Copyright © 2011 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. What’s provided ................................................................. 1
   1.2. What’s not provided ............................................................ 6
   1.3. Acknowledgments ................................................................. 7

2. Setup for GNU/Linux and UNIX
   2.1. Before obtaining libflame ...................................................... 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 ................................................................. 11
       2.1.5. Tracking source code revisions ....................................... 11
       2.1.6. If you have problems .................................................... 11
   2.2. Preparation ........................................................................... 11
   2.3. Configuration
       2.3.1. configure options ........................................................ 12
       2.3.2. Running configure ......................................................... 18
   2.4. Compiling
       2.4.1. Parallel make ................................................................ 21
   2.5. Installation ........................................................................... 21
   2.6. Linking against libflame ....................................................... 22
       2.6.1. Linking with the lapack2flame compatibility layer ................. 24

3. Setup for Microsoft Windows
   3.1. Before obtaining libflame ...................................................... 25
       3.1.1. System software requirements ........................................ 25
       3.1.2. System hardware support ............................................. 26
       3.1.3. License ........................................................................ 26
       3.1.4. Source code ................................................................. 26
       3.1.5. Tracking source code revisions ....................................... 26
       3.1.6. If you have problems .................................................... 26
   3.2. Preparation ........................................................................... 26
   3.3. Configuration
       3.3.1. IronPython ................................................................. 28
       3.3.2. Running configure.cmd ............................................... 30
   3.4. Compiling ........................................................................... 30
   3.5. Installation ........................................................................... 32
   3.6. Dynamic library generation .................................................. 32
   3.7. Linking against libflame ....................................................... 34
4. Using libflame

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1. FLAME/C examples</td>
<td>37</td>
</tr>
<tr>
<td>4.2. FLASH examples</td>
<td>39</td>
</tr>
<tr>
<td>4.3. SuperMatrix examples</td>
<td>40</td>
</tr>
</tbody>
</table>

5. User-level Application Programming Interfaces

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1. Conventions</td>
<td>45</td>
</tr>
<tr>
<td>5.1.1. General terms</td>
<td>45</td>
</tr>
<tr>
<td>5.1.2. Notation</td>
<td>46</td>
</tr>
<tr>
<td>5.1.3. Objects</td>
<td>49</td>
</tr>
<tr>
<td>5.2. FLAME/C Basics</td>
<td>50</td>
</tr>
<tr>
<td>5.2.1. Initialization and finalization</td>
<td>50</td>
</tr>
<tr>
<td>5.2.2. Object creation and destruction</td>
<td>51</td>
</tr>
<tr>
<td>5.2.3. General query functions</td>
<td>53</td>
</tr>
<tr>
<td>5.2.4. Interfacing with conventional matrix arrays</td>
<td>55</td>
</tr>
<tr>
<td>5.2.5. More query functions</td>
<td>62</td>
</tr>
<tr>
<td>5.2.6. Assignment/Update functions</td>
<td>67</td>
</tr>
<tr>
<td>5.2.7. Math-related functions</td>
<td>70</td>
</tr>
<tr>
<td>5.2.8. Miscellaneous functions</td>
<td>80</td>
</tr>
<tr>
<td>5.2.9. Advanced query routines</td>
<td>82</td>
</tr>
<tr>
<td>5.3. Managing Views</td>
<td>84</td>
</tr>
<tr>
<td>5.3.1. Vertical partitioning</td>
<td>84</td>
</tr>
<tr>
<td>5.3.2. Horizontal partitioning</td>
<td>85</td>
</tr>
<tr>
<td>5.3.3. Bidirectional partitioning</td>
<td>86</td>
</tr>
<tr>
<td>5.3.4. Merging views</td>
<td>88</td>
</tr>
<tr>
<td>5.4. FLASH</td>
<td>89</td>
</tr>
<tr>
<td>5.4.1. Motivation</td>
<td>89</td>
</tr>
<tr>
<td>5.4.2. Concepts</td>
<td>90</td>
</tr>
<tr>
<td>5.4.3. Interoperability with FLAME/C</td>
<td>91</td>
</tr>
<tr>
<td>5.4.4. Object creation and destruction</td>
<td>92</td>
</tr>
<tr>
<td>5.4.5. Interfacing with flat matrix objects</td>
<td>95</td>
</tr>
<tr>
<td>5.4.6. Interfacing with conventional matrix arrays</td>
<td>101</td>
</tr>
<tr>
<td>5.4.7. Object query functions</td>
<td>104</td>
</tr>
<tr>
<td>5.4.8. Managing Views</td>
<td>107</td>
</tr>
<tr>
<td>5.4.8.1. Vertical partitioning</td>
<td>107</td>
</tr>
<tr>
<td>5.4.8.2. Horizontal partitioning</td>
<td>108</td>
</tr>
<tr>
<td>5.4.8.3. Bidirectional partitioning</td>
<td>109</td>
</tr>
<tr>
<td>5.4.9. Utility functions</td>
<td>110</td>
</tr>
<tr>
<td>5.4.9.1. Miscellaneous functions</td>
<td>110</td>
</tr>
<tr>
<td>5.5. SuperMatrix</td>
<td>110</td>
</tr>
<tr>
<td>5.5.1. Overview</td>
<td>110</td>
</tr>
<tr>
<td>5.5.2. API</td>
<td>111</td>
</tr>
<tr>
<td>5.5.3. Integration with FLASH front-ends</td>
<td>115</td>
</tr>
<tr>
<td>5.6. Front-ends</td>
<td>115</td>
</tr>
<tr>
<td>5.6.1. BLAS operations</td>
<td>115</td>
</tr>
<tr>
<td>5.6.1.1. Level-1 BLAS</td>
<td>115</td>
</tr>
<tr>
<td>5.6.1.2. Level-2 BLAS</td>
<td>130</td>
</tr>
<tr>
<td>5.6.1.3. Level-3 BLAS</td>
<td>143</td>
</tr>
<tr>
<td>5.6.2. LAPACK operations</td>
<td>154</td>
</tr>
<tr>
<td>5.6.3. Utility functions</td>
<td>191</td>
</tr>
<tr>
<td>5.7. External wrappers</td>
<td>213</td>
</tr>
<tr>
<td>5.7.1. BLAS operations</td>
<td>214</td>
</tr>
<tr>
<td>5.7.1.1. Level-1 BLAS</td>
<td>214</td>
</tr>
<tr>
<td>5.7.1.2. Level-2 BLAS</td>
<td>223</td>
</tr>
</tbody>
</table>
5.7.1.3. Level-3 BLAS ................................................................. 231
5.7.2. LAPACK operations ......................................................... 238
5.7.3. LAPACK-related utility functions ....................................... 249
5.8. LAPACK compatibility (`lapack2flame`) ................................. 249
5.8.1. Supported routines .......................................................... 250

A. FLAME Project Related Publications ........................................... 259
   A.1. Books ........................................................................... 259
   A.2. Dissertations ................................................................. 259
   A.3. Journal Articles ............................................................ 259
   A.4. Conference Papers ........................................................ 260
   A.5. FLAME Working Notes .................................................. 262
   A.6. Other Technical Reports .................................................. 265

B. License ................................................................................. 267
   B.1. BSD 3-clause license ....................................................... 267
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 [9, 28, 19, 19, 18].

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 [31, 24, 23, 38]. 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 [23, 10, 38], which guarantees that the resulting algorithms are correct.

Object-based abstractions and API. The BLAS, LAPACK, and ScaLAPACK [17] 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 [12]. 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.

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

<table>
<thead>
<tr>
<th>Algorithm: $A := \text{Chol}_2\text{.blk.var2}(A)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Partition</strong> $A \rightarrow \begin{pmatrix} A_{TL} &amp; A_{TR} \ A_{BL} &amp; A_{BR} \end{pmatrix}$</td>
</tr>
<tr>
<td>where $A_{TL}$ is $0 \times 0$</td>
</tr>
<tr>
<td><strong>while</strong> $m(A_{TL}) &lt; m(A)$ do</td>
</tr>
<tr>
<td>Determine block size $b$</td>
</tr>
<tr>
<td>Repartition</td>
</tr>
<tr>
<td>$\begin{pmatrix} A_{TL} &amp; A_{TR} \ A_{BL} &amp; A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} A_{00} &amp; A_{01} &amp; A_{02} \ A_{10} &amp; A_{11} &amp; A_{12} \ A_{20} &amp; A_{21} &amp; A_{22} \end{pmatrix}$</td>
</tr>
<tr>
<td>where $A_{11}$ is $b \times b$</td>
</tr>
<tr>
<td>$A_{11} := A_{11} - A_{10}A_{11}^T$ (SYRK)</td>
</tr>
<tr>
<td>$A_{21} := A_{21} - A_{20}A_{11}^T$ (GEMM)</td>
</tr>
<tr>
<td>$A_{11} := \text{Chol}(A_{11})$</td>
</tr>
<tr>
<td>$A_{21} := A_{21}A_{11}^T$ (TRSM)</td>
</tr>
<tr>
<td><strong>Continue with</strong></td>
</tr>
<tr>
<td>$\begin{pmatrix} A_{TL} &amp; A_{TR} \ A_{BL} &amp; A_{BR} \end{pmatrix} \leftarrow \begin{pmatrix} A_{00} &amp; A_{01} &amp; A_{02} \ A_{10} &amp; A_{11} &amp; A_{12} \ A_{20} &amp; A_{21} &amp; A_{22} \end{pmatrix}$</td>
</tr>
<tr>
<td><strong>endwhile</strong></td>
</tr>
</tbody>
</table>

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.

The 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 [11]. 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 [11]. 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
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 24 core Intel “Dunnington” system. Theoretical peak system performance is 255 GFLOPS. libflame uses variant 3 while LAPACK uses variant 2. For non-SuperMatrix experiments, MKL was invoked in multithreaded mode. For SuperMatrix experiments, MKL parallelism was disabled. Blocksizes were tuned individually for each problem size tested.
1.1. What’s provided

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) [15, 16, 32, 34]. 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 [6, 20, 25]. 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 [34, 29].

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 relatively new and therefore should be considered experimental. However, we feel libflame for Windows is usable for many in our audience. We invite interested users to try the software and, of course, we welcome feedback to help improve our Windows support, and libflame in general.

Independence from Fortran and LAPACK. The libflame development team is pleased to offer a high-performance linear algebra solution that is 100% Fortran-free. libflame is a C-only implementation and does not depend on any external Fortran libraries such as LAPACK.³ That said, we happily provide an optional backward compatibility layer, lapack2flame, that maps LAPACK routine invocations to their

²Some of the lead developers of LAPACK have independently investigated these ideas as part of a spin-off project, PLASMA [13, 14, 7].

³In fact, besides the BLAS and the standard C libraries, libflame does not have any external dependencies—not even f2c.
corresponding native C implementations in libflame. This allows legacy applications to start taking advantage of libflame with virtually no changes to their source code. Furthermore, we understand that some users wish to leverage highly-optimized implementations that conform to the LAPACK interface, such as Intel’s Math Kernel Library (MKL). As such, we allow those users to configure libflame such that their external LAPACK implementation is called for the small, performance-sensitive unblocked subproblems that arise within libflame’s blocked algorithms and algorithms-by-blocks.

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 [8, 37], 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 [26]. While this prototype extension is not distributed with libflame, we believe that FLASH with its the 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 [33].

**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 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.

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 [1]. 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 [3]. 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 [5].

- **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 [27]. 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 [36]. MATLAB is licensed as a commerical 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 [30].

- **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 [2].

- **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 [4].

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

### 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 distributions\(^1\) 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 C compiler.** Most of libflame is written in C, and therefore building libflame on for GNU/Linux or UNIX requires a C (or C++) compiler.\(^3\) The GNU gcc, Intel icc, IBM xlc, and Pathscale pathcc compilers are explicitly supported. A generic compiler named cc should work too. Later subsections in this chapter describe how the user may specify a preferred C compiler.

- **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 \texttt{dgemm()} and \texttt{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 [21, 22, 35]. GotoBLAS provides excellent performance on a wide variety of mainstream architectures. Other BLAS libraries, \(^4\)

\(^1\)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.

\(^2\)On 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.

\(^3\) The libflame configuration script will probe for and query a Fortran compiler in order to detect the name mangling conventions necessary for C and Fortran functions to call each other. If your build fails because a Fortran compiler was not present at configure-time, please contact the libflame developers.
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.** Most of the computationally-intensive operations implemented in
  libflame are expressed as blocked algorithms or algorithms-by-blocks, both of which cast some of their
  computation in terms of smaller subproblems. libflame provides optimized, low-overhead unblocked
  functions to perform these small matrix computations. However, for performance reasons, some users
  might want these computations to be performed instead by an external implementation of LAPACK.
  See Section 2.3.1 for more information on making use of this optional 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.\(^4\)

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 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 compiler explicitly at configure-time. For more information
on this and related topics, refer to Section 2.3.1.

2.1.3 **License**

libflame is intellectual property of The University of Texas at Austin. Unless you or your organization
has made other arrangements, libflame is provided as free software under the 3-clause BSD license. Please
refer to Appendix B for the full text of this license.

\(^4\)Whether 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.
## 2.2. Preparation

<table>
<thead>
<tr>
<th>Build system type</th>
<th>C compiler search order</th>
</tr>
</thead>
<tbody>
<tr>
<td>i386-<em>-</em></td>
<td>gcc icc cc</td>
</tr>
<tr>
<td>x86_64-<em>-</em></td>
<td>gcc icc pathcc cc</td>
</tr>
<tr>
<td>ia64-<em>-</em></td>
<td>icc gcc cc</td>
</tr>
<tr>
<td>powerpc*-ibm-aix*</td>
<td>xlc</td>
</tr>
<tr>
<td>powerpc64-**-linux-gnu</td>
<td>gcc xlc</td>
</tr>
<tr>
<td>All others</td>
<td>gcc cc</td>
</tr>
</tbody>
</table>

Table 2.1: The list of compilers that are searched for as a function of build system type, which consists of 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.

### 2.1.4 Source code

The `libflame` source code is available via the web at [github.com](http://www.github.com/flame/libflame/): 

```
http://www.github.com/flame/libflame/
```

We encourage users to download a copy of `libflame` via the `git clone` command, rather than a gzipped-tarball. That way, you can update your copy of `libflame` (via `git pull`) without having to download an entirely new copy.

### 2.1.5 Tracking source code revisions

Each copy of `libflame` is named according to a human-designated version string, followed by a “patch” number that corresponds to the number of `git` commits applied since that version string (or tag) was applied. For example, version 5.1.0-15 is 15 commits newer than the commit to which the “5.1.0” tag was first attached. Each version also has a unique SHA-1 hash, which is used when identifying versions with `git`.

### 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 post to our mailing list at:

```
http://groups.google.com/group/libflame-discuss
```

A `libflame` developer (or perhaps a fellow user!) will try to get back in touch with you as soon as possible.

## 2.2 Preparation

Download a `git` clone of `libflame` from the github website.

```
> git clone https://github.com/flame/libflame.git
> ls
libflame
```

Change into the `libflame` directory:

```
> cd libflame
```
The top-level directory of the source tree should look something like this:

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

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

## 2.3 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`. The options that are supported are listed explicitly and described in the next subsection.

### 2.3.1 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

---

5 This is due to boilerplate content that autoconf inserts into the configure script regardless of whether it is desired.
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.
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 the library file. By default, `configure` instructs `make` to supress 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-static-build
Create `libflame` as a static library archive. Enabled by default.

--enable-dynamic-build
Create `libflame` as a dynamically-linked shared library. Linking an executable to a shared library has the advantage that only one copy of the library code will ever be loaded into memory. 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-autodetect-f77-ldflags
Enable automatic detection of any linker flags that may be needed to link against Fortran code. These flags are useful to know about when, for example, linking `libflame` to a BLAS library that was compiled with the system’s Fortran compiler. You may need to disable this option, along with autodetection of Fortran name-mangling, if the environment’s Fortran compiler is missing or broken. Enabled by default.

--enable-autodetect-f77-name-mangling
Enable automatic detection of the name-mangling necessary to invoke Fortran routines from C, and C-compiled routines from Fortran. Disabling this option causes a pre-defined default to be used, which may not work in some environments. You may need to disable this option, along with autodetection of Fortran linker flags, if the environment’s Fortran compiler is missing or broken. 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-built-in-blas
Enable code that provides a built-in implementation of the BLAS. Note that some routines may not be optimized yet. Disabled by default.

--enable-lapack2flame
Compile and build into libflame a compatibility layer that maps LAPACK invocations to their corresponding FLAME/C implementations. Note that erroneous input parameters are reported according to libflame conventions, NOT LAPACK conventions. That is, if libflame error checking is disabled, no error checking is performed, and if erroneous input parameters are detected, the library aborts. Also, if this option is enabled, then external-lapack-for-subproblems MUST be disabled. *Disabled by default.*

--enable-external-lapack-for-subproblems
Enable code that causes most of the computationally-intensive functions within libflame to compute their smallest subproblems by invoking a corresponding (usually unblocked) LAPACK routine. Note that if this option is enabled, lapack2flame MUST be disabled. Also, if this option is enabled, then external-lapack-interfaces MUST also be enabled. Enabling this option is useful when a libflame user wishes to leverage an optimized external implementation of LAPACK to speed up the subproblems that arise within libflame's blocked algorithms and algorithms-by-blocks. *Disabled by default.*

--enable-external-lapack-interfaces
Enable code that allows the user to interface with an external LAPACK implementation via object-based FLAME-like functions. Note that if this option is enabled, an LAPACK library will be required at link-time. *Disabled by default.*

--enable-blas3-front-end-cntl-trees
Enable code that uses control trees\(^6\) 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 pthreads 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-gpu
Enable code that takes advantage of graphical processing units (GPUs) when performing certain computations. If enabled, SuperMatrix must also be enabled via --enable-supermatrix. Note that this option is experimental. *Disabled by default.*

--enable-vector-intrinsics=type
Enable highly-optimized code that relies upon vector intrinsics to specify certain operations at a very low level. Valid values for type are sse and none. Specifying none is the same as disabling the option. *Disabled by default.*

--enable-memory-alignment=N

---

\(^6\) 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 ??.
Enable code that aligns dynamically allocated memory regions at \( N \)-byte boundaries. Specifically, this option configures libflame to use \emph{posix_memalign()} instead of \emph{malloc()} for all internal memory allocation. Note: \( N \) must be a power of two and multiple of \emph{sizeof(void*)}, which is usually 4 on 32-bit architectures and 8 on 64-bit architectures. \emph{Disabled by default.}

\textbf{--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 row or column begins at an aligned address. \emph{Disabled by default.}

\textbf{--enable-optimizations}

Employ traditional compiler optimizations when compiling C source code. \emph{Enabled by default.}

\textbf{--enable-warnings}

Use the appropriate flag(s) to request warnings when compiling C source code. \emph{Enabled by default.}

\textbf{--enable-debug}

Use the appropriate debug flag (usually \texttt{-g}) when compiling C source code. \emph{Disabled by default.}

\textbf{--enable-profiling}

Use the appropriate profiling flag (usually \texttt{-pg}) when compiling C source code. \emph{Disabled by default.}

\textbf{--enable-internal-error-checking=level}

Enable various internal runtime checks of function parameters and object properties to prevent functions from executing with unexpected values. Note that this option determines the default level, which may be changed at runtime (via \emph{FLA\_Check\_error\_level\_set()}). Valid values for \emph{level} are \emph{full}, \emph{minimal}, and \emph{none}. \emph{Enabled by default to full}. 

\textbf{--enable-memory-leak-counter}

Enable code that keeps track of the balance between calls to \emph{FLA\_malloc()} and \emph{FLA\_free()}. If enabled, the counter value is output to standard error upon calling \emph{FLA\_Finalize()}. Note that this option determines the default status, which may be changed at runtime (via \emph{FLA\_Memory\_leak\_counter\_set()}). \emph{Disabled by default.}

\textbf{--enable-blis-use-of-fla-malloc}

Enable code that defines \emph{bli\_malloc()} in terms of \emph{FLA\_malloc()}. One benefit of this is that BLIS memory allocations can be tracked, along with other libflame memory allocations, if the memory leak counter is enabled. A second benefit is that BLIS memory allocations can be aligned to boundaries if libflame memory alignment is enabled. Note this option may only be set at configure-time. \emph{Enabled by default.}

\textbf{--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. \emph{Disabled by default.}

\textbf{--enable-cblas-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. \emph{Disabled by default.}
Enable code that interfaces \texttt{libflame}'s external wrapper routines to the BLAS via the CBLAS rather than the traditional Fortran-77 API. \textit{Disabled by default.}

\begin{itemize}
  \item [--enable-default-m-blocksize=mb]
  \item [--enable-default-k-blocksize=kb]
  \item [--enable-default-n-blocksize=nb]
\end{itemize}

Enable user-defined block sizes in the \texttt{m}, \texttt{k}, and \texttt{n} dimensions. These options may be used to define the block sizes that will be returned from block size query functions when GotoBLAS interfaces are disabled. Note that these options have no effect when GotoBLAS interfaces are enabled. \textit{Disabled by default.}

\begin{itemize}
  \item [--enable-portable-timer]
\end{itemize}

Define the \texttt{FLA\_Clock()} timer function using \texttt{clock\_gettime()}. If that function is not available, then \texttt{gettimeofday()} is used. If neither function is available, \texttt{FLA\_Clock()} will return a static value. \textit{By default, a portable timer is used (if it exists).}

A few command line feature options are supported by \texttt{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:

\begin{itemize}
  \item [--enable-windows-build]
  Enable code that is needed for a Windows-friendly build of \texttt{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 \texttt{configure} to begin with.) \textit{Disabled by default.}
  \item [--enable-scc]
  Enable code that takes advantage of the SCC multicore architecture. When using this option, enabling SuperMatrix is recommended, though not strictly required. Note that this option is experimental. \textit{Disabled by default.}
  \item [--enable-tidsp]
  Enable code required for \texttt{libflame} to run under Texas Instruments' DSP. Note that this option is experimental. \textit{Disabled by default.}
\end{itemize}

The supported command line tool options are:

\begin{itemize}
  \item [--with-cc=cc]
  Search for and use a C compiler named \texttt{cc}. If \texttt{cc} is not found, then use the first compiler found from the default search list for the detected build architecture.
  \item [--with-extra-cflags=flags]
  When compiling C code, use the flags in \texttt{flags} in addition to the flags that \texttt{configure} would normally use. This is useful when the user wants some extra flags passed to the compiler but does not want to manually set the \texttt{CFLAGS} environment variable and thus override all of the default compiler flags. Note: Be sure to use quotations if the \texttt{flags} string contains spaces.
  \item [--with-ar=ar]
  Search for and use a library archiver named \texttt{ar}. If \texttt{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 \texttt{ar}.
  \item [--with-ranlib=ranlib]
2. Setup for GNU/Linux and UNIX

<table>
<thead>
<tr>
<th>Variable</th>
<th>Command line option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>--with-cc=cc</td>
<td>Use CC as the C compiler.</td>
</tr>
<tr>
<td>CFLAGS</td>
<td>none</td>
<td>The command line flags to use with the C compiler. Note: Do not set this</td>
</tr>
<tr>
<td></td>
<td></td>
<td>variable unless you know what you are doing! It overrides C compiler flags</td>
</tr>
<tr>
<td></td>
<td></td>
<td>that are set by configure to correspond to libflame feature options.</td>
</tr>
<tr>
<td>EXTRA_CFLAGS</td>
<td>--with-extra-cflags=flags</td>
<td>Use flags in addition to the C compiler flags that are set internally by</td>
</tr>
<tr>
<td></td>
<td></td>
<td>configure. Most users that need extra flags passed in to the C compiler</td>
</tr>
<tr>
<td></td>
<td></td>
<td>will want to set EXTRA_CFLAGS instead of CFLAGS.</td>
</tr>
<tr>
<td>AR</td>
<td>--with-ar=ar</td>
<td>Use AR to create and fill static library archives.</td>
</tr>
<tr>
<td>RANLIB</td>
<td>--with-ranlib=ranlib</td>
<td>Use RANLIB to generate the index to the static library archive. Note: In</td>
</tr>
<tr>
<td></td>
<td></td>
<td>modern environments, the functionality of ranlib has been superceded by ar;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>GNU ranlib is equivalent to running ar -s.</td>
</tr>
<tr>
<td>FIND</td>
<td>none</td>
<td>The find utility is needed by the clean targets defined in the Makefile.</td>
</tr>
<tr>
<td>XARGS</td>
<td>none</td>
<td>xargs, like find, is needed by the clean targets defined in the Makefile.</td>
</tr>
</tbody>
</table>

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.

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.

### 2.3.2 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 \texttt{libflame} with slightly different options by simply editing \texttt{run-configure.sh} and then re-running the script.

The primary purpose of running \texttt{configure} is to provide \texttt{make} with some of the information it needs in order to begin compiling \texttt{libflame}. As \texttt{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:

```plaintext
> ./run-conf/run-configure.sh
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
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 gcc, 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
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
```

\texttt{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 \texttt{configure} process with output that looks something like:

```plaintext
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
```

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

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

There is one section of the configuration summary that you should pay special attention to. If autodetection of Fortran name-mangling was enabled, at the end of the summary there will be output that looks like:

```plaintext
Autodetect Fortran linker flags............... : yes
  Fortran linker flags...................... : -L/usr/opt/gcc-4.2.2/lib/gcc/i686-pc-linux-gnu/4.2.2
  -L/usr/opt/gcc-4.2.2/lib/gcc/i686-pc-linux-gnu/4.2.2/../../.. -lgfortranbegin -lgfortran -lm
Autodetect Fortran name-mangling............. : yes
  Unmangled name........................... : foobar
  Mangled name............................. : foobar
```
**2. Setup for GNU/Linux and UNIX**

Compile with extra C compiler flags ............... : no
libflame install directory prefix ............... : /u/field/flame

Compile with extra C compiler flags ............... : no
libflame install directory prefix ............... : /u/field/flame

Configuration complete!

NOTE: Autodetection of Fortran linker flags was enabled. The configure script thinks that the flags listed above are necessary to successfully link a program to Fortran object code. If your program uses any Fortran libraries, you will probably need to link with these flags.

The purpose of this note is to inform the user of Fortran linker flags\(^8\) that may be needed in order to successfully link libflame and your application against Fortran code, potentially including the BLAS or LAPACK libraries. Sometimes, these flags are not necessary, but it is safer to always use them. Please see Section 2.6 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/i686-unknown-linux-gnu/post-configure.sh
-rwxr--r-- 1 field dept 5861 Nov 14 13:35 config/i686-pc-linux-gnu/post-configure.sh
```

### 2.4 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
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.

---

\(^8\) The flags shown were detected when libflame was configured to use Intel compilers in an i686-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 accommodate 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.
2.5. Installation

<table>
<thead>
<tr>
<th>Target</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>Invoke the libs target.</td>
</tr>
<tr>
<td>check</td>
<td>Verify that configure has been run.</td>
</tr>
<tr>
<td>libs</td>
<td>Invoke the check target and then build the libflame and liblapack2flame library archives.</td>
</tr>
<tr>
<td>install</td>
<td>Invoke the libs target and then copy the library archives and header files to their respective lib and include subdirectories of the install prefix directory, which is $HOME/flame by default. Also create symbolic links (with ln -s) to the library files and include subdirectory so that the symbolic links do not contain the architecture or version strings.</td>
</tr>
<tr>
<td>install-without-symlinks</td>
<td>Similar to install except that no symbolic links are installed.</td>
</tr>
<tr>
<td>clean</td>
<td>Remove all object files and previously-built library archives from the obj and lib directories for the current build system only. (Recall that the current build system is written to config.sys.type.) Build products built for other systems will not be touched.</td>
</tr>
<tr>
<td>cleanmk</td>
<td>Remove all makefile fragments from the source tree.</td>
</tr>
<tr>
<td>distclean</td>
<td>Invoke clean, cleanmk, and then remove all other intermediate build files and directories that are created either by configure or one of the autotools such as autoconf.</td>
</tr>
<tr>
<td>send-thanks</td>
<td>Use mail to send the libflame developers a short thank-you message.</td>
</tr>
</tbody>
</table>

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.

2.4.1 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.

```shell
> 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.

2.5 Installation

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

```shell
> ls -l lib/i686-unknown-linux-gnu/
total 33872
-rw-r--r-- 1 field dept 22548036 Nov 14 13:45 libflame.a
-rwxr-xr-x 1 field dept 12060827 Nov 14 13:45 libflame.so
```

In this example, libflame was built for an i686 system with a build system type of i686-unknown-linux-gnu, and so the library files reside in the directory lib/i686-unknown-linux-gnu.

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

Also notice that the `install` target automatically creates abbreviated symbolic links to the actual library files so that you may simply refer to `libflame.a` or `libflame.so` when linking your application. A similarly shortened symbolic link is created for the include directory as well. If you do not wish to create and install these symbolic links along with `libflame`, you may instead invoke `install-without-symlinks`.

Now that `libflame` has been installed, you are ready to use it!

### 2.6 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` target (and not the `install-without-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 2.3.2. 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 2.3.2.

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

```makefile
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:

```makefile
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)
LDFLAGS := -L/opt/intel/fc/em64t/10.0.026/lib
LDFLAGS += -L/usr/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

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

$(MYAPP): $(MYAPP_OBJS)
  $(LINKER) $(MYAPP_OBJS) $(LDFLAGS) $(FLAME_LIB) $(LAPACK_LIB) $(BLAS_LIB) -o $(MYAPP_BIN)
```
2. Setup for GNU/Linux and UNIX

```bash
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 `LDFLAGS` 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.

### 2.6.1 Linking with the `lapack2flame` compatibility layer

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 by configuring `libflame` to include the `lapack2flame` compatibility layer. When this option is provided at configure-time, `libflame` is built to include interfaces that map conventional LAPACK routine invocations to native FLAME/C function calls.

For more information about the routines supported by `lapack2flame`, refer to Section 5.8.
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 tested libflame under Windows XP and Windows 7. We have not yet been able to test the software under Windows Vista, though we suspect it would compile, link, 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.

- **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 environment.

- **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. When configured accordingly, libflame also provides direct access to BLAS routines by way of wrappers that map object-based APIs to traditional Fortran-77 routine interfaces. Any library that adheres to the BLAS interface should work fine. On Windows, libflame developers often use Intel’s MKL, which performs well and is included with the Intel C/C++ compiler suite.

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.** Most of the computationally-intensive operations implemented in libflame are expressed as blocked algorithms or algorithms-by-blocks, both of which cast some of their
computation in terms of smaller subproblems. \texttt{libflame} provides optimized, low-overhead unblocked functions to perform these small matrix computations. However, for performance reasons, some users might want these computations to be performed instead by an external implementation of LAPACK. See Section 2.3.1 for more information on making use of this optional feature.

- An OpenMP-aware C compiler. \texttt{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 \texttt{libflame} is configured to use an OpenMP-aware compiler.

3.1.2 System hardware support

Since \texttt{libflame} for Windows is still relatively 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

\texttt{libflame} is intellectual property of The University of Texas at Austin. Unless you or your organization has made other arrangements, \texttt{libflame} is provided as free software under the 3-clause BSD license. Please refer to Appendix B for the full text of this license.

3.1.4 Source code

The \texttt{libflame} source code is available via the web at \url{github.com}:

\begin{verbatim}
http://www.github.com/flame/libflame/
\end{verbatim}

We encourage users to download a copy of \texttt{libflame} via the \texttt{git clone} command, rather than a gzipped-tarball. That way, you can update your copy of \texttt{libflame} (via \texttt{git pull} without having to download an entirely new copy.

3.1.5 Tracking source code revisions

Each copy of \texttt{libflame} is named according to a human-designated version string, followed by a “patch” number that corresponds to the number of \texttt{git} commits applied since that version string (or tag) was applied. For example, version 5.1.0-15 is 15 commits newer than the commit to which the “5.1.0” tag was first attached. Each version also has a unique SHA-1 hash, which is used when identifying versions with \texttt{git}.

3.1.6 If you have problems

If you encounter trouble while trying to build and install \texttt{libflame}, if you think you’ve found a bug, or if you have a question not answered in this document, we invite you to post to our mailing list at:

\begin{verbatim}
http://groups.google.com/group/libflame-discuss
\end{verbatim}

A \texttt{libflame} developer (or perhaps a fellow user!) will try to get back in touch with you as soon as possible.

3.2 Preparation

Download the \texttt{.zip} package from the website and then unzip the source code.
3.2. Preparation

C:\field\temp>dir
Volume in drive C has no label.
Volume Serial Number is B4E3-D9FC

Directory of C:\field\temp

12/01/2009 01:03 PM <DIR> .
12/01/2009 01:03 PM <DIR> ..
12/01/2009 01:04 PM <DIR> libflame-5.1.0
12/01/2009 01:02 PM 5,324,397 libflame-5.1.0.zip
1 File(s) 5,324,397 bytes
3 Dir(s) 85,294,235,648 bytes free

Change into the libflame-5.1.0 directory:

C:\field\temp>cd libflame-5.1.0

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

C:\field\temp\libflame-5.1.0>dir
Volume in drive C has no label.
Volume Serial Number is B4E3-D9FC

Directory of C:\field\temp\libflame-5.1.0

12/01/2009 01:04 PM <DIR> .
12/01/2009 01:04 PM <DIR> ..
12/01/2009 01:03 PM 893 AUTHORS
12/01/2009 01:03 PM 91 bootstrap
12/01/2009 01:03 PM <DIR> build
12/01/2009 01:03 PM 5,836 CHANGELOG
12/01/2009 01:03 PM 293,036 configure
12/01/2009 01:03 PM 13,853 configure.ac
12/01/2009 01:03 PM 2,329 CONTRIBUTORS
12/01/2009 01:03 PM <DIR> docs
12/01/2009 01:03 PM 50,468 Doxyfile
12/01/2009 01:03 PM <DIR> examples
12/01/2009 01:03 PM 9,478 INSTALL
12/01/2009 01:03 PM 26,420 LICENSE
12/01/2009 01:03 PM 12,983 Makefile
12/01/2009 01:03 PM 1,216 README
12/01/2009 01:03 PM 5 revision
12/01/2009 01:03 PM <DIR> run-conf
12/01/2009 01:04 PM <DIR> src
12/01/2009 01:04 PM <DIR> test
12/01/2009 01:04 PM <DIR> tmp
12/01/2009 01:04 PM <DIR> windows

12 File(s) 416,608 bytes
10 Dir(s) 85,294,235,648 bytes free

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

C:\field\temp\libflame-5.1.0>cd windows
C:\field\temp\libflame-5.1.0\windows>dir
Volume in drive C has no label.
Volume Serial Number is B4E3-D9FC

1Table 2.2 describes the files present in the top-level GNU/Linux and UNIX build directory.
3. Setup for Microsoft Windows

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

3.3 Configuration

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

The next three sections describe how to build libflame as a static library. Please see Section 3.6 for supplemental instructions on building a dynamically-linked library.

The bulk of the configuration options are specified in the file build\FLA_config.h.² 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 2.3.1.

There is a single configuration option that must be set in the build\defs.mk:

• 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.

3.3.1 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.

²Unlike in the GNU/Linux build system, the user must set these options manually. We apologize for the inconvenience.
### 3.3. Configuration

<table>
<thead>
<tr>
<th>File</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Makefile</td>
<td>persistent</td>
<td>The top-level makefile for compiling <em>libflame</em> under Microsoft Windows. This makefile is written for Microsoft’s Program Maintenance Utility, <code>nmake</code>. It may only be run after <code>configure.cmd</code> is run.</td>
</tr>
<tr>
<td>build</td>
<td>persistent</td>
<td>This directory contains auxiliary build system files and scripts. These files are probably only of interest to developers of <em>libflame</em>, and so most users may safely ignore this directory.</td>
</tr>
<tr>
<td>config</td>
<td>build</td>
<td>A directory containing intermediate build files whose contents depend on how <em>libflame</em> was configured.</td>
</tr>
<tr>
<td>configure.cmd</td>
<td>persistent</td>
<td>The script used to prepare the Windows build environment for compiling <em>libflame</em>. <code>configure.cmd</code> has multiple required arguments, which are explained when <code>configure.cmd</code> is run with no arguments (or the wrong number of arguments).</td>
</tr>
<tr>
<td>dll</td>
<td>build</td>
<td>A directory containing the dynamic library files created after compilation.</td>
</tr>
<tr>
<td>gendll.cmd</td>
<td>persistent</td>
<td>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 <code>nmake</code> and so normal users should never need to invoke it manually.</td>
</tr>
<tr>
<td>include</td>
<td>build</td>
<td>A temporary directory containing copies of the source header files gathered from the top-level source directory tree.</td>
</tr>
<tr>
<td>lib</td>
<td>build</td>
<td>A directory containing the static library file created after compilation.</td>
</tr>
<tr>
<td>mmake-cc.log</td>
<td>build</td>
<td>A file capturing the standard output of the C compiler.</td>
</tr>
<tr>
<td>mmake-fc.log</td>
<td>build</td>
<td>A file capturing the standard output of the Fortran compiler.</td>
</tr>
<tr>
<td>mmake-copy.log</td>
<td>build</td>
<td>A file capturing the standard output of the copy command line utility.</td>
</tr>
<tr>
<td>linkargs.txt</td>
<td>persistent</td>
<td>A list of compiler arguments used by <code>gendll.cmd</code> when building a dynamically-linked library (DLL). This list includes link options, libraries, and library paths. For more details on what this file should contain and in what ways it should be customized by the user, refer to Section 3.3.2.</td>
</tr>
<tr>
<td>linkargs64.txt</td>
<td>persistent</td>
<td>Similar to <code>linkargs.txt</code>, but for use when generating 64-bit object code. To use this file to generate a 64-bit DLL, simply rename this file to <code>linkargs.txt</code> before invoking the <code>dll</code> target. The user may also use the file contents as a reference when determining the compiler arguments needed to link an application against a static 64-bit build of <em>libflame</em>.</td>
</tr>
<tr>
<td>obj</td>
<td>build</td>
<td>A directory containing the object files created during compilation.</td>
</tr>
<tr>
<td>revision</td>
<td>build/persistent</td>
<td>A file containing the subversion revision number of the source code.</td>
</tr>
<tr>
<td>src</td>
<td>build</td>
<td>A temporary directory containing copies of the source code files gathered from the top-level source directory tree.</td>
</tr>
</tbody>
</table>

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 “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 `nmake` target `distclean`. 
3. Setup for Microsoft Windows

<table>
<thead>
<tr>
<th>Argument</th>
<th>Accepted Values</th>
<th>Consequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>architecture string</td>
<td>any string</td>
<td>This string is inserted into the filename of the final library. It has no effect on how libflame is built.</td>
</tr>
<tr>
<td>build type</td>
<td>debug</td>
<td>Enables debugging symbols and disables all compiler optimizations.</td>
</tr>
<tr>
<td></td>
<td>release</td>
<td>Disables debugging symbols and enables maximum compiler optimizations.</td>
</tr>
<tr>
<td>C compiler string</td>
<td>icl</td>
<td>Compile C source code with the Intel C/C++ compiler, icl. Also, if a DLL is built, use icl to perform the linking.</td>
</tr>
<tr>
<td></td>
<td>cl</td>
<td>Compile C source code with the Microsoft C/C++ compiler, cl. Also, if a DLL is built, use cl to perform the linking.</td>
</tr>
</tbody>
</table>

Table 3.2: The arguments expected by configure.cmd.

3.3.2 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-5.1.0\windows>\configure.cmd x64 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 5.1.0". 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.
```

Here, we invoked the configure.cmd script with the “x64” architecture string, requested that debugging be enabled (and optimizations be disabled), and specified icl as the C compiler to use for compilation. The architecture string will be inserted into the library filename to help the user distinguish between any other subsequent builds.

The configure.cmd script first checks whether the revision file needs updating. 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.

Before proceeding to run nmake, 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 which allows the user to invoke the icl compiler command from the Windows shell prompt. Note that this step, wherein the user executes any applicable environment scripts, must be performed sometime before executing nmake.

3.4 Compiling

After running configure.cmd 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. Presently, the all target causes only the static library to be built.

---

3If 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.
### 3.4. Compiling

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

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-build target will not remove any object files if they do not exist.

```
C:\field\temp\libflame-5.1.0\windows>nmake
```

Table 3.3 lists the most useful nmake targets defined in the Makefile that resides in the windows directory. As nmake compiles individual source files into object files, it will output progress information. By default (ie: with verbose output disabled), this appears as:

```
C:\field\temp\libflame-5.1.0\windows>nmake
Microsoft (R) Program Maintenance Utility Version 9.00.30729.01
Copyright (C) Microsoft Corporation. All rights reserved.
nmake: Creating \obj\flamec\x64\debug directory
nmake: Compiling \src\flamec\bl1_amax.c
nmake: Compiling \src\flamec\bl1_asum.c
nmake: Compiling \src\flamec\bl1_axpy.c
nmake: Compiling \src\flamec\bl1_axpymt.c
nmake: Compiling \src\flamec\bl1_axpysmt.c
nmake: Compiling \src\flamec\bl1_axpysv.c
nmake: Compiling \src\flamec\bl1_axpyv.c
nmake: Compiling \src\flamec\bl1_check.c
nmake: Compiling \src\flamec\bl1_conjm.c
nmake: Compiling \src\flamec\bl1_conjmr.c
nmake: Compiling \src\flamec\bl1_conjv.c
```

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

```
nmake: Creating \lib\x64\debug directory
nmake: Creating static library \lib\x64\debug\libflame-x64-5.1.0.lib
```
As you can see, the “x64” architecture string (provided at configure-time) and “5.1.0” revision string were inserted into the final library name. `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`.

### 3.5 Installation

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

```plaintext
C:\field\temp\libflame-5.1.0\windows>dir lib\x64\debug
Volume in drive C has no label.
Volume Serial Number is B4E3-D9FC
Directory of C:\field\temp\libflame-5.1.0\windows\lib\x64\debug
12/01/2009 01:19 PM <DIR> .
12/01/2009 01:19 PM <DIR> ..
12/01/2009 01:19 PM 45,800,190 libflame-x64-5.1.0.lib
  1 File(s)  45,800,190 bytes
  2 Dir(s)  85,181,444,096 bytes free
```

Once library has been built, it 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 variable:

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

After this variable is set, the `nmake` install target may be invoked. This results in the static library being built, if it was not already, and then copied to its destination directory, specified by the `INSTALL_PREFIX` `nmake` variable. The `install` target also copies the `libflame` header files.

```plaintext
C:\field\temp\libflame-5.1.0\windows>nmake install
Microsoft (R) Program Maintenance Utility Version 9.00.30729.01
Copyright (C) Microsoft Corporation. All rights reserved.
nmake: Installing .\lib\x64\debug\libflame-x64-5.1.0.lib to c:\field\lib\libflame\lib
nmake: Installing libflame header files to c:\field\lib\libflame\include-x64-5.1.0
```

At this point, the static library and header files are ready to use.

### 3.6 Dynamic library generation

The Windows build system is equipped to optionally generate a dynamically-linked library (DLL). At the time of this writing, `libflame` developers consider the DLL generation to be experimental and likely to not work. Still, we provide instructions in this section for intrepid users, or experts who wish to tinker and/or provide us with feedback.

After running `configure.cmd`, edit the contents of the `linkargs.txt` file. This file should be modified to include (1) any linker options the user may need or want, (2) a list of system libraries necessary for successful linking, (3) a list of library paths to add to the list used when the aforementioned libraries are

---

4 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.
being searched for by the linker, and (4) a path to the BLAS (and LAPACK if the user enabled external LAPACK interfaces at configure-time). 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 options, libraries, and/or library paths. The following is an example of the contents of linkargs.txt.

