H );
```

Purpose: Query the scalar width of H . That is, query the number of columns in the matrix represented by the object H .

Notes: Using `FLASH_Obj_scalar_width()` on a flat matrix will always return the correct value. However, using `FLA_Obj_width()` on a hierarchical matrix will return the number of columns of child objects within the the top level of the hierarchy of H . The user should be aware of the difference, as the latter situation is usually only of interest to developers.

Returns: An unsigned integer value of type `dim_t` representing the number of columns in H .

Arguments:

H — An `FLA_Obj` representing matrix H .

Supports flat objects? Yes.

```
dim_t FLASH_Obj_depth( FLA_Obj H );
```

Purpose: Query the depth of the object representing matrix H . This corresponds to the number of links between the root the hierarchy and the leaf objects. A depth of zero indicates that H is a flat matrix.

Notes: This routine assumes that all leaves are equidistant from the root object H .

Returns: An unsigned integer value of type `dim_t` representing the depth of the hierarchy within the object representing matrix H .

Arguments:

H – An `FLA_Obj` representing matrix H .

Supports flat objects? Yes.

```
dim_t FLASH_Obj_blocksizes( FLA_Obj H, dim_t* b_m, dim_t* b_n );
```

Purpose: Query the row and column blocksizes used at each level of hierarchy within the object that represents matrix H and store the values within the array pointed to by `b_m` and `b_n`. The number of values stored to `b_m` and `b_n` will be equal to the depth of H , which is returned by the function.

Notes: If H is a flat matrix, then no values are written to `b_m` or `b_n` and zero is returned. It is important that the length of the `b_m` and `b_n` arrays be sufficiently large to handle the depth of H .

Returns: An unsigned integer value of type `dim_t` representing the depth of H and number of blocksizes stored to the `b_m` and `b_n` arrays.

Arguments:

H – An `FLA_Obj` representing matrix H .
`b_m` – A pointer to an array of unsigned integers in which to store the row blocksizes of the matrix hierarchy of H .
`b_n` – A pointer to an array of unsigned integers in which to store the column blocksizes of the matrix hierarchy of H .

Supports flat objects? Yes.

5.5 SuperMatrix

5.5.1 Overview

SuperMatrix is an extension to the FLAME/C and FLASH APIs that enables task-level parallel execution via algorithms-by-blocks [?]. The SuperMatrix runtime system itself is dependency-aware, and therefore is a major step forward when compared to more primitive workqueuing-based solutions [?].

The mechanism works as follows. Subproblems within a FLAME algorithm implementation are replaced, via macros, with calls to a routine that enqueues all pertinent information about the subproblem onto a global task queue. This information includes a function pointer to the computational routine that would normally execute the subproblem and references to the subproblem’s arguments. The algorithm is then run sequentially, at which time the subproblem instances, or tasks, are enqueued. As tasks are enqueued, a dependency graph is incrementally constructed, which tracks flow, anti-, and output dependencies between tasks. After enqueueing is complete, the SuperMatrix runtime system is invoked. Tasks marked as “ready” are dequeued by independent threads and executed. When a task is complete, the dependency graph is updated, and unexecuted tasks are marked as ready as soon as all of their dependencies are satisfied. This

process continues until all tasks have been executed.

A computational routine parallelized by SuperMatrix uses the same algorithmic variant implementations employed by sequential FLAME/C and sequential FLASH routines. For interested developers or other curious readers, you may find a discussion of the mechanism that makes this reuse of code possible in Section 6.5.

The interface to the SuperMatrix mechanism and characteristics of its `libflame` implementation have been thoroughly documented in the literature [?, ?]. Please see these texts for further information regarding SuperMatrix.

5.5.2 API

In this subsection we document all of the `libflame` interfaces needed to use SuperMatrix in your application. The developer-level interfaces are documented in Section 6.4.

```
FLA_Error FLASH_Queue_enable( void );
```

- Purpose:** Enable SuperMatrix. By enabling SuperMatrix, the user enables algorithm-level shared memory parallelism within FLASH-based computational routines. If SuperMatrix is already enabled, the function has no effect.
- Notes:** If SuperMatrix was enabled at configure-time, `FLA_Init()` will call this function, and thus the user does not need to invoke it unless SuperMatrix was temporarily disabled via `FLASH_Queue_disable()`. If SuperMatrix was disabled at configure-time, the function aborts with an error message.
- Returns:** `FLA_SUCCESS` if successful or if SuperMatrix is already enabled; `FLA_FAILURE` if the function was called from within a parallel region (ie: after `FLASH_Queue_begin()` and before `FLASH_Queue_end()`).

```
FLA_Error FLASH_Queue_disable( void );
```

- Purpose:** Disable SuperMatrix. By disabling SuperMatrix, the user disables algorithm-level shared memory parallelism within FLASH-based computational routines. When SuperMatrix is disabled, these routines revert back to executing sequentially, though they still expect hierarchical storage. If SuperMatrix is already disabled, the function has no effect.
- Notes:** If SuperMatrix was enabled at configure-time, the user should only invoke this function if he wants to temporarily disable SuperMatrix in order to run sequential FLASH implementations. If SuperMatrix was disabled at configure-time, the function unconditionally returns `FLA_SUCCESS`.
- Returns:** `FLA_SUCCESS` if successful or if SuperMatrix was disabled at configure-time; `FLA_FAILURE` if the function was called from within a parallel region (ie: after `FLASH_Queue_begin()` and before `FLASH_Queue_end()`).

```
FLA_Bool FLASH_Queue_get_enabled( void );
```

- Purpose:** Query whether SuperMatrix is currently enabled.
- Notes:** If SuperMatrix was disabled at configure-time, the function unconditionally returns `FALSE`.
- Returns:** `TRUE` if SuperMatrix was enabled at configure-time and is also currently enabled; `FALSE` if SuperMatrix was disabled at configure-time or if SuperMatrix was enabled at configure-time but is currently disabled.

```
void FLASH_Queue_begin( void );
```

Purpose: Mark the beginning of a parallel region. The parallel region continues until the user invokes `FLASH_Queue_end()`.

Notes: Any FLASH computational routines found in a parallel region will be parallelized in a way that overlaps the tasks' computation in whatever order the scheduler sees fit while still observing dependencies between tasks.

```
void FLASH_Queue_end( void );
```

Purpose: Mark the end of a parallel region. The parallel region begins when the user invokes `FLASH_Queue_begin()`.

Notes: Any FLASH computational routines found in a parallel region will be parallelized in a way that overlaps the tasks' computation in whatever order the scheduler sees fit while still observing dependencies between tasks.

```
void FLASH_Queue_set_num_threads( unsigned int n_threads );
```

Purpose: Set the number of threads that SuperMatrix will use when executing tasks in parallel.

Notes: This routine does not immediately cause SuperMatrix to spawn any threads.

Arguments:

`n_threads` – An unsigned integer representing the number of threads to be requested upon parallel execution.

```
unsigned int FLASH_Queue_get_num_threads( void );
```

Purpose: Query the number of threads that SuperMatrix is currently set to use when executing tasks in parallel.

Returns: An unsigned integer representing the number of threads that SuperMatrix is currently set to use in parallel execution.

```
void FLASH_Queue_set_verbose_output( FLA_Bool verbose );
```

Purpose: Enable or disable verbosity in SuperMatrix. In verbose mode, SuperMatrix will print extra information to standard output as execution progresses.

Arguments:

`verbose` – A boolean value that either enables (`TRUE`) or disables (`FALSE`) SuperMatrix verbosity.

```
FLA_Bool FLASH_Queue_get_verbose_output( void );
```

Purpose: Query the current status of verbosity in SuperMatrix.

Returns: A boolean value; `TRUE` if SuperMatrix is currently set to run in verbose mode, `FALSE` otherwise.

```
void FLASH_Queue_set_sorting( FLA_Bool sorting );
```

Purpose: Enable or disable task sorting in SuperMatrix. When sorting is enabled, SuperMatrix will sort its queue of ready-and-waiting tasks according to some heuristic.

Arguments:

`sorting` – A boolean value that either enables (TRUE) or disables (FALSE) SuperMatrix task sorting.

```
FLA_Bool FLASH_Queue_get_sorting( void );
```

Purpose: Query the current status of task sorting in SuperMatrix.

Returns: A boolean value; TRUE if SuperMatrix is currently set to sort tasks prior to execution, FALSE otherwise.

```
void FLASH_Queue_set_data_affinity( FLASH_Data_aff data_aff );
```

Purpose: Set the style of data affinity for use in SuperMatrix execution. This setting determines that manner in which blocks are assigned and bound to threads (if at all). Three constant values are accepted for `data_aff`:

- `FLASH_QUEUE_AFFINITY_NONE`. Data affinity is disabled altogether, allowing threads to execute tasks regardless of which blocks they update.
- `FLASH_QUEUE_AFFINITY_2D_BLOCK_CYCLIC`. Blocks are assigned and bound to threads in a two-dimensional block cyclic manner.
- `FLASH_QUEUE_AFFINITY_ROUND_ROBIN`. Blocks are assigned and bound to threads in a round-robin manner.

Notes: This feature is different but complimentary to CPU affinity implemented by some operating system schedulers, including the process scheduler present in the Linux kernel as of version 2.6.25. CPU affinity binds processes (and threads) to individual processors, or processor cores. Data affinity binds matrix blocks to individual threads. The idea behind using them together is to improve performance by reducing the need for matrix blocks to be migrate between CPU caches as the tasks are executed.

Caveats: The data affinity mode associated with `FLASH_QUEUE_AFFINITY_ROUND_ROBIN` has not yet been implemented. Therefore, for the time being the user should only use `FLASH_QUEUE_AFFINITY_NONE` and `FLASH_QUEUE_AFFINITY_2D_BLOCK_CYCLIC`.

Arguments:

`data_aff` – A constant value that specifies the kind of data affinity to use during parallel execution.

```
FLASH_Data_aff FLASH_Queue_get_data_affinity( void );
```

Purpose: Query the current status of data affinity in SuperMatrix.

Returns: A constant value: `FLASH_QUEUE_AFFINITY_NONE` if data affinity is disabled; `FLASH_QUEUE_AFFINITY_2D_BLOCK_CYCLIC` if data affinity is set to two-dimensional block cyclic; `FLASH_QUEUE_AFFINITY_ROUND_ROBIN` if data affinity is set to round-robin.

5.5.3 Integration with FLASH front-ends

SuperMatrix is invoked through the same FLASH front-end functions that are documented in Section 5.7.⁷ In order to enable the parallelized implementations, the following conditions must be met:

- Multithreading must be enabled at configure-time. This is accomplished by running configure with the `--enable-multithreading=openmp` or `--enable-multithreading=threads` option, depending on which multithreading implementation is desired.
- SuperMatrix must be enabled at configure-time. This is accomplished by running configure with the `--enable-supermatrix` option.
- SuperMatrix must be enabled at runtime. If SuperMatrix was enabled at configure-time, then it is automatically enabled at runtime by `FLA_Init()` and therefore the user does not need to take any further action. However, SuperMatrix may be disabled at runtime manually through `FLASH_Queue_disable()`, which causes all FLASH-based computational routines to revert to executing sequentially. Subsequently, the user can make the parallelized implementations available again by simply calling the `FLASH_Queue_enable()` routine.

SuperMatrix implementations may be run in an overlapped manner by enclosing the computational invocations with `FLASH_Queue_begin()` and `FLASH_Queue_end()`. Please see Section 4.3 concrete examples of how to use this and other features of SuperMatrix.

5.6 Utility functions

5.6.1 Advanced query functions

```
FLA_Bool FLA_Obj_is_int( FLA_Obj obj );
```

Purpose: Check if an object contains integer values.

Returns: A boolean value: TRUE if the datatype of `obj` is `FLA_INT`; FALSE otherwise.

Arguments:

`obj` – An `FLA_Obj`.

```
FLA_Bool FLA_Obj_is_floating_point( FLA_Obj obj );
```

Purpose: Check if an object contains floating-point (non-integer) numerical values.

Returns: A boolean value: TRUE if the datatype of `obj` is `FLA_FLOAT`, `FLA_DOUBLE`, `FLA_COMPLEX`, or `FLA_DOUBLE_COMPLEX`; FALSE otherwise.

Arguments:

`obj` – An `FLA_Obj`.

```
FLA_Bool FLA_Obj_is_constant( FLA_Obj obj );
```

Purpose: Check if an object is one of the standard `libflame` constants.

Returns: A boolean value: TRUE if the datatype of `obj` is `FLA_CONSTANT`; FALSE otherwise.

Arguments:

`obj` – An `FLA_Obj`.

⁷If a FLASH front-end does not exist for a particular operation, this means that the corresponding SuperMatrix implementation also does not yet exist.

```
FLA_Bool FLA_Obj_is_real( FLA_Obj obj );
```

Purpose: Check if an object contains real numerical values.

Returns: A boolean value: TRUE if the datatype of `obj` is `FLA_FLOAT` or `FLA_DOUBLE`; FALSE otherwise.

Arguments:

`obj` – An `FLA_Obj`.

```
FLA_Bool FLA_Obj_is_complex( FLA_Obj obj );
```

Purpose: Check if an object contains complex numerical values.

Returns: A boolean value: TRUE if the datatype of `obj` is `FLA_COMPLEX` or `FLA_DOUBLE_COMPLEX`; FALSE otherwise.

Arguments:

`obj` – An `FLA_Obj`.

```
FLA_Bool FLA_Obj_is_single_precision( FLA_Obj obj );
```

Purpose: Check if an object uses a single-precision floating-point datatype.

Returns: A boolean value: TRUE if the datatype of `obj` is `FLA_FLOAT` or `FLA_COMPLEX`; FALSE otherwise.

Arguments:

`obj` – An `FLA_Obj`.

```
FLA_Bool FLA_Obj_is_double_precision( FLA_Obj obj );
```

Purpose: Check if an object uses a double-precision floating-point datatype.

Returns: A boolean value: TRUE if the datatype of `obj` is `FLA_DOUBLE` or `FLA_DOUBLE_COMPLEX`; FALSE otherwise.

Arguments:

`obj` – An `FLA_Obj`.

```
FLA_Bool FLA_Obj_is_scalar( FLA_Obj obj );
```

Purpose: Check if an object is 1×1 .

Returns: A boolean value: TRUE if the row and column dimensions of `obj` are equal to 1; FALSE otherwise.

Arguments:

`obj` – An `FLA_Obj`.

```
FLA_Bool FLA_Obj_is_vector( FLA_Obj obj );
```

Purpose: Check if an object is $1 \times n$ or $m \times 1$.

Returns: A boolean value: TRUE if either the row or column dimension of *obj* is equal to 1; FALSE otherwise.

Arguments:

<i>obj</i>	–	An FLA_Obj.
------------	---	-------------

```
FLA_Bool FLA_Obj_has_zero_dim( FLA_Obj obj );
```

Purpose: Check if an object is $0 \times n$ or $m \times 0$.

Returns: A boolean value: TRUE if either the row or column dimension of *obj* is equal to 0; FALSE otherwise.

Arguments:

<i>obj</i>	–	An FLA_Obj.
------------	---	-------------

```
FLA_Bool FLA_Obj_is_conformal_to( FLA_Trans trans, FLA_Obj A, FLA_Obj B );
```

Purpose: Check if *A* and *B* have conformal dimensions. That is, check if the length and width of *A* are equal to the length and width of *B*, respectively. The *trans* argument may be used to perform the check as if *A* were transposed.

Returns: A boolean value: TRUE if the row and column dimensions of *A* are equal to the row and column dimensions of *B*, modulo a possible transposition of *A*; FALSE otherwise.

Arguments:

<i>trans</i>	–	Indicates whether to perform the check as if <i>A</i> were transposed.
<i>A</i>	–	An FLA_Obj.
<i>B</i>	–	An FLA_Obj.

```
FLA_Bool FLA_Obj_is( FLA_Obj A, FLA_Obj B );
```

Purpose: Check if *A* and *B* refer to the same underlying object.

Returns: A boolean value: TRUE if *A* and *B* are the same object; FALSE otherwise.

Dev. notes: This function needs to be reimplemented. Right now, it will return true even if two disjoint views to the same object are passed in.

Arguments:

<i>A</i>	–	An FLA_Obj.
<i>B</i>	–	An FLA_Obj.

```
FLA_Bool FLA_Obj_equals( FLA_Obj A, FLA_Obj B );
```

Purpose: Check if *A* and *B* contain the same numerical values, element-wise.

Returns: A boolean value: TRUE if *A* and *B* are equal; FALSE otherwise.

Arguments:

<i>A</i>	–	An FLA_Obj.
<i>B</i>	–	An FLA_Obj.

5.6.2 Assignment/Update functions

```
void FLA_Obj_set_to_scalar( FLA_Obj alpha, FLA_Obj A );
```

Purpose: Set every element in A to α .

Constraints:

- The numerical datatype of A must be floating-point and must not be FLA_CONSTANT.
- If α is not of datatype FLA_CONSTANT, then it must match the datatype of A .

Arguments:

- | | | |
|-------|---|---|
| alpha | – | An FLA_Obj representing scalar α . |
| A | – | An FLA_Obj representing matrix A . |

```
void FLA_Obj_set_diagonal_to_scalar( FLA_Obj alpha, FLA_Obj A );
```

Purpose: Set all diagonal elements of A to α .

Constraints:

- The numerical datatype of A must be floating-point and must not be FLA_CONSTANT.
- If α is not of datatype FLA_CONSTANT, then it must match the datatype of A .

Arguments:

- | | | |
|-------|---|---|
| alpha | – | An FLA_Obj representing scalar α . |
| A | – | An FLA_Obj representing matrix A . |

```
void FLA_Obj_set_to_identity( FLA_Obj A );
```

Purpose: Set a matrix to be the identity matrix:

$$A := I_n$$

where A is an $n \times n$ general matrix.

Constraints:

- The numerical datatype of A must be floating-point, and must not be FLA_CONSTANT.
- A must be square.

Arguments:

- | | | |
|---|---|--------------------------------------|
| A | – | An FLA_Obj representing matrix A . |
|---|---|--------------------------------------|

```
void FLA_Obj_add_to_diagonal( void *alpha, FLA_Obj A );
```

Purpose: Add α to the diagonal elements of A .

Notes: The datatype of A should match the datatype of the value pointed to by `alpha`.

Constraints:

- The numerical datatype of A must be floating-point and must not be FLA_CONSTANT.
- `alpha` must not be NULL.

Arguments:

- | | | |
|-------|---|--------------------------------------|
| alpha | – | A pointer to a scalar α . |
| A | – | An FLA_Obj representing matrix A . |

```
void FLA_Obj_shift_diagonal( FLA_Obj alpha, FLA_Obj A );
void FLASH_Obj_shift_diagonal( FLA_Obj alpha, FLA_Obj A );
```

Purpose: Add α to the diagonal elements of A .

Constraints:

- The numerical datatype of A must be floating-point and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatype of A if A is real and the precision of A if A is complex.

Arguments:

alpha – An `FLA_Obj` representing scalar α .
A – An `FLA_Obj` representing matrix A .

```
void FLA_Obj_scale_diagonal( FLA_Obj alpha, FLA_Obj A );
```

Purpose: Scale the diagonal of A by α .

Constraints:

- The numerical datatype of A must be floating-point and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatype of A if A is real and the precision of A if A is complex.

Arguments:

alpha – An `FLA_Obj` representing scalar α .
A – An `FLA_Obj` representing matrix A .

5.6.3 Math-related functions

```
void FLA_Absolute_square( FLA_Obj alpha );
```

Purpose: Compute the absolute square (or squared norm) of a complex scalar:

$$\alpha := |\alpha|^2$$

where α is a complex scalar and $|\alpha|^2$ is defined as

$$|\alpha|^2 = \alpha \bar{\alpha}$$

Notes: If α is real, then the operation reduces to

$$\alpha := \alpha^2$$

Constraints:

- The numerical datatype of α must be floating-point and must not be `FLA_CONSTANT`.

Arguments:

alpha – An `FLA_Obj` representing scalar α .

```
void FLA_Conjugate( FLA_Obj A );
```

Purpose: Conjugate a matrix:

$$A := \bar{A}$$

where A is a general matrix.

Notes: If A is real, then the function has no effect.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `*scal()`.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.

Arguments:

A – An `FLA_Obj` representing matrix A .

```
void FLA_Conjugate_r( FLA_Uplo uplo, FLA_Obj A );
```

Purpose: Conjugate the lower or upper triangular portion of a matrix A .

Notes: If A is real, then the function has no effect.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `*scal()`.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.

Arguments:

`uplo` – Indicates whether the lower or upper triangle of A is referenced during the operation.

A – An `FLA_Obj` representing matrix A .

```
void FLA_Transpose( FLA_Obj A );
```

Purpose: Transpose a matrix:

$$A := A^T$$

where A is a general matrix.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `?swap()`.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.
- A must be square.

Arguments:

A – An `FLA_Obj` representing matrix A .

```
void FLA_Invert( FLA_Obj alpha );
```

Purpose: Invert a scalar:

$$\alpha := \alpha^{-1}$$

where α is a scalar.

Constraints:

- The numerical datatype of α must be floating-point and must not be FLA_CONSTANT.

Arguments:

alpha – An FLA_Obj representing scalar α .

```
void FLA_Max_abs_value( FLA_Obj A, FLA_Obj amax );
```

Purpose: Find the maximum absolute value of all elements of a matrix:

$$A_{max} := \max_{ij} |\alpha_{ij}|$$

where A_{max} is a scalar and α_{ij} is the (i, j) element of general matrix A . Upon completion, the maximum absolute value A_{max} is stored to **amax**.

Notes: If A is complex, then $|\alpha_{ij}|$ is evaluated as the complex norm, which, for any complex number z , is defined as

$$\begin{aligned} |z| &= |x + iy| \\ &= \sqrt{x^2 + y^2} \end{aligned}$$

where x and y are the real and imaginary components, respectively, of z .

Constraints:

- The numerical datatype of A must be floating-point and must not be FLA_CONSTANT.
- The numerical datatype of A_{max} must be real and must not be FLA_CONSTANT.
- The precision of the datatype of A_{max} must be equal to that of A .

Arguments:

A – An FLA_Obj representing matrix A .
 amax – An FLA_Obj representing scalar A_{max} .

```
double FLA_Max_elemwise_diff( FLA_Obj A, FLA_Obj B );
double FLASH_Max_elemwise_diff( FLA_Obj A, FLA_Obj B );
```

Purpose: Find and return the maximum element-wise absolute difference between two matrices,

$$\max_{i,j} |\alpha_{ij} - \beta_{ij}|$$

where α_{ij} and β_{ij} are the (i, j) elements of matrices A and B , respectively.

Notes: If A and B are complex, then they are treated as real matrices for the purposes of computing the maximum absolute difference. That is, the real and imaginary components of A_{ij} are compared with the real and imaginary components of B_{ij} , respectively.

Returns: A positive double-precision floating-point value.

Constraints:

- The numerical datatypes of A and B must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The dimensions of A and B must be conformal.

Arguments:

A	–	An <code>FLA_Obj</code> representing matrix A .
B	–	An <code>FLA_Obj</code> representing matrix B .

```
void FLA_Mult_add( FLA_Obj alpha, FLA_Obj beta, FLA_Obj gamma );
```

Purpose: Multiply two scalars and add the result to a third scalar:

$$\gamma := \gamma + \alpha\beta$$

where α , β , and γ are scalars.

Constraints:

- The numerical datatype of α , β , and γ must be floating-point. Also, the datatype of γ must not be `FLA_CONSTANT`.

Arguments:

alpha	–	An <code>FLA_Obj</code> representing scalar α .
beta	–	An <code>FLA_Obj</code> representing scalar β .
gamma	–	An <code>FLA_Obj</code> representing scalar γ .

```
void FLA_Negate( FLA_Obj A );
```

Purpose: Negate a matrix:

$$A := -A$$

where A is a general matrix.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `*scal()`.

Constraints:

- The numerical datatype of A must be floating-point and must not be `FLA_CONSTANT`.

Arguments:

A	–	An <code>FLA_Obj</code> representing matrix A .
---	---	---

```
void FLA_Norm1( FLA_Obj A, FLA_Obj norm1 );
void FLASH_Norm1( FLA_Obj A, FLA_Obj norm1 );
```

Purpose: Compute the maximum absolute column sum norm of a matrix:

$$\|A\|_1 := \max_j \sum_{i=0}^{n-1} |\alpha_{ij}|$$

where $\|A\|_1$ is a scalar and α_{ij} is the (i, j) element of general matrix A . Upon completion, the maximum absolute column sum norm $\|A\|_1$ is stored to **norm1**.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine ***asum()**.

Constraints:

- The numerical datatype of A must be floating-point and must not be **FLA_CONSTANT**.
- The numerical datatype of **norm1** must be real and must not be **FLA_CONSTANT**.
- The precision of the datatype of **norm1** must be equal to that of A .

Arguments:

- | | | |
|--------------|---|---|
| A | – | An FLA_Obj representing matrix A . |
| norm1 | – | An FLA_Obj representing scalar $\ A\ _1$. |

```
void FLA_Norm_inf( FLA_Obj A, FLA_Obj norminf );
```

Purpose: Compute the maximum absolute row sum norm of a matrix:

$$\|A\|_\infty := \max_i \sum_{j=0}^{n-1} |\alpha_{ij}|$$

where $\|A\|_\infty$ is a scalar and α_{ij} is the (i, j) element of general matrix A . Upon completion, the maximum absolute row sum norm $\|A\|_\infty$ is stored to **norminf**.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine ***asum()**.

Constraints:

- The numerical datatype of A must be floating-point and must not be **FLA_CONSTANT**.
- The numerical datatype of **norminf** must be real and must not be **FLA_CONSTANT**.
- The precision of the datatype of **norminf** must be equal to that of A .

Arguments:

- | | | |
|----------------|---|--|
| A | – | An FLA_Obj representing matrix A . |
| norminf | – | An FLA_Obj representing scalar $\ A\ _\infty$. |

```
FLA_Error FLA_Sqrt( FLA_Obj alpha );
```

Purpose: Compute the square root of a scalar:

$$\alpha := \sqrt{\alpha}$$

where α is a positive real scalar.

Constraints:

- The numerical datatype of α must be real and must not be FLA_CONSTANT.

Returns: FLA_SUCCESS if α is non-negative on entry; otherwise FLA_FAILURE.

Arguments:

alpha – An FLA_Obj representing scalar α .

```
void FLA_Random_matrix( FLA_Obj A );
void FLASH_Random_matrix( FLA_Obj A );
```

Purpose: Overwrite a matrix A with a random matrix.

Notes: If A is complex, then elements are set by assigning separate random values to real and imaginary components.

Imp. Notes: The random numbers obtained are unseeded and therefore deterministic. Random numbers are obtained with the system function `drand48()` and then scaled and shifted to result in a uniform distribution over the interval $[-1.0, 1.0)$.

Constraints:

- The numerical datatype of A must be floating-point, and must not be FLA_CONSTANT.

Arguments:

A – An FLA_Obj representing matrix A .

```
void FLA_Random_herm_matrix( FLA_Uplo uplo, FLA_Obj A );
```

Purpose: Overwrite a matrix A with a random Hermitian matrix, ie: a matrix A such that

$$A = A^H$$

The `uplo` argument indicates whether the lower or upper triangle of A is initially stored by the operation.

Notes: If A is real, then the operation results in a random symmetric matrix. If A is complex, then elements are set by assigning separate random values to real and imaginary components.

Imp. Notes: The random numbers obtained are unseeded and therefore deterministic. Random numbers are obtained with the system function `drand48()` and then scaled and shifted to result in a uniform distribution over the interval $[-1.0, 1.0)$. Currently, the value of `uplo` has no net effect, as in both cases the specified triangle is randomized and then conjugate-transposed into the other. However, a future implementation of `FLA_Random_herm_matrix()` may only store to the triangle specified by `uplo`.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.

Arguments:

- | | |
|-------------------|--|
| <code>uplo</code> | – Indicates whether the lower or upper triangle of A is stored during the operation. This argument has no net effect on the operation. |
| <code>A</code> | – An <code>FLA_Obj</code> representing matrix A . |


```
void FLA_Random_spd_matrix( FLA_Uplo uplo, FLA_Obj A );
void FLASH_Random_spd_matrix( FLA_Uplo uplo, FLA_Obj A );
```

Purpose: Overwrite a matrix A with a random symmetric positive definite matrix. The `uplo` argument indicates whether the lower or upper triangle of A is stored by the operation.

Notes: If A is real, then the matrix is randomized by performing a symmetric rank- k update (with $\beta = 0$ and $\alpha = 1$):

$$A := RR^T$$

where R is a random real matrix conformal to A . If A is complex, then the matrix is randomized by performing a hermitian rank- k update (with $\beta = 0$ and $\alpha = 1$):

$$A := RR^H$$

where R is a random complex matrix conformal to A . In either case, the random matrix R is obtained via `FLA_Random_matrix()`.

Imp. Notes: The random numbers obtained are unseeded and therefore deterministic. Random numbers are obtained with the system function `drand48()` and then scaled and shifted to result in a uniform distribution over the interval $[-1.0, 1.0)$. Currently, the value of `uplo` has no net effect, as in both cases the specified triangle is randomized and then transposed into the other. However, a future implementation of `FLA_Random_spd_matrix()` may only store to the triangle specified by `uplo`.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.

Arguments:

- | | |
|-------------------|--|
| <code>uplo</code> | – Indicates whether the lower or upper triangle of A is stored during the operation. This argument is currently ignored. |
| <code>A</code> | – An <code>FLA_Obj</code> representing matrix A . |

```
void FLA_Random_tri_matrix( FLA_Uplo uplo, FLA_Diag diag, FLA_Obj A );
```

Purpose: Overwrite a matrix A with a random triangular matrix. The `uplo` argument indicates whether A will be lower or upper triangular. (The triangle opposite of that specified by `uplo` is set to zero.) The `diag` argument indicates how the diagonal of the matrix is set; `FLA_ZERO_DIAG` will set all diagonal entries to zero, `FLA_UNIT_DIAG` will set diagonal entries to one, and `FLA_NONUNIT_DIAG` will assign the diagonal random values.

Notes: If A is complex, then elements in the `uplo` triangle are set by assigning separate random values to real and imaginary components.

Imp. Notes: The random numbers obtained are unseeded and therefore deterministic. Random numbers are obtained with the system function `drand48()` and then scaled and shifted to result in a uniform distribution over the interval $[-1.0, 1.0)$.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.

Arguments:

- | | |
|-------------------|--|
| <code>uplo</code> | – Indicates whether the lower or upper triangle of A is stored during the operation. This argument is currently ignored. |
| <code>diag</code> | – Indicates whether the diagonal of A is set to be zero, unit, or non-unit (random). |
| <code>A</code> | – An <code>FLA_Obj</code> representing matrix A . |

```
void FLA_Symmetrize( FLA_Uplo uplo, FLA_Obj A );
```

Purpose: Transform a general matrix A into a symmetric matrix by copying the transpose of one triangle into the other triangle. The `uplo` argument indicates which triangle of A is preserved and copied.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `?copy()`.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.
- A must be square.

Arguments:

<code>uplo</code>	–	Indicates whether the lower or upper triangle of A is preserved and transposed into the other triangle.
<code>A</code>	–	An <code>FLA_Obj</code> representing matrix A .

```
void FLA_Hermitianize( FLA_Uplo uplo, FLA_Obj A );
```

Purpose: Transform a general complex matrix A into a Hermitian matrix by copying the conjugate-transpose of one triangle into the other triangle and then zeroing the imaginary components of the diagonal entries. The `uplo` argument indicates which triangle of A is preserved.

Notes: If A is real, then `FLA_Hermitianize()` behaves exactly as `FLA_Symmetrize()`.

Imp. Notes: This function uses external implementations of the level-1 BLAS routines `?copy()` and `*scal()`.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.
- A must be square.

Arguments:

<code>uplo</code>	–	Indicates whether the lower or upper triangle of A is preserved and conjugate-transposed into the other triangle.
<code>A</code>	–	An <code>FLA_Obj</code> representing matrix A .

```
void FLA_Triangularize( FLA_Uplo uplo, FLA_Diag diag, FLA_Obj A );
```

Purpose: Transform a general matrix A into a triangular matrix by preserving one triangle and zeroing the other triangle. The `uplo` argument indicates which triangle of A is preserved. The `diag` argument indicates whether to change the diagonal of the matrix; `FLA_ZERO_DIAG` will set all diagonal entries to zero, `FLA_UNIT_DIAG` will set diagonal entries to one, and `FLA_NONUNIT_DIAG` will leave the diagonal unchanged.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.
- A must be square.

Arguments:

<code>uplo</code>	–	Indicates whether the lower or upper triangle of A is preserved.
<code>diag</code>	–	Indicates whether the diagonal of A is set to be zero, unit, or left unchanged.
<code>A</code>	–	An <code>FLA_Obj</code> representing matrix A .

5.6.4 Miscellaneous functions

```
unsigned int FLA_Check_error_level( void );
```

Purpose: Query the current level of internal error and parameter checking in `libflame`. Valid return values are `FLA_FULL_ERROR_CHECKING`, `FLA_MIN_ERROR_CHECKING`, and `FLA_NO_ERROR_CHECKING`.

Notes: Error and parameter checking will have a small but sometimes noticeable impact on performance. We recommend full error checking for all users except those who are performing benchmarks who have already tested their code with error checking fully enabled.

Returns: An unsigned integer: `FLA_FULL_ERROR_CHECKING` if error and parameter checking is fully enabled; `FLA_MIN_ERROR_CHECKING` if minimal error and parameter checking is enabled; `FLA_NO_ERROR_CHECKING` if error and parameter checking is completely disabled.

```
unsigned int FLA_Check_error_level_set( unsigned int level );
```

Purpose: Set the level of internal error and parameter checking in `libflame` to `level`. Valid values for `level` are `FLA_FULL_ERROR_CHECKING`, `FLA_MIN_ERROR_CHECKING`, and `FLA_NO_ERROR_CHECKING`. The function returns the *previous* level of error checking regardless of whether the new value actually caused a change in the level.

Returns: An unsigned integer: `FLA_FULL_ERROR_CHECKING` if error and parameter checking was fully enabled; `FLA_MIN_ERROR_CHECKING` if minimal error and parameter checking was enabled; `FLA_NO_ERROR_CHECKING` if error and parameter checking was completely disabled.

Arguments:

<code>level</code>	–	The value corresponding to the desired error checking level.
--------------------	---	--

```
void FLA_Print_message( char* message, char* filename, unsigned int line );
```

Purpose: Print a message to standard output. The function interface assumes that the user will also want to print out the name of the file and the line number on which the `FLA_Print_message()` invocation appears.

Dev. notes: This function is most often used internally when outputting error messages just before the library aborts. However, it is general enough to be used by application programmers as well.

Arguments:

<code>message</code>	–	A pointer to a string containing the message to output.
<code>filename</code>	–	A pointer to a string containing the name of the file. This is typically obtained via the C preprocessor macro <code>__FILE__</code> .
<code>line</code>	–	An unsigned integer containing the line number that contained the invocation of <code>FLA_Print_message()</code> . This is typically obtained via the C preprocessor macro <code>__LINE__</code> .

```
void FLA_Abort( void );
```

Purpose: Abort execution of the application and output a corresponding message to standard error.

Imp. Notes: This function currently is implemented with the standard C library function `abort()`, which is often implemented by raising a `SIGABRT` signal. This usually allows the user to quickly perform a backtrace of the function stack in a debugger without setting break-points.

```
double FLA_Clock( void );
```

Purpose: Return a value representing the amount of time, in seconds, that has elapsed since an implementation-defined Epoch. The difference in successive return values may be used to determine elapsed wall clock time.

Returns: A double-precision floating-point value.

Imp. Notes: When possible, this routine uses architecture-specific code in order to achieve the highest possible precision. If one of the common architectures is not detected, then the implementation uses `gettimeofday()`, which provides microsecond accuracy. The user may force the use of this more portable `gettimeofday()` timer function at configure-time with the configure option `--enable-portable-timer`. For Microsoft Windows builds (ie: when `FLA_ENABLE_WINDOWS_BUILD` is defined) `FLA_Clock()` is implemented in terms of `QueryPerformanceCounter()` and `QueryPerformanceFrequency()`.

5.7 Front-ends

This section documents the interfaces to the featured computational routines provided by `libflame`. We refer to these interfaces as *front-ends*, because they form the primary set of APIs for use by users at the application-level. None of these routines are direct wrappers to external implementations. All computational front-ends employ FLAME algorithmic variants in some capacity, either to produce a blocked algorithm or an algorithm-by-blocks, the latter of which uses hierarchical storage and may be executed either sequentially or in parallel. For more information on the mechanisms behind hierarchical storage and parallel execution, please see Sections 5.4 and 5.5, respectively.

5.7.1 BLAS operations

5.7.1.1 Level-1 BLAS

```
void FLA_Asum( FLA_Obj x, FLA_Obj norm1 );
```

Purpose: Compute the 1-norm of a vector:

$$\|x\|_1 := \sum_{i=0}^{n-1} |\chi_i|$$

where $\|x\|_1$ is a scalar and χ_i is the i th element of general vector x of length n . Upon completion, the 1-norm $\|x\|_1$ is stored to **norm1**.

Imp. Notes: This function is implemented as a wrapper to `FLA_Asum_external()`.

Constraints:

- The numerical datatype of x must be floating-point and must not be `FLA_CONSTANT`.
- The numerical datatype of **norm1** must be real and must not be `FLA_CONSTANT`.
- The precision of the datatype of **norm1** must be equal to that of x .

Arguments:

- | | | |
|--------------|---|---|
| x | – | An <code>FLA_Obj</code> representing vector x . |
| norm1 | – | An <code>FLA_Obj</code> representing scalar $\ x\ _1$. |

```
void FLA_Axpy( FLA_Obj alpha, FLA_Obj X, FLA_Obj Y );
void FLASH_Axpy( FLA_Obj alpha, FLA_Obj X, FLA_Obj Y );
```

Purpose: Perform an AXPY operation:

$$Y := Y + \alpha X$$

where α is a scalar, and X and Y are general matrices.

Int. Notes: `FLA_Axpy()` expects X and Y to be a flat matrix objects.

Imp. Notes: `FLA_Axpy()` simply invokes the external BLAS wrapper `FLA_Axpy_external()`. `FLASH_Axpy()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the AXPY operation into subproblems expressed in terms of individual blocks of X and Y and then invokes `FLA_Axpy_external()` to perform the computation on these blocks.

Constraints:

- The numerical datatypes of X and Y must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatypes of X and Y .
- The dimensions of X and Y must be conformal.

Arguments:

- | | | |
|--------------|---|--|
| alpha | – | An <code>FLA_Obj</code> representing scalar α . |
| X | – | An <code>FLA_Obj</code> representing matrix X . |
| Y | – | An <code>FLA_Obj</code> representing matrix Y . |

```
void FLA_Axpyt( FLA_Trans trans, FLA_Obj alpha, FLA_Obj X, FLA_Obj Y );
```

Purpose: Perform one of the following extended AXPY operations:

$$\begin{aligned} Y &:= Y + \alpha X \\ Y &:= Y + \alpha X^T \\ Y &:= Y + \alpha \bar{X} \\ Y &:= Y + \alpha X^H \end{aligned}$$

where α is a scalar, and X and Y are general matrices. The **trans** argument allows the computation to proceed as if X were conjugated and/or transposed.

Notes: If X and Y are vectors, **FLA_Axpyt()** will implicitly and automatically perform the transposition necessary to achieve conformal dimensions regardless of the value of **trans**:

$$\begin{aligned} y_r &:= y_r + \alpha x_c^T \\ y_c &:= y_c + \alpha x_r^T \end{aligned}$$

where x_c and y_c are column vectors and x_r and y_r are row vectors. In these special cases where X and Y are vectors, **trans** argument values of **FLA_TRANSPOSE** and **FLA_CONJ_TRANSPOSE** will effectively be interpreted as **FLA_NO_TRANSPOSE** and **FLA_CONJ_NO_TRANSPOSE**, respectively, and thus the **trans** argument may still be used to request conjugation of vector X .

Imp. Notes: This function is implemented as a wrapper to **FLA_Axpyt_external()**.

Constraints:

- The numerical datatypes of X and Y must be identical and floating-point, and must not be **FLA_CONSTANT**.
- If α is not of datatype **FLA_CONSTANT**, then it must match the datatypes of X and Y .
- If X and Y are vectors, then their lengths must be equal. Otherwise, if **trans** equals **FLA_NO_TRANSPOSE** or **FLA_CONJ_NO_TRANSPOSE**, then the dimensions of X and Y must be conformal; otherwise, if **trans** equals **FLA_TRANSPOSE** or **FLA_CONJ_TRANSPOSE**, then the dimensions of X^T and Y must be conformal.

Arguments:

- | | |
|--------------|---|
| trans | – Indicates whether the operation proceeds as if X were conjugated and/or transposed. |
| alpha | – An FLA_Obj representing scalar α . |
| X | – An FLA_Obj representing matrix X . |
| Y | – An FLA_Obj representing matrix Y . |

```
void FLA_Axpys( FLA_Obj alpha0, FLA_Obj alpha1, FLA_Obj X, FLA_Obj beta, FLA_Obj Y );
```

Purpose: Perform the following extended AXPY operation:

$$Y := \beta Y + \alpha_0 \alpha_1 X$$

where α_0 , α_1 and β are scalars, and X and Y are general matrices.

Notes: If X and Y are vectors, `FLA_Axpys()` will implicitly and automatically perform the transposition necessary to achieve conformal dimensions:

$$\begin{aligned} y_r &:= \beta y_r + \alpha_0 \alpha_1 x_c^T \\ y_c &:= \beta y_c + \alpha_0 \alpha_1 x_r^T \end{aligned}$$

where x_c and y_c are column vectors and x_r and y_r are row vectors.

Imp. Notes: This function is implemented as a wrapper to `FLA_Axpys_external()`.

Constraints:

- The numerical datatypes of X and Y must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α_0 , α_1 , and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of X and Y .

Arguments:

- | | | |
|---------------------|---|--|
| <code>alpha0</code> | – | An <code>FLA_Obj</code> representing scalar α_0 . |
| <code>alpha1</code> | – | An <code>FLA_Obj</code> representing scalar α_1 . |
| <code>X</code> | – | An <code>FLA_Obj</code> representing matrix X . |
| <code>beta</code> | – | An <code>FLA_Obj</code> representing scalar β . |
| <code>Y</code> | – | An <code>FLA_Obj</code> representing matrix Y . |

```
void FLA_Copy( FLA_Obj A, FLA_Obj B );
void FLASH_Copy( FLA_Obj A, FLA_Obj B );
```

Purpose: Copy the numerical contents of matrix A to matrix B .

Int. Notes: `FLA_Copy()` expects A and B to be a flat matrix objects.

Imp. Notes: `FLA_Copy()` simply invokes the external BLAS wrapper `FLA_Copy_external()`. `FLASH_Copy()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the COPY operation into subproblems expressed in terms of individual blocks of A and B and then invokes `FLA_Copy_external()` to perform the computation on these blocks.

Constraints:

- The numerical datatypes of A and B must be identical and must not be `FLA_CONSTANT`.
- The dimensions of A and B must be conformal.

Arguments:

- | | | |
|----------------|---|---|
| <code>A</code> | – | An <code>FLA_Obj</code> representing matrix A . |
| <code>B</code> | – | An <code>FLA_Obj</code> representing matrix B . |

```
void FLA_Copy( FLA_Uplo uplo, FLA_Obj A, FLA_Obj B );
```

Purpose: Perform an extended copy operation on the lower or upper triangles of matrices A and B :

$$B := A$$

where A and B are general square matrices. The `uplo` argument indicates whether the lower or upper triangles of A and B are referenced and updated by the operation.

Imp. Notes: This function is implemented as a wrapper to `FLA_Copy_external()`.

Constraints:

- The numerical datatypes of A and B must be identical, and must not be `FLA_CONSTANT`.
- The dimensions of A and B must be conformal.
- A and B must be square.

Arguments:

- | | |
|-------------------|--|
| <code>uplo</code> | – Indicates whether the lower or upper triangles of A and B are referenced and updated during the operation. |
| <code>A</code> | – An <code>FLA_Obj</code> representing matrix A . |
| <code>B</code> | – An <code>FLA_Obj</code> representing matrix B . |


```
void FLA_Copyt( FLA_Trans trans, FLA_Obj A, FLA_Obj B );
```

Purpose: Copy the numerical contents of A to B with one of the following extended operations:

$$\begin{aligned} B &:= A \\ B &:= A^T \\ B &:= \bar{A} \\ B &:= A^H \end{aligned}$$

where A and B are general matrices. The **trans** argument allows the computation to proceed as if A were conjugated and/or transposed.

Notes: If A and B are vectors, **FLA_Copyt()** will implicitly and automatically perform the transposition necessary to achieve conformal dimensions regardless of the value of **trans**:

$$\begin{aligned} b_r &:= a_c^T \\ b_c &:= a_r^T \end{aligned}$$

where a_c and b_c are column vectors and a_r and b_r are row vectors. In these special cases where A and B are vectors, **trans** argument values of **FLA_TRANSPOSE** and **FLA_CONJ_TRANSPOSE** will effectively be interpreted as **FLA_NO_TRANSPOSE** and **FLA_CONJ_NO_TRANSPOSE**, respectively, and thus the **trans** argument may still be used to request conjugation of vector A .

Imp. Notes: This function is implemented as a wrapper to **FLA_Copyt_external()**.

Constraints:

- The numerical datatypes of A and B must be identical, and must not be **FLA_CONSTANT**.
- If A and B are vectors, then their lengths must be equal. Otherwise, if **trans** equals **FLA_NO_TRANSPOSE** or **FLA_CONJ_NO_TRANSPOSE**, then the dimensions of A and B must be conformal; otherwise, if **trans** equals **FLA_TRANSPOSE** or **FLA_CONJ_TRANSPOSE**, then the dimensions of A^T and B must be conformal.

Arguments:

- | | | |
|--------------|---|---|
| trans | – | Indicates whether the operation proceeds as if A were conjugated and/or transposed. |
| A | – | An FLA_Obj representing matrix A . |
| B | – | An FLA_Obj representing matrix B . |

```
void FLA_Dot( FLA_Obj x, FLA_Obj y, FLA_Obj rho );
```

Purpose: Perform a dot (inner) product operation between two vectors:

$$\rho := \sum_{i=0}^{n-1} \chi_i \psi_i$$

where ρ is a scalar, and χ_i and ψ_i are the i th elements of general vectors x and y , respectively, where both vectors are of length n . Upon completion, the dot product ρ is stored to **rho**.

Imp. Notes: This function is implemented as a wrapper to `FLA_Dot_external()`.

Constraints:

- The numerical datatypes of x , y , and ρ must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The lengths of vectors x and y must be equal.

Arguments:

- | | | |
|------------|---|--|
| x | – | An <code>FLA_Obj</code> representing vector x . |
| y | – | An <code>FLA_Obj</code> representing vector y . |
| rho | – | An <code>FLA_Obj</code> representing scalar ρ . |

```
void FLA_Dotc( FLA_Conj conj, FLA_Obj x, FLA_Obj y, FLA_Obj rho );
```

Purpose: Perform one of the following extended dot product operations:

$$\rho := \sum_{i=0}^{n-1} \chi_i \psi_i$$

$$\rho := \sum_{i=0}^{n-1} \bar{\chi}_i \bar{\psi}_i$$

where ρ is a scalar, and χ_i and ψ_i are the i th elements of general vectors x and y , respectively, where both vectors are of length n . Upon completion, the dot product ρ is stored to **rho**. The **conj** argument allows the computation to proceed as if x and y were conjugated.

Notes: If x , y , and ρ are real, the value of **conj** is ignored and `FLA_Dotc()` behaves exactly as `FLA_Dot()`.

Imp. Notes: This function is implemented as a wrapper to `FLA_Dotc_external()`.

Constraints:

- The numerical datatypes of x , y , and ρ must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The lengths of vectors x and y must be equal.

Arguments:

- | | | |
|-------------|---|--|
| conj | – | Indicates whether to conjugate the intermediate element-wise terms of the dot product. |
| x | – | An <code>FLA_Obj</code> representing vector x . |
| y | – | An <code>FLA_Obj</code> representing vector y . |
| rho | – | An <code>FLA_Obj</code> representing scalar ρ . |

```
void FLA_Dots( FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj beta, FLA_Obj rho );
```

Purpose: Perform the following extended dot product operation between two vectors:

$$\rho := \beta\rho + \alpha \sum_{i=0}^{n-1} \chi_i \psi_i$$

where α , β , and ρ are scalars, and χ_i and ψ_i are the i th elements of general vectors x and y , respectively, where both vectors are of length n . Upon completion, the dot product ρ is stored to **rho**.

Imp. Notes: This function is implemented as a wrapper to `FLA_Dots_external()`.

Constraints:

- The numerical datatypes of x , y , and ρ must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of x , y , and ρ .
- The lengths of vectors x and y must be equal.

Arguments:

alpha	–	An <code>FLA_Obj</code> representing scalar α .
x	–	An <code>FLA_Obj</code> representing vector x .
y	–	An <code>FLA_Obj</code> representing vector y .
beta	–	An <code>FLA_Obj</code> representing scalar β .
rho	–	An <code>FLA_Obj</code> representing scalar ρ .