```
/nologo
/LD /MT
/LIBPATH:"C:\Program Files\Microsoft SDKs\Windows\v6.0A\Lib"
/LIBPATH:"C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\lib"
/nodefaultlib:libcmt /nodefaultlib:libc /nodefaultlib:libmmt
msvcrt.lib
/LIBPATH:"C:\Program Files (x86)\Intel\Compiler\11.1\048\lib\ia32"
/LIBPATH:"C:\Program Files (x86)\Intel\Compiler\11.1\048\mkl\ia32\lib"
mkl_intel_c.lib
mkl_sequential.lib
mkl_core.lib
```

The libflame distribution also includes a file named linkargs64.txt which contains the equivalent paths and flags necessary for 64-bit linking:

```
/nologo
/LD /MT
/LIBPATH:"C:\Program Files\Microsoft SDKs\Windows\v6.0A\Lib\x64"
/LIBPATH:"C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\lib\amd64"
/nodefaultlib:libcmt /nodefaultlib:libc /nodefaultlib:libmmt
msvcrt.lib
/LIBPATH:"C:\Program Files (x86)\Intel\Compiler\11.1\048\lib\intel64"
/LIBPATH:"C:\Program Files (x86)\Intel\Compiler\11.1\048\mkl\em64t\lib"
mkl_intel_lp64.lib
mkl_sequential.lib
mkl_core.lib
```

<table>
<thead>
<tr>
<th>Link type</th>
<th>Component files</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>libflame-x64-r3692.lib</td>
<td>The static library containing the libflame object files. Link to this file when statically linking your application to libflame.</td>
</tr>
<tr>
<td>dynamic</td>
<td>libflame-x64-r3692.dll</td>
<td>The dynamic library containing the libflame object code. This file is loaded into memory by the operating system at runtime the first time a dependent program or library references libflame symbols.</td>
</tr>
<tr>
<td></td>
<td>libflame-x64-r3692.lib</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td>libflame-x64-r3692.exp</td>
<td>The export file. This file is necessary only when building other dynamic libraries that depend on a dynamic library build of libflame.</td>
</tr>
</tbody>
</table>

Table 3.4: The files generated when building revision r3692 of libflame as either a static or dynamic library. The filenames reflect using “x64” as the architecture string when running configure.cmd.

Simply replace the contents of linkargs.txt with the contents of linkargs64.txt if you wish to generate a 64-bit library. The file may need some tweaking, depending on your development environment.
Note that in the above examples we link against MKL. The dynamic build of libflame requires a BLAS implementation at the time the DLL is generated. This is necessary so the linker can resolve all BLAS symbol references within libflame at the time the library is built. To specify a different BLAS library, simply replace the /LIBPATH entries and .lib filenames accordingly.

After building the static library, the user may re-use the object files to generate the DLL. Simply invoke the dll target:

```
C:\field\temp\libflame-5.1.0\windows>nmake dll
nmake: Creating dynamic library \dll\\x64\debug\libflame-x64-5.1.0.dll
 Creating library libflame-x64-5.1.0.lib and object libflame-x64-5.1.0.exp
```

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 5.1.0 with the architecture string “x64”.

Once generated, the dynamic library files reside in a directory named dll:

```
C:\field\temp\libflame-5.1.0\windows>dir dll\x64\debug
Directory of C:\field\temp\libflame-5.1.0\windows\dll\x64\debug
12/01/2009 01:33 PM <DIR>    ..
12/01/2009 01:33 PM <DIR>    ..
12/01/2009 01:33 PM 19,907,584 libflame-x64-5.1.0.dll
12/01/2009 01:33 PM 618     libflame-x64-5.1.0.dll.manifest
12/01/2009 01:33 PM 330,905 libflame-x64-5.1.0.exp
12/01/2009 01:33 PM 573,048 libflame-x64-5.1.0.lib
  4 File(s)   20,812,155 bytes
  2 Dir(s)   85,156,532 bytes free
```

The user may then invoke the install-dll target to install the DLL files to the directory specified by INSTALL_PREFIX in build\defs.mk:

```
C:\field\libflame-wc\windows>nmake install-dll
nmake: Installing .\dll\x64\debug\libflame-x64-5.1.0.dll to c:\field\lib\libflame\dll
nmake: Installing .\dll\x64\debug\libflame-x64-5.1.0.lib to c:\field\lib\libflame\dll
nmake: Installing .\dll\x64\debug\libflame-x64-5.1.0.exp to c:\field\lib\libflame\dll
```

If you haven’t already run the install target for a static library install, you’ll need to manually invoke the install-headers target so that the libflame header files are copied to the install directory.

### 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.5

5 We acknowledge that most users will probably be using an integrated development environment (IDE) 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.
3.7. Linking against libflame

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 5.1.0 with the “x64” 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-x64-5.1.0.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:

```makefile
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 \
/LIBPATH:"C:\Program Files\Microsoft SDKs\Windows\v6.0A\Lib\x64" \
/LIBPATH:"C:\Program Files (x86)\Microsoft Visual Studio 9.0\VC\lib\amd64" \
/nodefaultlib:libcmt /nodefaultlib:libc /nodefaultlib:libmmt \
msvcrt.lib
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) $(MYAPP_OBJS) /Fe$(MYAPP_BIN) $(LDFLAGS) $(LAPACK_LIB) $(BLAS_LIB)
clean:
  del /F /Q $(MYAPP_OBJS) $(MYAPP_BIN)
  del /F /Q *.manifest
```

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

```makefile
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) /I$(FLAME_INC)
LDFLAGS = /nologo \
```

The Makefile updates the include path to include the libflame header directory and adds the libflame library to the link command.
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 static build of libflame. In principle, the user may also link to a dynamically-linked copy of libflame. However, as mentioned previously, the DLL instantiation of libflame is considered experimental and likely to not link properly.
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 acquires a matrix buffer and its dimension properties, performs a Cholesky factorization on the matrix, and then frees the memory associated with the matrix buffer.

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

    // Get the matrix buffer address, size, and row and column strides.
    get_matrix_info( &buffer, &m, &rs, &cs );

    // Compute the Cholesky factorization of the matrix, reading from and
    // updating the lower triangle.
    dpotrf_( &uplo, &m, buffer, &cs, &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 row- or column-major matrix buffer to the bufferless libflame object. This method almost always requires the fewest number of 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
#include "FLAME.h"

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

    // Initialize libflame.
    FLA_Init();

    // Get the matrix buffer address, size, and row and column strides.
    get_matrix_info( &buffer, &m, &rs, &cs );

    // Create an $m \times m$ 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, rs, cs, &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.

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_buffer_to_object() 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 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 row and column strides rs and cs. 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
4.2 FLASH examples

```c
#include "FLAME.h"

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

    // Initialize libflame.
    FLA_Init();

    // Get the matrix buffer address, size, and row and column strides.
    get_matrix_info( &buffer, &m, &rs, &cs );

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

    // Copy the contents of the conventional matrix into a libflame object.
    FLA_Copy_buffer_to_object( FLA_NO_TRANSPOSE, m, m, buffer, rs, cs, 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_buffer_to_object(). Note that the matrix is copied all at once, and thus here we assume that original matrix is fully initialized in initialize_matrix()

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_buffer_to_object() in a loop that copies a single submatrix per iteration. We use FLA_Submatrix_at() to compute the starting address of the submatrix whose top-left element is the (i,j) element within the overall matrix stored in buffer.

Note that FLA_Copy_buffer_to_object() 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 \texttt{libflame} function names are the same as in Figure 4.2 except that the prefix has changed from \texttt{FLA} to \texttt{FLASH}. Additionally, all of the function type signatures are the same, except for the invocation to \texttt{FLASH\_Obj\_create\_without\_buffer()}.

This function takes two additional arguments: a depth, and an array of block sizes.\footnote{Since 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.} 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 \texttt{FLA\_Copy\_buffer\_to\_object()} accepts a transposition argument, \texttt{FLASH\_Copy\_flat\_to\_hier()} 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 \texttt{FLASH\_Copy\_flat\_to\_hier()} 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 \texttt{FLASH} API (1) provides an easy interface for creating and manipulating hierarchical objects, and (2) is strikingly similar to the original \texttt{FLAME/C} API wherever possible.

### 4.3 SuperMatrix examples
#include "FLAME.h"

int main( void )
{
    double* buffer;
    int m, rs, cs, b;
    int i, j;
    FLA_Obj A;

    // Initialize libflame.
    FLA_Init();

    // Get the matrix buffer address, size, row and column strides, and block size.
    get_matrix_info( &buffer, &m, &rs, &cs, &b );

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

            // 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).
            ij_ptr = FLA_Submatrix_at( FLA_DOUBLE, buffer, i, j, rs, cs );

            // Copy the current block into the correct location within the libflame object.
            FLA_Copy_buffer_to_object( FLA_NO_TRANSPOSE, b_m, b_n, ij_ptr, rs, cs, 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.
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, rs, cs, b;
    FLA_Obj A;

    // Initialize libflame.
    FLA_Init();

    // Get the matrix buffer address, size, row and column strides, and blocksize.
    get_matrix_info( &m, &rs, &cs, &b );

    // 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_buffer_to_hier( m, m, buffer, rs, cs, 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.
```c
#include "FLAME.h"

int main( void )
{
  double* buffer;
  int m, rs, cs, b;
  int i, j;
  FLA_Obj A;

  // Initialize libflame.
  FLA_Init();

  // Get the matrix buffer address, size, row and column strides, and blocksize.
  get_matrix_info( &buffer, &m, &rs, &cs, &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;

      // 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).
      ij_ptr = FLA_Submatrix_at( FLA_DOUBLE, buffer, i, j, rs, cs );

      // Copy the current block into the correct location within the libflame object.
      FLASH_Copy_buffer_to_hier( b_m, b_n, ij_ptr, rs, cs, 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 \texttt{libflame}.

5.1 Conventions

Before describing the \texttt{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

- \textit{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 \texttt{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, length (number of rows), and width (number of columns).

- \textit{Real matrix}. A real matrix is one that contains only real numbers.

- \textit{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.

- \textit{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.

- \textit{Symmetric matrix}. A symmetric matrix is a square matrix whose \((i,j)\) entry is equal to its \((j,i)\). In \texttt{libflame}, only the upper or lower triangle of a symmetric matrix is referenced.\footnote{Symmetric, Hermitian, and triangular matrices use the same amount of storage space as a general matrix with identical dimensions. That is, \texttt{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.}

- \textit{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 \texttt{libflame}, only the upper or lower triangle of a Hermitian matrix is stored or referenced.\footnote{Symmetric, Hermitian, and triangular matrices use the same amount of storage space as a general matrix with identical dimensions. That is, \texttt{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: \( \alpha, \beta, \rho \)).

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 \( 1 \times 1 \) 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 \( 1 \times 1 \).

#### 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, \( \text{dgemm()} \) 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 \( \text{zdscal()} \) 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: \( (s, d, c, \text{or} z) \). Example: \( ?\text{gemm()} \) refers to the four level-3 BLAS routines that implement the GEMM operation: \( \text{sgemm()}, \text{dgemm()}, \text{cgemm()}, \text{and zgemm()} \).

- *: 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: \( *\text{scal()} \) refers to the six level-1 BLAS routines that implement the SCAL operation: \( \text{sscal()}, \text{dscal()}, \text{cscal()}, \text{csscal()}, \text{zscal()}, \text{and zdscal()} \).

#### 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[10].
<table>
<thead>
<tr>
<th>Type</th>
<th>Typical parameter name</th>
<th>Permitted values</th>
<th>Of interest to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLA_Bool</td>
<td>return value</td>
<td>TRUE, FALSE</td>
<td>all users</td>
</tr>
<tr>
<td>FLA_Datatype</td>
<td>datatype</td>
<td>FLA_INT, FLA_FLOAT, FLA_DOUBLE, FLA_COMPLEX, FLA_DOUBLE_COMPLEX, FLA_CONSTANT</td>
<td>all users</td>
</tr>
<tr>
<td>FLA_Elemtype</td>
<td>elemtype</td>
<td>FLA_SCALAR, FLA_MATRIX</td>
<td>advanced users and developers</td>
</tr>
<tr>
<td>FLA_Matrix_type</td>
<td>matrix_type</td>
<td>FLA_FLAT, FLA_HIER</td>
<td>advanced users and developers</td>
</tr>
<tr>
<td>FLA.Side</td>
<td>side</td>
<td>FLA_LEFT, FLA_RIGHT</td>
<td>all users</td>
</tr>
<tr>
<td>FLA_Uplo</td>
<td>uplo</td>
<td>FLA_LOWER_TRIANGULAR, FLA_UPPER_TRIANGULAR</td>
<td>all users</td>
</tr>
<tr>
<td>FLA_Trans</td>
<td>trans</td>
<td>FLA_NO_TRANSPOSE, FLA_TRANSPOSE, FLA_CONJ_NO_TRANSPOSE, FLA_CONJ_TRANSPOSE</td>
<td>all users</td>
</tr>
<tr>
<td>FLA_Conj</td>
<td>conj</td>
<td>FLA_NO_CONJUGATE, FLA_CONJUGATE</td>
<td>all users</td>
</tr>
<tr>
<td>FLA_Diag</td>
<td>diag</td>
<td>FLA_NONUNIT_DIAG, FLA_UNIT_DIAG, FLA_ZERO_DIAG</td>
<td>all users</td>
</tr>
<tr>
<td>FLA_Quadrant</td>
<td>quadrant</td>
<td>FLA_TL, FLA_TR, FLA_BL, FLA_BR</td>
<td>all users</td>
</tr>
<tr>
<td>FLA_Direct</td>
<td>direct</td>
<td>FLA_FORWARD, FLA_BACKWARD</td>
<td>all users</td>
</tr>
<tr>
<td>FLA_Store</td>
<td>storev</td>
<td>FLA_ROWWISE, FLA_COLUMNWISE</td>
<td>all users</td>
</tr>
<tr>
<td>FLA_Pivot_type</td>
<td>ptype</td>
<td>FLA_NATIVE_PIVOTS, FLA_LAPACK_PIVOTS</td>
<td>all users</td>
</tr>
<tr>
<td>FLA_Error</td>
<td>return value</td>
<td>FLA_SUCCESS, FLA_FAILURE...</td>
<td>all users</td>
</tr>
<tr>
<td>FLA_Inv</td>
<td>inv</td>
<td>FLA_NO_INVERSE, FLA_INVERSE</td>
<td>all users</td>
</tr>
</tbody>
</table>

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_Trmvex_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 contraints 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().

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.

---

3 The 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. Conventions

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 a matrix object refers to the distance in memory that separates adjacent columns (for column-major storage) or rows (for row-major storage). In this document, we prefer to identify the row and column strides explicitly to remove ambiguity as to the storage format. A row stride of 1 implies that the matrix is stored in column-major order, and likewise a column stride of 1 implies row-major storage. A matrix stored in column-major order often has a column stride equal to the \( m \) dimension, though it could be larger. Similarly, a row-major matrix will have a row stride equal to or greater than the \( n \) dimension. It is also quite common for a matrix object to refer to a submatrix of a larger matrix, in which case the row or column stride will exceed the \( m \) or \( n \) dimensions, respectively, for column- and row-major cases.

- **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 while row vectors typically have a non-unit increment. However, sometimes vectors are created individually (ie: they do not exist as part of a larger matrix) in which case they may be interpreted as either row or column vectors. 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.** **libflame** interfaces with three kinds of matrix storage schemes:
  - **Flat objects.** The primary means of storing matrices in **libflame** is within “flat” matrix objects. These objects store their numerical contents in either row- or column-major order, depending on the row and column strides given when the object is created. Most **libflame** functions operate on flat objects.
  - **Conventional matrix buffers.** Many legacy applications interface to their matrices by indexing directly into the matrix buffer. These so-called conventional matrices are essentially identical to a row-major or column-major flat object, except that the matrix properties are not encapsulated in a **libflame** object. To compute with conventional matrices, the user must first “attach” the matrix buffer and other information to a “bufferless” object. The user may then compute with the object as if it were a created natively within **libflame** and subsequently access the results directly via the buffer address. See the descriptions for `FLA_Obj_create_without_buffer()` and `FLA_Obj_attach_buffer()` for more information on interfacing with matrices stored conventionally.
  - **Hierarchical objects.** It is often advantageous to store a matrix by blocks that are contiguous in memory. When used within an algorithm-by-blocks, this storage scheme provides additional spatial locality when compared to conventional/flat matrix storage. The details of the hierarchical storage scheme, however, are intentionally hidden from the user. See Section 5.4 for more information on the motivation for hierarchical storage and the **libflame** APIs for creating and manipulating hierarchical objects.
• **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 $1 \times 1$ object to serve as a scaling factor in the operation in question. The GEMM operation, for example, has two of these scalars, $\alpha$ and $\beta$. 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.

## 5.2 FLAME/C Basics

### 5.2.1 Initialization and finalization

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

**Purpose:** Initialize the library.

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

```c
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**.

```c
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

```c
FLA_Error FLA_Obj_create( FLA_Datatype datatype, dim_t m, dim_t n,
                         dim_t rs, dim_t cs, 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`.

**Notes:** Currently, `libflame` supports both column-major storage and row-major storage, but *not* general storage (that is, storage in which neither rows nor columns are stored contiguously in memory). In most cases, the user should create objects according to the following policy: if column-major storage is desired, \( rs \) should be set to 1 and \( cs \) should be set to \( m \); otherwise, if row-major storage is desired, \( rs \) should be set to \( n \) and \( cs \) should be set to 1. Invoking `FLA_Obj_create()` with both \( rs \) and \( cs \) equal to zero is interpreted as a request for the default storage scheme, which is currently column-major storage.

**Returns:** `FLA_SUCCESS`

**Constraints:**
- \( rs \) and \( cs \) must either both be zero, or non-zero. Also, one of the two strides must be equal to 1. If \( rs \) is equal to 1, then \( cs \) must be at least \( m \); otherwise, if \( cs \) is equal to 1, then \( rs \) must be at least \( n \).
- `trans` may not be `FLA_CONJ_TRANSPOSE` or `FLA_CONJ_NO_TRANSPOSE`.
- The datatype of \( B \) may not be `FLA_CONSTANT`.

**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.
- \( rs \) – The row stride of the underlying data buffer in new object.
- \( cs \) – The column stride of the underlying data buffer in 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. After obj_new is created, it must be initialized before it is used in any computation which reads its numerical data.

Notes: The caller may use FLA_CONJ_NO_TRANSPOSE and FLA_CONJ_TRANSPOSE for the trans argument. The conjugation component of these values is ignored and thus for this routine they are effectively equivalent to FLA_NO_TRANSPOSE and FLA_TRANSPOSE, respectively.

Notes: The new object, obj_new, is created with similar storage properties as obj_cur. For example, if obj_cur is stored in column-major order, then obj_new is created with column-major order as well. However, the object is created with a minimal leading dimension (the column stride for column-major storage, or the row stride for row-major storage), such that there is no excess storage beyond the bounds of the matrix.

Returns: FLA_SUCCESS

Arguments:

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

FLA_Error FLA_Obj_create_copy_of( 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. After obj_new is created, it is initialized with the contents of obj_cur, applying a transposition according to trans.

Notes: The caller may use FLA_CONJ_NO_TRANSPOSE and FLA_CONJ_TRANSPOSE for the trans argument. The conjugation component of these values is ignored and thus for this routine they are effectively equivalent to FLA_NO_TRANSPOSE and FLA_TRANSPOSE, respectively.

Notes: The new object, obj_new, is created with similar storage properties as obj_cur. For example, if obj_cur is stored in column-major order, then obj_new is created with column-major order as well. However, the object is created with a minimal leading dimension (the column stride for column-major storage, or the row stride for row-major storage), such that there is no excess storage beyond the bounds of the matrix.

Returns: FLA_SUCCESS

Arguments:

- trans – Indicates whether to create the object pointed to by obj_new with transposed dimensions.
- obj_cur – An existing FLA_Obj.
- obj_new (on entry) – A pointer to an uninitialized FLA_Obj.
- obj_new (on exit) – A pointer to a new FLA_Obj parameterized by the datatype and dimensions of obj_cur with its numerical contents identical to that of obj_cur.
FLA_Error FLA_Obj_free( FLA_Obj* obj );

Purpose: Release all resources allocated to an object. This includes the object resources as well as the data buffer associated with the object. Upon returning, obj points to a structure which is, for all intents and purposes, uninitialized.

Returns: FLA_SUCCESS

Arguments:

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

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:

obj – An FLA_Obj.

dim_t FLA_Obj_length( FLA_Obj obj );

Purpose: Query the number of rows in a view into 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 a view into 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 view’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 view'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 view'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_Error FLA_Obj_show( char* header, FLA_Obj obj, char* format, char* footer );

Purpose: Display the numerical values contained in the object view obj. The string header is output first (followed by a newline), then formatted contents of obj, and finally the string footer (followed by a newline). The string format should contain a printf()-style format string that describes how to output each element of the matrix. Note that format must be set according to the numerical contents of obj. For example, if the datatype of obj is FLA_DOUBLE, the user may choose to use "%11.3e" as the format string. If the object were of type FLA_DOUBLE_COMPLEX, the user would use the same format string, however, internally it would be duplicated to denote both real and imaginary components (ie: "%11.3e + %11.3e").
Returns: FLA_SUCCESS
Arguments:
  header – A pointer to a string to precede the formatted output of obj.
  obj – An FLA_Obj.
  format – A pointer to a printf()-style format string.
  footer – A pointer to a string to proceed the formatted output of obj.
FLA_Error FLA_Obj_fshow( FILE* file, char* header, FLA_Obj obj, char* format, char* footer);

**Purpose:** Display the numerical values contained in `obj`. FLA_Obj_fshow() and FLA_Obj_show() are identical except that the former prints its output to a file stream whereas the latter prints to standard output.

**Notes:** The user must ensure that the file stream corresponding to `file` has been opened and is writable, and also that an error has not occurred on a previous write.

**Returns:** FLA_SUCCESS

**Imp. Notes:** FLA_Obj_fshow() uses `fprintf()` to write output to `file`. It is possible that one of these write requests will cause an error that prevents subsequent invocations of `fprintf()` from succeeding. As it is currently implemented, FLA_Obj_fshow() does not report such errors.

**Arguments:**
- `file` – A file pointer returned via `fopen()`.
- `header` – A pointer to a string to precede the formatted output of `obj`.
- `obj` – An FLA_Obj.
- `format` – A pointer to a `printf()`-style format string.
- `footer` – A pointer to a string to proceed the formatted output of `obj`.

### 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() or allocate a new buffer for the object via FLA_Obj_create_buffer().

**Notes:** The object’s datatype will have already been set when FLA_Obj_create_without_buffer() returns. Thus, if the user plans on attaching a buffer via FLA_Obj_attach_buffer(), he 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_create_buffer( dim_t rs, dim_t cs, FLA_Obj* obj );

**Purpose:** Allocate a new buffer for an object that was previously created via \texttt{FLA\_Obj\_create\_without\_buffer()}. The function uses \texttt{rs} and \texttt{cs} to set the row and column strides, respectively, which will be used when subsequent functions access the matrix elements.

**Notes:** Currently, one of \texttt{rs} and \texttt{cs} must be unit, corresponding to either column-major or row-major storage. Passing zero for both parameters is interpreted as a request for the default storage scheme, with is column-major.

**Returns:** FLA\_SUCCESS

**Arguments:**
- \texttt{rs} – The row stride of the matrix buffer that will be allocated.
- \texttt{cs} – The column stride of the matrix buffer that will be allocated.
- \texttt{obj} (on entry) – A pointer to a valid \texttt{FLA\_Obj} that was created without a buffer.
- \texttt{obj} (on exit) – A pointer to a valid \texttt{FLA\_Obj} with a buffer large enough to encapsulate an \(m \times n\) matrix, according to row and column strides \texttt{rs} and \texttt{cs}, where \(m, n\), and the datatype were previously determined via \texttt{FLA\_Obj\_create\_without\_buffer()}. 

FLA_Error FLA_Obj_free_without_buffer( FLA_Obj* obj );

**Purpose:** Release all resources allocated to an object, but do not release the buffer attached to the object. Upon returning, \texttt{obj} points to a structure which is, for all intents and purposes, uninitialized.

**Returns:** FLA\_SUCCESS

**Arguments:**
- \texttt{obj} (on entry) – A pointer to a valid \texttt{FLA\_Obj}.
- \texttt{obj} (on exit) – A pointer to an uninitialized \texttt{FLA\_Obj}.

FLA_Error FLA_Obj_free_buffer( FLA_Obj* obj );

**Purpose:** Release only the buffer memory associated with an object. The rest of the object is left untouched. After calling this routine, the user should ensure that the rest of the object is freed via \texttt{FLA\_Obj\_free\_without\_buffer()}. 

**Notes:** When freeing the buffer and object separately, the buffer \textit{must} be freed first. That is, \texttt{FLA\_Obj\_free\_buffer()} must be called before \texttt{FLA\_Obj\_free\_without\_buffer()}. 

**Returns:** FLA\_SUCCESS

**Arguments:**
- \texttt{obj} (on entry) – A pointer to a valid \texttt{FLA\_Obj}.
- \texttt{obj} (on exit) – A pointer to a bufferless \texttt{FLA\_Obj}. 

FLA_Error FLA_Obj_attach_buffer( void* buffer, dim_t rs, dim_t cs, 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 row and column strides `rs` and `cs`, which, combined with the `m` and `n` dimensions submitted when the object was created, will determine what region of memory is accessible. A row or column stride 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. Alternatively, you may call `FLA_Obj_free()` if you wish to free both the previously allocated buffer and the `FLA_Obj` itself.

**Returns:** `FLA_SUCCESS`

**Constraints:**

- `rs` and `cs` must either both be zero, or non-zero. Also, one of the two strides must be equal to 1. If `rs` is equal to 1, then `cs` must be at least `m`; otherwise, if `cs` is equal to 1, then `rs` must be at least `n`.

**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.
- `rs` – The row stride of the matrix stored conventionally in `buffer`.
- `cs` – The column stride of the matrix stored conventionally in `buffer`.
- `obj` (on entry) – A pointer to a valid `FLA_Obj` that was created without a buffer.
- `obj` (on exit) – A pointer to a valid `FLA_Obj` that encapsulates the data in `buffer`.

void* FLA_Obj_buffer_at_view( FLA_Obj obj );

**Purpose:** Query the starting address of an object view’s underlying numerical data buffer. The address of the view is computed according to current row and column offset of the object view, and is *not* necessarily the starting address of the overall object.

**Notes:** Since the address returned by `FLA_Obj_buffer_at_view()` 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`.
58 5. User-level Application Programming Interfaces

\texttt{dim\_t FLA\_Obj\_row\_stride( FLA\_Obj obj );}

**Purpose:** Query the row stride associated with the object’s underlying element data buffer. The row stride is the number of elements that separates matrix element \((r,c)\) from element \((r+1,c)\).

**Notes:** \texttt{libflame} supports both row- and column-major storage for matrix objects. When a matrix object is stored in column-major order, its row stride is, by definition, equal to 1. Likewise, when a matrix object is stored in row-major order, its column stride is by definition equal to 1.

**Returns:** An unsigned integer value of type \texttt{dim\_t}.

**Arguments:**
\begin{itemize}
  \item \texttt{obj} – An \texttt{FLA\_Obj}.
\end{itemize}

\texttt{dim\_t FLA\_Obj\_col\_stride( FLA\_Obj obj );}

**Purpose:** Query the column stride associated with the object’s underlying element data buffer. The column stride is the number of elements that separates matrix element \((r,c)\) from element \((r,c+1)\).

**Notes:** \texttt{libflame} supports both row- and column-major storage for matrix objects. When a matrix object is stored in column-major order, its row stride is, by definition, equal to 1. Likewise, when a matrix object is stored in row-major order, its column stride is by definition equal to 1.

**Returns:** An unsigned integer value of type \texttt{dim\_t}.

**Arguments:**
\begin{itemize}
  \item \texttt{obj} – An \texttt{FLA\_Obj}.
FLA_Error FLA_Copy_buffer_to_object( FLA_Trans trans, dim_t m, dim_t n, void* A,
               dim_t rs, dim_t cs, dim_t i, dim_t j, FLA_Obj B );

Purpose:  Copy the contents of an $m \times n$ conventional row- or column-major matrix $A$ with row
and column strides $rs$ and $cs$ 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.

Returns:  FLA_SUCCESS

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 $i + n \times j + m$.
- $rs$ and $cs$ must either both be zero, or non-zero. Also, one of the two strides must be
  equal to 1. If $rs$ is equal to 1, then $cs$ must be at least $m$; otherwise, if $cs$ is equal to
  1, then $rs$ must be at least $n$.
- trans may not be FLA_CONJ_TRANSPOSE or FLA_CONJ_NO_TRANSPOSE.
- The datatype of $B$ may not be FLA_CONSTANT.

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 $A$.
- rs  – The row stride of $A$.
- cs  – The column stride 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 FLA_Obj representing matrix $B$. 
FLA_Error FLA_Copy_object_to_buffer( FLA_Trans trans, dim_t i, dim_t j, FLA_Obj A, 
                      dim_t m, dim_t n, void* B, dim_t rs, dim_t cs );

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 row- or column-major matrix \( B \) with row and column strides \( rs \) 
and \( cs \). 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.

Returns: FLA_SUCCESS

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 \( i + n \times j + m \).
- \( rs \) and \( cs \) must either both be zero, or non-zero. Also, one of the two strides must be 
equal to 1. If \( rs \) is equal to 1, then \( cs \) must be at least \( m \); otherwise, if \( cs \) is equal to 
1, then \( rs \) must be at least \( n \).
- \( trans \) may not be FLA_CONJ_TRANSPOSE or FLA_CONJ_NO_TRANSPOSE.
- The datatype of \( A \) may not be FLA_CONSTANT.

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 FLA_Obj 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 \( B \).
- \( rs \) – The row stride of \( B \).
- \( cs \) – The column stride of \( B \).
5.2. FLAME/C Basics

FLA_Error FLA_Axpy_buffer_to_object( FLA_Trans trans, FLA_Obj alpha,  
    dim_t m, dim_t n, void* A, dim_t rs, dim_t cs,  
    dim_t i, dim_t j, FLA_Obj B );

Purpose: Perform one of the following operations:

\[
\begin{align*}
B_{ij} & := B_{ij} + \alpha A \\
B_{ij} & := B_{ij} + \alpha A^T
\end{align*}
\]

where \( \alpha \) is a scalar, \( A \) is an \( m \times n \) conventional row- or column-major matrix with row and column strides \( rs \) and \( cs \), and \( B_{ij} \) is the submatrix whose top-left element is the \((i,j)\) entry of \( B \). The \texttt{trans} argument may be used to optionally transpose \( A \) during the operation.

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

Returns: FLA_SUCCESS

Constraints:

- If \texttt{trans} equals \texttt{FLA_NO_TRANSPOSE}, then \( B \) must be at least \( i + m \times j + n \); otherwise, if \texttt{trans} equals \texttt{FLA_TRANSPOSE}, then \( B \) must be at least \( i + n \times j + m \).
- \( rs \) and \( cs \) must either both be zero, or non-zero. Also, one of the two strides must be equal to 1. If \( rs \) is equal to 1, then \( cs \) must be at least \( m \); otherwise, if \( cs \) is equal to 1, then \( rs \) must be at least \( n \).
- \texttt{trans} may not be \texttt{FLA_CONJ_TRANSPOSE} or \texttt{FLA_CONJ_NO_TRANSPOSE}.
- The datatype of \( B \) may not be \texttt{FLA_CONSTANT}.

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{trans}</td>
<td>Indicates whether to transpose the matrix ( A ) during the operation.</td>
</tr>
<tr>
<td>\texttt{alpha}</td>
<td>An \texttt{FLA_Obj} representing scalar ( \alpha ).</td>
</tr>
<tr>
<td>\texttt{m}</td>
<td>The number of rows in ( A ) and ( B_{ij} ) referenced by the operation.</td>
</tr>
<tr>
<td>\texttt{n}</td>
<td>The number of columns in ( A ) and ( B_{ij} ) referenced by the operation.</td>
</tr>
<tr>
<td>\texttt{A}</td>
<td>A pointer to the first element in ( A ).</td>
</tr>
<tr>
<td>\texttt{rs}</td>
<td>The row stride of ( A ).</td>
</tr>
<tr>
<td>\texttt{cs}</td>
<td>The column stride of ( A ).</td>
</tr>
<tr>
<td>\texttt{i}</td>
<td>The row offset in ( B ) of the submatrix ( B_{ij} ).</td>
</tr>
<tr>
<td>\texttt{j}</td>
<td>The column offset in ( B ) of the submatrix ( B_{ij} ).</td>
</tr>
<tr>
<td>\texttt{B}</td>
<td>An \texttt{FLA_Obj} representing ( B ).</td>
</tr>
</tbody>
</table>
FLA_Error FLA_Axpy_object_to_buffer( FLA_Trans trans, FLA_Obj alpha,
    dim_t i, dim_t j, FLA_Obj A,
    dim_t m, dim_t n, void* B, dim_t rs, dim_t cs );

Purpose: Perform one of the following operations:

\[
B := B + \alpha A_{ij} \\
B := B + \alpha A_{ij}^T
\]

where \( \alpha \) is a scalar, \( A_{ij} \) is the submatrix whose top-left element is the \((i,j)\) entry of \( A \), and \( B \) is an \( m \times n \) conventional row- or column-major matrix with row and column strides \( rs \) and \( cs \). The \( \text{trans} \) argument may be used to optionally transpose \( A_{ij} \) during the operation.

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

Returns: FLA_SUCCESS

Constraints:  
- If \( \text{trans} \) equals FLA_NO_TRANSPOSE, then \( A \) must be at least \( i + m \times j + n \); otherwise, if \( \text{trans} \) equals FLA_TRANSPOSE, then \( A \) must be at least \( i + n \times j + m \).  
- \( rs \) and \( cs \) must either both be zero, or non-zero. Also, one of the two strides must be equal to 1. If \( rs \) is equal to 1, then \( cs \) must be at least \( m \); otherwise, if \( cs \) is equal to 1, then \( rs \) must be at least \( n \).  
- \( \text{trans} \) may not be FLA_CONJ_TRANSPOSE or FLA_CONJ_NO_TRANSPOSE.  
- The datatype of \( A \) may not be FLA_CONSTANT.

Arguments:  
- \( \text{trans} \) – Indicates whether to transpose the matrix \( B \) during the operation.  
- \( \alpha \) – An FLA_Obj representing scalar \( \alpha \).  
- \( i \) – The row offset in \( A \) of the submatrix \( A_{ij} \).  
- \( j \) – The column offset in \( A \) of the submatrix \( A_{ij} \).  
- \( A \) – An FLA_Obj representing \( A \).  
- \( m \) – The number of rows in \( A_{ij} \) and \( B \) referenced by the operation.  
- \( n \) – The number of columns in \( A_{ij} \) and \( B \) referenced by the operation.  
- \( B \) – A pointer to the first element in \( B \).  
- \( rs \) – The row stride of \( B \).  
- \( cs \) – The column stride of \( B \).

5.2.5 More query functions

FLA_Datatype FLA_Obj_datatype_proj_to_real( FLA_Obj obj );

Purpose: Query the real projection of an object’s datatype. If the object datatype is single precision (ie: FLA_FLOAT or FLA_COMPLEX) then FLA_FLOAT is returned; otherwise, FLA_DOUBLE is returned.

Returns: One of \{FLA_FLOAT, FLA_DOUBLE\}.

Constraints:  
- The numerical datatype of \( obj \) must be floating-point, and must not be FLA_CONSTANT.

Arguments:  
- \( obj \) – An FLA_Obj.
5.2. FLAME/C Basics

FLA_Datatype FLA_Obj_datatype_proj_to_complex( FLA_Obj obj );

Purpose: Query the complex projection of an object's datatype. If the object datatype is single precision (ie: FLA_FLOAT or FLA_COMPLEX) then FLA_COMPLEX is returned; otherwise, FLA_DOUBLE_COMPLEX is returned.

Returns: One of {FLA_COMPLEX, FLA_DOUBLE_COMPLEX}.

Constraints:
- The numerical datatype of obj must be floating-point, and must not be FLA_CONSTANT.

Arguments:
  - obj – An FLA_Obj.

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.

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 × n or m × 1.

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

Arguments:

obj – 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:**
- obj – 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:**
- trans – Indicates whether to perform the check as if $A$ were transposed.
- A – An FLA_Obj.
- B – 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:**
- A – An FLA_Obj.
- B – 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:**
- A – An FLA_Obj.
- B – An FLA_Obj.
void FLA_Obj_extract_real_scalar( FLA_Obj alpha, double* val );

Purpose: Copy the numerical element of real scalar \( \alpha \) into the address specified by \( \text{val} \). If object \( \alpha \) is not a scalar (ie: contains more than one element), the value of the top-left element is copied instead.

Constraints:
- The numerical datatype of \( \alpha \) must be floating-point and real.

Arguments:
- \( \text{alpha} \) – An FLA_Obj representing scalar \( \alpha \).
- \( \text{val} \) – The address of the location to which to store the value of \( \alpha \).

void FLA_Obj_extract_complex_scalar( FLA_Obj alpha, dcomplex* val );

Purpose: Copy the numerical element of complex scalar \( \alpha \) into the address specified by \( \text{val} \). If object \( \alpha \) is not a scalar (ie: contains more than one element), the value of the top-left element is copied instead.

Constraints:
- The numerical datatype of \( \alpha \) must be floating-point and complex.

Arguments:
- \( \text{alpha} \) – An FLA_Obj representing scalar \( \alpha \).
- \( \text{val} \) – The address of the location to which to store the value of \( \alpha \).

void FLA_Obj_extract_real_part( FLA_Obj alpha, FLA_Obj beta );

Purpose: Copy the real component of scalar \( \alpha \) into a real scalar \( \beta \). If \( \alpha \) is real, then its contents are simply copied into \( \beta \).

Constraints:
- The numerical datatype of \( \alpha \) must be floating-point.
- The numerical datatype of \( \beta \) must be real and must not be FLA_CONSTANT.
- The precision of the datatype of \( \alpha \) must be equal to that of \( \beta \).

Arguments:
- \( \text{alpha} \) – An FLA_Obj representing scalar \( \alpha \).
- \( \text{beta} \) – An FLA_Obj representing scalar \( \beta \).

void FLA_Obj_extract_imag_part( FLA_Obj alpha, FLA_Obj beta );

Purpose: Copy the imaginary component of scalar \( \alpha \) into a real scalar \( \beta \). If \( \alpha \) is real, then \( \beta \) is set to zero.

Constraints:
- The numerical datatype of \( \alpha \) must be floating-point.
- The numerical datatype of \( \beta \) must be real and must not be FLA_CONSTANT.
- The precision of the datatype of \( \alpha \) must be equal to that of \( \beta \).

Arguments:
- \( \text{alpha} \) – An FLA_Obj representing scalar \( \alpha \).
- \( \text{beta} \) – An FLA_Obj representing scalar \( \beta \).
FLA_Bool FLA_Obj_buffer_is_null( FLA_Obj obj );

**Purpose:** Check if an object’s data buffer is NULL and therefore currently un-allocated. The function will also return **TRUE** if the object itself has not yet been created.

**Returns:** A boolean value: **TRUE** if either the object is unallocated or the object has a NULL buffer; **FALSE** otherwise.

**Arguments:**
- `obj` – An FLA_Obj.

void* FLA_Submatrix_at( FLA_Datatype datatype, void* buffer, dim_t i, dim_t j, 
    dim_t rs, dim_t cs );

**Purpose:** Compute the starting address of a submatrix whose top-left element is the \((i,j)\) element within the conventional row- or column-major order matrix stored in `buffer` with row and column strides `rs` and `cs`.

**Returns:** The starting address of the requested submatrix.

**Arguments:**
- `datatype` – A constant corresponding to the numerical datatype of the data stored in `buffer`.
- `buffer` – A pointer to a matrix stored in row- or column-major order.
- `i` – The row offset of the requested submatrix.
- `j` – The column offset of the requested submatrix.
- `rs` – The row stride of the matrix stored in `buffer`.
- `cs` – The column stride of the matrix stored in `buffer`.

### 5.2.6 Assignment/Update functions

void FLA_Set( FLA_Obj alpha, FLA_Obj A );

**Purpose:** Set every element in \(A\) to \(\alpha\).

**Constraints:**
- The numerical datatype of \(A\) must not be `FLA_CONSTANT`.
- If \(\alpha\) is not of datatype `FLA_CONSTANT`, then it must match the datatype of \(A\).

**Arguments:**
- `alpha` – An FLA_Obj representing scalar \(\alpha\).
- `A` – An FLA_Obj representing matrix \(A\).
void FLA_Setr( FLA_Uplo, FLA_Obj alpha, FLA_Obj A );

Purpose: Set every element in the upper or lower triangle of $A$ to $\alpha$. The triangle that is modified is determined by $\text{uplo}$.

Constraints:
- The numerical datatype of $A$ must not be FLA_CONSTANT.
- If $\alpha$ is not of datatype FLA_CONSTANT, then it must match the datatype of $A$.

Arguments:
- \text{uplo} – Indicates whether the lower or upper triangle of $A$ is referenced during the operation.
- \text{alpha} – An FLA_Obj representing scalar $\alpha$.
- \text{A} – An FLA_Obj representing matrix $A$.

void FLA_Set_diag( FLA_Obj alpha, FLA_Obj A );

Purpose: Set all diagonal elements of $A$ to $\alpha$.

Constraints:
- The numerical datatype of $A$ must not be FLA_CONSTANT.
- If $\alpha$ is not of datatype FLA_CONSTANT, then it must match the datatype of $A$.

Arguments:
- \text{alpha} – An FLA_Obj representing scalar $\alpha$.
- \text{A} – An FLA_Obj representing matrix $A$.

void FLA_Set_to_identity( FLA_Obj A );

Purpose: Set a matrix to be the identity matrix:

$$ A := \begin{bmatrix} I_n \end{bmatrix} $$

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

Constraints:
- The numerical datatype of $A$ must not be FLA_CONSTANT.
- $A$ must be square.

Arguments:
- \text{A} – An FLA_Obj representing matrix $A$.

void FLA_Add_to_diag( void *alpha, FLA_Obj A );

Purpose: Add $\alpha$ to the diagonal elements of $A$.

Notes: The datatype of $A$ should match the datatype of the value pointed to by $\text{alpha}$.

Constraints:
- The numerical datatype of $A$ must be floating-point and must not be FLA_CONSTANT.
- $\text{alpha}$ must not be NULL.

Arguments:
- \text{alpha} – A pointer to a scalar $\alpha$.
- \text{A} – An FLA_Obj representing matrix $A$. 
void FLA_Shift_diag( FLA_Conj conj, FLA_Obj alpha, FLA_Obj A );
void FLASH_Shift_diag( FLA_Conj conj, FLA_Obj alpha, FLA_Obj A );

**Purpose:** Add $\alpha$ (or $\bar{\alpha}$) to the diagonal elements of $A$.

**Constraints:**
- The numerical datatype of $A$ must be floating-point and must not be `FLA_CONSTANT`.
- If $\alpha$ 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:**
- **conj** – Indicates whether the operation proceeds as if $alpha$ were conjugated.
- **alpha** – An `FLA_Obj` representing scalar $\alpha$.
- **A** – An `FLA_Obj` representing matrix $A$.

void FLA_Scale_diag( FLA_Conj conj, FLA_Obj alpha, FLA_Obj A );

**Purpose:** Scale the diagonal elements of $A$ by $\alpha$ (or $\bar{\alpha}$).

**Constraints:**
- The numerical datatype of $A$ must be floating-point and must not be `FLA_CONSTANT`.
- If $\alpha$ 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:**
- **conj** – Indicates whether the operation proceeds as if $alpha$ were conjugated.
- **alpha** – An `FLA_Obj` representing scalar $\alpha$.
- **A** – An `FLA_Obj` representing matrix $A$.

void FLA_Obj_set_real_part( FLA_Obj alpha, FLA_Obj B );

**Purpose:** Copy the value of real scalar $\alpha$ into the real component of matrix $B$. If $B$ is real, then the value in $\alpha$ is simply copied into all elements of $B$.

**Constraints:**
- The numerical datatype of $\alpha$ must be real.
- The numerical datatype of $B$ must be floating-point and must not be `FLA_CONSTANT`.
- The precision of the datatype of $\alpha$ must be equal to that of $B$.

**Arguments:**
- **alpha** – An `FLA_Obj` representing scalar $\alpha$.
- **B** – An `FLA_Obj` representing scalar $B$.

void FLA_Obj_set_imag_part( FLA_Obj alpha, FLA_Obj beta );

**Purpose:** Copy the value of real scalar $\alpha$ into the imaginary components of matrix $B$. If $B$ is real, then no operation is performed.

**Constraints:**
- The numerical datatype of $\alpha$ must be real.
- The numerical datatype of $B$ must be floating-point and must not be `FLA_CONSTANT`.
- The precision of the datatype of $\alpha$ must be equal to that of $B$.

**Arguments:**
- **alpha** – An `FLA_Obj` representing scalar $\alpha$.
- **B** – An `FLA_Obj` representing scalar $B$. 
5.2.7 Math-related functions

void FLA_Absolute_value( FLA_Obj alpha );

Purpose: Compute the absolute value (or complex norm) of a complex scalar:
\[
\alpha := |\alpha|
\]
where \( \alpha \) is a complex scalar and \( |\alpha| \) is defined as
\[
|\alpha| = \sqrt{\alpha \bar{\alpha}}
\]

Notes: If \( \alpha \) is real, then the operation reduces to
\[
\alpha := |\alpha|
\]

Constraints:
- The numerical datatype of \( \alpha \) must be floating-point and must not be FLA_CONSTANT.

Arguments:
- alpha – An FLA_Obj representing scalar \( \alpha \).

void FLA_Absolute_square( FLA_Obj alpha );

Purpose: Compute the absolute square (or squared norm) of a complex scalar:
\[
\alpha := |\alpha|^2
\]
where \( \alpha \) is a complex scalar and \( |\alpha|^2 \) is defined as
\[
|\alpha|^2 = \alpha \bar{\alpha}
\]

Notes: If \( \alpha \) is real, then the operation reduces to
\[
\alpha := \alpha^2
\]

Constraints:
- The numerical datatype of \( \alpha \) must be floating-point and must not be FLA_CONSTANT.

Arguments:
- alpha – An FLA_Obj representing scalar \( \alpha \).
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.

**Constraints:**
- The numerical datatype of \( A \) must be floating-point, and must not be \texttt{FLA_CONSTANT}.

**Imp. Notes:** This function uses an external implementation of the level-1 BLAS routine \texttt{*scal()}.

**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.

**Constraints:**
- The numerical datatype of \( A \) must be floating-point, and must not be \texttt{FLA_CONSTANT}.

**Imp. Notes:** This function uses an external implementation of the level-1 BLAS routine \texttt{*scal()}.

**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.

**Constraints:**
- The numerical datatype of \( A \) must be floating-point, and must not be \texttt{FLA_CONSTANT}.
- \( A \) must be square.

**Imp. Notes:** This function uses an external implementation of the level-1 BLAS routine \texttt{?swap()}.

**Arguments:**
- \( A \) – An FLA_Obj representing matrix \( A \).
### void FLA_Invert( FLA_Conj conj, FLA_Obj x );

**Purpose:** Invert each element of a vector:

\[ \chi_i := \chi_i^{-1} \]

where \( \chi_i \) is the \( i \)th element of vector \( x \). If \( \text{conj} \) is \text{FLA_CONJUGATE}, then each element is also conjugated:

\[ \chi_i := \bar{\chi}_i^{-1} \]

**Constraints:**
- The numerical datatype of \( \alpha \) must be floating-point and must not be \text{FLA_CONSTANT}.
- \( x \) must be a vector (or a scalar).

**Arguments:**
- \text{conj} – Indicates whether to compute the conjugate of the inverse.
- \text{alpha} – An \text{FLA_Obj} representing scalar \( \alpha \).

### void FLA_Max_abs_value( FLA_Obj A, FLA_Obj amax );

**Purpose:** Find the maximum absolute value of all elements of a matrix:

\[ A_{\text{max}} := \max_{ij} |\alpha_{ij}| \]

where \( A_{\text{max}} \) is a scalar and \( \alpha_{ij} \) is the \((i,j)\) element of general matrix \( A \). Upon completion, the maximum absolute value \( A_{\text{max}} \) is stored to \( \text{amax} \).

**Notes:** If \( A \) is complex, then \(|\alpha_{ij}|\) is evaluated as the complex norm, which, for any complex number \( z \), is defined as

\[ |z| = |x + iy| = \sqrt{x^2 + y^2} \]

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 \text{FLA_CONSTANT}.
- The numerical datatype of \( A_{\text{max}} \) must be real and must not be \text{FLA_CONSTANT}.
- The precision of the datatype of \( A_{\text{max}} \) must be equal to that of \( A \).

**Arguments:**
- \( A \) – An \text{FLA_Obj} representing matrix \( A \).
- \( \text{amax} \) – An \text{FLA_Obj} representing scalar \( A_{\text{max}} \).
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 \( \alpha_{ij} \) and \( \beta_{ij} \) 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 \(\text{FLA\_CONSTANT}\).
- The dimensions of \(A\) and \(B\) must be conformal.

**Arguments:**
- \(A\) – An \(\text{FLA\_Obj}\) representing matrix \(A\).
- \(B\) – An \(\text{FLA\_Obj}\) 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 \(\alpha\), \(\beta\), and \(\gamma\) are scalars.

**Constraints:**
- The numerical datatype of \(\alpha\), \(\beta\), and \(\gamma\) must be floating-point. Also, the datatype of \(\gamma\) must not be \(\text{FLA\_CONSTANT}\).

**Arguments:**
- \(\alpha\) – An \(\text{FLA\_Obj}\) representing scalar \(\alpha\).
- \(\beta\) – An \(\text{FLA\_Obj}\) representing scalar \(\beta\).
- \(\gamma\) – An \(\text{FLA\_Obj}\) representing scalar \(\gamma\).

void FLA_Negate( FLA_Obj A );

**Purpose:** Negate a matrix:

\[ A := -A \]

where \(A\) is a general matrix.

**Constraints:**
- The numerical datatype of \(A\) must be floating-point and must not be \(\text{FLA\_CONSTANT}\).

**Imp. Notes:** This function uses an external implementation of the level-1 BLAS routine \(*scal()\).
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:

\[
\|A\|_1 := \max_{j} \sum_{i=0}^{n-1} |α_{ij}|
\]

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 \(\text{norm1}\).

**Constraints:**
- The numerical datatype of \(A\) must be floating-point and must not be \text{FLA_CONSTANT}.
- The numerical datatype of \(\text{norm1}\) must be real and must not be \text{FLA_CONSTANT}.
- The precision of the datatype of \(\text{norm1}\) must be equal to that of \(A\).

**Imp. Notes:** This function uses an external implementation of the level-1 BLAS routine \(*asum()\).

**Arguments:**
- \(A\) – An FLA_Obj representing matrix \(A\).
- \(\text{norm1}\) – 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:

\[
\|A\|_∞ := \max_{i} \sum_{j=0}^{n-1} |α_{ij}|
\]

where \(\|A\|_∞\) is a scalar and \(α_{ij}\) is the \((i, j)\) element of general matrix \(A\). Upon completion, the maximum absolute row sum norm \(\|A\|_∞\) is stored to \(\text{norminf}\).

**Constraints:**
- The numerical datatype of \(A\) must be floating-point and must not be \text{FLA_CONSTANT}.
- The numerical datatype of \(\text{norminf}\) must be real and must not be \text{FLA_CONSTANT}.
- The precision of the datatype of \(\text{norminf}\) must be equal to that of \(A\).

**Imp. Notes:** This function uses an external implementation of the level-1 BLAS routine \(*asum()\).

**Arguments:**
- \(A\) – An FLA_Obj representing matrix \(A\).
- \(\text{norminf}\) – An FLA_Obj representing scalar \(\|A\|_∞\).
void FLA_Norm_frob( FLA_Obj A, FLA_Obj norm );

Purpose: Compute the Frobenius norm of a matrix:

\[ \|A\|_F := \sqrt{\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} |\alpha_{ij}|^2} \]

where \(\|A\|_F\) is a scalar and \(\alpha_{ij}\) is the \((i, j)\) element of general matrix \(A\). Upon completion, the Frobenius norm \(\|A\|_F\) is stored to \(\text{norm}\).

Constraints:
- The numerical datatype of \(A\) must be floating-point and must not be \(\text{FLA_CONSTANT}\).
- The numerical datatype of \(\text{norm}\) must be real and must not be \(\text{FLA_CONSTANT}\).
- The precision of the datatype of \(\text{norm}\) must be equal to that of \(A\).

Arguments:

- \(A\) – An FLA_Obj representing matrix \(A\).
- \(\text{norm}\) – An FLA_Obj representing scalar \(\|A\|_F\).

void FLA_Scal_elemwise( FLA_Trans trans, FLA_Obj A, FLA_Obj B );

Purpose: Perform an element-wise scale of matrix \(B\) by matrix \(A\):

\[ \beta_{ij} := \alpha_{ij} \beta_{ij} \quad \forall i, j \in \{0,\ldots,m-1\}, \{0,\ldots,n-1\} \]

where \(\alpha_{ij}\) and \(\beta_{ij}\) are the \((i, j)\) elements within matrices \(A\) and \(B\), respectively. The \(\text{trans}\) argument allows the computation to proceed as if \(A\) were conjugated and/or transposed.

Constraints:
- The numerical datatypes of \(A\) and \(B\) must be identical and floating-point, and must not be \(\text{FLA_CONSTANT}\).
- If \(A\) and \(B\) are vectors, then their lengths must be equal. Otherwise, if \(\text{trans}\) equals \(\text{FLA_NO_TRANSPOSE}\) or \(\text{FLA_CONJ_NO_TRANSPOSE}\), then the dimensions of \(A\) and \(B\) must be conformal; otherwise, if \(\text{trans}\) equals \(\text{FLA_TRANSPOSE}\) or \(\text{FLA_CONJ_TRANSPOSE}\), then the dimensions of \(A^T\) and \(B\) must be conformal.

Arguments:

- \(\text{trans}\) – 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 \(B\).
FLA_Error FLA_Sqrt( FLA_Obj alpha );

**Purpose:** Compute the square root of a scalar:

\[ \alpha := \sqrt{\alpha} \]

where \( \alpha \) is a positive real scalar.

**Returns:** FLA_SUCCESS if \( \alpha \) is non-negative on entry; otherwise FLA_FAILURE.

**Constraints:**
- The numerical datatype of \( \alpha \) must be real and must not be FLA_CONSTANT.

**Arguments:**
- \( \text{alpha} \) – An FLA_Obj representing scalar \( \alpha \).

---

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.

**Constraints:**
- The numerical datatype of \( A \) must be floating-point, and must not be FLA_CONSTANT.

**Imp. Notes:** The random numbers obtained are unseeded and therefore deterministic. Random numbers are obtained from the C standard library function \texttt{rand()}, scaled by \texttt{RAND\_MAX}, and shifted to result in a uniform distribution over the interval \([-1.0, 1.0]\).

**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, i.e.: 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.

Constraints:
- The numerical datatype of $A$ must be floating-point, and must not be `FLA_CONSTANT`.

Imp. Notes: The random numbers obtained are unseeded and therefore deterministic. Random numbers are obtained from the C standard library function `rand()`, scaled by `RAND_MAX`, and shifted to result in a uniform distribution over the interval $[-1.0, 1.0]$.

Arguments:
- `uplo` – Indicates whether the lower or upper triangle of $A$ is stored during the operation. This argument has no net effect on the operation.
- $A$ – An `FLA_Obj` representing matrix $A$.

void FLA_Random_symm_matrix( FLA_Uplo uplo, FLA_Obj A );

Purpose: Overwrite a matrix $A$ with a random symmetric matrix, i.e.: a matrix $A$ such that

$$ A = A^T $$

The `uplo` argument indicates whether the lower or upper triangle of $A$ is initially stored by the operation.

Constraints:
- The numerical datatype of $A$ must be floating-point, and must not be `FLA_CONSTANT`.

Imp. Notes: The random numbers obtained are unseeded and therefore deterministic. Random numbers are obtained from the C standard library function `rand()`, scaled by `RAND_MAX`, and shifted to result in a uniform distribution over the interval $[-1.0, 1.0]$.

Imp. Notes: Currently, the value of `uplo` does not determine which triangle is written to. In either case, the specified triangle is randomized and then transpose-transposed into the other. However, a future implementation of `FLA_Random_symm_matrix()` may only store to the triangle specified by `uplo`.

Arguments:
- `uplo` – Indicates whether the lower or upper triangle of $A$ is stored during the operation. This argument has no net effect on the operation.
- $A$ – An `FLA_Obj` 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 if $A$ is real, or a random Hermitian positive definite matrix if $A$ is complex. The $\text{uplo}$ argument indicates whether the lower or upper triangle of $A$ is stored by the operation.

Constraints:
- The numerical datatype of $A$ must be floating-point, and must not be $\text{FLA_CONSTANT}$.

Imp. Notes: The random numbers obtained are unseeded and therefore deterministic. Random numbers are obtained from the C standard library function $\text{rand}()$, scaled by $\text{RAND_MAX}$, and shifted to result in a uniform distribution over the interval $[−1.0, 1.0)$.

Imp. Notes: If $\text{uplo}$ is $\text{FLA_LOWER_TRIANGULAR}$, then the random matrix $A$ is computed as

$$ A := RR^H $$

where $R$ is a lower triangular. Otherwise, if $\text{uplo}$ is $\text{FLA_UPPER_TRIANGULAR}$, the matrix is computed as

$$ A := R^H R $$

where $R$ is a upper triangular. In either case, $R$ is generated by $\text{FLA_Random_tri_matrix()}$ to have a unit diagonal.

Arguments:

- $\text{uplo}$ – Indicates whether the lower or upper triangle of $A$ is stored during the operation. This argument is currently ignored.
- $A$ – An $\text{FLA_Obj}$ 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 $\text{uplo}$ argument indicates whether $A$ will be lower or upper triangular. The off-diagonal elements of the triangle specified by $\text{uplo}$ are normalized by the order of $A$ (for numerical reasons), and the opposite triangle is explicitly set to zero. The $\text{diag}$ argument indicates how the diagonal of the matrix is set: $\text{FLA_ZERO_DIAG}$ will set all diagonal entries to zero, $\text{FLA_UNIT_DIAG}$ will set diagonal entries to one, and $\text{FLA_NONUNIT_DIAG}$ will assign the diagonal random values.

Constraints:
- The numerical datatype of $A$ must be floating-point, and must not be $\text{FLA_CONSTANT}$.

Imp. Notes: The random numbers obtained are unseeded and therefore deterministic. Random numbers are obtained from the C standard library function $\text{rand}()$, scaled by $\text{RAND_MAX}$, and shifted to result in a uniform distribution over the interval $[−1.0, 1.0)$.

Arguments:

- $\text{uplo}$ – Indicates whether the lower or upper triangle of $A$ is stored during the operation. This argument is currently ignored.
- $\text{diag}$ – Indicates whether the diagonal of $A$ is set to be zero, unit, or non-unit (random).
- $A$ – An $\text{FLA_Obj}$ representing matrix $A$. 
void FLA_Random_unitary_matrix( FLA_Obj A );

**Purpose:** Overwrite a matrix $A$ with a random unitary matrix.

**Imp. Notes:** $FLA\_Random\_unitary\_matrix()$ forms a random unitary matrix by first creating a random matrix via $FLA\_Random\_matrix()$ and then performing a QR factorization on this matrix via $FLA\_QR\_UT()$. The Householder transforms associated with the factorization are then applied to the identity matrix in such a way that minimizes the number of computations that must take place.

**Constraints:**
- The numerical datatype of $A$ must be floating-point, and must not be $FLA\_CONSTANT$.

**Imp. Notes:** The random numbers obtained are unseeded and therefore deterministic. Random numbers are obtained from the C standard library function $\text{rand}()$, scaled by $\text{RAND\_MAX}$, and shifted to result in a uniform distribution over the interval $[−1.0, 1.0)$.

**Arguments:**
- $A$ – An $FLA\_Obj$ 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.

**Constraints:**
- The numerical datatype of $A$ must be floating-point, and must not be $FLA\_CONSTANT$.
- $A$ must be square.

**Imp. Notes:** This function uses an external implementation of the level-1 BLAS routine $\text{?copy}()$.

**Arguments:**
- $uplo$ – Indicates whether the lower or upper triangle of $A$ is preserved and transposed into the other triangle.
- $A$ – An $FLA\_Obj$ representing matrix $A$.

void FLA_Hermitianize( FLA_Uplo uplo, FLA_Obj A );

**Purpose:** Transform a general complex matrix $A$ into a Hermitian matrix by conjugate-transposing 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 and conjugate-transposed.

**Notes:** If $A$ is real, then $FLA\_Hermitianize()$ behaves exactly as $FLA\_Symmetrize()$.

**Constraints:**
- The numerical datatype of $A$ must be floating-point, and must not be $FLA\_CONSTANT$.
- $A$ must be square.

**Imp. Notes:** This function uses external implementations of the level-1 BLAS routines $\text{?copy}()$ and $\text{*scal}()$.

**Arguments:**
- $uplo$ – Indicates whether the lower or upper triangle of $A$ is preserved and conjugate-transposed into the other triangle.
- $A$ – An $FLA\_Obj$ representing matrix $A$. 
5. User-level Application Programming Interfaces

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 \( \text{uplo} \) argument indicates which triangle of \( A \) is preserved. The \( \text{diag} \) argument indicates whether to change the diagonal of the matrix; \( \text{FLA}_\text{ZERO}_\text{DIAG} \) will set all diagonal entries to zero, \( \text{FLA}_\text{UNIT}_\text{DIAG} \) will set diagonal entries to one, and \( \text{FLA}_\text{NONUNIT}_\text{DIAG} \) will leave the diagonal unchanged.

**Constraints:**
- The numerical datatype of \( A \) must be floating-point, and must not be \( \text{FLA}_\text{CONSTANT} \).
- \( A \) must be square.

**Arguments:**
- \( \text{uplo} \) – Indicates whether the lower or upper triangle of \( A \) is preserved.
- \( \text{diag} \) – Indicates whether the diagonal of \( A \) is set to be zero, unit, or left unchanged.
- \( A \) – An \( \text{FLA}_\text{Obj} \) representing matrix \( A \).

### 5.2.8 Miscellaneous functions

**unsigned int FLA_Check_error_level( void );**

**Purpose:** Query the current level of internal error and parameter checking in \( \text{libflame} \). Valid return values are \( \text{FLA}_\text{FULL}_\text{ERROR}_\text{CHECKING} \), \( \text{FLA}_\text{MIN}_\text{ERROR}_\text{CHECKING} \), and \( \text{FLA}_\text{NO}_\text{ERROR}_\text{CHECKING} \).

**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. Use reduced error checking at your own risk, and be aware that your application may exhibit nondeterministic behavior if an error does arise.

**Returns:** An unsigned integer: \( \text{FLA}_\text{FULL}_\text{ERROR}_\text{CHECKING} \) if error and parameter checking is fully enabled; \( \text{FLA}_\text{MIN}_\text{ERROR}_\text{CHECKING} \) if minimal error and parameter checking is enabled; \( \text{FLA}_\text{NO}_\text{ERROR}_\text{CHECKING} \) 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 \( \text{libflame} \) to \( \text{level} \). Valid values for \( \text{level} \) are \( \text{FLA}_\text{FULL}_\text{ERROR}_\text{CHECKING} \), \( \text{FLA}_\text{MIN}_\text{ERROR}_\text{CHECKING} \), and \( \text{FLA}_\text{NO}_\text{ERROR}_\text{CHECKING} \). 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: \( \text{FLA}_\text{FULL}_\text{ERROR}_\text{CHECKING} \) if error and parameter checking was fully enabled; \( \text{FLA}_\text{MIN}_\text{ERROR}_\text{CHECKING} \) if minimal error and parameter checking was enabled; \( \text{FLA}_\text{NO}_\text{ERROR}_\text{CHECKING} \) if error and parameter checking was completely disabled.

**Arguments:**
- \( \text{level} \) – The value corresponding to the desired error checking level.
FLA_Bool FLA_Memory_leak_counter_set( FLA_Bool new_status );

Purpose: Set whether the memory leak counter is enabled or disabled. When enabled, the internal memory allocation functions FLA_malloc() and FLA_free() increment and decrement, respectively, an internal counter to keep track of outstanding number of memory regions still allocated. A positive number indicates a conventional memory leak while a negative number suggests that at least one region of allocated memory was freed more than once.\footnote{This latter kind of memory leak is more difficult to encounter since most modern C library implementations will disallow freeing the same memory address twice, usually by posting a fatal error.} If the counter is enabled upon entering FLA_Finalize(), the counter value is output to standard error. The function returns the previous status of the memory leak counter, regardless of whether new_status actually caused a change in the status.

Notes: If multithreading was enabled at runtime, the update of the internal memory counter is protected by a lock. Some applications that are intensive in object creation and destruction may wish to disable the memory leak counter to ensure maximum performance. Of course, this is only advisable if you are confident that your application has no existing memory leaks.

Returns: A boolean value: TRUE if the memory leak counter is currently enabled; FALSE otherwise.

Arguments:

new_status – A boolean value that either enables (TRUE) or disables (FALSE) libflame memory leak counter.

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 outputing error messages just before the library aborts. However, it is general enough to be used by application programmers as well.

Arguments:

message – A pointer to a string containing the message to output.
filename – A pointer to a string containing the name of the file. This is typically obtained via the C preprocessor macro \_FILE\_
line – An unsigned integer containing the line number that contained the invocation of FLA_Print_message(). This is typically obtained via the C preprocessor macro \_LINE\_

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 breakpoints.
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 (i.e. when `FLA_ENABLE_WINDOWS_BUILD` is defined) `FLA_Clock()` is implemented in terms of `QueryPerformanceCounter()` and `QueryPerformanceFrequency()`.

5.2.9 Advanced query routines

dim_t FLA_Obj_row_offset( FLA_Obj obj );

Purpose: Query the row offset of an object view `obj`.

Notes: This routine should only be used by advanced users and developers.

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

Arguments:

`obj` — An FLA_Obj.

dim_t FLA_Obj_col_offset( FLA_Obj obj );

Purpose: Query the column offset of an object view `obj`.

Notes: This routine should only be used by advanced users and developers.

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

Arguments:

`obj` — An FLA_Obj.

dim_t FLA_Obj_base_length( FLA_Obj obj );

Purpose: Query the number of rows in the base object of `obj`. In other words, query the number of rows in the object `obj` as it was originally allocated.

Notes: This routine should only be used by advanced users and developers.

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

Arguments:

`obj` — An FLA_Obj.
dim_t FLA_Obj_base_width( FLA_Obj obj );

**Purpose:** Query the number of columns in the base object of obj. In other words, query the number of columns in the object obj as it was originally allocated.

**Notes:** This routine should only be used by advanced users and developers.

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

**Arguments:**
- `obj` – An FLA_Obj.

void* FLA_Obj_base_buffer( FLA_Obj obj );

**Purpose:** Query the starting address of the base object underlying numerical data buffer. The address of the object is the address that was returned by FLA_malloc() when the object was created and not necessarily the same as the starting address of the object’s view.

**Notes:** Since the address returned by FLA_Obj_base_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().

**Notes:** This routine should only be used by advanced users and developers.

**Returns:** A pointer of type `void*`.

**Arguments:**
- `obj` – An FLA_Obj.

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:**
- `datatype` – An FLA_Datatype value.

FLA_Elemtype 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:**
- `obj` – An FLA_Obj.
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:
- obj – An FLA_Obj.

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 top and bottom side views where the side indicated by side has mb rows.

Returns: FLA_SUCCESS

Arguments:
- A – An FLA_Obj.
- AT – A pointer to an uninitialized FLA_Obj view into the top side of A.
- AB – A pointer to an uninitialized FLA_Obj view into the bottom side of A.
- mb – The number of rows to extract.
- side – The side from 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 – Pointers to uninitialized FLA_Obj structures.
- mb – The number of rows to extract.
- side – The side from which to extract mb rows.
5.3. Managing Views

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 \(2 \times 1\) partitioning of matrix \(A\) by moving the boundaries so that \(A_1\) is shifted to the side indicated by \(\text{side}\).

Returns: FLA_SUCCESS

Arguments:

\(\text{AT, AB}\)
(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.

\(\text{A0...A2}\)
– FLA_Obj structures that were repartitioned via FLA_Part_2x1_to_3x1().

\(\text{side}\)
– The side to which to shift the \(m_b\) 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 left and right side views where the side indicated by \(\text{side}\) has \(nb\) columns.

Returns: FLA_SUCCESS

Arguments:

\(\text{A}\)
– An FLA_Obj.

\(\text{AL}\)
(on entry) – A pointer to an uninitialized FLA_Obj.
(on exit) – A pointer to an FLA_Obj view into the left side of \(A\).

\(\text{AR}\)
(on entry) – A pointer to an uninitialized FLA_Obj.
(on exit) – A pointer to an FLA_Obj view into the right side of \(A\).

\(\text{nb}\)
– The number of columns to extract.

\(\text{side}\)
– 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 \(1 \times 2\) partitioning of matrix \(A\) into a \(1 \times 3\) partitioning where \(nb\) columns are split from the side indicated by \(\text{side}\).

Returns: FLA_SUCCESS

Arguments:

\(\text{AL, AR}\)
– FLA_Obj structures that were partitioned via FLA_Part_1x2().

\(\text{A0...A2}\)
(on entry) – Pointers to uninitialized FLA_Obj structures.
(on exit) – Pointers to FLA_Obj views into AL and AR.

\(\text{nb}\)
– The number of columns to extract.

\(\text{side}\)
– The side from which to extract \(nb\) columns.
5. User-level Application Programming Interfaces

FLA_Error FLA_Cont_with_1x3_to_1x2( FLA_Obj* AL, FLA_Obj* AR, FLA_Obj A0, FLA_Obj A1, FLA_Obj A2, FLA_Side side );

**Purpose:** Update the $1 \times 2$ 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 $A_1$.

### 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 four quadrant views 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 $2 \times 2$ partitioning of matrix $A$ into a $3 \times 3$ 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 A11.

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 $2 \times 2$ 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 A11.
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 AT and AB.

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.
5.4. FLASH

5.4.1 Motivation

Traditionally, dense matrices are stored in column-major order (or, alternatively, in row-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 \( l_{\text{dim}j} \), where \( l_{\text{dim}} \) 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 [21, 22]. 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 [21].

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.\(^4\) 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 [29]. 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 matricies, 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.

\(^4\)Presumably, 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. Individual blocks almost always use a column-major storage scheme.

- **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 object\(^5\). 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.

\(^5\)Currently, 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 5.6.
5.4.4 Object creation and destruction

```c
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 \( \text{datatype} \). Furthermore, \( H \) will have a hierarchical depth of \( \text{depth} \) and the value in \( \text{b}_\text{mn}[i] \) will specify the square block sizes for the \( i+1 \)th level of the hierarchy. Only the first \( \text{depth} \) values of \( \text{b}_\text{mn} \) will be referenced.

**Notes:** If \( \text{depth} > 0 \), 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 \( \text{depth} = 0 \), 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.
- \( \text{datatype} \) may not be FLA_CONSTANT.
- The pointer arguments \( \text{b}_\text{mn} \) and \( H \) must not be NULL.
- Each of the first \( \text{depth} \) values in \( \text{b}_\text{mn} \) must be greater than zero.

**Imp. Notes:** \( \text{FLASH}_\text{Obj}_\text{create}() \) creates hierarchical objects with leaf and non-leaf nodes in column-major order.

**Arguments:**
- \( \text{datatype} \) – 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.
- \( \text{depth} \) – The number of levels to create in the hierarchy of \( H \).
- \( \text{b}_\text{mn} \) – A pointer to an array of \( \text{depth} \) values to be used as block sizes in creating the matrix hierarchy of \( H \).
- \( H \) (on entry) – A pointer to an uninitialized FLA_Obj.
- \( H \) (on exit) – A pointer to a new hierarchical FLA_Obj parameterized by \( \text{datatype}, m, n, \text{depth}, \text{b}_\text{mn} \).
5.4. FLASH

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 \texttt{datatype}. Furthermore, \( H \) will have a hierarchical depth of \texttt{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 \texttt{depth} values of \( b.m \) and \( b.n \) will be referenced.

Notes: If \texttt{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 blockizes at each hierarchical level combined with the dimensions of the overall hierarchical matrix. If \texttt{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.
- \texttt{datatype} may not be \texttt{FLA_CONSTANT}.
- The pointer arguments \( b.m \), \( b.n \), and \( H \) must not be \texttt{NULL}.
- Each of the first \texttt{depth} values in \( b.m \) and \( b.n \) must be greater than zero.

Imp. Notes: \texttt{FLASH_Obj_create_ext()} creates hierarchical objects with leaf and non-leaf nodes in column-major order.

Arguments:

\texttt{datatype} – A constant corresponding to the numerical datatype requested.
\texttt{m} – The number of rows to be created in new object.
\texttt{n} – The number of columns to be created in the new object.
\texttt{depth} – The number of levels to create in the hierarchy of \( H \).
\texttt{b.m} – A pointer to an array of \texttt{depth} values to be used as the row dimensions of the blocksizes needed when creating the matrix hierarchy of \( H \).
\texttt{b.n} – A pointer to an array of \texttt{depth} values to be used as the column dimensions of the blocksizes needed when creating the matrix hierarchy of \( H \).
\texttt{H} (on entry) – A pointer to an uninitialized FLA_Obj.
(on exit) – A pointer to a new hierarchical FLA_Obj parameterized by \texttt{datatype}, \texttt{m}, \texttt{n}, \texttt{depth}, \texttt{b.m}, and \texttt{b.n}. 
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:

trans – Indicates whether to create the object pointed to by H_new with trans-
posed dimensions.

H_cur – An existing hierarchical FLA_Obj.

H_new (on entry) – A pointer to an uninitialized FLA_Obj.

H_new (on exit) – A pointer to a new hierarchical FLA_Obj parameterized by the
datatype, dimensions, depth, and blocksizes of H_cur.

void FLASH_Obj_create_copy_of( 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. After H_new is created, it is initialized with the contents of H_cur, applying a transposition according to trans.

Constraints: • trans may not be FLA_CONJ_TRANSPOSE or FLA_CONJ_NO_TRANSPOSE.

Arguments:

trans – Indicates whether to create the object pointed to by H_new with trans-
posed dimensions.

H_cur – An existing hierarchical FLA_Obj.

H_new (on entry) – A pointer to an uninitialized FLA_Obj.

H_new (on exit) – A pointer to a new hierarchical FLA_Obj parameterized by the
datatype, dimensions, depth, and blocksizes of H_cur with its numeri-
cal contents identical to that of H_cur.

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:

H (on entry) – A pointer to a valid hierarchical FLA_Obj.

H (on exit) – A pointer to an uninitialized FLA_Obj.
5.4.5 Interfacing with flat matrix objects

```c
void FLASH_Obj_create_hier_conf_to_flat( FLA_Trans trans, FLA_Obj F, dim_t depth,
    dim_t* b_mn, 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 blocksize arguments `depth` and `b_mn`. 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_mn` and `H` must not be `NULL`.
- Each of the first `depth` values in `b_mn` 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_mn` – A pointer to an array of `depth` values to be used as blocksize in creating the matrix hierarchy of $H$.
- `H` – A pointer to a new hierarchical `FLA_Obj` parameterized by the datatype and dimensions of $F$, `depth`, and `b_mn`.

(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 `b_mn`. 
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 
and blocksize arguments depth, b_m, and b_n. 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.

void FLASH_Obj_create_hier_copy_of_flat( FLA_Obj F, dim_t depth, 
    dim_t* b_mn, 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 blocksize arguments depth and 
b_mn.

Constraints:

• 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:

F – An existing flat FLA_Obj representing matrix F.
depth – The number of levels to create in the hierarchy of H.
b_mn – 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 b_mn, and which contains the contents of the flat matrix F.
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 blocksize arguments \( \text{depth} \), \( \text{b}_m \), and \( \text{b}_n \).

Constraints:

- The pointer arguments \( \text{b}_m \), \( \text{b}_n \), and \( H \) must not be NULL.
- Each of the first \( \text{depth} \) values in \( \text{b}_m \) and \( \text{b}_n \) must be greater than zero.

Arguments:

\[
\begin{align*}
F & \quad \text{An existing flat FLA_Obj representing matrix } F. \\
\text{depth} & \quad \text{The number of levels to create in the hierarchy of } H. \\
\text{b}_m & \quad \text{A pointer to an array of } \text{depth} \text{ values to be used as the row dimensions of the blocksizes needed when creating the matrix hierarchy of } H. \\
\text{b}_n & \quad \text{A pointer to an array of } \text{depth} \text{ values to be used as the column dimensions of the blocksizes needed when creating the matrix hierarchy of } H. \\
H & \quad \text{(on entry)} \quad \text{A pointer to an uninitialized FLA_Obj.} \\
& \quad \text{(on exit)} \quad \text{A pointer to a new hierarchical FLA_Obj parameterized by the datatype and dimensions of } F, \text{depth}, \text{b}_m \text{, and } \text{b}_n \text{, and which contains the contents of the flat matrix } F.
\end{align*}
\]

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 \text{FLA_TRANSPOSE} for the \text{trans} argument.

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

Constraints:

- \text{trans} may not be \text{FLA_CONJ_TRANSPOSE} or \text{FLA_CONJ_NO_TRANSPOSE}.
- The pointer argument \( F \) must not be NULL.

Arguments:

\[
\begin{align*}
\text{trans} & \quad \text{Indicates whether to create the object pointed to by } F \text{ with transposed dimensions.} \\
H & \quad \text{An existing hierarchical FLA_Obj representing matrix } H. \\
F & \quad \text{(on entry)} \quad \text{A pointer to an uninitialized FLA_Obj.} \\
& \quad \text{(on exit)} \quad \text{A pointer to a new flat FLA_Obj parameterized by the datatype and dimensions of } F.
\end{align*}
\]
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.
- $F$ (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 $F$.

void FLASH_Copy_buffer_to_hier( dim_t m, dim_t n, void* F, dim_t rs, dim_t cs, dim_t i, dim_t j, FLA_Obj H );

**Purpose:** Copy the contents of an conventional column-major matrix $F$ with row and column strides $rs$ and $cs$ 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 $i + m \times j + n$.
- $rs$ and $cs$ must either both be zero, or non-zero. Also, one of the two strides must be equal to 1. If $rs$ is equal to 1, then $cs$ must be at least $m$; otherwise, if $cs$ is equal to 1, then $rs$ must be at least $n$.
- 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 $F$.
- $rs$ – The row stride of $F$.
- $cs$ – The column stride 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 $H$. 
void FLASH_Copy_hier_to_buffer( dim_t i, dim_t j, FLA_Obj H,
    dim_t m, dim_t n, void* F, dim_t rs, dim_t cs );

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 row and column strides $rs$ and $cs$, 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 $i + m \times j + n$.
- $rs$ and $cs$ must either both be zero, or non-zero. Also, one of the two strides must be equal to 1. If $rs$ is equal to 1, then $cs$ must be at least $m$; otherwise, if $cs$ is equal to 1, then $rs$ must be at least $n$.
- 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 FLA_Obj 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 $F$.
- $rs$ – The row stride of $F$.
- $cs$ – The column stride of $F$.

void FLASH_Copy_flat_to_hier( 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 FLA_Obj 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 FLA_Obj representing matrix $H$.

void FLASH_Copy_hier_to_flat( 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 FLA_Obj representing matrix $H$.
- $F$ – A flat FLA_Obj representing matrix $F$. 
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 $m \times n$.

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

Imp. Notes: This function is currently implemented as:
FLASH_Copy_subobject_to_object( F, 0, 0, H );

Arguments:
- $F$ – A flat FLA_Obj representing matrix $F$.
- $H$ – A hierarchical FLA_Obj representing matrix $H$.

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 $m \times n$.

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

Imp. Notes: This function is currently implemented as:
FLASH_Copy_object_to_subobject( 0, 0, F, H );

Arguments:
- $H$ – A hierarchical FLA_Obj representing matrix $H$.
- $F$ – A flat FLA_Obj representing matrix $F$. 
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* b_mn, 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 $b_{mn}$ and $H$ must not be NULL.
• Each of the first $depth$ values in $b_{mn}$ must be greater than zero.

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datatype</td>
<td>A constant corresponding to the numerical datatype requested.</td>
</tr>
<tr>
<td>m</td>
<td>The number of rows to be created in new object.</td>
</tr>
<tr>
<td>n</td>
<td>The number of columns to be created in the new object.</td>
</tr>
<tr>
<td>depth</td>
<td>The number of levels of hierarchy in the object that represents matrix $H$.</td>
</tr>
<tr>
<td>b_{mn}</td>
<td>A pointer to an array of $depth$ values to be used as blocksizes in creating the matrix hierarchy of $H$.</td>
</tr>
<tr>
<td>H</td>
<td>(on entry) A pointer to an uninitialized FLA_Obj.</td>
</tr>
<tr>
<td>(on exit)</td>
<td>A pointer to a new, bufferless hierarchical FLA_Obj parameterized by $m$, $n$, $depth$, $b_{mn}$, and $datatype$.</td>
</tr>
</tbody>
</table>
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:  
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.  
depth – The number of levels of hierarchy in the object that represents matrix $H$.  
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, bufferless hierarchical FLA_Obj parameterized by 
m, n, depth, b_m, b_n, and datatype.

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:  
H (on entry) – A pointer to a valid hierarchical FLA_Obj.  
(on exit) – A pointer to an uninitialized FLA_Obj.
void FLASH_Obj_attach_buffer( void* buffer, dim_t rs, dim_t cs, FLA_Obj* H );

**Purpose:** Attach a user-allocated region of memory to a hierarchical object that was created with \texttt{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 row and column strides \( rs \) and \( cs \), 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 \texttt{FLA\_Obj} with an attached buffer, you should free it with \texttt{FLASH\_Obj\_free\_without\_buffer()}. However, you are still responsible for freeing the memory pointed to by \texttt{buffer} using \texttt{free()} or whatever memory deallocation function your system provides.

**Constraints:**
- \( rs \) and \( cs \) must either both be zero, or non-zero. Also, one of the two strides must be equal to 1. If \( rs \) is equal to 1, then \( cs \) must be at least \( m \); otherwise, if \( cs \) is equal to 1, then \( rs \) must be at least \( n \).

**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 (row or column stride) of the conventional matrix. Therefore, we highly encourage users to create hierarchical matrices one of two other ways:
  - Use \texttt{FLASH\_Obj\_create()} and then initialize the matrix elements incrementally, one submatrix at a time, with \texttt{FLASH\_Copy\_flat\_to\_hier()} or \texttt{FLASH\_Copy\_buffer\_to\_hier()}.
  - Use \texttt{FLASH\_Obj\_create\_hier\_copy\_of\_flat()} to create a hierarchical object and initialize it with the contents of an existing flat object.

**Arguments:**
- \texttt{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 \texttt{malloc()}, but the memory may also be statically allocated.
- \texttt{rs} – The row stride of the matrix stored conventionally in \texttt{buffer}.
- \texttt{cs} – The column stride of the matrix stored conventionally in \texttt{buffer}.
- \texttt{H} – A pointer to a valid hierarchical \texttt{FLA\_Obj} that was created without a buffer.
- \texttt{(on entry)} – A pointer to a valid hierarchical \texttt{FLA\_Obj} that encapsulates the data in \texttt{buffer}.
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 hierarchy.

Notes: Using FLASH_Obj_datatype() on a flat matrix will return the same value as FLA_Obj_datatype().

Returns: A constant of type FLA_Datatype.

Arguments:

H – An FLA_Obj representing matrix $H$.

dim_t FLASH_Obj_scalar_length( FLA_Obj H );

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

Notes: Using FLASH_Obj_scalar_length() on a flat matrix will always return the correct value (i.e., the same as that returned by FLA_Obj_length()). However, using FLA_Obj_length() on a hierarchical matrix will return the number of rows of child objects within 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 $H$.

dim_t FLASH_Obj_scalar_width( FLA_Obj H );

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

Notes: Using FLASH_Obj_scalar_width() on a flat matrix will always return the correct value (i.e., the same as that returned by FLA_Obj_width()). However, using FLA_Obj_width() on a hierarchical matrix will return the number of columns of child objects within 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 $H$. 
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: Using FLASH_Obj_depth() on a flat matrix will always return 0.

Imp. 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 $H$.

---

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 $H$.

---

dim_t FLASH_Obj_scalar_min_dim( FLA_Obj obj );

Purpose: Query the smaller of the hierarchical object view’s scalar length and width dimensions.

Notes: Using FLASH_Obj_scalar_min_dim() on a flat matrix will return the same value as FLA_Obj_min_dim().

Returns: An unsigned integer value of type dim_t.

Arguments:

obj – An FLA_Obj.
dim_t FLASH_Obj_scalar_max_dim( FLA_Obj obj );

Purpose: Query the larger of the hierarchical object view’s scalar length and width dimensions.

Notes: Using FLASH_Obj_scalar_max_dim() on a flat matrix will return the same value as FLA_Obj_max_dim().

Returns: An unsigned integer value of type dim_t.

Arguments:
   obj – An FLA_Obj.

---

dim_t FLASH_Obj_base_scalar_length( FLA_Obj H );

Purpose: Query the scalar length of the base object within H. That is, query the number of rows in the matrix represented by the object H as it was originally allocated.

Notes: Using FLASH_Obj_base_scalar_length() on a flat matrix will return the same value as FLA_Obj_base_length().

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

Arguments:
   H – An FLA_Obj representing matrix H.

---

dim_t FLASH_Obj_base_scalar_width( FLA_Obj H );

Purpose: Query the scalar width of the base object within H. That is, query the number of columns in the matrix represented by the object H as it was originally allocated.

Notes: Using FLASH_Obj_base_scalar_width() on a flat matrix will return the same value as FLA_Obj_base_width().

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

Arguments:
   H – An FLA_Obj representing matrix H.

---

dim_t FLASH_Obj_scalar_row_offset( FLA_Obj obj );

Purpose: Query the scalar row offset of an object view obj. That is, query the row offset of the view relative to the top-left corner of the underlying hierarchical matrix.

Notes: Using FLASH_Obj_scalar_row_offset() on a flat matrix will return the same value as FLA_Obj_row_offset().

Notes: This routine should only be used by advanced users and developers.

Returns: An unsigned integer value of type dim_t.

Arguments:
   obj – An FLA_Obj.
dim_t FLASH_Obj_scalar_col_offset( FLA_Obj obj );

Purpose: Query the scalar column offset of an object view obj. That is, query the column offset of the view relative to the top-left corner of the underlying hierarchical matrix.

Notes: Using FLASH_Obj_scalar_col_offset() on a flat matrix will return the same value as FLA_Obj_col_offset().

Notes: This routine should only be used by advanced users and developers.

Returns: An unsigned integer value of type dim_t.

Arguments:
obj – An FLA_Obj.

5.4.8 Managing Views

5.4.8.1 Vertical partitioning

FLA_Error FLASH_Part_create_2x1( FLA_Obj A, FLA_Obj* AT,
                                   FLA_Obj* AB,
                                   dim_t mb, FLA_Side side );

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

Notes: Unlike with FLA_Part_2x1(), the two views created by FLASH_Part_create_2x1() must be explicitly freed by a corresponding call to FLASH_Part_free_2x1().

Imp. Notes: This function performs a deep copy of the matrix hierarchy of A but creates leaf nodes that simply refer back to the original data in A.

Returns: FLA_SUCCESS

Arguments:
A – An FLA_Obj.
AT (on entry) – A pointer to an uninitialized FLA_Obj.
(on exit) – A pointer to a hierarchical FLA_Obj view into the top side of A.
AB (on entry) – A pointer to an uninitialized FLA_Obj.
(on exit) – A pointer to a hierarchical 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 FLASH_Part_free_2x1( FLA_Obj* AT, FLA_Obj* AB );

Purpose: Free the top and bottom side views that were previously created by FLASH_Part_create_2x1().

Returns: FLA_SUCCESS

Arguments:
AT (on entry) – A pointer to a valid hierarchical FLA_Obj view.
(on exit) – A pointer to an uninitialized FLA_Obj.
AB (on entry) – A pointer to a valid hierarchical FLA_Obj view.
(on exit) – A pointer to an uninitialized FLA_Obj.
5.4.8.2 Horizontal partitioning

FLA_Error FLASH_Part_create_1x2( FLA_Obj A, FLA_Obj* AL, FLA_Obj* AR,
       dim_t nb, FLA_Side side );

Purpose: Partition a hierarchical matrix \( A \) into left and right side views where the side indicated by \( \text{side} \) has \( nb \) columns.

Notes: Unlike with \( \text{FLA Part 1x2()} \), the two views created by \( \text{FLASH Part create 1x2()} \) must be explicitly freed by a corresponding call to \( \text{FLASH Part free 1x2()} \).

Imp. Notes: This function performs a deep copy of the matrix hierarchy of \( A \) but creates leaf nodes that simply refer back to the original data in \( A \).

Returns: \( \text{FLA SUCCESS} \)

Arguments:

\( A \) – An \( \text{FLA Obj} \).
\( AL \) – A pointer to an uninitialized \( \text{FLA Obj} \).
(on entry) – A pointer to a hierarchical \( \text{FLA Obj} \) view into the left side of \( A \).
(on exit) – A pointer to a hierarchical \( \text{FLA Obj} \) view into the left side of \( A \).
\( AR \) – A pointer to an uninitialized \( \text{FLA Obj} \).
(on entry) – A pointer to a hierarchical \( \text{FLA Obj} \) view into the right side of \( A \).
(on exit) – A pointer to a hierarchical \( \text{FLA Obj} \) view into the right side of \( A \).
\( nb \) – The number of columns to extract.
\( \text{side} \) – The side to which to extract \( nb \) columns.

FLA_Error FLASH_Part_free_1x2( FLA_Obj* AL, FLA_Obj* AR );

Purpose: Free the left and right side views that were previously created by \( \text{FLASH Part create 1x2()} \).

Returns: \( \text{FLA SUCCESS} \)

Arguments:

\( AL \) – A pointer to a valid hierarchical \( \text{FLA Obj} \) view.
(on entry) – A pointer to a valid hierarchical \( \text{FLA Obj} \) view.
(on exit) – A pointer to an uninitialized \( \text{FLA Obj} \).
\( AR \) – A pointer to a valid hierarchical \( \text{FLA Obj} \) view.
(on entry) – A pointer to an uninitialized \( \text{FLA Obj} \).
(on exit) – A pointer to an uninitialized \( \text{FLA Obj} \).
5.4.8.3 Bidirectional partitioning

FLA_Error FLASH_Part_create_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 hierarchical matrix $A$ into four quadrant views where the quadrant indicated by $\text{quadrant}$ is $mb \times nb$.

Notes: Unlike with \texttt{FLA\_Part\_2x2()}, the four quadrant views created by \texttt{FLASH\_Part\_create\_2x2()} must be explicitly freed by a corresponding call to \texttt{FLASH\_Part\_free\_2x2()}.

Imp. Notes: This function performs a deep copy of the matrix hierarchy of $A$ but creates leaf nodes that simply refer back to the original data in $A$.