```
void FLA_Dotcs( FLA_Conj conj, FLA_Obj alpha, FLA_Obj x, FLA_Obj y,
               FLA_Obj beta, FLA_Obj rho );
```

Purpose: Perform one of the following extended dot product operations between two vectors:

$$\rho := \beta\rho + \alpha \sum_{i=0}^{n-1} \chi_i \psi_i$$

$$\rho := \beta\rho + \alpha \sum_{i=0}^{n-1} \bar{\chi}_i \bar{\psi}_i$$

where α , β , and ρ are scalars, and χ_i and ψ_i are the i th elements of general vectors x and y , respectively, where both vectors are of length n . Upon completion, the dot product ρ is stored to `rho`. The `conj` argument allows the computation to proceed as if x and y were conjugated.

Notes: If x , y , and ρ are real, the value of `conj` is ignored and `FLA_Dotcs()` behaves exactly as `FLA_Dots()`.

Imp. Notes: This function is implemented as a wrapper to `FLA_Dotcs_external()`.

Constraints:

- The numerical datatypes of x , y , and ρ must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of x , y , and ρ .
- The lengths of vectors x and y must be equal.

Arguments:

<code>conj</code>	–	Indicates whether the operation proceeds as if x and y were conjugated.
<code>alpha</code>	–	An <code>FLA_Obj</code> representing scalar α .
<code>x</code>	–	An <code>FLA_Obj</code> representing vector x .
<code>y</code>	–	An <code>FLA_Obj</code> representing vector y .
<code>beta</code>	–	An <code>FLA_Obj</code> representing scalar β .
<code>rho</code>	–	An <code>FLA_Obj</code> representing scalar ρ .

```
void FLA_Dot2s( FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj beta, FLA_Obj rho );
```

Purpose: Perform the following extended dot product operation between two vectors:

$$\rho := \beta\rho + \alpha \sum_{i=0}^{n-1} \chi_i \psi_i + \bar{\alpha} \sum_{i=0}^{n-1} \chi_i \psi_i$$

where α , β , and ρ are scalars, and χ_i and ψ_i are the i th elements of general vectors x and y , respectively, where both vectors are of length n . Upon completion, the dot product ρ is stored to **rho**.

Notes: Though this operation may be reduced to:

$$\rho := \beta\rho + (\alpha + \bar{\alpha}) \sum_{i=0}^{n-1} \chi_i \psi_i$$

it is expressed above in unreduced form to allow a more clear contrast to `FLA_Dot2cs()`.

Imp. Notes: This function is implemented as a wrapper to `FLA_Dot2s_external()`.

Constraints:

- The numerical datatypes of x , y , and ρ must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of x , y , and ρ .
- The lengths of vectors x and y must be equal.

Arguments:

alpha	–	An <code>FLA_Obj</code> representing scalar α .
x	–	An <code>FLA_Obj</code> representing vector x .
y	–	An <code>FLA_Obj</code> representing vector y .
beta	–	An <code>FLA_Obj</code> representing scalar β .
rho	–	An <code>FLA_Obj</code> representing scalar ρ .

```
void FLA_Dot2cs( FLA_Conj conj, FLA_Obj alpha, FLA_Obj x, FLA_Obj y,
                FLA_Obj beta, FLA_Obj rho );
```

Purpose: Perform one of the following extended dot product operations between two vectors:

$$\rho := \beta\rho + \alpha \sum_{i=0}^{n-1} \chi_i \psi_i + \bar{\alpha} \sum_{i=0}^{n-1} \chi_i \bar{\psi}_i$$

$$\rho := \beta\rho + \alpha \sum_{i=0}^{n-1} \chi_i \bar{\psi}_i + \bar{\alpha} \sum_{i=0}^{n-1} \bar{\chi}_i \psi_i$$

where α , β , and ρ are scalars, and χ_i and ψ_i are the i th elements of general vectors x and y , respectively, where both vectors are of length n . Upon completion, the dot product ρ is stored to **rho**. The **conj** argument allows the computation to proceed as if x and y were conjugated.

Notes: If x , y , and ρ are real, the value of **conj** is ignored and `FLA_Dot2cs()` behaves exactly as `FLA_Dot2s()`.

Imp. Notes: This function is implemented as a wrapper to `FLA_Dot2cs_external()`.

Constraints:

- The numerical datatypes of x , y , and ρ must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of x , y , and ρ .
- The lengths of vectors x and y must be equal.

Arguments:

conj	–	Indicates whether the operation proceeds as if x and y were conjugated.
alpha	–	An <code>FLA_Obj</code> representing scalar α .
x	–	An <code>FLA_Obj</code> representing vector x .
y	–	An <code>FLA_Obj</code> representing vector y .
beta	–	An <code>FLA_Obj</code> representing scalar β .
rho	–	An <code>FLA_Obj</code> representing scalar ρ .

```
void FLA_Iamax( FLA_Obj x, FLA_Obj i );
```

Purpose: Find the index i of the element of x which has the maximum absolute value, where x is a general vector and i is a scalar. If the maximum absolute value is shared by more than one element, then the element whose index is highest is chosen.

Imp. Notes: This function is implemented as a wrapper to `FLA_Iamax_external()`.

Constraints:

- The numerical datatype of x must be floating-point, and must not be `FLA_CONSTANT`.
- The numerical datatype of i must be integer, and must not be `FLA_CONSTANT`.

Arguments:

x	–	An <code>FLA_Obj</code> representing vector x .
i	–	An <code>FLA_Obj</code> representing scalar i .

```
void FLA_Inv_scal( FLA_Obj alpha, FLA_Obj A );
```

Purpose: Perform an inverse scaling operation:

$$A := \alpha^{-1}A$$

where α is a scalar and A is a general matrix.

Imp. Notes: This function is implemented as a wrapper to `FLA_Inv_scal_external()`.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatype of A if A is real and the precision of A if A is complex.
- α may not be equal to zero.

Arguments:

- | | | |
|--------------------|---|--|
| <code>alpha</code> | – | An <code>FLA_Obj</code> representing scalar α . |
| <code>A</code> | – | An <code>FLA_Obj</code> representing matrix A . |

```
void FLA_Inv_scalc( FLA_Conj conjalpha, FLA_Obj alpha, FLA_Obj A );
```

Purpose: Perform one of the following extended inverse scaling operations:

$$A := \alpha^{-1}A$$

$$A := \bar{\alpha}^{-1}A$$

where α is a scalar and A is a general matrix. The `conjalpha` argument allows the computation to proceed as if α were conjugated.

Notes: If α is real, the value of `conjalpha` is ignored and `FLA_Inv_scalc()` behaves exactly as `FLA_Inv_scal()`.

Imp. Notes: This function is implemented as a wrapper to `FLA_Inv_scalc_external()`.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatype of A if A is real and the precision of A if A is complex.
- α may not be equal to zero.

Arguments:

- | | | |
|------------------------|---|--|
| <code>conjalpha</code> | – | Indicates whether the operation proceeds as if α were conjugated. |
| <code>alpha</code> | – | An <code>FLA_Obj</code> representing scalar α . |
| <code>A</code> | – | An <code>FLA_Obj</code> representing matrix A . |

```
void FLA_Nrm2( FLA_Obj x, FLA_Obj norm );
```

Purpose: Compute the 2-norm of a vector:

$$\|x\|_2 := \left(\sum_{i=0}^{n-1} |\chi_i|^2 \right)^{\frac{1}{2}}$$

where $\|x\|_2$ is a scalar and χ_i is the i th element of general vector x of length n . Upon completion, the 2-norm $\|x\|_2$ is stored to **norm**.

Imp. Notes: This function is implemented as a wrapper to `FLA_Nrm2_external()`.

Constraints:

- The numerical datatype of x must be floating-point and must not be `FLA_CONSTANT`.
- The numerical datatype of **norm** must be real and must not be `FLA_CONSTANT`.
- The precision of the datatype of **norm** must be equal to that of x .

Arguments:

- | | |
|-------------|---|
| x | – An <code>FLA_Obj</code> representing vector x . |
| norm | – An <code>FLA_Obj</code> representing scalar $\ x\ _2$. |

```
void FLA_Scal( FLA_Obj alpha, FLA_Obj A );
```

Purpose: Perform a scaling operation:

$$A := \alpha A$$

where α is a scalar and A is a general matrix.

Imp. Notes: This function is implemented as a wrapper to `FLA_Scal_external()`.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatype of A if A is real and the precision of A if A is complex.

Arguments:

- | | |
|--------------|--|
| alpha | – An <code>FLA_Obj</code> representing scalar α . |
| A | – An <code>FLA_Obj</code> representing matrix A . |


```
void FLA_Scalc( FLA_Conj conjalpha, FLA_Obj alpha, FLA_Obj A );
```

Purpose: Perform one of the following extended scaling operations:

$$A := \alpha A$$

$$A := \bar{\alpha} A$$

where α is a scalar and A is a general matrix. The `conjalpha` argument allows the computation to proceed as if α were conjugated.

Notes: If α is real, the value of `conjalpha` is ignored and `FLA_Scalc()` behaves exactly as `FLA_Scal()`.

Imp. Notes: This function is implemented as a wrapper to `FLA_Scalc_external()`.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatype of A if A is real and the precision of A if A is complex.

Arguments:

- | | | |
|------------------------|---|--|
| <code>conjalpha</code> | – | Indicates whether the operation proceeds as if α were conjugated. |
| <code>conjalpha</code> | – | Indicates whether the operation proceeds as if α were conjugated. |
| <code>alpha</code> | – | An <code>FLA_Obj</code> representing scalar α . |
| <code>A</code> | – | An <code>FLA_Obj</code> representing matrix A . |

```
void FLA_Scalr( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj A );
```

Purpose: Perform an extended scaling operation on the lower or upper triangle of a matrix:

$$A := \alpha A$$

where α is a scalar and A is a general square matrix. The `uplo` argument indicates whether the lower or upper triangle of A is referenced and updated by the operation.

Imp. Notes: This function is implemented as a wrapper to `FLA_Scalr_external()`.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.
- A must be square.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatype of A if A is real and the precision of A if A is complex.

Arguments:

- | | | |
|--------------------|---|--|
| <code>uplo</code> | – | Indicates whether the lower or upper triangle of A is referenced and updated during the operation. |
| <code>alpha</code> | – | An <code>FLA_Obj</code> representing scalar α . |
| <code>A</code> | – | An <code>FLA_Obj</code> representing matrix A . |

```
void FLA_Swap( FLA_Obj A, FLA_Obj B );
```

Purpose: Swap the contents of two general matrices A and B .

Imp. Notes: This function is implemented as a wrapper to `FLA_Swap_external()`.

Constraints:

- The numerical datatypes of A and B must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The dimensions of A and B must be conformal.

Arguments:

A	–	An <code>FLA_Obj</code> representing matrix A .
B	–	An <code>FLA_Obj</code> representing matrix B .

```
void FLA_Swapt( FLA_Trans transab, FLA_Obj A, FLA_Obj B );
```

Purpose: Swap the contents of two general matrices A and B . If `transab` is `FLA_TRANSPOSE` or `FLA_CONJ_TRANSPOSE`, the computation proceeds as if only A (or only B) were transposed. Furthermore, if `transab` is `FLA_CONJ_NO_TRANSPOSE` or `FLA_CONJ_TRANSPOSE`, both A and B are conjugated after their contents are swapped.

Imp. Notes: This function is implemented as a wrapper to `FLA_Swapt_external()`.

Constraints:

- The numerical datatypes of A and B must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If `transab` equals `FLA_NO_TRANSPOSE` or `FLA_CONJ_NO_TRANSPOSE`, then the dimensions of A and B must be conformal; otherwise, if `transab` equals `FLA_TRANSPOSE` or `FLA_CONJ_TRANSPOSE`, then the dimensions of A^T and B must be conformal.

Arguments:

<code>transab</code>	–	Indicates whether the operation proceeds as if A and B were conjugated and/or transposed.
A	–	An <code>FLA_Obj</code> representing matrix A .
B	–	An <code>FLA_Obj</code> representing matrix B .

5.7.1.2 Level-2 BLAS

```

void FLA_Gemv( FLA_Trans transa, FLA_Obj alpha, FLA_Obj A, FLA_Obj x,
               FLA_Obj beta, FLA_Obj y );
void FLASH_Gemv( FLA_Trans transa, FLA_Obj alpha, FLA_Obj A, FLA_Obj x,
                 FLA_Obj beta, FLA_Obj y );

```

Purpose: Perform one of the following general matrix-matrix multiplication (GEMV) operations:

$$\begin{aligned}
 y &:= \beta y + \alpha Ax \\
 y &:= \beta y + \alpha A^T x \\
 y &:= \beta y + \alpha \bar{A} x \\
 y &:= \beta y + \alpha A^H x
 \end{aligned}$$

where α and β are scalars, A is a general matrix, and x and y are general vectors. The **trans** argument allows the computation to proceed as if A were conjugated and/or transposed.

Constraints:

- The numerical datatypes of A , x , and y must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of A , x , and y .
- The length of y and the number of rows in A (or A^T or A^H) must be equal, and the number of columns in A (or A^T or A^H) and the length of x must be equal.

Int. Notes: `FLA_Gemv()` expects A , x , and y to be flat matrix objects.

Imp. Notes: `FLA_Gemv()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS. `FLASH_Gemv()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the GEMV operation into subproblems expressed in terms of individual blocks of A and subvectors of x and y and then invokes `FLA_Gemv_external()` to perform the computation on these blocks and subvectors.

Arguments:

transa	–	Indicates whether the operation proceeds as if A were conjugated and/or transposed.
alpha	–	An <code>FLA_Obj</code> representing scalar α .
A	–	An <code>FLA_Obj</code> representing matrix A .
x	–	An <code>FLA_Obj</code> representing vector x .
beta	–	An <code>FLA_Obj</code> representing scalar β .
y	–	An <code>FLA_Obj</code> representing vector y .

```
void FLA_Gemvc( FLA_Trans transa, FLA_Conj conjx, FLA_Obj alpha,
                FLA_Obj A, FLA_Obj x, FLA_Obj beta, FLA_Obj y );
```

Purpose: Perform one of the following extended general matrix-vector multiplication operations:

$$\begin{array}{ll}
 y &:= \beta y + \alpha Ax & y &:= \beta y + \alpha A\bar{x} \\
 y &:= \beta y + \alpha A^T x & y &:= \beta y + \alpha A^T \bar{x} \\
 y &:= \beta y + \alpha \bar{A}x & y &:= \beta y + \alpha \bar{A}\bar{x} \\
 y &:= \beta y + \alpha A^H x & y &:= \beta y + \alpha A^H \bar{x}
 \end{array}$$

where α and β are scalars, A is a general matrix, and x and y are general vectors. The **trans** argument allows the computation to proceed as if A were conjugated and/or transposed. Likewise, the **conjx** argument allows the computation to proceed as if x were conjugated.

Notes: The above matrix-vector operations implicitly assume x and y to be column vectors. However, since transposing a vector does not change the way its elements are accessed, we may also express the above operations as:

$$\begin{array}{ll}
 y_r &:= \beta y_r + \alpha x_r A^T & y_r &:= \beta y_r + \alpha \bar{x}_r A^T \\
 y_r &:= \beta y_r + \alpha x_r A & y_r &:= \beta y_r + \alpha \bar{x}_r A \\
 y_r &:= \beta y_r + \alpha x_r A^H & y_r &:= \beta y_r + \alpha \bar{x}_r A^H \\
 y_r &:= \beta y_r + \alpha x_r \bar{A} & y_r &:= \beta y_r + \alpha \bar{x}_r \bar{A}
 \end{array}$$

respectively, where x_r and y_r are row vectors.

If A , x , and y are real, the value of **conjx** is ignored and **FLA_Gemvc()** behaves exactly as **FLA_Gemv()**.

Imp. Notes: This function is implemented as a wrapper to **FLA_Gemvc_external()**.

Constraints:

- The numerical datatypes of A , x , and y must be identical and floating-point, and must not be **FLA_CONSTANT**.
- If α and β are not of datatype **FLA_CONSTANT**, then they must match the datatypes of A , x , and y .
- The length of y and the number of rows in A (or A^T or A^H) must be equal, and the number of columns in A (or A^T or A^H) and the length of x must be equal.

Arguments:

- | | | |
|---------------|---|---|
| transa | – | Indicates whether the operation proceeds as if A were conjugated and/or transposed. |
| conjx | – | Indicates whether the operation proceeds as if x were conjugated. |
| alpha | – | An FLA_Obj representing scalar α . |
| A | – | An FLA_Obj representing matrix A . |
| x | – | An FLA_Obj representing vector x . |
| beta | – | An FLA_Obj representing scalar β . |
| y | – | An FLA_Obj representing vector y . |

```
void FLA_Ger( FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj A );
```

Purpose: Perform a general rank-1 update:

$$A := A + \alpha xy^T$$

where α is a scalar, A is a general matrix, and x and y are general vectors.

Imp. Notes: This function is implemented as a wrapper to `FLA_Ger_external()`.

Constraints:

- The numerical datatypes of A , x , and y must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatypes of A , x , and y .
- The length of x and the number of rows in A must be equal, and the length of y and the number of columns in A must be equal.

Arguments:

- | | | |
|-------|---|--|
| alpha | – | An <code>FLA_Obj</code> representing scalar α . |
| x | – | An <code>FLA_Obj</code> representing vector x . |
| y | – | An <code>FLA_Obj</code> representing vector y . |
| A | – | An <code>FLA_Obj</code> representing matrix A . |

```
void FLA_Gerc( FLA_Conj conjx, FLA_Conj conjy, FLA_Obj alpha,
               FLA_Obj x, FLA_Obj y, FLA_Obj A );
```

Purpose: Perform one of the following extended general rank-1 updates:

$$A := A + \alpha xy^T$$

$$A := A + \alpha x\bar{y}^T$$

$$A := A + \alpha \bar{x}y^T$$

$$A := A + \alpha \bar{x}\bar{y}^T$$

where α is a scalar, A is a general matrix, and x and y are general vectors. The `conjx` and `conjy` arguments allow the computation to proceed as if x and/or y were conjugated.

Notes: If A , x , and y are real, the values of `conjx` and `conjy` are ignored and `FLA_Gerc()` behaves exactly as `FLA_Ger()`.

Imp. Notes: This function is implemented as a wrapper to `FLA_Gerc_external()`.

Constraints:

- The numerical datatypes of A , x , and y must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatypes of A , x , and y .
- The length of x and the number of rows in A must be equal, and the length of y and the number of columns in A must be equal.

Arguments:

- | | | |
|-------|---|---|
| conjx | – | Indicates whether the operation proceeds as if x were conjugated. |
| conjy | – | Indicates whether the operation proceeds as if y were conjugated. |
| alpha | – | An <code>FLA_Obj</code> representing scalar α . |
| x | – | An <code>FLA_Obj</code> representing vector x . |
| y | – | An <code>FLA_Obj</code> representing vector y . |
| A | – | An <code>FLA_Obj</code> representing matrix A . |

```
void FLA_Hemv( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj A, FLA_Obj x,
               FLA_Obj beta, FLA_Obj y );
```

Purpose: Perform a Hermitian matrix-vector multiplication operation:

$$y := \beta y + \alpha Ax$$

where α and β are scalars, A is a complex Hermitian matrix, and x and y are general complex vectors. The `uplo` argument indicates whether the lower or upper triangle of A is referenced by the operation.

Imp. Notes: This function is implemented as a wrapper to `FLA_Hemv_external()`.

Constraints:

- The numerical datatypes of A , x , and y must be identical and complex, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of A , x , and y .
- The length of x , the length of y , and the order of A must be equal.

Arguments:

<code>uplo</code>	–	Indicates whether the lower or upper triangle of A is referenced during the operation.
<code>alpha</code>	–	An <code>FLA_Obj</code> representing scalar α .
<code>A</code>	–	An <code>FLA_Obj</code> representing matrix A .
<code>x</code>	–	An <code>FLA_Obj</code> representing vector x .
<code>beta</code>	–	An <code>FLA_Obj</code> representing scalar β .
<code>y</code>	–	An <code>FLA_Obj</code> representing vector y .

```
void FLA_Hemvc( FLA_Uplo uplo, FLA_Conj conj, FLA_Obj alpha,
               FLA_Obj A, FLA_Obj x, FLA_Obj beta, FLA_Obj y );
```

Purpose: Perform one of the following extended Hermitian matrix-vector multiplication operations:

$$\begin{aligned} y &:= \beta y + \alpha Ax \\ y &:= \beta y + \alpha \bar{A}x \end{aligned}$$

where α and β are scalars, A is a complex Hermitian matrix, and x and y are general complex vectors. The `uplo` argument indicates whether the lower or upper triangle of A is referenced by the operation. The `conj` argument allows the computation to proceed as if A were conjugated.

Imp. Notes: This function is implemented as a wrapper to `FLA_Hemvc_external()`.

Constraints:

- The numerical datatypes of A , x , and y must be identical and complex, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of A , x , and y .
- The length of x , the length of y , and the order of A must be equal.

Arguments:

- | | | |
|--------------------|---|--|
| <code>uplo</code> | – | Indicates whether the lower or upper triangle of A is referenced during the operation. |
| <code>conj</code> | – | Indicates whether the operation proceeds as if A were conjugated. |
| <code>alpha</code> | – | An <code>FLA_Obj</code> representing scalar α . |
| <code>A</code> | – | An <code>FLA_Obj</code> representing matrix A . |
| <code>x</code> | – | An <code>FLA_Obj</code> representing vector x . |
| <code>beta</code> | – | An <code>FLA_Obj</code> representing scalar β . |
| <code>y</code> | – | An <code>FLA_Obj</code> representing vector y . |

```
void FLA_Her( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj A );
```

Purpose: Perform a complex Hermitian rank-1 update:

$$A := A + \alpha x x^H$$

where α is a scalar, A is a complex Hermitian matrix, and x is a general complex vector. The `uplo` argument indicates whether the lower or upper triangle of A is referenced and updated by the operation.

Imp. Notes: This function is implemented as a wrapper to `FLA_Her_external()`.

Constraints:

- The numerical datatypes of A and x must be identical and complex, and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatypes of A and x .
- The length of x and the order of A must be equal.

Arguments:

- | | | |
|--------------------|---|--|
| <code>uplo</code> | – | Indicates whether the lower or upper triangle of A is referenced and updated during the operation. |
| <code>alpha</code> | – | An <code>FLA_Obj</code> representing scalar α . |
| <code>x</code> | – | An <code>FLA_Obj</code> representing vector x . |
| <code>A</code> | – | An <code>FLA_Obj</code> representing matrix A . |

```
void FLA_Herc( FLA_Uplo uplo, FLA_Conj conjx, FLA_Obj alpha, FLA_Obj x, FLA_Obj A );
```

Purpose: Perform one of the following extended complex Hermitian rank-1 updates:

$$\begin{aligned} A &:= A + \alpha x x^H \\ A &:= A + \alpha \bar{x} \bar{x}^H \end{aligned}$$

where α is a scalar, A is a complex Hermitian matrix, and x is a general complex vector. The `uplo` argument indicates whether the lower or upper triangle of A is referenced and updated by the operation. The `conjx` argument allows the computation to proceed as if x were conjugated.

Imp. Notes: This function is implemented as a wrapper to `FLA_Herc_external()`.

Constraints:

- The numerical datatypes of A and x must be identical and complex, and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatypes of A and x .
- The length of x and the order of A must be equal.

Arguments:

- | | |
|--------------------|--|
| <code>uplo</code> | – Indicates whether the lower or upper triangle of A is referenced and updated during the operation. |
| <code>conjx</code> | – Indicates whether the operation proceeds as if x were conjugated. |
| <code>alpha</code> | – An <code>FLA_Obj</code> representing scalar α . |
| <code>x</code> | – An <code>FLA_Obj</code> representing vector x . |
| <code>A</code> | – An <code>FLA_Obj</code> representing matrix A . |

```
void FLA_Her2( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj A );
```

Purpose: Perform a complex Hermitian rank-2 update:

$$A := A + \alpha x y^H + \bar{\alpha} y x^H$$

where α is a scalar, A is a complex Hermitian matrix, and x and y are general complex vectors. The `uplo` argument indicates whether the lower or upper triangle of A is referenced and updated by the operation.

Imp. Notes: This function is implemented as a wrapper to `FLA_Her2_external()`.

Constraints:

- The numerical datatypes of A , x , and y must be identical and complex, and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatypes of A , x , and y .
- The length of x , the length of y , and the order of A must be equal.

Arguments:

- | | |
|--------------------|--|
| <code>uplo</code> | – Indicates whether the lower or upper triangle of A is referenced during the operation. |
| <code>alpha</code> | – An <code>FLA_Obj</code> representing scalar α . |
| <code>x</code> | – An <code>FLA_Obj</code> representing vector x . |
| <code>y</code> | – An <code>FLA_Obj</code> representing vector y . |
| <code>A</code> | – An <code>FLA_Obj</code> representing matrix A . |


```
void FLA_Her2c( FLA_Uplo uplo, FLA_Conj conjxy, FLA_Obj alpha, FLA_Obj x, FLA_Obj y,
               FLA_Obj A );
```

Purpose: Perform one of the following extended complex Hermitian rank-2 updates:

$$\begin{aligned} A &:= A + \alpha xy^H + \bar{\alpha} yx^H \\ A &:= A + \alpha \bar{x} \bar{y}^H + \bar{\alpha} \bar{y} \bar{x}^H \end{aligned}$$

where α is a scalar, A is a complex Hermitian matrix, and x and y are general complex vectors. The `uplo` argument indicates whether the lower or upper triangle of A is referenced and updated by the operation. The `conjxy` argument allows the computation to proceed as if x and y were conjugated.

Imp. Notes: This function is implemented as a wrapper to `FLA_Her2c_external()`.

Constraints:

- The numerical datatypes of A , x , and y must be identical and complex, and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatypes of A , x , and y .
- The length of x , the length of y , and the order of A must be equal.

Arguments:

- | | | |
|--------------------|---|--|
| <code>uplo</code> | – | Indicates whether the lower or upper triangle of A is referenced during the operation. |
| <code>alpha</code> | – | An <code>FLA_Obj</code> representing scalar α . |
| <code>x</code> | – | An <code>FLA_Obj</code> representing vector x . |
| <code>y</code> | – | An <code>FLA_Obj</code> representing vector y . |
| <code>A</code> | – | An <code>FLA_Obj</code> representing matrix A . |

```
void FLA_Symv( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj A, FLA_Obj x,
              FLA_Obj beta, FLA_Obj y );
```

Purpose: Perform a symmetric matrix-vector multiplication operation:

$$y := \beta y + \alpha Ax$$

where α and β are scalars, A is a symmetric matrix, and x and y are general vectors. The `uplo` argument indicates whether the lower or upper triangle of A is referenced by the operation.

Imp. Notes: This function is implemented as a wrapper to `FLA_Symv_external()`.

Constraints:

- The numerical datatypes of A , x , and y must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of A , x , and y .
- The length of x , the length of y , and the order of A must be equal.

Arguments:

- | | | |
|--------------------|---|--|
| <code>uplo</code> | – | Indicates whether the lower or upper triangle of A is referenced during the operation. |
| <code>alpha</code> | – | An <code>FLA_Obj</code> representing scalar α . |
| <code>A</code> | – | An <code>FLA_Obj</code> representing matrix A . |
| <code>x</code> | – | An <code>FLA_Obj</code> representing vector x . |
| <code>beta</code> | – | An <code>FLA_Obj</code> representing scalar β . |
| <code>y</code> | – | An <code>FLA_Obj</code> representing vector y . |

```
void FLA_Syr( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj A );
```

Purpose: Perform a symmetric rank-1 update:

$$A := A + \alpha x x^T$$

where α is a scalar, A is a symmetric matrix, and x is a general vector. The `uplo` argument indicates whether the lower or upper triangle of A is referenced and updated by the operation.

Imp. Notes: This function is implemented as a wrapper to `FLA_Syr_external()`.

Constraints:

- The numerical datatypes of A and x must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatypes of A and x .
- The length of x and the order of A must be equal.

Arguments:

- | | | |
|--------------------|---|--|
| <code>uplo</code> | – | Indicates whether the lower or upper triangle of A is referenced during the operation. |
| <code>alpha</code> | – | An <code>FLA_Obj</code> representing scalar α . |
| <code>x</code> | – | An <code>FLA_Obj</code> representing vector x . |
| <code>A</code> | – | An <code>FLA_Obj</code> representing matrix A . |

```
void FLA_Syr2( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj A );
```

Purpose: Perform a symmetric rank-2 update:

$$A := A + \alpha x y^T + \alpha y x^T$$

where α is a scalar, A is a symmetric matrix, and x and y are general vectors. The `uplo` argument indicates whether the lower or upper triangle of A is referenced and updated by the operation.

Imp. Notes: This function is implemented as a wrapper to `FLA_Syr2_external()`.

Constraints:

- The numerical datatypes of A , x , and y must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatypes of A , x , and y .
- The length of x , the length of y , and the order of A must be equal.

Arguments:

- | | | |
|--------------------|---|--|
| <code>uplo</code> | – | Indicates whether the lower or upper triangle of A is referenced during the operation. |
| <code>alpha</code> | – | An <code>FLA_Obj</code> representing scalar α . |
| <code>x</code> | – | An <code>FLA_Obj</code> representing vector x . |
| <code>y</code> | – | An <code>FLA_Obj</code> representing vector y . |
| <code>A</code> | – | An <code>FLA_Obj</code> representing matrix A . |

```
void FLA_Trmv( FLA_Uplo uplo, FLA_Trans transa, FLA_Diag diag, FLA_Obj A, FLA_Obj x );
```

Purpose: Perform one of the following triangular matrix-vector multiplication operations:

$$\begin{aligned} x &:= Ax \\ x &:= A^T x \\ x &:= \bar{A}x \\ x &:= A^H x \end{aligned}$$

where A is a triangular matrix and x is a general vector. The **uplo** argument indicates whether the lower or upper triangle of A is referenced by the operation. The **transa** argument allows the computation to proceed as if A were conjugated and/or transposed. The **diag** argument indicates whether the diagonal of A is unit or non-unit.

Imp. Notes: This function is implemented as a wrapper to `FLA_Trmv_external()`.

Constraints:

- The numerical datatypes of A and x must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The length of x and the order of A must be equal.
- **diag** may not be `FLA_ZERO_DIAG`.

Arguments:

- | | |
|---------------|--|
| uplo | – Indicates whether the lower or upper triangle of A is referenced during the operation. |
| transa | – Indicates whether the operation proceeds as if A were conjugated and/or transposed. |
| diag | – Indicates whether the diagonal of A is unit or non-unit. |
| A | – An <code>FLA_Obj</code> representing matrix A . |
| x | – An <code>FLA_Obj</code> representing vector x . |

```
void FLA_Trmvsx( FLA_Uplo uplo, FLA_Trans transa, FLA_Diag diag, FLA_Obj alpha,
                 FLA_Obj A, FLA_Obj x, FLA_Obj beta, FLA_Obj y );
```

Purpose: Perform one of the following extended triangular matrix-vector multiplication operations:

$$\begin{aligned} y &:= \beta y + \alpha Ax \\ y &:= \beta y + \alpha A^T x \\ y &:= \beta y + \alpha \bar{A}x \\ y &:= \beta y + \alpha A^H x \end{aligned}$$

where α and β are scalars, A is a triangular matrix, and x and y are general vectors. The **uplo** argument indicates whether the lower or upper triangle of A is referenced by the operation. The **transa** argument allows the computation to proceed as if A were conjugated and/or transposed. The **diag** argument indicates whether the diagonal of A is unit or non-unit.

Imp. Notes: This function is implemented as a wrapper to `FLA_Trmvsx_external()`.

Constraints:

- The numerical datatypes of A , x , and y must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of A , x , and y .
- The length of x , the length of y , and the order of A must be equal.
- **diag** may not be `FLA_ZERO_DIAG`.

Arguments:

uplo	–	Indicates whether the lower or upper triangle of A is referenced during the operation.
transa	–	Indicates whether the operation proceeds as if A were conjugated and/or transposed.
diag	–	Indicates whether the diagonal of A is unit or non-unit.
alpha	–	An <code>FLA_Obj</code> representing scalar α .
A	–	An <code>FLA_Obj</code> representing matrix A .
x	–	An <code>FLA_Obj</code> representing vector x .
beta	–	An <code>FLA_Obj</code> representing scalar β .
y	–	An <code>FLA_Obj</code> representing vector y .

```
void FLA_Trsv( FLA_Uplo uplo, FLA_Trans transa, FLA_Diag diag, FLA_Obj A, FLA_Obj b );
void FLASH_Trsv( FLA_Uplo uplo, FLA_Trans transa, FLA_Diag diag, FLA_Obj A, FLA_Obj b );
```

Purpose: Perform one of the following triangular solves with multiple right-hand sides:

$$\begin{aligned} Ax &= b \\ A^T x &= b \\ \bar{A}x &= b \\ A^H x &= b \end{aligned}$$

which, respectively, are solved by overwriting b with the contents of the solution vector x as follows:

$$\begin{aligned} b &:= A^{-1}b \\ b &:= A^{-T}b \\ b &:= \bar{A}^{-1}b \\ b &:= A^{-H}b \end{aligned}$$

where A is a triangular matrix and x and b are general vectors. The `uplo` argument indicates whether the lower or upper triangle of A is referenced by the operation. The `transa` argument allows the computation to proceed as if A were conjugated and/or transposed. The `diag` argument indicates whether the diagonal of A is unit or non-unit.

Constraints:

- The numerical datatypes of A and b must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The length of b and the order of A must be equal.
- `diag` may not be `FLA_ZERO_DIAG`.

Int. Notes: `FLA_Trsv()` expects A and b to be flat matrix objects.

Imp. Notes: `FLA_Trsv()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS. `FLASH_Trsv()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the TRSV operation into subproblems expressed in terms of individual blocks of A and subvectors of b and then invokes external BLAS to perform the computation on these blocks and subvectors.

Arguments:

- | | |
|---------------------|--|
| <code>uplo</code> | – Indicates whether the lower or upper triangle of A is referenced during the operation. |
| <code>transa</code> | – Indicates whether the operation proceeds as if A were conjugated and/or transposed. |
| <code>diag</code> | – Indicates whether the diagonal of A is unit or non-unit. |
| <code>A</code> | – An <code>FLA_Obj</code> representing matrix A . |
| <code>b</code> | – An <code>FLA_Obj</code> representing vector b . |

```
void FLA_Trsvsx( FLA_Uplo uplo, FLA_Trans transa, FLA_Diag diag, FLA_Obj alpha,
                FLA_Obj A, FLA_Obj b, FLA_Obj beta, FLA_Obj y );
```

Purpose: Perform one of the following extended triangular solves with multiple right-hand sides:

$$\begin{aligned} y &:= \beta y + \alpha A^{-1} b \\ y &:= \beta y + \alpha A^{-T} b \\ y &:= \beta y + \alpha \bar{A}^{-1} b \\ y &:= \beta y + \alpha A^{-H} b \end{aligned}$$

where α and β are scalars, A is a triangular matrix, and b and y are general vectors. The `uplo` argument indicates whether the lower or upper triangle of A is referenced by the operation. The `transa` argument allows the computation to proceed as if A were conjugated and/or transposed. The `diag` argument indicates whether the diagonal of A is unit or non-unit.

Imp. Notes: This function is implemented as a wrapper to `FLA_Trsvsx_external()`.

Constraints:

- The numerical datatypes of A , b , and y must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of A , b , and y .
- The length of b , the length of y , and the order of A must be equal.
- `diag` may not be `FLA_ZERO_DIAG`.

Arguments:

<code>uplo</code>	–	Indicates whether the lower or upper triangle of A is referenced during the operation.
<code>transa</code>	–	Indicates whether the operation proceeds as if A were conjugated and/or transposed.
<code>diag</code>	–	Indicates whether the diagonal of A is unit or non-unit.
<code>alpha</code>	–	An <code>FLA_Obj</code> representing scalar α .
<code>A</code>	–	An <code>FLA_Obj</code> representing matrix A .
<code>b</code>	–	An <code>FLA_Obj</code> representing vector b .
<code>beta</code>	–	An <code>FLA_Obj</code> representing scalar β .
<code>y</code>	–	An <code>FLA_Obj</code> representing vector y .

5.7.1.3 Level-3 BLAS

```

void FLA_Gemm( FLA_Trans transa, FLA_Trans transb, FLA_Obj alpha,
               FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C );
void FLASH_Gemm( FLA_Trans transa, FLA_Trans transb, FLA_Obj alpha,
                 FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C );

```

Purpose: Perform one of the following general matrix-matrix multiplication (GEMM) operations:

$$\begin{array}{ll}
C := \beta C + \alpha AB & C := \beta C + \alpha \bar{A}B \\
C := \beta C + \alpha AB^T & C := \beta C + \alpha \bar{A}B^T \\
C := \beta C + \alpha A\bar{B} & C := \beta C + \alpha \bar{A}\bar{B} \\
C := \beta C + \alpha AB^H & C := \beta C + \alpha \bar{A}B^H \\
C := \beta C + \alpha A^T B & C := \beta C + \alpha A^H B \\
C := \beta C + \alpha A^T B^T & C := \beta C + \alpha A^H B^T \\
C := \beta C + \alpha A^T \bar{B} & C := \beta C + \alpha A^H \bar{B} \\
C := \beta C + \alpha A^T B^H & C := \beta C + \alpha A^H B^H
\end{array}$$

where α and β are scalars and A , B , and C are general matrices. The **transa** and **transb** arguments allows the computation to proceed as if A and/or B were conjugated and/or transposed.

Constraints:

- The numerical datatypes of A , B , and C must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of A , B , and C .
- The number of rows in C and the number of rows in A (or A^T) must be equal; the number of columns in C and the number of columns of B (or B^T) must be equal; and the number of columns in A (or A^T) and the number of rows in B (or B^T) must be equal.

Int. Notes: `FLA_Gemm()` expects A , B , and C to be flat matrix objects.

Imp. Notes: `FLA_Gemm()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS. `FLASH_Gemm()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the GEMM operation into subproblems expressed in terms of individual blocks of A , B , and C and then invokes `FLA_Gemm_external()` to perform the computation on these blocks.

Arguments:

- | | | |
|---------------|---|---|
| transa | – | Indicates whether the operation proceeds as if A were conjugated and/or transposed. |
| transb | – | Indicates whether the operation proceeds as if B were conjugated and/or transposed. |
| alpha | – | An <code>FLA_Obj</code> representing scalar α . |
| A | – | An <code>FLA_Obj</code> representing matrix A . |
| B | – | An <code>FLA_Obj</code> representing matrix B . |
| beta | – | An <code>FLA_Obj</code> representing scalar β . |
| C | – | An <code>FLA_Obj</code> representing matrix C . |

```

void FLA_Hemm( FLA_Side side, FLA_Uplo uplo, FLA_Obj alpha,
               FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C );
void FLASH_Hemm( FLA_Side side, FLA_Uplo uplo, FLA_Obj alpha,
                 FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C );

```

Purpose: Perform one of the following Hermitian matrix-matrix multiplication (HEMM) operations:

$$\begin{aligned}
 C &:= \beta C + \alpha AB \\
 C &:= \beta C + \alpha BA
 \end{aligned}$$

where α and β are scalars, A is a complex Hermitian matrix, and B and C are general complex matrices. The `side` argument indicates whether matrix A is multiplied on the left or the right side of B . The `uplo` argument indicates whether the lower or upper triangle of A is referenced by the operation.

Constraints:

- The numerical datatypes of A , B , and C must be identical and complex, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of A , B , and C .
- The dimensions of C and B must be conformal.
- If `side` equals `FLA_LEFT`, then the number of rows in C and the order of A must be equal; otherwise, if `side` equals `FLA_RIGHT`, then the number of columns in C and the order of A must be equal.

Int. Notes: `FLA_Hemm()` expects A , B , and C to be flat matrix objects.

Imp. Notes: `FLA_Hemm()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS. `FLASH_Hemm()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the HEMM operation into subproblems expressed in terms of individual blocks of A , B , and C and then invokes external BLAS to perform the computation on these blocks.

Arguments:

<code>side</code>	–	Indicates whether A is multiplied on the left or right side of B .
<code>uplo</code>	–	Indicates whether the lower or upper triangle of A is referenced during the operation.
<code>alpha</code>	–	An <code>FLA_Obj</code> representing scalar α .
<code>A</code>	–	An <code>FLA_Obj</code> representing matrix A .
<code>B</code>	–	An <code>FLA_Obj</code> representing matrix B .
<code>beta</code>	–	An <code>FLA_Obj</code> representing scalar β .
<code>C</code>	–	An <code>FLA_Obj</code> representing matrix C .


```

void FLA_Herk( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha,
               FLA_Obj A, FLA_Obj beta, FLA_Obj C );
void FLASH_Herk( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha,
                 FLA_Obj A, FLA_Obj beta, FLA_Obj C );

```

Purpose: Perform one of the following Hermitian rank-k update (HERK) operations:

$$\begin{aligned}
 C &:= \beta C + \alpha A A^H \\
 C &:= \beta C + \alpha A^H A
 \end{aligned}$$

where α and β are scalars, C is a complex Hermitian matrix, and A is a general complex matrix. The `uplo` argument indicates whether the lower or upper triangle of C is referenced and updated by the operation. The `trans` argument allows the computation to proceed as if A were conjugate-transposed, which results in the alternate rank-k product $A^H A$.

Constraints:

- The numerical datatypes of A and C must be identical and complex, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must be real and match the precision of the datatypes of A and C .
- If `trans` equals `FLA_NO_TRANSPOSE`, then the order of matrix C and the the number of rows in A must be equal; otherwise, if `trans` equals `FLA_CONJ_TRANSPOSE`, then the order of matrix C and the number of columns in A must be equal.
- `trans` may not be `FLA_TRANSPOSE` or `FLA_CONJ_NO_TRANSPOSE`.

Int. Notes: `FLA_Herk()` expects A and C to be flat matrix objects.

Imp. Notes: `FLA_Herk()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS. `FLASH_Herk()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the HERK operation into subproblems expressed in terms of individual blocks of A and C and then invokes external BLAS to perform the computation on these blocks.

Arguments:

- | | |
|---------------------|--|
| <code>uplo</code> | – Indicates whether the lower or upper triangle of C is referenced during the operation. |
| <code>transa</code> | – Indicates whether the operation proceeds as if A were conjugate-transposed. |
| <code>alpha</code> | – An <code>FLA_Obj</code> representing scalar α . |
| <code>A</code> | – An <code>FLA_Obj</code> representing matrix A . |
| <code>beta</code> | – An <code>FLA_Obj</code> representing scalar β . |
| <code>C</code> | – An <code>FLA_Obj</code> representing matrix C . |

```

void FLA_Her2k( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha,
               FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C );
void FLASH_Her2k( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha,
                 FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C );

```

Purpose: Perform one of the following Hermitian rank-2k update (HER2K) operations:

$$\begin{aligned}
 C &:= \beta C + \alpha AB^H + \bar{\alpha} BA^H \\
 C &:= \beta C + \alpha A^H B + \bar{\alpha} B^H A
 \end{aligned}$$

where α and β are scalars, C is a complex Hermitian matrix, and A and B are general complex matrices. The `uplo` argument indicates whether the lower or upper triangle of C is referenced and updated by the operation. The `trans` argument allows the computation to proceed as if A and B were conjugate-transposed, which results in the alternate rank-2k products $A^H B$ and $B^H A$.

Constraints:

- The numerical datatypes of A , B , and C must be identical and complex, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then their datatypes must be real and complex, respectively, and match the precision of the datatypes of A , B , and C .
- The dimensions of A and B must be conformal.
- If `trans` equals `FLA_NO_TRANSPOSE`, then the order of matrix C and the the number of rows in A and B must be equal; otherwise, if `trans` equals `FLA_CONJ_TRANSPOSE`, then the order of matrix C and the number of columns in A and B must be equal.
- `trans` may not be `FLA_TRANSPOSE` or `FLA_CONJ_NO_TRANSPOSE`.

Int. Notes: `FLA_Her2k()` expects A , B , and C to be flat matrix objects.

Imp. Notes: `FLA_Her2k()` invokes a single FLAME/C variant to induce a blocked algorithm with sub-problems performed by calling wrappers to external BLAS. `FLASH_Her2k()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the HER2K operation into subproblems expressed in terms of individual blocks of A , B , and C and then invokes external BLAS to perform the computation on these blocks.

Arguments:

<code>uplo</code>	–	Indicates whether the lower or upper triangle of C is referenced during the operation.
<code>transa</code>	–	Indicates whether the operation proceeds as if A and B were conjugate-transposed.
<code>alpha</code>	–	An <code>FLA_Obj</code> representing scalar α .
<code>A</code>	–	An <code>FLA_Obj</code> representing matrix A .
<code>B</code>	–	An <code>FLA_Obj</code> representing matrix B .
<code>beta</code>	–	An <code>FLA_Obj</code> representing scalar β .
<code>C</code>	–	An <code>FLA_Obj</code> representing matrix C .