Returns: \texttt{FLA\_SUCCESS}

Arguments:
\begin{itemize}
  \item $A$ – An \texttt{FLA\_Obj}.
  \item ATL...ABR (on entry) – Pointers to uninitialized \texttt{FLA\_Obj} structures.
  \item (on exit) – Pointers to hierarchical \texttt{FLA\_Obj} views into the four quadrants of $A$.
  \item $mb$ – The number of rows to extract.
  \item $nb$ – The number of columns to extract.
  \item $\text{quadrant}$ – The quadrant to which to extract $mb$ rows and $nb$ columns.
\end{itemize}

FLA_Error FLASH_Part_free_2x2( FLA_Obj* ATL, FLA_Obj* ATR,
    FLA_Obj* ABL, FLA_Obj* ABR );

Purpose: Free the quadrant views that were previously created by \texttt{FLASH\_Part\_create\_2x2()}.

Returns: \texttt{FLA\_SUCCESS}

Arguments:
\begin{itemize}
  \item ATL (on entry) – A pointer to a valid hierarchical \texttt{FLA\_Obj} view.
  \item (on exit) – A pointer to an uninitialized \texttt{FLA\_Obj}.
  \item ABL (on entry) – A pointer to a valid hierarchical \texttt{FLA\_Obj} view.
  \item (on exit) – A pointer to an uninitialized \texttt{FLA\_Obj}.
  \item ATR (on entry) – A pointer to a valid hierarchical \texttt{FLA\_Obj} view.
  \item (on exit) – A pointer to an uninitialized \texttt{FLA\_Obj}.
  \item ABR (on entry) – A pointer to a valid hierarchical \texttt{FLA\_Obj} view.
  \item (on exit) – A pointer to an uninitialized \texttt{FLA\_Obj}.
\end{itemize}
5.4.9 Utility functions

5.4.9.1 Miscellaneous functions

FLA_Error FLASH_Obj_show( char* header, FLA_Obj H, char* format, char* footer );

Purpose: Display the numerical values contained in the hierarchical object view $H$. The string header is output first (followed by a newline), then formatted contents of obj, and finally the string footer (followed by a newline). The string format should contain a printf()-style format string that describes how to output each element of the matrix. Note that format must be set according to the numerical contents of obj. For example, if the datatype of obj is FLA_DOUBLE, the user may choose to use "%11.3e" as the format string. Similarly, if the object datatype were FLA_DOUBLE_COMPLEX, the user would want to use something like "%11.3e + %11.3e" in order to denote the real and imaginary components.

Notes: Using FLASH_Obj_show() on a flat matrix object will yield the same output as using FLA_Obj_show().

Returns: FLA_SUCCESS

Arguments:

- header – A pointer to a string to precede the formatted output of obj.
- format – A pointer to a printf()-style format string.
- obj – A hierarchical FLA_Obj.
- footer – A pointer to a string to proceed the formatted output of obj.

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 [15]. The SuperMatrix runtime system itself is dependency-aware, and therefore is a major step forward when compared to more primitive workqueuing-based solutions [39].

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 enqueuing 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 ??.

The interface to the SuperMatrix mechanism and characteristics of its libflame implementation have been thoroughly documented in the literature [16, 15]. 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 ??.

**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:

\begin{itemize}
  \item \texttt{n\_threads} – An unsigned integer representing the number of threads to be requested upon parallel execution.
\end{itemize}

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( FLASH_Verbose verbose );

Purpose: Set or disable verboseness in SuperMatrix, particularly with regard to the dependency graph as it is generated. Three constant values are accepted for \texttt{verbose}:

\begin{itemize}
  \item \texttt{FLASH\_QUEUE\_VERBOSE\_NONE}. Verbese mode is disabled altogether.
  \item \texttt{FLASH\_QUEUE\_VERBOSE\_READABLE}. Human-readable dependency information is printed to standard output as execution progresses.
  \item \texttt{FLASH\_QUEUE\_VERBOSE\_GRAPHVIZ}. Dependency information is printed to standard output in the DOT language format, which is readable by the graphviz utility.
\end{itemize}

Arguments:

\begin{itemize}
  \item \texttt{verbose} – A value that sets or disables SuperMatrix verbosity.
\end{itemize}
FLASH_Verbose FLASH_Queue_get_verbose_output( void );

Purpose: Query the current status of verbosity in SuperMatrix.
Returns: A constant value of type FLASH_Verbose.

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). Five 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_1D_ROW_BLOCK_CYCLIC. Blocks are assigned and bound to threads in a one-dimensional block cyclic manner within rows.
  • FLASH_QUEUE_AFFINITY_1D_COLUMN_BLOCK_CYCLIC. Blocks are assigned and bound to threads in a one-dimensional block cyclic manner within columns.
  • 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.

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 of type FLASH_Data_aff.

FLA_Error FLASH_Queue_enable_gpu( void );

Purpose: Enable run-time support for GPU execution. When enabled, SuperMatrix tasks that are GPU-supported are executed on GPUs, while all other tasks are run on the CPU.
Returns: FLA_SUCCESS if SuperMatrix is enabled and a parallel region has not yet begun; FLA_FAILURE otherwise.

FLA_Error FLASH_Queue_disable_gpu( void );

Purpose: Disable run-time support for GPU execution. When disabled, all SuperMatrix tasks are run on the CPU.
Returns: FLA_SUCCESS if a parallel region has not yet begun; FLA_FAILURE otherwise.

FLA_Bool FLASH_Queue_get_enabled_gpu( void );

Purpose: Query whether GPU execution is currently enabled.
Notes: If SuperMatrix is currently disabled, the function returns FALSE regardless of whether GPU execution was previously enabled.
Returns: TRUE if SuperMatrix and GPU execution are both enabled; FALSE if SuperMatrix is disabled, or if SuperMatrix is enabled but GPU execution is disabled.

void FLASH_Queue_set_gpu_num_blocks( dim_t n_blocks );

Purpose: Set the number of storage blocks maintained by each GPU.
Notes: If the user encounters a run-time error reporting that an attempt to allocate memory on the GPU failed, it may be necessary to set n_blocks to a lower value.
Arguments:

n_blocks – An unsigned integer representing the number of blocks maintained by each GPU.

dim_t FLASH_Queue_get_gpu_num_blocks( void );

Purpose: Query the number of storage blocks maintained by each GPU.
Returns: An unsigned integer representing the number of blocks maintained by each GPU.
5.6. Front-ends

5.5.3 Integration with FLASH front-ends

SuperMatrix is invoked through the same FLASH front-end functions that are documented in Section 5.6.6. 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=omp` 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_Qdisable()`, 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_Qenable()` routine.

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

5.6 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.7 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.

5.6.1 BLAS operations

5.6.1.1 Level-1 BLAS

void FLA_Amax( 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.

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`.

Imp. Notes: This function is implemented as a wrapper to `FLA_Amax_external()`.

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>An FLA_Obj representing vector x.</td>
</tr>
<tr>
<td>i</td>
<td>An FLA_Obj representing scalar i.</td>
</tr>
</tbody>
</table>

6If a FLASH front-end does not exist for a particular operation, this means that the corresponding SuperMatrix implementation also does not yet exist.

7There are two exceptions to this: `FLA_Trmm()` and `FLA_Trsms()`. These routines are in fact direct wrappers to external implementations, as `libflame` does not contain native implementations of the `?trmm()` and `?trsms()` operations. These routines are also convenient for those who do not wish to call the somewhat longer functions named `FLA_Trmmexternal()` and `FLA_Trsmsexternal()`.
void FLA_Asum( FLA_Obj x, FLA_Obj norm );

Purpose: Compute the 1-norm of a vector:

\[ \|x\|_1 := \sum_{i=0}^{n-1} |x_i| \]

where \( \|x\|_1 \) is a scalar and \( x_i \) is the \( i \)th element of general vector \( x \) of length \( n \). Upon completion, the 1-norm \( \|x\|_1 \) is stored to \( \text{norm} \).

Imp. Notes: This function is implemented as a wrapper to \( \text{FLA}\_\text{Asum\_external}() \).

Constraints:
- The numerical datatype of \( x \) must be floating-point and must not be \( \text{FLA\_CONSTANT} \).
- The numerical datatype of \( \text{norm} \) must be real and must not be \( \text{FLA\_CONSTANT} \).
- The precision of the datatype of \( \text{norm} \) must be equal to that of \( x \).

Arguments:
- \( x \) – An \( \text{FLA\_Obj} \) representing vector \( x \).
- \( \text{norm} \) – An \( \text{FLA\_Obj} \) representing scalar \( \|x\|_1 \).

void FLA_Axpy( FLA_Obj alpha, FLA_Obj A, FLA_Obj B );
void FLASH_Axpy( FLA_Obj alpha, FLA_Obj A, FLA_Obj B );

Purpose: Perform an \texttt{AXPY} operation:

\[ B := B + \alpha A \]

where \( \alpha \) is a scalar, and \( A \) and \( B \) are general matrices.

Constraints:
- The numerical datatypes of \( A \) and \( B \) must be identical and floating-point, and must not be \( \text{FLA\_CONSTANT} \).
- If \( \alpha \) is not of datatype \( \text{FLA\_CONSTANT} \), then it must match the datatypes of \( A \) and \( B \).
- The dimensions of \( A \) and \( B \) must be conformal.

Int. Notes: \( \text{FLA}\_\text{Axpy}() \) expects \( A \) and \( B \) to be flat matrix objects.

Imp. Notes: \( \text{FLA}\_\text{Axpy}() \) simply invokes the external BLAS wrapper \( \text{FLA}\_\text{Axpy\_external}() \). \( \text{FLASH}\_\text{Axpy}() \) uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the \texttt{AXPY} operation into subproblems expressed in terms of individual blocks of \( A \) and \( B \) and then invokes \( \text{FLA}\_\text{Axpy\_external}() \) to perform the computation on these blocks.

Arguments:
- \( \alpha \) – An \( \text{FLA\_Obj} \) representing scalar \( \alpha \).
- \( A \) – An \( \text{FLA\_Obj} \) representing matrix \( A \).
- \( B \) – An \( \text{FLA\_Obj} \) representing matrix \( B \).
void FLA_Axpyt( FLA_Trans trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj B );
void FLASH_Axpyt( FLA_Trans trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj B );

Purpose: Perform one of the following extended AXPY operations:

\[
\begin{align*}
B & := B + \alpha A \\
B & := B + \alpha A^T \\
B & := B + \alpha \bar{A} \\
B & := B + \alpha A^H
\end{align*}
\]

where \( \alpha \) is a scalar, and \( A \) and \( B \) are general matrices. The \texttt{trans} argument allows the computation to proceed as if \( A \) were conjugated and/or transposed.

Notes: If \( A \) and \( B \) are vectors, \texttt{FLA_Axpyt()} will implicitly and automatically perform the transposition necessary to achieve conformal dimensions regardless of the value of \texttt{trans}.

Constraints:
- The numerical datatypes of \( A \) and \( B \) must be identical and floating-point, and must not be \texttt{FLA_CONSTANT}.
- If \( \alpha \) is not of datatype \texttt{FLA_CONSTANT}, then it must match the datatypes of \( A \) and \( B \).
- If \( A \) and \( B \) are vectors, then their lengths must be equal. Otherwise, if \texttt{trans} equals \texttt{FLA_NO_TRANSPOSE} or \texttt{FLA_CONJ_NO_TRANSPOSE}, then the dimensions of \( A \) and \( B \) must be conformal; otherwise, if \texttt{trans} equals \texttt{FLA_TRANSPOSE} or \texttt{FLA_CONJ_TRANSPOSE}, then the dimensions of \( A^T \) and \( B \) must be conformal.

Int. Notes: \texttt{FLA_Axpyt()} expects \( A \) and \( B \) to be flat matrix objects.

Imp. Notes: \texttt{FLA_Axpyt()} simply invokes the external BLAS wrapper \texttt{FLA_Axpyt_external()}. \texttt{FLASH_Axpyt()} uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the extended AXPY operation into subproblems expressed in terms of individual blocks of \( A \) and \( B \) and then invokes \texttt{FLA_Axpyt_external()} to perform the computation on these blocks.

Arguments:

- \texttt{trans} – Indicates whether the operation proceeds as if \( A \) were conjugated and/or transposed.
- \texttt{alpha} – An \texttt{FLA_Obj} representing scalar \( \alpha \).
- \texttt{A} – An \texttt{FLA_Obj} representing matrix \( A \).
- \texttt{B} – An \texttt{FLA_Obj} representing matrix \( B \).
void FLA_Axpyrt( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj B );

**Purpose:** Perform one of the following extended \textit{AXPY} operations:

\begin{align*}
B & := B + \alpha A \\
B & := B + \alpha A^T \\
B & := B + \alpha \bar{A} \\
B & := B + \alpha A^H
\end{align*}

where $A$ and $B$ are triangular (or trapezoidal) matrices. The \texttt{uplo} argument indicates whether the lower or upper triangle of $B$ is updated by the operation. The \texttt{trans} argument allows the computation to proceed as if $A$ were conjugated and/or transposed. Note that the \texttt{uplo} and \texttt{trans} arguments together determine which triangle of $A$ is read and which triangle of $B$ is updated.

**Constraints:**
- The numerical datatypes of $A$ and $B$ must be identical, and must not be \texttt{FLA_CONSTANT}.
- If $\alpha$ is not of datatype \texttt{FLA_CONSTANT}, then it must match the datatypes of $X$ and $Y$.
- If \texttt{trans} equals \texttt{FLA_NO_TRANSPOSE} or \texttt{FLA_CONJ_NO_TRANSPOSE}, then the dimensions of $A$ and $B$ must be conformal; otherwise, if \texttt{trans} equals \texttt{FLA_TRANSPOSE} or \texttt{FLA_CONJ_TRANSPOSE}, then the dimensions of $A^T$ and $B$ must be conformal.

**Int. Notes:** \texttt{FLA_Axpyrt()} expects $A$ and $B$ to be flat matrix objects.

**Imp. Notes:** This function is implemented as a wrapper to \texttt{FLA_Axpyrt_external()}.  

**Arguments:**

- \texttt{uplo} – Indicates whether the lower or upper triangles of $A$ and $B$ are referenced and updated during the operation.
- \texttt{trans} – Indicates whether the operation proceeds as if $A$ were conjugated and/or transposed.
- \texttt{alpha} – An \texttt{FLA_Obj} representing scalar $\alpha$.
- \texttt{A} – An \texttt{FLA_Obj} representing matrix $A$.
- \texttt{B} – An \texttt{FLA_Obj} representing matrix $B$. 
void FLA_Axpys( FLA_Obj alpha0, FLA_Obj alpha1, FLA_Obj A, FLA_Obj beta, FLA_Obj B );

**Purpose:** Perform the following extended \texttt{AXPY} operation:

\[
B := \beta B + \alpha_0 \alpha_1 A
\]

where \( \alpha_0, \alpha_1 \) and \( \beta \) are scalars, and \( A \) and \( B \) are general matrices.

**Notes:** If \( A \) and \( B \) are vectors, \texttt{FLA_Axpys()} will implicitly and automatically perform the transposition necessary to achieve conformal dimensions.

**Constraints:**
- The numerical datatypes of \( A \) and \( B \) must be identical and floating-point, and must not be \texttt{FLA_CONSTANT}.
- If \( \alpha_0, \alpha_1, \) and \( \beta \) are not of datatype \texttt{FLA_CONSTANT}, then they must match the datatypes of \( A \) and \( B \).

**Imp. Notes:** This function is implemented as a wrapper to \texttt{FLA_Axpys\_external()}. 

**Arguments:**

- \texttt{alpha0} – An \texttt{FLA\_Obj} representing scalar \( \alpha_0 \).
- \texttt{alpha1} – An \texttt{FLA\_Obj} representing scalar \( \alpha_1 \).
- \texttt{A} – An \texttt{FLA\_Obj} representing matrix \( A \).
- \texttt{beta} – An \texttt{FLA\_Obj} representing scalar \( \beta \).
- \texttt{B} – An \texttt{FLA\_Obj} representing matrix \( B \).

---

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 \( B \).

**Constraints:**
- The numerical datatypes of \( A \) and \( B \) must be identical and must not be \texttt{FLA\_CONSTANT}.
- The dimensions of \( A \) and \( B \) must be conformal.

**Int. Notes:** \texttt{FLA\_Copy()} expects \( A \) and \( B \) to be flat matrix objects.

**Imp. Notes:** \texttt{FLA\_Copy()} simply invokes the external BLAS wrapper \texttt{FLA\_Copy\_external()}. \texttt{FLASH\_Copy()} uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the \texttt{COPY} operation into subproblems expressed in terms of individual blocks of \( A \) and \( B \) and then invokes \texttt{FLA\_Copy\_external()} to perform the computation on these blocks.

**Arguments:**

- \texttt{A} – An \texttt{FLA\_Obj} representing matrix \( A \).
- \texttt{B} – An \texttt{FLA\_Obj} representing matrix \( B \).
void FLA_Copyr( FLA_Uplo uplo, FLA_Obj A, FLA_Obj B );

**Purpose:** Perform an extended copy operation on triangular matrices $A$ and $B$:

\[ B := A \]

where $A$ and $B$ are triangular (or trapezoidal) matrices. The `uplo` argument indicates whether the lower or upper triangles of $A$ and $B$ are referenced and updated by the operation.

**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.

**Imp. Notes:** This function is implemented as a wrapper to `FLA_Copyr_external()`.

**Arguments:**
- `uplo` – Indicates whether the lower or upper triangles of $A$ and $B$ are referenced and updated during the operation.
- `A` – An `FLA_Obj` representing matrix $A$.
- `B` – An `FLA_Obj` representing matrix $B$.

---

void FLA_Copyrt( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj A, FLA_Obj B );

**Purpose:** Perform an extended copy operation on triangular matrices $A$ and $B$:

\[
\begin{align*}
B & := A \\
B & := A^T \\
B & := \bar{A} \\
B & := A^H
\end{align*}
\]

where $A$ and $B$ are triangular (or trapezoidal) matrices. The `uplo` argument indicates whether the lower or upper triangle of $B$ is updated by the operation. The `trans` argument allows the computation to proceed as if $A$ were conjugated and/or transposed. Note that the `uplo` and `trans` arguments together determine which triangle of $A$ is read and which triangle of $B$ is overwritten.

**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.

**Imp. Notes:** This function is implemented as a wrapper to `FLA_Copyrt_external()`.

**Arguments:**
- `uplo` – Indicates whether the lower or upper triangles of $A$ and $B$ are referenced and updated during the operation.
- `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_Copyt( FLA_Trans trans, FLA_Obj A, FLA_Obj B );
void FLASH_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 copy operations:

\[
\begin{align*}
B & := A \\
B & := A^T \\
B & := \bar{A} \\
B & := A^H
\end{align*}
\]

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.

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.

Int. Notes: FLA_Copyt() expects A and B to be flat matrix objects.

Imp. Notes: FLA_Copyt() simply invokes the external BLAS wrapper FLA_Copyt_external(). FLASH_Copyt() uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the extended copy operation into subproblems expressed in terms of individual blocks of A and B and then invokes FLA_Copyt_external() to perform the computation on these blocks.

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 \( \rho \) is a scalar, and \( \chi_i \) and \( \psi_i \) are the \( i \)th elements of general vectors \( x \) and \( y \), respectively, where both vectors are of length \( n \). Upon completion, the dot product \( \rho \) is stored to \( \text{rho} \).

**Constraints:**
- The numerical datatypes of \( x \), \( y \), and \( \rho \) must be identical and floating-point, and must not be FLA_CONSTANT.
- The lengths of vectors \( x \) and \( y \) must be equal.

**Imp. Notes:** This function is implemented as a wrapper to FLA_Dot_external().

**Arguments:**
- \( x \) – An FLA_Obj representing vector \( x \).
- \( y \) – An FLA_Obj representing vector \( y \).
- \( \rho \) – An FLA_Obj representing scalar \( \rho \).

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:

\[ \rho := \sum_{i=0}^{n-1} \chi_i \psi_i \]

\[ \rho := \sum_{i=0}^{n-1} \bar{\chi}_i \psi_i \]

where \( \rho \) is a scalar, and \( \chi_i \) and \( \psi_i \) are the \( i \)th elements of general vectors \( x \) and \( y \), respectively, where both vectors are of length \( n \). Upon completion, the dot product \( \rho \) is stored to \( \text{rho} \). The \( \text{conj} \) argument allows the computation to proceed as if \( x \) were conjugated.

**Notes:** If \( x \), \( y \), and \( \rho \) are real, the value of \( \text{conj} \) is ignored and FLA_Dotc() behaves exactly as FLA_Dot().

**Constraints:**
- The numerical datatypes of \( x \), \( y \), and \( \rho \) must be identical and floating-point, and must not be FLA_CONSTANT.
- The lengths of vectors \( x \) and \( y \) must be equal.

**Imp. Notes:** This function is implemented as a wrapper to FLA_Dotc_external().

**Arguments:**
- \( \text{conj} \) – Indicates whether to conjugate the intermediate element-wise terms of the dot product.
- \( x \) – An FLA_Obj representing vector \( x \).
- \( y \) – An FLA_Obj representing vector \( y \).
- \( \rho \) – An FLA_Obj representing scalar \( \rho \).
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 \( \alpha, \beta, \) and \( \rho \) are scalars, and \( \chi_i \) and \( \psi_i \) are the \( i \)th elements of general vectors \( x \) and \( y \), respectively, where both vectors are of length \( n \). Upon completion, the dot product \( \rho \) is stored to \( \text{rho} \).

Constraints:
- The numerical datatypes of \( x, y, \) and \( \rho \) must be identical and floating-point, and must not be FLA_CONSTANT.
- If \( \alpha \) and \( \beta \) are not of datatype FLA_CONSTANT, then they must match the datatypes of \( x, y, \) and \( \rho \).
- The lengths of vectors \( x \) and \( y \) must be equal.

Imp. Notes: This function is implemented as a wrapper to FLA_Dots_external().

Arguments:
- \( \text{alpha} \) – An FLA_Obj representing scalar \( \alpha \).
- \( x \) – An FLA_Obj representing vector \( x \).
- \( y \) – An FLA_Obj representing vector \( y \).
- \( \text{beta} \) – An FLA_Obj representing scalar \( \beta \).
- \( \text{rho} \) – An FLA_Obj representing scalar \( \rho \).
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:

\[
\rho := \beta \rho + \alpha \sum_{i=0}^{n-1} \chi_i \psi_i \\
\rho := \beta \rho + \alpha \sum_{i=0}^{n-1} \bar{\chi}_i \psi_i
\]

where \( \alpha, \beta, \) and \( \rho \) are scalars, and \( \chi_i \) and \( \psi_i \) are the \( i \)th elements of general vectors \( x \) and \( y \), respectively, where both vectors are of length \( n \). Upon completion, the dot product \( \rho \) is stored to \( \rho \). The \( \text{conj} \) argument allows the computation to proceed as if \( x \) were conjugated.

Notes: If \( x, y, \) and \( \rho \) are real, the value of \( \text{conj} \) is ignored and \( \text{FLA\_Dotcs()} \) behaves exactly as \( \text{FLA\_Dots()} \).

Constraints:
- The numerical datatypes of \( x, y, \) and \( \rho \) must be identical and floating-point, and must not be \( \text{FLA\_CONSTANT} \).
- If \( \alpha \) and \( \beta \) are not of datatype \( \text{FLA\_CONSTANT} \), then they must match the datatypes of \( x, y, \) and \( \rho \).
- The lengths of vectors \( x \) and \( y \) must be equal.

Imp. Notes: This function is implemented as a wrapper to \( \text{FLA\_Dotcs\_external()} \).

Arguments:
- \( \text{conj} \) – Indicates whether the operation proceeds as if \( x \) and \( y \) were conjugated.
- \( \text{alpha} \) – An \( \text{FLA\_Obj} \) representing scalar \( \alpha \).
- \( \text{x} \) – An \( \text{FLA\_Obj} \) representing vector \( x \).
- \( \text{y} \) – An \( \text{FLA\_Obj} \) representing vector \( y \).
- \( \text{beta} \) – An \( \text{FLA\_Obj} \) representing scalar \( \beta \).
- \( \text{rho} \) – An \( \text{FLA\_Obj} \) representing scalar \( \rho \).
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:

\[ \rho := \beta \rho + \alpha \sum_{i=0}^{n-1} \chi_i \psi_i \]

where \( \alpha, \beta, \) and \( \rho \) are scalars, and \( \chi_i \) and \( \psi_i \) are the \( i \)th elements of general vectors \( x \) and \( y \), respectively, where both vectors are of length \( n \). Upon completion, the dot product \( \rho \) is stored to \( \text{rho} \).

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().

Constraints:
- The numerical datatypes of \( x \), \( y \), and \( \rho \) must be identical and floating-point, and must not be FLA_CONSTANT.
- If \( \alpha \) and \( \beta \) are not of datatype FLA_CONSTANT, then they must match the datatypes of \( x \), \( y \), and \( \rho \).
- The lengths of vectors \( x \) and \( y \) must be equal.

Imp. Notes: This function is implemented as a wrapper to FLA_Dot2s_external().

Arguments:

- \( \text{alpha} \) – An FLA_Obj representing scalar \( \alpha \).
- \( x \) – An FLA_Obj representing vector \( x \).
- \( y \) – An FLA_Obj representing vector \( y \).
- \( \text{beta} \) – An FLA_Obj representing scalar \( \beta \).
- \( \text{rho} \) – An FLA_Obj representing scalar \( \rho \).
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:

\[
\rho := \beta \rho + \alpha \sum_{i=0}^{n-1} x_i \psi_i + \alpha \sum_{i=0}^{n-1} \chi_i \psi_i \\
\rho := \beta \rho + \alpha \sum_{i=0}^{n-1} \overline{x_i} \psi_i + \overline{\alpha} \sum_{i=0}^{n-1} \psi_i \chi_i
\]

where \(\alpha\), \(\beta\), and \(\rho\) are scalars, and \(\chi_i\) and \(\psi_i\) are the \(i\)th elements of general vectors \(x\) and \(y\), respectively, where both vectors are of length \(n\). Upon completion, the dot product \(\rho\) is stored to \(\text{rho}\). The \(\text{conj}\) argument allows the computation to proceed as if \(x\) were conjugated.

Notes: If \(x\), \(y\), and \(\rho\) are real, the value of \(\text{conj}\) is ignored and \(\text{FLA Dot2cs()}\) behaves exactly as \(\text{FLA Dot2s()}\).

Constraints:
- The numerical datatypes of \(x\), \(y\), and \(\rho\) must be identical and floating-point, and must not be \(\text{FLA_CONSTANT}\).
- If \(\alpha\) and \(\beta\) are not of datatype \(\text{FLA_CONSTANT}\), then they must match the datatypes of \(x\), \(y\), and \(\rho\).
- The lengths of vectors \(x\) and \(y\) must be equal.

Imp. Notes: This function is implemented as a wrapper to \(\text{FLA Dot2cs_external()}\).

Arguments:
- \(\text{conj}\) – Indicates whether the operation proceeds as if \(x\) and \(y\) were conjugated.
- \(\text{alpha}\) – An \(\text{FLA_Obj}\) representing scalar \(\alpha\).
- \(\text{x}\) – An \(\text{FLA_Obj}\) representing vector \(x\).
- \(\text{y}\) – An \(\text{FLA_Obj}\) representing vector \(y\).
- \(\text{beta}\) – An \(\text{FLA_Obj}\) representing scalar \(\beta\).
- \(\text{rho}\) – An \(\text{FLA_Obj}\) representing scalar \(\rho\).

void FLA_Inv_scal( FLA_Obj alpha, FLA_Obj A );

Purpose: Perform an inverse scaling operation:

\[
A := \alpha^{-1} A
\]

where \(\alpha\) is a scalar and \(A\) is a general matrix.

Constraints:
- The numerical datatype of \(A\) must be floating-point, and must not be \(\text{FLA_CONSTANT}\).
- If \(\alpha\) is not of datatype \(\text{FLA_CONSTANT}\), then it must match the datatype of \(A\) if \(A\) is real and the precision of \(A\) if \(A\) is complex.
- \(\alpha\) may not be equal to zero.

Imp. Notes: This function is implemented as a wrapper to \(\text{FLA_Inv_scal_external()}\).

Arguments:
- \(\text{alpha}\) – An \(\text{FLA_Obj}\) representing scalar \(\alpha\).
- \(\text{A}\) – An \(\text{FLA_Obj}\) 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 \( \alpha \) is a scalar and \( A \) is a general matrix. The \texttt{conalpha} argument allows the computation to proceed as if \( \alpha \) were conjugated.

**Notes:** If \( \alpha \) is real, the value of \texttt{conalpha} is ignored and \texttt{FLA_Inv_scalc()} behaves exactly as \texttt{FLA_Inv_scal()}.  

**Constraints:**
- The numerical datatype of \( A \) must be floating-point, and must not be \texttt{FLA_CONSTANT}.  
- If \( \alpha \) is not of datatype \texttt{FLA_CONSTANT}, then it must match the datatype of \( A \) if \( A \) is real and the precision of \( A \) if \( A \) is complex.  
- \( \alpha \) may not be equal to zero.

**Imp. Notes:** This function is implemented as a wrapper to \texttt{FLA_Inv_scalc_external()}.  

**Arguments:**
- \texttt{conalpha} – Indicates whether the operation proceeds as if \( \alpha \) were conjugated.  
- \texttt{alpha} – An \texttt{FLA_Obj} representing scalar \( \alpha \).  
- \texttt{A} – An \texttt{FLA_Obj} representing matrix \( A \).

---

void FLA_Nrm2( FLA_Obj x, FLA_Obj norm );

**Purpose:** Compute the 2-norm of a vector:

\[
\|x\|_2 := \left( \sum_{i=0}^{n-1} |x_i|^2 \right)^{\frac{1}{2}}
\]

where \( \|x\|_2 \) is a scalar and \( x_i \) is the \( i \)th element of general vector \( x \) of length \( n \). Upon completion, the 2-norm \( \|x\|_2 \) is stored to \texttt{norm}.

**Constraints:**
- The numerical datatype of \( x \) must be floating-point and must not be \texttt{FLA_CONSTANT}.  
- The numerical datatype of \texttt{norm} must be real and must not be \texttt{FLA_CONSTANT}.  
- The precision of the datatype of \texttt{norm} must be equal to that of \( x \).

**Imp. Notes:** This function is implemented as a wrapper to \texttt{FLA_Nrm2_external()}.  

**Arguments:**
- \texttt{x} – An \texttt{FLA_Obj} representing vector \( x \).  
- \texttt{norm} – An \texttt{FLA_Obj} representing scalar \( \|x\|_2 \).
void FLA_Scal( FLA_Obj alpha, FLA_Obj A );

**Purpose:** Perform a scaling operation:

\[
A := \alpha A
\]

where \( \alpha \) is a scalar and \( A \) is a general matrix.

**Constraints:**
- The numerical datatype of \( A \) must be floating-point, and must not be `FLA_CONSTANT`.
- If \( \alpha \) 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.

**Imp. Notes:** This function is implemented as a wrapper to `FLA_Scal_external()`.

**Arguments:**
- `alpha` – An `FLA_Obj` representing scalar \( \alpha \).
- `A` – An `FLA_Obj` 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 \( \alpha \) is a scalar and \( A \) is a general matrix. The `conjalpha` argument allows the computation to proceed as if \( \alpha \) were conjugated.

**Notes:** If \( \alpha \) is real, the value of `conjalpha` is ignored and `FLA_Scalc()` behaves exactly as `FLA_Scal()`.

**Constraints:**
- The numerical datatype of \( A \) must be floating-point, and must not be `FLA_CONSTANT`.
- If \( \alpha \) 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.

**Imp. Notes:** This function is implemented as a wrapper to `FLA_Scalc_external()`.

**Arguments:**
- `conjalpha` – Indicates whether the operation proceeds as if \( \alpha \) were conjugated.
- `alpha` – An `FLA_Obj` representing scalar \( \alpha \).
- `A` – An `FLA_Obj` 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 \( \alpha \) 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.

**Constraints:**
- The numerical datatype of \( A \) must be floating-point, and must not be `FLA_CONSTANT`.
- If \( \alpha \) 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.

**Imp. Notes:** This function is implemented as a wrapper to `FLA_Scalr_external()`.

**Arguments:**
- `uplo` – Indicates whether the lower or upper triangle of \( A \) is referenced and updated during the operation.
- `alpha` – An `FLA_Obj` representing scalar \( \alpha \).
- `A` – An `FLA_Obj` representing matrix \( A \).

void FLA_Swap( FLA_Obj A, FLA_Obj B );

**Purpose:** Swap the contents of two general matrices \( A \) and \( B \).

**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.

**Imp. Notes:** This function is implemented as a wrapper to `FLA_Swap_external()`.

**Arguments:**
- `A` – An `FLA_Obj` representing matrix \( A \).
- `B` – An `FLA_Obj` 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.

**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.

**Imp. Notes:** This function is implemented as a wrapper to `FLA_Swapt_external()`.

**Arguments:**
- `transab` – Indicates whether the operation proceeds as if \( A \) and \( B \) were conjugated and/or transposed.
- `A` – An `FLA_Obj` representing matrix \( A \).
- `B` – An `FLA_Obj` representing matrix \( B \).
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-vector multiplication (gemv) operations:

\[
\begin{align*}
  y & := \beta y + \alpha A x \\
  y & := \beta y + \alpha A^T x \\
  y & := \beta y + \alpha A^H x \\
\end{align*}
\]

where \( \alpha \) and \( \beta \) are scalars, \( A \) is a general matrix, and \( x \) and \( y \) are general vectors.

The \texttt{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 \texttt{FLA_CONSTANT}.
- If \( \alpha \) and \( \beta \) are not of datatype \texttt{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: \texttt{FLA_Gemv()} expects \( A \), \( x \), and \( y \) to be flat matrix objects.

Imp. Notes: \texttt{FLA_Gemv()} invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS. \texttt{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 \texttt{FLA_Gemv external()} to perform the computation on these blocks and subvectors.

Arguments:

- \texttt{transa} — Indicates whether the operation proceeds as if \( A \) were conjugated and/or transposed.
- \texttt{alpha} — An \texttt{FLA_Obj} representing scalar \( \alpha \).
- \texttt{A} — An \texttt{FLA_Obj} representing matrix \( A \).
- \texttt{x} — An \texttt{FLA_Obj} representing vector \( x \).
- \texttt{beta} — An \texttt{FLA_Obj} representing scalar \( \beta \).
- \texttt{y} — An \texttt{FLA_Obj} 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 (gemv) operations:

\[
\begin{align*}
    y & := \beta y + \alpha A \bar{x} \\
    y & := \beta y + \alpha A^T \bar{x} \\
    y & := \beta y + \alpha \bar{A} \bar{x} \\
    y & := \beta y + \alpha \bar{A}^T \bar{x} \\
    y & := \beta y + \alpha \bar{A} \bar{\bar{x}} \\
    y & := \beta y + \alpha \bar{A}^H \bar{x}
\end{align*}
\]

where \( \alpha \) and \( \beta \) are scalars, \( A \) is a general matrix, and \( x \) and \( y \) are general vectors. The \text{trans} argument allows the computation to proceed as if \( A \) were conjugated and/or transposed. Likewise, the \text{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:

\[
\begin{align*}
    y_r & := \beta y_r + \alpha x_r A^T \\
    y_r & := \beta y_r + \alpha x_r A \\
    y_r & := \beta y_r + \alpha x_r A^H \\
    y_r & := \beta y_r + \alpha x_r \bar{\bar{A}} \\
    y_r & := \beta y_r + \alpha x_r \bar{A}
\end{align*}
\]

respectively, where \( x_r \) and \( y_r \) are row vectors.

If \( A, x, \) and \( y \) are real, the value of \text{conjx} is ignored and \text{FLA_Gemvc()} behaves exactly as \text{FLA_Gemv()}.

Constraints:

- The numerical datatypes of \( A, x, \) and \( y \) must be identical and floating-point, and must not be \text{FLA_CONSTANT}.
- If \( \alpha \) and \( \beta \) are not of datatype \text{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.

Imp. Notes: This function is implemented as a wrapper to \text{FLA_Gemvc_external()}.

Arguments:

- \text{transa} – Indicates whether the operation proceeds as if \( A \) were conjugated and/or transposed.
- \text{conjx} – Indicates whether the operation proceeds as if \( x \) were conjugated.
- \text{alpha} – An \text{FLA_Obj} representing scalar \( \alpha \).
- \text{A} – An \text{FLA_Obj} representing matrix \( A \).
- \text{x} – An \text{FLA_Obj} representing vector \( x \).
- \text{beta} – An \text{FLA_Obj} representing scalar \( \beta \).
- \text{y} – An \text{FLA_Obj} representing vector \( y \).
```c
void FLA_Ger( FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj A );
```

**Purpose:** Perform a general rank-1 update (GER) operation:

\[
A := A + \alpha xy^T
\]

where \( \alpha \) is a scalar, \( A \) is a general matrix, and \( x \) and \( y \) are general vectors.

**Constraints:**
- The numerical datatypes of \( A, x, \) and \( y \) must be identical and floating-point, and must not be FLA_CONSTANT.
- If \( \alpha \) 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.

**Imp. Notes:** This function is implemented as a wrapper to FLA_Ger_external().

**Arguments:**
- **alpha** – An FLA_Obj representing scalar \( \alpha \).
- **x** – An FLA_Obj representing vector \( x \).
- **y** – An FLA_Obj representing vector \( y \).
- **A** – An FLA_Obj representing matrix \( A \).

```c
void FLA_Gerc( FLA_Conj conjx, FLA_Conj conjoy, FLA_Obj alpha,
             FLA_Obj x, FLA_Obj y, FLA_Obj A );
```

**Purpose:** Perform one of the following extended general rank-1 update (GER) operations:

\[
A := A + \alpha xy^T \\
A := A + \alpha x\overline{y}^T \\
A := A + \overline{\alpha} y^T \\
A := A + \overline{\alpha} x\overline{y}^T
\]

where \( \alpha \) is a scalar, \( A \) is a general matrix, and \( x \) and \( y \) are general vectors. The **conjx** and **conjoy** 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 **conjoy** are ignored and FLA_Gerc() behaves exactly as FLA_Ger().

**Constraints:**
- The numerical datatypes of \( A, x, \) and \( y \) must be identical and floating-point, and must not be FLA_CONSTANT.
- If \( \alpha \) 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.

**Imp. Notes:** This function is implemented as a wrapper to FLA_Gerc_external().

**Arguments:**
- **conjx** – Indicates whether the operation proceeds as if \( x \) were conjugated.
- **conjoy** – Indicates whether the operation proceeds as if \( y \) were conjugated.
- **alpha** – An FLA_Obj representing scalar \( \alpha \).
- **x** – An FLA_Obj representing vector \( x \).
- **y** – An FLA_Obj representing vector \( y \).
- **A** – An FLA_Obj 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 (HEMV) operation:

\[ y := \beta y + \alpha Ax \]

where \( \alpha \) and \( \beta \) are scalars, \( A \) is a Hermitian matrix, and \( x \) and \( y \) are general vectors. The \texttt{uplo} argument indicates whether the lower or upper triangle of \( A \) is referenced by the operation.

Notes: When invoked with real objects, this function performs the SYMV operation.

Constraints:
- The numerical datatypes of \( A \), \( x \), and \( y \) must be identical and must not be \texttt{FLA_CONSTANT}.
- If \( \alpha \) and \( \beta \) are not of datatype \texttt{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.

Imp. Notes: This function is implemented as a wrapper to \texttt{FLA_Hemv_external}.

Arguments:

- \texttt{uplo} – Indicates whether the lower or upper triangle of \( A \) is referenced during the operation.
- \texttt{alpha} – An \texttt{FLA_Obj} representing scalar \( \alpha \).
- \texttt{A} – An \texttt{FLA_Obj} representing matrix \( A \).
- \texttt{x} – An \texttt{FLA_Obj} representing vector \( x \).
- \texttt{beta} – An \texttt{FLA_Obj} representing scalar \( \beta \).
- \texttt{y} – An \texttt{FLA_Obj} 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 (HEMV) operations:

\[ y := \beta y + \alpha A x \]
\[ y := \beta y + \alpha \bar{A} x \]

where \( \alpha \) and \( \beta \) are scalars, \( A \) is a Hermitian 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 conj argument allows the computation to proceed as if \( A \) were conjugated.

Notes: When invoked with real objects, this function performs the SYMV operation.

Constraints:

- The numerical datatypes of \( A \), \( x \), and \( y \) must be identical and must not be FLA_CONSTANT.
- If \( \alpha \) and \( \beta \) 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.
- \( \text{trans} \) may not be FLA_CONJ_TRANSPOSE or FLA_CONJ_NO_TRANSPOSE.

Imp. Notes: This function is implemented as a wrapper to FLA_Hemvc_external().

Arguments:

\begin{itemize}
\item \text{uplo} \quad \text{Indicates whether the lower or upper triangle of \( A \) is referenced during the operation.}
\item \text{conj} \quad \text{Indicates whether the operation proceeds as if \( A \) were conjugated.}
\item \text{alpha} \quad \text{An FLA_Obj representing scalar \( \alpha \).}
\item \text{A} \quad \text{An FLA_Obj representing matrix \( A \).}
\item \text{x} \quad \text{An FLA_Obj representing vector \( x \).}
\item \text{beta} \quad \text{An FLA_Obj representing scalar \( \beta \).}
\item \text{y} \quad \text{An FLA_Obj representing vector \( y \).}
\end{itemize}
void FLA_Her( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj A );

**Purpose:** Perform a Hermitian rank-1 update (HER) operation:

\[
A := A + \alpha xx^H
\]

where \( \alpha \) is a scalar, \( A \) is a Hermitian matrix, and \( x \) is a general vector. The \( \text{uplo} \) argument indicates whether the lower or upper triangle of \( A \) is referenced and updated by the operation.

**Notes:** When invoked with real objects, this function performs the \text{SYR} operation.

**Constraints:**
- The numerical datatypes of \( A \) and \( x \) must be identical and must not be \text{FLA_CONSTANT}.
- If \( \alpha \) is not of datatype \text{FLA_CONSTANT}, then it must match the datatypes of \( A \) and \( x \).
- The length of \( x \) and the order of \( A \) must be equal.

**Imp. Notes:** This function is implemented as a wrapper to \text{FLA_Her_external}().

**Arguments:**
- \( \text{uplo} \) – Indicates whether the lower or upper triangle of \( A \) is referenced and updated during the operation.
- \( \text{alpha} \) – An \text{FLA_Obj} representing scalar \( \alpha \).
- \( \text{x} \) – An \text{FLA_Obj} representing vector \( x \).
- \( \text{A} \) – An \text{FLA_Obj} representing matrix \( A \).

void FLA_Herc( FLA_Uplo uplo, FLA_Conj conj, FLA_Obj alpha, FLA_Obj x, FLA_Obj A );

**Purpose:** Perform one of the following extended Hermitian rank-1 update (HER) operations:

\[
A := A + \alpha xx^H
\]
\[
A := A + \alpha \bar{x}x^T
\]

where \( \alpha \) is a scalar, \( A \) is a Hermitian matrix, and \( x \) is a general vector. The \( \text{uplo} \) argument indicates whether the lower or upper triangle of \( A \) is referenced and updated by the operation. The \( \text{conj} \) argument allows the computation of the conjugated rank-1 product \( \bar{x}x^T \).

**Notes:** When invoked with real objects, this function performs the \text{SYR} operation.

**Constraints:**
- The numerical datatypes of \( A \) and \( x \) must be identical and must not be \text{FLA_CONSTANT}.
- If \( \alpha \) is not of datatype \text{FLA_CONSTANT}, then it must match the datatypes of \( A \) and \( x \).
- The length of \( x \) and the order of \( A \) must be equal.
- \text{trans} may not be \text{FLA_CONJ_TRANSPOSE} or \text{FLA_CONJ_NO_TRANSPOSE}.

**Imp. Notes:** This function is implemented as a wrapper to \text{FLA_Herc_external}().

**Arguments:**
- \( \text{uplo} \) – Indicates whether the lower or upper triangle of \( A \) is referenced and updated during the operation.
- \( \text{trans} \) – Indicates whether the operation proceeds as if the rank-1 product is conjugated.
- \( \text{alpha} \) – An \text{FLA_Obj} representing scalar \( \alpha \).
- \( \text{x} \) – An \text{FLA_Obj} representing vector \( x \).
- \( \text{A} \) – An \text{FLA_Obj} representing matrix \( A \).
void FLA_Her2( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj A );

**Purpose:** Perform a Hermitian rank-2 update (her2) operation:

\[
A := A + \alpha xy^H + \bar{\alpha}yx^H
\]

where \(\alpha\) is a scalar, \(A\) is a Hermitian 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.

**Notes:** When invoked with real objects, this function performs the syr2 operation.

**Constraints:**
- The numerical datatypes of \(A\), \(x\), and \(y\) must be identical and must not be `FLA_CONSTANT`.
- If \(\alpha\) 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.

**Imp. Notes:** This function is implemented as a wrapper to `FLA_Her2_external()`.

**Arguments:**
- `uplo` – Indicates whether the lower or upper triangle of \(A\) is referenced during the operation.
- `alpha` – An `FLA_Obj` representing scalar \(\alpha\).
- `x` – An `FLA_Obj` representing vector \(x\).
- `y` – An `FLA_Obj` representing vector \(y\).
- `A` – An `FLA_Obj` representing matrix \(A\).
void FLA_Her2c( FLA_Uplo uplo, FLA_Conj conj, FLA_Obj alpha, FLA_Obj x, FLA_Obj y, FLA_Obj A );

Purpose: Perform one of the following extended Hermitian rank-2 update (her2) operations:

\[
A := A + \alpha xy^H + \overline{\alpha}yx^H
\]
\[
A := A + \alpha \bar{x}y^T + \overline{\alpha} \bar{y}x^T
\]

where \( \alpha \) is a scalar, \( A \) is a Hermitian matrix, and \( x \) and \( y \) are general vectors. The \texttt{uplo} argument indicates whether the lower or upper triangle of \( A \) is referenced and updated by the operation. The \texttt{trans} argument allows the computation of the conjugated rank-2 products \( \bar{x}y^T \) and \( \bar{y}x^T \).

Notes: When invoked with real objects, this function performs the \texttt{SYR2} operation.

Constraints:

- The numerical datatypes of \( A \), \( x \), and \( y \) must be identical and must not be \texttt{FLA_CONSTANT}.
- If \( \alpha \) is not of datatype \texttt{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.
- \texttt{trans} may not be \texttt{FLA_CONJ_TRANSPOSE} or \texttt{FLA_CONJ_NO_TRANSPOSE}.

Imp. Notes: This function is implemented as a wrapper to \texttt{FLA_Her2c外部}().

Arguments:

- \texttt{uplo} – Indicates whether the lower or upper triangle of \( A \) is referenced during the operation.
- \texttt{trans} – Indicates whether the operation proceeds as if the rank-2 products are conjugated.
- \texttt{alpha} – An \texttt{FLA_Obj} representing scalar \( \alpha \).
- \texttt{x} – An \texttt{FLA_Obj} representing vector \( x \).
- \texttt{y} – An \texttt{FLA_Obj} representing vector \( y \).
- \texttt{A} – An \texttt{FLA_Obj} 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 (symv) operation:

\[
y := \beta y + \alpha A x
\]

where \( \alpha \) and \( \beta \) are scalars, \( A \) is a symmetric matrix, and \( x \) and \( y \) are general vectors. The \( \text{uplo} \) argument indicates whether the lower or upper triangle of \( A \) is referenced by the operation.

Constraints:
- The numerical datatypes of \( A \), \( x \), and \( y \) must be identical and floating-point, and must not be \text{FLA\_CONSTANT}.
- If \( \alpha \) and \( \beta \) are not of datatype \text{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.

Imp. Notes: This function is implemented as a wrapper to \text{FLA\_Symv\_external}().

Arguments:
- \( \text{uplo} \) – Indicates whether the lower or upper triangle of \( A \) is referenced during the operation.
- \( \text{alpha} \) – An \text{FLA\_Obj} representing scalar \( \alpha \).
- \( A \) – An \text{FLA\_Obj} representing matrix \( A \).
- \( x \) – An \text{FLA\_Obj} representing vector \( x \).
- \( \text{beta} \) – An \text{FLA\_Obj} representing scalar \( \beta \).
- \( y \) – An \text{FLA\_Obj} 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 (syr) operation:

\[
A := A + \alpha x x^T
\]

where \( \alpha \) is a scalar, \( A \) is a symmetric matrix, and \( x \) is a general vector. The \( \text{uplo} \) argument indicates whether the lower or upper triangle of \( A \) is referenced and updated by the operation.

Constraints:
- The numerical datatypes of \( A \) and \( x \) must be identical and floating-point, and must not be \text{FLA\_CONSTANT}.
- If \( \alpha \) is not of datatype \text{FLA\_CONSTANT}, then it must match the datatypes of \( A \) and \( x \).
- The length of \( x \) and the order of \( A \) must be equal.

Imp. Notes: This function is implemented as a wrapper to \text{FLA\_Syr\_external}().

Arguments:
- \( \text{uplo} \) – Indicates whether the lower or upper triangle of \( A \) is referenced during the operation.
- \( \text{alpha} \) – An \text{FLA\_Obj} representing scalar \( \alpha \).
- \( x \) – An \text{FLA\_Obj} representing vector \( x \).
- \( A \) – An \text{FLA\_Obj} 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 (SYR2) operation:

\[ A := A + \alpha xy^T + \alpha yx^T \]

where \( \alpha \) is a scalar, \( A \) is a symmetric matrix, and \( x \) and \( y \) are general vectors. The \( \text{uplo} \) argument indicates whether the lower or upper triangle of \( A \) is referenced and updated by the operation.

**Constraints:**
- The numerical datatypes of \( A \), \( x \), and \( y \) must be identical and floating-point, and must not be \text{FLA_CONSTANT}.
- If \( \alpha \) is not of datatype \text{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.

**Imp. Notes:** This function is implemented as a wrapper to \text{FLA_Syr2_external}().

**Arguments:**

- \( \text{uplo} \) – Indicates whether the lower or upper triangle of \( A \) is referenced during the operation.
- \( \text{alpha} \) – An \text{FLA_Obj} representing scalar \( \alpha \).
- \( x \) – An \text{FLA_Obj} representing vector \( x \).
- \( y \) – An \text{FLA_Obj} representing vector \( y \).
- \( A \) – An \text{FLA_Obj} 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 (TRMV) operations:

\[
\begin{align*}
  x &:= Ax \\
  x &:= A^T x \\
  x &:= \bar{A} x \\
  x &:= A^H x
\end{align*}
\]

where \( A \) is a triangular matrix and \( x \) is a general vector. The \( \text{uplo} \) argument indicates whether the lower or upper triangle of \( A \) is referenced by the operation. The \( \text{transa} \) argument allows the computation to proceed as if \( A \) were conjugated and/or transposed. The \( \text{diag} \) argument indicates whether the diagonal of \( A \) is unit or non-unit.

**Constraints:**
- The numerical datatypes of \( A \) and \( x \) must be identical and floating-point, and must not be \text{FLA_CONSTANT}.
- The length of \( x \) and the order of \( A \) must be equal.
- \( \text{diag} \) may not be \text{FLA_ZERO_DIAG}.

**Imp. Notes:** This function is implemented as a wrapper to \text{FLA_Trmv_external}().

**Arguments:**

- \( \text{uplo} \) – Indicates whether the lower or upper triangle of \( A \) is referenced during the operation.
- \( \text{transa} \) – Indicates whether the operation proceeds as if \( A \) were conjugated and/or transposed.
- \( \text{diag} \) – Indicates whether the diagonal of \( A \) is unit or non-unit.
- \( A \) – An \text{FLA_Obj} representing matrix \( A \).
- \( x \) – An \text{FLA_Obj} representing vector \( x \).
void FLA_Trmvsx( FLA_Upl o 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 (TRMV) operations:

\[
\begin{align*}
    y &= \beta y + \alpha Ax \\
    y &= \beta y + \alpha A^T x \\
    y &= \beta y + \bar{\alpha} Ax \\
    y &= \beta y + \alpha A^H x
\end{align*}
\]

where \(\alpha\) and \(\beta\) 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.

Constraints:

- The numerical datatypes of \(A\), \(x\), and \(y\) must be identical and floating-point, and must not be \(FLA\_CONSTANT\).
- If \(\alpha\) and \(\beta\) 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 \(FLA\_ZERO\_DIAG\).

Imp. Notes: This function is implemented as a wrapper to \(FLA\_Trmvsx\_external()\).

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 \(FLA\_Obj\) representing scalar \(\alpha\).
- \(A\) – An \(FLA\_Obj\) representing matrix \(A\).
- \(x\) – An \(FLA\_Obj\) representing vector \(x\).
- \(beta\) – An \(FLA\_Obj\) representing scalar \(\beta\).
- \(y\) – An \(FLA\_Obj\) 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 solve (TRSV) operations:

\[
\begin{align*}
Ax & = b \\
A^T x & = b \\
\bar{A}x & = b \\
A^H x & = b \\
\end{align*}
\]

which, respectively, are solved by overwriting \( b \) with the contents of the solution vector \( x \) as follows:

\[
\begin{align*}
b & := A^{-1}b \\
b & := A^{-T}b \\
b & := \bar{A}^{-1}b \\
b & := A^{-H}b \\
\end{align*}
\]

where \( A \) is a triangular matrix and \( x \) and \( b \) are general vectors. The \textit{uplo} argument indicates whether the lower or upper triangle of \( A \) is referenced by the operation. The \textit{transa} argument allows the computation to proceed as if \( A \) were conjugated and/or transposed. The \textit{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.
- \textit{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:

- \textit{uplo} – Indicates whether the lower or upper triangle of \( A \) is referenced during the operation.
- \textit{transa} – Indicates whether the operation proceeds as if \( A \) were conjugated and/or transposed.
- \textit{diag} – Indicates whether the diagonal of \( A \) is unit or non-unit.
- \( A \) – An FLA_Obj representing matrix \( A \).
- \( b \) – An FLA_Obj 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 solve (TRSV) operations:

\[
\begin{align*}
    y &= \beta y + \alpha A^{-1}b \\
    y &= \beta y + \alpha A^{-T}b \\
    y &= \beta y + \alpha \bar{A}^{-1}b \\
    y &= \beta y + \alpha A^{-H}b
\end{align*}
\]

where \( \alpha \) and \( \beta \) are scalars, \( A \) is a triangular matrix, and \( b \) and \( y \) are general vectors. The \text{uplo} argument indicates whether the lower or upper triangle of \( A \) is referenced by the operation. The \text{transa} argument allows the computation to proceed as if \( A \) were conjugated and/or transposed. The \text{diag} argument indicates whether the diagonal of \( A \) is unit or non-unit.

Constraints:
- The numerical datatypes of \( A \), \( b \), and \( y \) must be identical and floating-point, and must not be \text{FLA_CONSTANT}.
- If \( \alpha \) and \( \beta \) are not of datatype \text{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.
- \text{diag} may not be \text{FLA_ZERO_DIAG}.

Imp. Notes: This function is implemented as a wrapper to \text{FLA_Trsvsx_external}().

Arguments:

- \text{uplo} – Indicates whether the lower or upper triangle of \( A \) is referenced during the operation.
- \text{transa} – Indicates whether the operation proceeds as if \( A \) were conjugated and/or transposed.
- \text{diag} – Indicates whether the diagonal of \( A \) is unit or non-unit.
- \text{alpha} – An \text{FLA_Obj} representing scalar \( \alpha \).
- \text{A} – An \text{FLA_Obj} representing matrix \( A \).
- \text{b} – An \text{FLA_Obj} representing vector \( b \).
- \text{beta} – An \text{FLA_Obj} representing scalar \( \beta \).
- \text{y} – An \text{FLA_Obj} representing vector \( y \).
5.6.1.3 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:

\[
\begin{align*}
C & := \beta C + \alpha A B \\
C & := \beta C + \alpha A B^T \\
C & := \beta C + \alpha A \bar{B} \\
C & := \beta C + \alpha A B^H \\
C & := \beta C + \alpha A^T B \\
C & := \beta C + \alpha A^T B^T \\
C & := \beta C + \alpha A^T \bar{B} \\
C & := \beta C + \alpha A^T B^H
\end{align*}
\]

where \( \alpha \) and \( \beta \) are scalars and \( A, B, \) and \( C \) are general matrices. The \texttt{transa} and \texttt{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 \texttt{FLA_CONSTANT}.
- If \( \alpha \) and \( \beta \) are not of datatype \texttt{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: \texttt{FLA_Gemm()} expects \( A, B, \) and \( C \) to be flat matrix objects.

Imp. Notes: \texttt{FLA_Gemm()} invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS. \texttt{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 \texttt{FLA_Gemm_external()} to perform the computation on these blocks.

Arguments:

- \texttt{transa} – Indicates whether the operation proceeds as if \( A \) were conjugated and/or transposed.
- \texttt{transb} – Indicates whether the operation proceeds as if \( B \) were conjugated and/or transposed.
- \texttt{alpha} – An \texttt{FLA_Obj} representing scalar \( \alpha \).
- \texttt{A} – An \texttt{FLA_Obj} representing matrix \( A \).
- \texttt{B} – An \texttt{FLA_Obj} representing matrix \( B \).
- \texttt{beta} – An \texttt{FLA_Obj} representing scalar \( \beta \).
- \texttt{C} – An \texttt{FLA_Obj} 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:

\[
C := \beta C + \alpha AB \\
C := \beta C + \alpha BA
\]

where \(\alpha\) and \(\beta\) are scalars, \(A\) is a Hermitian matrix, and \(B\) and \(C\) are general matrices. The \texttt{side} argument indicates whether matrix \(A\) is multiplied on the left or the right side of \(B\). The \texttt{uplo} argument indicates whether the lower or upper triangle of \(A\) is referenced by the operation.

Notes: When invoked with real objects, this function performs the \texttt{SYMM} operation.

Constraints:
- The numerical datatypes of \(A\), \(B\), and \(C\) must be identical and must not be \texttt{FLA_CONSTANT}.
- If \(\alpha\) and \(\beta\) are not of datatype \texttt{FLA_CONSTANT}, then they must match the datatypes of \(A\), \(B\), and \(C\).
- The dimensions of \(C\) and \(B\) must be conformal.
- If \texttt{side} equals \texttt{FLA_LEFT}, then the number of rows in \(C\) and the order of \(A\) must be equal; otherwise, if \texttt{side} equals \texttt{FLA_RIGHT}, then the number of columns in \(C\) and the order of \(A\) must be equal.

Int. Notes: \texttt{FLA_Hemm()} expects \(A\), \(B\), and \(C\) to be flat matrix objects.

Imp. Notes: \texttt{FLA_Hemm()} invokes a single FLAME/C variant to induce a blocked algorithm with sub-problems performed by calling wrappers to external BLAS. \texttt{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:
- \texttt{side} – Indicates whether \(A\) is multiplied on the left or right side of \(B\).
- \texttt{uplo} – Indicates whether the lower or upper triangle of \(A\) is referenced during the operation.
- \texttt{alpha} – An \texttt{FLA_Obj} representing scalar \(\alpha\).
- \texttt{A} – An \texttt{FLA_Obj} representing matrix \(A\).
- \texttt{B} – An \texttt{FLA_Obj} representing matrix \(B\).
- \texttt{beta} – An \texttt{FLA_Obj} representing scalar \(\beta\).
- \texttt{C} – An \texttt{FLA_Obj} 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:

\[
\begin{align*}
C & := \beta C + \alpha A A^H \\
C & := \beta C + \alpha A^H A
\end{align*}
\]

where \( \alpha \) and \( \beta \) are scalars, \( C \) is a Hermitian matrix, and \( A \) is a general matrix. The \texttt{uplo} argument indicates whether the lower or upper triangle of \( C \) is referenced and updated by the operation. The \texttt{trans} argument allows the computation to proceed as if \( A \) were conjugate-transposed, which results in the alternate rank-k product \( A^H A \).

Notes: When invoked with real objects, this function performs the SYRK operation.

Constraints:
- The numerical datatypes of \( A \) and \( C \) must be identical and must not be \texttt{FLA_CONSTANT}.
- If \( \alpha \) and \( \beta \) are not of datatype \texttt{FLA_CONSTANT}, then they must be real and match the precision of the datatypes of \( A \) and \( C \).
- If \texttt{trans} equals \texttt{FLA_NO_TRANSPOSE}, then the order of matrix \( C \) and the number of rows in \( A \) must be equal; otherwise, if \texttt{trans} equals \texttt{FLA_CONJ_TRANSPOSE}, then the order of matrix \( C \) and the number of columns in \( A \) must be equal.

Int. Notes: \texttt{FLA_Herk()} expects \( A \) and \( C \) to be flat matrix objects.

Imp. Notes: \texttt{FLA_Herk()} invokes a single FLAME/C variant to induce a blocked algorithm with sub-problems performed by calling wrappers to external BLAS. \texttt{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:
- \texttt{uplo} – Indicates whether the lower or upper triangle of \( C \) is referenced during the operation.
- \texttt{transa} – Indicates whether the operation proceeds as if \( A \) were conjugate-transposed.
- \texttt{alpha} – An \texttt{FLA_Obj} representing scalar \( \alpha \).
- \texttt{A} – An \texttt{FLA_Obj} representing matrix \( A \).
- \texttt{beta} – An \texttt{FLA_Obj} representing scalar \( \beta \).
- \texttt{C} – An \texttt{FLA_Obj} 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:

\[
\begin{align*}
C & := \beta C + \alpha AB^H + \bar{\alpha} B A^H \\
C & := \beta C + \alpha A^H B + \bar{\alpha} B^H A
\end{align*}
\]

where \( \alpha \) and \( \beta \) are scalars, \( C \) is a Hermitian matrix, and \( A \) and \( B \) are general matrices. The \( \text{uplo} \) argument indicates whether the lower or upper triangle of \( C \) is referenced and updated by the operation. The \( \text{trans} \) 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 \( B^H A \).

Notes: When invoked with real objects, this function performs the \text{SYR2K} operation.

Constraints:
- The numerical datatypes of \( A \), \( B \), and \( C \) must be identical and must not be \text{FLA_CONSTANT}.
- If \( \alpha \) and \( \beta \) are not of datatype \text{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 \( \text{trans} \) equals \text{FLA_NO_TRANSPOSE}, then the order of matrix \( C \) and the number of rows in \( A \) and \( B \) must be equal; otherwise, if \( \text{trans} \) equals \text{FLA_CONJ_TRANSPOSE}, then the order of matrix \( C \) and the number of columns in \( A \) and \( B \) must be equal.

Int. Notes: \text{FLA_Her2k()} expects \( A \), \( B \), and \( C \) to be flat matrix objects.

Imp. Notes: \text{FLA_Her2k()} invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS. \text{FLASH_Her2k()} uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the \text{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:
- \( \text{uplo} \) – Indicates whether the lower or upper triangle of \( C \) is referenced during the operation.
- \( \text{trans} \) – Indicates whether the operation proceeds as if \( A \) and \( B \) were conjugate-transposed.
- \( \alpha \) – An \text{FLA_Obj} representing scalar \( \alpha \).
- \( A \) – An \text{FLA_Obj} representing matrix \( A \).
- \( B \) – An \text{FLA_Obj} representing matrix \( B \).
- \( \beta \) – An \text{FLA_Obj} representing scalar \( \beta \).
- \( C \) – An \text{FLA_Obj} 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:

\[
C := \beta C + \alpha AB \\
C := \beta C + \alpha BA
\]

where \( \alpha \) and \( \beta \) are scalars, \( A \) is a symmetric matrix, and \( B \) and \( C \) are general matrices. The \texttt{side} argument indicates whether the symmetric matrix \( A \) is multiplied on the left or the right side of \( B \). The \texttt{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 \texttt{FLA_CONSTANT}.
- If \( \alpha \) and \( \beta \) are not of datatype \texttt{FLA_CONSTANT}, then they must match the datatypes of \( A \), \( B \), and \( C \).
- The dimensions of \( C \) and \( B \) must be conformal.
- If \texttt{side} equals \texttt{FLA_LEFT}, then the number of rows in \( C \) and the order of \( A \) must be equal; otherwise, if \texttt{side} equals \texttt{FLA_RIGHT}, then the number of columns in \( C \) and the order of \( A \) must be equal.

Int. Notes:  \texttt{FLA_Symm()} expects \( A \), \( B \), and \( C \) to be flat matrix objects.

Imp. Notes:  \texttt{FLA_Symm()} invokes a single FLAME/C variant to induce a blocked algorithm with sub-problems performed by calling wrappers to external BLAS. \texttt{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:
- \texttt{side} – Indicates whether \( A \) is multiplied on the left or right side of \( B \).
- \texttt{uplo} – Indicates whether the lower or upper triangle of \( A \) is referenced during the operation.
- \texttt{alpha} – An \texttt{FLA_Obj} representing scalar \( \alpha \).
- \texttt{A} – An \texttt{FLA_Obj} representing matrix \( A \).
- \texttt{B} – An \texttt{FLA_Obj} representing matrix \( B \).
- \texttt{beta} – An \texttt{FLA_Obj} representing scalar \( \beta \).
- \texttt{C} – An \texttt{FLA_Obj} 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:

\[
\begin{align*}
C &= \beta C + \alpha AA^T \\
C &= \beta C + \alpha A^T A
\end{align*}
\]

where \(\alpha\) and \(\beta\) are scalars, \(C\) is a symmetric matrix, and \(A\) is a general matrix. The \text{uplo} argument indicates whether the lower or upper triangle of \(C\) is referenced and updated by the operation. The \text{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 \(\alpha\) and \(\beta\) are not of datatype FLA\_CONSTANT, then they must match the datatypes of \(A\) and \(C\).
- If \text{trans} equals FLA\_NO\_TRANSPOSE, then the order of matrix \(C\) and the the number of rows in \(A\) must be equal; otherwise, if \text{trans} equals FLA\_TRANSPOSE, then the order of matrix \(C\) and the number of columns in \(A\) must be equal.
- \text{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 sub-problems 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:
- \text{uplo} – Indicates whether the lower or upper triangle of \(C\) is referenced during the operation.
- \text{transa} – Indicates whether the operation proceeds as if \(A\) is transposed.
- \text{alpha} – An FLA\_Obj representing scalar \(\alpha\).
- \text{A} – An FLA\_Obj representing matrix \(A\).
- \text{beta} – An FLA\_Obj representing scalar \(\beta\).
- \text{C} – An FLA\_Obj 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:

\[
C := \beta C + \alpha AB^T + \alpha BA^T \\
C := \beta C + \alpha A^T B + \alpha B^T A
\]

where \(\alpha\) and \(\beta\) are scalars, \(C\) is a symmetric matrix, and \(A\) and \(B\) are general matrices. The \(\text{uplo}\) argument indicates whether the lower or upper triangle of \(C\) is referenced and updated by the operation. The \(\text{trans}\) 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 \(\alpha\) and \(\beta\) 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 \(\text{trans}\) 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 \(\text{trans}\) equals FLA_TRANSPOSE, then the order of matrix \(C\) and the number of columns in \(A\) and \(B\) must be equal.
- \(\text{trans}\) 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:
- \(\text{uplo}\) – Indicates whether the lower or upper triangle of \(C\) is referenced during the operation.
- \(\text{trans}\) – Indicates whether the operation proceeds as if \(A\) and \(B\) were transposed.
- \(\alpha\) – An FLA_Obj representing scalar \(\alpha\).
- \(A\) – An FLA_Obj representing matrix \(A\).
- \(B\) – An FLA_Obj representing matrix \(B\).
- \(\beta\) – An FLA_Obj representing scalar \(\beta\).
- \(C\) – An FLA_Obj representing matrix \(C\).
void FLA_Trmn( FLA_Side side, FLA_Uplo uplo, FLA_Trans trans,
               FLA_Diag diag, FLA_Obj alpha, FLA_Obj A, FLA_Obj B );
void FLASH_Trmn( 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:

\[
\begin{align*}
B & := \alpha AB \\
B & := \alpha A^T B \\
B & := \alpha \bar{A} \bar{B} \\
B & := \alpha A^H B \\
\end{align*}
\]

where \(\alpha\) is a scalar, \(A\) is a triangular matrix, and \(B\) is a general matrix. The \textit{side} argument indicates whether the triangular matrix \(A\) is multiplied on the left or the right side of \(B\). The \textit{uplo} argument indicates whether the lower or upper triangle of \(A\) is referenced by the operation. The \textit{trans} argument may be used to perform the check as if \(A\) were conjugated and/or transposed. The \textit{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 \text{FLA\_CONSTANT}.
- If \(\alpha\) is not of datatype \text{FLA\_CONSTANT}, then it must match the datatypes of \(A\) and \(B\).
- If \textit{side} equals \text{FLA\_LEFT}, then the number of rows in \(B\) and the order of \(A\) must be equal; otherwise, if \textit{side} equals \text{FLA\_RIGHT}, then the number of columns in \(B\) and the order of \(A\) must be equal.
- \textit{diag} may not be \text{FLA\_ZERO\_DIAG}.

Int. Notes: FLA_Trmn() expects \(A\) and \(B\) to be flat matrix objects.

Imp. Notes: FLA_Trmn() invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS. FLASH_Trmn() 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:
- \textit{side} – Indicates whether \(A\) is multiplied on the left or right side of \(B\).
- \textit{uplo} – Indicates whether the lower or upper triangle of \(A\) is referenced during the operation.
- \textit{trans} – Indicates whether the operation proceeds as if \(A\) were conjugated and/or transposed.
- \textit{diag} – Indicates whether the diagonal of \(A\) is unit or non-unit.
- \textit{alpha} – An \text{FLA\_Obj} representing scalar \(\alpha\).
- \textit{A} – An \text{FLA\_Obj} representing matrix \(A\).
- \textit{B} – An \text{FLA\_Obj} representing matrix \(B\).
void FLA_Trmmsx( 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:

\[
\begin{align*}
C & \leftarrow \beta C + \alpha AB \\
C & \leftarrow \beta C + \alpha A^T B \\
C & \leftarrow \beta C + \alpha \bar{A} B \\
C & \leftarrow \beta C + \alpha A^H B
\end{align*}
\]

where \( \alpha \) and \( \beta \) are scalars, \( A \) is a triangular matrix, and \( B \) and \( C \) are general matrices. The \texttt{side} argument indicates whether the triangular matrix \( A \) is multiplied on the left or the right side of \( B \). The \texttt{uplo} argument indicates whether the lower or upper triangle of \( A \) is referenced by the operation. The \texttt{trans} argument allows the computation to proceed as if \( A \) were conjugated and/or transposed. The \texttt{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 \texttt{FLA_CONSTANT}.
- If \( \alpha \) and \( \beta \) are not of datatype \texttt{FLA_CONSTANT}, they must match the datatypes of \( A \), \( B \), and \( C \).
- If \texttt{side} equals \texttt{FLA_LEFT}, then the number of rows in \( B \) and the order of \( A \) must be equal; otherwise, if \texttt{side} equals \texttt{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.
- \texttt{diag} may not be \texttt{FLA_ZERO_DIAG}.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine \(?trmm()\) along with the level-1 BLAS routines \(?copy(), \*scal(), \text{ and } ?axpy()\).

Arguments:
- \texttt{side} – Indicates whether \( A \) is multiplied on the left or right side of \( B \).
- \texttt{uplo} – Indicates whether the lower or upper triangle of \( A \) is referenced during the operation.
- \texttt{trans} – Indicates whether the operation proceeds as if \( A \) were conjugated and/or transposed.
- \texttt{diag} – Indicates whether the diagonal of \( A \) is unit or non-unit.
- \texttt{alpha} – An \texttt{FLA_Obj} representing scalar \( \alpha \).
- \texttt{A} – An \texttt{FLA_Obj} representing matrix \( A \).
- \texttt{B} – An \texttt{FLA_Obj} representing matrix \( B \).
- \texttt{beta} – An \texttt{FLA_Obj} representing scalar \( \beta \).
- \texttt{C} – An \texttt{FLA_Obj} representing matrix \( C \).
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 solve with multiple right-hand sides (TRSM) operations:

\[
\begin{align*}
AX &= \alpha B & XA &= \alpha B \\
A^T X &= \alpha B & XA^T &= \alpha B \\
\bar{A} X &= \alpha B & X\bar{A} &= \alpha B \\
A^H X &= \alpha B & XA^H &= \alpha B
\end{align*}
\]

and overwrite B with the contents of the solution matrix X as follows:

\[
\begin{align*}
B &= \alpha A^{-1} B & B &= \alpha BA^{-1} \\
B &= \alpha A^{-T} B & B &= \alpha BA^{-T} \\
B &= \alpha \bar{A}^{-1} B & B &= \alpha B\bar{A}^{-1} \\
B &= \alpha A^{-H} B & B &= \alpha BA^{-H}
\end{align*}
\]

where \(\alpha\) is a scalar, \(A\) is a triangular matrix, and \(X\) and \(B\) are general matrices. The \texttt{side} argument indicates whether the triangular matrix \(A\) is multiplied on the left or the right side of \(X\). The \texttt{uplo} argument indicates whether the lower or upper triangle of \(A\) is referenced by the operation. The \texttt{trans} argument allows the computation to proceed as if \(A\) were conjugated and/or transposed. The \texttt{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 \texttt{FLA_CONSTANT}.
- If \(\alpha\) is not of datatype \texttt{FLA_CONSTANT}, then it must match the datatypes of \(A\) and \(B\).
- If \texttt{side} equals \texttt{FLA_LEFT}, then the number of rows in \(B\) and the order of \(A\) must be equal; otherwise, if \texttt{side} equals \texttt{FLA_RIGHT}, then the number of columns in \(B\) and the order of \(A\) must be equal.
- \texttt{diag} may not be \texttt{FLA_ZERO_DIAG}.

Int. Notes: \texttt{FLA_Trmm()} expects \(A\) and \(B\) to be flat matrix objects.

Imp. Notes: \texttt{FLA_Trsm()} invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS. \texttt{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:
- \texttt{side} – Indicates whether \(A\) is multiplied on the left or right side of \(X\).
- \texttt{uplo} – Indicates whether the lower or upper triangle of \(A\) is referenced during the operation.
- \texttt{trans} – Indicates whether the operation proceeds as if \(A\) were conjugated and/or transposed.
- \texttt{diag} – Indicates whether the diagonal of \(A\) is unit or non-unit.
- \texttt{alpha} – An \texttt{FLA_Obj} representing scalar \(\alpha\).
- \texttt{A} – An \texttt{FLA_Obj} representing matrix \(A\).
- \texttt{B} – An \texttt{FLA_Obj} representing matrix \(B\).
void FLA_Trsmsx( 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 solve with multiple right-hand sides (TRSM) operations:

\[
AX = \alpha B \\
A^T X = \alpha B \\
\bar{AX} = \alpha B \\
A^H X = \alpha B \\
\bar{A}X = \alpha B \\
A^{-1} X = \alpha B \\
A^{-T} X = \alpha B \\
A^{-H} X = \alpha B
\]

and update \(C\) with the contents of the solution matrix \(X\) as follows:

\[
C := \beta C + \alpha A^{-1} B \\
C := \beta C + \alpha A^{-T} B \\
C := \beta C + \alpha A^{-1} B \\
C := \beta C + \alpha A^{-H} B
\]

where \(\alpha\) and \(\beta\) are scalars, \(A\) is a triangular matrix, and \(X, B,\) and \(C\) are general matrices. The \texttt{side} argument indicates whether the triangular matrix \(A\) is multiplied on the left or the right side of \(X\). The \texttt{uplo} argument indicates whether the lower or upper triangle of \(A\) is referenced by the operation. The \texttt{trans} argument allows the computation to proceed as if \(A\) were conjugated and/or transposed. The \texttt{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 \texttt{FLA_CONSTANT}.
- If \(\alpha\) and \(\beta\) are not of datatype \texttt{FLA_CONSTANT}, then they must match the datatypes of \(A, B,\) and \(C\).
- If \texttt{side} equals \texttt{FLA_LEFT}, then the number of rows in \(B\) and the order of \(A\) must be equal; otherwise, if \texttt{side} equals \texttt{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.
- \texttt{diag} may not be \texttt{FLA_ZERO_DIAG}.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine \(?\text{trsm}()\) along with the level-1 BLAS routines \(?\text{copy}(), \*\text{scal}(),\) and \(?\text{axpy}()\).

Arguments:
- \texttt{side} - Indicates whether \(A\) is multiplied on the left or right side of \(X\).
- \texttt{uplo} - Indicates whether the lower or upper triangle of \(A\) is referenced during the operation.
- \texttt{trans} - Indicates whether the operation proceeds as if \(A\) were conjugated and/or transposed.
- \texttt{diag} - Indicates whether the diagonal of \(A\) is unit or non-unit.
- \texttt{alpha} - An \texttt{FLA_Obj} representing scalar \(\alpha\).
- \texttt{A} - An \texttt{FLA_Obj} representing matrix \(A\).
- \texttt{B} - An \texttt{FLA_Obj} representing matrix \(B\).
- \texttt{beta} - An \texttt{FLA_Obj} representing scalar \(\beta\).
- \texttt{C} - An \texttt{FLA_Obj} representing matrix \(C\).
5.6.2 LAPACK operations

**FLA_Chol** (FLA_Uplo uplo, FLA_Obj A);
**FLASH_Chol** (FLA_Uplo uplo, FLA_Obj A);

**Purpose:** Perform one of the following Cholesky factorizations (\texttt{chol}):
\[
\begin{align*}
A & \rightarrow LL^T \\
A & \rightarrow U^TU \\
A & \rightarrow LL^H \\
A & \rightarrow U^HU
\end{align*}
\]
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 \texttt{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. The row/column index is zero-based, and thus its possible range extends inclusively from 0 to \(n - 1\).

**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 one or more FLAME/C variants to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS routines. FLASH_Chol() uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the \texttt{chol} operation into subproblems expressed in terms of individual blocks of \(A\) and then invokes external BLAS routines to perform the computation on these blocks. By default, the unblocked Cholesky subproblems are computed by internal implementations. However, if the external-lapack-for-subproblems option is enabled at configure-time, these subproblems are computed by external unblocked LAPACK routines.

**Arguments:**
- **uplo** — Indicates whether the lower or upper triangle of \(A\) is referenced and overwritten during the operation.
- **A** — An FLA_Obj representing matrix \(A\).
FLA_Error FLA_Chol_solve( FLA_Uplo uplo, FLA_Obj A, FLA_Obj B, FLA_Obj X );
FLA_Error FLASH_Chol_solve( FLA_Uplo uplo, FLA_Obj A, FLA_Obj B, FLA_Obj X );

**Purpose:** Solve one or more symmetric (or Hermitian) positive definite linear systems,

\[ AX = B \]

by applying the results of a Cholesky factorization stored in \( A \) to a set of right-hand sides stored in \( B \). Thus, the solution vectors overwrite \( X \) according to one of the following operations:

\[
\begin{align*}
X & := L^{-T}L^{-1}B \\
X & := U^{-1}U^{-T}B
\end{align*}
\]

where \( L \) and \( U \) are the lower and upper triangles of \( A \). The operation references only one triangle of \( A \), depending on the value of \( uplo \). This value for \( uplo \) should be the same as the \( uplo \) argument passed to FLA_Chol() or FLASH_Chol().

**Notes:** It is assumed that the prior Cholesky factorization which wrote to \( A \) completed successfully.

**Returns:** FLA_SUCCESS

**Constraints:**
- The numerical datatypes of \( A \), \( B \), and \( X \) must be identical and floating-point, and must not be FLA_CONSTANT.
- \( A \) must be square.
- The number of rows in \( B \) and \( X \) must be equal to the order of \( A \), and the number of columns in \( B \) and \( X \) must be equal.

**Int. Notes:** FLA_Chol_solve() expects \( A \), \( B \), and \( X \) to be flat matrix objects.

**Arguments:**
- `uplo` - Indicates whether the lower or upper triangle of \( A \) is referenced during the operation.
- `A` - An FLA_Obj representing matrix \( A \).
- `B` - An FLA_Obj representing matrix \( B \).
- `X` - An FLA_Obj representing matrix \( X \).
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() and FLA_LU_nopiv() do not perform pivoting and are therefore numerically unstable. Almost all applications should use FLA_LU_piv() or FLASH_LU_piv() instead.

Returns: FLA_SUCCESS if \( A \) is nonsingular; otherwise, a signed integer corresponding to the row/column index of the first zero diagonal entry in \( U \). The row/column index is zero-based, and thus its possible range extends inclusively from 0 to \( \min(m,n) - 1 \).

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 one or more FLAME/C variants to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS 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 routines to perform the computation on these blocks. By default, the unblocked LU factorization subproblems are computed by internal implementations. However, if the external-lapack-for-subproblems option is enabled at configure-time, these subproblems are computed by external unblocked LAPACK routines.

Arguments:
\[ A \quad \text{— An FLA_Obj representing matrix } A. \]
FLA_Error FLA_LU_piv( FLA_Obj A, FLA_Obj p );
FLA_Error FLASH_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() and FLASH_LU_piv() fill 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 \( p_i + i \). 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(). (However, this routine only works with flat pivot vectors, and thus a hierarchically-stored pivot vector must first be flattened.)

Returns: FLA_SUCCESS if \( A \) is nonsingular; otherwise, a signed integer corresponding to the row/column index of the first zero diagonal entry in \( U \). The row/column index is zero-based, and thus its possible range extends inclusively from 0 to \( \min(m,n) - 1 \).

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 one or more FLAME/C variants to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS routines. FLASH_LU_piv() uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the LUPIV operation into subproblems expressed in terms of individual blocks (or panels of blocks) of \( A \) and then invokes external BLAS routines to perform the computation on these blocks. By default, the unblocked LU factorization subproblems are computed by internal implementations. However, if the external-lapack-for-subproblems option is enabled at configure-time, these subproblems are computed by external unblocked LAPACK routines.

Arguments:
- \( A \) – An FLA_Obj representing matrix \( A \).
- \( p \) – An FLA_Obj representing vector \( p \).
FLA_Error FLA_LU_piv_solve( FLA_Obj A, FLA_Obj p, FLA_Obj B, FLA_Obj X );
FLA_Error FLASH_LU_piv_solve( FLA_Obj A, FLA_Obj p, FLA_Obj B, FLA_Obj X );

Purpose: Solve one or more general linear systems,

\[ AX = B \]

by applying the results of an LU factorization (with partial pivoting) stored in \( A \) and \( p \) to a set of right-hand sides stored in \( B \). Thus, the solution vectors overwrite \( X \) according to the following operation:

\[ X := U^{-1}L^{-1}PB \]

where \( L \) is the strictly lower triangle (with unit diagonal) of \( A \), \( U \) is the upper triangle of \( A \), and \( P \) represents the permutation matrix which applies the row interchanges encoded in the pivot vector \( p \).

Notes: It is assumed that the prior LU factorization which wrote to \( A \) completed successfully.

Returns: FLA_SUCCESS

Constraints:
- The numerical datatypes of \( A \), \( B \), and \( X \) must be identical and floating-point, and must not be FLA_CONSTANT.
- The numerical datatype of \( p \) must be FLA_INT.
- The length of \( p \) must be \( \min(m, n) \) where \( A \) is \( m \times n \).
- The number of rows in \( B \) and \( X \) must be equal to the order of \( A \), and the number of columns in \( B \) and \( X \) must be equal.

Int. Notes: FLA_LU_piv_solve() expects \( A \), \( p \), \( B \), and \( X \) to be flat matrix objects.

Arguments:

- \( A \) – An FLA_Obj representing matrix \( A \).
- \( p \) – An FLA_Obj representing vector \( p \).
- \( B \) – An FLA_Obj representing matrix \( B \).
- \( X \) – An FLA_Obj representing matrix \( X \).
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 \( A \) (APIV).

\[
\begin{align*}
A &:= PA \\
A &:= P^T A \\
A &:= AP \\
A &:= A P^T 
\end{align*}
\]

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 \texttt{FLA_Shift pivots to()} before it may be used with \texttt{FLA_Apply pivots unb external()}. Please see the description for \texttt{FLA_LU piv()} in Section 5.6.2 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 \texttt{FLA_CONSTANT}.
- The numerical datatype of \( p \) must be \texttt{FLA_INT}.

Int. Notes: \texttt{FLA_Apply pivots()} expects \( A \) to be a flat matrix object.

Imp. Notes: By default, the APIV operation is performed by an internal implementation. However, if the \texttt{external-lapack-for-subproblems} option is enabled at configure-time, the operation is performed by an external unblocked LAPACK routine.

Caveats: This function is currently only implemented for applying \( P \) from the left (ie: \texttt{side} equal to \texttt{FLA_LEFT} and \texttt{trans} equal to \texttt{FLA_NO_TRANSPOSE}).

Arguments:

- **side** – Indicates whether the operation proceeds as if the permutation matrix \( P \) is applied from the left or the right.
- **trans** – Indicates whether the operation proceeds as if the permutation matrix \( P \) were transposed.
- **p** – An \texttt{FLA_Obj} representing vector \( p \).
- **A** – An \texttt{FLA_Obj} representing matrix \( A \).
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 $L_{inter}$ from the original flat matrix.

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. The row/column index is zero-based, and thus its possible range extends inclusively from 0 to min($m, n$) – 1.

Constraints:
- The numerical datatypes of $A$ and $L_{inter}$ 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.

Int. Notes: In addition to the input matrix $A$ and pivot vector $p$, the function requires an additional object $L_{inter}$, 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.

Arguments:

- $A$ – A hierarchical FLA_Obj representing matrix $A$.
- $L_{inter}$ – A hierarchical FLA_Obj representing matrix $L_{inter}$.
5.6. Front-ends

FLA_Error FLASH_LU_incpiv_solve( FLA_Obj A, FLA_Obj p, FLA_Obj L_inter,
                                    FLA_Obj B, FLA_Obj X );

Purpose: Solve one or more general linear systems,

\[ AX = B \]

by applying the results of an LU factorization with incremental pivoting stored in \( A \), \( p \), and \( L_{inter} \) to a set of right-hand sides stored in \( B \). Thus, the solution vectors overwrite \( X \) according to the following operation:

\[ X := U^{-1} L^{-1} P B \]

where \( L \) is the strictly lower triangle (with unit diagonal) of \( A \), \( U \) is the upper triangle of \( A \), and \( P \) represents the permutation matrix which applies the row interchanges encoded in the pivot vector \( p \).

Notes: Note that `FLASH_LU_incpiv_solve()` may only be used in conjunction with matrices that have been factorized via `FLASH_LU_incpiv()`. The output from `FLA_LU_piv()` is not compatible with this function.

Returns: FLA_SUCCESS

Constraints:
- The numerical datatypes of \( A \), \( L_{inter} \), \( B \), and \( X \) 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.
- The number of rows in \( B \) and \( X \) must be equal to the number of columns in \( A \), and the number of columns in \( B \) and \( X \) must be equal.

Caveats: Currently, this function only supports matrices with hierarchical depths of exactly 1.

Arguments:

- \( A \) – A hierarchical FLA_Obj representing matrix \( A \).
- \( p \) – A hierarchical FLA_Obj representing vector \( p \).
- \( L_{inter} \) – A hierarchical FLA_Obj representing matrix \( L_{inter} \).
- \( B \) – A hierarchical FLA_Obj representing matrix \( B \).
- \( X \) – A hierarchical FLA_Obj representing matrix \( X \).
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 FLA_Obj representing matrix $A$.
- $L_{inter}$ – A hierarchical FLA_Obj representing matrix $L_{inter}$.
- $b$ – A hierarchical FLA_Obj representing vector $b$. 
void FLA_QR_UT( FLA_Obj A, FLA_Obj T );
void FLASH_QR_UT( FLA_Obj A, FLA_Obj T );

Purpose: Perform a QR factorization via the UT transform (qrut):

\[ A \rightarrow QR \]

where \( Q \) is an orthogonal matrix (or, a unitary matrix if \( A \) is complex) and \( R \) is an upper triangular matrix. The resulting Householder vectors associated with \( Q \) are stored column-wise below the diagonal of \( A \) and should only be used with other UT transform operations. Upon completion, matrix \( T \) contains the triangular factors of the block Householder transformations that were used in the factorization algorithm.

Notes: The matrix factor \( Q \) determined by \texttt{FLA\_QR\_UT()} and \texttt{FLASH\_QR\_UT()} is equal to \( H_0H_1\cdots H_{k-1} \), where \( H_i \) is the Householder transformation which annihilates the sub-diagonal entries in the \( i \)th column of the original matrix \( A \).

Constraints:
- The numerical datatypes of \( A \) and \( T \) must be identical and floating-point, and must not be \texttt{FLA\_CONSTANT}.
- The width of \( T \) must be equal to the width of \( A \).

Int. Notes: \texttt{FLA\_QR\_UT()} expects \( A \) and \( T \) to be flat matrix objects.

Imp. Notes: \texttt{FLA\_QR\_UT()} invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS routines. The unblocked qrut subproblems are computed by internal implementations. \texttt{FLASH\_QR\_UT()} uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the qrut operation into subproblems expressed in terms of individual blocks (or panels of blocks) of \( A \) and then invokes external BLAS routines to perform the computation on these blocks. The unblocked qrut subproblems are computed by internal implementations. (External LAPACK routines are not used, even when \texttt{external-lapack-for-subproblems} option is enabled.)

Imp. Notes: For \texttt{FLA\_QR\_UT()} , the algorithmic blocksize is determined by the length of \( T \). When in doubt, create \( T \) via \texttt{FLA\_QR\_UT\_create\_T()}. For \texttt{FLASH\_QR\_UT()} , the algorithmic blocksize \( b \), which corresponds to the scalar length a single block of \( T \), must be equal to the storage blocksize used in \( A \) and \( T \). When in doubt, create \( T \) via \texttt{FLASH\_QR\_UT\_create\_hier\_matrices()}. 