```

void FLA_Symm( FLA_Side side, FLA_Uplo uplo, FLA_Obj alpha,
               FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C );
void FLASH_Symm( FLA_Side side, FLA_Uplo uplo, FLA_Obj alpha,
                 FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C );

```

Purpose: Perform one of the following symmetric matrix-matrix multiplication (SYMM) operations:

$$\begin{aligned}
 C &:= \beta C + \alpha AB \\
 C &:= \beta C + \alpha BA
 \end{aligned}$$

where α and β are scalars, A is a symmetric matrix, and B and C are general matrices. The `side` argument indicates whether the symmetric matrix A is multiplied on the left or the right side of B . The `uplo` argument indicates whether the lower or upper triangle of A is referenced by the operation.

Constraints:

- The numerical datatypes of A , B , and C must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of A , B , and C .
- The dimensions of C and B must be conformal.
- If `side` equals `FLA_LEFT`, then the number of rows in C and the order of A must be equal; otherwise, if `side` equals `FLA_RIGHT`, then the number of columns in C and the order of A must be equal.

Int. Notes: `FLA_Symm()` expects A , B , and C to be flat matrix objects.

Imp. Notes: `FLA_Symm()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS. `FLASH_Symm()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the SYMM operation into subproblems expressed in terms of individual blocks of A , B , and C and then invokes external BLAS to perform the computation on these blocks.

Arguments:

<code>side</code>	–	Indicates whether A is multiplied on the left or right side of B .
<code>uplo</code>	–	Indicates whether the lower or upper triangle of A is referenced during the operation.
<code>alpha</code>	–	An <code>FLA_Obj</code> representing scalar α .
<code>A</code>	–	An <code>FLA_Obj</code> representing matrix A .
<code>B</code>	–	An <code>FLA_Obj</code> representing matrix B .
<code>beta</code>	–	An <code>FLA_Obj</code> representing scalar β .
<code>C</code>	–	An <code>FLA_Obj</code> representing matrix C .

```

void FLA_Syrk( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha,
              FLA_Obj A, FLA_Obj beta, FLA_Obj C );
void FLASH_Syrk( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha,
                FLA_Obj A, FLA_Obj beta, FLA_Obj C );

```

Purpose: Perform one of the following symmetric rank-k update (SYRK) operations:

$$\begin{aligned}
 C &:= \beta C + \alpha A A^T \\
 C &:= \beta C + \alpha A^T A
 \end{aligned}$$

where α and β are scalars, C is a symmetric matrix, and A is a general matrix. The `uplo` argument indicates whether the lower or upper triangle of C is referenced and updated by the operation. The `trans` argument allows the computation to proceed as if A were transposed, which results in the alternate rank-k product $A^T A$.

Constraints:

- The numerical datatypes of A and C must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of A and C .
- If `trans` equals `FLA_NO_TRANSPOSE`, then the order of matrix C and the the number of rows in A must be equal; otherwise, if `trans` equals `FLA_TRANSPOSE`, then the order of matrix C and the number of columns in A must be equal.
- `trans` may not be `FLA_CONJ_TRANSPOSE` or `FLA_CONJ_NO_TRANSPOSE`.

Int. Notes: `FLA_Syrk()` expects A and C to be flat matrix objects.

Imp. Notes: `FLA_Syrk()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS. `FLASH_Syrk()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the SYRK operation into subproblems expressed in terms of individual blocks of A and C and then invokes external BLAS to perform the computation on these blocks.

Arguments:

- | | | |
|---------------------|---|--|
| <code>uplo</code> | – | Indicates whether the lower or upper triangle of C is referenced during the operation. |
| <code>transa</code> | – | Indicates whether the operation proceeds as if A is transposed. |
| <code>alpha</code> | – | An <code>FLA_Obj</code> representing scalar α . |
| <code>A</code> | – | An <code>FLA_Obj</code> representing matrix A . |
| <code>beta</code> | – | An <code>FLA_Obj</code> representing scalar β . |
| <code>C</code> | – | An <code>FLA_Obj</code> representing matrix C . |

```

void FLA_Syr2k( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha,
               FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C );
void FLASH_Syr2k( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha,
                 FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C );

```

Purpose: Perform one of the following symmetric rank-2k update (SYR2K) operations:

$$\begin{aligned}
 C &:= \beta C + \alpha AB^T + \alpha BA^T \\
 C &:= \beta C + \alpha A^T B + \alpha B^T A
 \end{aligned}$$

where α and β are scalars, C is a symmetric matrix, and A and B are general matrices. The `uplo` argument indicates whether the lower or upper triangle of C is referenced and updated by the operation. The `trans` argument allows the computation to proceed as if A and B were transposed, which results in the alternate rank-2k products $A^T B$ and $B^T A$.

Constraints:

- The numerical datatypes of A , B , and C must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of A , B , and C .
- The dimensions of A and B must be conformal.
- If `trans` equals `FLA_NO_TRANSPOSE`, then the order of matrix C and the the number of rows in A and B must be equal; otherwise, if `trans` equals `FLA_TRANSPOSE`, then the order of matrix C and the number of columns in A and B must be equal.
- `trans` may not be `FLA_CONJ_TRANSPOSE` or `FLA_CONJ_NO_TRANSPOSE`.

Int. Notes: `FLA_Syr2k()` expects A , B , and C to be flat matrix objects.

Imp. Notes: `FLA_Syr2k()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS. `FLASH_Syr2k()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the SYR2K operation into subproblems expressed in terms of individual blocks of A , B , and C and then invokes external BLAS to perform the computation on these blocks.

Arguments:

- | | |
|---------------------|--|
| <code>uplo</code> | – Indicates whether the lower or upper triangle of C is referenced during the operation. |
| <code>transa</code> | – Indicates whether the operation proceeds as if A and B were transposed. |
| <code>alpha</code> | – An <code>FLA_Obj</code> representing scalar α . |
| <code>A</code> | – An <code>FLA_Obj</code> representing matrix A . |
| <code>B</code> | – An <code>FLA_Obj</code> representing matrix B . |
| <code>beta</code> | – An <code>FLA_Obj</code> representing scalar β . |
| <code>C</code> | – An <code>FLA_Obj</code> representing matrix C . |

```

void FLA_Trmm( FLA_Side side, FLA_Uplo uplo, FLA_Trans trans,
               FLA_Diag diag, FLA_Obj alpha, FLA_Obj A, FLA_Obj B );
void FLASH_Trmm( FLA_Side side, FLA_Uplo uplo, FLA_Trans trans,
                 FLA_Diag diag, FLA_Obj alpha, FLA_Obj A, FLA_Obj B );

```

Purpose: Perform one of the following triangular matrix-matrix multiplication (TRMM) operations:

$$\begin{array}{ll}
B &:= \alpha AB & B &:= \alpha BA \\
B &:= \alpha A^T B & B &:= \alpha BA^T \\
B &:= \alpha \bar{A} B & B &:= \alpha B \bar{A} \\
B &:= \alpha A^H B & B &:= \alpha BA^H
\end{array}$$

where α is a scalar, A is a triangular matrix, and B is a general matrix. The **side** argument indicates whether the triangular matrix A is multiplied on the left or the right side of B . The **uplo** argument indicates whether the lower or upper triangle of A is referenced by the operation. The **trans** argument may be used to perform the check as if A were conjugated and/or transposed. The **diag** argument indicates whether the diagonal of A is unit or non-unit.

Constraints:

- The numerical datatypes of A and B must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatypes of A and B .
- If **side** equals `FLA_LEFT`, then the number of rows in B and the order of A must be equal; otherwise, if **side** equals `FLA_RIGHT`, then the number of columns in B and the order of A must be equal.
- **diag** may not be `FLA_ZERO_DIAG`.

Int. Notes: `FLA_Trmm()` expects A and B to be flat matrix objects.

Imp. Notes: `FLA_Trmm()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS. `FLASH_Trmm()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the TRMM operation into subproblems expressed in terms of individual blocks of A and B and then invokes external BLAS to perform the computation on these blocks.

Arguments:

side	–	Indicates whether A is multiplied on the left or right side of B .
uplo	–	Indicates whether the lower or upper triangle of A is referenced during the operation.
trans	–	Indicates whether the operation proceeds as if A were conjugated and/or transposed.
diag	–	Indicates whether the diagonal of A is unit or non-unit.
alpha	–	An <code>FLA_Obj</code> representing scalar α .
A	–	An <code>FLA_Obj</code> representing matrix A .
B	–	An <code>FLA_Obj</code> representing matrix B .

```

void FLA_Trsm( FLA_Side side, FLA_Uplo uplo, FLA_Trans trans, FLA_Diag diag,
               FLA_Obj alpha, FLA_Obj A, FLA_Obj B );
void FLASH_Trsm( FLA_Side side, FLA_Uplo uplo, FLA_Trans trans, FLA_Diag diag,
                 FLA_Obj alpha, FLA_Obj A, FLA_Obj B );

```

Purpose: Perform one of the following triangular solves with multiple right-hand sides (TRSM):

$$\begin{array}{ll}
AX &= \alpha B & XA &= \alpha B \\
A^T X &= \alpha B & XA^T &= \alpha B \\
\bar{A}X &= \alpha B & X\bar{A} &= \alpha B \\
A^H X &= \alpha B & XA^H &= \alpha B
\end{array}$$

and overwrite B with the contents of the solution matrix X as follows:

$$\begin{array}{ll}
B &:= \alpha A^{-1} B & B &:= \alpha B A^{-1} \\
B &:= \alpha A^{-T} B & B &:= \alpha B A^{-T} \\
B &:= \alpha \bar{A}^{-1} B & B &:= \alpha B \bar{A}^{-1} \\
B &:= \alpha A^{-H} B & B &:= \alpha B A^{-H}
\end{array}$$

where α is a scalar, A is a triangular matrix, and X and B are general matrices. The **side** argument indicates whether the triangular matrix A is multiplied on the left or the right side of X . The **uplo** argument indicates whether the lower or upper triangle of A is referenced by the operation. The **trans** argument allows the computation to proceed as if A were conjugated and/or transposed. The **diag** argument indicates whether the diagonal of A is unit or non-unit.

Constraints:

- The numerical datatypes of A and B must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α is not of datatype `FLA_CONSTANT`, then it must match the datatypes of A and B .
- If **side** equals `FLA_LEFT`, then the number of rows in B and the order of A must be equal; otherwise, if **side** equals `FLA_RIGHT`, then the number of columns in B and the order of A must be equal.
- **diag** may not be `FLA_ZERO_DIAG`.

Int. Notes: `FLA_Trmm()` expects A and B to be flat matrix objects.

Imp. Notes: `FLA_Trsm()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS. `FLASH_Trsm()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the TRSM operation into subproblems expressed in terms of individual blocks of A and B and then invokes external BLAS to perform the computation on these blocks.

Arguments:

- | | | |
|--------------|---|--|
| side | – | Indicates whether A is multiplied on the left or right side of X . |
| uplo | – | Indicates whether the lower or upper triangle of A is referenced during the operation. |
| trans | – | Indicates whether the operation proceeds as if A were conjugated and/or transposed. |
| diag | – | Indicates whether the diagonal of A is unit or non-unit. |
| alpha | – | An <code>FLA_Obj</code> representing scalar α . |
| A | – | An <code>FLA_Obj</code> representing matrix A . |
| B | – | An <code>FLA_Obj</code> representing matrix B . |

5.7.2 LAPACK operations

```
FLA_Error FLA_Chol( FLA_Uplo uplo, FLA_Obj A );
FLA_Error FLASH_Chol( FLA_Uplo uplo, FLA_Obj A );
```

Purpose: Perform one of the following Cholesky factorizations (CHOL):

$$\begin{aligned} A &\rightarrow LL^T \\ A &\rightarrow U^T U \\ A &\rightarrow LL^H \\ A &\rightarrow U^H U \end{aligned}$$

where A is positive definite. If A is real, then it is assumed to be symmetric; otherwise, if A is complex, then it is assumed to be Hermitian. The operation references and then overwrites the lower or upper triangle of A with the Cholesky factor L or U , depending on the value of `uplo`.

Returns: FLA_SUCCESS if the operation is successful; otherwise, if A is not positive definite, a signed integer corresponding to the row/column index at which the algorithm detected a negative or non-real entry along the diagonal.

Constraints:

- The numerical datatype of A must be floating-point, and must not be FLA_CONSTANT.
- A must be square.

Int. Notes: FLA_Chol() expects A to be a flat matrix object.

Imp. Notes: FLA_Chol() invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS and LAPACK routines. FLASH_Chol() uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the CHOL operation into subproblems expressed in terms of individual blocks of A and then invokes external BLAS and LAPACK routines to perform the computation on these blocks.

Arguments:

- | | |
|-------------------|--|
| <code>uplo</code> | – Indicates whether the lower or upper triangle of A is referenced and overwritten during the operation. |
| <code>A</code> | – An FLA_Obj representing matrix A . |


```
FLA_Error FLA_LU_nopiv( FLA_Obj A );
FLA_Error FLASH_LU_nopiv( FLA_Obj A );
```

Purpose: Perform an LU factorization without pivoting (LUNOPIV):

$$A \rightarrow LU$$

where A is a general matrix, L is lower triangular (or lower trapezoidal if $m > n$) with a unit diagonal, and U is upper triangular (or upper trapezoidal if $m < n$). The operation overwrites the strictly lower triangular portion of A with L and the upper triangular portion of A with U . The diagonal elements of L are not stored.

Notes: The algorithms used by `FLA_LU_nopiv_blk_external()` and `FLA_LU_nopiv_unb_external()` do not perform pivoting and are therefore numerically unstable. Almost all applications should use `FLA_LU_piv_blk_external()` or `FLA_LU_piv_unb_external()` instead.

Returns: `FLA_SUCCESS` if the operation is successful; otherwise, if A is singular, a signed integer corresponding to the row/column index at which the algorithm detected a zero entry along the diagonal.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.
- A must be square.

Int. Notes: `FLA_LU_nopiv()` expects A to be a flat matrix object.

Imp. Notes: `FLA_LU_nopiv()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS and LAPACK routines. `FLASH_LU_nopiv()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the LUNOPIV operation into subproblems expressed in terms of individual blocks of A and then invokes external BLAS and LAPACK routines to perform the computation on these blocks.

Arguments:

A — An `FLA_Obj` representing matrix A .

```
FLA_Error FLA_LU_piv( FLA_Obj A, FLA_Obj p );
```

Purpose: Perform an LU factorization with partial row pivoting (LUPIV):

$$A \rightarrow PLU$$

where A is a general matrix, L is lower triangular (or lower trapezoidal if $m > n$) with a unit diagonal, U is upper triangular (or upper trapezoidal if $m < n$), and P is a permutation matrix, which is encoded into the pivot vector p . The operation overwrites the strictly lower triangular portion of A with L and the upper triangular portion of A with U . The diagonal elements of L are not stored.

Notes: `FLA_LU_piv()` fills the pivot vector p differently than the LAPACK routines `?getrf()` and `?getf2()`. The latter routines fill the vector to indicate that row i of matrix A was permuted with row p_i . By contrast, the `libflame` routines fill the vector to indicate that row i of matrix A was permuted with row $i + p_i$. In other words, an index value stored within the `libflame` pivot vector indicates a row swap *relative* to the current index, while the corresponding LAPACK pivot vector contains *absolute* row indices (ie: relative to the first row). A secondary difference is that the LAPACK routines store index values ranging from 1 to $\min(m, n)$ while the corresponding `libflame` routines store indices ranging from 0 to $\min(m, n) - 1$. The user may convert back and forth between `libflame` and LAPACK-style pivot indices using the routine `FLA_Shift_pivots_to()`.

Returns: `FLA_SUCCESS` if the operation is successful; otherwise, if A is singular, a signed integer corresponding to the row/column index at which the algorithm detected a zero entry along the diagonal.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.
- The numerical datatype of p must be `FLA_INT`.
- The length of p must be $\min(m, n)$.

Int. Notes: `FLA_LU_piv()` expects A to be a flat matrix object.

Imp. Notes: `FLA_LU_piv()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS and LAPACK routines.

Arguments:

- | | |
|---|---|
| A | – An <code>FLA_Obj</code> representing matrix A . |
| p | – An <code>FLA_Obj</code> representing vector p . |

```
FLA_Error FLASH_LU_incpiv( FLA_Obj A, FLA_Obj p, FLA_Obj L );
```

Purpose: Perform an LU factorization with incremental pivoting (LUINCPIV). The operation is similar to that of LU with partial row pivoting, except that the algorithm is SuperMatrix-aware. As a consequence, the arguments must be hierarchical objects.

Notes: It is *highly* recommended that the user create and initialize a flat object containing the matrix to be factorized and then call `FLASH_LU_incpiv_create_hier_matrices()` to create hierarchical matrices A , p , and L from the original flat matrix.

Int. Notes: In addition to the input matrix A and pivot vector p , the function requires an additional object L , which stores interim matrices that are used in a subsequent forward substitution.

Caveats: Currently, this function only supports matrices with hierarchical depths of exactly 1.

Returns: `FLA_SUCCESS` if the operation is successful; otherwise, if A is singular, a signed integer corresponding to the row/column index at which the algorithm detected a zero entry along the diagonal.

Constraints:

- The numerical datatypes of A and L must be floating-point, and must not be `FLA_CONSTANT`.
- The numerical datatype of p must be `FLA_INT`.
- A must be square.

Arguments:

A	–	A hierarchical <code>FLA_Obj</code> representing matrix A .
p	–	A hierarchical <code>FLA_Obj</code> representing vector p .
L	–	A hierarchical <code>FLA_Obj</code> representing matrix L .

```
void FLASH_FS_incpiv( FLA_Obj A, FLA_Obj p, FLA_Obj L_inter, FLA_Obj b );
```

Purpose: Perform a forward substitution with the unit lower triangular L factor (residing in the lower triangle of hierarchical matrix A) and a right-hand side vector b , overwriting b with an intermediate vector y .

$$y := L^{-1}b$$

The matrix p contains the incremental pivot vectors that were used during the LU factorization with incremental pivoting performed via `FLASH_LU_incpiv()`. The matrix L_{inter} contains intermediate lower triangular factors computed during the factorization, which are reused in the forward substitution. Note that p and L_{inter} are hierarchical, and provided by `FLASH_LU_incpiv()`.

Constraints:

- The numerical datatypes of A , L_{inter} , and b must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The numerical datatype of p must be `FLA_INT`.
- A must be square.

Imp. Notes: `FLASH_FS_incpiv()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the operation into subproblems expressed in terms of individual blocks of A , p , L_{inter} , and b and then invokes external BLAS routines to perform the computation on these blocks.

Caveats: `FLASH_FS_incpiv()` currently only works for hierarchical matrices of depth 1 where A refers to a single storage block.

Arguments:

- | | |
|-------------|---|
| A | – A hierarchical <code>FLA_Obj</code> representing matrix A . |
| p | – A hierarchical <code>FLA_Obj</code> representing matrix p . |
| L_{inter} | – A hierarchical <code>FLA_Obj</code> representing matrix L_{inter} . |
| b | – A hierarchical <code>FLA_Obj</code> representing vector b . |

```
void FLA_Ttmm( FLA_Uplo uplo, FLA_Obj A );
void FLASH_Ttmm( FLA_Uplo uplo, FLA_Obj A );
```

Purpose: Perform one of the following triangular-transpose matrix multiplies (TTMM):

$$\begin{aligned} A &:= L^T L \\ A &:= U U^T \\ A &:= L^H L \\ A &:= U U^H \end{aligned}$$

where A is a triangular matrix with a real diagonal. The operation references and then overwrites the lower or upper triangle of A with its inverse, A^{-1} , depending on the value of `uplo`. The `diag` argument indicates whether the diagonal of A is unit or non-unit.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.
- `diag` may not be `FLA_ZERO_DIAG`.
- A must be square.

Int. Notes: `FLA_Ttmm()` expects A to be a flat matrix object.

Imp. Notes: `FLA_Ttmm()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS and LAPACK routines. `FLASH_Ttmm()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the TTMM operation into subproblems expressed in terms of individual blocks of A and then invokes external BLAS and LAPACK routines to perform the computation on these blocks.

Arguments:

- | | |
|-------------------|--|
| <code>uplo</code> | – Indicates whether the lower or upper triangle of A is referenced and overwritten during the operation. |
| <code>diag</code> | – Indicates whether the diagonal of A is unit or non-unit. |
| <code>A</code> | – An <code>FLA_Obj</code> representing matrix A . |

```
FLA_Error FLA_Trinv( FLA_Uplo uplo, FLA_Diag diag, FLA_Obj A );
FLA_Error FLASH_Trinv( FLA_Uplo uplo, FLA_Diag diag, FLA_Obj A );
```

Purpose: Perform a triangular matrix inversion (TRINV):

$$A := A^{-1}$$

where A is a general triangular matrix. The operation references and then overwrites the lower or upper triangle of A with its inverse, A^{-1} , depending on the value of `uplo`. The `diag` argument indicates whether the diagonal of A is unit or non-unit.

Returns: FLA_SUCCESS if the operation is successful; otherwise, if A is singular, a signed integer corresponding to the row/column index at which the algorithm detected a zero entry along the diagonal.

Constraints:

- The numerical datatype of A must be floating-point, and must not be FLA_CONSTANT.
- `diag` may not be FLA_ZERO_DIAG.
- A must be square.

Int. Notes: FLA_Trinv() expects A to be a flat matrix object.

Imp. Notes: FLA_Trinv() invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS and LAPACK routines. FLASH_Trinv() uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the TRINV operation into subproblems expressed in terms of individual blocks of A and then invokes external BLAS and LAPACK routines to perform the computation on these blocks.

Arguments:

<code>uplo</code>	– Indicates whether the lower or upper triangle of A is referenced and overwritten during the operation.
<code>diag</code>	– Indicates whether the diagonal of A is unit or non-unit.
<code>A</code>	– An FLA_Obj representing matrix A .

```
void FLA_SPDinv( FLA_Uplo uplo, FLA_Obj A );
void FLASH_SPDinv( FLA_Uplo uplo, FLA_Obj A );
```

Purpose: Perform a positive definite matrix inversion (SPDINV):

$$A := A^{-1}$$

where A is positive definite. If A is real, then it is assumed to be symmetric; otherwise, if A is complex, then it is assumed to be Hermitian. The operation references and then overwrites the lower or upper triangle of A with its inverse, A^{-1} , depending on the value of `uplo`.

Notes: Given a real symmetric positive definite matrix A , there exists a factor L such that $A = LL^T$. Therefore,

$$\begin{aligned} A^{-1} &= (LL^T)^{-1} \\ &= L^{-T}L^{-1} \end{aligned}$$

Similarly, for a complex Hermitian positive definite matrix A , there exists a factor such that $A = LL^H$:

$$\begin{aligned} A^{-1} &= (LL^H)^{-1} \\ &= L^{-H}L^{-1} \end{aligned}$$

From this, we observe that the inverse of symmetric positive definite matrices may be computed by multiplying the inverse of the the Cholesky factor L by its transpose, or in the case of Hermitian positive definite matrices, its conjugate-transpose. Similar observations may be made provided $L = U^T$ and $L = U^H$ for real and complex matrices, respectively.

Returns: If A is not positive definite, then `FLASH_SPDinv()` will return the row/column index at which the algorithm detected a negative or non-real entry along the diagonal. If the Cholesky factorization of A succeeds but the Cholesky factor is found to be singular, then `FLASH_SPDinv()` will return the row/column index at which the algorithm detected a zero entry along the diagonal. Otherwise, `FLASH_SPDinv()` returns `FLA_SUCCESS` if the operation is successful.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.
- A must be square.

Int. Notes: `FLA_SPDinv()` expects A to be a flat matrix object.

Imp. Notes: `FLA_SPDinv()` is implemented in terms of `FLA_Chol()`, `FLA_Trinv()`, and `FLA_Ttmm()`. `FLASH_SPDinv()` is implemented in terms of `FLASH_Chol()`, `FLASH_Trinv()`, and `FLASH_Ttmm()`.

Arguments:

- | | |
|-------------------|--|
| <code>uplo</code> | – Indicates whether the lower or upper triangle of A is referenced and overwritten during the operation. |
| <code>diag</code> | – Indicates whether the diagonal of A is unit or non-unit. |
| <code>A</code> | – An <code>FLA_Obj</code> representing matrix A . |

```
void FLA_Hess( FLA_Obj A, FLA_Obj t, int ilo, int ihi );
```

Purpose: Perform a reduction to upper Hessenberg form operation (HESS) via Householder transformations such that:

$$A = QHQ^T$$

where A is a real matrix, Q is an orthogonal matrix, and H is an upper Hessenberg matrix. Matrix Q is expressed as a product of $(i_{hi} - i_{lo})$ Householder reflectors:

$$Q = H(i_{lo})H(i_{lo} + 1) \dots H(i_{hi} - 1)$$

Each $H(i)$ has the form

$$H(i) = I - \tau vv^T$$

where τ is a real scalar and v is a real vector of length n . If ν_j is the j th element of v , we may describe v such that, for a given $H(i)$, the element $\nu_{i+1} = 1$ while elements $\nu_{0:i}$ and $\nu_{i_{hi}+1:n-1}$ are zero, with other entries holding non-zero values. The operation overwrites the the upper triangle and first subdiagonal of A with H . However, the matrix Q is not stored explicitly. Instead, the operation stores the τ associated with $H(i)$ to the i th element of vector t , and also stores the non-unit, non-zero entries $\nu_{i+2:i_{hi}}$ of Householder reflectors $H_{i_{lo}}$ through $H_{i_{hi}-2}$ to the elements below the first subdiagonal of A . More specifically, entries $\nu_{i+2:i_{hi}}$ are stored to elements $i + 2 : i_{hi}$ of the i th column of matrix A .

Constraints:

- The numerical datatypes of A and t must be identical and floating-point, and must not be FLA_CONSTANT. **Currently, only real matrices are supported by this function.**
- A must be square.
- The vector t must have at least length $n - 1$ where n is the order of A .
- If $n > 0$, then $0 \leq i_{lo} \leq i_{hi} \leq n - 1$. If $n = 0$, $i_{lo} = 0$ and $i_{hi} = -1$.

Int. Notes: FLA_Hess() expects A to be a flat matrix object.

Imp. Notes: FLA_Hess() is not yet implemented as a front-end function.

Arguments:

- | | |
|-----|---|
| A | – An FLA_Obj representing matrix A . |
| t | – An FLA_Obj representing vector t . |
| ilo | – An integer representing the index of the first column of A to reference.
Most users should use 0 here. |
| ihi | – An integer representing the index of the first column of A to reference.
Most users should use $n - 1$ here, where n is the order of A . |


```
void FLA_QR_UT( FLA_Obj A, FLA_Obj T );
```

Purpose: Perform a QR factorization with the UT transform (QRUT). The resulting vectors stored below the diagonal of A should only be used with other UT transform-aware operations.

Constraints:

- The numerical datatypes of A and T must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The width of T and the width of A must be equal.

Int. Notes: `FLA_QR_UT()` expects A and T to be a flat matrix objects.

Imp. Notes: `FLA_QR_UT()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS and LAPACK routines. The algorithmic blocksize b is ultimately determined by the length of T . When in doubt, create T via `FLA_QR_UT_create.T()`.

Arguments:

A	–	An <code>FLA_Obj</code> representing matrix A .
T	–	An <code>FLA_Obj</code> representing matrix T .

```
void FLASH_QR_UT_inc( FLA_Obj A, FLA_Obj TW );
```

Purpose: Perform an incremental QR factorization using the UT transform (QRUTINC). The operation is similar to that of QR factorization with the UT transform, except that the algorithm is SuperMatrix-aware. As a consequence, the arguments must be hierarchical objects.

Notes: It is *highly* recommended that the user create and initialize a flat object containing the matrix to be factorized and then call `FLASH_QR_UT_inc_create_hier_matrices()` to create hierarchical matrices A and TW from the original flat matrix.

Int. Notes: In addition to the input matrix A , the function requires an additional matrix TW to hold the interim triangular block Householder transformations computed for each storage block. These transformations are used when applying Q (via `FLASH_Apply_Q_UT_inc()`). The matrix TW also contains temporary workspace needed by the incremental QR algorithm.

Imp. Notes: Strictly speaking, the blocks in the lower triangle (including the diagonal) of TW are used to store the block Householder transformations corresponding to T in `FLA_QR_UT()` while the blocks in the upper triangle of TW are used as workspace only.

Caveats: Currently, this function only supports matrices with hierarchical depths of exactly 1.

Constraints:

- The numerical datatypes of A and TW must be floating-point, and must not be `FLA_CONSTANT`.
- A must be square.
- A and TW must each have the same number of blocks in the row and column dimensions.

Arguments:

A	–	A hierarchical <code>FLA_Obj</code> representing matrix A .
TW	–	A hierarchical <code>FLA_Obj</code> representing matrix TW .

```
void FLA_LQ_UT( FLA_Obj A, FLA_Obj T );
```

Purpose: Perform an LQ factorization with the UT transform (LQUT). The resulting vectors stored above the diagonal of A should only be used with other UT transform-aware operations.

Constraints:

- The numerical datatypes of A and T must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The length of T and the length of A must be equal.

Int. Notes: `FLA_LQ_UT()` expects A and T to be a flat matrix objects.

Imp. Notes: `FLA_LQ_UT()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS and LAPACK routines. The algorithmic blocksize b is ultimately determined by the width of T . When in doubt, create T via `FLA_LQ_UT_create.T()` .

Arguments:

- | | | |
|-----|---|---|
| A | – | An <code>FLA_Obj</code> representing matrix A . |
| T | – | An <code>FLA_Obj</code> representing matrix T . |

```
void FLA_Apply_Q_UT( FLA_Side side, FLA_Trans trans, FLA_Store storev,
                    FLA_Obj A, FLA_Obj T, FLA_Obj W, FLA_Obj B );
```

Purpose: Apply a matrix Q (or Q^T or Q^H) to a general matrix B from either the left or the right (APQUT):

$$\begin{array}{ll} B &:= QB & B &:= BQ \\ B &:= Q^T B & B &:= BQ^T \\ B &:= Q^H B & B &:= BQ^H \end{array}$$

where Q is the orthogonal (or, if A is complex, unitary) matrix implicitly defined by the Householder vectors stored in matrix A and the block Householder transforms stored in matrix T by `FLA_QR_UT()`. Matrix W is used as workspace. The `side` argument indicates whether Q is applied to B from the left or the right. The `trans` argument indicates whether Q or Q^T (or Q^H) is applied to B . The `storev` argument indicates whether the Householder vectors which define Q are stored column-wise (in the strictly lower triangle) or row-wise (in the strictly upper triangle) of A .

Constraints:

- The numerical datatypes of A , T , W , and B must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If `side` equals `FLA_LEFT`, then the number of rows in B and the order of A must be equal; otherwise, if `side` equals `FLA_RIGHT`, then the number of columns in B and the order of A must be equal.
- If A is real, then `trans` must be `FLA_NO_TRANSPOSE` or `FLA_TRANSPOSE`; otherwise if A is complex, then `trans` must be `FLA_NO_TRANSPOSE` or `FLA_CONJ_TRANSPOSE`.
- The dimensions of W must be $m_T \times n_B$ where m_T is the number of rows in T and n_B is the number of columns in B .

Int. Notes: `FLA_Apply_Q_UT()` expects A , T , W , and B to be a flat matrix objects.

Imp. Notes: `FLA_Apply_Q_UT()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS and LAPACK routines.

Caveats: `FLA_Apply_Q_UT()` is currently only implemented for the case where `side` is `FLA_LEFT`, `trans` is `FLA_TRANSPOSE` or `FLA_CONJ_TRANSPOSE`, and `storev` is `FLA_COLUMNWISE`.

Arguments:

- | | |
|---------------------|---|
| <code>side</code> | – Indicates whether Q (or Q^T or Q^H) is multiplied on the left or right side of B . |
| <code>trans</code> | – Indicates whether the operation proceeds as if Q were transposed (or conjugate-transposed). |
| <code>storev</code> | – Indicates whether the vectors stored within A are stored column-wise or row-wise. |
| <code>A</code> | – An <code>FLA_Obj</code> representing matrix A . |
| <code>T</code> | – An <code>FLA_Obj</code> representing matrix T . |
| <code>W</code> | – An <code>FLA_Obj</code> representing matrix W . |
| <code>B</code> | – An <code>FLA_Obj</code> representing matrix B . |

```
void FLASH_Apply_Q_UT_inc( FLA_Side side, FLA_Trans trans, FLA_Store storev,
                          FLA_Obj A, FLA_Obj TW, FLA_Obj W, FLA_Obj B );
```

Purpose: Apply a matrix Q (or Q^T or Q^H) to a general matrix B from either the left or the right (APQUTINC):

$$\begin{aligned} B &:= QB & B &:= BQ \\ B &:= Q^T B & B &:= BQ^T \\ B &:= Q^H B & B &:= BQ^H \end{aligned}$$

where Q is the orthogonal (or, if A is complex, unitary) matrix implicitly defined by the Householder vectors stored in matrix A and the block Householder transforms stored in matrix TW by `FLASH_QR_UT_inc()`. Matrix W is used as workspace. The `side` argument indicates whether Q is applied to B from the left or the right. The `trans` argument indicates whether Q or Q^T (or Q^H) is applied to B . The `storev` argument indicates whether the Householder vectors which define Q are stored column-wise (in the strictly lower triangle) or row-wise (in the strictly upper triangle) of A .

Constraints:

- The numerical datatypes of A , TW , W , and B must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If `side` equals `FLA_LEFT`, then the number of rows in B and the order of A must be equal; otherwise, if `side` equals `FLA_RIGHT`, then the number of columns in B and the order of A must be equal.
- If A is real, then `trans` must be `FLA_NO_TRANSPOSE` or `FLA_TRANSPOSE`; otherwise if A is complex, then `trans` must be `FLA_NO_TRANSPOSE` or `FLA_CONJ_TRANSPOSE`.
- The dimensions of W must be $m_{TW} \times n_B$ where m_{TW} is the scalar length of a single block of TW and n_B is the scalar width of B .

Imp. Notes: `FLASH_Apply_Q_UT_inc()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the APQUTINC operation into subproblems expressed in terms of individual blocks of A , TW , W , and B and then invokes external BLAS routines to perform the computation on these blocks.

Caveats: `FLASH_Apply_Q_UT_inc()` currently only works for hierarchical matrices of depth 1 where A refers to a single storage block. `FLASH_Apply_Q_UT_inc()` is currently only implemented for the case where `side` is `FLA_LEFT`, `trans` is `FLA_TRANSPOSE` or `FLA_CONJ_TRANSPOSE`, and `storev` is `FLA_COLUMNWISE`.

Arguments:

- | | |
|---------------------|---|
| <code>side</code> | – Indicates whether Q (or Q^T or Q^H) is multiplied on the left or right side of B . |
| <code>trans</code> | – Indicates whether the operation proceeds as if Q were transposed (or conjugate-transposed). |
| <code>storev</code> | – Indicates whether the vectors stored within A are stored column-wise or row-wise. |
| <code>A</code> | – An <code>FLA_Obj</code> representing matrix A . |
| <code>TW</code> | – An <code>FLA_Obj</code> representing matrix TW . |
| <code>W</code> | – An <code>FLA_Obj</code> representing matrix W . |
| <code>B</code> | – An <code>FLA_Obj</code> representing matrix B . |

```

void FLA_Sylv( FLA_Trans transa, FLA_Trans transb, FLA_Obj isgn,
               FLA_Obj A, FLA_Obj B, FLA_Obj C, FLA_Obj scale );
void FLASH_Sylv( FLA_Trans transa, FLA_Trans transb, FLA_Obj isgn,
                 FLA_Obj A, FLA_Obj B, FLA_Obj C, FLA_Obj scale );

```

Purpose: Solve one of the following triangular Sylvester equations (SYLV):

$$\begin{aligned}
AX &\pm XB &= C \\
AX &\pm XB^T &= C \\
A^T X &\pm XB &= C \\
A^T X &\pm XB^T &= C
\end{aligned}$$

where A and B are real upper triangular matrices and C is a real general matrix. If A , B , and C are complex matrices, then the possible operations are:

$$\begin{aligned}
AX &\pm XB &= C \\
AX &\pm XB^H &= C \\
A^H X &\pm XB &= C \\
A^H X &\pm XB^H &= C
\end{aligned}$$

where A and B are complex upper triangular matrices and C is a complex general matrix. The operation references and then overwrites matrix C with the solution matrix X . The `isgn` argument is a scalar integer object that indicates whether the \pm sign between terms is a plus or a minus. The `scale` argument is not referenced and set to 1.0 upon completion.

Constraints:

- The numerical datatypes of A , B , and C must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The `isgn` argument must be either `FLA_ONE` or `FLA_MINUS_ONE`.
- The numerical datatype of `scale` must not be `FLA_CONSTANT`. Furthermore, the precision of the datatype of `scale` must be equal to that of A , B , and C .
- A and B must be square.
- The order of A and the order of B must be equal to the number of rows in C and the number of columns in C , respectively.

Int. Notes: `FLA_Sylv()` expects A , B , and C to be flat matrix objects.

Imp. Notes: `FLA_Sylv()` invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS and LAPACK routines. `FLASH_Sylv()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the SYLV operation into subproblems expressed in terms of individual blocks of A , B , and C and then invokes external BLAS and LAPACK routines to perform the computation on these blocks.

Arguments:

<code>transa</code>	–	Indicates whether the operation proceeds as if A were [conjugate] transposed.
<code>transb</code>	–	Indicates whether the operation proceeds as if B were [conjugate] transposed.
<code>isgn</code>	–	Indicates whether the terms of the Sylvester equation are added or subtracted.
<code>A</code>	–	An <code>FLA_Obj</code> representing matrix A .
<code>B</code>	–	An <code>FLA_Obj</code> representing matrix B .
<code>C</code>	–	An <code>FLA_Obj</code> representing matrix C .
<code>scale</code>	–	Not referenced; set to 1.0 upon exit.

5.7.3 Utility functions

```
void FLA_Apply_pivots( FLA_Side side, FLA_Trans trans, FLA_Obj p, FLA_Obj A );
```

Purpose: Apply a permutation matrix P to a matrix A .

$$\begin{aligned} A &:= PA \\ A &:= P^T A \\ A &:= AP \\ A &:= AP^T \end{aligned}$$

where A is a general matrix and P is a permutation matrix corresponding to the pivot vector p .

Notes: The pivot vector p must contain pivot values that conform to `libflame` pivot indexing. If the pivot vector was filled using an LAPACK routine, it must first be converted to `libflame` pivot indexing with `FLA_Shift_pivots_to()` before it may be used with `FLA_Swap_rows()`. Please see the description for `FLA_LU_piv()` in Section 5.7.2 for details on the differences between LAPACK-style pivot vectors and `libflame` pivot vectors.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.
- The numerical datatype of p must be `FLA_INT`.

Imp. Notes: This function is currently only implemented for applying P from the left (ie: `side` equal to `FLA_LEFT` and `trans` equal to `FLA_NO_TRANSPOSE`).

Arguments:

- | | | |
|--------------------|---|--|
| <code>side</code> | – | Indicates whether the operation proceeds as if the permutation matrix P is applied from the left or the right. |
| <code>trans</code> | – | Indicates whether the operation proceeds as if the permutation matrix P were transposed. |
| <code>p</code> | – | An <code>FLA_Obj</code> representing vector p . |
| <code>A</code> | – | An <code>FLA_Obj</code> representing matrix A . |

```
void FLA_Shift_pivots_to( FLA_Pivot_type ptype, FLA_Obj p );
```

Purpose: Convert a pivot vector from `libflame` pivot indexing to LAPACK indexing, or vice versa. If p currently contains `libflame` pivots, setting `ptype` to `FLA_LAPACK_PIVOTS` will update the contents of p to reflect the pivoting style found in LAPACK. Likewise, if p currently contains LAPACK pivots, setting `ptype` to `FLA_NATIVE_PIVOTS` will update the contents of p to reflect the pivoting style used natively within `libflame`.

Notes: The user should always be aware of the current state of the indexing style used by p . There is nothing stopping the user from applying the shift in the wrong direction. For example, attempting to shift the pivot format from `libflame` to LAPACK when the vector already uses LAPACK pivot indexing will result in an undefined format. Please see the description for `FLA_LU_piv()` in Section 5.7.2 for details on the differences between LAPACK-style pivot vectors and `libflame` pivot vectors.

Constraints:

- The numerical datatype of p must be integer, and must not be `FLA_CONSTANT`.

Arguments:

- | | | |
|--------------------|---|---|
| <code>ptype</code> | – | Indicates the desired pivot indexing. |
| <code>p</code> | – | An <code>FLA_Obj</code> representing vector p . |

```
void FLA_Form_perm_matrix( FLA_Obj p, FLA_Obj A );
```

Purpose: Explicitly form a permutation matrix P from a pivot vector p and then store the contents of P into A .

Notes: This function assumes that p uses native `libflame` pivots. Please see the description for `FLA_LU_piv()` in Section 5.7.2 for details on the differences between LAPACK-style pivot vectors and `libflame` pivot vectors.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.
- The numerical datatype of p must be integer, and must not be `FLA_CONSTANT`.
- A must be square.
- The number of rows in p must be equal to the order of A .

Imp. Notes: This function is currently implemented as:

```
FLA_Obj_set_to_identity( A );
FLA_Apply_pivots( FLA_LEFT, FLA_NO_TRANSPOSE, p, A );
```

Arguments:

p – An `FLA_Obj` representing vector p .
 A – An `FLA_Obj` representing matrix A .

```
void FLA_Househ2( FLA_Obj chi_1, FLA_Obj x2, FLA_Obj beta );
```

Purpose: Compute the Householder transform,

$$\left(I - \beta \begin{pmatrix} 1 \\ u_2 \end{pmatrix} \begin{pmatrix} 1 & u_2^H \end{pmatrix} \right)$$

by computing u_2 and β such that

$$\left(I - \beta \begin{pmatrix} 1 \\ u_2 \end{pmatrix} \begin{pmatrix} 1 & u_2^H \end{pmatrix} \right)^H \begin{pmatrix} \chi_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \gamma \|x\|_2 \\ 0 \end{pmatrix}$$

where:

$$\begin{aligned} x &= \begin{pmatrix} \chi_1 \\ x_2 \end{pmatrix} \\ \beta &= \frac{2}{(1 + u_2^H u_2)} \\ \gamma &= -\text{sign}(\text{Re}(\chi_1)) \end{aligned}$$

Upon completion, $\gamma \|x\|_2$ has overwritten χ_1 and u_2 has overwritten x_2

Imp. Notes: This function uses external implementations of the level-1 BLAS routines `*nrm2()`, `*scal()`, `?dot()`, `?dotu()`, and `?dotc()`.

Dev. notes: This function is a prototype and implemented only for matrices of datatype `FLA_DOUBLE`. It will be re-implemented in the future.

Arguments:

χ_1 – An `FLA_Obj` representing scalar χ_1 .
 x_2 – An `FLA_Obj` representing vector x_2 .
 β – An `FLA_Obj` representing scalar β .

```
void FLA_Househ2_UT( FLA_Obj chi_1, FLA_Obj x2, FLA_Obj tau );
```

Purpose: Compute the UT transform,

$$\left(I - \frac{1}{\tau} \begin{pmatrix} 1 \\ u_2 \end{pmatrix} \begin{pmatrix} 1 & u_2^H \end{pmatrix} \right)$$

by computing u_2 and τ such that

$$\left(I - \frac{1}{\tau} \begin{pmatrix} 1 \\ u_2 \end{pmatrix} \begin{pmatrix} 1 & u_2^H \end{pmatrix} \right)^H \begin{pmatrix} \chi_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \gamma \|x\|_2 \\ 0 \end{pmatrix}$$

where:

$$\begin{aligned} x &= \begin{pmatrix} \chi_1 \\ x_2 \end{pmatrix} \\ \tau &= \frac{(1 + u_2^H u_2)}{2} \\ \gamma &= -\text{sign}(\text{Re}(\chi_1)) \end{aligned}$$

For complex Householder transforms, τ is computed from real $\gamma \|x\|_2$ and complex $\hat{\chi}_1$ as

$$\tau = \left(\frac{\gamma \|x\|_2 - \hat{\chi}_1}{\gamma \|x\|_2} \right)^{-1}$$

where $\hat{\chi}_1$ is the original value of χ_1 . Upon completion, $\gamma \|x\|_2$ has overwritten χ_1 and u_2 has overwritten x_2 .

Imp. Notes: This function uses external implementations of the level-1 BLAS routines `*nrm2()`, `*scal()`, and `?dot()`.

Arguments:

- `chi_1` – An `FLA_Obj` representing scalar χ_1 .
- `x2` – An `FLA_Obj` representing vector x_2 .
- `tau` – An `FLA_Obj` representing scalar τ .


```
void FLA_Accum_T_UT( FLA_Direct direct, FLA_Store storev,
                    FLA_Obj V, FLA_Obj t, FLA_Obj T );
```

Purpose: Compute one or more triangular factors T_j of a block Householder transform H_j from a set of Householder reflectors, which were computed via the UT transform. The Householder reflectors are given via the Householder vectors stored in the strictly lower triangle of the $m \times n$ matrix V and the τ scalar factors in vector t of length $k = \min(m, n)$. The triangular factors T_j are stored horizontally within a $b \times k$ matrix T as:

$$T = (T_0 \mid T_1 \mid \cdots \mid T_{p-1})$$

where $p = \lceil k/b \rceil$. All factors T_j are $b \times b$, except T_{p-1} which may be smaller if the remainder of k/b is nonzero.

Notes: Each reflector is defined as $H(i) = I - \tau v v^T$, where τ is a scalar stored at the i th element of vector t and v is a vector stored in matrix V . If **direct** is **FLA_FORWARD**, then H is the forward product of k Householder reflectors, $H(0)H(1)\dots H(k-1)$, and T is upper triangular upon completion. If **direct** is **FLA_BACKWARD**, then H is the backward product of k Householder reflectors, $H(k-1)\dots H(1)H(0)$, and T is lower triangular upon completion. If **storev** is **FLA_COLUMNWISE**, the vector which defines reflector $H(i)$ is assumed to be stored in the i th column of V , and $H = I - VTV^T$, where the order of H is equal to the number of rows in V . If **storev** is **FLA_ROWWISE**, the vector which defines reflector $H(i)$ is assumed to be stored in the i th row of V , and $H = I - V^T T V$, where the order of H is equal to the number of columns in V . The dimensions and storage layout of V depend on the values of **direct** and **storev**, which should be set according to how V was filled. The following example, with $k = 3$ Householder reflectors and H of order $n = 5$, illustrates the possible storage schemes for matrix V .

		storev	
		FLA_COLUMNWISE	FLA_ROWWISE
direct	FLA_FORWARD	$\begin{pmatrix} 1 & & & & \\ \nu_0 & 1 & & & \\ \nu_0 & \nu_1 & 1 & & \\ \nu_0 & \nu_1 & \nu_2 & & \\ \nu_0 & \nu_1 & \nu_2 & & \end{pmatrix}$	$\begin{pmatrix} 1 & \nu_0 & \nu_0 & \nu_0 & \nu_0 \\ & 1 & \nu_1 & \nu_1 & \nu_1 \\ & & 1 & \nu_2 & \nu_2 \\ & & & & \\ & & & & \end{pmatrix}$
	FLA_BACKWARD	$\begin{pmatrix} \nu_0 & \nu_1 & \nu_2 \\ \nu_0 & \nu_1 & \nu_2 \\ 1 & \nu_1 & \nu_2 \\ & 1 & \nu_2 \\ & & 1 \end{pmatrix}$	$\begin{pmatrix} \nu_0 & \nu_0 & 1 & & \\ \nu_1 & \nu_1 & \nu_1 & 1 & \\ \nu_2 & \nu_2 & \nu_2 & \nu_2 & 1 \end{pmatrix}$

Here, elements ν_j for some constant j all belong to the same vector v that defines the Householder reflector $H(i)$. Note that the unit diagonal elements are not stored, and the rest of the matrix is not referenced.

Notes: This function should only be used with matrices V and vectors t that were filled by other UT transform-aware operations, such as **FLA_QR_UT()** and **FLA_QR_UT_recover_tau()**.

Int. Notes: Since **FLA_QR_UT()** and **FLA_LQ_UT()** provide T upon return, this routine is rarely needed. However, there may be occasions when the user wishes to save the τ values of T to t (via **FLA_QR_UT_recover_tau()**), discard the matrix T , and then subsequently rebuild T from t . This routine facilitates the final step of such a process.

Caveats: **FLA_Accum_T_UT()** is currently only implemented for the case where **direct** is **FLA_FORWARD** and **storev** is **FLA_COLUMNWISE**.

Constraints:

- The numerical datatypes of V , t , and T must be identical and floating-point, and must not be **FLA_CONSTANT**.
- The length of t and the width of T must be $\min(m, n)$ where V is $m \times n$.

Arguments:

direct – Indicates whether H is formed from the forward or backward product of its constituent Householder reflectors.

```
void FLA_Apply_househ2_UT( FLA_Obj tau, FLA_Obj u2, FLA_Obj a1t, FLA_Obj A2 );
```

Purpose: Apply a single Householder transformation from the left to a row vector a_1^T and matrix A_2 :

$$\begin{pmatrix} a_1^T \\ A_2 \end{pmatrix} := \left(I - \frac{1}{\tau} \begin{pmatrix} 1 \\ u_2 \end{pmatrix} \begin{pmatrix} 1 & u_2^H \end{pmatrix} \right)^H \begin{pmatrix} a_1^T \\ A_2 \end{pmatrix}$$

where τ is the scalar and u_2 is the vector computed by `FLA_Househ2_UT()`. Note that a_1^T and A_2 are typically vertically adjacent views into the same matrix object, though this is not a requirement.

Constraints:

- The numerical datatypes of τ , u_2 , a_1^T , and A_2 must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The length of u_2 must be equal to the number of rows in A_2 and the length of a_1^T must be equal to the number of columns in A_2 .

Arguments:

- | | | |
|------------------|---|---|
| <code>tau</code> | – | An <code>FLA_Obj</code> representing scalar τ . |
| <code>u2</code> | – | An <code>FLA_Obj</code> representing vector u_2 . |
| <code>a1t</code> | – | An <code>FLA_Obj</code> representing vector a_1^T . |
| <code>A2</code> | – | An <code>FLA_Obj</code> representing matrix A_2 . |

```
void FLA_QR_UT_create_T( FLA_Obj A, FLA_Obj* T );
```

Purpose: Given an $m \times n$ matrix A upon which the user intends to perform a QR factorization via the UT transform, create a $b \times n$ matrix T where b is chosen to be a reasonable blocksize. This matrix T is required as input to `FLA_QR_UT()` so that the upper triangular block Householder transformations may be accumulated during each iteration of the factorization algorithm. Once created, T may be freed normally via `FLA_Obj_free()`. This routine is provided in case the user is not comfortable choosing the length of T , and thus implicitly setting the algorithmic blocksize of `FLA_QR_UT()`.

Notes: Matrix T is created so that its numerical datatype is the same as that of A .

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.

Arguments:

- | | | |
|----------------|---|--|
| <code>A</code> | – | An <code>FLA_Obj</code> representing matrix A . |
| <code>T</code> | | |
| (on entry) | – | A pointer to an uninitialized <code>FLA_Obj</code> . |
| (on exit) | – | A pointer to a new <code>FLA_Obj</code> parameterized by b , n , and the datatype of A . |

```
void FLA_QR_UT_recover_tau( FLA_Obj T, FLA_Obj t );
```

Purpose: Subsequent to a QR factorization via the UT transform, recover the τ values along the diagonals of the upper triangular block Householder submatrices of T and store them to a vector t .

Notes: This routine is rarely needed. However, there may be occasions when the user wishes to save the τ values of T to t , discard the matrix T , and then subsequently rebuild T from t (via `FLA_Accum_T_UT()`). This routine facilitates the first step of such a process.

Constraints:

- The numerical datatypes of T and t must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The width of T must be equal to $\text{dim}(t)$.

Arguments:

T	–	An <code>FLA_Obj</code> representing matrix T .
t	–	An <code>FLA_Obj</code> representing vector t .

```
void FLA_LQ_UT_create_T( FLA_Obj A, FLA_Obj* T );
```

Purpose: Given an $m \times n$ matrix A upon which the user intends to perform a LQ factorization via the UT transform, create an $m \times b$ matrix T where b is chosen to be a reasonable blocksize. This matrix T is required as input to `FLA_LQ_UT()` so that the upper triangular block Householder transformations may be accumulated during each iteration of the factorization algorithm. Once created, T may be freed normally via `FLA_Obj_free()`. This routine is provided in case the user is not comfortable choosing the width of T , and thus implicitly setting the algorithmic blocksize of `FLA_LQ_UT()`.

Notes: Matrix T is created so that its numerical datatype is the same as that of A .

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.

Arguments:

A	–	An <code>FLA_Obj</code> representing matrix A .
T		
	(on entry)	– A pointer to an uninitialized <code>FLA_Obj</code> .
	(on exit)	– A pointer to a new <code>FLA_Obj</code> parameterized by b , n , and the datatype of A .

```
void FLA_LQ_UT_recover_tau( FLA_Obj T, FLA_Obj t );
```

Purpose: Subsequent to an LQ factorization via the UT transform, recover the τ values along the diagonals of the upper triangular block Householder submatrices of T and store them to a vector t .

Notes: This routine is rarely needed. However, there may be occasions when the user wishes to save the τ values of T to t , discard the matrix T , and then subsequently rebuild T from t (via `FLA_Accum_T_UT()`). This routine facilitates the first step of such a process.

Constraints:

- The numerical datatypes of T and t must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The length of T must be equal to $\text{dim}(t)$.

Arguments:

<code>T</code>	–	An <code>FLA_Obj</code> representing matrix T .
<code>t</code>	–	An <code>FLA_Obj</code> representing vector t .

```
void FLASH_LU_incpiv_create_hier_matrices( FLA_Obj A_flat, dim_t depth, dim_t* b_flash,
                                           dim_t b_alg, FLA_Obj* A, FLA_Obj* p,
                                           FLA_Obj* L );
```

Purpose: Create a hierarchical matrix A conformal to a flat matrix A_{flat} and then copy the contents of A_{flat} into A . The hierarchy of A is specified by the depth and square blocksize values in `depth` and `b_flash`, respectively. Also, create hierarchical matrix objects p and L with proper datatypes, dimensions, and hierarchies relative to A so that the objects may be used together with `FLASH_LU_incpiv()` and `FLASH_FS_incpiv()`. If `b_alg` is greater than zero, it is used as the width of the storage blocks in L , which determines the algorithmic blocksize used in `FLASH_LU_incpiv()`. If `b_alg` is zero, the width of the storage blocks in L is set to a reasonable default value. Once created, A , p , and L may be freed normally via `FLASH_Obj_free()`.

Notes: This function is provided as a convenience to users of `FLASH_LU_incpiv()` so they do not need to worry about creating each auxiliary matrix object with the correct properties.

Caveats: Currently, this function only supports hierarchical depths of exactly 1.

Constraints:

- The numerical datatype of A_{flat} must be floating-point, and must not be `FLA_CONSTANT`.
- A_{flat} must be square.
- The pointer arguments `b_flash`, A , p , and L must not be `NULL`.
- Each of the first `depth` values in `b_flash` must be greater than zero.

Arguments:

<code>A_flat</code>	–	An <code>FLA_Obj</code> representing matrix A_{flat} .
<code>depth</code>	–	The number of levels to create in the matrix hierarchies of A , p , and L .
<code>b_flash</code>	–	A pointer to an array of <code>depth</code> values to be used as blocksizes in creating the matrix hierarchies of A , p , and L .
<code>b_alg</code>	–	The value to be used as the width of the storage blocks in L (ie: the number of columns in the leaves of L), which determines the algorithmic blocksize used in <code>FLASH_LU_incpiv()</code> and <code>FLASH_FS_incpiv()</code> , or zero if the user wishes to use a default value.
A		
(on entry)	–	A pointer to an uninitialized <code>FLA_Obj</code> .
(on exit)	–	A pointer to a new hierarchical <code>FLA_Obj</code> to represent matrix A , conformal to and initialized with the contents of A_{flat} .
p		
(on entry)	–	A pointer to an uninitialized <code>FLA_Obj</code> .
(on exit)	–	A pointer to a new hierarchical <code>FLA_Obj</code> to represent vector p .
L		
(on entry)	–	A pointer to an uninitialized <code>FLA_Obj</code> .
(on exit)	–	A pointer to a new hierarchical <code>FLA_Obj</code> to represent matrix L .

```
void FLASH_QR_UT_inc_create_hier_matrices( FLA_Obj A_flat, dim_t depth, dim_t* b_flash,
                                           dim_t b_alg, FLA_Obj* A, FLA_Obj* TW );
```

Purpose: Create a hierarchical matrix A conformal to a flat matrix A_{flat} and then copy the contents of A_{flat} into A . The hierarchy of A is specified by the depth and square blocksize values in `depth` and `b_flash`, respectively. Also, create hierarchical matrix object TW with proper datatype, dimensions, and hierarchy relative to A so that the objects may be used together with `FLASH_QR_UT_inc()` and `FLASH_Apply_Q_UT_inc()`. If `b_alg` is greater than zero, it is used as the length of the storage blocks in TW , which determines the algorithmic blocksize used in `FLASH_QR_UT_inc()`. If `b_alg` is zero, the length of the storage blocks in TW is set to a reasonable default value. Once created, A and TW may be freed normally via `FLASH_Obj_free()`.

Notes: This function is provided as a convenience to users of `FLASH_QR_UT_inc()` so they do not need to worry about creating the auxiliary TW matrix object with the correct properties.

Caveats: Currently, this function only supports hierarchical depths of exactly 1.

Constraints:

- The numerical datatype of A_{flat} must be floating-point, and must not be `FLA_CONSTANT`.
- A_{flat} must be square.
- The pointer arguments `b_flash`, `A`, and `TW` must not be `NULL`.
- Each of the first `depth` values in `b_flash` must be greater than zero.

Arguments:

<code>A_flat</code>	–	An <code>FLA_Obj</code> representing matrix A_{flat} .
<code>depth</code>	–	The number of levels to create in the matrix hierarchies of A and TW .
<code>b_flash</code>	–	A pointer to an array of <code>depth</code> values to be used as blocksizes in creating the matrix hierarchies of A and TW .
<code>b_alg</code>	–	The value to be used as the length of the storage blocks in TW (ie: the number of rows in the leaves of TW), which determines the algorithmic blocksize used in <code>FLASH_QR_UT_inc()</code> and <code>FLASH_Apply_Q_UT_inc()</code> , or zero if the user wishes to use a default value.
<code>A</code>		
(on entry)	–	A pointer to an uninitialized <code>FLA_Obj</code> .
(on exit)	–	A pointer to a new hierarchical <code>FLA_Obj</code> to represent matrix A , conformal to and initialized with the contents of A_{flat} .
<code>TW</code>		
(on entry)	–	A pointer to an uninitialized <code>FLA_Obj</code> .
(on exit)	–	A pointer to a new hierarchical <code>FLA_Obj</code> to represent matrix TW .

```
void FLASH_Apply_Q_UT_inc_create_workspace( FLA_Obj TW, FLA_Obj B, FLA_Obj* W );
```

Purpose: Create a hierarchical workspace matrix W needed when applying Q^T (or Q^H) to B via `FLASH_Apply_Q_UT_inc()`. Once created, W may be freed normally via `FLASH_Obj_free()`.

Notes: This function is provided as a convenience to users of `FLASH_Apply_Q_UT_inc()` so they do not need to worry about creating the workspace matrix object W with the correct properties.

Caveats: Currently, this function only supports hierarchical depths of exactly 1.

Constraints:

- The numerical datatype of TW and B must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The pointer argument W must not be `NULL`.

Arguments:

TW	–	A hierarchical <code>FLA_Obj</code> representing matrix TW .
B	–	A hierarchical <code>FLA_Obj</code> representing matrix B .
W		
	(on entry) –	A pointer to an uninitialized <code>FLA_Obj</code> .
	(on exit) –	A pointer to a new hierarchical <code>FLA_Obj</code> to represent matrix W .

5.8 External wrappers

This section documents the wrapper interfaces to the external implementations of all operations supported within `libflame`. We refer to these interfaces as *wrappers* because they wrap the less aesthetically pleasing Fortran-77 interfaces of the BLAS and LAPACK with easy-to-use functions that operate upon `libflame` objects. Furthermore, we refer to them as interfacing to *external* code because they interface to implementations that reside outside of `libflame`. Usually, these external implementations are provided by a separate BLAS and LAPACK library at link-time. However, they could be provided by some other source. The user may even request, at configure-time, that `libflame` be built to include basic netlib implementations of all LAPACK-level operations supported within the library. The only requirement is that the external implementation adhere to the original Fortran-77 BLAS or LAPACK interface.

5.8.1 BLAS operations

5.8.1.1 Level-1 BLAS

```
void FLA_Asum_external( FLA_Obj x, FLA_Obj norm1 );
```

Purpose: Compute the 1-norm of a vector:

$$\|x\|_1 := \sum_{i=0}^{n-1} |\chi_i|$$

where $\|x\|_1$ is a scalar and χ_i is the i th element of general vector x of length n . Upon completion, the 1-norm $\|x\|_1$ is stored to `norm1`.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `*asum()`.

More Info: This function's interface is similar to that of `FLA_Asum()`. Please see the description for `FLA_Asum()` for further details.

```
void FLA_Axpy_external( FLA_Obj alpha, FLA_Obj X, FLA_Obj Y );
```

Purpose: Perform an AXPY operation:

$$Y := Y + \alpha X$$

where α is a scalar, and X and Y are general matrices.

Notes: If X and Y are vectors, `FLA_Axpy_external()` will implicitly and automatically perform the transposition necessary to achieve conformal dimensions:

$$\begin{aligned} y_r &:= y_r + \alpha x_c^T \\ y_c &:= y_c + \alpha x_r^T \end{aligned}$$

where x_c and y_c are column vectors and x_r and y_r are row vectors.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `?axpy()`.

More Info: This function's interface is similar to that of `FLA_Axpy()`. Please see the description for `FLA_Axpy()` for further details.

```
void FLA_Axpyt_external( FLA_Trans trans, FLA_Obj alpha, FLA_Obj X, FLA_Obj Y );
```

Purpose: Perform one of the following extended AXPY operations:

$$\begin{aligned} Y &:= Y + \alpha X \\ Y &:= Y + \alpha X^T \\ Y &:= Y + \alpha \bar{X} \\ Y &:= Y + \alpha X^H \end{aligned}$$

where α is a scalar, and X and Y are general matrices. The `trans` argument allows the computation to proceed as if X were conjugated and/or transposed.

Notes: If X and Y are vectors, `FLA_Axpyt_external()` will implicitly and automatically perform the transposition necessary to achieve conformal dimensions regardless of the value of `trans`:

$$\begin{aligned} y_r &:= y_r + \alpha x_c^T \\ y_c &:= y_c + \alpha x_r^T \end{aligned}$$

where x_c and y_c are column vectors and x_r and y_r are row vectors. In these special cases where X and Y are vectors, `trans` argument values of `FLA_TRANSPOSE` and `FLA_CONJ_TRANSPOSE` will effectively be interpreted as `FLA_NO_TRANSPOSE` and `FLA_CONJ_NO_TRANSPOSE`, respectively, and thus the `trans` argument may still be used to request conjugation of vector X .

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `?axpyt()`.

More Info: This function's interface is similar to that of `FLA_Axpyt()`. Please see the description for `FLA_Axpyt()` for further details.


```
void FLA_Axpys_external( FLA_Obj alpha0, FLA_Obj alpha1, FLA_Obj X,
                        FLA_Obj beta, FLA_Obj Y );
```

Purpose: Perform the following extended AXPY operation:

$$Y := \beta Y + \alpha_0 \alpha_1 X$$

where α_0 , α_1 and β are scalars, and X and Y are general matrices.

Notes: If X and Y are vectors, `FLA_Axpys_external()` will implicitly and automatically perform the transposition necessary to achieve conformal dimensions:

$$\begin{aligned} y_r &:= \beta y_r + \alpha_0 \alpha_1 x_c^T \\ y_c &:= \beta y_c + \alpha_0 \alpha_1 x_r^T \end{aligned}$$

where x_c and y_c are column vectors and x_r and y_r are row vectors.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `?axpy()`.

More Info: This function's interface is similar to that of `FLA_Axpys()`. Please see the description for `FLA_Axpys()` for further details.

```
void FLA_Copy_external( FLA_Obj A, FLA_Obj B );
```

Purpose: Copy the numerical contents of A to B :

$$B := A$$

where A and B are general matrices.

Notes: If A and B are vectors, `FLA_Copy_external()` will implicitly and automatically perform the transposition necessary to achieve conformal dimensions:

$$\begin{aligned} b_r &:= a_c^T \\ b_c &:= a_r^T \end{aligned}$$

where a_c and b_c are column vectors and a_r and b_r are row vectors.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `?copy()`.

More Info: This function's interface is similar to that of `FLA_Copy()`. Please see the description for `FLA_Copy()` for further details.

```
void FLA_Copyr_external( FLA_Uplo uplo, FLA_Obj A, FLA_Obj B );
```

Purpose: Perform an extended copy operation on the lower or upper triangles of matrices A and B :

$$B := A$$

where A and B are general square matrices. The `uplo` argument indicates whether the lower or upper triangles of A and B are referenced and updated by the operation.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `?copyr()`.

More Info: This function's interface is similar to that of `FLA_Copyr()`. Please see the description for `FLA_Copyr()` for further details.

```
void FLA_Copyt_external( FLA_Trans trans, FLA_Obj A, FLA_Obj B );
```

Purpose: Copy the numerical contents of A to B with one of the following extended operations:

$$\begin{aligned} B &:= A \\ B &:= A^T \\ B &:= \bar{A} \\ B &:= A^H \end{aligned}$$

where A and B are general matrices. The **trans** argument allows the computation to proceed as if A were conjugated and/or transposed.

Notes: If A and B are vectors, `FLA_Copyt_external()` will implicitly and automatically perform the transposition necessary to achieve conformal dimensions regardless of the value of **trans**:

$$\begin{aligned} b_r &:= a_c^T \\ b_c &:= a_r^T \end{aligned}$$

where a_c and b_c are column vectors and a_r and b_r are row vectors. In these special cases where A and B are vectors, **trans** argument values of `FLA_TRANSPOSE` and `FLA_CONJ_TRANSPOSE` will effectively be interpreted as `FLA_NO_TRANSPOSE` and `FLA_CONJ_NO_TRANSPOSE`, respectively, and thus the **trans** argument may still be used to request conjugation of vector A .

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `?copy()`.

More Info: This function's interface is similar to that of `FLA_Copyt()`. Please see the description for `FLA_Copyt()` for further details.

```
void FLA_Dot_external( FLA_Obj x, FLA_Obj y, FLA_Obj rho );
```

Purpose: Perform a dot (inner) product operation between two vectors:

$$\rho := \sum_{i=0}^{n-1} \chi_i \psi_i$$

where ρ is a scalar, and χ_i and ψ_i are the i th elements of general vectors x and y , respectively, where both vectors are of length n . Upon completion, the dot product ρ is stored to **rho**.

Imp. Notes: This function uses external implementations of the level-1 BLAS routines `?dot()` and `?dotu()`.

More Info: This function's interface is similar to that of `FLA_Dot()`. Please see the description for `FLA_Dot()` for further details.

```
void FLA_Dotc_external( FLA_Conj conj, FLA_Obj x, FLA_Obj y, FLA_Obj rho );
```

Purpose: Perform one of the following extended dot product operations:

$$\rho := \sum_{i=0}^{n-1} \chi_i \psi_i$$

$$\rho := \sum_{i=0}^{n-1} \bar{\chi}_i \bar{\psi}_i$$

where ρ is a scalar, and χ_i and ψ_i are the i th elements of general vectors x and y , respectively, where both vectors are of length n . Upon completion, the dot product ρ is stored to **rho**. The **conj** argument allows the computation to proceed as if x and y were conjugated.

Notes: If x , y , and ρ are real, the value of **conj** is ignored and `FLA_Dotc_external()` behaves exactly as `FLA_Dot_external()`.

Imp. Notes: This function uses external implementations of the level-1 BLAS routines `?dot()`, `?dotu()`, and `?dotc()`.

More Info: This function's interface is similar to that of `FLA_Dotc()`. Please see the description for `FLA_Dotc()` for further details.

```
void FLA_Dots_external( FLA_Obj alpha, FLA_Obj x, FLA_Obj y,
                       FLA_Obj beta, FLA_Obj rho );
```

Purpose: Perform the following extended dot product operation between two vectors:

$$\rho := \beta\rho + \alpha \sum_{i=0}^{n-1} \chi_i \psi_i$$

where α , β , and ρ are scalars, and χ_i and ψ_i are the i th elements of general vectors x and y , respectively, where both vectors are of length n . Upon completion, the dot product ρ is stored to **rho**.

Imp. Notes: This function uses external implementations of the level-1 BLAS routines `?dot()` and `?dotu()`.

More Info: This function's interface is similar to that of `FLA_Dots()`. Please see the description for `FLA_Dots()` for further details.

```
void FLA_Dotcs_external( FLA_Conj conj, FLA_Obj alpha, FLA_Obj x, FLA_Obj y,
                        FLA_Obj beta, FLA_Obj rho );
```

Purpose: Perform one of the following extended dot product operations between two vectors:

$$\rho := \beta\rho + \alpha \sum_{i=0}^{n-1} \chi_i \psi_i$$

$$\rho := \beta\rho + \alpha \sum_{i=0}^{n-1} \bar{\chi}_i \bar{\psi}_i$$

where α , β , and ρ are scalars, and χ_i and ψ_i are the i th elements of general vectors x and y , respectively, where both vectors are of length n . Upon completion, the dot product ρ is stored to **rho**. The **conj** argument allows the computation to proceed as if x and y were conjugated.

Notes: If x , y , and ρ are real, the value of **conj** is ignored and `FLA_Dotcs_external()` behaves exactly as `FLA_Dots_external()`.

Imp. Notes: This function uses external implementations of the level-1 BLAS routines `?dot()`, `?dotu()`, and `?dotc()`.

More Info: This function's interface is similar to that of `FLA_Dotcs()`. Please see the description for `FLA_Dotcs()` for further details.

```
void FLA_Dot2s_external( FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj beta, FLA_Obj rho );
```

Purpose: Perform the following extended dot product operation between two vectors:

$$\rho := \beta\rho + \alpha \sum_{i=0}^{n-1} \chi_i \psi_i + \bar{\alpha} \sum_{i=0}^{n-1} \chi_i \psi_i$$

where α , β , and ρ are scalars, and χ_i and ψ_i are the i th elements of general vectors x and y , respectively, where both vectors are of length n . Upon completion, the dot product ρ is stored to **rho**.

Notes: Though this operation may be reduced to:

$$\rho := \beta\rho + (\alpha + \bar{\alpha}) \sum_{i=0}^{n-1} \chi_i \psi_i$$

it is expressed above in unreduced form to allow a more clear contrast to `FLA_Dot2cs_external()`.

Imp. Notes: This function uses external implementations of the level-1 BLAS routines `?dot()` and `?dotu()`.

More Info: This function's interface is similar to that of `FLA_Dot2s()`. Please see the description for `FLA_Dot2s()` for further details.

```
void FLA_Dot2cs_external( FLA_Conj conj, FLA_Obj alpha, FLA_Obj x, FLA_Obj y,
                        FLA_Obj beta, FLA_Obj rho );
```

Purpose: Perform one of the following extended dot product operations between two vectors:

$$\rho := \beta\rho + \alpha \sum_{i=0}^{n-1} \chi_i \psi_i + \bar{\alpha} \sum_{i=0}^{n-1} \chi_i \psi_i$$

$$\rho := \beta\rho + \alpha \sum_{i=0}^{n-1} \chi_i \psi_i + \bar{\alpha} \sum_{i=0}^{n-1} \bar{\chi}_i \bar{\psi}_i$$

where α , β , and ρ are scalars, and χ_i and ψ_i are the i th elements of general vectors x and y , respectively, where both vectors are of length n . Upon completion, the dot product ρ is stored to **rho**. The **conj** argument allows the computation to proceed as if x and y were conjugated.

Notes: If x , y , and ρ are real, the value of **conj** is ignored and `FLA_Dot2cs_external()` behaves exactly as `FLA_Dot2s_external()`.

Imp. Notes: This function uses external implementations of the level-1 BLAS routines `?dot()`, `?dotu()`, and `?dotc()`.

More Info: This function's interface is similar to that of `FLA_Dot2cs()`. Please see the description for `FLA_Dot2cs()` for further details.

```
void FLA_Iamax_external( FLA_Obj x, FLA_Obj i );
```

Purpose: Find the index i of the element of x which has the maximum absolute value, where x is a general vector and i is a scalar. If the maximum absolute value is shared by more than one element, then the element whose index is highest is chosen.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `i?amax()`.

More Info: This function's interface is similar to that of `FLA_Iamax()`. Please see the description for `FLA_Iamax()` for further details.

```
void FLA_Inv_scal_external( FLA_Obj alpha, FLA_Obj A );
```

Purpose: Perform an inverse scaling operation:

$$A := \alpha^{-1}A$$

where α is a scalar and A is a general matrix.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `*scal()`.

More Info: This function's interface is similar to that of `FLA_Inv_scal()`. Please see the description for `FLA_Inv_scal()` for further details.

```
void FLA_Inv_scalc_external( FLA_Conj conjalpha, FLA_Obj alpha, FLA_Obj A );
```

Purpose: Perform one of the following extended inverse scaling operations:

$$\begin{aligned} A &:= \alpha^{-1}A \\ A &:= \bar{\alpha}^{-1}A \end{aligned}$$

where α is a scalar and A is a general matrix. The `conjalpha` argument allows the computation to proceed as if α were conjugated.

Notes: If α is real, the value of `conjalpha` is ignored and `FLA_Inv_scalc_external()` behaves exactly as `FLA_Inv_scal_external()`.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `*scal()`.

More Info: This function's interface is similar to that of `FLA_Inv_scalc()`. Please see the description for `FLA_Inv_scalc()` for further details.

```
void FLA_Nrm2_external( FLA_Obj x, FLA_Obj norm );
```

Purpose: Compute the 2-norm of a vector:

$$\|x\|_2 := \left(\sum_{i=0}^{n-1} |\chi_i|^2 \right)^{\frac{1}{2}}$$

where $\|x\|_2$ is a scalar and χ_i is the i th element of general vector x of length n . Upon completion, the 2-norm $\|x\|_2$ is stored to `norm`.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `*nrm2()`.

More Info: This function's interface is similar to that of `FLA_Nrm2()`. Please see the description for `FLA_Nrm2()` for further details.

```
void FLA_Scal_external( FLA_Obj alpha, FLA_Obj A );
```

Purpose: Perform a scaling operation:

$$A := \alpha A$$

where α is a scalar and A is a general matrix.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `*scal()`.

More Info: This function's interface is similar to that of `FLA_Scal()`. Please see the description for `FLA_Scal()` for further details.

```
void FLA_Scalc_external( FLA_Conj conjalpha, FLA_Obj alpha, FLA_Obj A );
```

Purpose: Perform one of the following extended scaling operations:

$$A := \alpha A$$

$$A := \bar{\alpha} A$$

where α is a scalar and A is a general matrix. The `conjalpha` argument allows the computation to proceed as if α were conjugated.

Notes: If α is real, the value of `conjalpha` is ignored and `FLA_Scalc_external()` behaves exactly as `FLA_Scal_external()`.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `*scal()`.

More Info: This function's interface is similar to that of `FLA_Scalc()`. Please see the description for `FLA_Scalc()` for further details.

```
void FLA_Scalr_external( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj A );
```

Purpose: Perform an extended scaling operation on the lower or upper triangle of a matrix:

$$A := \alpha A$$

where α is a scalar and A is a general square matrix. The `uplo` argument indicates whether the lower or upper triangle of A is referenced and updated by the operation.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `*scal()`.

More Info: This function's interface is similar to that of `FLA_Scalr()`. Please see the description for `FLA_Scalr()` for further details.

```
void FLA_Swap_external( FLA_Obj A, FLA_Obj B );
```

Purpose: Swap the contents of two general matrices A and B .

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `?swap()`.

More Info: This function's interface is similar to that of `FLA_Swap()`. Please see the description for `FLA_Swap()` for further details.

```
void FLA_Swapt_external( FLA_Trans transab, FLA_Obj A, FLA_Obj B );
```

Purpose: Swap the contents of two general matrices A and B . If `transab` is `FLA_TRANSPOSE` or `FLA_CONJ_TRANSPOSE`, the computation proceeds as if only A (or only B) were transposed. Furthermore, if `transab` is `FLA_CONJ_NO_TRANSPOSE` or `FLA_CONJ_TRANSPOSE`, both A and B are conjugated after their contents are swapped.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine `?swapt()`.

More Info: This function's interface is similar to that of `FLA_Swapt()`. Please see the description for `FLA_Swapt()` for further details.

5.8.1.2 Level-2 BLAS

```
void FLA_Gemv_external( FLA_Trans transa, FLA_Obj alpha, FLA_Obj A, FLA_Obj x,
                       FLA_Obj beta, FLA_Obj y );
```

Purpose: Perform one of the following general matrix-vector multiplication operations:

$$\begin{aligned} y &:= \beta y + \alpha Ax \\ y &:= \beta y + \alpha A^T x \\ y &:= \beta y + \alpha \bar{A}x \\ y &:= \beta y + \alpha A^H x \end{aligned}$$

where α and β are scalars, A is a general matrix, and x and y are general vectors. The **trans** argument allows the computation to proceed as if A were conjugated and/or transposed.

Notes: The above matrix-vector operations implicitly assume x and y to be column vectors. However, since transposing a vector does not change the way its elements are accessed, we may also express the above operations as:

$$\begin{aligned} y_r &:= \beta y_r + \alpha x_r A^T \\ y_r &:= \beta y_r + \alpha x_r A \\ y_r &:= \beta y_r + \alpha x_r A^H \\ y_r &:= \beta y_r + \alpha x_r \bar{A} \end{aligned}$$

respectively, where x_r and y_r are row vectors.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?gemv()`.

More Info: This function's interface is similar to that of `FLA_Gemv()`. Please see the description for `FLA_Gemv()` for further details.


```
void FLA_Gemvc_external( FLA_Trans transa, FLA_Conj conjx, FLA_Obj alpha,
                        FLA_Obj A, FLA_Obj x, FLA_Obj beta, FLA_Obj y );
```

Purpose: Perform one of the following extended general matrix-vector multiplication operations:

$$\begin{array}{ll}
 y &:= \beta y + \alpha Ax & y &:= \beta y + \alpha A\bar{x} \\
 y &:= \beta y + \alpha A^T x & y &:= \beta y + \alpha A^T \bar{x} \\
 y &:= \beta y + \alpha \bar{A}x & y &:= \beta y + \alpha \bar{A}\bar{x} \\
 y &:= \beta y + \alpha A^H x & y &:= \beta y + \alpha A^H \bar{x}
 \end{array}$$

where α and β are scalars, A is a general matrix, and x and y are general vectors. The **trans** argument allows the computation to proceed as if A were conjugated and/or transposed. Likewise, the **conjx** argument allows the computation to proceed as if x were conjugated.

Notes: The above matrix-vector operations implicitly assume x and y to be column vectors. However, since transposing a vector does not change the way its elements are accessed, we may also express the above operations as:

$$\begin{array}{ll}
 y_r &:= \beta y_r + \alpha x_r A^T & y_r &:= \beta y_r + \alpha \bar{x}_r A^T \\
 y_r &:= \beta y_r + \alpha x_r A & y_r &:= \beta y_r + \alpha \bar{x}_r A \\
 y_r &:= \beta y_r + \alpha x_r A^H & y_r &:= \beta y_r + \alpha \bar{x}_r A^H \\
 y_r &:= \beta y_r + \alpha x_r \bar{A} & y_r &:= \beta y_r + \alpha \bar{x}_r \bar{A}
 \end{array}$$

respectively, where x_r and y_r are row vectors.

If A , x , and y are real, the value of **conjx** is ignored and **FLA_Gemvc_external()** behaves exactly as **FLA_Gemv_external()**.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine **?gemv()**.

More Info: This function's interface is similar to that of **FLA_Gemvc()**. Please see the description for **FLA_Gemvc()** for further details.

```
void FLA_Ger_external( FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj A );
```

Purpose: Perform a general rank-1 update:

$$A := A + \alpha xy^T$$

where α is a scalar, A is a general matrix, and x and y are general vectors.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine **?ger()**.

More Info: This function's interface is similar to that of **FLA_Ger()**. Please see the description for **FLA_Ger()** for further details.

```
void FLA_Gerc_external( FLA_Conj conjx, FLA_Conj conjy, FLA_Obj alpha,
                      FLA_Obj x, FLA_Obj y, FLA_Obj A );
```

Purpose: Perform one of the following extended general rank-1 updates:

$$\begin{aligned} A &:= A + \alpha xy^T \\ A &:= A + \alpha x\bar{y}^T \\ A &:= A + \alpha \bar{x}y^T \\ A &:= A + \alpha \bar{x}\bar{y}^T \end{aligned}$$

where α is a scalar, A is a general matrix, and x and y are general vectors. The `conjx` and `conjy` arguments allow the computation to proceed as if x and/or y were conjugated.

Notes: If A , x , and y are real, the values of `conjx` and `conjy` are ignored and `FLA_Gerc_external()` behaves exactly as `FLA_Ger_external()`.

Imp. Notes: This function uses external implementations of the level-3 BLAS routines `?ger()`, `?geru()`, and `?gerc()`.

More Info: This function's interface is similar to that of `FLA_Gerc()`. Please see the description for `FLA_Gerc()` for further details.

```
void FLA_Hemv_external( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj A, FLA_Obj x,
                      FLA_Obj beta, FLA_Obj y );
```

Purpose: Perform a Hermitian matrix-vector multiplication operation:

$$y := \beta y + \alpha Ax$$

where α and β are scalars, A is a complex Hermitian matrix, and x and y are general complex vectors. The `uplo` argument indicates whether the lower or upper triangle of A is referenced by the operation.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?hemv()`.

More Info: This function's interface is similar to that of `FLA_Hemv()`. Please see the description for `FLA_Hemv()` for further details.

```
void FLA_Hemvc_external( FLA_Uplo uplo, FLA_Conj conjA, FLA_Obj alpha,
                      FLA_Obj A, FLA_Obj x, FLA_Obj beta, FLA_Obj y );
```

Purpose: Perform one of the following extended Hermitian matrix-vector multiplication operations:

$$\begin{aligned} y &:= \beta y + \alpha Ax \\ y &:= \beta y + \alpha \bar{A}x \end{aligned}$$

where α and β are scalars, A is a complex Hermitian matrix, and x and y are general complex vectors. The `uplo` argument indicates whether the lower or upper triangle of A is referenced by the operation. The `conjA` argument allows the computation to proceed as if A were conjugated.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?hemv()`.

More Info: This function's interface is similar to that of `FLA_Hemvc()`. Please see the description for `FLA_Hemvc()` for further details.

```
void FLA_Her_external( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj A );
```

Purpose: Perform a complex Hermitian rank-1 update:

$$A := A + \alpha x x^H$$

where α is a scalar, A is a complex Hermitian matrix, and x is a general complex vector. The `uplo` argument indicates whether the lower or upper triangle of A is referenced and updated by the operation.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?her()`.

More Info: This function's interface is similar to that of `FLA_Her()`. Please see the description for `FLA_Her()` for further details.

```
void FLA_Herc_external( FLA_Uplo uplo, FLA_Conj conjx, FLA_Obj alpha, FLA_Obj x,
                       FLA_Obj A );
```

Purpose: Perform one of the following extended complex Hermitian rank-1 updates:

$$\begin{aligned} A &:= A + \alpha x x^H \\ A &:= A + \alpha \bar{x} \bar{x}^H \end{aligned}$$

where α is a scalar, A is a complex Hermitian matrix, and x is a general complex vector. The `uplo` argument indicates whether the lower or upper triangle of A is referenced and updated by the operation. The `conjx` argument allows the computation to proceed as if x were conjugated.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?herc()`.

More Info: This function's interface is similar to that of `FLA_Herc()`. Please see the description for `FLA_Herc()` for further details.

```
void FLA_Her2_external( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj y,
                       FLA_Obj A );
```

Purpose: Perform a complex Hermitian rank-2 update:

$$A := A + \alpha x y^H + \bar{\alpha} y x^H$$

where α is a scalar, A is a complex Hermitian matrix, and x and y are general complex vectors. The `uplo` argument indicates whether the lower or upper triangle of A is referenced and updated by the operation.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?her2()`.

More Info: This function's interface is similar to that of `FLA_Her2()`. Please see the description for `FLA_Her2()` for further details.

```
void FLA_Her2c_external( FLA_Uplo uplo, FLA_Conj conjxy, FLA_Obj alpha,
                        FLA_Obj x, FLA_Obj y, FLA_Obj A );
```

Purpose: Perform one of the following extended complex Hermitian rank-2 updates:

$$\begin{aligned} A &:= A + \alpha xy^H + \bar{\alpha} yx^H \\ A &:= A + \alpha \bar{x} \bar{y}^H + \bar{\alpha} \bar{y} \bar{x}^H \end{aligned}$$

where α is a scalar, A is a complex Hermitian matrix, and x and y are general complex vectors. The `uplo` argument indicates whether the lower or upper triangle of A is referenced and updated by the operation. The `conjxy` argument allows the computation to proceed as if x and y were conjugated.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?her2()`.

More Info: This function's interface is similar to that of `FLA_Her2c()`. Please see the description for `FLA_Her2c()` for further details.

```
void FLA_Symv_external( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj A, FLA_Obj x,
                        FLA_Obj beta, FLA_Obj y );
```

Purpose: Perform a symmetric matrix-vector multiplication operation:

$$y := \beta y + \alpha Ax$$

where α and β are scalars, A is a symmetric matrix, and x and y are general vectors. The `uplo` argument indicates whether the lower or upper triangle of A is referenced by the operation.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?symv()`.

More Info: This function's interface is similar to that of `FLA_Symv()`. Please see the description for `FLA_Symv()` for further details.

```
void FLA_Syr_external( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj A );
```

Purpose: Perform a symmetric rank-1 update:

$$A := A + \alpha xx^T$$

where α is a scalar, A is a symmetric matrix, and x is a general vector. The `uplo` argument indicates whether the lower or upper triangle of A is referenced and updated by the operation.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?syr()`.

More Info: This function's interface is similar to that of `FLA_Syr()`. Please see the description for `FLA_Syr()` for further details.

```
void FLA_Syr2_external( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj y,
                      FLA_Obj A );
```

Purpose: Perform a symmetric rank-2 update:

$$A := A + \alpha xy^T + \alpha yx^T$$

where α is a scalar, A is a symmetric matrix, and x and y are general vectors. The `uplo` argument indicates whether the lower or upper triangle of A is referenced and updated by the operation.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?syr2()`.

More Info: This function's interface is similar to that of `FLA_Syr2()`. Please see the description for `FLA_Syr2()` for further details.

```
void FLA_Trmv_external( FLA_Uplo uplo, FLA_Trans transa, FLA_Diag diag, FLA_Obj A,
                      FLA_Obj x );
```

Purpose: Perform one of the following triangular matrix-vector multiplication operations:

$$\begin{aligned} x &:= Ax \\ x &:= A^T x \\ x &:= \bar{A}x \\ x &:= A^H x \end{aligned}$$

where A is a triangular matrix and x is a general vector. The `uplo` argument indicates whether the lower or upper triangle of A is referenced by the operation. The `transa` argument allows the computation to proceed as if A were conjugated and/or transposed. The `diag` argument indicates whether the diagonal of A is unit or non-unit.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?trmv()`.

More Info: This function's interface is similar to that of `FLA_Trmv()`. Please see the description for `FLA_Trmv()` for further details.

```
void FLA_Trmvsx_external( FLA_Uplo uplo, FLA_Trans transa, FLA_Diag diag, FLA_Obj alpha,
                        FLA_Obj A, FLA_Obj x, FLA_Obj beta, FLA_Obj y );
```

Purpose: Perform one of the following extended triangular matrix-vector multiplication operations:

$$\begin{aligned} y &:= \beta y + \alpha Ax \\ y &:= \beta y + \alpha A^T x \\ y &:= \beta y + \alpha \bar{A} x \\ y &:= \beta y + \alpha A^H x \end{aligned}$$

where α and β are scalars, A is a triangular matrix, and x and y are general vectors. The **uplo** argument indicates whether the lower or upper triangle of A is referenced by the operation. The **transa** argument allows the computation to proceed as if A were conjugated and/or transposed. The **diag** argument indicates whether the diagonal of A is unit or non-unit.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?trmv()`.

More Info: This function's interface is similar to that of `FLA_Trmvsx()`. Please see the description for `FLA_Trmvsx()` for further details.

```
void FLA_Trsv_external( FLA_Uplo uplo, FLA_Trans transa, FLA_Diag diag, FLA_Obj A,
                      FLA_Obj b );
```

Purpose: Perform one of the following triangular solves with multiple right-hand sides:

$$\begin{aligned} Ax &= b \\ A^T x &= b \\ \bar{A} x &= b \\ A^H x &= b \end{aligned}$$

which, respectively, are solved by overwriting b with the contents of the solution vector x as follows:

$$\begin{aligned} b &:= A^{-1}b \\ b &:= A^{-T}b \\ b &:= \bar{A}^{-1}b \\ b &:= A^{-H}b \end{aligned}$$

where A is a triangular matrix and x and b are general vectors. The **uplo** argument indicates whether the lower or upper triangle of A is referenced by the operation. The **transa** argument allows the computation to proceed as if A were conjugated and/or transposed. The **diag** argument indicates whether the diagonal of A is unit or non-unit.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?trsv()`.

More Info: This function's interface is similar to that of `FLA_Trsv()`. Please see the description for `FLA_Trsv()` for further details.

```
void FLA_Trsvsx_external( FLA_Uplo uplo, FLA_Trans transa, FLA_Diag diag, FLA_Obj alpha,
                        FLA_Obj A, FLA_Obj b, FLA_Obj beta, FLA_Obj y );
```

Purpose: Perform one of the following extended triangular solves with multiple right-hand sides:

$$\begin{aligned} y &:= \beta y + \alpha A^{-1} b \\ y &:= \beta y + \alpha A^{-T} b \\ y &:= \beta y + \alpha \bar{A}^{-1} b \\ y &:= \beta y + \alpha A^{-H} b \end{aligned}$$

where α and β are scalars, A is a triangular matrix, and b and y are general vectors. The **uplo** argument indicates whether the lower or upper triangle of A is referenced by the operation. The **transa** argument allows the computation to proceed as if A were conjugated and/or transposed. The **diag** argument indicates whether the diagonal of A is unit or non-unit.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?trsv()`.

More Info: This function's interface is similar to that of `FLA_Trsvsx()`. Please see the description for `FLA_Trsvsx()` for further details.

5.8.1.3 Level-3 BLAS

```
void FLA_Gemm_external( FLA_Trans transa, FLA_Trans transb, FLA_Obj alpha,
                      FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C );
```

Purpose: Perform one of the following general matrix-matrix multiplication (GEMM) operations:

$$\begin{aligned} C &:= \beta C + \alpha AB & C &:= \beta C + \alpha \bar{A}B \\ C &:= \beta C + \alpha AB^T & C &:= \beta C + \alpha \bar{A}B^T \\ C &:= \beta C + \alpha A\bar{B} & C &:= \beta C + \alpha \bar{A}\bar{B} \\ C &:= \beta C + \alpha AB^H & C &:= \beta C + \alpha \bar{A}B^H \\ C &:= \beta C + \alpha A^T B & C &:= \beta C + \alpha A^H B \\ C &:= \beta C + \alpha A^T B^T & C &:= \beta C + \alpha A^H B^T \\ C &:= \beta C + \alpha A^T \bar{B} & C &:= \beta C + \alpha A^H \bar{B} \\ C &:= \beta C + \alpha A^T B^H & C &:= \beta C + \alpha A^H B^H \end{aligned}$$

where α and β are scalars and A , B , and C are general matrices. The **transa** and **transb** arguments allows the computation to proceed as if A and/or B were conjugated and/or transposed.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?gemm()`.

More Info: This function's interface is similar to that of `FLA_Gemm()`. Please see the description for `FLA_Gemm()` for further details.

```
void FLA_Hemm_external( FLA_Side side, FLA_Uplo uplo, FLA_Obj alpha,
                       FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C );
```

Purpose: Perform one of the following Hermitian matrix-matrix multiplication (HEMM) operations:

$$\begin{aligned} C &:= \beta C + \alpha AB \\ C &:= \beta C + \alpha BA \end{aligned}$$

where α and β are scalars, A is a complex Hermitian matrix, and B and C are general complex matrices. The **side** argument indicates whether matrix A is multiplied on the left or the right side of B . The **uplo** argument indicates whether the lower or upper triangle of A is referenced by the operation.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?hemm()`.

More Info: This function's interface is similar to that of `FLA_Hemm()`. Please see the description for `FLA_Hemm()` for further details.

```
void FLA_Herk_external( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha,
                       FLA_Obj A, FLA_Obj beta, FLA_Obj C );
```

Purpose: Perform one of the following Hermitian rank-k update (HERK) operations:

$$\begin{aligned} C &:= \beta C + \alpha AA^H \\ C &:= \beta C + \alpha A^H A \end{aligned}$$

where α and β are scalars, C is a complex Hermitian matrix, and A is a general complex matrix. The **uplo** argument indicates whether the lower or upper triangle of C is referenced and updated by the operation. The **trans** argument allows the computation to proceed as if A were conjugate-transposed, which results in the alternate rank-k product $A^H A$.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?herk()`.

More Info: This function's interface is similar to that of `FLA_Herk()`. Please see the description for `FLA_Herk()` for further details.

```
void FLA_Her2k_external( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha,
                        FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C );
```

Purpose: Perform one of the following Hermitian rank-2k update (HER2K) operations:

$$\begin{aligned} C &:= \beta C + \alpha AB^H + \bar{\alpha} BA^H \\ C &:= \beta C + \alpha A^H B + \bar{\alpha} B^H A \end{aligned}$$

where α and β are scalars, C is a complex Hermitian matrix, and A and B are general complex matrices. The **uplo** argument indicates whether the lower or upper triangle of C is referenced and updated by the operation. The **trans** argument allows the computation to proceed as if A and B were conjugate-transposed, which results in the alternate rank-2k products $A^H B$ and $B^H A$.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?her2k()`.

More Info: This function's interface is similar to that of `FLA_Her2k()`. Please see the description for `FLA_Her2k()` for further details.


```
void FLA_Symm_external( FLA_Side side, FLA_Uplo uplo, FLA_Obj alpha,
                       FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C );
```

Purpose: Perform one of the following symmetric matrix-matrix multiplication (SYMM) operations:

$$\begin{aligned} C &:= \beta C + \alpha AB \\ C &:= \beta C + \alpha BA \end{aligned}$$

where α and β are scalars, A is a symmetric matrix, and B and C are general matrices. The **side** argument indicates whether the symmetric matrix A is multiplied on the left or the right side of B . The **uplo** argument indicates whether the lower or upper triangle of A is referenced by the operation.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?symm()`.

More Info: This function's interface is similar to that of `FLA_Symm()`. Please see the description for `FLA_Symm()` for further details.