Arguments:
- \( A \) – An \texttt{FLA\_Obj} representing matrix \( A \).
- \( T \) – An \texttt{FLA\_Obj} representing matrix \( T \).
void FLA_QR_UT_solve( FLA_Obj A, FLA_Obj T, FLA_Obj B, FLA_Obj X );
void FLASH_QR_UT_solve( FLA_Obj A, FLA_Obj T, FLA_Obj B, FLA_Obj X );

**Purpose:** Solve one or more general linear systems,

\[ AX = B \]

by applying the results of a QR factorization via the UT transform stored in \( A \) and \( T \) to a set of right-hand sides stored in \( B \). Thus, the solution vectors overwrite \( X \) according to the following operation:

\[ X := R^{-1}Q^H B \]

where \( R \) is an upper triangular matrix, stored in \( A \), and \( Q \) is an orthogonal (or unitary) matrix formed from the upper triangular Householder factors in \( T \) and the Householder vectors stored column-wise below the diagonal of \( A \).

**Notes:** Note that FLA_QR_UT_solve() and FLASH_QR_UT_solve() may only be used in conjunction with matrices that have been factorized via FLA_QR_UT() and FLASH_QR_UT(), respectively. The output from FLASH_QR_UT_inc() is *not* compatible with these functions.

**Returns:** FLA_SUCCESS

**Constraints:**
- The numerical datatypes of \( A, T, B, \) and \( X \) must be identical and floating-point, and must not be FLA_CONSTANT.
- The width of \( T \) must be equal to the width of \( A \).
- The number of rows in \( A \) and \( B \) must be equal, the number of columns of \( A \) and the number of rows of \( X \) must be equal, and the number of columns in \( X \) and \( B \) must be equal.

**Int. Notes:** FLA_QR_UT_solve() expects \( A, T, B, \) and \( X \) to be flat matrix objects.

**Arguments:**

- \( A \) – An FLA_Obj representing matrix \( A \).
- \( T \) – An FLA_Obj representing matrix \( T \).
- \( B \) – An FLA_Obj representing matrix \( B \).
- \( X \) – An FLA_Obj representing matrix \( X \).
void FLASH_QR_UT_inc( FLA_Obj A, FLA_Obj TW );

**Purpose:** Perform an incremental QR factorization via the UT transform (qrutinc). The operation is similar to the operation implemented by FLA_QR_UT(), 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.

**Constraints:**
- The numerical datatypes of A and TW must be identical and 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.

**Int. Notes:** In addition to the input matrix A, the function requires an additional matrix TW to hold the triangular factors of the 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:** FLASH_QR_UT_inc() uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the qrutinc operation into subproblems expressed in terms of individual blocks of A and then invokes external BLAS routines to perform the computation on these blocks. The unblocked qrut subproblems are computed by internal implementations. (External LAPACK routines are not used, even when external-lapack-for-subproblems option is enabled.)

**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.

**Arguments:**
- **A** – A hierarchical FLA_Obj representing matrix A.
- **TW** – A hierarchical FLA_Obj representing matrix TW.
void FLASH_QR_UT_inc_solve( FLA_Obj A, FLA_Obj TW, FLA_Obj B, FLA_Obj X );

**Purpose:** Solve one or more general linear systems,

\[ AX = B \]

by applying the results of an incremental QR factorization via the UT transform stored in \( A \) and \( TW \) to a set of right-hand sides stored in \( B \). Thus, the solution vectors overwrite \( X \) according to the following operation:

\[ X := R^{-1}Q^H B \]

where \( R \) is an upper triangular matrix, stored in \( A \), and \( Q \) is an orthogonal (or unitary) matrix formed from the upper triangular Householder factors in \( TW \) and the Householder vectors stored column-wise below the diagonal of \( A \).

**Notes:** Note that \( \text{FLASH_QR_UT_inc_solve()} \) may only be used in conjunction with matrices that have been factorized via \( \text{FLASH_QR_UT()} \). The output from \( \text{FLA_QR_UT()} \) is *not* compatible with this function.

**Returns:** FLA_SUCCESS

**Constraints:**
- The numerical datatypes of \( A \), \( TW \), \( B \), and \( X \) must be identical and 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.
- The number of rows in \( A \) and \( B \) must be equal, the number of columns of \( A \) and the number of rows of \( X \) must be equal, and the number of columns in \( X \) and \( B \) must be equal.

**Caveats:** Currently, this function only supports matrices with hierarchical depths of exactly 1.

**Arguments:**

- \( A \) – A hierarchical FLA_Obj representing matrix \( A \).
- \( TW \) – A hierarchical FLA_Obj representing matrix \( TW \).
- \( B \) – A hierarchical FLA_Obj representing matrix \( B \).
- \( X \) – A hierarchical FLA_Obj representing matrix \( X \).
void FLA_LQ_UT( FLA_Obj A, FLA_Obj T );
void FLASH_LQ_UT( FLA_Obj A, FLA_Obj T );

**Purpose:**
Perform a LQ factorization via the UT transform (LQUT):

\[ A \rightarrow LQ^H \]

where \( Q \) is an orthogonal matrix (or, a unitary matrix if \( A \) is complex) and \( L \) is an lower triangular matrix. The resulting Householder vectors associated with \( Q \) are stored row-wise above the diagonal of \( A \) and should only be used with other UT transform operations. Upon completion, matrix \( T \) contains the triangular factors of the block Householder transformations that were used in the factorization algorithm.

**Notes:**
The matrix factor \( Q \) determined by \texttt{FLA\_LQ\_UT()} and \texttt{FLASH\_LQ\_UT()} is equal to \( H_0H_1\cdots H_{k-1} \), where \( H_i \) is the Householder transformation which annihilates the superdiagonal entries in the \( i \)th row of the original matrix \( A \).

**Constraints:**
- The numerical datatypes of \( A \) and \( T \) must be identical and floating-point, and must not be \texttt{FLA\_CONSTANT}.
- The width of \( T \) must be equal to the length of \( A \).

**Int. Notes:** \texttt{FLA\_QR\_UT()} expects \( A \) and \( T \) to be flat matrix objects.

**Imp. Notes:** \texttt{FLA\_LQ\_UT()} invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS routines. The unblocked LQUT subproblems are computed by internal implementations. \texttt{FLASH\_LQ\_UT()} uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the LQUT operation into subproblems expressed in terms of individual blocks (or panels of blocks) of \( A \) and then invokes external BLAS routines to perform the computation on these blocks. The unblocked LQUT subproblems are computed by internal implementations. (External LAPACK routines are not used, even when \texttt{external-lapack-for-subproblems} option is enabled.)

**Imp. Notes:** For \texttt{FLA\_LQ\_UT()} , the algorithmic blocksize is determined by the length of \( T \). When in doubt, create \( T \) via \texttt{FLA\_LQ\_UT\_create\_T()}. 

**Imp. Notes:** For \texttt{FLASH\_LQ\_UT()} , the algorithmic blocksize \( b \), which corresponds to the scalar length a single block of \( T \), must be equal to the storage blocksize used in \( A \) and \( T \). When in doubt, create \( T \) via \texttt{FLASH\_LQ\_UT\_create\_hier\_matrices()}. 

**Arguments:**

- \( A \) – An \texttt{FLA\_Obj} representing matrix \( A \).
- \( T \) – An \texttt{FLA\_Obj} representing matrix \( T \).
void FLA_LQ_UT_solve( FLA_Obj A, FLA_Obj T, FLA_Obj B, FLA_Obj X );
void FLASH_LQ_UT_solve( FLA_Obj A, FLA_Obj T, FLA_Obj B, FLA_Obj X );

Purpose: Solve one or more general linear systems,

\[ AX = B \]

by applying the results of a LQ factorization via the UT transform stored in \( A \) and \( T \) to a set of right-hand sides stored in \( B \). Thus, the solution vectors overwrite \( X \) according to the following operation:

\[ X := QL^{-1}B \]

where \( L \) is an lower triangular matrix, stored in \( A \), and \( Q \) is an orthogonal (or unitary) matrix formed from the upper triangular Householder factors in \( T \) and the Householder vectors stored row-wise above the diagonal of \( A \).

Notes: Note that FLA_LQ_UT_solve() and FLASH_LQ_UT_solve() may only be used in conjunction with matrices that have been factorized via FLA_LQ_UT() and FLASH_LQ_UT(), respectively.

Returns: FLA_SUCCESS

Constraints:
- The numerical datatypes of \( A \), \( T \), \( B \), and \( X \) must be identical and floating-point, and must not be FLA_CONSTANT.
- The width of \( T \) must be equal to the length of \( A \).
- The number of rows in \( A \) and \( B \) must be equal, the number of columns of \( A \) and the number of rows of \( X \) must be equal, and the number of columns in \( X \) and \( B \) must be equal.

Int. Notes: FLA_LQ_UT_solve() expects \( A \), \( T \), \( B \), and \( X \) to be flat matrix objects.

Arguments:
- A – An FLA_Obj representing matrix \( A \).
- T – An FLA_Obj representing matrix \( T \).
- B – An FLA_Obj representing matrix \( B \).
- X – An FLA_Obj representing matrix \( X \).
void FLASH_CAQR_UT_inc( dim_t p, FLA_Obj A, FLA_Obj ATW, FLA_Obj R, FLA_Obj RTW );

**Purpose:** Perform an incremental communication-avoiding QR factorization via the UT transform (caqrutinc). The operation is performed in two stages. First, the input matrix $A$ is partitioned into $p$ panels,

$$
A ightarrow \begin{pmatrix} 
A_0 \\
A_1 \\
\vdots \\
A_{p-1}
\end{pmatrix}
$$

and incremental QR factorizations are performed on each submatrix $A_i$, with the resulting Householder vectors overwriting the blocks of $A_i$ and the corresponding block Householder triangular factors stored to $ATW$. This results in $p$ upper triangular factors overwriting the upper triangles of $A_i$. In the second stage, the upper triangles of $A_i$ are copied to $R_i$, and then $R_1, \ldots, R_{p-1}$ are individually factored against submatrix $R_0$ in a manner similar to that used in an incremental QR factorization, except the upper triangular structure of each $R_i$ submatrix is leveraged to avoid computing with implicit zeros. The resulting Householder vectors of the second stage overwrite the various blocks of $R_1, \ldots, R_{p-1}$. 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_CAQR_UT_inc_create_hier_matrices()` to create hierarchical matrices $A$ and $TW$ from the original flat matrix.

**Constraints:**
- The numerical datatypes of $A$, $ATW$, $R$, and $RTW$ must be identical and floating-point, and must not be FLA_CONSTANT.
- $A$, $ATW$, $R$, and $RTW$ must each have the same number of blocks in the row and column dimensions.

**Int. Notes:** In addition to the input matrix $A$ and workspace/output matrix $R$, the function requires two additional matrices $ATW$ and $RTW$ to hold the triangular factors of the block Householder transformations computed for each storage block. These transformations are used when applying the matrices $Q_A$ and $Q_R$ associated with the first and second factorization stages. The matrices $ATW$ and $RTW$ also contain temporary workspace needed by both stages.

**Imp. Notes:** `FLASH_CAQR_UT_inc()` uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the `CAQRUTINC` operation into subproblems expressed in terms of individual blocks of $A$ and then invokes external BLAS routines to perform the computation on these blocks. The unblocked `CAQRUT subproblems are computed by internal implementations. (External LAPACK routines are not used, even when external-lapack-for-subproblems option is enabled.)

**Imp. Notes:** Strictly speaking, the blocks in the lower triangle (including the diagonal) of $ATW$ are used to store the block Householder transformations corresponding to $T$ in `FLA_QR_UT()` while the blocks in the upper triangle of $ATW$ are used as workspace only. This is the case for $RTW$ as well.

**Caveats:** Currently, this function only supports matrices with hierarchical depths of exactly 1.

**Arguments:**

- $p$ – An unsigned integer representing the number of panels into which $A$ is partitioned during the first stage of factorization.
- $A$ – A hierarchical FLA_Obj representing matrix $A$.
- $ATW$ – A hierarchical FLA_Obj representing matrix $ATW$.
- $R$ – A hierarchical FLA_Obj representing matrix $R$.
- $RTW$ – A hierarchical FLA_Obj representing matrix $RTW$. 

void FLASH_CAQR_UT_inc_solve( dim_t p, FLA_Obj A, FLA_Obj ATW, FLA_Obj R, FLA_Obj RTW,
                                FLA_Obj B, FLA_Obj X );

Purpose: Solve one or more general linear systems,

\[ AX = B \]

by applying the results of an incremental communication-avoiding QR factorization via
the UT transform stored in \( A \) and \( ATW \), and \( R \) and \( RTW \), to a set of right-hand sides
stored in \( B \). Thus, the solution vectors overwrite \( X \) according to the following operation:

\[ X := R_0^{-1}Q_R^H Q_A^H B \]

where \( R_0 \) is the first upper triangular submatrix in \( R \), \( Q_A \) is an orthogonal (or unitary)
matrix formed from the upper triangular Householder factors in \( ATW \) and the Householder
vectors stored column-wise below the diagonals of the \( p \) subpartitions of \( A \), and
\( Q_R \) is the orthogonal (or unitary) matrix formed from the upper triangular Householder
factors in \( RTW \) and the Householder vectors stored columnwise in the upper triangles
of the \( p \) subpartitions of \( R \).

Notes: Note that \texttt{FLASH\_CAQR\_UT\_inc\_solve()} may only be used in conjunction with matrices
that have been factorized via \texttt{FLASH\_CAQR\_UT\_inc()}. The output from \texttt{FLA\_QR\_UT()} and
\texttt{FLASH\_QR\_UT\_inc()} are \textit{not} compatible with this function.

Returns: \texttt{FLA\_SUCCESS}

Constraints:
- The numerical datatypes of \( A \), \( ATW \), \( R \), \( RTW \), \( B \), and \( X \) must be identical and
floating-point, and must not be \texttt{FLA\_CONSTANT}.
- \( A \), \( ATW \), \( R \), and \( RTW \) must each have the same number of blocks in the row and
column dimensions.
- The number of rows in \( A \) and \( B \) must be equal, the number of columns of \( A \) and the
number of rows of \( X \) must be equal, and the number of columns in \( X \) and \( B \) must be
equal.

Caveats: Currently, this function only supports matrices with hierarchical depths of exactly 1.

Arguments:
- \( p \) – An unsigned integer representing the number of panels into which \( A \)
is partitioned during the first stage of factorization.
- \( A \) – A hierarchical \texttt{FLA\_Obj} representing matrix \( A \).
- \( ATW \) – A hierarchical \texttt{FLA\_Obj} representing matrix \( ATW \).
- \( R \) – A hierarchical \texttt{FLA\_Obj} representing matrix \( R \).
- \( RTW \) – A hierarchical \texttt{FLA\_Obj} representing matrix \( RTW \).
- \( B \) – A hierarchical \texttt{FLA\_Obj} representing matrix \( B \).
- \( X \) – A hierarchical \texttt{FLA\_Obj} representing matrix \( X \).
void FLA_UDdate_UT( FLA_Obj R, FLA_Obj C, FLA_Obj D, FLA_Obj T );

**Purpose:** Perform an up-and-downdate (uddateut) of the upper triangular factor \( R \) (via up-and-downdating UT transforms) that arises from solving a linear least-squares problem, \( Ax = y \). Note that such a problem

\[
Ax = y
\]

is typically solved via one of two methods. In the first method, the Cholesky factor \( R \) of \( A^H A \) is used to solve

\[
A^H Ax = A^H y \\
R^H Rx = y
\]

In the second method, the QR factorization of \( A \) is used to solve

\[
Rx = Q^H y
\]

Let us assume that we begin with \( A \) and \( y \) such that

\[
\begin{pmatrix}
A \\
y
\end{pmatrix} = \begin{pmatrix}
B \\
b \\
D \\
d
\end{pmatrix}
\]

and \( R \) has already been computed, via one of the two methods above. Let us further assume that we wish to update \( R \) to reflect a new system consisting of \( \tilde{A} \) and \( \tilde{y} \) such that

\[
\begin{pmatrix}
\tilde{A} \\
\tilde{y}
\end{pmatrix} = \begin{pmatrix}
B \\
b \\
C \\
c
\end{pmatrix}
\]

The \texttt{uddateut} operation simultaneously (a) updates the upper triangular factor \( R \) to include the contributions of \( C \) and (b) downdates \( R \) to remove the contributions of \( D \) without explicitly performing a new factorization (Cholesky or QR) using \( \tilde{A} \). Upon completion, the operation will have overwritten the \( j \)-th columns of \( C \) and \( D \) with the vectors \( u_j \) and \( v_j \), respectively, associated with the up-and-downdating Householder transforms \( G_j \) used to annihilate the corresponding columns of \( C \) and \( D \). Similarly, the operation sets matrix \( T \) to contain the upper triangular factors of the block Householder transforms used in the up-and-downdate. These triangular factors are re-used when applying the transforms to the right-hand sides.

**Notes:**
This operation only up-and-downdates \( R \). To up-and-downdate the right-hand side of a linear least-squares system, use \texttt{FLA_UDdate_UT_update_rhs()}.  

**Constraints:**
- The numerical datatypes of \( R, C, D, \) and \( T \) must be identical and floating-point, and must not be \texttt{FLA\_CONSTANT}.
- \( R \) must be square.
- The widths of \( R, C, D, \) and \( T \) must be equal.

**Int. Notes:** \texttt{FLA_UDdate_UT()} expects \( R, C, D, \) and \( T \) to be flat matrix objects.

**Imp. Notes:** \texttt{FLA_UDdate_UT()} invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS routines. The unblocked \texttt{uddateut} subproblems are computed by internal implementations. (External LAPACK routines are not used, even when \texttt{external-lapack-for-subproblems} option is enabled.)

**Imp. Notes:** The algorithmic blocksize \( b \) is determined by the length of \( T \). When in doubt, create \( T \) via \texttt{FLA_UDdate_UT\_create_T()}.  

**Arguments:**
- \( R \) – An \texttt{FLA\_Obj} representing matrix \( R \).
- \( C \) – An \texttt{FLA\_Obj} representing matrix \( C \).
- \( D \) – An \texttt{FLA\_Obj} representing matrix \( D \).
- \( T \) – An \texttt{FLA\_Obj} representing matrix \( T \).
void FLA_UDdate_UT_update_rhs( FLA_Obj T, FLA_Obj bR,
                                FLA_Obj C, FLA_Obj bC,
                                FLA_Obj D, FLA_Obj bD );

Purpose: Perform an up-and-downdate of the right-hand maintained when solving a linear least-squares system $Ax = y$. Note that the right-hand side that is updated, $b_R$, is initially computed either as

$$A^H y = b_R$$

when the method of normal equations is used, or

$$Q^H y = b_R$$

when a QR factorization is used, where $y = \begin{pmatrix} b_B \\ b_D \end{pmatrix}$. This operation assumes the user wishes to be able to solve a new system, $\tilde{A}x = \tilde{y}$, that would result in $\tilde{R}^H \tilde{R}x = \tilde{A}^H \tilde{y}$ (normal equations) or $\tilde{R}x = \tilde{Q}^H \tilde{y}$ (QR factorization), where $\tilde{y} = \begin{pmatrix} b_B \\ b_C \end{pmatrix}$, and $\tilde{R}$ has already been computed from the original matrix $R$ by FLA_UDdate_UT(). Thus, FLA_UDdate_UT_update_rhs() updates $b_R$ such that it removes the contributions of $b_D$ and includes the contributions of of $b_C$. In other words, upon completion, $b_R$ contains the values it would have contained as if it had been computed via fully-formed $\tilde{y}$ and $\tilde{A}$. Note that the operation preserves the original values of $b_C$ and $b_D$.

Notes: FLA_UDdate_UT_update_rhs() should be invoked using the $C$, $D$, and $T$ matrices that were updated by the FLA_UDdate_UT() during the up-and-downdate of the upper triangular factor $R$. Subsequent to the up-and-downdate of the right-hand side, the user may use FLA_UDdate_UT_solve() to solve the updated system.

Constraints:
- The numerical datatypes of $T$, $C$, $D$, $b_R$, $b_C$, and $b_D$ must be identical and floating-point, and must not be FLA_CONSTANT.
- The widths of $T$, $C$, and $D$ must be equal.
- The widths of $b_R$, $b_C$, and $b_D$ must be equal.
- The length of $b_R$ must equal the width of $T$; the length of $b_C$ must equal the length of $C$; and the length of $b_D$ must equal the length of $D$.

Int. Notes: FLA_UDdate_UT() expects $R$, $C$, $D$, and $T$ to be flat matrix objects.

Imp. Notes: FLA_UDdate_UT_update_rhs() is implemented as a convenience wrapper to FLA_Apply_QUD_UT_create_workspace() and FLA_Apply_QUD_UT().

Imp. Notes: The algorithmic blocksize $b$ is determined by the length of $T$. When in doubt, create $T$ via FLA_UDdate_UT_create_T().

Arguments:
- $T$ – An FLA_Obj representing matrix $T$.
- $b_R$ – An FLA_Obj representing matrix $b_R$.
- $C$ – An FLA_Obj representing matrix $C$.
- $b_C$ – An FLA_Obj representing matrix $b_C$.
- $D$ – An FLA_Obj representing matrix $D$.
- $b_D$ – An FLA_Obj representing matrix $b_D$. 
void FLA_UDdate_UT_solve( FLA_Obj R, FLA_Obj bR, FLA_Obj x );

**Purpose:** Solve one or more linear least-squares systems using the upper triangular factor $R$ and the right-hand side $b_R$. Presumably the user has already up-and-downdated $R$, via `FLA_UDdate_UT()`, and $b_R$, via `FLA_UDdate_UT_update_rhs()`.

**Notes:** Note that `FLA_UDdate_UT_solve()` may only be used in conjunction with matrices that have been factorized via `FLA_UDdate_UT()`. The output from `FLASH_UDdate_UT_inc()` is not compatible with this function.

**Returns:** `FLA_SUCCESS`

**Constraints:**
- The numerical datatypes of $R$, $b_R$, and $x$ must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The order of $R$ and the length of $b_R$ must be equal; the width of $b_R$ and the width of $x$ must be equal.

**Int. Notes:** `FLA_UDdate_UT_solve()` expects $R$, $b_R$, and $x$ to be flat matrix objects.

**Arguments:**
- $R$ – An `FLA_Obj` representing matrix $R$.
- $b_R$ – An `FLA_Obj` representing matrix $b_R$.
- $x$ – An `FLA_Obj` representing matrix $x$. 
void FLASH_UDdate_UT_inc( FLA_Obj R, FLA_Obj C, FLA_Obj D, FLA_Obj T, FLA_Obj W );

Purpose: Perform an incremental up-and-downdate (uddateutinc) of the upper triangular factor $R$ (via up-and-downdating UT transforms) that arises from solving a linear least-squares problem, $Ax = y$. The operation is similar to the operation implemented by FLA_UDdate_UT(), 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 flat objects containing the matrices to be used in the up-and-downdate and then call FLASH_UDdate_UT_inc_create_hier_matrices() to create hierarchical matrices $R$, $C$, $D$, $T$, and $W$ from the original flat matrices.

Constraints:
- The numerical datatypes of $R$, $C$, $D$, $T$, and $W$ must be identical and floating-point, and must not be FLA_CONSTANT.
- $R$ must be square.
- The widths of $R$, $C$, $D$, and $T$ must be equal.
- The number of blocks in the column dimension of $T$ must be equal to the number of blocks in the column dimension of $R$; the number of blocks in the row dimension of $T$ must be equal to the greater of the number of blocks in the row dimension of $C$ and $D$.
- The block dimensions of $W$ must be conformal to that of $R$.

Int. Notes: In addition to the input matrices $R$, $C$, and $D$, the function requires an additional matrix $T$ to hold the upper triangular factors of the block up-and-downdating UT Householder transformations computed for each storage block. These transformations are used when applying $Q$ (via FLASH_Apply_QUD_UT_inc()). The matrix $W$ contains temporary workspace needed by the incremental up-and-downdating algorithm.

Imp. Notes: FLASH_UDdate_UT_inc() uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the uddateutinc operation into subproblems expressed in terms of individual blocks of $R$, $C$, and $D$, and then invokes external BLAS routines to perform the computation on these blocks. The unblocked uddateut subproblems are computed by internal implementations. (External LAPACK routines are not used, even when external-lapack-for-subproblems option is enabled.)

Caveats: Currently, this function only supports matrices with hierarchical depths of exactly 1.

Arguments:
- $R$ – A hierarchical FLA_Obj representing matrix $R$.
- $C$ – A hierarchical FLA_Obj representing matrix $C$.
- $D$ – A hierarchical FLA_Obj representing matrix $D$.
- $T$ – A hierarchical FLA_Obj representing matrix $T$.
- $W$ – A hierarchical FLA_Obj representing matrix $W$. 
void FLASH_UDdate_UT_inc_update_rhs( FLA_Obj T, FLA_Obj bR, 
FLA_Obj C, FLA_Obj bC, 
FLA_Obj D, FLA_Obj bD );

Purpose: Perform an incremental up-and-downdate of the right-hand maintained when solving a linear least-squares system $Ax = y$. The operation is similar to the operation implemented by FLA_UDdate_UT_update_rhs(), except that the algorithm is SuperMatrix-aware. As a consequence, the arguments must be hierarchical objects.

Notes: FLASH_UDdate_UT_inc_update_rhs() should be invoked using the $C$, $D$, and $T$ matrices that were updated by the FLASH_UDdate_UT_inc() during the up-and-downdate of the upper triangular factor $R$. Subsequent to the up-and-downdate of the right-hand side, the user may use FLASH_UDdate_UT_inc_solve() to solve the updated system.

Constraints:
- The numerical datatypes of $T$, $C$, $D$, $bR$, $bC$, and $bD$ must be identical and floating-point, and must not be FLA_CONSTANT.
- The widths of $T$, $C$, and $D$ must be equal.
- The widths of $bR$, $bC$, and $bD$ must be equal.
- The length of $bR$ must equal the width of $T$; the length of $bC$ must equal the length of $C$; and the length of $bD$ must equal the length of $D$.

Imp. Notes: FLASH_UDdate_UT_inc_update_rhs() is implemented as a convenience wrapper to FLASH_Apply_QUD_UT_inc_create_workspace() and FLASH_Apply_QUD_UT_inc().

Arguments:
- $T$ – A hierarchical FLA_Obj representing matrix $T$.
- $bR$ – A hierarchical FLA_Obj representing matrix $bR$.
- $C$ – A hierarchical FLA_Obj representing matrix $C$.
- $bC$ – A hierarchical FLA_Obj representing matrix $bC$.
- $D$ – A hierarchical FLA_Obj representing matrix $D$.
- $bD$ – A hierarchical FLA_Obj representing matrix $bD$.

void FLASH_UDdate_UT_inc_solve( FLA_Obj R, FLA_Obj bR, FLA_Obj x );

Purpose: Solve one or more linear least-squares systems using the upper triangular factor $R$ and the right-hand side $bR$. Presumably the user has already up-and-downdated $R$, via FLASH_UDdate_UT_inc(), and $bR$, via FLASH_UDdate_UT_inc_update_rhs(). The operation is similar to the operation implemented by FLA_UDdate_UT_solve(), except that the algorithm is SuperMatrix-aware. As a consequence, the arguments must be hierarchical objects.

Notes: Note that FLASH_UDdate_UT_inc_solve() may only be used in conjunction with matrices that have been factorized via FLASH_UDdate_UT_inc(). The output from FLASH_UDdate_UT() is not compatible with this function.

Returns: FLA_SUCCESS

Constraints:
- The numerical datatypes of $R$, $bR$, and $x$ must be identical and floating-point, and must not be FLA_CONSTANT.
- The order of $R$ and the length of $bR$ must be equal; the width of $bR$ and the width of $x$ must be equal.

Arguments:
- $R$ – A hierarchical FLA_Obj representing matrix $R$.
- $bR$ – A hierarchical FLA_Obj representing matrix $bR$.
- $x$ – A hierarchical FLA_Obj representing matrix $x$. 
void FLA_Hess_UT( FLA_Obj A, FLA_Obj T );

**Purpose:** Perform a reduction to upper Hessenberg form via the UT transform (hessut)

\[ A \rightarrow QRQ^H \]

where \( A \) is a general square matrix, \( Q \) is an orthogonal matrix (or, a unitary matrix if \( A \) is complex) and \( R \) is an upper Hessenberg matrix (zeroes below the first subdiagonal). The resulting Householder vectors associated with \( Q \) are stored column-wise below the first subdiagonal of \( A \) and should only be used with other UT transform operations. Upon completion, matrix \( T \) contains the upper triangular factors of the block Householder transformations that were used in the reduction algorithm.

**Notes:** When using \( \text{FLA\_Hess\_UT()} \), the Householder vectors associated with matrix \( Q \) are stored, in which case \( Q \) is equal to \( H_0H_1\cdots H_{k-2} \), where \( H_i^H \) is the Householder transformation which annihilates entries below the first subdiagonal in the \( i \)th column of the original matrix \( A \).

**Constraints:**

- The numerical datatypes of \( A \) and \( T \) must be identical and floating-point, and must not be \( \text{FLA\_CONSTANT} \).
- \( A \) must be square.
- The width of \( T \) must be \( n \) where \( A \) is \( n \times n \).

**Int. Notes:** \( \text{FLA\_Hess\_UT()} \) expects \( A \) and \( T \) to be flat matrix objects.

**Imp. Notes:** \( \text{FLA\_Hess\_UT()} \) invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS routines. The unblocked hessut subproblems are computed by internal implementations. (External LAPACK routines are not used, even when \text{external-lapack-for-subproblems} option is enabled.)

**Imp. Notes:** The algorithmic blocksize \( b \) is determined by the length of \( T \). When in doubt, create \( T \) via \( \text{FLA\_Hess\_UT\_create\_T()} \).

**Arguments:**

- \( A \) – An FLA_Obj representing matrix \( A \).
- \( T \) – An FLA_Obj representing matrix \( T \).
void FLA_Tridiag_UT( FLA_Uplo uplo, FLA_Obj A, FLA_Obj T );

**Purpose:** Perform a reduction to tridiagonal form via the UT transform (TRIDIAGUT)

\[ A \rightarrow QRQ^H \]

where \( A \) is a symmetric (or, if \( A \) is complex, Hermitian) matrix, \( Q \) is an orthogonal (or, if \( A \) is complex, unitary) matrix and \( R \) is a tridiagonal matrix (zeroes below the first subdiagonal and above the first superdiagonal). Note, however, that FLA_Tridiag_UT() only reads and updates the triangle specified by \( \text{uplo} \). The resulting Householder vectors associated with \( Q \) are stored column-wise below the first subdiagonal of \( A \) if \( \text{uplo} \) is FLA_LOWER_TRIANGULAR and row-wise above the first superdiagonal if \( \text{uplo} \) is FLA_UPPER_TRIANGULAR. Upon completion, matrix \( T \) contains the upper triangular factors of the block Householder transformations that were used in the reduction algorithm.

**Notes:** If \( A \) is complex, the tridiagonal matrix that results from the reduction operation contains complex sub- and super-diagonals (though, only one of which is stored, as specified by \( \text{uplo} \)). The matrix may further be reduced to real tridiagonal form via FLA_Tridiag_UT_realify().

**Constraints:**
- The numerical datatypes of \( A \) and \( T \) must be identical and floating-point, and must not be FLA_CONSTANT.
- \( A \) must be square.
- The width of \( T \) must be \( n \) where \( A \) is \( n \times n \).

**Int. Notes:** FLA_Tridiag_UT() expects \( A \) and \( T \) to be flat matrix objects.

**Imp. Notes:** FLA_Tridiag_UT() invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS routines. The unblocked TRIDIAGUT subproblems are computed by internal implementations. (External LAPACK routines are not used, even when external-lapack-for-subproblems option is enabled.)

**Imp. Notes:** The algorithmic blocksize \( b \) is determined by the length of \( T \). When in doubt, create \( T \) via FLA_Tridiag_UT_create_T().

**Arguments:**
- \( \text{uplo} \) – Indicates whether the lower or upper triangle of \( A \) is referenced and overwritten during the operation.
- \( A \) – An FLA_Obj representing matrix \( A \).
- \( T \) – An FLA_Obj representing matrix \( T \).
void FLA_Bidiag_UT( FLA_Obj A, FLA_Obj TU, FLA_Obj TV );

**Purpose:** Perform a reduction to bidiagonal form via the UT transform (bidiagut)

\[ A \rightarrow Q_U R Q_V^H \]

where \( A \) is a general \( m \times n \) matrix, \( Q_U \) and \( Q_V \) are orthogonal (or, if \( A \) is complex, unitary) matrices, and \( R \) is a bidiagonal matrix. If \( m \geq n \), \( R \) is upper bidiagonal (zeroes below the diagonal and above the first superdiagonal). Otherwise, if \( m < n \), \( R \) is lower bidiagonal (zeroes above the diagonal and below the first subdiagonal). When \( R \) is upper bidiagonal, the resulting Householder vectors associated with \( Q_U \) and \( Q_V \) are stored column-wise below the diagonal and row-wise above the first superdiagonal, respectively. When \( R \) is lower bidiagonal, the resulting Householder vectors associated with \( Q_U \) and \( Q_V \) are stored column-wise below the first subdiagonal and row-wise above the diagonal, respectively. Upon completion, matrices \( T_U \) and \( T_V \) contain the upper triangular factors of the block Householder transformations corresponding to \( Q_U \) and \( Q_V \), respectively, that were used in the reduction algorithm.

**Constraints:**
- The numerical datatypes of \( A \), \( T_U \), and \( T_V \) must be identical and floating-point, and must not be FLA_CONSTANT.
- The widths of \( T_U \) and \( T_V \) must be \( \min(m,n) \).

**Int. Notes:** FLA_Bidiag_UT() expects \( A \), \( T_U \), and \( T_V \) to be flat matrix objects.

**Imp. Notes:** FLA_Bidiag_UT() invokes a single FLAME/C variant to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS routines. The unblocked bidiagut subproblems are computed by internal implementations. (External LAPACK routines are not used, even when external-lapack-for-subproblems option is enabled.)

**Imp. Notes:** The algorithmic blocksize \( b \) is determined by the length of \( T_U \) and \( T_V \). When in doubt, create \( T_U \) and \( T_V \) via FLA_Bidiag_UT_create_T().

**Arguments:**
- \( A \) – An FLA_Obj representing matrix \( A \).
- \( T_U \) – An FLA_Obj representing matrix \( T_U \).
- \( T_V \) – An FLA_Obj representing matrix \( T_V \).
void FLA_Apply_Q_UT( FLA_Side side, FLA_Trans trans, FLA_Direct direct, FLA_Store storev, 
FLA_Obj A, FLA_Obj T, FLA_Obj W, FLA_Obj B );
void FLASH_Apply_Q_UT( FLA_Side side, FLA_Trans trans, FLA_Direct direct, FLA_Store storev, 
FLA_Obj A, FLA_Obj T, FLA_Obj W, FLA_Obj B );

Purpose: Apply a matrix $Q$ (or $Q^H$) to a general matrix $B$ from either the left or the right (APQUT):

$$B := QB \quad B := BQ$$

where $Q$ is the orthogonal (or, if $A$ is complex, unitary) matrix implicitly defined by the Householder vectors stored in matrix $A$ and the triangular factors stored in matrix $T$ by FLA_QR_UT() (or FLASH_QR_UT()) or FLA_LQ_UT() (or FLASH_LQ_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 $Q^H$ is applied to $B$. The direct argument indicates whether $Q$ is assumed to be the forward product $H_0H_1\cdots H_{k-1}$ or the backward product $H_{k-1}\cdots H_1H_0$ of Householder transforms, where $k$ is the width of $T$. The storev argument indicates whether the Householder vectors which correspond to $H_0H_1\cdots H_{k-1}$ are stored column-wise (in the strictly lower triangle, as computed by a QR factorization) or row-wise (in the strictly upper triangle, as computed by an LQ factorization) in $A$.

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 flat matrix objects.

Imp. Notes: FLA_Apply_Q_UT() invokes one or more FLAME/C variants to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS routines. FLASH_Apply_Q_UT() invokes one or more FLAME/C variants to induce an algorithm-by-blocks with subproblems performed by calling wrappers to external BLAS routines.

Arguments:
- side - Indicates whether $Q$ (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).
- direct - Indicates whether $Q$ is formed from the forward or backward product of its constituent Householder reflectors.
- storev - Indicates whether the vectors stored within $A$ are stored column-wise or row-wise.
- $T$ - An FLA_Obj representing matrix $T$.
- $W$ - An FLA_Obj representing matrix $W$.
- $B$ - An FLA_Obj representing matrix $B$. 
void FLASH_Apply_Q_UT_inc( FLA_Side side, FLA_Trans trans, FLA_Direct direct, 
FLA_Store storev, 
FLA_Obj A, FLA_Obj TW, FLA_Obj W, FLA_Obj B );

Purpose: Apply a matrix $Q$ (or $Q^H$) to a general matrix $B$ from either the left or the right (apqutinc):

$$
\begin{align*}
B & := QB \\
B & := Q^H B
\end{align*}
$$

where $Q$ is the orthogonal (or, if $A$ is complex, unitary) matrix implicitly defined by the Householder vectors stored in matrix $A$ and the triangular factors 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 $Q^H$ is applied to $B$. The direct argument indicates whether $Q$ was computed as the forward product $H_0 H_1 \cdots H_k - 1$ or the backward product $H_k - 1 \cdots H_1 H_0$.

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 cases where side is FLA_LEFT, direct is FLA_FORWARD, and storev is FLA_COLUMNWISE.

Arguments:

- side – Indicates whether $Q$ (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).
- direct – Indicates whether $Q$ is formed from the forward or backward product of its constituent Householder reflectors.
- storev – Indicates whether the vectors stored within $A$ are stored column-wise or row-wise.
- $A$ – A hierarchical FLA_Obj representing matrix $A$.
- $TW$ – A hierarchical FLA_Obj representing matrix $TW$.
- $W$ – A hierarchical FLA_Obj representing matrix $W$.
- $B$ – A hierarchical FLA_Obj representing matrix $B$. 
void FLA_Apply_QUD_UT( FLA_Side side, FLA_Trans trans, FLA_Direct direct, FLA_Store storev,
                     FLA_Obj T, FLA_Obj W,
                     FLA_Obj R,
                     FLA_Obj U, FLA_Obj C,
                     FLA_Obj V, FLA_Obj D );

Purpose: Apply a matrix $Q^H$ to general matrices $R$, $C$, and $D$ from the left (APQUDUT):

$$
\begin{pmatrix}
  R \\
  C \\
  D 
\end{pmatrix}
:=
Q^H
\begin{pmatrix}
  R \\
  C \\
  D 
\end{pmatrix}
$$

where $Q$ is the orthogonal (or, if the matrices are complex, unitary) matrix implicitly defined by the up-and-downdating UT Householder vectors stored columnwise in $U$ and $V$ and the upper triangular factors stored in matrix $T$ by FLA_UDdate_UT(). Matrix $W$ is used as workspace.

Constraints:
- The numerical datatypes of $T$, $W$, $R$, $U$, $C$, $V$, and $D$ must be identical and floating-point, and must not be FLA_CONSTANT.
- The number of columns in $T$ must be equal to the number of columns in $U$ and $V$.
- The number of columns in $W$ must be equal to the number of columns in $R$.
- The number of rows in $C$ and the number of rows in $U$ must be equal; the number of columns in $C$ and the number of columns of $R$ must be equal; and the number of columns in $U$ and the number of rows in $R$ must be equal.
- The number of rows in $D$ and the number of rows in $V$ must be equal; the number of columns in $D$ and the number of columns of $R$ must be equal; and the number of columns in $V$ and the number of rows in $R$ must be equal.

Int. Notes: FLA_Apply_QUD_UT() expects $T$, $W$, $R$, $U$, $C$, $V$, and $D$ to be flat matrix objects.

Imp. Notes: FLA_Apply_QUD_UT() invokes one or more FLAME/C variants to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS routines.

Caveats: FLA_Apply_QUD_UT() is currently only implemented for the case where side is FLA_LEFT, trans is FLA_CONJTRANSPOSE (or FLA_TRANSPOSE for real matrices), direct is FLA_FORWARD, and storev is FLA_COLUMNWISE.

Arguments:
- side - Indicates whether $Q$ (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).
- direct - Indicates whether $Q$ is formed from the forward or backward product of its constituent Householder reflectors.
- storev - Indicates whether the vectors stored within $U$ and $V$ are stored column-wise or row-wise.
- $T$ - An FLA_Obj representing matrix $T$.
- $W$ - An FLA_Obj representing matrix $W$.
- $R$ - An FLA_Obj representing matrix $R$.
- $U$ - An FLA_Obj representing matrix $U$.
- $C$ - An FLA_Obj representing matrix $C$.
- $V$ - An FLA_Obj representing matrix $V$.
- $D$ - An FLA_Obj representing matrix $D$. 
void FLASH_Apply_QUD_UT_inc( FLA_Side side, FLA_Trans trans, 
FLA_Direct direct, FLA_Store storev, 
FLA_Obj T, FLA_Obj W, 
     FLA_Obj R, 
FLA_Obj U, FLA_Obj C, 
FLA_Obj V, FLA_Obj D );

Purpose: Apply a matrix $Q^H$ to general matrices $R$, $C$, and $D$ from the left (APQUDUTINC):

\[
\begin{pmatrix}
R \\
C \\
D
\end{pmatrix} := Q^H \begin{pmatrix}
R \\
C \\
D
\end{pmatrix}
\]

where $Q$ is the orthogonal (or, if the matrices are complex, unitary) matrix implicitly defined by the up-and-downdating UT Householder vectors stored columnwise in $U$ and $V$ and the upper triangular factors stored in matrix $T$ by FLASH_UPDATE_UT_inc(). Matrix $W$ is used as workspace. The operation is similar to the operation implemented by FLA_Apply_QUD_UT(), except that the algorithm is SuperMatrix-aware. As a consequence, the arguments must be hierarchical objects.

Constraints:

- The numerical datatypes of $T$, $W$, $R$, $U$, $C$, $V$, and $D$ must be identical and floating-point, and must not be FLA_CONSTANT.
- The number of columns in $T$ must be equal to the number of columns in $U$ and $V$.
- The number of columns in $W$ must be equal to the number of columns in $R$.
- The number of rows in $C$ and the number of rows in $U$ must be equal; the number of columns in $C$ and the number of columns of $R$ must be equal; and the number of columns in $U$ and the number of rows in $R$ must be equal.
- The number of rows in $D$ and the number of rows in $V$ must be equal; the number of columns in $D$ and the number of columns of $R$ must be equal; and the number of columns in $V$ and the number of rows in $R$ must be equal.

Imp. Notes: FLASH_Apply_QUD_UT_inc() uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the APQUDUTINC operation into subproblems expressed in terms of individual blocks of $A$ and then invokes external BLAS routines to perform the computation on these blocks. (External LAPACK routines are not used, even when external-lapack-for-subproblems option is enabled.)

Caveats: FLASH_Apply_QUD_UT_inc() is currently only implemented for the case where side is FLA_LEFT, trans is FLA_CONJ_TRANSPOSE (or FLA_TRANSPOSE for real matrices), direct is FLA_FORWARD, and storev is FLA_COLUMNWISE.

Arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>side</td>
<td>Indicates whether $Q$ (or $Q^H$) is multiplied on the left or right side of $B$.</td>
</tr>
<tr>
<td>trans</td>
<td>Indicates whether the operation proceeds as if $Q$ were transposed (or conjugate-transposed).</td>
</tr>
<tr>
<td>direct</td>
<td>Indicates whether $Q$ is formed from the forward or backward product of its constituent Householder reflectors.</td>
</tr>
<tr>
<td>storev</td>
<td>Indicates whether the vectors stored within $U$ and $V$ are stored column-wise or row-wise.</td>
</tr>
<tr>
<td>T</td>
<td>A hierarchical FLA_Obj representing matrix $T$.</td>
</tr>
<tr>
<td>W</td>
<td>A hierarchical FLA_Obj representing matrix $W$.</td>
</tr>
<tr>
<td>R</td>
<td>A hierarchical FLA_Obj representing matrix $R$.</td>
</tr>
<tr>
<td>U</td>
<td>A hierarchical FLA_Obj representing matrix $U$.</td>
</tr>
<tr>
<td>C</td>
<td>A hierarchical FLA_Obj representing matrix $C$.</td>
</tr>
<tr>
<td>V</td>
<td>A hierarchical FLA_Obj representing matrix $V$.</td>
</tr>
<tr>
<td>D</td>
<td>A hierarchical FLA_Obj representing matrix $D$.</td>
</tr>
</tbody>
</table>
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):

\[
A := L^T L \\
A := UU^T \\
A := L^H L \\
A := UU^H
\]

where \( A \) is a triangular matrix with a real diagonal. The operation references and then
overwrites the lower or upper triangle of \( A \) with one of the products specified above,
depending on the value of \( \text{uplo} \).

Notes: \( \text{FLA}_\text{Ttmm}() \) may not be used for a general-purpose triangular matrix since the function
assumes that the diagonal of \( L \) (or \( U \)) is real.