```
void FLA_Syrk_external( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha,
                       FLA_Obj A, FLA_Obj beta, FLA_Obj C );
```

Purpose: Perform one of the following symmetric rank-k update (SYRK) operations:

$$\begin{aligned} C &:= \beta C + \alpha AA^T \\ C &:= \beta C + \alpha A^T A \end{aligned}$$

where α and β are scalars, C is a symmetric matrix, and A is a general matrix. The **uplo** argument indicates whether the lower or upper triangle of C is referenced and updated by the operation. The **trans** argument allows the computation to proceed as if A were transposed, which results in the alternate rank-k product $A^T A$.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?syrk()`.

More Info: This function's interface is similar to that of `FLA_Syrk()`. Please see the description for `FLA_Syrk()` for further details.

```
void FLA_Syr2k_external( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha,
                        FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C );
```

Purpose: Perform one of the following symmetric rank-2k update (SYR2K) operations:

$$\begin{aligned} C &:= \beta C + \alpha AB^T + \alpha BA^T \\ C &:= \beta C + \alpha A^T B + \alpha B^T A \end{aligned}$$

where α and β are scalars, C is a symmetric matrix, and A and B are general matrices. The **uplo** argument indicates whether the lower or upper triangle of C is referenced and updated by the operation. The **trans** argument allows the computation to proceed as if A and B were transposed, which results in the alternate rank-2k products $A^T B$ and $B^T A$.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?syr2k()`.

More Info: This function's interface is similar to that of `FLA_Syr2k()`. Please see the description for `FLA_Syr2k()` for further details.

```
void FLA_Trmm_external( FLA_Side side, FLA_Uplo uplo, FLA_Trans trans,
                       FLA_Diag diag, FLA_Obj alpha, FLA_Obj A, FLA_Obj B );
```

Purpose: Perform one of the following triangular matrix-matrix multiplication (TRMM) operations:

$$\begin{array}{ll}
 B &:= \alpha AB & B &:= \alpha BA \\
 B &:= \alpha A^T B & B &:= \alpha BA^T \\
 B &:= \alpha \bar{A} B & B &:= \alpha B \bar{A} \\
 B &:= \alpha A^H B & B &:= \alpha BA^H
 \end{array}$$

where α is a scalar, A is a triangular matrix, and B is a general matrix. The **side** argument indicates whether the triangular matrix A is multiplied on the left or the right side of B . The **uplo** argument indicates whether the lower or upper triangle of A is referenced by the operation. The **trans** argument may be used to perform the check as if A were conjugated and/or transposed. The **diag** argument indicates whether the diagonal of A is unit or non-unit.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?trmm()`.

More Info: This function's interface is similar to that of `FLA_Trmm()`. Please see the description for `FLA_Trmm()` for further details.

```
void FLA_Trmmxs_external( FLA_Side side, FLA_Uplo uplo, FLA_Trans trans,
                          FLA_Diag diag, FLA_Obj alpha, FLA_Obj A, FLA_Obj B,
                          FLA_Obj beta, FLA_Obj C );
```

Purpose: Perform one of the following extended triangular matrix-matrix multiplication operations:

$$\begin{array}{ll}
 C &:= \beta C + \alpha AB & C &:= \beta C + \alpha BA \\
 C &:= \beta C + \alpha A^T B & C &:= \beta C + \alpha B A^T \\
 C &:= \beta C + \alpha \bar{A} B & C &:= \beta C + \alpha B \bar{A} \\
 C &:= \beta C + \alpha A^H B & C &:= \beta C + \alpha B A^H
 \end{array}$$

where α and β are scalars, A is a triangular matrix, and B and C are general matrices. The **side** argument indicates whether the triangular matrix A is multiplied on the left or the right side of B . The **uplo** argument indicates whether the lower or upper triangle of A is referenced by the operation. The **trans** argument allows the computation to proceed as if A were conjugated and/or transposed. The **diag** argument indicates whether the diagonal of A is unit or non-unit.

Constraints:

- The numerical datatypes of A , B , and C must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of A , B , and C .
- If **side** equals `FLA_LEFT`, then the number of rows in B and the order of A must be equal; otherwise, if **side** equals `FLA_RIGHT`, then the number of columns in B and the order of A must be equal.
- The dimensions of B and C must be conformal.
- **diag** may not be `FLA_ZERO_DIAG`.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?trmm()`.

Arguments:

side	–	Indicates whether A is multiplied on the left or right side of B .
uplo	–	Indicates whether the lower or upper triangle of A is referenced during the operation.
trans	–	Indicates whether the operation proceeds as if A were conjugated and/or transposed.
diag	–	Indicates whether the diagonal of A is unit or non-unit.
alpha	–	An <code>FLA_Obj</code> representing scalar α .
A	–	An <code>FLA_Obj</code> representing matrix A .
B	–	An <code>FLA_Obj</code> representing matrix B .
beta	–	An <code>FLA_Obj</code> representing scalar β .
C	–	An <code>FLA_Obj</code> representing matrix C .

```
void FLA_Trsm_external( FLA_Side side, FLA_Uplo uplo, FLA_Trans trans, FLA_Diag diag,
                       FLA_Obj alpha, FLA_Obj A, FLA_Obj B );
```

Purpose: Perform one of the following triangular solves with multiple right-hand sides (TRSM):

$$\begin{array}{ll}
 AX &= \alpha B & XA &= \alpha B \\
 A^T X &= \alpha B & XA^T &= \alpha B \\
 \bar{A}X &= \alpha B & X\bar{A} &= \alpha B \\
 A^H X &= \alpha B & XA^H &= \alpha B
 \end{array}$$

and overwrite B with the contents of the solution matrix X as follows:

$$\begin{array}{ll}
 B &:= \alpha A^{-1} B & B &:= \alpha B A^{-1} \\
 B &:= \alpha A^{-T} B & B &:= \alpha B A^{-T} \\
 B &:= \alpha \bar{A}^{-1} B & B &:= \alpha B \bar{A}^{-1} \\
 B &:= \alpha A^{-H} B & B &:= \alpha B A^{-H}
 \end{array}$$

where α is a scalar, A is a triangular matrix, and X and B are general matrices. The **side** argument indicates whether the triangular matrix A is multiplied on the left or the right side of X . The **uplo** argument indicates whether the lower or upper triangle of A is referenced by the operation. The **trans** argument allows the computation to proceed as if A were conjugated and/or transposed. The **diag** argument indicates whether the diagonal of A is unit or non-unit.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?trsm()`.

More Info: This function's interface is similar to that of `FLA_Trsm()`. Please see the description for `FLA_Trsm()` for further details.

```
void FLA_Trmsx_external( FLA_Side side, FLA_Uplo uplo, FLA_Trans trans,
                        FLA_Diag diag, FLA_Obj alpha, FLA_Obj A, FLA_Obj B,
                        FLA_Obj beta, FLA_Obj C );
```

Purpose: Perform one of the following extended triangular solves with multiple right-hand sides:

$$\begin{array}{ll}
 AX &= \alpha B & XA &= \alpha B \\
 A^T X &= \alpha B & XA^T &= \alpha B \\
 \bar{A}X &= \alpha B & X\bar{A} &= \alpha B \\
 A^H X &= \alpha B & XA^H &= \alpha B
 \end{array}$$

and update C with the contents of the solution matrix X as follows:

$$\begin{array}{ll}
 C &:= \beta C + \alpha A^{-1} B & C &:= \beta C + \alpha B A^{-1} \\
 C &:= \beta C + \alpha A^{-T} B & C &:= \beta C + \alpha B A^{-T} \\
 C &:= \beta C + \alpha \bar{A}^{-1} B & C &:= \beta C + \alpha B \bar{A}^{-1} \\
 C &:= \beta C + \alpha A^{-H} B & C &:= \beta C + \alpha B A^{-H}
 \end{array}$$

where α and β are scalars, A is a triangular matrix, and X , B , and C are general matrices. The **side** argument indicates whether the triangular matrix A is multiplied on the left or the right side of X . The **uplo** argument indicates whether the lower or upper triangle of A is referenced by the operation. The **trans** argument allows the computation to proceed as if A were conjugated and/or transposed. The **diag** argument indicates whether the diagonal of A is unit or non-unit.

Constraints:

- The numerical datatypes of A , B , and C must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If α and β are not of datatype `FLA_CONSTANT`, then they must match the datatypes of A , B , and C .
- If **side** equals `FLA_LEFT`, then the number of rows in B and the order of A must be equal; otherwise, if **side** equals `FLA_RIGHT`, then the number of columns in B and the order of A must be equal.
- The dimensions of B and C must be conformal.
- **diag** may not be `FLA_ZERO_DIAG`.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine `?trsm()`.

Arguments:

side	–	Indicates whether A is multiplied on the left or right side of X .
uplo	–	Indicates whether the lower or upper triangle of A is referenced during the operation.
trans	–	Indicates whether the operation proceeds as if A were conjugated and/or transposed.
diag	–	Indicates whether the diagonal of A is unit or non-unit.
alpha	–	An <code>FLA_Obj</code> representing scalar α .
A	–	An <code>FLA_Obj</code> representing matrix A .
B	–	An <code>FLA_Obj</code> representing matrix B .
beta	–	An <code>FLA_Obj</code> representing scalar β .
C	–	An <code>FLA_Obj</code> representing matrix C .

5.8.2 LAPACK operations

```
FLA_Error FLA_Chol_blk_external( FLA_Uplo uplo, FLA_Obj A );
FLA_Error FLA_Chol_unb_external( FLA_Uplo uplo, FLA_Obj A );
```

Purpose: Perform one of the following Cholesky factorizations (CHOL):

$$\begin{aligned} A &\rightarrow LL^T \\ A &\rightarrow U^T U \\ A &\rightarrow LL^H \\ A &\rightarrow U^H U \end{aligned}$$

where A is positive definite. If A is real, then it is assumed to be symmetric; otherwise, if A is complex, then it is assumed to be Hermitian. The operation references and then overwrites the lower or upper triangle of A with the Cholesky factor L or U , depending on the value of `uplo`.

Imp. Notes: `FLA_Chol_blk_external()` and `FLA_Chol_unb_external()` perform their computation by calling external implementations of the LAPACK routines `?potrf()` and `?potf2()`, respectively. The algorithmic variants employed by these routines, as well as the blocksize used by `?potrf()`, are implementation-dependent.

More Info: This function's interface is similar to that of `FLA_Chol()`. Please see the description for `FLA_Chol()` for further details.

```
FLA_Error FLA_Trinv_blk_external( FLA_Uplo uplo, FLA_Diag diag, FLA_Obj A );
FLA_Error FLA_Trinv_unb_external( FLA_Uplo uplo, FLA_Diag diag, FLA_Obj A );
```

Purpose: Perform a triangular matrix inversion (TRINV):

$$A := A^{-1}$$

where A is a general triangular matrix. The operation references and then overwrites the lower or upper triangle of A with its inverse, A^{-1} , depending on the value of `uplo`. The `diag` argument indicates whether the diagonal of A is unit or non-unit.

Imp. Notes: `FLA_Trinv_blk_external()` and `FLA_Trinv_unb_external()` perform their computation by calling external implementations of the LAPACK routines `?trtri()` and `?trti2()`, respectively. The algorithmic variants employed by these routines, as well as the blocksize used by `?trtri()`, are implementation-dependent.

More Info: This function's interface is similar to that of `FLA_Trinv()`. Please see the description for `FLA_Trinv()` for further details.

```
void FLA_Ttmm_blk_external( FLA_Uplo uplo, FLA_Obj A );
void FLA_Ttmm_unb_external( FLA_Uplo uplo, FLA_Obj A );
```

Purpose: Perform one of the following triangular-transpose matrix multiplies (TTMM):

$$\begin{aligned} A &:= L^T L \\ A &:= U U^T \\ A &:= L^H L \\ A &:= U U^H \end{aligned}$$

where A is a triangular matrix with a real diagonal. The operation references and then overwrites the lower or upper triangle of A with its inverse, A^{-1} , depending on the value of `uplo`. The `diag` argument indicates whether the diagonal of A is unit or non-unit.

Imp. Notes: `FLA_Ttmm_blk_external()` and `FLA_Ttmm_unb_external()` perform their computation by calling external implementations of the LAPACK routines `?lauum()` and `?lauu2()`, respectively. The algorithmic variants employed by these routines, as well as the blocksize used by `?lauum()`, are implementation-dependent.

More Info: This function's interface is similar to that of `FLA_Ttmm()`. Please see the description for `FLA_Ttmm()` for further details.

```
void FLA_SPDinv_blk_external( FLA_Uplo uplo, FLA_Obj A );
```

Purpose: Perform a positive definite matrix inversion (SPDINV):

$$A := A^{-1}$$

where A is positive definite. If A is real, then it is assumed to be symmetric; otherwise, if A is complex, then it is assumed to be Hermitian. The operation references and then overwrites the lower or upper triangle of A with its inverse, A^{-1} , depending on the value of `uplo`.

Imp. Notes: `FLA_SPDinv_blk_external()` performs its computation by calling external implementations of the LAPACK routines `?potrf()`, `?trtri()`, and `?lauum()`. The algorithmic variants and blocksizes used by these routines are implementation-dependent.

More Info: This function's interface is similar to that of `FLA_SPDinv()`. Please see the description for `FLA_SPDinv()` for further details.

```
void FLA_Hess_blk_external( FLA_Obj A, FLA_Obj t, int ilo, int ihi );
void FLA_Hess_unb_external( FLA_Obj A, FLA_Obj t, int ilo, int ihi );
```

Purpose: Perform a reduction to upper Hessenberg form operation (HESS) via Householder transformations such that:

$$A = QH Q^T$$

where A is a real matrix, Q is an orthogonal matrix, and H is an upper Hessenberg matrix. Matrix Q is expressed as a product of $(i_{hi} - i_{lo})$ Householder reflectors:

$$Q = H(i_{lo})H(i_{lo} + 1) \dots H(i_{hi} - 1)$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v^T$$

where τ is a real scalar and v is a real vector of length n . If ν_j is the j th element of v , we may describe v such that, for a given $H(i)$, the element $\nu_{i+1} = 1$ while elements $\nu_{0:i}$ and $\nu_{i_{hi}+1:n-1}$ are zero, with other entries holding non-zero values. The operation overwrites the the upper triangle and first subdiagonal of A with H . However, the matrix Q is not stored explicitly. Instead, the operation stores the τ associated with $H(i)$ to the i th element of vector t , and also stores the non-unit, non-zero entries $\nu_{i+2:i_{hi}}$ of Householder reflectors $H_{i_{lo}}$ through $H_{i_{hi}-2}$ to the elements below the first subdiagonal of A . More specifically, entries $\nu_{i+2:i_{hi}}$ are stored to elements $i + 2 : i_{hi}$ of the i th column of matrix A .

Imp. Notes: `FLA_Hess_blk_external()` and `FLA_Hess_unb_external()` perform their computation by calling external implementations of the LAPACK routines `?gehrd()` and `?gehd2()`, respectively. The algorithmic variants employed by these routines, as well as the blocksize used by `?gehrd()`, are implementation-dependent.

More Info: This function's interface is similar to that of `FLA_Hess()`. Please see the description for `FLA_Hess()` for further details.

```
FLA_Error FLA_LU_nopiv_blk_external( FLA_Obj A );
FLA_Error FLA_LU_nopiv_unb_external( FLA_Obj A );
```

Purpose: Perform an LU factorization without pivoting (LUNOPIV):

$$A \rightarrow LU$$

where A is a general matrix, L is lower triangular (or lower trapezoidal if $m > n$) with a unit diagonal, and U is upper triangular (or upper trapezoidal if $m < n$). The operation overwrites the strictly lower triangular portion of A with L and the upper triangular portion of A with U . The diagonal elements of L are not stored.

Imp. Notes: `FLA_LU_nopiv_blk_external()` and `FLA_LU_nopiv_unb_external()` perform their computation by calling internal implementations of LAPACK-like routines `?getnf()` and `?getn2()`, respectively. The algorithmic variants employed by these routines, as well as the blocksize used by `?getnf()`, are identical to those used by `?getrf()` and `?getf2()`, which are implementation-dependent.

More Info: This function's interface is similar to that of `FLA_LU_nopiv()`. Please see the description for `FLA_LU_nopiv()` for further details.


```
FLA_Error FLA_LU_piv_blk_external( FLA_Obj A, FLA_Obj p );
FLA_Error FLA_LU_piv_unb_external( FLA_Obj A, FLA_Obj p );
```

Purpose: Perform an LU factorization with partial row pivoting (LUPIV):

$$A \rightarrow PLU$$

where A is a general matrix, L is lower triangular (or lower trapezoidal if $m > n$) with a unit diagonal, U is upper triangular (or upper trapezoidal if $m < n$), and P is a permutation matrix. The operation overwrites the strictly lower triangular portion of A with L and the upper triangular portion of A with U . The diagonal elements of L are not stored.

Imp. Notes: `FLA_LU_piv_blk_external()` and `FLA_LU_piv_unb_external()` perform their computation by calling external implementations of the LAPACK routines `?getrf()` and `?getf2()`, respectively. The algorithmic variants employed by these routines, as well as the blocksize used by `?getrf()`, are implementation-dependent.

More Info: This function's interface is similar to that of `FLA_LU_nopiv()`. Please see the description for `FLA_LU_nopiv()` for further details.

```
void FLA_QR_blk_external( FLA_Obj A, FLA_Obj t );
void FLA_QR_unb_external( FLA_Obj A, FLA_Obj t );
```

Purpose: Perform a QR factorization (QR):

$$A \rightarrow QR$$

where A is a general matrix, R is upper triangular (or upper trapezoidal if $m < n$), and Q is the product of $k = \min(m, n)$ Householder reflectors:

$$Q = H(0)H(1)\dots H(k-1)$$

Each $H(i)$ has the form

$$H(i) = I - \tau vv^T$$

where τ is a scalar and v is a vector of length m . If v_j is the j th element of v , we may describe v such that, for a given $H(i)$, the element $v_i = 1$ while elements $v_{0:i-1}$ are zero, with other entries holding non-zero values. The operation overwrites the upper triangle (or upper trapezoid) of A with R . However, the matrix Q is not stored explicitly. Instead, the operation stores the τ associated with $H(i)$ to the i th element of vector t , and also stores the non-unit, non-zero entries $v_{i+1:m-1}$ of Householder reflectors H_0 through H_k column-wise below the diagonal of A . More specifically, entries $v_{i+1:m-1}$ are stored to elements $i+1 : m-1$ of the i th column of matrix A .

Constraints:

- The numerical datatypes of A and t must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The length of t must be $\min(m, n)$ where A is $m \times n$.

Imp. Notes: `FLA_QR_blk_external()` and `FLA_QR_unb_external()` perform their computation by calling external implementations of the LAPACK routines `?geqrf()` and `?geqr2()`, respectively. The algorithmic variants employed by these routines, as well as the blocksize used by `?geqrf()`, are implementation-dependent.

Arguments:

- | | | |
|----------------|---|---|
| <code>A</code> | – | An <code>FLA_Obj</code> representing matrix A . |
| <code>t</code> | – | An <code>FLA_Obj</code> representing vector t . |

```
void FLA_LQ_blk_external( FLA_Obj A, FLA_Obj t );
void FLA_LQ_unb_external( FLA_Obj A, FLA_Obj t );
```

Purpose: Perform an LQ factorization (LQ):

$$A \rightarrow LQ$$

where A is a general matrix, L is a lower triangular (or lower trapezoidal if $m > n$), and Q is the product of $k = \min(m, n)$ Householder reflectors:

$$Q = H(k-1) \dots H(1)H(0)$$

Each $H(i)$ has the form

$$H(i) = I - \tau v v^T$$

where τ is a scalar and v is a vector of length n . If ν_j is the j th element of v , we may describe v such that, for a given $H(i)$, the element $\nu_i = 1$ while elements $\nu_{0:i-1}$ are zero, with other entries holding non-zero values. The operation overwrites the lower triangle (or lower trapezoid) of A with L . However, the matrix Q is not stored explicitly. Instead, the operation stores the τ associated with $H(i)$ to the i th element of vector t , and also stores the non-unit, non-zero entries $\nu_{i+1:n-1}$ of Householder reflectors H_0 through H_k row-wise above the diagonal of A . More specifically, entries $\nu_{i+1:n-1}$ are stored to elements $i+1 : n-1$ of the i th row of matrix A .

Constraints:

- The numerical datatypes of A and t must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The length of t must be $\min(m, n)$ where A is $m \times n$.

Imp. Notes: `FLA_LQ_blk_external()` and `FLA_LQ_unb_external()` perform their computation by calling external implementations of the LAPACK routines `?gelqf()` and `?gelq2()`, respectively. The algorithmic variants employed by these routines, as well as the blocksize used by `?gelqf()`, are implementation-dependent.

Arguments:

- | | | |
|----------------|---|---|
| <code>A</code> | – | An <code>FLA_Obj</code> representing matrix A . |
| <code>t</code> | – | An <code>FLA_Obj</code> representing vector t . |

```
void FLA_QR_UT_blk_external( FLA_Obj A, FLA_Obj t );
void FLA_QR_UT_unb_external( FLA_Obj A, FLA_Obj t );
```

Purpose: Perform a QR factorization with the UT transform (QRUT). The resulting Householder vectors stored below the diagonal of A should only be used with other UT transform-aware operations, such as `FLA_Accum_T_UT()`.

Imp. Notes: `FLA_QR_UT_blk_external()` and `FLA_QR_UT_unb_external()` perform their computation by calling internal implementations of LAPACK-like routines `?geqrfut()` and `?geqr2ut()`, respectively. The algorithmic variants employed by these routines, as well as the blocksize used by `?geqrfut()`, are identical to those used by `?geqrf()` and `?geqr2()`, which are implementation-dependent.

More Info: This function's interface is similar to that of `FLA_QR_UT()`. Please see the description for `FLA_QR_UT()` for further details.

```
void FLA_QR_UT_Accum_T_unb_external( FLA_Obj A, FLA_Obj T );
```

Purpose: Perform a QR factorization with the UT transform using an unblocked algorithm while accumulating the upper triangular factor T of the block Householder transformation H . See the descriptions of `FLA_QR_UT_unb_external()` and `FLA_Accum_T_UT_unb_external()` for more details.

Imp. Notes: This function is currently implemented simply as:

```
    FLA_QR_UT_unb_external( A, t );
    FLA_Accum_T_UT_unb_external( FLA_FORWARD, FLA_COLUMNWISE, A, t, T );
```

where t is a temporary vector.

```
void FLA_LQ_UT_blk_external( FLA_Obj A, FLA_Obj t );
```

```
void FLA_LQ_UT_unb_external( FLA_Obj A, FLA_Obj t );
```

Purpose: Perform an LQ factorization with the UT transform (LQUT). The resulting Householder vectors stored above the diagonal of A should only be used with other UT transform-aware operations, such as `FLA_Accum_T_UT()`.

Imp. Notes: `FLA_LQ_UT_blk_external()` and `FLA_LQ_UT_unb_external()` perform their computation by calling internal implementations of LAPACK-like routines `?gelqfut()` and `?gelq2ut()`, respectively. The algorithmic variants employed by these routines, as well as the blocksize used by `?gelqfut()`, are identical to those used by `?gelqf()` and `?gelq2()`, which are implementation-dependent.

More Info: This function's interface is similar to that of `FLA_LQ_UT()`. Please see the description for `FLA_LQ_UT()` for further details.

```
void FLA_LQ_UT_Accum_T_unb_external( FLA_Obj A, FLA_Obj T );
```

Purpose: Perform an LQ factorization with the UT transform using an unblocked algorithm while accumulating the upper triangular factor T of the block Householder transformation H . See the descriptions of `FLA_LQ_unb_external()` and `FLA_Accum_T_UT_unb_external()` for more details.

Imp. Notes: This function is currently implemented simply as:

```
    FLA_LQ_UT_unb_external( A, t );
    FLA_Accum_T_UT_unb_external( FLA_FORWARD, FLA_ROWWISE, A, t, T );
```

where t is a temporary vector.

```

void FLA_Apply_Q_blk_external( FLA_Side side, FLA_Trans trans, FLA_Store storev,
                               FLA_Obj A, FLA_Obj t, FLA_Obj B );
void FLA_Apply_Q_unb_external( FLA_Side side, FLA_Trans trans, FLA_Store storev,
                               FLA_Obj A, FLA_Obj t, FLA_Obj B );

```

Purpose: Apply a matrix Q (or Q^T or Q^H) to a general matrix B from either the left or the right:

$$\begin{aligned}
 B &:= QB & B &:= BQ \\
 B &:= Q^T B & B &:= BQ^T \\
 B &:= Q^H B & B &:= BQ^H
 \end{aligned}$$

where Q is the orthogonal (or, if A is complex, unitary) matrix implicitly defined by the Householder vectors stored in matrix A and the τ values stored in vector t . The **side** argument indicates whether Q is applied to B from the left or the right. The **trans** argument indicates whether Q or Q^T (or Q^H) is applied to B . The **storev** argument indicates whether the Householder vectors which define Q are stored column-wise (in the strictly lower triangle) or row-wise (in the strictly upper triangle) of A .

Imp. Notes: `FLA_Apply_Q_blk_external()` and `FLA_Apply_Q_unb_external()` perform their computation by calling external implementations of the LAPACK routines `?ormqr()`/`?unmqr()`/`?ormlq()`/`?unmlq()` and `?orm2r()`/`?unm2r()`/`?orml2()`/`?unml2()`, respectively. The algorithmic variants employed by these routines, as well as the blocksizes used by `?ormqr()`, `?unmqr()`, `?ormlq()`, and `?unmlq()` are implementation-dependent.

Constraints:

- The numerical datatypes of A , t , and B must be identical and floating-point, and must not be `FLA_CONSTANT`.
- If **side** equals `FLA_LEFT`, then the number of rows in B and the order of A must be equal; otherwise, if **side** equals `FLA_RIGHT`, then the number of columns in B and the order of A must be equal.
- If A is real, then **trans** must be `FLA_NO_TRANSPOSE` or `FLA_TRANSPOSE`; otherwise if A is complex, then **trans** must be `FLA_NO_TRANSPOSE` or `FLA_CONJ_TRANSPOSE`.
- The length of t must be $\min(m, n)$ where A is $m \times n$.

Arguments:

- | | |
|---------------|---|
| side | – Indicates whether Q (or Q^T or Q^H) is multiplied on the left or right side of B . |
| trans | – Indicates whether the operation proceeds as if Q were transposed (or conjugate-transposed). |
| storev | – Indicates whether the vectors stored within A are stored column-wise or row-wise. |
| A | – An <code>FLA_Obj</code> representing matrix A . |
| t | – An <code>FLA_Obj</code> representing vector t . |
| B | – An <code>FLA_Obj</code> representing matrix B . |

```
void FLA_Sylv_unb_external( FLA_Trans transa, FLA_Trans transb, FLA_Obj isgn,
                           FLA_Obj A, FLA_Obj B, FLA_Obj C, FLA_Obj scale );
```

Purpose: Solve one of the following triangular Sylvester equations (SYLV):

$$\begin{aligned} AX &\pm XB &= C \\ AX &\pm XB^T &= C \\ A^T X &\pm XB &= C \\ A^T X &\pm XB^T &= C \end{aligned}$$

where A and B are real upper triangular matrices and C is a real general matrix. If A , B , and C are complex matrices, then the possible operations are:

$$\begin{aligned} AX &\pm XB &= C \\ AX &\pm XB^H &= C \\ A^H X &\pm XB &= C \\ A^H X &\pm XB^H &= C \end{aligned}$$

where A and B are complex upper triangular matrices and C is a complex general matrix. The operation references and then overwrites matrix C with the solution matrix X . The `isgn` argument is a scalar integer object that indicates whether the \pm sign between terms is a plus or a minus. The `scale` argument is not referenced and set to 1.0 upon completion.

Imp. Notes: `FLA_Sylv_blk_external()` and `FLA_Sylv_unb_external()` perform their computation by calling an external implementation of the LAPACK routine `?trsyl()`. The algorithmic variant employed by this routine is implementation-dependent.

More Info: This function's interface is similar to that of `FLA_Sylv()`. Please see the description for `FLA_Sylv()` for further details.

5.8.3 LAPACK-related utility functions

```
void FLA_Accum_T_UT_unb_external( FLA_Direct direct, FLA_Store storev,
                                  FLA_Obj V, FLA_Obj t, FLA_Obj T );
```

Purpose: Form the triangular factor T of a block Householder transform from a set of Householder reflectors using the UT transform. See the description `FLA_Accum_T_UT()` for more details.

Notes: This function should only be used with matrices V and vectors t that were filled by another UT transform-aware operation, such as `FLA_QR_UT_blk_external()` or `FLA_LQ_UT_blk_external()`.

Int. Notes: Unlike `FLA_Accum_T_UT()`, which potentially computes several smaller triangular factors, depending on the length of T , this routine computes a single triangular factor of order $k = \dim(t) = \min(m, n)$.

Constraints:

- The numerical datatypes of V , t , and T must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The length of t and the order of T must be $\min(m, n)$ where V is $m \times n$.

Imp. Notes: This function uses an internal implementation of a LAPACK-like routine `?larftut()`.

Arguments:

- | | | |
|---------------------|---|---|
| <code>direct</code> | – | Indicates whether H is formed from the forward or backward product of its constituent Householder reflectors. |
| <code>storev</code> | – | Indicates whether the vectors stored within V are stored column-wise below the diagonal or row-wise above the diagonal. |
| V | – | An <code>FLA_Obj</code> representing matrix V . |
| t | – | An <code>FLA_Obj</code> representing vector t . |
| T | – | An <code>FLA_Obj</code> representing matrix T . |

```
void FLA_Swap_rows( FLA_Obj A, dim_t k1, dim_t k2, FLA_Obj p );
```

Purpose: Swap the rows of a matrix A according to the pivot vector p . Only the row swaps for elements k_1 through k_2 are performed, where k_1 and k_2 are zero-based indices into p .

Notes: The pivot vector p must contain pivot values that conform to `libflame` pivot indexing. If the pivot vector was filled using an LAPACK routine, it must first be converted to `libflame` pivot indexing with `FLA_Shift_pivots_to()` before it may be used with `FLA_Swap_rows()`. Please see the description for `FLA_LU_piv()` in Section 5.7.2 for details on the differences between LAPACK-style pivot vectors and `libflame` pivot vectors.

Constraints:

- The numerical datatype of A must be floating-point, and must not be `FLA_CONSTANT`.
- The numerical datatype of p must be integer, and must not be `FLA_CONSTANT`.

Imp. Notes: This function uses an external implementation of the LAPACK routine `?laswp()`.

Arguments:

- | | | |
|-------|---|---|
| A | – | An <code>FLA_Obj</code> representing matrix A . |
| k_1 | – | The index of the first element of p for which a row swap in A will be done. |
| k_2 | – | The index of the last element of p for which a row swap in A will be done. |
| p | – | An <code>FLA_Obj</code> representing vector p . |

5.9 LAPACK compatibility (liblapack2flame)

As part of the `libflame` package, we provide a companion library, which we call `liblapack2flame`, that allows the user to take advantage of some of the performance benefits of `libflame` without rewriting their code to use the native FLAME/C or FLASH APIs. More specifically, `liblapack2flame` consists of interfaces that map LAPACK routine invocations to their corresponding `libflame` implementations. For example, linking your application to `liblapack2flame` at link-time would cause any invocation of `dpotrf()` to invoke the code associated with `FLA_Chol()`. This compatibility layer does not come without overhead, though. Matrix data must be attached to objects, which must be created within the LAPACK interface routines, and then these objects must be freed before returning. These operations may incur a small but noticeable cost, especially if portions of the application are being profiled for performance measurement.

5.9.1 Supported routines

This section summarizes the LAPACK interfaces currently supported within `liblapack2flame`. Table 5.2 lists all LAPACK interfaces which map directly to functionality implemented within `libflame`.

Operation	datatypes supported	LAPACK interface	Maps to...
Cholesky factorization	sdcz	?potrf()	FLA_Chol()
LU factorization with partial row pivoting	sdcz	?getrf()	FLA_LU_piv()
QR factorization	sdcz	?geqrf()	FLA_QR()
LQ factorization	sdcz	?gelqf()	FLA_LQ()
Triangular matrix inversion	sdcz	?trtri()	FLA_Trinv()
Triangular-transpose matrix multiply	sdcz	?lauum()	FLA_Ttmm()
Triangular Sylvester equation solve	sdcz	?trsyl()	FLA_Sylv()
Linear system solve via QR or LQ factorization	d	?gels()	N/A

Table 5.2: A list of LAPACK interfaces supported directly within `liblapack2flame`.

However, many routines within LAPACK utilize other routines as subproblems toward their larger goal. Table 5.3 summarizes the LAPACK interfaces which indirectly depend upon routines listed in 5.2.

These routines may exhibit improved performance by virtue of the fact that they utilize higher-performing implementations of their suboperations. However, the performance improvement may be markedly lower than that observed when invoking the routines in Table 5.2 directly.

Directions on how to link `liblapack2flame` to your application may be found in Section 2.7.

Operation	datatypes supported	LAPACK interface	Depends on...
Linear SPD system solve via Cholesky fact.	d	?posv()	?potrf()
Linear SPD system solve via Cholesky fact. (expert)	d	?posvx()	?potrf()
Linear system solve via LU factorization	d	?gesv()	?getrf()
Linear system solve via LU factorization (expert)	d	?gesvx()	?getrf()
Linear least squares solve (via simple SVD)	d	?gelss()	?geqrf() ?gelqf()
Linear least squares solve (via divide-and-conquer SVD)	d	?gelsd()	?geqrf() ?gelqf()
Linear equality-constrained least squares solve (via GRQ factorization)	d	?gglse()	?
Linear least squares solve (via complete orthogonal factorization)	d	?gelsy()	?
General Gauss-Markov linear model solve	d	?ggglm()	?
General SD eigenproblem	d	?sygv()	?potrf()
General SD eigenproblem (expert)	d	?sygvx()	?potrf()
General SD eigenproblem, divide-and-conquer	d	?sygvd()	?potrf()
General SD eigenproblem and Schur decomposition for pair of matrices	d	?gges()	?geqrf()
General SD eigenproblem and Schur decomposition for pair of matrices (expert)	d	?ggesx()	?geqrf()
General SD eigenproblem and Schur decomposition for pair of matrices (deprecated)	d	?gegs()	?geqrf()
SVD, simple algorithm	d	?gesvd()	?geqrf() ?gelqf()
SVD, divide-and-conquer algorithm	d	?gesdd()	?geqrf() ?gelqf()
QR factorization with column pivoting	d	?geqp3()	?geqrf()
QR factorization for two matrices	d	?ggqrf()	?geqrf()
RQ factorization for two matrices	d	?ggrqf()	?geqrf()
General eigenvalue and Schur decomposition	d	?gees()	?gehrd()
General eigenvalue and Schur decomposition (expert)	d	?geesx()	?gehrd()
General eigenvalue decomposition	d	?geev()	?gehrd()
General eigenvalue decomposition (expert)	d	?geevx()	?gehrd()
General EVD for pair of matrices	d	?ggeev()	?gehrd()
General EVD for pair of matrices (expert)	d	?ggeevx()	?gehrd()
General EVD for pair of matrices (deprecated)	d	?gegv()	?geqrf()
SPD inversion from pre-computed Cholesky factor	d	?potri()	?trtri() ?lauum()
General matrix inversion from pre-computed LU factors	d	?getri()	?trtri()

Table 5.3: A list of LAPACK interfaces supported indirectly within liblapack2flame. In other words, these routines depend on functionality that has been implemented natively within libflame.

Chapter 6

Developer Application Programming Interfaces

This chapter describes APIs of interest to developers of `libflame`, including advanced users seeking to extend existing functionality to suit their own application.

6.1 Locks

There are a few instances, most notably within the `libflame` implementation of SuperMatrix, where locks are needed to ensure that certain data structures are updated synchronously by multiple threads. `libflame` abstracts the implementation of the locking mechanism from the user by exporting a general interface that operates upon an internally defined `FLA_Lock` structure. This structure contains all of the information needed to identify the actual lock, however it may be defined by the implementation. `libflame` will define its implementation in terms of whichever multithreading interface is enabled. See Section 2.4.1 for more information on how to specify the type of multithreading at configure-time.

Note that this API provides only basic locking functionality. The functions do *not* return any status value, and thus the caller cannot check whether the function succeeded or not. This was done to simplify the implementation, and also because our primary application, SuperMatrix, did not require the ability to recover from the kinds of errors that might occur beyond the control of the user, such as system errors.

6.1.1 API

```
void FLA_Lock_init( FLA_Lock* lock_ptr );
```

Purpose: Initialize the lock structure pointed to by `FLA_Lock`. Upon successful return, the state of the lock becomes initialized and unlocked.

Notes: Attempting to initialize a lock that has already been initialized (and not yet released) may result in undefined behavior.

Constraints:

- `lock_ptr` may not be NULL.

Arguments:

`lock_ptr` – A pointer to an `FLA_Lock` structure.

```
void FLA_Lock_acquire( FLA_Lock* lock_ptr );
```

Purpose: Attempt to acquire the lock pointed to by `lock_ptr`. If the lock is unavailable (if its state is already locked), the call blocks and returns only upon successful acquisition of the lock.

Constraints:

- `lock_ptr` may not be NULL.

Arguments:

`lock_ptr` – A pointer to an `FLA_Lock` structure.

```
void FLA_Lock_release( FLA_Lock* lock_ptr )
```

Purpose: Release the lock associated with the structure pointed to by `FLA_Lock`.

Notes: Attempting to release a lock that is uninitialized or that has not yet been acquired may result in undefined behavior.

Constraints:

- `lock_ptr` may not be NULL.

Arguments:

`lock_ptr` – A pointer to an `FLA_Lock` structure.

```
void FLA_Lock_destroy( FLA_Lock* lock_ptr );
```

Purpose: Destroy the the lock structure pointed to by `FLA_Lock`. This causes all system resources associated with the lock that had been previously allocated by `FLA_Lock_init()` to be freed. Upon returning, the state of the lock becomes uninitialized.

Notes: Attempting to destroy a lock that is currently in the locked state may result in undefined behavior.

Constraints:

- `lock_ptr` may not be NULL.

Arguments:

`lock_ptr` – A pointer to an `FLA_Lock` structure.

6.2 Memory management

```
void* FLA_malloc( size_t size );
```

Purpose: Request a pointer to `size` bytes of heap-allocated memory from the system. Note that a value of zero for `size` will guarantee that `NULL` is returned.

Notes: The programmer is encouraged to use `FLA_malloc()` instead of calling `malloc()` directly. Using `FLA_malloc()` and `FLA_free()` allows `libflame` to output via standard error the balance of allocations and releases when `FLA_Finalize()` is called, which provides a basic memory leak detection.

Imp. Notes: If `libflame` was configured with `--enable-memory-alignment=N`, then memory will be allocated using `posix_memalign()` using `N` as the alignment factor. Otherwise, `malloc()` is used, which typically only guarantees memory alignment at 8-byte boundaries.

Caveats: If by chance `malloc()` (or `posix_memalign()`) fails to allocate the requested number of bytes, the library raises an abort signal and the program is ended. This may seem like overkill, and it probably is. But it ensures that this situation does not go unreported. Besides, in the unlikely event that `malloc()` does return a `NULL` pointer, it is most likely because the memory heap is exhausted, which would prevent most programs from running correctly (or at all).

Returns: A `void*` pointer to a heap-allocated region of memory `size` bytes long.

Arguments:

`size` – The number of bytes to allocate.

```
void* FLA_realloc( void* old_ptr, size_t size );
```

Purpose: Request a reallocation of previously-allocated memory such that the new region is `size` bytes in length and contains the original contents of the region pointed to by `old_ptr`.

Imp. Notes: This function does not guarantee adherence to the library-wide memory alignment factor set during configuration via the `--enable-memory-alignment=N` option, if it was given. We fundamentally cannot implement our own version of `realloc()` in user-space because we cannot know how much memory was allocated at `old_ptr`. This information is needed if the original contents are to be copied over to the new memory region. Thus, `FLA_realloc()` is implemented with `realloc()`, which guarantees only 8-byte memory alignment on most systems.

Returns: A `void*` pointer to a heap-allocated region of memory `size` bytes long.

Arguments:

`old_ptr` – A pointer to the region of memory the user wishes to reallocate to `size` bytes.
`size` – The number of bytes to allocate for the new region.

```
void FLA_free( void* ptr );
```

Purpose: Release a heap-allocated region of memory back to the system. Note that passing a value of NULL for `ptr` will cause `FLA_free()` to return immediately without performing any action.

Notes: The programmer is encouraged to use `FLA_free()` instead of calling `free()` directly. Using `FLA_malloc()` and `FLA_free()` allows `libflame` to output via standard error the balance of allocations and releases when `FLA_Finalize()` is called, which provides a basic memory leak detection.

Arguments:

`ptr` – A pointer to a region of memory previously allocated by `FLA_malloc()` (or `FLA_realloc()`).

6.3 Object creation

```
FLA_Error FLA_Obj_create_ext( FLA_Datatype datatype, FLA_Elemtypes elemtype, dim_t m,
                             dim_t n, FLA_Obj* obj );
```

Purpose: Create a new object using an extended FLASH-aware interface. Upon returning, `obj` points to a valid heap-allocated $m \times n$ object.

Notes: The total size of the underlying allocated array depends on the value of `elemtype`. If the elements are requested to be `FLA_SCALAR`, then the size of each element is determined by the value of `datatype`. If the elements are of type `FLA_MATRIX`, then each element is allocated to store an `FLA_Obj`.

Returns: `FLA_SUCCESS`

Arguments:

`datatype` – A constant corresponding to the numerical datatype requested.
`elemtype` – A constant corresponding to the object element type requested.
`m` – The number of rows to be created in new object.
`n` – The number of columns to be created in the new object.
`obj`
 (on entry) – A pointer to an uninitialized `FLA_Obj`.
 (on exit) – A pointer to a new `FLA_Obj` parameterized by `m`, `n`, `elemtype`, and `datatype`.

6.4 SuperMatrix

```
void FLASH_Queue_init( void );
```

Purpose: Initialize SuperMatrix. This function is normally called from within `FLA_Init()`.

```
void FLASH_Queue_finalize( void );
```

Purpose: Finalize SuperMatrix. This function is normally called from within `FLA_Finalize()`.

```
unsigned int FLASH_Queue_get_thread_id( void );
```

Purpose: Query the ID number of the calling thread. The thread ID ranges from zero to $t-1$ where t is the total number of SuperMatrix threads, equal to the unsigned integer returned by `FLASH_Queue_get_num_threads()`.

Dev. notes: This function is not yet implemented.

Returns: An unsigned integer representing the thread ID number of the calling thread.

6.5 Control trees

6.5.1 Motivation

While `libflame` was in its early stages of development, we encountered a basic and recurring problem: coding blocked FLAME/C algorithms forced us to statically specify the implementation used when invoking algorithmic subproblems. Sometimes we chose to hard-code a function call an unblocked implementation that was itself coded with the FLAME/C API. Other times we chose to invoke external BLAS and/or unblocked LAPACK implementations via FLAME wrappers. And still other situations, such as those encountered when using FLASH and algorithms-by-blocks, call for us to invoke yet another blocked routine, creating more than one level of “recursion” in the overall algorithm.

The problem becomes more unavoidable. Consider that not all situations call for using the same set of algorithmic variants. Perhaps for small standalone instances of SYRK, variant 5 works well, but when the SYRK operation is a subproblem within a larger Cholesky factorization, then variant 2 is more appropriate. How can we handle both of these cases while statically coding the choice of implementation for blocked algorithm subproblems? The most straightforward and naive solution would be to duplicate the code as many times as there are different situations. It is not difficult to see that this would quickly become a nightmare for the maintainers of the library.

So, in summary, we wish to code our algorithms in such a manner that a subproblem operation is specified *without* binding it to a particular implementation, and in such a manner that we may only maintain *one* copy of each blocked algorithm.

6.5.2 The solution in `libflame`

The aforementioned problem is addressed in `libflame` using a technique we call *control trees*. The general idea is simple: encode control information *a priori* into a tree structure that is passed into algorithmic subproblems and decoded by internal functions that remain hidden from the user-level API. This approach has five separate but related aspects, all of which require us to extend the original FLAME/C API in some way:

- **Define control tree node structures and API.** Our solution is built upon a tree structure where internal nodes encapsulate information used by blocked algorithms and leaf nodes specify external implementations. Specifically, all individual control tree node structures are defined with blocksize and variant fields, which allow us to specify which algorithmic variant to execute and what blocksize to use within that algorithm. Each structure will also have an matrix type field which will allow us to handle both flat and arbitrary depth hierarchical matrices. We define a control tree structure for each operation that we wish to support (`fla_syrk_t` for SYRK, `fla_chol_t` for CHOL, etc.) These structures contain the standard three fields and also some number of fields which may contain pointers to child nodes. The number and types of these fields depend on the operation for which the control tree is being defined, but in general, the set of fields present will be enough to handle all algorithmic variants provided by `libflame`. Lastly, we must define an API that allows the programmer to easily create individual control tree structures and build trees up from child nodes.

- **Control trees created at library startup.** Control trees are created dynamically via the structure creation interface and stored on the heap. The default set used internally by `libflame` is instantiated at the time that the library is initialized, with one class of trees being configured for flat matrices, and another for hierarchical matrices. Within each class, several different trees may be created for a given operation, depending on the desired execution characteristics. A different tree may be created for problem sizes deemed to be “small” and those considered to be “large”, which would vary as a function of the blocksize and cache size. Alternatively, if the tree is structured properly, the same tree may be used for all problem sizes and shapes with only a very small additional cost in overhead incurred from the extra levels of blocked algorithms. Pointers to the root nodes of the trees are stored in global variable space. The default set of control trees is destroyed at library shutdown, in `FLA_Finalize()`.
- **Control tree is selected within operation front-ends.** The default set of control trees are used within operation “front-end” routines. These routines are defined as user-level computational routines for Level-3 BLAS and LAPACK-level operations that are designed for “global” problems, not subproblems. In other words, front-ends are for end users only and are not called internally by `libflame` developers. Examples of front-end routines are: `FLA_Syrk()`, `FLA_Trsm()`, `FLA_Chol()`, and `FLA_SPDinv()`. The root node pointers are accessed via `extern` declarations within the files that define the front-end routines. There, the appropriate tree is selected, if more than one exists, and the root node pointer is passed down into the operation’s internal “back-end”.
- **Internal back-ends handle parameters and decode trees.** Internal back-end functions must be defined in order to parse and decode the control tree for a particular operation. The implementation handles cases where (1) computation should execute immediately, such as for flat matrices or for the leaf matrices of hierarchical matrices when SuperMatrix is disabled, (2) more recursion is necessary, in order to handle arbitrary depth hierarchical matrices, and (3) the library should enqueue tasks for parallel execution via SuperMatrix. The back-end also parses the parameters defined by the operation, such as `side`, `uplo`, and `trans` arguments, and then the appropriate variant or external implementation is called depending on the `variant` field of the control tree node. In the context of flat matrices, a `variant` field equal to `FLA_SUBPROBLEM` refers to a node that induces execution of an external implementation. For hierarchical matrices, the `FLA_SUBPROBLEM` variant refers to nodes that may or may not cause further recursion, and reuse of the control tree, to reach the leaf levels of the hierarchy. Otherwise, if a blocked algorithmic variant is called, the control tree is passed on.
- **Algorithmic subproblems invoke internal back-ends.** The modified blocked algorithms feature a `cntl` control tree pointer argument instead of the integer `nb.alg` argument. Recall that the pointer will refer to a control tree structure which contains all the information necessary to specify the execution of the blocked algorithmic variant, including the blocksize. The statement which computes the blocksize is replaced by one which uses a new routine, `FLA_Determine_blocksize()`. The subproblems are replaced with corresponding calls to the internal back-end routine for the operation in question. C preprocessor macros are used to access the fields within the `cntl` argument, specifically to extract the blocksize argument and the pointers to the child nodes of the current control tree node. The child nodes are passed in as the last argument to the internal back-ends, and recursion continues.

6.5.3 Structure fields

There are three fields common to every control tree, regardless of its type.

- **matrix_type.** The `matrix_type` field denotes the type of matrices, flat or hierarchical, that the control tree is to assume. A matrix type of `FLA_HIER` allows hierarchical matrices, where the matrices may be of arbitrary depth. A matrix type of `FLA_FLAT` implies that the matrix operands will be flat matrix objects. Though `libflame` objects contain the `elemtype` field in each node in the matrix hierarchy, the `matrix_type` field is needed because the control trees for flat and hierarchical matrices differ. Control trees for flat matrices explicitly prescribe the execution for every level of algorithmic recursion. However, control trees for hierarchical matrices must allow recursion to an arbitrary depth. Thus, we leave it to the internal back-end to detect when the leaves of the hierarchy are reached and stop

GEMM	SYMM	SYRK
<pre> struct FLA_Gemm_s { FLA_Exec exec_type; int variant; fla_blocksize_t* blocksize; struct FLA_Gemm_s* sub_gemm; }; typedef struct FLA_Gemm_s fla_gemm_t; </pre>	<pre> struct FLA_Symm_s { FLA_Exec exec_type; int variant; fla_blocksize_t* blocksize; struct FLA_Symm_s* sub_symm; struct FLA_Gemm_s* sub_gemm1; struct FLA_Gemm_s* sub_gemm2; }; typedef struct FLA_Symm_s fla_symm_t; </pre>	<pre> struct FLA_Syrk_s { FLA_Exec exec_type; int variant; fla_blocksize_t* blocksize; struct FLA_Syrk_s* sub_syrk; struct FLA_Gemm_s* sub_gemm; }; typedef struct FLA_Syrk_s fla_syrk_t; </pre>
TRSM	CHOL	TRINV
<pre> struct FLA_Trsm_s { FLA_Exec exec_type; int variant; fla_blocksize_t* blocksize; struct FLA_Trsm_s* sub_trsm; struct FLA_Gemm_s* sub_gemm; }; typedef struct FLA_Trsm_s fla_trsm_t; </pre>	<pre> struct FLA_Chol_s { FLA_Exec exec_type; int variant; fla_blocksize_t* blocksize; struct FLA_Chol_s* sub_chol; struct FLA_Syrk_s* sub_syrk; struct FLA_Herk_s* sub_herk; struct FLA_Trsm_s* sub_trsm; struct FLA_Gemm_s* sub_gemm; }; typedef struct FLA_Chol_s fla_chol_t; </pre>	<pre> struct FLA_Trinv_s { FLA_Exec exec_type; int variant; fla_blocksize_t* blocksize; struct FLA_Trinv_s* sub_trinv; struct FLA_Gemm_s* sub_gemm; struct FLA_Trsm_s* sub_trsm; struct FLA_Trsm_s* sub_trsm1; struct FLA_Trsm_s* sub_trsm2; }; typedef struct FLA_Trinv_s fla_trinv_t; </pre>

Figure 6.1: Control tree structure definitions for various operations.

recursion accordingly. Note that the `matrix.type` field of every node in the control tree must be identical.

- **variant.** The `variant` field specifies which algorithmic variant should execute. Valid values are `FLA_SUBPROBLEM` and `FLA_BLOCKED_VARIANT1` through `FLA_BLOCKED_VARIANT20`. Note that the semantic meaning of `FLA_SUBPROBLEM` differs depending on whether the tree is configured for flat or hierarchical matrices. For control trees of matrix type `FLA_FLAT`, `FLA_SUBPROBLEM` refers to a node that invokes the operation’s external implementation. For trees of matrix type `FLA_HIER`, `FLA_SUBPROBLEM` refers to a node that causes the back-end to either recurse further if the bottom of the hierarchy has not yet been reached, or execute (or enqueue) the subproblem otherwise. Native unblocked FLAME algorithm implementations are not yet supported.
- **blocksize.** The `blocksize` field specifies a structure which contains four blocksize values. This allows the user to specify different blocksizes depending on the numerical datatype being used. blocksize structures should be created with `FLA_Blocksize_create()`.

Control tree nodes also contain fields which are unique to the operation type. Specifically, a control tree type is made unique by the number and type of sub-tree fields it contains. Each control tree structure has at least one pointer to another control tree node. These child nodes contain the relevant information for executing the the operation’s algorithmic subproblems. For example, the `fla_syrk_t` type allows for two child nodes, one for a SYRK subproblem and one for a GEMM subproblem. Figure 6.1 lists the code defining the structure for the `fla_syrk_t` type along a sample of structures for other operations supported in `libflame`.

There are two circumstances under which you may leave a subproblem field uninitialized.

- **The algorithmic variant does not need the full set of subproblems.** Some algorithmic variants only use a subset of the subproblem fields made available in the control tree structure. For example, variants 5 and 6 of the SYRK algorithm only invoke smaller SYRK subproblems, and thus do not need to perform any GEMM subproblems. In this case, the `sub_gemm` field is not referenced by the runtime system and thus it may safely be initialized to `NULL`.
- **The control tree node’s variant field is `FLA_SUBPROBLEM`.** The nodes at the “leaves” of the tree should contain a `variant` field with a special value: `FLA_SUBPROBLEM`. Every flat matrix control tree contains at least one leaf node that indicates that blocked algorithm recursion should stop, and an external implementation should be invoked for that node. Likewise, every hierarchical matrix control tree contains at least one “recurse” node where further recursion in the algorithm-by-blocks is performed

if the matrix hierarchy requires it. In both cases, *none* of the subproblem fields are referenced, and thus they may all be safely initialized to `NULL`.

When a control tree contains more than one subproblem field for a given operation type, the order of the subproblems matters. Consider the SYMM operation and its corresponding control tree structure. All ten variants of SYMM contain one SYMM subproblem, and blocked variants 1 through 8 contain two GEMM subproblems. So control tree nodes for blocked variants 1 through 8 must also initialize both GEMM fields. But which GEMM field corresponds to which GEMM subproblem instance in the SYMM algorithm? In situations like this, the mapping is simple: the `sub_gemm1` field corresponds to the GEMM subproblem that occurs first in SYMM algorithm, while the `sub_gemm2` field corresponds to the GEMM subproblem that occurs second. This rule for disambiguating subproblems of identical operation types must be observed for all cases where the algorithm contains more than one subproblem of a particular operation.

6.5.4 Control tree API

First, we present the interface for creating and manipulating blocksize structures, which are a necessary component of control tree nodes.

6.5.4.1 Blocksize structures

```
fla_blocksize_t* FLA_Blocksize_create( dim_t b_s,
                                       dim_t b_d,
                                       dim_t b_c,
                                       dim_t b_z );
```

Purpose: Create a structure containing a set of four blocksize, one for each of the numerical datatypes supported by `libflame`, and initialize the structure fields according to the function arguments.

Int. Notes: Though the interface allows the programmer to set blocksize values for all four datatypes, it is permissible to assign meaningful values to only the fields that correspond to the datatypes that will be used by the application.

Notes: Blocksize structures are allocated on the heap and should be released with `FLA_Blocksize_free()` when they are no longer needed.

Constraints:

- None of the blocksize arguments may be zero.

Returns: A pointer to a heap-allocated `fla_blocksize_t` structure.

Arguments:

- | | | |
|------------------|---|---|
| <code>b_s</code> | – | The blocksize to use for single precision real data. |
| <code>b_d</code> | – | The blocksize to use for double precision real data. |
| <code>b_c</code> | – | The blocksize to use for single precision complex data. |
| <code>b_z</code> | – | The blocksize to use for double precision complex data. |

```
void FLA_Blocksize_set( fla_blocksize_t* bp,
                        dim_t           b_s,
                        dim_t           b_d,
                        dim_t           b_c,
                        dim_t           b_z );
```

Purpose: Set the individual fields in an existing blocksize structure.

Int. Notes: Providing a value of zero for one of the blocksize arguments causes the function to leave the existing value unchanged for that particular blocksize field.

Constraints:

- bp may not be NULL.

Arguments:

- | | | |
|-----|---|---|
| bp | – | A pointer to an existing blocksize structure. |
| b_s | – | The blocksize to use for single precision real data. |
| b_d | – | The blocksize to use for double precision real data. |
| b_c | – | The blocksize to use for single precision complex data. |
| b_z | – | The blocksize to use for double precision complex data. |

```
void FLA_Blocksize_scale( fla_blocksize_t* bp,
                          double           factor );
```

Purpose: Scale the individual fields of an existing blocksize structure.

Imp. Notes: The scaling occurs as follows: the blocksize fields are typecast to `double`, then multiplied by the scaling `factor`, and finally typecast back to `int` before being stored to the blocksize structure.

Constraints:

- bp may not be NULL.

Arguments:

- | | | |
|--------|---|--|
| bp | – | A pointer to an existing blocksize structure. |
| factor | – | The scaling factor to apply to the blocksize values. |

```
fla_blocksize_t* FLA_Blocksize_create_copy( fla_blocksize_t* bp );
```

Purpose: Create a copy of an existing blocksize structure.

Constraints:

- bp may not be NULL.

Returns: A pointer to a heap-allocated `fla_blocksize_t` structure.

Arguments:

- | | | |
|----|---|---|
| bp | – | A pointer to an existing blocksize structure. |
|----|---|---|

```
void FLA_Blocksize_free( fla_blocksize_t* bp );
```

Purpose: Release the memory allocated to a blocksize structure.

Notes: FLA_Blocksize_free() should only be used with pointers to blocksize structures that were allocated with FLA_Blocksize_create() or FLA_Blocksize_create_copy().

Constraints:

- bp may not be NULL.

Arguments:

bp – A pointer to an existing blocksize structure.

```
dim_t FLA_Blocksize_extract( FLA_Datatype    datatype,
                             fla_blocksize_t* bp );
```

Purpose: Extract the value associated with a specific numerical datatype from a blocksize structure.

Constraints:

- The value of datatype must refer to a floating-point datatype.
- bp may not be NULL.

Returns: An unsigned integer value of type dim_t.

Arguments:

datatype – A constant corresponding to the numerical datatype requested.
bp – A pointer to an existing blocksize structure.

```
fla_blocksize_t* FLA_Query_blocksizes( FLA_Dimension dim_tag );
```

Purpose: Query the library for a reasonable set of blocksizes. The user must specify how the block-sizes are chosen by specifying a dimension *tag*. Valid tag values are FLA_DIMENSION_M, FLA_DIMENSION_K, FLA_DIMENSION_N, and FLA_DIMENSION_MIN. If libflame was configured with `--enable-goto-interfaces`, the first three values correspond to architecture-specific blocksizes associated with the *m*, *k*, and *n* dimensions of the inner-most matrix-matrix multiplication kernel in GotoBLAS. Otherwise, these three constants are associated with default values that may not be optimal. The last tag, FLA_DIMENSION_MIN, will cause FLA_Query_blocksizes() to return the smallest of the *m*, *k*, and *n* blocksizes. If unsure, use FLA_DIMENSION_MIN.

Notes: This function dynamically allocates memory for the structure in which the block-sizes are returned. It is the user's responsibility to deallocate this structure with FLA_Blocksize_free() when it is no longer needed.

Returns: A pointer to a heap-allocated fla_blocksize_t structure.

Arguments:

dim_tag – A constant specifying how to choose the blocksize.

```
dim_t FLA_Query_blocksize( FLA_Datatype  datatype,
                          FLA_Dimension dim_tag );
```

Purpose: Query the library for a reasonable blocksize for a specific datatype. The behavior of this function is similar to that of `FLA_Query_blocksizes()`, except that only a single `dim_t` scalar (for the datatype in question) is returned instead of a pointer to an entire `fla_blocksize_t` structure.

Notes: The values returned by this function are the same as those attainable by calling `FLA_Query_blocksizes()` and then using `FLA_Blocksize_extract()` to extract the blocksize for `datatype`.

Constraints:

- The value of `datatype` must refer to a floating-point datatype.

Returns: An unsigned integer value of type `dim_t`.

Arguments:

- | | |
|-----------------------|---|
| <code>datatype</code> | – A constant corresponding to the numerical datatype requested. |
| <code>dim_tag</code> | – A constant specifying how to choose the blocksize. |

```
dim_t FLA_Determine_blocksize( FLA_Obj      A_unproc,
                              FLA_Quadrant to_dir,
                              fla_blocksize_t* bp );
```

Purpose: Determine the blocksize given the contents of a blocksize structure and the current state of the matrix partitioning. If the blocksize is larger than the dimension of `A_unproc`, the dimension of `A_unproc` is returned instead. In this case, the dimension in question, be it the length or width, is determined by the value of `to_dir`. Specifically, if `to_dir` denotes vertical movement, the length of `A_unproc` is used, and for lateral movement, the width is used. If `to_dir` denotes diagonal movement, then the minimum dimension (as would be returned by `FLA_Obj_min_dim()`) is used.

Constraints:

- `bp` may not be NULL.

Returns: An unsigned integer value of type `dim_t`.

Arguments:

- | | |
|-----------------------|---|
| <code>A_unproc</code> | – An <code>FLA_Obj</code> view into the unprocessed portion of a matrix being tracked by a blocked algorithm. |
| <code>to_dir</code> | – The direction in which the algorithm is moving through the parent matrix of <code>A_unproc</code> . |
| <code>bp</code> | – A pointer to an existing blocksize structure. |

The remainder of this subsection describes the functions that create and initialize control tree nodes for each of the supported linear algebra operations. These functions share the same first three arguments, which correspond to the fields described in the previous subsection. Note that you should always invoke these interfaces for the leaves of the trees first, and then use the pointers returned from those routines in the initialization of higher internal nodes. Simply put, you cannot create a non-leaf node until you have created and initialized its children nodes.

6.5.4.2 Level-3 BLAS operations

```
fla_gemm_t* FLA_Cntl_gemm_obj_create( FLA_Matrix_type  matrix_type,
                                     int                variant,
                                     fla_blocksize_t*  blocksize,
                                     fla_gemm_t*       sub_gemm );
```

Purpose: Create a structure representing a node in a control tree for a general matrix-matrix multiplication (GEMM) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be one of `FLA_BLOCKED_VARIANT1` through `FLA_BLOCKED_VARIANT6`.

Returns: A pointer to a heap-allocated `fla_gemm_t` structure.

Arguments:

- `matrix_type` – The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used.
- `variant` – A constant value indicating the choice of variant for executing the computation associated with the control tree node being created.
- `blocksize` – A pointer to a blocksize structure to be used for the node being created.
- `sub_gemm` – A pointer to the node to be used for the GEMM subproblem.

```
fla_hemm_t* FLA_Cntl_hemm_obj_create( FLA_Matrix_type  matrix_type,
                                     int                variant,
                                     fla_blocksize_t*  blocksize,
                                     fla_hemm_t*       sub_hemm,
                                     fla_gemm_t*       sub_gemm1,
                                     fla_gemm_t*       sub_gemm2 );
```

Purpose: Create a structure representing a node in a control tree for a Hermitian matrix-matrix multiplication (HEMM) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`. Even if `variant` specifies a blocked variant, some algorithms contain fewer subproblems and thus do not use every subproblem field argument. In such cases, these arguments may be safely set to `NULL`. Please refer to the blocked algorithmic variant implementations to determine which subproblem fields are unused.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be one of `FLA_BLOCKED_VARIANT1` through `FLA_BLOCKED_VARIANT10`.

Returns: A pointer to a heap-allocated `fla_hemm_t` structure.

Arguments:

- `matrix_type` – The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used.
- `variant` – A constant value indicating the choice of variant for executing the computation associated with the control tree node being created.
- `blocksize` – A pointer to a blocksize structure to be used for the node being created.
- `sub_hemm` – A pointer to the node to be used for the HEMM subproblem.
- `sub_gemm1` – A pointer to the node to be used for the first GEMM subproblem.
- `sub_gemm2` – A pointer to the node to be used for the second GEMM subproblem.

```
fla_herk_t* FLA_Cntl_herk_obj_create( FLA_Matrix_type  matrix_type,
                                     int                variant,
                                     fla_blocksize_t*  blocksize,
                                     fla_herk_t*       sub_herk,
                                     fla_gemm_t*       sub_gemm );
```

Purpose: Create a structure representing a node in a control tree for a Hermitian rank-k update (HERK) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`. Even if `variant` specifies a blocked variant, some algorithms contain fewer subproblems and thus do not use every subproblem field argument. In such cases, these arguments may be safely set to `NULL`. Please refer to the blocked algorithmic variant implementations to determine which subproblem fields are unused.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be one of `FLA_BLOCKED_VARIANT1` through `FLA_BLOCKED_VARIANT6`.

Returns: A pointer to a heap-allocated `fla_herk_t` structure.

Arguments:

<code>matrix_type</code>	–	The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used.
<code>variant</code>	–	A constant value indicating the choice of variant for executing the computation associated with the control tree node being created.
<code>blocksize</code>	–	A pointer to a blocksize structure to be used for the node being created.
<code>sub_herk</code>	–	A pointer to the node to be used for the HERK subproblem.
<code>sub_gemm</code>	–	A pointer to the node to be used for the GEMM subproblem.

```
fla_her2k_t* FLA_Cntl_her2k_obj_create( FLA_Matrix_type  matrix_type,
                                       int                variant,
                                       fla_blocksize_t*   blocksize,
                                       fla_her2k_t*       sub_her2k,
                                       fla_gemm_t*        sub_gemm1,
                                       fla_gemm_t*        sub_gemm2 );
```

Purpose: Create a structure representing a node in a control tree for a Hermitian rank-2k update (HER2K) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`. Even if `variant` specifies a blocked variant, some algorithms contain fewer subproblems and thus do not use every subproblem field argument. In such cases, these arguments may be safely set to `NULL`. Please refer to the blocked algorithmic variant implementations to determine which subproblem fields are unused.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be one of `FLA_BLOCKED_VARIANT1` through `FLA_BLOCKED_VARIANT10`.

Returns: A pointer to a heap-allocated `fla_her2k_t` structure.

Arguments:

- `matrix_type` – The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used.
- `variant` – A constant value indicating the choice of variant for executing the computation associated with the control tree node being created.
- `blocksize` – A pointer to a blocksize structure to be used for the node being created.
- `sub_her2k` – A pointer to the node to be used for the HER2K subproblem.
- `sub_gemm1` – A pointer to the node to be used for the first GEMM subproblem.
- `sub_gemm2` – A pointer to the node to be used for the second GEMM subproblem.


```

fla_symm_t* FLA_Cntl_symm_obj_create( FLA_Matrix_type  matrix_type,
                                     int                variant,
                                     fla_blocksize_t*    blocksize,
                                     fla_symm_t*         sub_symm,
                                     fla_gemm_t*         sub_gemm1,
                                     fla_gemm_t*         sub_gemm2 );

```

Purpose: Create a structure representing a node in a control tree for a symmetric matrix-matrix multiplication (SYMM) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`. Even if `variant` specifies a blocked variant, some algorithms contain fewer subproblems and thus do not use every subproblem field argument. In such cases, these arguments may be safely set to `NULL`. Please refer to the blocked algorithmic variant implementations to determine which subproblem fields are unused.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be one of `FLA_BLOCKED_VARIANT1` through `FLA_BLOCKED_VARIANT10`.

Returns: A pointer to a heap-allocated `fla_symm_t` structure.

Arguments:

- | | | |
|--------------------------|---|--|
| <code>matrix_type</code> | – | The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used. |
| <code>variant</code> | – | A constant value indicating the choice of variant for executing the computation associated with the control tree node being created. |
| <code>blocksize</code> | – | A pointer to a blocksize structure to be used for the node being created. |
| <code>sub_symm</code> | – | A pointer to the node to be used for the SYMM subproblem. |
| <code>sub_gemm1</code> | – | A pointer to the node to be used for the first GEMM subproblem. |
| <code>sub_gemm2</code> | – | A pointer to the node to be used for the second GEMM subproblem. |

```
fla_syrk_t* FLA_Cntl_syrk_obj_create( FLA_Matrix_type  matrix_type,
                                     int                variant,
                                     fla_blocksize_t*  blocksize,
                                     fla_syrk_t*       sub_syrk,
                                     fla_gemm_t*       sub_gemm );
```

Purpose: Create a structure representing a node in a control tree for a symmetric rank-k update (SYRK) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`. Even if `variant` specifies a blocked variant, some algorithms contain fewer subproblems and thus do not use every subproblem field argument. In such cases, these arguments may be safely set to `NULL`. Please refer to the blocked algorithmic variant implementations to determine which subproblem fields are unused.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be one of `FLA_BLOCKED_VARIANT1` through `FLA_BLOCKED_VARIANT6`.

Returns: A pointer to a heap-allocated `fla_syrk_t` structure.

Arguments:

- `matrix_type` – The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used.
- `variant` – A constant value indicating the choice of variant for executing the computation associated with the control tree node being created.
- `blocksize` – A pointer to a blocksize structure to be used for the node being created.
- `sub_syrk` – A pointer to the node to be used for the SYRK subproblem.
- `sub_gemm` – A pointer to the node to be used for the GEMM subproblem.

```
fla_syr2k_t* FLA_Cntl_syr2k_obj_create( FLA_Matrix_type  matrix_type,
                                       int                variant,
                                       fla_blocksize_t*   blocksize,
                                       fla_syr2k_t*       sub_syr2k,
                                       fla_gemm_t*        sub_gemm1,
                                       fla_gemm_t*        sub_gemm2 );
```

Purpose: Create a structure representing a node in a control tree for a symmetric rank-2k update (SYR2K) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`. Even if `variant` specifies a blocked variant, some algorithms contain fewer subproblems and thus do not use every subproblem field argument. In such cases, these arguments may be safely set to `NULL`. Please refer to the blocked algorithmic variant implementations to determine which subproblem fields are unused.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be one of `FLA_BLOCKED_VARIANT1` through `FLA_BLOCKED_VARIANT10`.

Returns: A pointer to a heap-allocated `fla_syr2k_t` structure.

Arguments:

- `matrix_type` – The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used.
- `variant` – A constant value indicating the choice of variant for executing the computation associated with the control tree node being created.
- `blocksize` – A pointer to a blocksize structure to be used for the node being created.
- `sub_syr2k` – A pointer to the node to be used for the SYR2K subproblem.
- `sub_gemm1` – A pointer to the node to be used for the first GEMM subproblem.
- `sub_gemm2` – A pointer to the node to be used for the second GEMM subproblem.

```
fla_trmm_t* FLA_Cntl_trmm_obj_create( FLA_Matrix_type  matrix_type,
                                     int                variant,
                                     fla_blocksize_t*    blocksize,
                                     fla_trmm_t*         sub_trmm,
                                     fla_gemm_t*         sub_gemm );
```

Purpose: Create a structure representing a node in a control tree for a triangular matrix-matrix multiplication (TRMM) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`. Even if `variant` specifies a blocked variant, some algorithms contain fewer subproblems and thus do not use every subproblem field argument. In such cases, these arguments may be safely set to `NULL`. Please refer to the blocked algorithmic variant implementations to determine which subproblem fields are unused.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be one of `FLA_BLOCKED_VARIANT1` through `FLA_BLOCKED_VARIANT4`.

Returns: A pointer to a heap-allocated `fla_trmm_t` structure.

Arguments:

- `matrix_type` – The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used.
- `variant` – A constant value indicating the choice of variant for executing the computation associated with the control tree node being created.
- `blocksize` – A pointer to a blocksize structure to be used for the node being created.
- `sub_trmm` – A pointer to the node to be used for the TRMM subproblem.
- `sub_gemm` – A pointer to the node to be used for the GEMM subproblem.

```

fla_trsm_t* FLA_Cntl_trsm_obj_create( FLA_Matrix_type  matrix_type,
                                     int                variant,
                                     fla_blocksize_t*   blocksize,
                                     fla_trsm_t*        sub_trsm,
                                     fla_gemm_t*        sub_gemm );

```

Purpose: Create a structure representing a node in a control tree for a triangular solve with multiple right-hand sides (TRSM) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`. Even if `variant` specifies a blocked variant, some algorithms contain fewer subproblems and thus do not use every subproblem field argument. In such cases, these arguments may be safely set to `NULL`. Please refer to the blocked algorithmic variant implementations to determine which subproblem fields are unused.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be one of `FLA_BLOCKED_VARIANT1` through `FLA_BLOCKED_VARIANT4`.

Returns: A pointer to a heap-allocated `fla_trsm_t` structure.

Arguments:

- | | | |
|--------------------------|---|--|
| <code>matrix_type</code> | – | The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used. |
| <code>variant</code> | – | A constant value indicating the choice of variant for executing the computation associated with the control tree node being created. |
| <code>blocksize</code> | – | A pointer to a blocksize structure to be used for the node being created. |
| <code>sub_trsm</code> | – | A pointer to the node to be used for the TRSM subproblem. |
| <code>sub_gemm</code> | – | A pointer to the node to be used for the GEMM subproblem. |

6.5.4.3 LAPACK-level operations

```

fla_chol_t* FLA_Cntl_chol_obj_create( FLA_Matrix_type  matrix_type,
                                     int                variant,
                                     fla_blocksize_t*   blocksize,
                                     fla_chol_t*        sub_chol,
                                     fla_syrk_t*        sub_syrk,
                                     fla_herk_t*        sub_herk,
                                     fla_trsm_t*        sub_trsm,
                                     fla_gemm_t*        sub_gemm );

```

Purpose: Create a structure representing a node in a control tree for a Cholesky factorization (CHOL) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`. Even if `variant` specifies a blocked variant, some algorithms contain fewer subproblems and thus do not use every subproblem field argument. In such cases, these arguments may be safely set to `NULL`. Please refer to the blocked algorithmic variant implementations to determine which subproblem fields are unused.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be one of `FLA_BLOCKED_VARIANT1` through `FLA_BLOCKED_VARIANT3`.

Returns: A pointer to a heap-allocated `fla_chol_t` structure.

Arguments:

<code>matrix_type</code>	–	The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used.
<code>variant</code>	–	A constant value indicating the choice of variant for executing the computation associated with the control tree node being created.
<code>blocksize</code>	–	A pointer to a blocksize structure to be used for the node being created.
<code>sub_chol</code>	–	A pointer to the node to be used for the CHOL subproblem.
<code>sub_syrk</code>	–	A pointer to the node to be used for the SRYK subproblem.
<code>sub_herk</code>	–	A pointer to the node to be used for the HERK subproblem.
<code>sub_trsm</code>	–	A pointer to the node to be used for the TRSM subproblem.
<code>sub_gemm</code>	–	A pointer to the node to be used for the GEMM subproblem.

```

fla_lu_t* FLA_Cntl_lu_obj_create( FLA_Matrix_type  matrix_type,
                                int                variant,
                                fla_blocksize_t*   blocksize,
                                fla_lu_t*         sub_lu,
                                fla_gemm_t*       sub_gemm1,
                                fla_gemm_t*       sub_gemm2,
                                fla_gemm_t*       sub_gemm3,
                                fla_trsm_t*       sub_trsm1,
                                fla_trsm_t*       sub_trsm2 );

```

Purpose: Create a structure representing a node in a control tree for an LU factorization (LU) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`. Even if `variant` specifies a blocked variant, some algorithms contain fewer subproblems and thus do not use every subproblem field argument. In such cases, these arguments may be safely set to `NULL`. Please refer to the blocked algorithmic variant implementations to determine which subproblem fields are unused.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be one of `FLA_BLOCKED_VARIANT1` through `FLA_BLOCKED_VARIANT3`.

Returns: A pointer to a heap-allocated `fla_lu_t` structure.

Arguments:

<code>matrix_type</code>	–	The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used.
<code>variant</code>	–	A constant value indicating the choice of variant for executing the computation associated with the control tree node being created.
<code>blocksize</code>	–	A pointer to a blocksize structure to be used for the node being created.
<code>sub_chol</code>	–	A pointer to the node to be used for the CHOL subproblem.
<code>sub_syrk</code>	–	A pointer to the node to be used for the SRYK subproblem.
<code>sub_herk</code>	–	A pointer to the node to be used for the HERK subproblem.
<code>sub_trsm</code>	–	A pointer to the node to be used for the TRSM subproblem.
<code>sub_gemm</code>	–	A pointer to the node to be used for the GEMM subproblem.

```
fla_qrut_t* FLA_Cntl_qrut_obj_create( FLA_Matrix_type  matrix_type,
                                     int                variant,
                                     fla_blocksize_t*   blocksize,
                                     fla_qrut_t*        sub_qrut,
                                     fla_trmm_t*        sub_trmm1,
                                     fla_trmm_t*        sub_trmm2,
                                     fla_gemm_t*        sub_gemm1,
                                     fla_gemm_t*        sub_gemm2,
                                     fla_trsm_t*        sub_trsm,
                                     fla_axpy_t*        sub_axpy,
                                     fla_copy_t*        sub_copy );
```

Purpose: Create a structure representing a node in a control tree for a QR factorization via the UT transform (QRUT) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`. Even if `variant` specifies a blocked variant, some algorithms contain fewer subproblems and thus do not use every subproblem field argument. In such cases, these arguments may be safely set to `NULL`. Please refer to the blocked algorithmic variant implementations to determine which subproblem fields are unused.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be `FLA_BLOCKED_VARIANT1`.

Returns: A pointer to a heap-allocated `fla_qrut_t` structure.

Arguments:

- | | | |
|--------------------------|---|--|
| <code>matrix_type</code> | – | The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used. |
| <code>variant</code> | – | A constant value indicating the choice of variant for executing the computation associated with the control tree node being created. |
| <code>blocksize</code> | – | A pointer to a blocksize structure to be used for the node being created. |
| <code>sub_qrut</code> | – | A pointer to the node to be used for the QRUT subproblem. |
| <code>sub_trmm1</code> | – | A pointer to the node to be used for the first TRMM subproblem. |
| <code>sub_trmm2</code> | – | A pointer to the node to be used for the second TRMM subproblem. |
| <code>sub_gemm1</code> | – | A pointer to the node to be used for the first GEMM subproblem. |
| <code>sub_gemm2</code> | – | A pointer to the node to be used for the second GEMM subproblem. |
| <code>sub_trsm</code> | – | A pointer to the node to be used for the TRSM subproblem. |
| <code>sub_axpy</code> | – | A pointer to the node to be used for the AXPY subproblem. |
| <code>sub_copy</code> | – | A pointer to the node to be used for the COPY subproblem. |


```

fla_lqut_t* FLA_Cntl_lqut_obj_create( FLA_Matrix_type  matrix_type,
                                     int                variant,
                                     fla_blocksize_t*    blocksize,
                                     fla_lqut_t*         sub_lqut,
                                     fla_trmm_t*         sub_trmm1,
                                     fla_trmm_t*         sub_trmm2,
                                     fla_gemm_t*         sub_gemm1,
                                     fla_gemm_t*         sub_gemm2,
                                     fla_trsm_t*         sub_trsm,
                                     fla_axpy_t*         sub_axpy,
                                     fla_copy_t*         sub_copy );

```

Purpose: Create a structure representing a node in a control tree for a LQ factorization via the UT transform (LQUT) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`. Even if `variant` specifies a blocked variant, some algorithms contain fewer subproblems and thus do not use every subproblem field argument. In such cases, these arguments may be safely set to `NULL`. Please refer to the blocked algorithmic variant implementations to determine which subproblem fields are unused.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be `FLA_BLOCKED_VARIANT1`.

Returns: A pointer to a heap-allocated `fla_lqut_t` structure.

Arguments:

- | | | |
|--------------------------|---|--|
| <code>matrix_type</code> | – | The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used. |
| <code>variant</code> | – | A constant value indicating the choice of variant for executing the computation associated with the control tree node being created. |
| <code>blocksize</code> | – | A pointer to a blocksize structure to be used for the node being created. |
| <code>sub_lqut</code> | – | A pointer to the node to be used for the LQUT subproblem. |
| <code>sub_trmm1</code> | – | A pointer to the node to be used for the first TRMM subproblem. |
| <code>sub_trmm2</code> | – | A pointer to the node to be used for the second TRMM subproblem. |
| <code>sub_gemm1</code> | – | A pointer to the node to be used for the first GEMM subproblem. |
| <code>sub_gemm2</code> | – | A pointer to the node to be used for the second GEMM subproblem. |
| <code>sub_trsm</code> | – | A pointer to the node to be used for the TRSM subproblem. |
| <code>sub_axpy</code> | – | A pointer to the node to be used for the AXPY subproblem. |
| <code>sub_copy</code> | – | A pointer to the node to be used for the COPY subproblem. |

```

fla_trinv_t* FLA_Cntl_trinv_obj_create( FLA_Matrix_type  matrix_type,
                                       int               variant,
                                       fla_blocksize_t*  blocksize,
                                       fla_trinv_t*      sub_trinv,
                                       fla_trmm_t*       sub_trmm,
                                       fla_trsm_t*       sub_trsm1,
                                       fla_trsm_t*       sub_trsm2,
                                       fla_gemm_t*       sub_gemm );

```

Purpose: Create a structure representing a node in a control tree for a triangular matrix inversion (TRINV) operation and initialize its fields according to the function arguments.

Notes: If **variant** is **FLA_SUBPROBLEM**, none of the pointer arguments are used and thus they may be safely set to **NULL**. Even if **variant** specifies a blocked variant, some algorithms contain fewer subproblems and thus do not use every subproblem field argument. In such cases, these arguments may be safely set to **NULL**. Please refer to the blocked algorithmic variant implementations to determine which subproblem fields are unused.

Constraints:

- If **variant** is not **FLA_SUBPROBLEM**, then it must be one of **FLA_BLOCKED_VARIANT1** through **FLA_BLOCKED_VARIANT4**.

Returns: A pointer to a heap-allocated **fla_trinv_t** structure.

Arguments:

matrix_type	–	The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used.
variant	–	A constant value indicating the choice of variant for executing the computation associated with the control tree node being created.
blocksize	–	A pointer to a blocksize structure to be used for the node being created.
sub_trinv	–	A pointer to the node to be used for the TRINV subproblem.
sub_trmm	–	A pointer to the node to be used for the TRMM subproblem.
sub_trsm1	–	A pointer to the node to be used for the first TRSM subproblem.
sub_trsm2	–	A pointer to the node to be used for the second TRSM subproblem.
sub_gemm	–	A pointer to the node to be used for the GEMM subproblem.

```

fla_spdinv_t* FLA_Cntl_spdinv_obj_create( FLA_Matrix_type  matrix_type,
                                          int               variant,
                                          fla_blocksize_t*  blocksize,
                                          fla_chol_t*       sub_chol,
                                          fla_trinv_t*      sub_trinv,
                                          fla_ttmm_t*       sub_ttmm );

```

Purpose: Create a structure representing a node in a control tree for a symmetric (or Hermitian) positive definite matrix inversion (SPDINV) operation and initialize its fields according to the function arguments.

Notes: Since SPDINV is implemented as the sequence of its three constituent suboperations, CHOL, TRINV, and TTMM without any matrix partitioning at the SPDINV level, the **variant** field is not used. Also, the SPDINV front-end interprets the **blocksize** field as the cutoff at which to switch from external routines to internal **libflame** variants.

Returns: A pointer to a heap-allocated **fla_spdinv_t** structure.

Arguments:

- matrix_type** – The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used.
- variant** – Not referenced.
- blocksize** – A pointer to a blocksize structure to be used for the node being created. Note that the front-end interprets these values as the cutoffs at which to switch from external implementations to internal **libflame** variants.
- sub_chol** – A pointer to the node to be used for the CHOL suboperation.
- sub_trinv** – A pointer to the node to be used for the TRINV suboperation.
- sub_ttmm** – A pointer to the node to be used for the TTMM suboperation.

```

fla_hess_t* FLA_Cntl_hess_obj_create( FLA_Matrix_type  matrix_type,
                                     int                variant,
                                     fla_blocksize_t*   blocksize,
                                     fla_hess_t*        sub_hess,
                                     fla_trmm_t*        sub_trmm1,
                                     fla_trmm_t*        sub_trmm2,
                                     fla_trmm_t*        sub_trmm3,
                                     fla_trmm_t*        sub_trmm4,
                                     fla_gemm_t*        sub_gemm1,
                                     fla_gemm_t*        sub_gemm2,
                                     fla_gemm_t*        sub_gemm3 );

```

Purpose: Create a structure representing a node in a control tree for a reduction to upper Hessenberg form (HESS) operation and initialize its fields according to the function arguments.

Notes: If **variant** is **FLA_SUBPROBLEM**, none of the pointer arguments are used and thus they may be safely set to **NULL**. Even if **variant** specifies a blocked variant, some algorithms contain fewer subproblems and thus do not use every subproblem field argument. In such cases, these arguments may be safely set to **NULL**. Please refer to the blocked algorithmic variant implementations to determine which subproblem fields are unused.

Dev. notes: The algorithmic variant implementations for reduction to upper Hessenberg form do not exist. If this operation is needed, use the external wrapper routine **FLA_Hess_blk_external()**.

Constraints:

- If **variant** must be **FLA_SUBPROBLEM** since no blocked variants for the operation exist yet.

Returns: A pointer to a heap-allocated **fla_hess_t** structure.

Arguments:

- | | | |
|--------------------|---|--|
| matrix_type | – | The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used. |
| variant | – | A constant value indicating the choice of variant for executing the computation associated with the control tree node being created. |
| blocksize | – | A pointer to a blocksize structure to be used for the node being created. |
| sub_hess | – | A pointer to the node to be used for the HESS subproblem. |
| sub_trmm1 | – | A pointer to the node to be used for the first TRMM subproblem. |
| sub_trmm2 | – | A pointer to the node to be used for the second TRMM subproblem. |
| sub_trmm3 | – | A pointer to the node to be used for the third TRMM subproblem. |
| sub_trmm4 | – | A pointer to the node to be used for the fourth TRMM subproblem. |
| sub_gemm1 | – | A pointer to the node to be used for the first GEMM subproblem. |
| sub_gemm2 | – | A pointer to the node to be used for the second GEMM subproblem. |
| sub_gemm3 | – | A pointer to the node to be used for the third GEMM subproblem. |

```
fla_ttmm_t* FLA_Cntl_ttmm_obj_create( FLA_Matrix_type  matrix_type,
                                     int                variant,
                                     fla_blocksize_t*   blocksize,
                                     fla_ttmm_t*        sub_ttmm,
                                     fla_syrk_t*        sub_syrk,
                                     fla_herk_t*        sub_herk,
                                     fla_trmm_t*        sub_trmm,
                                     fla_gemm_t*        sub_gemm );
```

Purpose: Create a structure representing a node in a control tree for a triangular-transpose matrix multiply (TTMM) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`. Even if `variant` specifies a blocked variant, some algorithms contain fewer subproblems and thus do not use every subproblem field argument. In such cases, these arguments may be safely set to `NULL`. Please refer to the blocked algorithmic variant implementations to determine which subproblem fields are unused.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be one of `FLA_BLOCKED_VARIANT1` through `FLA_BLOCKED_VARIANT3`.

Returns: A pointer to a heap-allocated `fla_ttmm_t` structure.

Arguments:

- | | | |
|--------------------------|---|--|
| <code>matrix_type</code> | – | The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used. |
| <code>variant</code> | – | A constant value indicating the choice of variant for executing the computation associated with the control tree node being created. |
| <code>blocksize</code> | – | A pointer to a blocksize structure to be used for the node being created. |
| <code>sub_ttmm</code> | – | A pointer to the node to be used for the TTMM subproblem. |
| <code>sub_syrk</code> | – | A pointer to the node to be used for the SRYK subproblem. |
| <code>sub_herk</code> | – | A pointer to the node to be used for the HERK subproblem. |
| <code>sub_trmm</code> | – | A pointer to the node to be used for the TRMM subproblem. |
| <code>sub_gemm</code> | – | A pointer to the node to be used for the GEMM subproblem. |

```

fla_sylv_t* FLA_Cntl_sylv_obj_create( FLA_Matrix_type  matrix_type,
                                     int                variant,
                                     fla_blocksize_t*   blocksize,
                                     fla_sylv_t*        sub_sylv1,
                                     fla_sylv_t*        sub_sylv2,
                                     fla_sylv_t*        sub_sylv3,
                                     fla_gemm_t*        sub_gemm1,
                                     fla_gemm_t*        sub_gemm2,
                                     fla_gemm_t*        sub_gemm3,
                                     fla_gemm_t*        sub_gemm4,
                                     fla_gemm_t*        sub_gemm5,
                                     fla_gemm_t*        sub_gemm6,
                                     fla_gemm_t*        sub_gemm7,
                                     fla_gemm_t*        sub_gemm8 );

```

Purpose: Create a structure representing a node in a control tree for a triangular Sylvester equation solve (SYLV) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`. Even if `variant` specifies a blocked variant, some algorithms contain fewer subproblems and thus do not use every subproblem field argument. In such cases, these arguments may be safely set to `NULL`. Please refer to the blocked algorithmic variant implementations to determine which subproblem fields are unused.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be one of `FLA_BLOCKED_VARIANT1` through `FLA_BLOCKED_VARIANT18`.

Returns: A pointer to a heap-allocated `fla_sylv_t` structure.

Arguments:

- | | | |
|--------------------------|---|--|
| <code>matrix_type</code> | – | The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used. |
| <code>variant</code> | – | A constant value indicating the choice of variant for executing the computation associated with the control tree node being created. |
| <code>blocksize</code> | – | A pointer to a blocksize structure to be used for the node being created. |
| <code>sub_sylv1</code> | – | A pointer to the node to be used for the first SYLV subproblem. |
| <code>sub_sylv2</code> | – | A pointer to the node to be used for the second SYLV subproblem. |
| <code>sub_sylv3</code> | – | A pointer to the node to be used for the third SYLV subproblem. |
| <code>sub_gemm1</code> | – | A pointer to the node to be used for the first GEMM subproblem. |
| <code>sub_gemm2</code> | – | A pointer to the node to be used for the second GEMM subproblem. |
| <code>sub_gemm3</code> | – | A pointer to the node to be used for the third GEMM subproblem. |
| <code>sub_gemm4</code> | – | A pointer to the node to be used for the fourth GEMM subproblem. |
| <code>sub_gemm5</code> | – | A pointer to the node to be used for the fifth GEMM subproblem. |
| <code>sub_gemm6</code> | – | A pointer to the node to be used for the sixth GEMM subproblem. |
| <code>sub_gemm7</code> | – | A pointer to the node to be used for the seventh GEMM subproblem. |
| <code>sub_gemm8</code> | – | A pointer to the node to be used for the eighth GEMM subproblem. |

6.5.4.4 Miscellaneous operations

```
fla_swap_t* FLA_Cntl_swap_obj_create( FLA_Matrix_type  matrix_type,
                                     int                variant,
                                     fla_blocksize_t*   blocksize,
                                     fla_swap_t*        sub_swap );
```

Purpose: Create a structure representing a node in a control tree for a matrix swap (SWAP) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be either `FLA_BLOCKED_VARIANT1` or `FLA_BLOCKED_VARIANT2`.

Returns: A pointer to a heap-allocated `fla_swap_t` structure.

Arguments:

- `matrix_type` – The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used.
- `variant` – A constant value indicating the choice of variant for executing the computation associated with the control tree node being created.
- `blocksize` – A pointer to a blocksize structure to be used for the node being created.
- `sub_swap` – A pointer to the node to be used for the SWAP subproblem.

```
fla_transpose_t* FLA_Cntl_transpose_obj_create( FLA_Matrix_type  matrix_type,
                                                int                variant,
                                                fla_blocksize_t*   blocksize,
                                                fla_trans_t*       sub_trans,
                                                fla_swap_t*        sub_swap );
```

Purpose: Create a structure representing a node in a control tree for a matrix transposition (TRANSPOSE) operation and initialize its fields according to the function arguments.

Notes: If `variant` is `FLA_SUBPROBLEM`, none of the pointer arguments are used and thus they may be safely set to `NULL`.

Constraints:

- If `variant` is not `FLA_SUBPROBLEM`, then it must be either `FLA_BLOCKED_VARIANT1` or `FLA_BLOCKED_VARIANT2`.

Returns: A pointer to a heap-allocated `fla_tpose_t` structure.

Arguments:

- `matrix_type` – The type of matrix (flat or hierarchical) to support in the control tree in which the node will be used.
- `variant` – A constant value indicating the choice of variant for executing the computation associated with the control tree node being created.
- `blocksize` – A pointer to a blocksize structure to be used for the node being created.
- `sub_trans` – A pointer to the node to be used for the TRANSPOSE subproblem.
- `sub_swap` – A pointer to the node to be used for the SWAP subproblem.