Constraints:
- The numerical datatype of \( A \) must be floating-point, and must not be \( \text{FLA}_\text{CONSTANT} \).
- \( \text{diag} \) may not be \( \text{FLA}_\text{ZERO}_\text{DIAG} \).
- \( A \) must be square.

Int. Notes: \( \text{FLA}_\text{Ttmm}() \) expects \( A \) to be a flat matrix object.

Imp. Notes: \( \text{FLA}_\text{Ttmm}() \) invokes one or more FLAME/C variants to induce a blocked algorithm with
subproblems performed by calling wrappers to external BLAS routines. \( \text{FLASH}_\text{Ttmm}() \)
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 routines to perform the computation on these blocks.
By default, the unblocked TTMM subproblems are computed by internal implementations.
However, if the \text{external-lapack-for-subproblems} option is enabled at configure-time,
these subproblems are computed by external unblocked LAPACK routines.

Arguments:

\( \text{uplo} \) — Indicates whether the lower or upper triangle of \( A \) is referenced and
overwritten during the operation.

\( A \) — An \( \text{FLA}_\text{Obj} \) representing matrix \( A \).
Purposes: 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 \texttt{uplo}. The \texttt{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. The row/column index is zero-based, and thus its possible range extends inclusively from 0 to \( n - 1 \).

Constraints:
- The numerical datatype of \( A \) must be floating-point, and must not be FLA\_CONSTANT.
- \texttt{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 one or more FLAME/C variants to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS 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 routines to perform the computation on these blocks. By default, the unblocked TRINV subproblems are computed by internal implementations. However, if the \texttt{external-lapack-for-subproblems} option is enabled at configure-time, these subproblems are computed by external unblocked LAPACK routines.

Arguments:

- \texttt{uplo} – Indicates whether the lower or upper triangle of \( A \) is referenced and overwritten during the operation.
- \texttt{diag} – Indicates whether the diagonal of \( A \) is unit or non-unit.
- \( A \) – 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 \((\text{spdinv})\):

\[
A := A^{-1}
\]

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 the corresponding triangle of its inverse, \(A^{-1}\). The triangle referenced and overwritten is determined by the value of \(\text{uplo}\).

**Notes:** Given a real symmetric positive definite matrix \(A\), there exists a factor \(L\) such that \(A = LL^T\). Therefore,

\[
A^{-1} = (LL^T)^{-1} = L^{-T}L^{-1}
\]

Similarly, for a complex Hermitian positive definite matrix \(A\), there exists a factor such that \(A = LL^H\):

\[
A^{-1} = (LL^H)^{-1} = L^{-H}L^{-1}
\]

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 \(\text{FLASH_SPDinv()}\) 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 \(\text{FLASH_SPDinv()}\) will return the row/column index at which the algorithm detected a zero entry along the diagonal. In either case, the row/column index is zero-based, and thus its possible range extends inclusively from 0 to \(n - 1\). Otherwise, \(\text{FLASH_SPDinv()}\) returns \(\text{FLA\_SUCCESS}\) if the operation is successful.

**Constraints:**
- The numerical datatype of \(A\) must be floating-point, and must not be \(\text{FLA\_CONSTANT}\).
- \(A\) must be square.

**Int. Notes:** \(\text{FLA_SPDinv()}\) expects \(A\) to be a flat matrix object.

**Imp. Notes:** \(\text{FLA_SPDinv()}\) is implemented in terms of \(\text{FLA\_Chol()}\), \(\text{FLA\_Trinv()}\), and \(\text{FLA\_Ttmm()}\). \(\text{FLASH_SPDinv()}\) is implemented in terms of \(\text{FLASH\_Chol()}\), \(\text{FLASH\_Trinv()}\), and \(\text{FLASH\_Ttmm()}\).

**Arguments:**
- `uplo` – Indicates whether the lower or upper triangle of \(A\) is referenced and overwritten during the operation.
- `diag` – Indicates whether the diagonal of \(A\) is unit or non-unit.
- `A` – An \(\text{FLA\_Obj}\) representing matrix \(A\).
void FLA_Eig_gest( FLA_Inv inv, FLA_Uplo uplo, FLA_Obj A, FLA_Obj B );
void FLASH_Eig_gest( FLA_Inv inv, FLA_Uplo uplo, FLA_Obj A, FLA_Obj B );

**Purpose:** Perform one of the following operations to reduce a symmetric- or Hermitian-definite eigenproblem to standard form (eiggest):

\[
\begin{align*}
A & := L^H A L \\
A & := U A U^H \\
A & := L A L^{-H} \\
A & := U^{-H} A U
\end{align*}
\]

where \( A \), on input and output, is symmetric (or Hermitian) and \( B \) contains either a lower (\( L \)) or upper (\( U \)) triangular Cholesky factor. The value of \( \text{inv} \) determines whether the operation, as expressed above, requires an inversion of \( L \) or \( U \). The value of \( \text{uplo} \) determines which triangle of \( A \) is read on input, which triangle of the symmetric (or Hermitian) right-hand side is stored, and also which Cholesky factor exists in \( B \).

**Constraints:**
- The numerical datatypes of \( A \) and \( B \) must be identical and floating-point, and must not be FLA_CONSTANT.
- \( A \) and \( B \) must be square.

**Int. Notes:** FLA_Eig_gest() expects \( A \) and \( B \) to be flat matrix objects.

**Imp. Notes:** FLA_Eig_gest() invokes one or more FLAME/C variants to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS routines. FLASH_Eig_gest() uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the eiggest operation into subproblems expressed in terms of individual blocks of \( A \) and then invokes external BLAS routines to perform the computation on these blocks. By default, the unblocked eiggest subproblems are computed by internal implementations. However, if the external-lapack-for-subproblems option is enabled at configure-time, these subproblems are computed by external unblocked LAPACK routines.

**Arguments:**
- \( \text{inv} \) – Indicates whether the operation requires a multiplication by the inverse of \( L \) or \( U \).
- \( \text{uplo} \) – Indicates whether the lower or upper triangle of \( A \) is referenced and overwritten (and whether the lower or upper triangle of \( B \) is referenced) during the operation.
- \( A \) – An FLA_Obj representing matrix \( A \).
- \( B \) – An FLA_Obj representing matrix \( B \).
5.6. Front-ends

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):

\[
\begin{align*}
AX & \pm XB = C \\
AX & \pm XB^T = C \\
A^T X & \pm XB = C \\
A^T X & \pm XB^T = C
\end{align*}
\]

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:

\[
\begin{align*}
AX & \pm XB = C \\
AX & \pm XB^H = C \\
A^H X & \pm XB = C \\
A^H X & \pm XB^H = C
\end{align*}
\]

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 \texttt{isgn} argument is a scalar integer object that indicates whether the \( \pm \) sign between terms is a plus or a minus. The \texttt{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 \texttt{isgn} argument must be either FLA\_ONE or FLA\_MINUS\_ONE.
- The numerical datatype of \texttt{scale} must not be FLA\_CONSTANT. Furthermore, the precision of the datatype of \texttt{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 the number of rows in \( C \) and the number of columns in \( C \), respectively.
- \texttt{trans} may not be FLA\_CONJ\_NO\_TRANSPOSE.

Int. Notes: FLA\_Sylv() expects \( A \), \( B \), and \( C \) to be flat matrix objects.

Imp. Notes: FLA\_Sylv() invokes one or more FLAME/C variants to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS 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 routines to perform the computation on these blocks. By default, the unblocked SYLV subproblems are computed by internal implementations. However, if the \texttt{external-lapack-for-subproblems} option is enabled at configure-time, these subproblems are computed by external unblocked LAPACK routines.

Arguments:

\begin{itemize}
  \item \texttt{transa} – Indicates whether the operation proceeds as if \( A \) were [conjugate] transposed.
  \item \texttt{transb} – Indicates whether the operation proceeds as if \( B \) were [conjugate] transposed.
  \item \texttt{isgn} – Indicates whether the terms of the Sylvester equation are added or subtracted.
  \item \texttt{A} – An FLA\_Obj representing matrix \( A \).
  \item \texttt{B} – An FLA\_Obj representing matrix \( B \).
  \item \texttt{C} – An FLA\_Obj representing matrix \( C \).
  \item \texttt{scale} – Not referenced; set to 1.0 upon exit.
\end{itemize}
void FLA_Lyap( FLA_Trans trans, FLA_Obj isgn, FLA_Obj A, FLA_Obj C, FLA_Obj scale );
void FLASH_Lyap( FLA_Trans trans, FLA_Obj isgn, FLA_Obj A, FLA_Obj C, FLA_Obj scale );

Purpose: Solve one of the following triangular Lyapunov equations (lyap):

\[
AX + XA^T = \pm C \\
A^T X + XA = \pm C
\]

where \( A \) is upper triangular matrix and \( C \) is symmetric. If \( A \) and \( C \) are complex matrices, then the possible operations are:

\[
AX + XA^H = \pm C \\
A^H X + XA = \pm C
\]

where \( A \) is upper triangular matrix and \( C \) is Hermitian. The operation references and then overwrites the upper triangle of matrix \( C \) with the upper triangle of the solution matrix \( X \), which is also symmetric (or Hermitian). The \texttt{trans} argument determines whether the equation is solved with \( AX \) (FLA\_NO\_TRANSPOSE) or \( A^H X \) (FLA\_TRANSPOSE or FLA\_CONJ\_TRANSPOSE). The \texttt{isgn} argument is a scalar integer object that indicates whether the \( \pm \) sign is a plus or a minus. The \texttt{scale} argument is used as workspace.

Constraints:

- The numerical datatypes of \( A \) and \( C \) must be identical and floating-point, and must not be FLA\_CONSTANT.
- The \texttt{isgn} argument must be either FLA\_ONE or FLA\_MINUS\_ONE.
- The numerical datatype of \texttt{scale} must not be FLA\_CONSTANT. Furthermore, the precision of the datatype of \texttt{scale} must be equal to that of \( A \) and \( C \).
- The dimensions of \( A \) and \( C \) must be conformal.
- \( A \) and \( C \) must be square.
- \texttt{trans} may not be FLA\_CONJ\_NO\_TRANSPOSE.

Int. Notes: FLA\_Lyap() expects \( A \) and \( C \) to be flat matrix objects.

Imp. Notes: FLA\_Lyap() invokes one or more FLAME/C variants to induce a blocked algorithm with subproblems performed by calling wrappers to external BLAS routines. FLASH\_Lyap() uses multiple FLAME/C algorithmic variants to form an algorithm-by-blocks, which breaks the \texttt{lyap} operation into subproblems expressed in terms of individual blocks of \( A \) and \( C \) and then invokes external BLAS routines to perform the computation on these blocks. The unblocked \texttt{lyap} subproblems are computed by internal implementations. (External LAPACK routines are not used, even when external-lapack-for-subproblems option is enabled.)

Caveats: FLA\_Lyap() and FLASH\_Lyap() are currently only implemented for the case where \texttt{trans} is FLA\_TRANSPOSE (or FLA\_CONJ\_TRANSPOSE).

Arguments:

- \texttt{trans} – Indicates whether the operation proceeds as if the instance of \( A \) in the term \( AX \) were [conjugate] transposed.
- \texttt{isgn} – Indicates whether the Lyapunov equation is solved with \( C \) or \(-C\).
- \texttt{A} – An FLA\_Obj representing matrix \( A \).
- \texttt{C} – An FLA\_Obj representing matrix \( C \).
- \texttt{scale} – A scalar used as workspace.
FLA_Error FLA_Hevd( FLA_Evd_type jobz, FLA_Uplo uplo, FLA_Obj A, FLA_Obj l );

**Purpose:** Perform a Hermitian eigenvalue decomposition (HEVD):

\[ A \rightarrow U \Lambda U^H \]

where \( \Lambda \) is a diagonal matrix whose elements contain the eigenvalues of \( A \), and the columns of \( U \) contain the eigenvectors of \( A \). The \( \text{jobz} \) argument determines whether only eigenvalues (FLA_EVD_WITHOUT_VECTORS) or both eigenvalues and eigenvectors (FLA_EVD_WITH_VECTORS) are computed. The \( \text{uplo} \) argument determines whether \( A \) is stored in the lower or upper triangle. Upon completion, the eigenvalues are stored to the vector \( l \) in ascending order, and the eigenvectors \( U \), if requested, overwrite matrix \( A \) such that vector element \( l_j \) contains the eigenvalue corresponding to the eigenvector stored in the \( j \)th column of \( U \). If eigenvectors are not requested, then the triangle specified by \( \text{uplo} \) is destroyed.

**Returns:** FLA_Hevd() returns the total number of Francis steps performed by the underlying QR algorithm.

**Caveats:** FLA_Hevd() is currently only implemented for the case where \( \text{jobz} \) is FLA_EVD_WITH_VECTORS.

**Constraints:**
- The numerical datatype of \( A \) must be floating-point and must not be FLA_CONSTANT.
- The numerical datatype of \( l \) must be real and must not be FLA_CONSTANT.
- The precision of the datatype of \( l \) must be equal to that of \( A \).
- \( l \) must be a contiguously-stored vector of length \( n \), where \( A \) is \( n \times n \).

**Arguments:**
- \( \text{jobz} \) – Indicates whether only eigenvalues or both eigenvalues and eigenvectors are computed.
- \( \text{uplo} \) – Indicates whether the lower or upper triangle of \( A \) is read during the operation.
- \( A \) – An FLA_Obj representing matrix \( A \).
- \( l \) – An FLA_Obj representing vector \( l \).
FLA_Error FLA_Svd( FLA_Svd_type jobu, FLA_Svd_type jobv, FLA_Obj A, FLA_Obj s,
FLA_Obj U, FLA_Obj V );

Purpose: Perform a singular value decomposition (svd):

$$A \rightarrow U\Sigma V^H$$

where $\Sigma$ is an $m \times n$ diagonal matrix whose elements contain the singular values of $A$, $U$ is an $m \times m$ matrix whose columns contain the left singular vectors of $A$, and $V$ is an $n \times n$ matrix whose rows of $V$ contain the right singular vectors of $A$. The jobu and jobv arguments determine if (and how many of) the left and right singular vectors, respectively, are computed and where they are stored. The jobu and jobv arguments accept the following values:

- **FLA_SVD_VECTORS_ALL.** For jobu: compute all $m$ columns of $U$, storing the result in $U$. For jobv: compute all $n$ columns of $V$, storing the result in $V$.

- **FLA_SVD_VECTORS_MIN_COPY.** For jobu: compute the first $\min(m, n)$ columns of $U$ and store them in $U$. For jobv: compute the first $\min(m, n)$ columns of $V$ and store them in $V$.

- **FLA_SVD_VECTORS_MIN_OVERWRITE.** For jobu: compute the first $\min(m, n)$ columns of $U$ and store them in $A$. For jobv: compute the first $\min(m, n)$ columns of $V$ and store them in $A$. Note that jobu and jobv cannot both be FLA_SVD_VECTORS_MIN_OVERWRITE.

- **FLA_SVD_VECTORS_NONE.** For jobu: no columns of $U$ are computed. For jobv: no columns of $V$ are computed.

Upon completion, the $\min(m, n)$ singular values of $A$ are stored to $s$, sorted in descending order and singular vectors, if computed, are stored to either $A$ or $U$ and $V$, depending on the values of jobu and jobv. If neither jobu nor jobv is FLA_SVD_VECTORS_MIN_OVERWRITE, then $A$ is destroyed.

Returns: FLA_Svd() returns the total number of Francis steps performed by the underlying QR algorithm.

Caveats: FLA_Svd() is currently only implemented for the case where jobu and jobv are both FLA_SVD_VECTORS_ALL.

Constraints:
- The numerical datatypes of $A$, $U$, and $V$ must be identical and floating-point, and must not be FLA_CONSTANT.
- The numerical datatype of $s$ must be real and must not be FLA_CONSTANT.
- The precision of the datatype of $s$ must be equal to that of $A$.
- $e$ must be a contiguously-stored vector of length $\min(m, n)$, where $A$ is $m \times n$.
- $U$ and $V$ must be square.
- The order of $U$ and the order of $V$ must be equal to the the number of rows in $A$ and the number of columns in $A$, respectively.

Arguments:

- **jobu** – Indicates whether the left singular vectors are computed, how many are computed, and where they are stored.
- **jobv** – Indicates whether the right singular vectors are computed, how many are computed, and where they are stored.
- **A** – An FLA_Obj representing matrix $A$.
- **s** – An FLA_Obj representing vector $s$.
- **U** – An FLA_Obj representing matrix $U$.
- **V** – An FLA_Obj representing matrix $V$. 
5.6.3 Utility functions

void FLA_Apply_diag_matrix( FLA_Side side, FLA_Conj conj, FLA_Obj x, FLA_Obj A );

Purpose: Apply a diagonal matrix to a general matrix, where the diagonal is stored in a vector (APDIAGMV):

\[ A := DA \]
\[ A := \bar{DA} \]
\[ A := AD \]
\[ A := A\bar{D} \]

where \( D \) is a diagonal matrix whose diagonal is stored in vector \( x \) and \( A \) is a general matrix. The \texttt{side} argument indicates whether the diagonal matrix \( D \) is multiplied on the left or the right side of \( A \). The \texttt{conj} argument allows the computation to proceed as if \( D \) (i.e., the entries stored in \( x \)) were conjugated.

Constraints:
- The numerical datatypes of \( x \) and \( A \) must be floating-point and must not be \texttt{FLA_CONSTANT}.
- The precision of the datatype of \( x \) must be equal to that of \( A \).
- If \texttt{side} equals \texttt{FLA_LEFT}, then the length of \( x \) and the number of rows in \( A \) must be equal; otherwise, if \texttt{side} equals \texttt{FLA_RIGHT}, then the length of \( x \) must be equal to the number of columns in \( A \).

Arguments:
- \texttt{side} – Indicates whether the operation proceeds as if the diagonal matrix \( D \) is applied from the left or the right.
- \texttt{conj} – Indicates whether the operation proceeds as if the diagonal matrix \( D \) were conjugated.
- \texttt{x} – An FLA_Obj representing vector \( x \).
- \texttt{A} – An FLA_Obj representing matrix \( A \).

void FLA_Shift_pivots_to( FLA_Pivot_type ptype, FLA_Obj p );

Purpose: Convert a pivot vector from \texttt{libflame} pivot indexing to LAPACK indexing, or vice versa. If \( p \) currently contains \texttt{libflame} pivots, setting \texttt{ptype} to \texttt{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 \texttt{ptype} to \texttt{FLA_NATIVE_PIVOTS} will update the contents of \( p \) to reflect the pivoting style used natively within \texttt{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 \texttt{libflame} to LAPACK when the vector already uses LAPACK pivot indexing will result in an undefined format. Please see the description for \texttt{FLA_LU_piv()} in Section 5.6.2 for details on the differences between LAPACK-style pivot vectors and \texttt{libflame} pivot vectors.

Constraints:
- The numerical datatype of \( p \) must be integer, and must not be \texttt{FLA_CONSTANT}.

Arguments:
- \texttt{ptype} – Indicates the desired pivot indexing.
- \texttt{p} – An FLA_Obj 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 5.6.2 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:

```c
FLA_Obj_set_to_identity( A );
FLA_Apply_pivots( FLA_LEFT, FLA_NO_TRANSPOSE, p, A );
```

**Arguments:**
- $A$ — An FLA_Obj representing matrix $A$. 
void FLA_Househ2_UT( FLA_Side side, FLA_Obj chi_1, FLA_Obj x2, FLA_Obj tau );

Purpose: Compute the UT Householder transform, otherwise known as the "UT transform",

\[
H = \left( I - \frac{1}{\tau} \begin{pmatrix} 1 & u_2 \\ u_2 & u_2 \end{pmatrix} \right)^H
\]

by computing \( \tau \) and \( u_2 \) such that one of the following equations is satisfied:

\[
\begin{align*}
H \begin{pmatrix} \chi_1 \\ x_2 \end{pmatrix} &= \begin{pmatrix} \alpha \\ 0 \end{pmatrix} \\
(\chi_1 \ x_2^T)H &= \begin{pmatrix} \alpha \\ 0 \end{pmatrix}
\end{align*}
\]

where

\[
\alpha = - \frac{\|x\|_2 \chi_1}{|\chi_1|} \\
x = \begin{pmatrix} \chi_1 \\ x_2 \end{pmatrix}
\]

The \texttt{side} parameter determines whether the the transform generated by the function annihilates the elements below \( \chi_1 \) in \( x \) (when applied from the left) or the elements to the right of \( \chi_1 \) in \( x^T \) (when applied from the right). On input \( \chi_1 \) and \( x_2 \) are assumed to hold \( \chi_1 \) and \( x_2 \) (or \( x_2^T \)), respectively. Upon completion, \( \chi_1 \), \( x_2 \), and \( \tau \) are overwritten by \( \alpha \), \( u_2 \) (or \( u_2^T \)), and \( \tau \), respectively.

Notes: When \texttt{side} is \texttt{FLA_LEFT}, the function computes \( u_2 \) as

\[
u_2 = \frac{x_2}{\chi_1 - \alpha}
\]

and when \texttt{side} is \texttt{FLA_RIGHT}, the function computes \( u_2 \) as

\[
u_2 = \frac{x_2}{\chi_1 - \alpha}
\]

In either case, \( \tau \) is subsequently computed as

\[
\tau = \frac{1 + u_2^H u_2}{2}
\]

Constraints: • The numerical datatypes of \( \chi_1 \), \( x_2 \), and \( \tau \) must be identical and floating-point, and must not be \texttt{FLA_CONSTANT}.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine \texttt{*nrm2()}.

Arguments:

\begin{itemize}
\item \texttt{chi}_1 \quad \text{An FLA}_\text{Obj} \text{ representing scalar } \chi_1.
\item \texttt{x2} \quad \text{An FLA}_\text{Obj} \text{ representing vector } x_2 \text{ or } x_2^T.
\item \texttt{tau} \quad \text{An FLA}_\text{Obj} \text{ representing scalar } \tau.
\end{itemize}
void FLA_Househ2s_UT ( FLA_Side side, FLA_Obj chi_1, FLA_Obj x2, 
    FLA_Obj alpha, FLA_Obj gamma, FLA_Obj tau );

Purpose: Compute scalars associated with the UT Householder transform, otherwise known as the “UT transform”,

\[ H = \left( I - \frac{1}{\tau} \begin{pmatrix} 1 & 1 \\ u_2 & u_2 \end{pmatrix} \right)^H \]

On input chi_1 and x2 are assumed to hold \( \chi_1 \) and \( x_2 \) (or \( x_2^T \)), respectively. Upon completion, alpha, gamma, and tau are overwritten by \( \alpha, \chi_1 - \alpha, \) and \( \tau \), respectively. Objects chi_1 and x2 are only referenced and not stored.

Notes: The routine does not need a side parameter. The difference in output between the FLA_LEFT and FLA_RIGHT cases comes down to a conjugation of \( u_2 \), and thus the same scalars may be computed regardless of whether the transform is being applied from the left or the right.

More Info: This function is similar to that of FLA_Househ2_UT(). Please see the description for FLA_Househ2_UT() for further details.

Constraints: The numerical datatypes of \( \chi_1, x_2, \) and \( \tau \) must be identical and floating-point, and must not be FLA_CONSTANT.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine *nrm2().

Arguments:

- \( \text{chi}_1 \) – An FLA_Obj representing scalar \( \chi_1 \).
- \( x2 \) – An FLA_Obj representing vector \( x_2 \) or \( x_2^T \).
- \( \text{alpha} \) – An FLA_Obj representing scalar \( \alpha \).
- \( \text{gamma} \) – An FLA_Obj representing scalar \( \chi_1 - \alpha \).
- \( \text{tau} \) – An FLA_Obj representing scalar \( \tau \).
void FLA_Househ3UD_UT( FLA_Obj chi_0, FLA_Obj x1, FLA_Obj y2, FLA_Obj tau );

**Purpose:** Compute the up-and-downdating UT Householder transform, otherwise known as the “up-and-downdating UT transform,”

\[
H = \begin{bmatrix}
1 & 0 & 0 \\
0 & I_{mC} & 0 \\
0 & 0 & I_{mD}
\end{bmatrix} - \frac{1}{\tau} \begin{bmatrix}
1 & 0 & 0 \\
0 & I_{mC} & 0 \\
0 & 0 & -I_{mD}
\end{bmatrix} \begin{bmatrix}
1 \\
u_1 \\
v_2
\end{bmatrix} \begin{bmatrix}
1 \\
u_1 \\
v_2
\end{bmatrix}^T
\]

by computing \( \tau, u_1, \) and \( v_2 \) such that the following equation is satisfied:

\[
H \begin{bmatrix}
\chi_0 \\
x_1 \\
y_2
\end{bmatrix} = \begin{bmatrix}
\alpha \\
0 \\
0
\end{bmatrix}
\]

where

\[
\alpha = -\frac{\lambda \chi_0}{|\chi_0|} \\
\lambda = \sqrt{\bar{\chi}_0 \chi_0 + x_1^H x_1 - y_2^H y_2}.
\]

On input \( \text{chi}_0, x1, \) and \( y2 \) are assumed to hold \( \chi_0, x_1, \) and \( y_2, \) respectively, and upon completion they are overwritten by \( \alpha, u_1, \) and \( v_2, \) respectively.

**Notes:** The function computes \( \tau, u_1, \) and \( v_2 \) as:

\[
\tau = \frac{1 + u_1^H u_1 - v_2^H v_2}{2} \\
u_1 = \frac{x_1}{\chi_0 - \alpha} \\
v_2 = -\frac{y_2}{\chi_0 - \alpha}
\]

**Constraints:** The numerical datatypes of \( \chi_0, x_1, y_2, \) and \( \tau \) must be identical and floating-point, and must not be FLA_CONSTANT.

**Imp. Notes:** This function uses an external implementation of the level-1 BLAS routine \( \ast \text{nrm2}() \).

**Arguments:**
- \( \text{chi}_0 \) – An FLA_Obj representing scalar \( \chi_0 \).
- \( x1 \) – An FLA_Obj representing vector \( x_1 \).
- \( y2 \) – An FLA_Obj representing vector \( y_2 \).
- \( \text{tau} \) – An FLA_Obj representing scalar \( \tau \).
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 transformation $H_j$ 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 or strictly upper triangle of the $m \times n$ matrix $V$ and the $\tau$ scalar factors stored 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:

$$ T = \left( \begin{array}{c|c|c} T_0 & T_1 & \cdots & T_{p-1} \end{array} \right) $$

where $p = [k/b]$. 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 $H(i) = I - \frac{1}{\nu} v v^H$, where $\tau$ 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, $H(0)H(1)\cdots H(k-1)$, and $T$ is upper triangular upon completion. If `direct` is FLA_BACKWARD, then $H$ is the backward product of $k$ Householder reflectors, $H(k-1)\cdots H(1)H(0)$, 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 $H = I - V T^{-1} V^H$, 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 $H = I - V^H T^{-1} V$, 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 $V$.

$$
\begin{array}{rl}
\text{storev} & \\
\text{FLA_COLUMNWISE} & \text{FLA_ROWWISE} \\
\text{direct} & \\
\text{FLA_FORWARD} & \\
\left( \begin{array}{ccc} 1 & & \\ \nu_0 & 1 & \\ \nu_0 & \nu_1 & 1 \\ \nu_0 & \nu_1 & \nu_2 \\ \nu_0 & \nu_1 & \nu_2 \end{array} \right) & \left( \begin{array}{ccccc} 1 & \nu_0 & \nu_0 & \nu_0 & \nu_0 \\ & 1 & \nu_1 & \nu_1 & \nu_1 \\ & & 1 & \nu_2 & \nu_2 \\ & & & 1 & \nu_2 \\ & & & & 1 \end{array} \right) \\
\text{FLA_BACKWARD} & \\
\left( \begin{array}{ccc} \nu_0 & \nu_1 & \nu_2 \\ \nu_0 & \nu_1 & \nu_2 \\ 1 & \nu_1 & \nu_2 \\ 1 & \nu_2 & \nu_2 \end{array} \right) & \left( \begin{array}{ccc} \nu_0 & \nu_0 & 1 \\ \nu_0 & \nu_0 & 1 \\ \nu_1 & \nu_1 & \nu_1 \\ \nu_2 & \nu_2 & \nu_2 \end{array} \right) \\
\text{FLA_CONSTANT} & \\
\end{array}
$$

Here, elements $\nu_j$ 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 operations, such as FLA_QR_UT() and FLA_QR_UT_recover_tmu().

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 $m \times n$.

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 $\tau$ values of $T$ to $t$ (via FLA_QR_UT_recover_tmu()), 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 two cases where `direct` is FLA_FORWARD.

Arguments:
- `direct` - Indicates whether $H$ is formed from the forward or backward product
- `storev` - Indicates whether the vectors stored within $V$ are stored column-wise or row-wise above the diagonal.
- `V` - The matrix $V$.
- `t` - The vector $t$, containing $\tau$.
- `T` - The matrix $T$. 
void FLA_Apply_H2_UT( FLA_Side side, FLA_Obj tau, FLA_Obj u2, FLA_Obj a1, FLA_Obj A2 );

**Purpose:** Apply a single UT Householder transformation, $H$, to a row vector $a_1^T$ and a matrix $A_2$ from the left,

\[
\begin{pmatrix}
  a_1^T \\
  A_2
\end{pmatrix}
:=
H
\begin{pmatrix}
  a_1^T \\
  A_2
\end{pmatrix}
\]

or to a column vector $a_1$ and a matrix $A_2$ from the right,

\[
\begin{pmatrix}
  a_1 \\
  A_2
\end{pmatrix}
:=
\begin{pmatrix}
  a_1 \\
  A_2
\end{pmatrix}H
\]

where $H$ is determined by the scalar $\tau$ and vector $u_2$ computed by FLA_Househ2_UT(). The `side` argument indicates whether the transform is applied from the left or the right. Note that $a_1$ and $A_2$ are typically either vertically (if applying from the left) or horizontally (if applying from the right) adjacent views into the same matrix object, though this is not a requirement.

**Constraints:**
- The numerical datatypes of $\tau$, $u_2$, $a_1$, and $A_2$ must be identical and floating-point, and must not be FLA_CONSTANT.
- If `side` equals FLA_LEFT, then the length of $u_2$ and the number of rows in $A_2$ must be equal; otherwise, if `side` equals FLA_RIGHT, then the length of $u_2$ must be equal to the number of columns in $A_2$.
- If `side` equals FLA_LEFT, then the length of $a_1^T$ and the number of columns in $A_2$ must be equal; otherwise, if `side` equals FLA_RIGHT, then the length of $a_1$ must be equal to the number of rows in $A_2$.

**Arguments:**
- `side` – Indicates whether the Householder transformation is applied from the left or the right.
- `tau` – An FLA_Obj representing scalar $\tau$.
- `u2` – An FLA_Obj representing vector $u_2$.
- `a1` – An FLA_Obj representing vector $a_1$.
- `A2` – An FLA_Obj 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 $b \times k$ matrix $T$ where $b$ is chosen to be a reasonable blocksize and $k = \min(m, n)$. This matrix $T$ is required as input to FLA_QR_UT() so that the upper triangular factors of the 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 and storage format (row- or column-major) 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 FLA_Obj representing matrix $A$.
- $T$ – A pointer to an uninitialized FLA_Obj.
  
  (on entry) – A pointer to an uninitialized FLA_Obj.
  (on exit) – A pointer to a new FLA_Obj parameterized by $b$, $k$, 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 $\tau$ values along the diagonals of the upper triangular factors of the 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 $\tau$ 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 $\dim(t)$.

**Arguments:**
- $T$ – An FLA_Obj representing matrix $T$.
- $t$ – An FLA_Obj representing vector $t$. 
void FLA_QR_UT_form_Q( FLA_Obj A, FLA_Obj T, FLA_Obj Q );

**Purpose:** Form a unitary matrix $Q$ from the Householder vectors stored below the diagonal of $A$ and the block Householder submatrices of $T$:

$$Q := H_0 H_1 \cdots H_{k-1}$$

where $H_i$ is the Householder transform associated with the Householder vector stored below the diagonal in the $i$th column of $A$.

**Imp. Notes:** This operation is implemented such the minimum number of computations are performed in forming $Q$.

**Constraints:**
- The numerical datatypes of $A$, $T$, and $Q$ must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The width of $T$ must be equal to the width of $A$.
- The dimension of $Q$ must be equal to the number of rows in $A$.

**Arguments:**

- $A$ – An `FLA_Obj` representing matrix $A$.
- $T$ – An `FLA_Obj` representing matrix $T$.
- $Q$ – An `FLA_Obj` representing matrix $Q$.

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 a $b \times k$ matrix $T$ where $b$ is chosen to be a reasonable blocksize and $k = \min(m, n)$. This matrix $T$ is required as input to `FLA_LQ_UT()` so that the upper triangular factors of the 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_LQ_UT()`.

**Notes:** Matrix $T$ is created so that its numerical datatype and storage format (row- or column-major) 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 `FLA_Obj` representing matrix $A$.
- $T$ (on entry) – A pointer to an uninitialized `FLA_Obj`.
- $T$ (on exit) – A pointer to a new `FLA_Obj` 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 \( \tau \) values along the diagonals of the upper triangular factors of the 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 \( \tau \) 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 FLA_Obj representing matrix \( T \).
- \( t \) – An FLA_Obj representing vector \( t \).

void FLA_UDdate_UT_create_T( FLA_Obj R, FLA_Obj* T );

Purpose: Given an \( n \times n \) matrix \( R \) that the user intends to up-and-downdate via up-and-downdating UT transforms, 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_UDdate_UT() so that the upper triangular factors of the 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_UDdate_UT().

Notes: Matrix \( T \) is created so that its numerical datatype and storage format (row- or column-major) is the same as that of \( R \).

Constraints:
- The numerical datatype of \( R \) must be floating-point, and must not be FLA_CONSTANT.

Arguments:
- \( R \) – An FLA_Obj representing matrix \( R \).
- \( T \) (on entry) – A pointer to an uninitialized FLA_Obj.
  (on exit) – A pointer to a new FLA_Obj parameterized by \( b \), \( n \), and the datatype of \( A \).
void FLA_LQ_UT_form_Q( FLA_Obj A, FLA_Obj T, FLA_Obj Q );

**Purpose:**  Form a unitary matrix \( Q \) from the Householder vectors stored above the diagonal of \( A \) and the block Householder submatrices of \( T \):

\[
Q := H_{k-1} \cdots H_1 H_0
\]

where \( H_i \) is the Householder transform associated with the Householder vector stored above the diagonal in the \( i \)th row of \( A \).

**Imp. Notes:** This operation is implemented such the minimum number of computations are performed in forming \( Q \).

**Constraints:**

- The numerical datatypes of \( A \), \( T \), and \( Q \) must be identical and floating-point, and must not be \text{FLA\_CONSTANT}.
- The width of \( T \) must be equal to the length of \( A \).
- The dimension of \( Q \) must be equal to the number of columns in \( A \).

**Arguments:**

- \( A \) – An \text{FLA\_Obj} representing matrix \( A \).
- \( T \) – An \text{FLA\_Obj} representing matrix \( T \).
- \( Q \) – An \text{FLA\_Obj} representing matrix \( Q \).

void FLA_Hess_UT_create_T( FLA_Obj A, FLA_Obj* T );

**Purpose:**  Given an \( n \times n \) matrix \( A \) upon which the user intends to perform a reduction to upper Hessenberg form, create a \( b \times n \) matrix \( T \) where \( b \) is chosen to be a reasonable blocksize. This matrix \( T \) is required as input to \text{FLA\_Hess\_UT()} so that the upper triangular factors of the block Householder transformations may be accumulated during each iteration of the factorization algorithm. Once created, \( T \) may be freed normally via \text{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 \text{FLA\_Hess\_UT()}.  

**Notes:** Matrix \( T \) is created so that its numerical datatype and storage format (row- or column-major) is the same as that of \( A \).

**Constraints:**

- The numerical datatype of \( A \) must be floating-point, and must not be \text{FLA\_CONSTANT}.

**Arguments:**

- \( A \) – An \text{FLA\_Obj} representing matrix \( A \).
- \( T \) (on entry) – A pointer to an uninitialized \text{FLA\_Obj}.
- \( T \) (on exit) – A pointer to a new \text{FLA\_Obj} parameterized by \( b \), \( n \), and the datatype of \( A \).
void FLA_Hess_UT_recover_tau( FLA_Obj T, FLA_Obj t );

Purpose: Subsequent to a reduction to upper Hessenberg form via the UT transform, recover the τ values along the diagonals of the upper triangular factors of the 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 dim(t).

Arguments:
T – An FLA_Obj representing matrix T.  
t – An FLA_Obj representing vector t.

void FLA_Tridiag_UT_create_T( FLA_Obj A, FLA_Obj* T );

Purpose: Given an n×n matrix A upon which the user intends to perform a reduction to tridiagonal form via the UT transform, create a b×n matrix T where b is chosen to be a reasonable blocksize. This matrix T is required as input to FLA_Tridiag_UT() so that the upper triangular factors of the block Householder transformations may be accumulated during each iteration of the reduction 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_Tridiag_UT().

Notes: Matrix T is created so that its numerical datatype and storage format (row- or column-major) 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 FLA_Obj representing matrix A.  
T (on entry) – A pointer to an uninitialized FLA_Obj.  
(on exit) – A pointer to a new FLA_Obj parameterized by b, n, and the datatype of A.
void FLA_Tridiag_UT_recover_tau( FLA_Obj T, FLA_Obj t );

**Purpose:** Subsequent to a reduction to tridiagonal form via the UT transform, recover the $\tau$ values along the diagonals of the upper triangular factors of the 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 $\tau$ 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 dim($t$).

**Arguments:**
- $T$ – An FLA_Obj representing matrix $T$.
- $t$ – An FLA_Obj representing vector $t$.

---

void FLA_Tridiag_UT_realify( FLA_Uplo uplo, FLA_Obj A, FLA_Obj r );

**Purpose:** Subsequent to a reduction to tridiagonal form via the UT transform, reduce matrix $A$ to real tridiagonal form and store the scalars used in the reduction in vector $r$. If the matrix datatype is real to begin with, then $A$ is left unchanged and the elements of $r$ are set to one.

**Constraints:**
- The numerical datatypes of $A$ and $r$ must be identical and floating-point, and must not be FLA_CONSTANT.
- $A$ must be square.
- The length and width of $A$ must be equal to dim($r$).

**Arguments:**
- $uplo$ – Indicates whether the lower or upper triangle of $A$ is referenced during the operation.
- $A$ – An FLA_Obj representing matrix $A$.
- $r$ – An FLA_Obj representing vector $r$. 
void FLA_Bidiag_UT_create_T( FLA_Obj A, FLA_Obj* TU, FLA_Obj* TV );

### Purpose:
Given an \( m \times n \) matrix \( A \) upon which the user intends to perform a reduction to bidiagonal form via the UT transform, create \( b \times k \) matrices \( T_U \) and \( T_V \) where \( b \) is chosen to be a reasonable blocksize and \( k = \min(m, n) \). These matrices \( T_U \) and \( T_V \) are required as input to \( \text{FLA_Bidiag_UT}() \) so that the upper triangular factors of the block Householder transformations may be accumulated during each iteration of the reduction algorithm. Once created, \( T_U \) and \( T_V \) may be freed normally via \( \text{FLA_Obj_free}() \). This routine is provided in case the user is not comfortable choosing the length of \( T_U \) and \( T_V \), and thus implicitly setting the algorithmic blocksize of \( \text{FLA_Bidiag_UT}() \).

### Notes:
Matrices \( T_U \) and \( T_V \) are created so that their numerical datatypes and storage formats (row- or column-major) are the same as that of \( A \).

### Constraints:
- The numerical datatype of \( A \) must be floating-point, and must not be \( \text{FLA_CONSTANT} \).

### Arguments:
- **A** – An \( \text{FLA_Obj} \) representing matrix \( A \).
- **TU** – A pointer to an uninitialized \( \text{FLA_Obj} \).
- **TV** – A pointer to an uninitialized \( \text{FLA_Obj} \).

void FLA_Bidiag_UT_recover_tau( FLA_Obj TU, FLA_Obj TV, FLA_Obj tU, FLA_Obj tV );

### Purpose:
Subsequent to a reduction to bidiagonal form via the UT transform, recover the \( \tau \) values along the diagonals of the upper triangular factors of the block Householder submatrices of \( T_U \) and \( T_V \) and store them to vectors \( t_U \) and \( t_V \), respectively.

### Notes:
This routine is rarely needed. However, there may be occasions when the user wishes to save the \( \tau \) values of \( T_U \) and \( T_V \) to \( t_U \) and \( t_V \), discard the matrices \( T_U \) and \( T_V \), and then subsequently rebuild \( T_U \) and \( T_V \) from \( t_U \) and \( t_V \) (via \( \text{FLA_Accum_T_UT}() \)). This routine facilitates the first step of such a process.

### Constraints:
- The numerical datatypes of \( T_U \), \( T_V \), \( t_U \), and \( t_V \) must be identical and floating-point, and must not be \( \text{FLA_CONSTANT} \).
- The width of \( T_U \) and must be equal \( \text{dim}(t_U) \).
- The width of \( T_V \) and must be equal \( \text{dim}(t_V) \).
- \( \text{dim}(t_U) \) must equal \( \text{dim}(t_V) \).

### Arguments:
- **TU** – An \( \text{FLA_Obj} \) representing matrix \( T_U \).
- **TV** – An \( \text{FLA_Obj} \) representing matrix \( T_V \).
- **tU** – An \( \text{FLA_Obj} \) representing vector \( t_U \).
- **tV** – An \( \text{FLA_Obj} \) representing vector \( t_V \).
void FLA_Bidiag_UT_realify( FLA_Obj A, FLA_Obj rL, FLA_Obj rR );

Purpose: Subsequent to a reduction to bidiagonal form via the UT transform, reduce matrix \( A \) to real bidiagonal form and store the left and right scalars used in the reduction in vectors \( r_L \) and \( r_R \), respectively. If the matrix datatype is real to begin with, then \( A \) is left unchanged and the elements of \( r_L \) and \( r_R \) are set to one.

Constraints:
- The numerical datatypes of \( A \), \( r_L \), and \( r_R \) must be identical and floating-point, and must not be FLA_CONSTANT.
- The vector lengths of \( r_L \) and \( r_R \) must be \( \min(m,n) \) where \( A \) is \( m \times n \).

Arguments:
- \( A \) – An FLA_Obj representing matrix \( A \).
- \( rL \) – An FLA_Obj representing vector \( r_L \).
- \( rR \) – An FLA_Obj representing vector \( r_R \).

void FLA_Apply_Q_UT_create_workspace( FLA_Obj T, FLA_Obj B, FLArances Obj* W );
void FLASH_Apply_Q_UT_create_workspace( FLA_Obj T, FLA_Obj B, FLA_Obj* W );

Purpose: Create a flat (or hierarchical) workspace matrix \( W \) needed when applying \( Q \) or \( Q^H \) to \( B \) via FLA_Apply_Q_UT() (or FLASH_Apply_Q_UT()). Once created, \( W \) may be freed normally via FLA_Obj_free() (or FLASH_Obj_free()).

Notes: This function is provided as a convenience to users of FLA_Apply_Q_UT() and FLASH_Apply_Q_UT() so they do not need to worry about creating the workspace matrix object \( W \) with the correct properties.

Constraints:
- The numerical datatypes of \( T \) and \( B \) must be identical and floating-point, and must not be FLA_CONSTANT.
- The pointer argument \( W \) must not be NULL.

Arguments:
- \( T \) – An FLA_Obj representing matrix \( T \).
- \( B \) – An FLA_Obj representing matrix \( B \).
- \( W \) (on entry) – A pointer to an uninitialized FLA_Obj.
- \( W \) (on exit) – A pointer to a new FLA_Obj to represent matrix \( W \).
void FLA_Apply_QUD_UT_create_workspace( FLA_Obj T, FLA_Obj B, FLA_Obj* W );

**Purpose:** Create a flat workspace matrix \( W \) needed when applying \( Q^H \) to \( B \) via FLA_Apply_QUD_UT(). Once created, \( W \) may be freed normally via FLA_Obj_free( ).

**Notes:** This function is provided as a convenience to users of FLA_Apply_QUD_UT( ) so they do not need to worry about creating the workspace matrix object \( W \) with the correct properties.

**Constraints:**
- The numerical datatypes of \( T \) and \( B \) must be identical and floating-point, and must not be FLA_CONSTANT.
- The pointer argument \( W \) must not be NULL.

**Arguments:**
- \( T \) — An FLA_Obj representing matrix \( T \).
- \( B \) — An FLA_Obj representing matrix \( B \).
- \( W \)
  - (on entry) — A pointer to an uninitialized FLA_Obj.
  - (on exit) — A pointer to a new FLA_Obj to represent matrix \( W \).
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 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.

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.

Caveats: Currently, this function only supports hierarchical depths of exactly 1.