```
void FLA_Cntl_obj_free( void* cntl );
```

Purpose: Release the memory allocated for a structure representing a node in a control tree.

Notes: `FLA_Cntl_obj_free()` should only be used with pointers to control tree structures that were allocated with the `FLA_Cntl.*_create()` routines.

Arguments:

`cntl` – A pointer to the node to be freed.

6.5.5 Default control trees

The default control trees are created when `libflame` is initialized via `FLA_Init()`. The subroutines in which the actual creation and initialization takes place are named according to the operation name and execution type. For example, control trees for a Cholesky factorization that is to be executed sequentially with conventional storage are initialized in the subroutine `FLA_Chol_cntl_init()`. Likewise, control trees for a triangular matrix inversion that is to be executed sequentially with hierarchical storage are initialized in `FLASH_Trinv_cntl_init()`. Figure 6.2 shows examples of these routines for the Cholesky factorization with hierarchical storage, which gives the reader an idea of how control trees should be initialized.

6.5.6 Operation front-ends

Once the library has been initialized, the default set of control trees are ready to use. Figure 6.3 shows examples of some front-end routines found in `libflame`. This illustrates how control trees are used at the highest level.

6.5.7 Internal back-ends

The `libflame` front-ends and algorithmic variant implementations both directly invoke internal back-end functions. It is here that the control tree is decoded and used to determine how execution will proceed with respect to variant and execution type. Figure 6.4 shows the internal routines for Cholesky factorization.

Interfaces for the supported operation back-ends follow.

6.5.7.1 Level-3 BLAS operations

```
void FLA_Gemm_internal( FLA_Trans transa, FLA_Trans transb, FLA_Obj alpha,
                       FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C,
                       fla_gemm_t* cntl );
```

Purpose: Perform a GEMM operation on A , B , and C according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_Gemm()`. Please see the description for `FLA_Gemm()` for further details.

GEMM with heirarchical storage	SYRK with heirarchical storage
<pre> #include "FLAME.h" fla_gemm_t* flash_gemm_cntl_blas; fla_gemm_t* flash_gemm_cntl_mm_op; fla_gemm_t* flash_gemm_cntl_mp_bp; fla_gemm_t* flash_gemm_cntl_pm_bp; fla_gemm_t* flash_gemm_cntl_op_bp; fla_gemm_t* flash_gemm_cntl_pb_bb; fla_gemm_t* flash_gemm_cntl_bp_bb; fla_blocksize_t* flash_gemm_bsize; void FLASH_Gemm_cntl_init() { // Set gemm blocksize based on hierarchical storage. flash_gemm_bsize = FLA_Blocksize_create(1, 1, 1, 1); // Create a control tree node that executes a gemm subproblem. flash_gemm_cntl_blas = FLA_Cntl_gemm_obj_create(FLA_EXECUTE_DREF, FLA_UNBLOCKED_EXTERN, NULL, NULL); // Create control trees for situations where one dimension is large. flash_gemm_cntl_pb_bb = FLA_Cntl_gemm_obj_create(FLA_EXECUTE_ONLY, FLA_BLOCKED_VARIANT1, flash_gemm_bsize, flash_gemm_cntl_blas); flash_gemm_cntl_bp_bb = FLA_Cntl_gemm_obj_create(FLA_EXECUTE_ONLY, FLA_BLOCKED_VARIANT3, flash_gemm_bsize, flash_gemm_cntl_blas); // Create control trees for situations where two dimensions are large. flash_gemm_cntl_op_bp = FLA_Cntl_gemm_obj_create(FLA_EXECUTE_ONLY, FLA_BLOCKED_VARIANT1, flash_gemm_bsize, flash_gemm_cntl_bp_bb); flash_gemm_cntl_mp_bp = FLA_Cntl_gemm_obj_create(FLA_EXECUTE_ONLY, FLA_BLOCKED_VARIANT5, flash_gemm_bsize, flash_gemm_cntl_pb_bb); flash_gemm_cntl_pm_bp = FLA_Cntl_gemm_obj_create(FLA_EXECUTE_ONLY, FLA_BLOCKED_VARIANT5, flash_gemm_bsize, flash_gemm_cntl_bp_bb); // Create control trees for situations where all dimensions are large. flash_gemm_cntl_mm_op = FLA_Cntl_gemm_obj_create(FLA_EXECUTE_ONLY, FLA_BLOCKED_VARIANT5, flash_gemm_bsize, flash_gemm_cntl_op_bp); } </pre>	<pre> #include "FLAME.h" extern fla_gemm_t* flash_gemm_cntl_pb_bb; fla_syrk_t* flash_syrk_cntl_blas; fla_syrk_t* flash_syrk_cntl_ip; fla_syrk_t* flash_syrk_cntl_op; fla_syrk_t* flash_syrk_cntl_sq; fla_blocksize_t* flash_syrk_bsize; void FLASH_Syrk_cntl_init() { // Set syrk blocksize based on hierarchical storage. flash_syrk_bsize = FLA_Blocksize_create(1, 1, 1, 1); // Create a control tree that assumes A is a b x b block. flash_syrk_cntl_blas = FLA_Cntl_syrk_obj_create(FLA_EXECUTE_DREF, FLA_UNBLOCKED_EXTERN, NULL, NULL); // Create a control tree that assumes A * A' forms an inner panel product. flash_syrk_cntl_ip = FLA_Cntl_syrk_obj_create(FLA_EXECUTE_ONLY, FLA_BLOCKED_VARIANT5, flash_syrk_bsize, flash_syrk_cntl_blas); // Create a control tree that assumes A * A' forms an outer panel product. flash_syrk_cntl_op = FLA_Cntl_syrk_obj_create(FLA_EXECUTE_ONLY, FLA_BLOCKED_VARIANT2, flash_syrk_bsize, flash_syrk_cntl_blas); // Create a control tree that assumes A is large. flash_syrk_cntl_sq = FLA_Cntl_syrk_obj_create(FLA_EXECUTE_ONLY, FLA_BLOCKED_VARIANT5, flash_syrk_bsize, flash_syrk_cntl_op); } </pre>
TRSM with hierarchical storage	CHOL with hierarchical storage
<pre> #include "FLAME.h" extern fla_gemm_t* flash_gemm_cntl_op_bp; fla_trsm_t* flash_trsm_cntl_blas; fla_trsm_t* flash_trsm_cntl_bp; fla_trsm_t* flash_trsm_cntl_mp; fla_trsm_t* flash_trsm_cntl_mm; fla_blocksize_t* flash_trsm_bsize; void FLASH_Trsm_cntl_init() { // Set trsm blocksize based on hierarchical storage. flash_trsm_bsize = FLA_Blocksize_create(1, 1, 1, 1); // Create a control tree that assumes A and B are b x b blocks. flash_trsm_cntl_blas = FLA_Cntl_trsm_obj_create(FLA_EXECUTE_DREF, FLA_UNBLOCKED_EXTERN, NULL, NULL); // Create a control tree that assumes A is a block and B is a panel. flash_trsm_cntl_bp = FLA_Cntl_trsm_obj_create(FLA_EXECUTE_ONLY, FLA_BLOCKED_VARIANT3, flash_trsm_bsize, flash_trsm_cntl_blas); // Create a control tree that assumes A is large and B is a panel. flash_trsm_cntl_mp = FLA_Cntl_trsm_obj_create(FLA_EXECUTE_ONLY, FLA_BLOCKED_VARIANT2, flash_trsm_bsize, flash_trsm_cntl_blas); // Create a control tree that assumes A and B are both large. flash_trsm_cntl_mm = FLA_Cntl_trsm_obj_create(FLA_EXECUTE_ONLY, FLA_BLOCKED_VARIANT3, flash_trsm_bsize, flash_trsm_cntl_mp); } </pre>	<pre> #include "FLAME.h" extern fla_herk_t* flash_herk_cntl_op; extern fla_syrk_t* flash_syrk_cntl_op; extern fla_trsm_t* flash_trsm_cntl_bp; fla_chol_t* flash_chol_cntl_lapack; fla_chol_t* flash_chol_cntl; fla_blocksize_t* flash_chol_bsize; void FLASH_Chol_cntl_init() { // Set blocksize based on hierarchical storage. flash_chol_bsize = FLA_Blocksize_create(1, 1, 1, 1); // Create a control tree that assumes A is a b x b block. flash_chol_cntl_lapack = FLA_Cntl_chol_obj_create(FLA_EXECUTE_DREF, FLA_UNBLOCKED_EXTERN, NULL, NULL); // Create a control tree that assumes A is large. flash_chol_cntl = FLA_Cntl_chol_obj_create(FLA_EXECUTE_ONLY, FLA_BLOCKED_VARIANT3, flash_chol_bsize, flash_chol_cntl_lapack); } </pre>

Figure 6.2: Control tree initialization subroutines for various operations.

CHOL front-end for conventional storage	CHOL front-end for hierarchical storage
<pre> #include "FLAME.h" extern fla_chol_t* fla_chol_cntl; extern fla_chol_t* fla_chol_cntl2; FLA_Error FLA_Chol(FLA_Uplo uplo, FLA_Obj A) { FLA_Error r_val; // Check parameters. if (FLA_Check_error_level() >= FLA_MIN_ERROR_CHECKING) FLA_Chol_check(uplo, A); // Invoke FLA_Chol_internal() with the a control tree for sequential // execution with conventional storage. r_val = FLA_Chol_internal(uplo, A, fla_chol_cntl2); return r_val; } </pre>	<pre> #include "FLAME.h" #ifdef FLA_ENABLE_SUPERMATRIX extern fla_chol_t* flash_sm_chol_cntl; #endif extern fla_chol_t* flash_chol_cntl; FLA_Error FLASH_Chol(FLA_Uplo uplo, FLA_Obj A) { FLA_Error r_val; // Check parameters. if (FLA_Check_error_level() >= FLA_MIN_ERROR_CHECKING) FLA_Chol_check(uplo, A); // Begin a parallel region. FLASH_Queue_begin(); // Enqueue tasks via a SuperMatrix-aware control tree. r_val = FLA_Chol_internal(uplo, A, flash_sm_chol_cntl); // End the parallel region. FLASH_Queue_end(); return r_val; } </pre>

Figure 6.3: Front-end routines for Cholesky factorization with conventional storage (left) and hierarchical storage (right). Notice that `FLASH_Chol()` is capable of invoking both sequential and parallel (SuperMatrix) execution types.

```

void FLA_Hemm_internal( FLA_Side side, FLA_Uplo uplo, FLA_Obj alpha,
                       FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C,
                       fla_hemm_t* cntl );

```

Purpose: Perform a HEMM operation on A , B , and C according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_Hemm()`. Please see the description for `FLA_Hemm()` for further details.

```

void FLA_Herk_internal( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha,
                       FLA_Obj A, FLA_Obj beta, FLA_Obj C,
                       fla_herk_t* cntl );

```

Purpose: Perform a HERK operation on A and C according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_Herk()`. Please see the description for `FLA_Herk()` for further details.

CHOL internal back-end function	CHOL back-end subroutine for lower triangular case
<pre> #include "FLAME.h" FLA_Error FLA_Chol_internal(FLA_Uplo uplo, FLA_Obj A, fla_chol_t* cntl) { FLA_Error r_val = FLA_SUCCESS; #ifdef FLA_ENABLE_INTERNAL_ERROR_CHECKING FLA_Chol_internal_check(uplo, A, cntl); #endif if (FLA_Cntl_matrix_type(cntl) == FLA_HIER && FLA_Obj_elemtype(A) == FLA_MATRIX && FLA_Cntl_variant(cntl) == FLA_SUBPROBLEM) { // Recurse r_val = FLA_Chol_internal(uplo, *FLASH_OBJ_PTR_AT(A), flash_chol_cntl); } else if (FLA_Cntl_matrix_type(cntl) == FLA_HIER && FLA_Obj_elemtype(A) == FLA_SCALAR && FLASH_Queue_get_enabled()) { // Enqueue ENQUEUE_FLASH_Chol(uplo, A, cntl); } else { if (FLA_Cntl_matrix_type(cntl) == FLA_HIER && FLA_Obj_elemtype(A) == FLA_SCALAR && !FLASH_Queue_get_enabled()) { // Execute leaf cntl = flash_chol_cntl_lapack; } // Parameter combinations if (uplo == FLA_LOWER_TRIANGULAR) { r_val = FLA_Chol_l(A, cntl); } else if (uplo == FLA_UPPER_TRIANGULAR) { r_val = FLA_Chol_u(A, cntl); } } return r_val; } </pre>	<pre> #include "FLAME.h" FLA_Error FLA_Chol_l(FLA_Obj A, fla_chol_t* cntl) { FLA_Error r_val = FLA_SUCCESS; if (FLA_Cntl_variant(cntl) == FLA_SUBPROBLEM) { r_val = FLA_Chol_unb_external(FLA_LOWER_TRIANGULAR, A, cntl); } #ifdef FLA_ENABLE_NON_CRITICAL_CODE else if (FLA_Cntl_variant(cntl) == FLA_BLOCKED_VARIANT1) { r_val = FLA_Chol_l_blk_var1(A, cntl); } else if (FLA_Cntl_variant(cntl) == FLA_BLOCKED_VARIANT2) { r_val = FLA_Chol_l_blk_var2(A, cntl); } #endif else if (FLA_Cntl_variant(cntl) == FLA_BLOCKED_VARIANT3) { r_val = FLA_Chol_l_blk_var3(A, cntl); } return r_val; } </pre>

Figure 6.4: Internal back-end function for Cholesky factorization (left) and helper subroutine to handle the lower triangular case (right).

```

void FLA_Her2k_internal( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha,
                        FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C,
                        fla_her2k_t* cntl );

```

Purpose: Perform a HER2K operation on A , B , and C according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_Her2k()`. Please see the description for `FLA_Her2k()` for further details.

```
void FLA_Symm_internal( FLA_Side side, FLA_Uplo uplo, FLA_Obj alpha,
                      FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C,
                      fla_symm_t* cntl );
```

Purpose: Perform a SYMM operation on A , B , and C according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_Symm()`. Please see the description for `FLA_Symm()` for further details.

```
void FLA_Syrk_internal( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha,
                      FLA_Obj A, FLA_Obj beta, FLA_Obj C,
                      fla_syrk_t* cntl );
```

Purpose: Perform a SYRK operation on A and C according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_Syrk()`. Please see the description for `FLA_Syrk()` for further details.

```
void FLA_Syr2k_internal( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha,
                      FLA_Obj A, FLA_Obj B, FLA_Obj beta, FLA_Obj C,
                      fla_syr2k_t* cntl );
```

Purpose: Perform a SYR2K operation on A , B , and C according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_Syr2k()`. Please see the description for `FLA_Syr2k()` for further details.

```
void FLA_Trmm_internal( FLA_Side side, FLA_Uplo uplo, FLA_Trans trans,
                      FLA_Diag diag, FLA_Obj alpha, FLA_Obj A, FLA_Obj B,
                      fla_trmm_t* cntl );
```

Purpose: Perform a TRMM operation on A and B according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_Trmm()`. Please see the description for `FLA_Trmm()` for further details.

```
void FLA_Trsm_internal( FLA_Side side, FLA_Uplo uplo, FLA_Trans trans, FLA_Diag diag,
                       FLA_Obj alpha, FLA_Obj A, FLA_Obj B,
                       fla_trsm_t* cntl );
```

Purpose: Perform a TRSM operation on A and B according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_Trsm()`. Please see the description for `FLA_Trsm()` for further details.

6.5.7.2 LAPACK operations

```
FLA_Error FLA_Chol_internal( FLA_Uplo uplo, FLA_Obj A, fla_chol_t* cntl );
```

Purpose: Perform a CHOL operation on A according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_Chol()`. Please see the description for `FLA_Chol()` for further details.

```
FLA_Error FLA_Trinv_internal( FLA_Uplo uplo, FLA_Diag diag, FLA_Obj A,
                              fla_trinv_t* cntl );
```

Purpose: Perform a TRINV operation on A according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_Trinv()`. Please see the description for `FLA_Trinv()` for further details.

```
void FLA_Ttmm_internal( FLA_Uplo uplo, FLA_Obj A, fla_ttmm_t* cntl );
```

Purpose: Perform a TTMM operation on A according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_Ttmm()`. Please see the description for `FLA_Ttmm()` for further details.

```
void FLA_SPDinv_internal( FLA_Uplo uplo, FLA_Obj A, fla_spdinv_t* cntl );
```

Purpose: Perform a SPDINV operation on A according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_SPDinv()`. Please see the description for `FLA_SPDinv()` for further details.

```
void FLA_Hess_internal( FLA_Obj A, FLA_Obj t, int ilo, int ihi,  
                      fla_hess_t* cntl );
```

Purpose: Perform a HESS operation on A according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_Hess()`. Please see the description for `FLA_Hess()` for further details.

```
FLA_Error FLA_LU_nopiv_internal( FLA_Obj A, fla_lu_t* cntl );
```

Purpose: Perform a LUNOPIV operation on A according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_LU_nopiv()`. Please see the description for `FLA_LU_nopiv()` for further details.

```
FLA_Error FLA_LU_piv_internal( FLA_Obj A, FLA_Obj p, fla_lu_t* cntl );
```

Purpose: Perform a LUPIV operation on A according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_LU_nopiv()`. Please see the description for `FLA_LU_nopiv()` for further details.

```
void FLA_QR_UT_internal( FLA_Obj A, FLA_Obj T, fla_qrut_t* cntl );
```

Purpose: Perform a QRUT operation on A according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_QR_UT()`. Please see the description for `FLA_QR_UT()` for further details.

```
void FLA_LQ_UT_internal( FLA_Obj A, FLA_Obj T, fla_lq_t* cntl );
```

Purpose: Perform a LQUT operation on A according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_LQ_UT()`. Please see the description for `FLA_LQ_UT()` for further details.

```
void FLA_Sylv_internal( FLA_Trans transa, FLA_Trans transb, FLA_Obj isgn,
                       FLA_Obj A, FLA_Obj B, FLA_Obj C, FLA_Obj scale,
                       fla_sylv_t* cntl );
```

Purpose: Perform a SYLV operation on A , B , and C according to the parameters specified by control tree node `cntl`.

Constraints:

- `cntl` must not be NULL.

More Info: This function's interface is similar to that of `FLA_Sylv()`. Please see the description for `FLA_Sylv()` for further details.

6.5.8 Algorithmic variants

The algorithmic variants in `libflame` are coded differently than earlier incarnations of FLAME/C. Figure 6.5 illustrates these differences for the blocked FLAME/C implementation of algorithmic variant 3 of Cholesky factorization. The top-left code example shows what the code might look like when the programmer used unblocked FLAME variants to perform subproblems. The top-right code is similar, except that it uses blocked variants. Note that this code uses the same algorithmic blocksize for its subproblems as it does for matrix partitioning within the Cholesky algorithm. The bottom-left code again shows how the FLAME group used to present its codes during lectures and presentations, where the routines `FLA_Chol()`, `FLA_Trsm()`, and `FLA_Syrk()` were wrappers to external implementations of those operations.¹ Finally, the bottom-right code shows how algorithms are now coded within `libflame`, using control trees. The most notable difference between this code and the others is that the subproblems invoke internal back-end routines for the operation in question rather than statically specifying an unblocked, blocked, or external implementation.

¹ Notice that this class of function names is now reserved for the user-level front-end interfaces documented in Sections 5.7.1.3 and 5.7.2, and the corresponding external routines are now explicitly named as `FLA_Chol_unb_external()`, `FLA_Trsm_external()`, and `FLA_Syrk_external()`.

calling unblocked FLAME/C variants	calling blocked FLAME/C variants
<pre> FLA_Error FLA_Chol_l_blk_var3(FLA_Obj A, dim_t nb_alg) { FLA_Obj ATL, ATR, A00, A01, A02, ABL, ABR, A10, A11, A12, A20, A21, A22; dim_t b; int value; FLA_Part_2x2(A, &ATL, &ATR, &ABL, &ABR, 0, 0, FLA_TL); while (FLA_Obj_length(ATL) < FLA_Obj_length(A)) { b = min(FLA_Obj_length(ABR), nb_alg); FLA_Repart_2x2_to_3x3(ATL, /**/ ATR, &A00, /**/ &A01, &A02, /* ***** */ /* ***** */ &A10, /**/ &A11, &A12, ABL, /**/ ABR, &A20, /**/ &A21, &A22, b, b, FLA_BR); /*-----*/ value = FLA_Chol_l_unb_var2(A11); if (value != FLA_SUCCESS) return (FLA_Obj_length(A00) + value); FLA_Trsm_rlt_unb_var3(FLA_NONUNIT_DIAG, FLA_ONE, A11, A21); FLA_Syrk_ln_unb_var2(FLA_MINUS_ONE, A21, FLA_ONE, A22); /*-----*/ FLA_Cont_with_3x3_to_2x2(&ATL, /**/ &ATR, A00, A01, /**/ A02, A10, A11, /**/ A12, /* ***** */ /* ***** */ &ABL, /**/ &ABR, A20, A21, /**/ A22, FLA_TL); } return value; } </pre>	<pre> FLA_Error FLA_Chol_l_blk_var3(FLA_Obj A, dim_t nb_alg) { FLA_Obj ATL, ATR, A00, A01, A02, ABL, ABR, A10, A11, A12, A20, A21, A22; dim_t b; int value; FLA_Part_2x2(A, &ATL, &ATR, &ABL, &ABR, 0, 0, FLA_TL); while (FLA_Obj_length(ATL) < FLA_Obj_length(A)) { b = min(FLA_Obj_length(ABR), nb_alg); FLA_Repart_2x2_to_3x3(ATL, /**/ ATR, &A00, /**/ &A01, &A02, /* ***** */ /* ***** */ &A10, /**/ &A11, &A12, ABL, /**/ ABR, &A20, /**/ &A21, &A22, b, b, FLA_BR); /*-----*/ value = FLA_Chol_l_blk_var2(A11, nb_alg); if (value != FLA_SUCCESS) return (FLA_Obj_length(A00) + value); FLA_Trsm_rlt_blk_var3(FLA_NONUNIT_DIAG, FLA_ONE, A11, A21, nb_alg); FLA_Syrk_ln_blk_var2(FLA_MINUS_ONE, A21, FLA_ONE, A22, nb_alg); /*-----*/ FLA_Cont_with_3x3_to_2x2(&ATL, /**/ &ATR, A00, A01, /**/ A02, A10, A11, /**/ A12, /* ***** */ /* ***** */ &ABL, /**/ &ABR, A20, A21, /**/ A22, FLA_TL); } return value; } </pre>
calling unblocked external implementations	using control trees
<pre> FLA_Error FLA_Chol_l_blk_var3(FLA_Obj A, dim_t nb_alg) { FLA_Obj ATL, ATR, A00, A01, A02, ABL, ABR, A10, A11, A12, A20, A21, A22; dim_t b; int value; FLA_Part_2x2(A, &ATL, &ATR, &ABL, &ABR, 0, 0, FLA_TL); while (FLA_Obj_length(ATL) < FLA_Obj_length(A)) { b = min(FLA_Obj_length(ABR), nb_alg); FLA_Repart_2x2_to_3x3(ATL, /**/ ATR, &A00, /**/ &A01, &A02, /* ***** */ /* ***** */ &A10, /**/ &A11, &A12, ABL, /**/ ABR, &A20, /**/ &A21, &A22, b, b, FLA_BR); /*-----*/ value = FLA_Chol(FLA_LOWER_TRIANGULAR, A11); if (value != FLA_SUCCESS) return (FLA_Obj_length(A00) + value); FLA_Trsm(FLA_RIGHT, FLA_LOWER_TRIANGULAR, FLA_TRANSPOSE, FLA_NONUNIT_DIAG, FLA_ONE, A11, A21); FLA_Syrk(FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_MINUS_ONE, A21, FLA_ONE, A22); /*-----*/ FLA_Cont_with_3x3_to_2x2(&ATL, /**/ &ATR, A00, A01, /**/ A02, A10, A11, /**/ A12, /* ***** */ /* ***** */ &ABL, /**/ &ABR, A20, A21, /**/ A22, FLA_TL); } return value; } </pre>	<pre> FLA_Error FLA_Chol_l_blk_var3(FLA_Obj A, fla_chol_t* cntl) { FLA_Obj ATL, ATR, A00, A01, A02, ABL, ABR, A10, A11, A12, A20, A21, A22; dim_t b; int value; FLA_Part_2x2(A, &ATL, &ATR, &ABL, &ABR, 0, 0, FLA_TL); while (FLA_Obj_length(ATL) < FLA_Obj_length(A)) { b = FLA_Determine_blocksize(ABR, FLA_BR, FLA_Cntl_blocksize(cntl)); FLA_Repart_2x2_to_3x3(ATL, /**/ ATR, &A00, /**/ &A01, &A02, /* ***** */ /* ***** */ &A10, /**/ &A11, &A12, ABL, /**/ ABR, &A20, /**/ &A21, &A22, b, b, FLA_BR); /*-----*/ value = FLA_Chol_internal(FLA_LOWER_TRIANGULAR, A11, FLA_Cntl_sub_chol(cntl)); if (value != FLA_SUCCESS) return (FLA_Obj_length(A00) + value); FLA_Trsm_internal(FLA_RIGHT, FLA_LOWER_TRIANGULAR, FLA_TRANSPOSE, FLA_NONUNIT_DIAG, FLA_ONE, A11, A21, FLA_Cntl_sub_trsm(cntl)); FLA_Syrk_internal(FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, FLA_MINUS_ONE, A21, FLA_ONE, A22, FLA_Cntl_sub_syrk(cntl)); /*-----*/ FLA_Cont_with_3x3_to_2x2(&ATL, /**/ &ATR, A00, A01, /**/ A02, A10, A11, /**/ A12, /* ***** */ /* ***** */ &ABL, /**/ &ABR, A20, A21, /**/ A22, FLA_TL); } return value; } </pre>

Figure 6.5: A Cholesky factorization algorithm (variant 3) implemented using various deprecated methods for statically encoding the subproblem implementation (top-left, top-right, bottom-left) and the more modern style using control trees (bottom-right).

6.6 Parameter and error checking

6.6.1 Linear algebra parameters

```
FLA_Error FLA_Check_valid_side( FLA_Side side );
```

Purpose: Confirm that `side` is one of the following values defined for the `FLA_Side` type: `FLA_LEFT`, `FLA_RIGHT`, `FLA_TOP`, `FLA_BOTTOM`.

Returns: `FLA_SUCCESS` if `side` is valid; `FLA_INVALID_SIDE` otherwise.

```
FLA_Error FLA_Check_valid_uplo( FLA_Uplo uplo );
```

Purpose: Confirm that `uplo` is one of the following values defined for the `FLA_Uplo` type: `FLA_LOWER_TRIANGULAR`, `FLA_UPPER_TRIANGULAR`.

Returns: `FLA_SUCCESS` if `uplo` is valid; `FLA_INVALID_UPLO` otherwise.

```
FLA_Error FLA_Check_valid_trans( FLA_Trans trans );
```

Purpose: Confirm that `trans` is one of the following values defined for the `FLA_Trans` type: `FLA_NO_TRANSPOSE`, `FLA_TRANSPOSE`, `FLA_CONJ_TRANSPOSE`, `FLA_CONJ_NO_TRANSPOSE`.

Returns: `FLA_SUCCESS` if `trans` is valid; `FLA_INVALID_TRANS` otherwise.

```
FLA_Error FLA_Check_valid_real_trans( FLA_Trans trans );
```

Purpose: Confirm that `trans` is either `FLA_NO_TRANSPOSE` or `FLA_TRANSPOSE`.

Notes: This check is typically used with `trans` arguments that are expected to be applied to real matrices.

Returns: `FLA_SUCCESS` if the `trans` argument is either `FLA_NO_TRANSPOSE` or `FLA_TRANSPOSE`. `FLA_INVALID_REAL_TRANS` otherwise.

```
FLA_Error FLA_Check_valid_complex_trans( FLA_Trans trans );
```

Purpose: Confirm that `trans` is either `FLA_NO_TRANSPOSE` or `FLA_CONJ_TRANSPOSE`.

Notes: This check is typically used with `trans` arguments that are expected to be applied to Hermitian matrices.

Returns: `FLA_SUCCESS` if the `trans` argument is either `FLA_NO_TRANSPOSE` or `FLA_CONJ_TRANSPOSE`. `FLA_INVALID_COMPLEX_TRANS` otherwise.

```
FLA_Error FLA_Check_valid_blas_trans( FLA_Trans trans );
```

Purpose: Confirm that `trans` is either `FLA_NO_TRANSPOSE`, `FLA_TRANSPOSE`, or `FLA_CONJ_TRANSPOSE`.

Notes: This check is typically used with `trans` arguments that are expected to be applied to general matrices. Valid values correspond to those supported by the BLAS interface, and thus `FLA_CONJ_NO_TRANSPOSE` is not allowed.

Returns: `FLA_SUCCESS` if the `trans` argument is either `FLA_NO_TRANSPOSE`, `FLA_TRANSPOSE`, or `FLA_CONJ_TRANSPOSE`. `FLA_INVALID_BLAS_TRANS` otherwise.

```
FLA_Error FLA_Check_valid_diag( FLA_Diag diag );
```

Purpose: Confirm that `diag` is one of the following values defined for the `FLA_Diag` type: `FLA_NONUNIT_DIAG`, `FLA_UNIT_DIAG`, `FLA_ZERO_DIAG`.

Returns: `FLA_SUCCESS` if `diag` is valid; `FLA_INVALID_DIAG` otherwise.

```
FLA_Error FLA_Check_valid_conj( FLA_Conj conj );
```

Purpose: Confirm that `conj` is one of the following values defined for the `FLA_Conj` type: `FLA_NO_CONJUGATE`, `FLA_CONJUGATE`.

Returns: `FLA_SUCCESS` if `conj` is valid; `FLA_INVALID_CONJ` otherwise.

```
FLA_Error FLA_Check_valid_direct( FLA_Direct direct );
```

Purpose: Confirm that `direct` is one of the following values defined for the `FLA_Direct` type: `FLA_FORWARD`, `FLA_BACKWARD`.

Returns: `FLA_SUCCESS` if `direct` is valid; `FLA_INVALID_DIRECT` otherwise.

```
FLA_Error FLA_Check_valid_storev( FLA_Store storev );
```

Purpose: Confirm that `storev` is one of the following values defined for the `FLA_Store` type: `FLA_COLUMNWISE`, `FLA_ROWWISE`.

Returns: `FLA_SUCCESS` if `storev` is valid; `FLA_INVALID_STOREV` otherwise.

```
FLA_Error FLA_Check_valid_quadrant( FLA_Quadrant quadrant );
```

Purpose: Confirm that `quadrant` is one of the following values defined for the `FLA_Quadrant` type: `FLA_TL`, `FLA_TR`, `FLA_BL`, `FLA_BR`.

Returns: `FLA_SUCCESS` if `quadrant` is valid; `FLA_INVALID_QUADRANT` otherwise.

6.6.2 Datatypes

```
FLA_Error FLA_Check_valid_datatype( FLA_Datatype datatype );
```

Purpose: Confirm that `datatype` is one of the following values defined for the `FLA_Datatype` type: `FLA_FLOAT`, `FLA_DOUBLE`, `FLA_COMPLEX`, `FLA_DOUBLE_COMPLEX`, `FLA_INT`, `FLA_CONSTANT`.

Returns: `FLA_SUCCESS` if `datatype` is valid; `FLA_INVALID_DATATYPE` otherwise.

```
FLA_Error FLA_Check_valid_object_datatype( FLA_Obj A );
```

Purpose: Confirm that the datatype of `A` is one of the following values defined for the `FLA_Datatype` type: `FLA_INT`, `FLA_FLOAT`, `FLA_DOUBLE`, `FLA_COMPLEX`, `FLA_DOUBLE_COMPLEX`, `FLA_CONSTANT`.

Returns: `FLA_SUCCESS` if the datatype of `A` is valid; `FLA_INVALID_DATATYPE` otherwise.

```
FLA_Error FLA_Check_floating_datatype( FLA_Datatype datatype );
```

Purpose: Confirm that `datatype` refers to one of the following floating point type values defined for `FLA_Datatype`: `FLA_FLOAT`, `FLA_DOUBLE`, `FLA_COMPLEX`, `FLA_DOUBLE_COMPLEX`, `FLA_CONSTANT`.

Notes: Though it is a distinct type, `FLA_CONSTANT` is polymorphic and thus may be considered floating point for the purposes of this function.

Returns: `FLA_SUCCESS` if `datatype` is floating point; `FLA_INVALID_FLOATING_DATATYPE` otherwise.

```
FLA_Error FLA_Check_int_datatype( FLA_Datatype datatype );
```

Purpose: Confirm that `datatype` refers to one of the following integer type values defined for `FLA_Datatype`: `FLA_INT`, `FLA_CONSTANT`.

Notes: Though it is a distinct type, `FLA_CONSTANT` is polymorphic and thus may be considered integer for the purposes of this function.

Returns: `FLA_SUCCESS` if `datatype` is integer; `FLA_INVALID_INTEGER_DATATYPE` otherwise.

```
FLA_Error FLA_Check_real_datatype( FLA_Datatype datatype );
```

Purpose: Confirm that `datatype` refers to one of the following real numerical type values defined for `FLA_Datatype`: `FLA_FLOAT`, `FLA_DOUBLE`, `FLA_CONSTANT`.

Notes: Though it is a distinct type, `FLA_CONSTANT` is polymorphic and thus may be considered real for the purposes of this function.

Returns: `FLA_SUCCESS` if `datatype` is real; `FLA_INVALID_REAL_DATATYPE` otherwise.

```
FLA_Error FLA_Check_complex_datatype( FLA_Datatype datatype );
```

Purpose: Confirm that `datatype` refers to one of the following complex numerical type values defined for `FLA_Datatype`: `FLA_COMPLEX`, `FLA_DOUBLE_COMPLEX`, `FLA_CONSTANT`.

Notes: Though it is a distinct type, `FLA_CONSTANT` is polymorphic and thus may be considered complex for the purposes of this function.

Returns: `FLA_SUCCESS` if `datatype` is complex; `FLA_INVALID_COMPLEX_DATATYPE` otherwise.

```
FLA_Error FLA_Check_nonconstant_datatype( FLA_Datatype datatype );
```

Purpose: Confirm that `datatype` is one of the following non-constant values defined for the `FLA_Datatype` type: `FLA_FLOAT`, `FLA_DOUBLE`, `FLA_COMPLEX`, `FLA_DOUBLE_COMPLEX`, `FLA_INT`.

Notes: This function is similar to `FLA_Check_valid_datatype()`, except that it does not allow `FLA_CONSTANT`.

Returns: `FLA_SUCCESS` if `datatype` specifies a non-constant datatype; `FLA_INVALID_NONCONSTANT_DATATYPE` otherwise.

```
FLA_Error FLA_Check_floating_object( FLA_Obj A );
```

Purpose: Confirm that the datatype of `A` is one of the following floating point type values defined for `FLA_Datatype`: `FLA_FLOAT`, `FLA_DOUBLE`, `FLA_COMPLEX`, `FLA_DOUBLE_COMPLEX`, `FLA_CONSTANT`.

Notes: Though it is a distinct type, `FLA_CONSTANT` is polymorphic and thus may be considered floating point for the purposes of this function.

Returns: `FLA_SUCCESS` if the datatype of `A` is floating point; `FLA_OBJECT_NOT_FLOATING_POINT` otherwise.

```
FLA_Error FLA_Check_int_object( FLA_Obj A );
```

Purpose: Confirm that the datatype of `A` is one of the following integer type values defined for `FLA_Datatype`: `FLA_INT`, `FLA_CONSTANT`.

Notes: Though it is a distinct type, `FLA_CONSTANT` is polymorphic and thus may be considered integer for the purposes of this function.

Returns: `FLA_SUCCESS` if the datatype of `A` is integer; `FLA_OBJECT_NOT_INTEGER` otherwise.

```
FLA_Error FLA_Check_real_object( FLA_Obj A );
```

Purpose: Confirm that the datatype of `A` is one of the following real numerical type values defined for `FLA_Datatype`: `FLA_FLOAT`, `FLA_DOUBLE`, `FLA_CONSTANT`.

Notes: Though it is a distinct type, `FLA_CONSTANT` is polymorphic and thus may be considered real for the purposes of this function.

Returns: `FLA_SUCCESS` if the datatype of `A` is real; `FLA_OBJECT_NOT_REAL` otherwise.

```
FLA_Error FLA_Check_complex_object( FLA_Obj A );
```

- Purpose:** Confirm that the datatype of *A* is one of the following complex numerical type values defined for FLA_Datatype: FLA_COMPLEX, FLA_DOUBLE_COMPLEX, FLA_CONSTANT.
- Notes:** Though it is a distinct type, FLA_CONSTANT is polymorphic and thus may be considered complex for the purposes of this function.
- Returns:** FLA_SUCCESS if the datatype of *A* is complex; FLA_OBJECT_NOT_COMPLEX otherwise.

```
FLA_Error FLA_Check_nonconstant_object( FLA_Obj A );
```

- Purpose:** Confirm that the datatype of *A* is one of the following non-constant values defined for the FLA_Datatype type: FLA_INT, FLA_FLOAT, FLA_DOUBLE, FLA_COMPLEX, FLA_DOUBLE_COMPLEX.
- Returns:** FLA_SUCCESS if the datatype of *A* is a non-constant datatype; FLA_OBJECT_NOT_NONCONSTANT otherwise.

```
FLA_Error FLA_Check_identical_object_datatype( FLA_Obj A, FLA_Obj B );
```

- Purpose:** Confirm that *A* and *B* have identical datatypes.
- Notes:** This function enforces literal equality between the datatype fields of *A* and *B*.
- Returns:** FLA_SUCCESS if *A* and *B* have identical datatypes; FLA_OBJECT_DATATYPES_NOT_EQUAL otherwise.

```
FLA_Error FLA_Check_consistent_object_datatype( FLA_Obj A, FLA_Obj B );
```

- Purpose:** Confirm that the datatype of *A* is consistent with the datatype of *B*.
- Notes:** This function is similar to FLA_Check_identical_object_datatype(), except that it considers objects of datatype FLA_CONSTANT to be consistent with all other datatypes.
- Returns:** FLA_SUCCESS if the datatype of *A* is equal to the datatype of *B*; FLA_INCONSISTENT_DATATYPES otherwise.

```
FLA_Error FLA_Check_consistent_datatype( FLA_Datatype datatype, FLA_Obj A );
```

- Purpose:** Confirm that *datatype* is consistent with the datatype of *A*.
- Notes:** This function is similar to FLA_Check_consistent_object_datatype() except that it takes one datatype value and one object as its arguments instead of two objects.
- Returns:** FLA_SUCCESS if *datatype* is equal to the datatype of *A*; FLA_INCONSISTENT_DATATYPES otherwise.

```
FLA_Error FLA_Check_identical_object_precision( FLA_Obj A, FLA_Obj B );
```

Purpose: Confirm that the numerical precision of the datatype of A matches the numerical precision of the datatype of B .

Notes: The function first verifies that both A and B are floating point objects. If one or both objects are not floating point, `FLA_OBJECT_NOT_FLOATING_POINT` is returned.

Returns: `FLA_SUCCESS` if the datatype precision of A is equal to the datatype precision of B ; `FLA_INCONSISTENT_OBJECT_PRECISION` otherwise.

```
FLA_Error FLA_Check_conj_and_datatype( FLA_Conj conj, FLA_Obj A );
```

Purpose: Confirm, if A is real, that `conj` is not `FLA_CONJUGATE`.

Returns: `FLA_SUCCESS` if the `conj` argument is not in conflict with the complexness of A ; `FLA_INVALID_CONJ_GIVEN_DATATYPE` otherwise.

```
FLA_Error FLA_Check_conj_trans_and_datatype( FLA_Trans trans, FLA_Obj A );
```

Purpose: Confirm, if A is real, that `trans` is neither `FLA_CONJ_TRANSPOSE` nor `FLA_CONJ_NO_TRANSPOSE`.

Returns: `FLA_SUCCESS` if the `trans` argument is not in conflict with the complexness of A ; `FLA_INVALID_TRANS_GIVEN_DATATYPE` otherwise.

6.6.3 Element types

```
FLA_Error FLA_Check_valid_elemtype( FLA_Elemtype elemtype );
```

Purpose: Confirm that `elemtype` is one of the following values defined for the `FLA_Elemtype` type: `FLA_SCALAR`, `FLA_MATRIX`.

Returns: `FLA_SUCCESS` if `elemtype` is valid; `FLA_INVALID_ELEMTYPE` otherwise.

```
FLA_Error FLA_Check_object_scalar_elemtype( FLA_Obj A );
```

Purpose: Confirm that the element type of A is `FLA_SCALAR`.

Returns: `FLA_SUCCESS` if the element type of A is `FLA_SCALAR`; `FLA_OBJECT_NOT_SCALAR_ELEMTYPE` otherwise.

```
FLA_Error FLA_Check_object_matrix_elemtype( FLA_Obj A );
```

Purpose: Confirm that the element type of A is `FLA_MATRIX`.

Returns: `FLA_SUCCESS` if the element type of A is `FLA_MATRIX`; `FLA_OBJECT_NOT_MATRIX_ELEMTYPE` otherwise.

6.6.4 Object dimensions

```
FLA_Error FLA_Check_square( FLA_Obj A );
```

Purpose: Confirm that A is square.

Returns: FLA_SUCCESS if A is square; FLA_OBJECT_NOT_SQUARE otherwise.

```
FLA_Error FLA_Check_if_scalar( FLA_Obj A );
```

Purpose: Confirm that A is a scalar (ie: that A is 1×1).

Returns: FLA_SUCCESS if A is a scalar; FLA_OBJECT_NOT_SCALAR otherwise.

```
FLA_Error FLA_Check_if_vector( FLA_Obj A );
```

Purpose: Confirm that A is a vector (ie: that A is $n \times 1$ or $1 \times n$ for $n \geq 0$).

Returns: FLA_SUCCESS if A is a vector; FLA_OBJECT_NOT_VECTOR otherwise.

```
FLA_Error FLA_Check_conformal_dims( FLA_Trans trans, FLA_Obj A, FLA_Obj B );
```

Purpose: Confirm that A and B have conformal dimensions. If **trans** is FLA_TRANSPOSE or FLA_CONJ_TRANSPOSE, then the function confirms that A and B^T have conformal dimensions.

Returns: FLA_SUCCESS if A and B have conformal dimensions; FLA_NONCONFORMAL_DIMENSIONS otherwise.

```
FLA_Error FLA_Check_matrix_matrix_dims( FLA_Trans transa, FLA_Trans transb,
                                         FLA_Obj A, FLA_Obj B, FLA_Obj C );
```

Purpose: Confirm that A , B , and C have conformal dimensions suitable for a matrix-matrix operation of the form $C := C + AB$ where C is $m \times n$ and A and B are $m \times k$ and $k \times n$, respectively, after optionally transpositions, per **transa** and **transb**.

Returns: FLA_SUCCESS if A , B , and C have conformal dimensions; FLA_NONCONFORMAL_DIMENSIONS otherwise.

```
FLA_Error FLA_Check_matrix_vector_dims( FLA_Trans trans, FLA_Obj A, FLA_Obj x, FLA_Obj y );
```

Purpose: Confirm that A , x , and y have conformal dimensions suitable for a matrix-vector operation of the form $y := y + Ax$ where A is optionally transposed, per **trans**.

Returns: FLA_SUCCESS if A , x , and y have conformal dimensions; FLA_NONCONFORMAL_DIMENSIONS otherwise.

```
FLA_Error FLA_Check_equal_vector_lengths( FLA_Obj x, FLA_Obj y );
```

- Purpose:** Confirm that x and y , which are assumed to be vectors, have equal lengths.
- Notes:** This function works as expected if one or both arguments are row vectors. That is, “length” for the purposes of this function refers to the length of the vector, not the number of rows in the object.
- Returns:** FLA_SUCCESS if x and y have equal lengths; FLA_UNEQUAL_VECTOR_LENGTHS otherwise.

```
FLA_Error FLA_Check_vector_length( FLA_Obj x, dim_t expected_length );
```

- Purpose:** Confirm that x , which is assumed to be a column vector, is of length `expected_length`.
- Notes:** This function checks only the number of rows of x . Therefore, this function will not work as expected with row vectors.
- Returns:** FLA_SUCCESS if the number of rows in x is `expected_length`; FLA_INVALID_VECTOR_LENGTH otherwise.

```
FLA_Error FLA_Check_vector_length_min( FLA_Obj x, dim_t min_length );
```

- Purpose:** Confirm that x , which is assumed to be a column vector, is at least `min_length` in length.
- Notes:** This function checks only the number of rows of x . Therefore, this function will not work as expected with row vectors.
- Returns:** FLA_SUCCESS if the number of rows in x is at least `min_length`; FLA_VECTOR_LENGTH_BELOW_MIN otherwise.

```
FLA_Error FLA_Check_object_dims( FLA_Trans trans, dim_t m, dim_t n, FLA_Obj A );
```

- Purpose:** Confirm that matrix A is $m \times n$. If `trans` is FLA_TRANSPOSE or FLA_CONJ_TRANSPOSE, then the function will instead confirm that A is $n \times m$.
- Returns:** FLA_SUCCESS if the dimensions of A are identical to those specified; FLA_SPECIFIED_OBJ_DIM_MISMATCH otherwise.


```
FLA_Error FLA_Check_submatrix_dims_and_offset( int m, int n, int i, int j, FLA_Obj A );
```

Purpose: Confirm that the $m \times n$ submatrix that has its top-left element located at row-column offset (i, j) (indexed from zero) does not exceed the bounds of matrix A . In other words, the following constraints are enforced:

- $i \leq m(A)$
- $j \leq n(A)$
- $i + m \leq m(A)$
- $j + n \leq n(A)$

where $m(A)$ and $n(A)$ denote the number of rows and number of columns in A , respectively.

Notes: Strictly speaking, only the last two constraints are needed. However, we first check against the first two constraints to allow us to distinguish between situations where the offsets are invalid (in which case the value of the matrix dimensions are moot), and situations where the offsets are valid, but the submatrix dimensions places the submatrix beyond the bounds of A .

Returns: FLA_SUCCESS if the specified submatrix is within the bounds of A ; otherwise, FLA_INVALID_SUBMATRIX_DIMS if one of the first two constraints is violated, and FLA_INVALID_SUBMATRIX_OFFSET if the first two constraints are met but one of the last two constraints is violated.

```
FLA_Error FLA_Check_m_against_ldim( int m, int ldim );
```

Purpose: Confirm that m is valid given $ldim$. That is, confirm that m is less than or equal to $ldim$.

Returns: FLA_SUCCESS if m is valid given $ldim$; FLA_INVALID_M_GIVEN_LDIM otherwise.

```
FLA_Error FLA_Check_adjacent_objects_2x2( FLA_Obj ATL, FLA_Obj ATR,
                                           FLA_Obj ABL, FLA_Obj ABR );
```

Purpose: Confirm that views A_{TL} , A_{TR} , A_{BL} , and A_{BR} are vertically and horizontally aligned and adjacent, and that views that are vertically or horizontally adjacent have dimensions appropriately matched for them to form quadrants of a single larger view of the base object.

Returns: FLA_SUCCESS if the four object views are aligned and adjacent, and have matching dimensions; otherwise, one of the following, depending on the mismatch:

- FLA_OBJECTS_NOT_VERTICALLY_ADJ
- FLA_OBJECTS_NOT_VERTICALLY_ALIGNED
- FLA_OBJECTS_NOT_HORIZONTALLY_ADJ
- FLA_OBJECTS_NOT_HORIZONTALLY_ALIGNED
- FLA_ADJACENT_OBJECT_DIM_MISMATCH

```
FLA_Error FLA_Check_adjacent_objects_2x1( FLA_Obj AT,
                                           FLA_Obj AB );
```

Purpose: Confirm that views A_T and A_B are vertically and aligned and adjacent, and that the views have column dimensions appropriately matched for them to form a vertical panel a single larger view of the base object.

Returns: FLA_SUCCESS if the two object views are vertically aligned and adjacent, and have matching column dimensions; otherwise, one of the following, depending on the mismatch:

- FLA_OBJECTS_NOT_VERTICALLY_ADJ
- FLA_OBJECTS_NOT_VERTICALLY_ALIGNED
- FLA_ADJACENT_OBJECT_DIM_MISMATCH

```
FLA_Error FLA_Check_adjacent_objects_1x2( FLA_Obj AL, FLA_Obj AR );
```

Purpose: Confirm that views A_L and A_R are horizontally and aligned and adjacent, and that the views have row dimensions appropriately matched for them to form a horizontal panel a single larger view of the base object.

Returns: FLA_SUCCESS if the two object views are horizontally aligned and adjacent, and have matching row dimensions; otherwise, one of the following, depending on the mismatch:

- FLA_OBJECTS_NOT_HORIZONTALLY_ADJ
- FLA_OBJECTS_NOT_HORIZONTALLY_ALIGNED
- FLA_ADJACENT_OBJECT_DIM_MISMATCH

6.6.5 UNIX file I/O

```
FLA_Error FLA_Check_file_descriptor( int fd );
```

Purpose: Confirm that `fd`, which is assumed to have been returned from the UNIX/Linux `open()` function, is a valid file descriptor.

Notes: The UNIX/Linux `open()` function returns `-1` when it fails to successfully open a file.

Returns: FLA_SUCCESS if `fd` is valid; FLA_OPEN_RETURNED_ERROR otherwise.

```
FLA_Error FLA_Check_close_result( int r_val );
```

Purpose: Confirm that `r_val`, which is assumed to have been returned from `close()`, does not indicate that an error has occurred.

Notes: The UNIX/Linux `close()` function returns `-1` when it fails to successfully close a file.

Returns: FLA_SUCCESS if `r_val` indicates no error; FLA_CLOSE_RETURNED_ERROR otherwise.

```
FLA_Error FLA_Check_lseek_result( off_t requested_offset, int r_val );
```

- Purpose:** Confirm that `r_val`, which is assumed to have been returned from `lseek()`, does not indicate that an error has occurred. It is further assumed that `requested_offset` was the byte offset passed to `lseek()` when `lseek()` returned `r_val`.
- Notes:** The UNIX/Linux `lseek()` function returns the resulting byte offset relative to the beginning of the file. If `lseek()` is unsuccessful, it returns `-1`.
- Returns:** `FLA_SUCCESS` if `r_val` indicates no error; `FLA_LSEEK_RETURNED_ERROR` otherwise.

```
FLA_Error FLA_Check_read_result( size_t requested_size, ssize_t r_val );
```

- Purpose:** Confirm that `r_val`, which is assumed to have been returned from `read()`, does not indicate that an error has occurred.
- Notes:** Under normal circumstances, the UNIX/Linux `read()` function returns the number of bytes successfully read into the destination buffer, where zero indicates no bytes were read due to attempting to read at end of file. It is not considered an error for this return value to be less than the requested number of bytes. If an actual error occurs, `read()` returns `-1`. Currently `FLA_Check_read_result()` only returns an error code if `read()` returns `-1`. Thus, the `requested_size` argument is not referenced, but may be used in the future to provide warnings.
- Returns:** `FLA_SUCCESS` if `r_val` indicates no error; `FLA_READ_RETURNED_ERROR` otherwise.

```
FLA_Error FLA_Check_write_result( size_t requested_size, ssize_t r_val );
```

- Purpose:** Confirm that `r_val`, which is assumed to have been returned from `write()`, does not indicate that an error has occurred.
- Notes:** Under normal circumstances, the UNIX/Linux `write()` function returns the number of bytes successfully written from the source buffer, where zero indicates no bytes were written. It is not considered an error for this return value to be less than the requested number of bytes. If an actual error occurs, `write()` returns `-1`. Currently `FLA_Check_write_result()` only returns an error code if `write()` returns `-1`. Thus, the `requested_size` argument is not referenced, but may be used in the future to provide warnings.
- Returns:** `FLA_SUCCESS` if `r_val` indicates no error; `FLA_WRITE_RETURNED_ERROR` otherwise.

```
FLA_Error FLA_Check_unlink_result( int r_val );
```

- Purpose:** Confirm that `r_val`, which is assumed to have been returned from `unlink()`, does not indicate that an error has occurred.
- Notes:** The UNIX/Linux `unlink()` function returns `-1` when it fails to successfully delete a file from the filesystem.
- Returns:** `FLA_SUCCESS` if `r_val` indicates no error; `FLA_UNLINK_RETURNED_ERROR` otherwise.

6.6.6 Operation-specific errors

```
FLA_Error FLA_Check_chol_failure( FLA_Error r_val );
```

Purpose: Confirm that `r_val`, which is assumed to have been returned from the an LAPACK-compatible Cholesky factorization routine, does not indicate that the input matrix was found to be non-symmetric positive definite (or non-Hermitian positive definite).

Notes: For the purposes of this function, “LAPACK-compatible” means the Cholesky factorization routine returns the row/column offset (indexing from one) of the diagonal entry that was found to be negative.

Returns: `FLA_SUCCESS` if `r_val` is valid; `FLA_CHOL_FAILED_MATRIX_NOT_SPD` otherwise.

```
FLA_Error FLA_Check_block_householder_transform( FLA_Store storev,
                                                  FLA_Obj A, FLA_Obj S );
```

Purpose: Confirm that matrix the dimensions of A are compatible with the dimensions of a block Householder matrix S . Specifically, if `storev` is `FLA_COLUMNWISE`, then the number of columns of A must match the order of S . Otherwise, if `storev` is `FLA_ROWWISE`, then the number of rows of A must match the order of S .

Returns: `FLA_SUCCESS` if the dimensions of A and S are compatible; `FLA_BLOCK_HOUSEH_DIM_MISMATCH` otherwise.

```
FLA_Error FLA_Check_hess_indices( FLA_Obj A, int ilo, int ihi );
```

Purpose: Confirm that the indices `ilo` and `ihi` are reasonable values for a reduction to upper Hessenberg operation.

Notes: Any of the following conditions will cause the function to return an `FLA_INVALID_HESSENBERG_INDICES` error value:

- $n(A) = 0$ and $ilo \neq 1$ and $ihi \neq 1$
- $ilo < 1$ or $n(A) < ilo$
- $ihi < 1$ or $n(A) < ihi$
- $ihi < ilo$

where $n(A)$ denotes the number of columns in matrix A .

Returns: `FLA_SUCCESS` if `ilo` and `ihi` are valid indices for a reduction to upper Hessenberg operation; `FLA_INVALID_HESSENBERG_INDICES` otherwise.

```
FLA_Error FLA_Check_sylv_matrix_dims( FLA_Obj A, FLA_Obj B, FLA_Obj C );
```

Purpose: Confirm that A , B , and C have conformal dimensions suitable for a triangular Sylvester equation solve of the form $AX + XB = C$ where A and B are $m \times m$ and $n \times n$, respectively, and X and C are $m \times n$.

Returns: `FLA_SUCCESS` if A , B , and C have conformal dimensions; `FLA_NONCONFORMAL_DIMENSIONS` otherwise.

```
FLA_Error FLA_Check_valid_isgn_value( FLA_Obj isgn );
```

- Purpose:** Confirm that `isgn` is either `FLA_ONE` or `FLA_MINUS_ONE`.
- Notes:** This function currently compares `isgn` against `FLA_ONE` and `FLA_MINUS_ONE` with the function `FLA_Obj_is()`, which creates a stronger constraint than just comparing the *values* contained within the objects.
- Returns:** `FLA_SUCCESS` if `isgn` is valid; `FLA_INVALID_SIDE` otherwise.

6.6.7 Other system errors

```
FLA_Error FLA_Check_pthread_create_result( int r_val );
```

- Purpose:** Confirm that `r_val`, which is assumed to have been returned from the POSIX `pthread_create()` function, does not indicate that an error has occurred.
- Returns:** `FLA_SUCCESS` if `r_val` indicates no error; `FLA_PTHREAD_CREATE_RETURNED_ERROR` otherwise.

```
FLA_Error FLA_Check_pthread_join_result( int r_val );
```

- Purpose:** Confirm that `r_val`, which is assumed to have been returned from the POSIX `pthread_join()` function, does not indicate that an error has occurred.
- Returns:** `FLA_SUCCESS` if `r_val` indicates no error; `FLA_PTHREAD_JOIN_RETURNED_ERROR` otherwise.

```
FLA_Error FLA_Check_malloc_pointer( void* ptr );
```

- Purpose:** Confirm that `ptr`, which is assumed to have been returned by a memory allocation function such as `malloc()`, is not a `NULL` pointer.
- Notes:** This routine is similar in behavior to `FLA_Check_null_pointer()`. This only difference is that this function is used specifically to check the validity of pointers returned by `malloc()`, and thus is set up to return a `malloc()`-specific error message.
- Returns:** `FLA_SUCCESS` if `ptr` is not `NULL`; `FLA_MALLOC_RETURNED_NULL_POINTER` otherwise.

```
FLA_Error FLA_Check_posix_memalign_failure( int r_val );
```

- Purpose:** Confirm that `r_val`, which is assumed to have been returned by the POSIX `posix_memalign()` function, does not indicate that an error has occurred.
- Notes:** Unlike `malloc()`, `posix_memalign()` returns an integer value to indicate success (zero) or failure (a non-zero value), and the pointer to the requested memory region is obtained from the function by providing the pointer's address as an argument, which allows `posix_memalign()` to set the pointer value directly.
- Returns:** `FLA_SUCCESS` if `r_val` is valid; `FLA_POSIX_MEMALIGN_FAILED` otherwise.

6.6.8 Misc. errors

```
FLA_Error FLA_Check_valid_pivot_type( FLA_Pivot_type ptype );
```

Purpose: Confirm that `ptype` is one of the following values defined for the `FLA_Pivot_type` type: `FLA_NATIVE_PIVOTS`, `FLA_LAPACK_PIVOTS`.

Returns: `FLA_SUCCESS` if `ptype` is valid; `FLA_INVALID_CONJ` otherwise.

```
FLA_Error FLA_Check_pivot_vector_length( FLA_Obj ipiv );
```

Purpose: Confirm that the number of rows in `ipiv` is less than or equal to the current length of the internal pivoting work buffer. If the length of `ipiv` exceeds the current length of the work buffer, then the work buffer is reallocated to the length of `ipiv`. The length of this internal buffer begins as `FLA_MAX_LU_PIVOT_LENGTH`. If a reallocation takes place, the function confirms that the reallocated pointer is not `NULL`.

Returns: `FLA_SUCCESS` if no reallocation was needed, or if a reallocation took place and the returned pointer is valid; `FLA_NULL_POINTER` otherwise.

```
FLA_Error FLA_Check_divide_by_zero( FLA_Obj alpha );
```

Purpose: Confirm that α , which is assumed to be a potential denominator in a future floating point division operation, is non-zero.

Returns: `FLA_SUCCESS` if α is non-zero; `FLA_DIVIDE_BY_ZERO` otherwise.

```
FLA_Error FLA_Check_blocksize_value( dim_t b );
```

Purpose: Confirm that b is a valid blocksize.

Notes: Since `dim_t` is a type of unsigned integer, the only invalid value that b might take on is zero. Thus, this function simply confirms that b is non-zero.

Returns: `FLA_SUCCESS` if b is valid (non-zero); `FLA_INVALID_BLOCKSIZE_VALUE` otherwise.

```
FLA_Error FLA_Check_blocksize_object( FLA_Datatype datatype, fla_blocksize_t* bp );
```

Purpose: Confirm that the blocksize field associated with `datatype` that resides within the structure pointed to by `bp` is valid.

Notes: Similar to `FLA_Check_blocksize_value()`, this function only confirms that the blocksize value is non-zero.

Returns: `FLA_SUCCESS` if the blocksize field associated with `datatype` is valid (non-zero); `FLA_INVALID_BLOCKSIZE_VALUE` otherwise.

```
FLA_Error FLA_Check_null_pointer( void* ptr );
```

Purpose: Confirm that `ptr` is not a NULL pointer.

Returns: FLA_SUCCESS if `ptr` is not NULL; FLA_NULL_POINTER otherwise.

```
FLA_Error FLA_Check_base_buffer_mismatch( FLA_Obj A, FLA_Obj B );
```

Purpose: Confirm that views *A* and *B* refer to the same underlying object.

Notes: This check is performed by comparing the addresses of the views' base objects.

Returns: FLA_SUCCESS if *A* and *B* refer to the same underlying object;
FLA_OBJECT_BASE_BUFFER_MISMATCH otherwise.

```
FLA_Error FLA_Check_num_threads( unsigned int n_threads );
```

Purpose: Confirm that `n_threads` is at least one.

Notes: Since `n_threads` is declared as an `unsigned int`, the only invalid value that `n_threads` may take on is zero.

Returns: FLA_SUCCESS if `n_threads` is at least one; FLA_ENCOUNTERED_NON_POSITIVE_NTHREADS otherwise.

Bibliography

- [1] BLAS (Basic Linear Algebra Subprograms). <http://www.netlib.org/blas/>, 2008.
- [2] GNU Octave. <http://www.gnu.org/software/octave/>, 2008.
- [3] LAPACK – Linear Algebra PACKage. <http://www.netlib.org/lapack/>, 2008.
- [4] Sage. <http://www.sagemath.org/>, 2008.
- [5] The ScaLAPACK Project. <http://www.netlib.org/scalapack/>, 2008.
- [6] R. C. Agarwal and F. G. Gustavson. Vector and parallel algorithms for Cholesky factorization on IBM 3090. In *Supercomputing '89: Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 225–233, New York, NY, USA, 1989. ACM Press.
- [7] Philip Alpatov, Greg Baker, Carter Edwards, John Gunnels, Greg Morrow, James Overfelt, Robert van de Geijn, and Yuan-Jye J. Wu. PLAPACK: Parallel linear algebra package – design overview. In *Proceedings of SC97*, 1997.
- [8] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.
- [9] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft.*, 31(1):1–26, March 2005.
- [10] Paolo Bientinesi, Brian Gunter, and Robert A. van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Soft.*, 35(1).
- [11] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Trans. Math. Soft.*, 31(1):27–59, March 2005.
- [12] Alfredo Buttari, Julien Langou, Jakub Kurzak, , and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. LAPACK Working Note 190 UT-CS-07-598, University of Tennessee, July 2007.
- [13] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. LAPACK Working Note 190 UT-CS-07-600, University of Tennessee, September 2007.
- [14] Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–125, San Diego, CA, USA, June 9-11 2007a. ACM.

- [15] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In *ACM SIGPLAN 2008 symposium on Principles and practices of parallel programming (PPoPP'08)*, pages 123–132, 2008.
- [16] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Comput. Soc. Press, 1992.
- [17] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.
- [18] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.
- [19] Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kagstrom. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.
- [20] Kazushige Goto and Robert A. van de Geijn. On reducing TLB misses in matrix multiplication. Technical Report CS-TR-02-??, Department of Computer Sciences, The University of Texas at Austin, 2002. in preparation.
- [21] Kazushige Goto and Robert A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Trans. Math. Soft.*, 34(3):12, May 2008. Article 12, 25 pages.
- [22] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.
- [23] John A. Gunnels and Robert A. van de Geijn. Formal methods for high-performance linear algebra libraries. In Ronald F. Boisvert and Ping Tak Peter Tang, editors, *The Architecture of Scientific Software*, pages 193–210. Kluwer Academic Press, 2001.
- [24] Fred G. Gustavson, Lars Karlsson, and Bo Kagstrom. Three algorithms for Cholesky factorization on distributed memory using packed storage. In *Applied Parallel Computing. State of the Art in Scientific Computing*, pages 550–559. Springer Berlin / Heidelberg, 2007.
- [25] Thierry Joffrain, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Rapid development of high-performance out-of-core solvers. In *Proceedings of PARA 2004*, number 3732 in LNCS, pages 413–422. Springer-Verlag Berlin Heidelberg, 2005.
- [26] Argonne National Laboratory. Portable, Extensible Toolkit for Scientific computation. <http://acts.nersc.gov/petsc/>, 2008.
- [27] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.
- [28] Tze Meng Low and Robert van de Geijn. An api for manipulating matrices stored by blocks. Technical Report TR-2004-15, Department of Computer Sciences, The University of Texas at Austin, May 2004.
- [29] National Instruments. LabVIEW. <http://nationalinstruments.com/labview/>, 2008.
- [30] Enrique S. Quintana, Gregorio Quintana, Xiaobai Sun, and Robert van de Geijn. A note on parallel matrix inversion. *SIAM J. Sci. Comput.*, 22(5):1762–1771, 2001.
- [31] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert van de Geijn. Solving dense linear algebra problems on platforms with multiple hardware accelerators. In *ACM SIGPLAN 2009 symposium on Principles and practices of parallel programming (PPoPP'08)*, 2009.
- [32] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Alfredo Remon, and Robert A. van de Geijn. An algorithm-by-blocks for SuperMatrix band Cholesky factorization. In *Proceedings of the 8th International Meeting on High Performance Computing for Computational Science*, June 2008.

- [33] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. van de Geijn, Field G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Soft.* accepted.
- [34] Texas Advanced Computing Center. Software and Tools. <http://www.tacc.utexas.edu/resources/software/>, 2008.
- [35] The MathWorks. MATLAB. <http://www.mathworks.com/products/matlab/>, 2008.
- [36] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.
- [37] Robert A. van de Geijn and Enrique S. Quintana-Ortí. *The Science of Programming Matrix Computations*. www.lulu.com/contents/contents/1911788/, 2008.
- [38] Field G. Van Zee, Paolo Bientinesi, Tze Meng Low, and Robert A. van de Geijn. Scalable parallelization of FLAME code via the workqueuing model. *ACM Trans. Math. Soft.*, 34(2):1–29, 2008.

Index

libflame

- features, [1](#)
- for GNU/Linux and UNIX
 - configure options, [14](#)
 - compiling, [19](#)
 - configuration, [12](#)
 - hardware support, [10](#)
 - installing, [20](#)
 - linking against, [22](#)
 - linking with liblapack2flame, [24](#)
 - running configure, [17](#)
 - software requirements, [9](#)
 - source code, [10](#)
- for Microsoft Windows
 - compiling, [34](#)
 - configuration, [31](#)
 - hardware support, [28](#)
 - installing, [35](#)
 - linking against, [37](#)
 - running configure.cmd, [33](#)
 - software requirements, [27](#)
 - source code, [28](#)
- license, [10](#), [28](#)
- obtaining, [12](#), [29](#)
- revision numbering, [11](#), [29](#)

API

- section descriptions, [50](#)

constant

- types, *see* types
- values, *see* types

control trees

- description, [199](#)
- motivation, [199](#)

developer APIs

- control trees, [202](#)
- internal back-ends, [224](#)
- locks, [195](#)
- memory allocation, [197](#)
- object creation, [198](#)
- parameter and error checking, [233](#)
- SuperMatrix, [198](#)

FLAME/C

- notational conventions, [48](#)
- object concepts, [51](#)
- terminology, [47](#)

FLAME/C functions

- FLA_Abort(), [100](#)
- FLA_Absolute_square(), [90](#)
- FLA_Accum_T_UT(), [153](#)
- FLA_Accum_T_UT_unb_external(), [191](#)
- FLA_Apply_househ2_UT(), [154](#)
- FLA_Apply_pivots(), [150](#)
- FLA_Apply_Q_blk_external(), [189](#)
- FLA_Apply_Q_unb_external(), [189](#)
- FLA_Apply_Q_UT(), [147](#)
- FLA_Asum_external(), [159](#)
- FLA_Asum(), [101](#)
- FLA_Axpy_external(), [160](#)
- FLA_Axpy_global_to_submatrix(), [60](#)
- FLA_Axpy(), [101](#)
- FLA_Axpys_external(), [161](#)
- FLA_Axpys(), [103](#)
- FLA_Axpy_submatrix_to_global(), [59](#)
- FLA_Axpyt_external(), [160](#)
- FLA_Axpyt(), [102](#)
- FLA_Chol_blk_external(), [182](#)
- FLA_Chol(), [136](#)
- FLA_Chol_unb_external(), [182](#)
- FLA_Clock(), [100](#)
- FLA_Conjugate(), [91](#)
- FLA_Conjugate_r(), [91](#)
- FLA_Cont_with_1x3_to_1x2(), [63](#)
- FLA_Cont_with_3x1_to_2x1(), [62](#)
- FLA_Cont_with_3x3_to_2x2(), [64](#)
- FLA_Copy_external(), [161](#)
- FLA_Copy_global_to_submatrix(), [58](#)
- FLA_Copy(), [103](#)
- FLA_Copy_external(), [161](#)
- FLA_Copy(), [104](#)
- FLA_Copy_submatrix_to_global(), [58](#)
- FLA_Copyt_external(), [162](#)
- FLA_Copyt(), [105](#)
- FLA_Dotc_external(), [163](#)
- FLA_Dotc(), [106](#)
- FLA_Dotcs_external(), [164](#)

FLA_Dotcs(), 108
 FLA_Dot_external(), 162
 FLA_Dot(), 106
 FLA_Dots_external(), 163
 FLA_Dots(), 107
 FLA_Dot2cs_external(), 165
 FLA_Dot2cs(), 110
 FLA_Dot2s_external(), 164
 FLA_Dot2s(), 109
 FLA_Finalize(), 52
 FLA_Form_perm_matrix(), 151
 FLA_Gemm_external(), 175
 FLA_Gemm(), 127
 FLA_Gemvc_external(), 169
 FLA_Gemvc(), 116
 FLA_Gemv_external(), 168
 FLA_Gemv(), 115
 FLA_Gerc_external(), 170
 FLA_Gerc(), 117
 FLA_Ger_external(), 169
 FLA_Ger(), 117
 FLA_Hemm_external(), 176
 FLA_Hemm(), 128
 FLA_Hemvc_external(), 170
 FLA_Hemvc(), 119
 FLA_Hemv_external(), 170
 FLA_Hemv(), 118
 FLA_Herc_external(), 171
 FLA_Herc(), 120
 FLA_Her_external(), 171
 FLA_Herk_external(), 176
 FLA_Herk(), 129
 FLA_Hermitianize(), 98
 FLA_Her(), 119
 FLA_Her2c_external(), 172
 FLA_Her2c(), 121
 FLA_Her2_external(), 171
 FLA_Her2k_external(), 176
 FLA_Her2k(), 130
 FLA_Her2(), 120
 FLA_Hess_blk_external(), 184
 FLA_Hess(), 144
 FLA_Hess_unb_external(), 184
 FLA_Househ2(), 151
 FLA_Househ2_UT(), 152
 FLA_Iamax_external(), 165
 FLA_Iamax(), 110
 FLA_Initialized(), 52
 FLA_Init(), 52
 FLA_Invert(), 92
 FLA_Inv_scalc_external(), 166
 FLA_Inv_scalc(), 111
 FLA_Inv_scal_external(), 165
 FLA_Inv_scal(), 111
 FLA_LQ_blk_external(), 187
 FLA_LQ_unb_external(), 187
 FLA_LQ_UT_Accum_T_unb_external(), 188
 FLA_LQ_UT_blk_external(), 188
 FLA_LQ_UT_create_T(), 155
 FLA_LQ_UT(), 146
 FLA_LQ_UT_recover_tau(), 156
 FLA_LQ_UT_unb_external(), 188
 FLA_LU_nopiv_blk_external(), 184
 FLA_LU_nopiv(), 137
 FLA_LU_nopiv_unb_external(), 184
 FLA_LU_piv_blk_external(), 185
 FLA_LU_piv(), 138
 FLA_LU_piv_unb_external(), 185
 FLA_Max_abs_value(), 92
 FLA_Max_elemwise_diff(), 93
 FLA_Merge_1x2(), 65
 FLA_Merge_2x1(), 65
 FLA_Merge_2x2(), 66
 FLA_Mult_add(), 93
 FLA_Negate(), 93
 FLA_Norm_inf(), 94
 FLA_Norm1(), 94
 FLA_Nrm2_external(), 166
 FLA_Nrm2(), 112
 FLA_Obj_add_to_diagonal(), 89
 FLA_Obj_attach_buffer(), 57
 FLA_Obj_buffer(), 57
 FLA_Obj_create_conf_to(), 53
 FLA_Obj_create(), 52
 FLA_Obj_create_without_buffer(), 56
 FLA_Obj_datatype(), 53
 FLA_Obj_datatype_size(), 55
 FLA_Obj_elem_size(), 55
 FLA_Obj_etype(), 55
 FLA_Obj_equals(), 88
 FLA_Obj_free(), 53
 FLA_Obj_free_without_buffer(), 56
 FLA_Obj_has_zero_dim(), 88
 FLA_Obj_is_complex(), 87
 FLA_Obj_is_conformal_to(), 88
 FLA_Obj_is_constan(), 86
 FLA_Obj_is_double_precision(), 87
 FLA_Obj_is_floating_point(), 86
 FLA_Obj_is_int(), 86
 FLA_Obj_is(), 88
 FLA_Obj_is_real(), 87
 FLA_Obj_is_scalar(), 87
 FLA_Obj_is_single_precision(), 87
 FLA_Obj_is_vector(), 88
 FLA_Obj_ldim(), 57
 FLA_Obj_length(), 54
 FLA_Obj_max_dim(), 54
 FLA_Obj_min_dim(), 54

- FLA_Obj_scale_diagonal(), 90
- FLA_Obj_set_diagonal_to_scalar(), 89
- FLA_Obj_set_to_identity(), 89
- FLA_Obj_set_to_scalar(), 89
- FLA_Obj_shift_diagonal(), 90
- FLA_Obj_show(), 55
- FLA_Obj_vector_dim(), 54
- FLA_Obj_vector_inc(), 54
- FLA_Obj_width(), 54
- FLA_Part_1x2(), 62
- FLA_Part_2x1(), 61
- FLA_Part_2x2(), 63
- FLA_Print_message(), 99
- FLA_QR_blk_external(), 186
- FLA_QR_unb_external(), 186
- FLA_QR_UT_Accum_T_unb_external(), 188
- FLA_QR_UT_blk_external(), 187
- FLA_QR_UT_create_T(), 154
- FLA_QR_UT(), 145
- FLA_QR_UT_recover_tau(), 155
- FLA_QR_UT_unb_external(), 187
- FLA_Random_herm_matrix(), 96
- FLA_Random_matrix(), 95
- FLA_Random_spd_matrix(), 97
- FLA_Random_tri_matrix(), 97
- FLA_Part_1x2_to_1x3(), 62
- FLA_Part_2x1_to_3x1(), 61
- FLA_Part_2x2_to_3x3(), 64
- FLA_Scalc_external(), 167
- FLA_Scalc(), 113
- FLA_Scal_external(), 166
- FLA_Scal(), 112
- FLA_Scalr_external(), 167
- FLA_Scalr(), 113
- FLA_Shift_pivots_to(), 150
- FLA_SPDinv_blk_external(), 183
- FLA_SPDinv(), 143
- FLA_Sqrt(), 95
- FLA_Swap_external(), 167
- FLA_Swap(), 114
- FLA_Swap_rows(), 191
- FLA_Swapt_external(), 167
- FLA_Swapt(), 114
- FLA_Sylv_blk_external(), 190
- FLA_Sylv(), 149
- FLA_Sylv_unb_external(), 190
- FLA_Symmetrize(), 98
- FLA_Symm_external(), 177
- FLA_Symm(), 131
- FLA_Symv_external(), 172
- FLA_Symv(), 121
- FLA_Syr_external(), 172
- FLA_Syrk_external(), 177
- FLA_Syrk(), 132
- FLA_Syr(), 122
- FLA_Syr2_external(), 173
- FLA_Syr2k_external(), 177
- FLA_Syr2k(), 133
- FLA_Syr2(), 122
- FLA_Transpose(), 91
- FLA_Triangularize(), 98
- FLA_Trinv_blk_external(), 182
- FLA_Trinv(), 142
- FLA_Trinv_unb_external(), 182
- FLA_Trmm_external(), 178
- FLA_Trmm(), 134
- FLA_Trmmxs_external(), 179
- FLA_Trmv_external(), 173
- FLA_Trmv(), 123
- FLA_Trmvxs_external(), 174
- FLA_Trmvxs(), 124
- FLA_Trsm_external(), 180
- FLA_Trsm(), 135
- FLA_Trsmxs_external(), 181
- FLA_Trsv_external(), 174
- FLA_Trsv(), 125
- FLA_Trsvxs_external(), 175
- FLA_Trsvxs(), 126
- FLA_Ttmm_blk_external(), 183
- FLA_Ttmm(), 141
- FLA_Ttmm_unb_external(), 183
- FLASH
 - description, 66
 - interoperability with FLAME/C, 68
 - terminology, 67
- FLASH functions
 - FLASH_Apply_Q_UT_inc_create_workspace(), 159
 - FLASH_Apply_Q_UT_inc(), 148
 - FLASH_Axpy(), 101
 - FLASH_Chol(), 136
 - FLASH_Copy_global_to_submatrix(), 76
 - FLASH_Copy_global_to_subobject(), 77
 - FLASH_Copy(), 103
 - FLASH_Copy_submatrix_to_global(), 75
 - FLASH_Copy_subobject_to_global(), 76
 - FLASH_FS_incpiv(), 140
 - FLASH_Gemm(), 127
 - FLASH_Gemv(), 115
 - FLASH_Hemm(), 128
 - FLASH_Herk(), 129
 - FLASH_Her2k(), 130
 - FLASH_LU_incpiv_create_hier_matrices(), 157
 - FLASH_LU_incpiv(), 139
 - FLASH_LU_nopiv(), 137
 - FLASH_Max_elemwise_diff(), 93
 - FLASH_Norm1(), 94
 - FLASH_Obj_attach_buffer(), 80

FLASH_Obj_blocksizes(), 82
 FLASH_Obj_create_conf_to(), 71
 FLASH_Obj_create_ext(), 70
 FLASH_Obj_create_flat_conf_to_hier(), 74
 FLASH_Obj_create_flat_copy_of_hier(), 75
 FLASH_Obj_create_hier_conf_to_flat_ext(), 73
 FLASH_Obj_create_hier_conf_to_flat(), 72
 FLASH_Obj_create_hier_copy_of_flat_ext(), 74
 FLASH_Obj_create_hier_copy_of_flat(), 73
 FLASH_Obj_create(), 69
 FLASH_Obj_create_without_buffer_ext(), 79
 FLASH_Obj_create_without_buffer(), 78
 FLASH_Obj_datatype(), 81
 FLASH_Obj_depth(), 82
 FLASH_Obj_flatten(), 77
 FLASH_Obj_free(), 71
 FLASH_Obj_free_without_buffer(), 79
 FLASH_Obj_hierarchify(), 77
 FLASH_Obj_scalar_length(), 81
 FLASH_Obj_scalar_width(), 81
 FLASH_Obj_shift_diagonal(), 90
 FLASH_QR_UT_inc_create_hier_matrices(), 158
 FLASH_QR_UT_inc(), 145
 FLASH_Random_matrix(), 95
 FLASH_Random_spd_matrix(), 97
 FLASH_SPDinv(), 143
 FLASH_Sylv(), 149
 FLASH_Symm(), 131
 FLASH_Syrk(), 132
 FLASH_Syr2k(), 133
 FLASH_Trinv(), 142
 FLASH_Trmm(), 134
 FLASH_Trsm(), 135
 FLASH_Trsv(), 125
 FLASH_Ttmm(), 141
 FLASH_Queue_get_enabled(), 83
 FLASH_Queue_get_num_threads(), 84
 FLASH_Queue_get_sorting(), 85
 FLASH_Queue_get_verbose_output(), 84
 FLASH_Queue_set_data_affinity(), 85
 FLASH_Queue_set_num_threads(), 84
 FLASH_Queue_set_sorting(), 85
 FLASH_Queue_set_verbose_output(), 84

types

constant-valued, 49

List of Contributors, iii

routines

FLAME/C, *see* FLAME/C functions
 FLASH, *see* FLASH functions
 SuperMatrix, *see* SuperMatrix functions

SuperMatrix

description, 82
 integration with FLASH, 86
 preconditions for enabling, 86

SuperMatrix functions

FLASH_Queue_begin(), 84
 FLASH_Queue_disable(), 83
 FLASH_Queue_enable(), 83
 FLASH_Queue_end(), 84
 FLASH_Queue_get_data_affinity(), 85

Appendix A

FLAME Project Related Publications

Many of following publications can be found on-line at

<http://www.cs.utexas.edu/users/flame/publications/>

A.1 Books

- B1 Robert A. van de Geijn. *Using LAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.
- B2 Robert A. van de Geijn and Enrique S. Quintana-Ortí. *The Science of Programming Matrix Computations*. www.lulu.com/contents/1911788/, 2008.

A.2 Dissertations

- D1 John A. Gunnels. A Systematic Approach to the Design and Analysis of Linear Algebra Algorithms. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-01-44. December 2001.
- D2 Paolo Bientinesi. Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms. The University of Texas at Austin, Department of Computer Sciences. August 2006.

A.3 Journal Articles

- J1 Enrique S. Quintana, Gregorio Quintana, Xiaobai Sun, and Robert van de Geijn. A note on parallel matrix inversion. *SIAM Journal on Scientific Computing*, 22(5):1762–1771, 2001.
- J2 John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422-455, December 2001.
- J3 Enrique S. Quintana-Ortí and Robert van de Geijn. Formal Derivation of Algorithms: The Triangular Sylvester Equation. *ACM Transactions on Mathematical Software*, (29) 2, June 2003.
- J4 Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert van de Geijn. The Science of Deriving Dense Linear Algebra Algorithms. *ACM Transactions on Mathematical Software*, 31(1):1-26, March 2005.
- J5 Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert van de Geijn. Representing Linear Algebra Algorithms in Code: The FLAME APIs. *ACM Transactions on Mathematical Software*, 31(1):27-59, March 2005.

- J6 Brian Gunter and Robert van de Geijn. Parallel Out-of-Core Computation and Updating of the QR Factorization. *ACM Transactions on Mathematical Software*, 32(1):60-78, March 2005.
- J7 Thierry Joffrain, Tze Meng Low, Enrique S. Quintana-Ortí, Robert van de Geijn, and Field Van Zee. On Accumulating Householder Transformations. *ACM Transactions on Mathematical Software*, 32(2):169-179, 2006.
- J8 Gregorio Quintana-Ortí and Robert van de Geijn. Improving the Performance of Reduction to Hessenberg Form. *ACM Transactions on Mathematical Software*, 32(2):180-194, 2006.
- J9 Kazushige Goto and Robert A. van de Geijn. Anatomy of a High-Performance Matrix Multiplication. *ACM Transactions on Mathematical Software*. 34(2) Article 12, 25 pages, 2008.
- J10 Field G. Van Zee, Paolo Bientinesi, Tze Meng Low, and Robert A. van de Geijn. Scalable Parallelization of FLAME Code via the Workqueuing Model. *ACM Transactions on Mathematical Software*, 34(2) Article 10, 29 pages, 2008.
- J11 Paolo Bientinesi, Brian Gunter, and Robert van de Geijn. Families of Algorithms Related to the Inversion of a Symmetric Positive Definite Matrix *ACM Transactions on Mathematical Software*, 35(1) Article 3, 22 pages, 2009.
- J12 Kazushige Goto and Robert A. van de Geijn. High-Performance Implementation of the Level-3 BLAS. *ACM Transactions on Mathematical Software*, 35(1) Article 4, 14 pages, 2009.
- J13 Enrique Quintana-Ortí and Robert van de Geijn. Updating an LU Factorization with Pivoting. *ACM Transactions on Mathematical Software*. 35(2) Article 11, 16 pages, 2009.
- J14 Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. van de Geijn, Field G. Van Zee, and Ernie Chan. Programming Matrix Algorithms-by-Blocks for Thread-Level Parallelism. *ACM Transactions on Mathematical Software*, to appear.

A.4 Conference Papers

- C1 John A. Gunnels and Robert A. van de Geijn. Formal methods for high-performance linear algebra libraries. In Ronald F. Boisvert and Ping Tak Peter Tang, editors, *The Architecture of Scientific Software*, pages 193–210. Kluwer Academic Press, 2001. Proceedings of Working Conference on Software Architectures for Scientific Computing Applications (IFIP WG 2.5 WoCo 8).
- C2 John A. Gunnels, Greg M. Henry, and Robert A. van de Geijn. A family of high-performance matrix multiplication algorithms. In Vassil N. Alexandrov, Jack J. Dongarra, Benjoe A. Juliano, René S. Renner, and C.J. Kenneth Tan, editors, *Computational Science - ICCS 2001, Part I*, Lecture Notes in Computer Science 2073, pages 51–60. Springer-Verlag, 2001.
- C3 John A. Gunnels, Daniel S. Katz, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Fault-tolerant high-performance matrix multiplication: Theory and practice. In *Proceedings of the International Conference for Dependable Systems and Networks (DSN-2001)*, pp. 47–56, July 2-4, 2001.
- C4 Paolo Bientinesi, John Gunnels, Fred Gustavson, Greg Henry, Margaret Myers, Enrique Quintana-Ortí, and Robert A. van de Geijn. Rapid Development of High-Performance Linear Algebra Libraries. *PARA 2004, LNCS 3732*, pp. 376–384, 2005.
- C5 Paolo Bientinesi, Sergey Kolos, and Robert A. van de Geijn. Automatic Derivation of Linear Algebra Algorithms with Application to Control Theory. *PARA 2004, LNCS 3732*, pp. 385–394, 2005.
- C6 Thierry Joffrain, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Rapid Development of High-Performance Out-of-Core Solvers. *PARA 2004, LNCS 3732*, pp. 413–422, 2005.

- C7 Tze Meng Low, Robert van de Geijn, and Field Van Zee. Extracting SMP Parallelism for Dense Linear Algebra Algorithms from High-Level Specifications. *Proceedings of 2005 ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, 2005.
- C8 Ernie Chan, Enrique Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix Out-of-Order Scheduling of Matrix Operations for SMP and Multi-Core Architectures. *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*, pp. 116-125. 2007.
- C9 Bryan Marker, Field Van Zee, Kazushige Goto, Gregorio Quintana-Ortí, and Robert van de Geijn. Toward Scalable Matrix Multiply on Multithreaded Architectures. *Proceedings of European Conference on Parallel and Distributed Computing (EuroPar07)*, pp. 748-757, Rennes, France, August 2007.
- C10 Ernie Chan, Field G. Van Zee, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, Robert van de Geijn. Satisfying your Dependencies with SuperMatrix. *Proceedings of IEEE Cluster Computing 2007*, pp. 91-99, Austin, Texas, September 2007.
- C11 Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Ernie Chan, Field G. Van Zee, and Robert A. van de Geijn. Scheduling of QR factorization algorithms on SMP and multi-core architectures. *Proceedings of the 16th Euromicro International Conference on Parallel, Distributed and Network-Wased Processing*, Toulouse, France, February 2008.
- C12 Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix: A Multithreaded Runtime Scheduling System for Algorithms-by-Blocks. *Proceedings of 2008 ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, p. 123-132, Salt Lake City, Utah, February 2008.
- C13 Jeff Diamond, Behnam Robatmili, Stephen W. Keckler, Robert van de Geijn, Kazushige Goto, Doug Burger. High Performance Dense Linear Algebra on a Spatially Distributed Processor. *Proceedings of 2008 ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Salt Lake City, Utah, February 2008.
- C14 Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Ernie Chan, Robert van de Geijn, and Field G. Van Zee. Design of Scalable Dense Linear Algebra Libraries for Multithreaded Architectures: the LU Factorization. *Proceedings of the Workshop on Multithreaded Architectures and Applications*, Miami, Florida, April 2008.
- C15 Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Alfredo Remon, and Robert A. van de Geijn. An Algorithm-by-Blocks for SuperMatrix Band Cholesky Factorization. *Proceedings of the 8th International Meeting on High Performance Computing for Computational Science*, Toulouse, France, June 2008.
- C16 Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, Robert van de Geijn. Solving Dense Linear Systems on Platforms with Multiple Hardware Accelerators. *Proceedings of 2009 ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, Raleigh, North Carolina, February 2009. To Appear.
- C17 Robert van de Geijn. Beautiful Parallel Code: Evolution vs. Intelligent Design. *Supercomputing 2008 Workshop on Node Level Parallelism for Large Scale Supercomputers*, Austin, Texas, November 2008.

A.5 FLAME Working Notes

- W1 John Gunnels, Greg Henry, and Robert van de Geijn. Formal Linear Algebra Methods Environment (FLAME): Overview. FLAME Working Note #1. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2000-28. November 2000.

- W2 John A. Gunnels, Daniel S. Katz, Enrique S. Quintana-Ortí, and Robert van de Geijn. Fault-Tolerant High-Performance Matrix-Matrix Multiplication, FLAME Working Note #2. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2000-34. December 2000.
- W3 John Gunnels and Robert van de Geijn. Developing Linear Algebra Algorithms: A Collection of Class Projects. FLAME Working Note #3. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2001-19. May 2001.
- W4 John Gunnels, Greg Henry, and Robert van de Geijn. High-Performance Matrix Multiplication Algorithms for Architectures with Hierarchical Memories. FLAME Working Note #4. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2001-22. June 2001.
- W5 Enrique S. Quintana-Ortí and Robert van de Geijn. Formal Derivation of Algorithms: The Triangular Sylvester Equation. FLAME Working Note #5. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2001-35. Sept. 2001.
- W6 John A. Gunnels. A Systematic Approach to the Design and Analysis of Linear Algebra Algorithms. Ph.D. Dissertation. FLAME Working Note #6, The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2001-44. Nov. 2001.
- W7 Greg M. Henry. Flexible High-Performance Matrix Multiply via a Self-Modifying Runtime Code. FLAME Working Note #7. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2001-46. Dec. 2001.
- W8 Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert van de Geijn. The Science of Deriving Dense Linear Algebra Algorithms. FLAME Working Note #8. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2002-53. Sept. 2002.
- W9 Kazushige Goto and Robert van de Geijn. On Reducing TLB Misses in Matrix Multiplication. FLAME Working Note #9. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2002-55. Nov. 2002.
- W10 Robert A. van de Geijn. Representing Linear Algebra Algorithms in Code: The FLAME API. FLAME Working Note #10. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2003-01. Jan. 2003.
- W11 Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert van de Geijn. FLAME@lab: A Farewell to Indices. FLAME Working Note #11. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2003-11. April 2003.
- W12 Tze Meng Low and Robert van de Geijn. An API for Manipulating Matrices Stored by Blocks. FLAME Working Note #12. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2004-15. May 2004.
- W13 Thierry Joffrain, Tze Meng Low, Enrique S. Quintana-Ortí, Robert van de Geijn, and Field Van Zee. On Accumulating Householder Transformations. FLAME Working Note #13. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2004-43. Oct 2004.
- W14 Gregorio Quintana-Ortí and Robert van de Geijn. Improving the Performance of Reduction to Hessenberg Form. FLAME Working Note #14. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2004-44. Oct 2004.
- W15 Tze Meng Low, Kent Milfeld, Robert van de Geijn, and Field Van Zee. Parallelizing FLAME Code with OpenMP Task Queues. FLAME Working Note #15. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2004-50.

- W16 Paolo Bientinesi, Kazushige Goto, Tze Meng Low, Enrique S. Quintana-Ortí, Robert van de Geijn, and Field Van Zee. FLAME 2005 Prospectus: Towards the Final Generation of Dense Linear Algebra Libraries. FLAME Working Note #16. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2005-15.
- W17 Paolo Bientinesi and Robert van de Geijn. Representing Dense Linear Algebra Algorithms: A Farewell to Indices. FLAME Working Note #17. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2006-10.
- W18 H. Carter Edwards and Robert A. van de Geijn. Application Interface to Parallel Dense Matrix Libraries: Just let me solve my problem! FLAME Working Note #18. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2006-15.
- W19 Paolo Bientinesi, Brian Gunter, and Robert van de Geijn. Families of Algorithms Related to the Inversion of a Symmetric Positive Definite Matrix. FLAME Working Note #19. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2006-20.
- W20 Kazushige Goto and Robert van de Geijn. High-Performance Implementation of the Level-3 BLAS. FLAME Working Note #20. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2006-23.
- W21 Enrique S. Quintana-Ortí and Robert van de Geijn. Updating an LU Factorization with Pivoting. FLAME Working Note #21. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-2006-42.
- W22 Ernie Chan, Marcel Heimlich, Avijit Purkayastha, and Robert van de Geijn. Collective Communication: Theory, Practice, and Experience. FLAME Working Note #22. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-06-44. September 26, 2006.
- W23 Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Orti, and Robert van de Geijn. SuperMatrix Out-of-Order Scheduling of Matrix Operations for SMP and Multi-Core Architectures. FLAME Working Note #23. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-06-67. December 18, 2006.
- W24 Gregorio Quintana-Ortí, Enrique S. Quintana-Orti, Ernie Chan, Field G. Van Zee, and Robert van de Geijn. Scheduling of QR factorization algorithms on SMP and multi-core architectures. FLAME Working Note #24. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-07-37. July 31, 2007.
- W25 Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Ortí, Gregorio Quintana-Orti, and Robert van de Geijn. SuperMatrix: A Multithreaded Runtime Scheduling System for Algorithms-by-Blocks. FLAME Working Note #25. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-07-41. August 22, 2007.
- W26 Gregorio Quintana-Ortí, Enrique S. Quintana-Orti, Ernie Chan, Robert van de Geijn, Field G. Van Zee. Design and Scheduling of an Algorithm-by-Blocks for LU Factorization on Multithreaded Architectures. FLAME Working Note #26. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-07-50. September 19, 2007.
- W27 Gregorio Quintana-Ortí, Enrique S. Quintana-Orti, Alfredo Remon, Robert van de Geijn. SuperMatrix for the Factorization of Band Matrices. FLAME Working Note #27. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-07-51. September 24, 2007.
- W28 Bryan Marker. On Composing Matrix Multiplication from Kernels. FLAME Working Note #28. The University of Texas at Austin, Department of Computer Sciences. Report #HR-07-32 (honors thesis). Spring 2007. 21 pages.

- W29 Gregorio Quintana-Ortí, Enrique S. Quintana-Orti, Ernie Chan, Field G. Van Zee, and Robert van de Geijn. Programming Algorithms-by-Blocks for Matrix Computations on Multithreaded Architectures. FLAME Working Note #29. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-08-04. January 15, 2008.
- W30 Sergio Barrachina, Maribel Castillo, Francisco D. Igual, Rafael Mayo, Enrique S. Quintana-Ortí. FLAG@lab: An M-script API for Linear Algebra Operations on Graphics Processors. FLAME Working Note #30. Universidad Jaume I, Depto. de Ingenieria y Ciencia de Computadores. Technical Report ICC 01-02-2008. February 14, 2008.
- W31 Maribel Castillo, Ernie Chan, Francisco D. Igual, Rafael Mayo, Enrique S. Quintana-Ortí, Gregorio Quintana-Orti, Robert van de Geijn, Field G. Van Zee. Making Programming Synonymous with Programming for Linear Algebra Libraries. FLAME Working Note #31. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-08-20. April 17, 2008.
- W32 Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Orti, Robert van de Geijn. Solving Dense Linear Algebra Problems on Platforms with Multiple Hardware Accelerators. FLAME Working Note #32. The University of Texas at Austin, Department of Computer Sciences. Technical Report TR-08-22. May 9, 2008.

A.6 Other Technical Reports

- R1 Rosa M. Badia, Jose R. Herrero, Jesus Labarta, Josep M. Perez, Enrique S. Quintana-Ortí and Gregorio Quintana-Orti. Parallelizing dense and banded linear algebra libraries using SMPs. Departament of Computer Architecture, Universitat Politècnica de Catalunya. Technical Report UPC-DAC-RR-2008-64. 2008.
- R2 Paolo Bientinesi and Robert van de Geijn. Automation in Dense Linear Algebra. Aachen Institute for Computational Engineering Science, RWTH Aachen. Technical Report AICES-2008-2. October 2008.

Appendix B

License

B.1

GNU LESSER GENERAL PUBLIC LICENSE

Version 2.1, February 1999

Copyright © 1991, 1999 Free Software Foundation, Inc. <http://fsf.org/>

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages—typically libraries—of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the “Lesser” General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a “work based on the library” and a “work that uses the library”. The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0 This License Agreement applies to any software library or other program which contains a notice placed

by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called “this License”). Each licensee is addressed as “you”.

A “library” means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The “Library”, below, refers to any such software library or work which has been distributed under these terms. A “work based on the Library” means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term “modification”).

“Source code” for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

- 1 You may copy and distribute verbatim copies of the Library’s complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

- 2 You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- (a) The modified work must itself be a software library.
- (b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
- (c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
- (d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

- 3 You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

- 4 You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

- 5 A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a “work that uses the Library”. Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a “work that uses the Library” with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a “work that uses the library”. The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a “work that uses the Library” uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

- 6 As an exception to the Sections above, you may also combine or link a “work that uses the Library” with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer’s own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library

among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

- (a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable “work that uses the Library”, as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
- (b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user’s computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.
- (c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
- (d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
- (e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the “work that uses the Library” must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

- 7 You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:
 - (a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
 - (b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.
- 8 You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
- 9 You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

- 10 Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.
- 11 If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

- 12 If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
- 13 The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.
- 14 If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

- 15 BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED

OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

- 16 IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the “copyright” line and a pointer to where the full notice is found.

```
<one line to give the library's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>
```

```
This library is free software; you can redistribute it and/or modify
it under the terms of the GNU Lesser General Public License as
published by the Free Software Foundation; either version 2.1 of the
License, or (at your option) any later version.
```

```
This library is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
Lesser General Public License for more details.
```

```
You should have received a copy of the GNU Lesser General Public
License along with libflame; if you did not receive a copy, see
http://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a “copyright disclaimer” for the library, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
library ‘Frob’ (a library for tweaking knobs) written by James Random Hacker.
```

```
<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice
```

That’s all there is to it!