Arguments:

- $A_{flat}$ – An FLA_Obj representing matrix $A_{flat}$.  
- $depth$ – The number of levels to create in the matrix hierarchies of $A$, $p$, and $L$.  
- $b_{flash}$ – A pointer to an array of $depth$ values to be used as blocksizes in creating the matrix hierarchies of $A$, $p$, and $L$.  
- $b_{alg}$ – 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 FLASH_LU_incpiv() and FLASH_FS_incpiv(), or zero if the user wishes to use a default value.
- $A$ (on entry) – A pointer to an uninitialized FLA_Obj.  
  (on exit) – A pointer to a new hierarchical FLA_Obj to represent matrix $A$, conformal to and initialized with the contents of $A_{flat}$.  
- $p$ (on entry) – A pointer to an uninitialized FLA_Obj.  
  (on exit) – A pointer to a new hierarchical FLA_Obj to represent vector $p$.  
- $L$ (on entry) – A pointer to an uninitialized FLA_Obj.  
  (on exit) – A pointer to a new hierarchical FLA_Obj to represent matrix $L$.  

void FLASH_QR_UT_create_hier_matrices( FLA_Obj A_flat, dim_t depth, dim_t* b_flash, 
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() and FLASH_Apply_Q_UT(). Unlike with FLASH_QR_UT_inc_create_hier_matrices(), the algorithmic blocksize specified by b_alg must equal the storage blocksize, b_flash. 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() so they do not need to worry about creating the auxiliary $TW$ matrix object with the correct properties.

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.

Caveats: Currently, this function only supports hierarchical depths of exactly 1.

Arguments:

- A_flat – An FLA_Obj representing matrix $A_{flat}$.
- depth – The number of levels to create in the matrix hierarchies of $A$ and $TW$.
- b_flash – A pointer to an array of depth values to be used as blockizes in creating the matrix hierarchies of $A$ and $TW$.

- A (on entry) – A pointer to an uninitialized FLA_Obj.
  (on exit) – A pointer to a new hierarchical FLA_Obj to represent matrix $A$, conformal to and initialized with the contents of $A_{flat}$.

- TW (on entry) – A pointer to an uninitialized FLA_Obj.
  (on exit) – A pointer to a new hierarchical FLA_Obj to represent matrix $TW$. 
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 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.

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.

Caveats: Currently, this function only supports hierarchical depths of exactly 1.

Arguments:

- $A_{flat}$ – An `FLA_Obj` representing matrix $A_{flat}$.
- `depth` – The number of levels to create in the matrix hierarchies of $A$ and $TW$.
- `b_flash` – A pointer to an array of `depth` values to be used as blocksizes in creating the matrix hierarchies of $A$ and $TW$.
- `b_alg` – The value to be used as the length of the storage blocks in $TW$ (i.e., the number of rows in the leaves of $TW$), which determines the algorithmic blocksize used in `FLASH_QR_UT_inc()` and `FLASH_Apply_Q_UT_inc()`, or zero if the user wishes to use a default value.

- $A$ (on entry) – A pointer to an uninitialized `FLA_Obj`.
- (on exit) – A pointer to a new hierarchical `FLA_Obj` to represent matrix $A$, conformal to and initialized with the contents of $A_{flat}$.

- $TW$ (on entry) – A pointer to an uninitialized `FLA_Obj`.
- (on exit) – A pointer to a new hierarchical `FLA_Obj` to represent matrix $TW$. 
void FLASH_LQ_UT_create_hier_matrices( FLA_Obj $A_{\text{flat}}$, dim_t $\text{depth}$, dim_t* $b_{\text{flash}}$, 
FLA_Obj* $A$, FLA_Obj* $TW$ );

Purpose: Create a hierarchical matrix $A$ conformal to a flat matrix $A_{\text{flat}}$ and then copy the contents of $A_{\text{flat}}$ into $A$. The hierarchy of $A$ is specified by the depth and square blocksize values in $\text{depth}$ and $b_{\text{flash}}$, 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_LQ_UT() and FLASH_Apply_Q_UT().

Notes: This function is provided as a convenience to users of FLASH_LQ_UT() so they do not need to worry about creating the auxiliary $TW$ matrix object with the correct properties.

Constraints:
- The numerical datatype of $A_{\text{flat}}$ must be floating-point, and must not be FLA_CONSTANT.
- $A_{\text{flat}}$ must be square.
- The pointer arguments $b_{\text{flash}}$, $A$, and $TW$ must not be NULL.
- Each of the first $\text{depth}$ values in $b_{\text{flash}}$ must be greater than zero.

Caveats: Currently, this function only supports hierarchical depths of exactly 1.

Arguments:

- $A_{\text{flat}}$ – An FLA_Obj representing matrix $A_{\text{flat}}$.
- $\text{depth}$ – The number of levels to create in the matrix hierarchies of $A$ and $TW$.
- $b_{\text{flash}}$ – A pointer to an array of $\text{depth}$ values to be used as blocksizes in creating the matrix hierarchies of $A$ and $TW$.
- $A$ (on entry) – A pointer to an uninitialized FLA_Obj.
- (on exit) – A pointer to a new hierarchical FLA_Obj to represent matrix $A$, conformal to and initialized with the contents of $A_{\text{flat}}$.
- $TW$ (on entry) – A pointer to an uninitialized FLA_Obj.
- (on exit) – A pointer to a new hierarchical FLA_Obj to represent matrix $TW$. 
void FLASH_CAQR_UT_inc_create_hier_matrices( dim_t p, FLA_Obj A_flat, dim_t depth, dim_t* b_flash, dim_t b_alg, FLA_Obj* A, FLA_Obj* ATW, FLA_Obj* R, FLA_Obj* RTW );

Purpose: Create hierarchical matrices \( A \) and \( R \) 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 \( \text{depth} \) and \( \text{b\_flash} \), respectively. Also, create hierarchical matrix objects \( ATW \) and \( RTW \) with proper datatype, dimensions, and hierarchy relative to \( A \) and \( R \) so that the objects may be used together with \( \text{FLASH\_CAQR\_UT\_inc()} \). If \( \text{b\_alg} \) is greater than zero, it is used as the the length of the storage blocks in \( ATW \) and \( RTW \), which determines the algorithmic blocksize used in \( \text{FLASH\_CAQR\_UT\_inc()} \). If \( \text{b\_alg} \) is zero, the length of the storage blocks in \( ATW \) and \( RTW \) are set to a reasonable default value. Once created, \( A \), \( ATW \), \( R \), and \( RTW \) may be freed normally via \( \text{FLASH\_Obj\_free()} \).

Notes: This function is provided as a convenience to users of \( \text{FLASH\_CAQR\_UT\_inc()} \) so they do not need to worry about creating the auxiliary \( ATW \), \( R \), \( RTW \) matrix object with the correct properties.

Constraints:
- The numerical datatype of \( A_{flat} \) must be floating-point, and must not be \( \text{FLA\_CONSTANT} \).
- The pointer arguments \( \text{b\_flash}, \ A, \ ATW, \ R, \ \text{and RTW} \) must not be \( \text{NULL} \).
- Each of the first \( \text{depth} \) values in \( \text{b\_flash} \) must be greater than zero.

Caveats: Currently, this function only supports hierarchical depths of exactly 1.

Arguments:

- \( A_{\text{flat}} \) – An \( \text{FLA\_Obj} \) representing matrix \( A_{\text{flat}} \).
- \( \text{depth} \) – The number of levels to create in the matrix hierarchies of \( A \), \( ATW \), \( R \), and \( RTW \).
- \( \text{b\_flash} \) – A pointer to an array of \( \text{depth} \) values to be used as blocksize in creating the matrix hierarchies of \( A \), \( ATW \), \( R \), and \( RTW \).
- \( \text{b\_alg} \) – The value to be used as the length of the storage blocks in \( ATW \) and \( RTW \) (ie: the number of rows in the leaves of their hierarchies), which determines the algorithmic blocksize used in \( \text{FLASH\_CAQR\_UT\_inc()} \), or zero if the user wishes to use a default value.
- \( A \) (on entry) – A pointer to an uninitialized \( \text{FLA\_Obj} \).
- \( A \) (on exit) – A pointer to a new hierarchical \( \text{FLA\_Obj} \) to represent matrix \( A \), conformal to and initialized with the contents of \( A_{\text{flat}} \).
- \( ATW \) (on entry) – A pointer to an uninitialized \( \text{FLA\_Obj} \).
- \( ATW \) (on exit) – A pointer to a new hierarchical \( \text{FLA\_Obj} \) to represent matrix \( ATW \).
- \( R \) (on entry) – A pointer to an uninitialized \( \text{FLA\_Obj} \).
- \( R \) (on exit) – A pointer to a new hierarchical \( \text{FLA\_Obj} \) to represent matrix \( R \).
- \( RTW \) (on entry) – A pointer to an uninitialized \( \text{FLA\_Obj} \).
- \( RTW \) (on exit) – A pointer to a new hierarchical \( \text{FLA\_Obj} \) to represent matrix \( RTW \).
void FLASH_UDdate_UT_inc_createHierMatrices( FLA_Obj R_flat, FLA_Obj C_flat, FLA_Obj D_flat,
    dim_t depth, dim_t* b_flash, dim_t* b_alg,
    FLA_Obj* R, FLA_Obj* C, FLA_Obj* D,
    FLA_Obj* T, FLA_Obj* W );

Purpose: Create hierarchical matrices $R$, $C$, and $D$ conformal to a flat matrices $R_{flat}$, $C_{flat}$, and $D_{flat}$, respectively, then copy the contents of the former into the latter. The hierarchies of $R$, $C$, and $D$ are specified by the depth and square blocksize values in `depth` and `b_flash`, respectively. Also, create hierarchical matrix objects $T$ and $W$ with proper datatype, dimensions, and hierarchy relative to $R$, $C$, and $D$ so that the objects may be used together with `FLASH_UDdate_UT_inc()` and `FLASH_Apply_QUD_UT_inc()`. If `b_alg` is greater than zero, it is used as the the length of the storage blocks in $T$ and $W$, which determines the algorithmic blocksize used in `FLASH_UDdate_UT_inc()`. If `b_alg` is zero, the length of the storage blocks in $T$ and $W$ are set to a reasonable default value. Once created, $R$, $C$, $D$, $T$ and $W$ may be freed normally via `FLASH_Obj_free()`.

Notes: This function is provided as a convenience to users of `FLASH_UDdate_UT_inc()` so they do not need to worry about creating the auxiliary $T$ and $W$ matrix objects with the correct properties.

Constraints: 
- The numerical datatypes of $R_{flat}$, $C_{flat}$, and $D_{flat}$ must be identical and floating-point, and must not be `FLA_CONSTANT`.
- The pointer arguments `b_flash`, `R`, `C`, `D`, `T`, and `W` must not be `NULL`.
- Each of the first `depth` values in `b_flash` must be greater than zero.

Caveats: Currently, this function only supports hierarchical depths of exactly 1.

Arguments:

- **R_flat** – An `FLA_Obj` representing matrix $R_{flat}$.
- **C_flat** – An `FLA_Obj` representing matrix $C_{flat}$.
- **D_flat** – An `FLA_Obj` representing matrix $D_{flat}$.
- **depth** – The number of levels to create in the matrix hierarchies of $A$ and $TW$.
- **b_flash** – A pointer to an array of `depth` values to be used as blocksize in creating the matrix hierarchies of $A$ and $TW$.
- **b_alg** – 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 `FLASH_QR_UT_inc()` and `FLASH_Apply_Q_UT_inc()`, or zero if the user wishes to use a default value.

- **R** (on entry) – A pointer to an uninitialized `FLA_Obj`.
  (on exit) – A pointer to a new hierarchical `FLA_Obj` to represent matrix $R$, conformal to and initialized with the contents of $R_{flat}$.

- **C** (on entry) – A pointer to an uninitialized `FLA_Obj`.
  (on exit) – A pointer to a new hierarchical `FLA_Obj` to represent matrix $C$, conformal to and initialized with the contents of $C_{flat}$.

- **D** (on entry) – A pointer to an uninitialized `FLA_Obj`.
  (on exit) – A pointer to a new hierarchical `FLA_Obj` to represent matrix $D$, conformal to and initialized with the contents of $D_{flat}$.

- **T** (on entry) – A pointer to an uninitialized `FLA_Obj`.
  (on exit) – A pointer to a new hierarchical `FLA_Obj` to represent matrix $T$.

- **W** (on entry) – A pointer to an uninitialized `FLA_Obj`.
  (on exit) – A pointer to a new hierarchical `FLA_Obj` to represent matrix $W$. 


5.7. 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.

5.7.1 BLAS operations

5.7.1.1 Level-1 BLAS

void FLA_Amax_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 \texttt{amx}.

More Info: This function is similar to that of \texttt{FLA_Amax()}. Please see the description for \texttt{FLA_Amax()} for further details.

void FLA_Asum_external( FLA_Obj x, FLA_Obj norm1 );

Purpose: Compute the 1-norm of a vector:

\[
\| x \|_1 := \sum_{i=0}^{n-1} |x_i|
\]

where \( \| x \|_1 \) is a scalar and \( x_i \) is the \( i \)th element of general vector \( x \) of length \( n \). Upon completion, the 1-norm \( \| x \|_1 \) is stored to \texttt{norm1}.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine \*asum().

More Info: This function is similar to that of \texttt{FLA_Asum()}. Please see the description for \texttt{FLA_Asum()} for further details.

void FLA_Axpy_external( FLA_Obj alpha, FLA_Obj A, FLA_Obj B );

Purpose: Perform an \texttt{AXPY} operation:

\[
B := B + \alpha A
\]

where \( \alpha \) is a scalar, and \( A \) and \( B \) are general matrices.

Notes: If \( A \) and \( B \) are vectors, \texttt{FLA_Axpy_external()} will implicitly and automatically perform the transposition necessary to achieve conformal dimensions.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine \*axpy().

More Info: This function is similar to that of \texttt{FLA_Axpy()}. Please see the description for \texttt{FLA_Axpy()} for further details.
void FLA_Axpyt_external( FLA_Trans trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj B );

Purpose: Perform one of the following extended AXPY operations:

\[
\begin{align*}
B &= B + \alpha A \\
B &= B + \alpha A^T \\
B &= B + \alpha \bar{A} \\
B &= B + \alpha A^H
\end{align*}
\]

where \( \alpha \) is a scalar, and \( A \) and \( B \) are general matrices. The \texttt{trans} argument allows the computation to proceed as if \( A \) were conjugated and/or transposed.

Notes: If \( A \) and \( B \) are vectors, \texttt{FLA_Axpyt_external()} will implicitly and automatically perform the transposition necessary to achieve conformal dimensions regardless of the value of \texttt{trans}.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine \texttt{?axpy()}.

More Info: This function is similar to that of \texttt{FLA_Axpyt()}. Please see the description for \texttt{FLA_Axpyt()} for further details.

---

void FLA_Axpyrt_external( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj alpha, FLA_Obj A, FLA_Obj B );

Purpose: Perform one of the following extended AXPY operations:

\[
\begin{align*}
B &= B + \alpha A \\
B &= B + \alpha A^T \\
B &= B + \alpha \bar{A} \\
B &= B + \alpha A^H
\end{align*}
\]

where \( A \) and \( B \) are triangular (or trapezoidal) matrices. The \texttt{uplo} argument indicates whether the lower or upper triangle of \( B \) is updated by the operation. The \texttt{trans} argument allows the computation to proceed as if \( A \) were conjugated and/or transposed. Note that the \texttt{uplo} and \texttt{trans} arguments together determine which triangle of \( A \) is read and which triangle of \( B \) is updated.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine \texttt{?axpy()}.

More Info: This function is similar to that of \texttt{FLA_Axpyrt()}. Please see the description for \texttt{FLA_Axpyrt()} for further details.
void FLA_Axpys_external( FLA_Obj alpha0, FLA_Obj alpha1, FLA_Obj A,
                     FLA_Obj beta, FLA_Obj B );

Purpose: Perform the following extended AXPY operation:

\[ B := \beta B + \alpha_0 \alpha_1 A \]

where \( \alpha_0, \alpha_1 \) and \( \beta \) are scalars, and \( A \) and \( B \) are general matrices.

Notes: If \( A \) and \( B \) are vectors, FLA_Axpys_external() will implicitly and automatically perform
the transposition necessary to achieve conformal dimensions.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine ?axpy().

More Info: This function 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.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine ?copy().

More Info: This function 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 triangular (or trapezoidal) matrices. The \( \text{uplo} \) 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 ?copy().

More Info: This function is similar to that of FLA_Copyr(). Please see the description for
FLA_Copyr() for further details.
void FLA_Copyrt_external( FLA_Uplo uplo, FLA_Trans trans, FLA_Obj A, FLA_Obj B );

Purpose: Perform an extended copy operation on triangular matrices $A$ and $B$:

\[
\begin{align*}
B & := A \\
B & := A^T \\
B & := \bar{A} \\
B & := A^H
\end{align*}
\]

where $A$ and $B$ are triangular (or trapezoidal) matrices. The \texttt{uplo} argument indicates whether the lower or upper triangle of $B$ is updated by the operation. The \texttt{trans} argument allows the computation to proceed as if $A$ were conjugated and/or transposed. Note that the \texttt{uplo} and \texttt{trans} arguments together determine which triangle of $A$ is read and which triangle of $B$ is overwritten.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine \texttt{?copy()}.

More Info: This function is similar to that of \texttt{FLA_Copyrt()}. Please see the description for \texttt{FLA_Copyrt()} 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:

\[
\begin{align*}
B & := A \\
B & := A^T \\
B & := \bar{A} \\
B & := A^H
\end{align*}
\]

where $A$ and $B$ are general matrices. The \texttt{trans} argument allows the computation to proceed as if $A$ were conjugated and/or transposed.

Notes: If $A$ and $B$ are vectors, \texttt{FLA_Copyt_external()} will implicitly and automatically perform the transposition necessary to achieve conformal dimensions regardless of the value of \texttt{trans}.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine \texttt{?copy()}.

More Info: This function is similar to that of \texttt{FLA_Copyt()}. Please see the description for \texttt{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 \( \rho \) is a scalar, and \( \chi_i \) and \( \psi_i \) are the \( i \)th elements of general vectors \( x \) and \( y \), respectively, where both vectors are of length \( n \). Upon completion, the dot product \( \rho \) is stored to \( \rho \).

Imp. Notes: This function uses external implementations of the level-1 BLAS routines \( \text{?dot()} \) and \( \text{?dotu()} \).

More Info: This function 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:

\[ \rho := \sum_{i=0}^{n-1} \chi_i \psi_i \]

\[ \rho := \sum_{i=0}^{n-1} \bar{\chi}_i \psi_i \]

where \( \rho \) is a scalar, and \( \chi_i \) and \( \psi_i \) are the \( i \)th elements of general vectors \( x \) and \( y \), respectively, where both vectors are of length \( n \). Upon completion, the dot product \( \rho \) is stored to \( \rho \). The \( \text{conj} \) argument allows the computation to proceed as if \( x \) were conjugated.

Notes: If \( x, y, \) and \( \rho \) are real, the value of \( \text{conj} \) is ignored and FLA_Dotc_external() behaves exactly as FLA_Dotc_external().

Imp. Notes: This function uses external implementations of the level-1 BLAS routines \( \text{?dot()} \), \( \text{?dotu()} \), and \( \text{?dotc()} \).

More Info: This function is similar to that of FLA_Dotc(). Please see the description for FLA_Dotc() for further details.
### 5.7. External wrappers

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 \(\alpha, \beta,\) and \(\rho\) are scalars, and \(\chi_i\) and \(\psi_i\) are the \(i\)th elements of general vectors \(x\) and \(y\), respectively, where both vectors are of length \(n\). Upon completion, the dot product \(\rho\) is stored to \(\rho\).

**Imp. Notes:** This function uses external implementations of the level-1 BLAS routines ?dot() and ?dotu().

**More Info:** This function 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:

\[
\rho := \beta \rho + \alpha \sum_{i=0}^{n-1} \chi_i \psi_i
\]

\[
\rho := \beta \bar{\rho} + \alpha \sum_{i=0}^{n-1} \bar{\chi}_i \psi_i
\]

where \(\alpha, \beta,\) and \(\rho\) are scalars, and \(\chi_i\) and \(\psi_i\) are the \(i\)th elements of general vectors \(x\) and \(y\), respectively, where both vectors are of length \(n\). Upon completion, the dot product \(\rho\) is stored to \(\rho\). The \(\text{conj}\) argument allows the computation to proceed as if \(x\) were conjugated.

**Notes:** If \(x, y,\) and \(\rho\) are real, the value of \(\text{conj}\) 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 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 \(\alpha\), \(\beta\), and \(\rho\) are scalars, and \(\chi_i\) and \(\psi_i\) are the \(i\)th elements of general vectors \(x\) and \(y\), respectively, where both vectors are of length \(n\). Upon completion, the dot product \(\rho\) is stored to \(\text{rho}\).

**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_Dot2s_external().

**Imp. Notes:** This function uses external implementations of the level-1 BLAS routines \?dot() and \?dotu().

**More Info:** This function 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:

\[
\rho := \beta \rho + \alpha \sum_{i=0}^{n-1} \chi_i \psi_i + \bar{\alpha} \sum_{i=0}^{n-1} \chi_i \psi_i
\]

\[
\rho := \beta \rho + \alpha \sum_{i=0}^{n-1} \bar{\chi_i} \bar{\psi_i} + \bar{\alpha} \sum_{i=0}^{n-1} \bar{\psi_i} \chi_i
\]

where \(\alpha\), \(\beta\), and \(\rho\) are scalars, and \(\chi_i\) and \(\psi_i\) are the \(i\)th elements of general vectors \(x\) and \(y\), respectively, where both vectors are of length \(n\). Upon completion, the dot product \(\rho\) is stored to \(\text{rho}\). The \(\text{conj}\) argument allows the computation to proceed as if \(x\) were conjugated.

**Notes:** If \(x\), \(y\), and \(\rho\) are real, the value of \(\text{conj}\) 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 is similar to that of FLA_Dot2s(). Please see the description for FLA_Dot2s() for further details.
5.7. External wrappers

void FLA_Inv_scal_external( FLA_Obj alpha, FLA_Obj A );

Purpose: Perform an inverse scaling operation:

\[ A := \alpha^{-1} A \]

where \( \alpha \) 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 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:

\[ A := \alpha^{-1} A \]
\[ A := \bar{\alpha}^{-1} A \]

where \( \alpha \) is a scalar and \( A \) is a general matrix. The \texttt{conjalpha} argument allows the computation to proceed as if \( \alpha \) were conjugated.

Notes: If \( \alpha \) is real, the value of \texttt{conjalpha} is ignored and FLA_Inv_scalc_external() behaves exactly as FLA_Inv_scalc().

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine \(*scal()\).

More Info: This function 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 \( \chi_i \) is the \( i \)th element of general vector \( x \) of length \( n \). Upon completion, the 2-norm \( \|x\|_2 \) is stored to \texttt{norm}.

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine \(*nrm2()\).

More Info: This function 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 \( \alpha \) 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 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:

\[ A := \alpha A \]
\[ A := \overline{\alpha} A \]

where \( \alpha \) is a scalar and \( A \) is a general matrix. The \texttt{conjalpha} argument allows the computation to proceed as if \( \alpha \) were conjugated.

Notes: If \( \alpha \) is real, the value of \texttt{conjalpha} is ignored and FLA_Scalc_external() behaves exactly as FLA_Scalc_external().

Imp. Notes: This function uses an external implementation of the level-1 BLAS routine *scal().

More Info: This function 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 \( \alpha \) is a scalar and \( A \) is a general square matrix. The \texttt{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 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 is similar to that of FLA_Swap(). Please see the description for FLA_Swap() for further details.
5.7. External wrappers

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 ?swap().

More Info: This function is similar to that of FLA\_Swapt(). Please see the description for FLA\_Swapt() for further details.

5.7.1.2 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:

$$
\begin{align*}
  y & := \beta y + \alpha Ax \\
  y & := \beta y + \alpha A^T x \\
  y & := \beta y + \alpha \bar{A} x \\
  y & := \beta y + \alpha A^H x
\end{align*}
$$

where $\alpha$ and $\beta$ 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:

$$
\begin{align*}
  y_r & := \beta y_r + \alpha x_r A^T \\
  y_r & := \beta y_r + \alpha x_r A \\
  y_r & := \beta y_r + \alpha x_r A^H \\
  y_r & := \beta y_r + \alpha x_r \bar{A}
\end{align*}
$$

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 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:

\[
\begin{align*}
  y &= \beta y + \alpha A x & y &= \beta y + \alpha A \bar{x} \\
  y &= \beta y + \alpha A^T x & y &= \beta y + \alpha A^T \bar{x} \\
  y &= \beta y + \alpha \bar{A} x & y &= \beta y + \alpha \bar{A} \bar{x} \\
  y &= \beta y + \alpha A^H x & y &= \beta y + \alpha A^H \bar{x}
\end{align*}
\]

where \(\alpha\) and \(\beta\) are scalars, \(A\) is a general matrix, and \(x\) and \(y\) are general vectors. The \texttt{trans} argument allows the computation to proceed as if \(A\) were conjugated and/or transposed. Likewise, the \texttt{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:

\[
\begin{align*}
  y_r &= \beta y_r + \alpha x_r A^T & y_r &= \beta y_r + \alpha \bar{x}_r A^T \\
  y_r &= \beta y_r + \alpha x_r A & y_r &= \beta y_r + \alpha \bar{x}_r A \\
  y_r &= \beta y_r + \alpha x_r A^H & y_r &= \beta y_r + \alpha \bar{x}_r A^H \\
  y_r &= \beta y_r + \alpha x_r \bar{A} & y_r &= \beta y_r + \alpha \bar{x}_r \bar{A}
\end{align*}
\]

respectively, where \(x_r\) and \(y_r\) are row vectors. If \(A\), \(x\), and \(y\) are real, the value of \texttt{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 \texttt{?gemv}. 

More Info: This function 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:

\[
A := A + \alpha x y^T
\]

where \(\alpha\) 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 \texttt{?ger}. 

More Info: This function 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:

\[ A := A + \alpha xy^T \]
\[ A := A + \alpha xy^T \]
\[ A := A + \alpha \bar{y}^T \]
\[ A := A + \alpha \bar{x}y^T \]

where \( \alpha \) is a scalar, \( A \) is a general matrix, and \( x \) and \( y \) are general vectors. The \texttt{conjx} and \texttt{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 \texttt{conjx} and \texttt{conjy} are ignored and \texttt{FLA_Gerc_external()} behaves exactly as \texttt{FLA_Ger_external()}.

Imp. Notes: This function uses external implementations of the level-3 BLAS routines \texttt{?ger()}, \texttt{?geru()}, and \texttt{?gerc()}.

More Info: This function is similar to that of \texttt{FLA_Gerc()}. Please see the description for \texttt{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 (HEMV) operation:

\[ y := \beta y + \alpha Ax \]

where \( \alpha \) and \( \beta \) are scalars, \( A \) is a Hermitian matrix, and \( x \) and \( y \) are general vectors. The \texttt{uplo} argument indicates whether the lower or upper triangle of \( A \) is referenced by the operation.

Notes: When invoked with real objects, this function performs the SYMV operation.

Imp. Notes: This function uses external implementations of the level-3 BLAS routines \texttt{?hemv()} and \texttt{?symv()}.

More Info: This function is similar to that of \texttt{FLA_Hemv()}. Please see the description for \texttt{FLA_Hemv()} for further details.
void FLA_Hemvc_external( FLA_Uplo uplo, FLA_Comp 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 (HEMV) 
operations:

\[
y := \beta y + \alpha Ax \\
y := \beta y + \alpha A\bar{x}
\]

where \( \alpha \) and \( \beta \) are scalars, \( A \) is a Hermitian matrix, and \( x \) and \( y \) are general vectors. The \( \text{uplo} \) argument indicates whether the lower or upper triangle of \( A \) is referenced 
by the operation. The \( \text{conj} \) argument allows the computation to proceed as if \( A \) were 
conjugated.

Notes: When invoked with real objects, this function performs the SYMV operation.

Imp. Notes: This function uses external implementations of the level-3 BLAS routines \(?\text{hemv}()\) and 
\(?\text{symv}()\).

More Info: This function is similar to that of \text{FLA_Hemvc()}. Please see the description for 
\text{FLA_Hemvc()} for further details.


void FLA_Her_external( FLA_Uplo uplo, FLA_Obj alpha, FLA_Obj x, FLA_Obj A );

Purpose: Perform a Hermitian rank-1 update (HER) operation:

\[
A := A + \alpha xx^H
\]

where \( \alpha \) is a scalar, \( A \) is a Hermitian matrix, and \( x \) is a general vector. The \( \text{uplo} \) 
argument indicates whether the lower or upper triangle of \( A \) is referenced and updated 
by the operation.

Notes: When invoked with real objects, this function performs the HER operation.

Imp. Notes: This function uses external implementations of the level-3 BLAS routines \(?\text{her}()\) and 
\(?\text{syr}()\).

More Info: This function is similar to that of \text{FLA_Her()}. Please see the description for \text{FLA_Her()} 
for further details.
void FLA_Herc_external( FLA_Uplo uplo, FLA_Conj conj, FLA_Obj alpha, FLA_Obj x, FLA_Obj A );

**Purpose:** Perform one of the following extended Hermitian rank-1 update (HER) operations:

\[
\begin{align*}
A & := A + \alpha xx^H \\
A & := A + \alpha \bar{x}x^T
\end{align*}
\]

where \( \alpha \) is a scalar, \( A \) is a Hermitian matrix, and \( x \) is a general vector. The \texttt{uplo} argument indicates whether the lower or upper triangle of \( A \) is referenced and updated by the operation. The \texttt{conj} argument allows the computation of the transposed rank-1 product \( \bar{x}x^T \).

**Notes:** When invoked with real objects, this function performs the \texttt{her} operation.

**Imp. Notes:** This function uses external implementations of the level-3 BLAS routines \texttt{her()} and \texttt{syr()}. 

**More Info:** This function is similar to that of \texttt{FLA_Herc()}. Please see the description for \texttt{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 Hermitian rank-2 update (HER2) operation:

\[
A := A + \alpha xy^H + \bar{\alpha}yx^H
\]

where \( \alpha \) is a scalar, \( A \) is a Hermitian matrix, and \( x \) and \( y \) are general vectors. The \texttt{uplo} argument indicates whether the lower or upper triangle of \( A \) is referenced and updated by the operation.

**Notes:** When invoked with real objects, this function performs the \texttt{her2} operation.

**Imp. Notes:** This function uses external implementations of the level-3 BLAS routines \texttt{her2()} and \texttt{syr2()}. 

**More Info:** This function is similar to that of \texttt{FLA_Her2()}. Please see the description for \texttt{FLA_Her2()} for further details.
void FLA_Her2c_external( FLA_Uplo uplo, FLA_Conj conj, FLA_Obj alpha,  
   FLA_Obj x, FLA_Obj y, FLA_Obj A );

**Purpose:** Perform one of the following extended Hermitian rank-2 update (\texttt{her2}) operations:

\[
A := A + \alpha xy^H + \bar{\alpha} yx^H \\
A := A + \alpha \bar{x}y^T + \bar{\alpha} x\bar{y}^T
\]

where \(\alpha\) is a scalar, \(A\) is a Hermitian matrix, and \(x\) and \(y\) are general vectors. The \texttt{uplo} argument indicates whether the lower or upper triangle of \(A\) is referenced and updated by the operation. The \texttt{conj} argument allows the computation of the transposed rank-2 products \(\bar{x} y^T\) and \(\bar{y} x^T\).

**Notes:** When invoked with real objects, this function performs the \texttt{her2} operation.

**Imp. Notes:** This function uses external implementations of the level-3 BLAS routines \texttt{?her2()} and \texttt{?syr2()}.

**More Info:** This function is similar to that of \texttt{FLA_Her2c()}. Please see the description for \texttt{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 (\texttt{symv}) operation:

\[
y := \beta y + \alpha Ax
\]

where \(\alpha\) and \(\beta\) are scalars, \(A\) is a symmetric matrix, and \(x\) and \(y\) are general vectors. The \texttt{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 \texttt{?symv()}.

**More Info:** This function is similar to that of \texttt{FLA_Symv()}. Please see the description for \texttt{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 (\texttt{syr}) operation:

\[
A := A + \alpha xx^T
\]

where \(\alpha\) is a scalar, \(A\) is a symmetric matrix, and \(x\) is a general vector. The \texttt{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 \texttt{?syr()}.

**More Info:** This function is similar to that of \texttt{FLA_Syr()}. Please see the description for \texttt{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 (SYR2) operation:

\[ A := A + \alpha xy^T + \alpha yx^T \]

where \( \alpha \) is a scalar, \( A \) is a symmetric matrix, and \( x \) and \( y \) are general vectors. The \( \text{uplo} \) 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 \( \text{?syr2()} \).

More Info: This function is similar to that of \( \text{FLA_Syr2()} \). Please see the description for \( \text{FLA_Syr2()} \) 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 (TRMV) operations:

\[
\begin{align*}
x & := Ax \\
x & := A^T x \\
x & := \tilde{A} x \\
x & := A^H x
\end{align*}
\]

where \( A \) is a triangular matrix and \( x \) is a general vector. The \( \text{uplo} \) argument indicates whether the lower or upper triangle of \( A \) is referenced by the operation. The \( \text{transa} \) argument allows the computation to proceed as if \( A \) were conjugated and/or transposed. The \( \text{diag} \) 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 \( \text{?trmv()} \).

More Info: This function is similar to that of \( \text{FLA_Trmv()} \). Please see the description for \( \text{FLA_Trmv()} \) 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 (TRMV) operations:

\[
\begin{align*}
  y & := \beta y + \alpha Ax \\
  y & := \beta y + \alpha A^T x \\
  y & := \beta y + \alpha \bar{A} x \\
  y & := \beta y + \alpha A^H x
\end{align*}
\]

where \( \alpha \) and \( \beta \) are scalars, \( A \) is a triangular matrix, and \( x \) and \( y \) are general vectors. The \( \text{uplo} \) argument indicates whether the lower or upper triangle of \( A \) is referenced by the operation. The \( \text{transa} \) argument allows the computation to proceed as if \( A \) were conjugated and/or transposed. The \( \text{diag} \) 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 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 solve (TRSV) operations:

\[
\begin{align*}
  Ax &= b \\
  A^T x &= b \\
  \bar{A} x &= b \\
  A^H x &= b
\end{align*}
\]

which, respectively, are solved by overwriting \( b \) with the contents of the solution vector \( x \) as follows:

\[
\begin{align*}
  b & := A^{-1} b \\
  b & := A^{-T} b \\
  b & := \bar{A}^{-1} b \\
  b & := A^{-H} b
\end{align*}
\]

where \( A \) is a triangular matrix and \( x \) and \( b \) are general vectors. The \( \text{uplo} \) argument indicates whether the lower or upper triangle of \( A \) is referenced by the operation. The \( \text{transa} \) argument allows the computation to proceed as if \( A \) were conjugated and/or transposed. The \( \text{diag} \) 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 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 solve (TRSV) operations:

\[
\begin{align*}
  y &= \beta y + \alpha A^{-1} b \\
  y &= \beta y + \alpha A^{-T} b \\
  y &= \beta y + \alpha \bar{A}^{-1} b \\
  y &= \beta y + \alpha A^{-H} b
\end{align*}
\]

where \( \alpha \) and \( \beta \) 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 is similar to that of FLA_Trsvsx(). Please see the description for FLA_Trsvsx() for further details.

5.7.1.3 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:

\[
\begin{align*}
  C &= \beta C + \alpha AB \\
  C &= \beta C + \alpha A^T B \\
  C &= \beta C + \alpha A B \\
  C &= \beta C + \alpha A^H B \\
  C &= \beta C + \alpha A^T B^T \\
  C &= \beta C + \alpha A B^T \\
  C &= \beta C + \alpha A^H B^T \\
  C &= \beta C + \alpha A^H B
\end{align*}
\]

where \( \alpha \) and \( \beta \) 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 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:

\[
C := \beta C + \alpha AB \\
C := \beta C + \alpha BA
\]

where \( \alpha \) and \( \beta \) are scalars, \( A \) is a Hermitian matrix, and \( B \) and \( C \) are general matrices. The \texttt{side} argument indicates whether matrix \( A \) is multiplied on the left or the right side of \( B \). The \texttt{uplo} argument indicates whether the lower or upper triangle of \( A \) is referenced by the operation.

Imp. Notes: This function uses external implementations of the level-3 BLAS routines \texttt{?hemm()} and \texttt{?symm()}. More Info: This function is similar to that of \texttt{FLA\_Hemm()}. Please see the description for \texttt{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:

\[
C := \beta C + \alpha AA^H \\
C := \beta C + \alpha A^H A
\]

where \( \alpha \) and \( \beta \) are scalars, \( C \) is a Hermitian matrix, and \( A \) is a general matrix. The \texttt{uplo} argument indicates whether the lower or upper triangle of \( C \) is referenced and updated by the operation. The \texttt{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 external implementations of the level-3 BLAS routines \texttt{?herk()} and \texttt{?syrk()}. More Info: This function is similar to that of \texttt{FLA\_Herk()}. Please see the description for \texttt{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:

\[
\begin{align*}
C &= \beta C + \alpha AB^H + \bar{\alpha} BA^H \\
C &= \beta C + \alpha A^H B + \bar{\alpha} B^H A
\end{align*}
\]

where \( \alpha \) and \( \beta \) are scalars, \( C \) is a Hermitian matrix, and \( A \) and \( B \) are general matrices. The \text{uplo} argument indicates whether the lower or upper triangle of \( C \) is referenced and updated by the operation. The \text{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 external implementations of the level-3 BLAS routines \?her2k() and ?syr2k().

More Info: This function 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:

\[
\begin{align*}
C &= \beta C + \alpha AB \\
C &= \beta C + \alpha BA
\end{align*}
\]

where \( \alpha \) and \( \beta \) are scalars, \( A \) is a symmetric matrix, and \( B \) and \( C \) are general matrices. The \text{side} argument indicates whether the symmetric matrix \( A \) is multiplied on the left or the right side of \( B \). The \text{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 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:

\[
\begin{align*}
C &= \beta C + \alpha A A^T \\
C &= \beta C + \alpha A^T A
\end{align*}
\]

where \( \alpha \) and \( \beta \) are scalars, \( C \) is a symmetric matrix, and \( A \) is a general matrix. The \text{uplo} argument indicates whether the lower or upper triangle of \( C \) is referenced and updated by the operation. The \text{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 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:

\[
C := \beta C + \alpha AB' + \alpha BA' \\
C := \beta C + \alpha A'B + \alpha B'A
\]

where \( \alpha \) and \( \beta \) are scalars, \( C \) is a symmetric matrix, and \( A \) and \( B \) are general matrices. The \texttt{uplo} argument indicates whether the lower or upper triangle of \( C \) is referenced and updated by the operation. The \texttt{trans} argument allows the computation to proceed as if \( A \) and \( B \) were transposed, which results in the alternate rank-2k products \( A'B \) and \( B'A \).

**Imp. Notes:** This function uses an external implementation of the level-3 BLAS routine \texttt{?syr2k}().

**More Info:** This function 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:

\[
B := \alpha AB \quad B := \alpha BA \\
B := \alpha A'B \quad B := \alpha BA' \\
B := \alpha A'H \quad B := \alpha BA'H
\]

where \( \alpha \) is a scalar, \( A \) is a triangular matrix, and \( B \) is a general matrix. The \texttt{side} argument indicates whether the triangular matrix \( A \) is multiplied on the left or the right side of \( B \). The \texttt{uplo} argument indicates whether the lower or upper triangle of \( A \) is referenced by the operation. The \texttt{trans} argument may be used to perform the check as if \( A \) were conjugated and/or transposed. The \texttt{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 \texttt{?trmm}().

**More Info:** This function is similar to that of FLA_Trmm(). Please see the description for FLA_Trmm() for further details.
void FLA_Trmmsx_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:

\[
\begin{align*}
C &= \beta C + \alpha AB \\
C &= \beta C + \alpha A^T B \\
C &= \beta C + \alpha \bar{A} B \\
C &= \beta C + \alpha A^H B
\end{align*}
\]

where \(\alpha\) and \(\beta\) 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 \(\text{FLA_CONSTANT}\).
- If \(\alpha\) and \(\beta\) are not of datatype \(\text{FLA_CONSTANT}\), then they must match the datatypes of \(A\), \(B\), and \(C\).
- If `side` equals \(\text{FLA_LEFT}\), then the number of rows in \(B\) and the order of \(A\) must be equal; otherwise, if `side` equals \(\text{FLA_RIGHT}\), 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 \(\text{FLA_ZERO_DIAG}\).

**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 FLA_Obj representing scalar \(\alpha\).
- `A` - An FLA_Obj representing matrix \(A\).
- `B` - An FLA_Obj representing matrix \(B\).
- `beta` - An FLA_Obj representing scalar \(\beta\).
- `C` - An FLA_Obj 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 solve with multiple right-hand sides (TRSM) operations:

\[
\begin{align*}
AX &= \alpha B \\
A^T X &= \alpha B \\
\bar{A}X &= \alpha B \\
A^H X &= \alpha B \\
\end{align*}
\]

and overwrite \( B \) with the contents of the solution matrix \( X \) as follows:

\[
\begin{align*}
B &:= \alpha A^{-1}B & B &:= \alpha BA^{-1} \\
B &:= \alpha A^{-T}B & B &:= \alpha BA^{-T} \\
B &:= \alpha \bar{A}^{-1}B & B &:= \alpha B\bar{A}^{-1} \\
B &:= \alpha A^{-H}B & B &:= \alpha BA^{-H} \\
\end{align*}
\]

where \( \alpha \) is a scalar, \( A \) is a triangular matrix, and \( X \) and \( B \) are general matrices. The \texttt{side} argument indicates whether the triangular matrix \( A \) is multiplied on the left or the right side of \( X \). The \texttt{uplo} argument indicates whether the lower or upper triangle of \( A \) is referenced by the operation. The \texttt{trans} argument allows the computation to proceed as if \( A \) were conjugated and/or transposed. The \texttt{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 \texttt{?trsm}().

**More Info:** This function is similar to that of \texttt{FLA_Trsm()} Please see the description for \texttt{FLA_Trsm()} for further details.
void FLA_Trsmsx_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 solve with multiple right-hand sides (TRSM) operations:

\[ AX = \alpha B \quad XA = \alpha B \]
\[ A^T X = \alpha B \quad XA^T = \alpha B \]
\[ \tilde{A}X = \alpha B \quad \tilde{X}A = \alpha B \]
\[ A^H X = \alpha B \quad XA^H = \alpha B \]

and update \( C \) with the contents of the solution matrix \( X \) as follows:

\[ C := \beta C + \alpha A^{-1}B \quad C := \beta C + \alpha BA^{-1} \]
\[ C := \beta C + \alpha A^{-T}B \quad C := \beta C + \alpha BA^{-T} \]
\[ C := \beta C + \alpha \tilde{A}^{-1}B \quad C := \beta C + \alpha B\tilde{A}^{-1} \]
\[ C := \beta C + \alpha A^{-H}B \quad C := \beta C + \alpha BA^{-H} \]

where \( \alpha \) and \( \beta \) are scalars, \( A \) is a triangular matrix, and \( X, B, \) and \( C \) are general matrices. The \textit{side} argument indicates whether the triangular matrix \( A \) is multiplied on the left or the right side of \( X \). The \textit{uplo} argument indicates whether the lower or upper triangle of \( A \) is referenced by the operation. The \textit{trans} argument allows the computation to proceed as if \( A \) were conjugated and/or transposed. The \textit{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 \texttt{FLA_CONSTANT}.
- If \( \alpha \) and \( \beta \) are not of datatype \texttt{FLA_CONSTANT}, then they must match the datatypes of \( A, B, \) and \( C \).
- If \textit{side} equals \texttt{FLA_LEFT}, then the number of rows in \( B \) and the order of \( A \) must be equal; otherwise, if \textit{side} equals \texttt{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.
- \textit{diag} may not be \texttt{FLA_ZERO_DIAG}.

Imp. Notes: This function uses an external implementation of the level-3 BLAS routine \texttt{?trsm}.

Arguments:
- \textit{side} – Indicates whether \( A \) is multiplied on the left or right side of \( X \).
- \textit{uplo} – Indicates whether the lower or upper triangle of \( A \) is referenced during the operation.
- \textit{trans} – Indicates whether the operation proceeds as if \( A \) were conjugated and/or transposed.
- \textit{diag} – Indicates whether the diagonal of \( A \) is unit or non-unit.
- \textit{alpha} – An \texttt{FLA_Obj} representing scalar \( \alpha \).
- \textit{A} – An \texttt{FLA_Obj} representing matrix \( A \).
- \textit{B} – An \texttt{FLA_Obj} representing matrix \( B \).
- \textit{beta} – An \texttt{FLA_Obj} representing scalar \( \beta \).
- \textit{C} – An \texttt{FLA_Obj} representing matrix \( C \).
5.7.2 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):

\[
\begin{align*}
A & \rightarrow LL^T \\
A & \rightarrow U^TU \\
A & \rightarrow LL^H \\
A & \rightarrow U^HU
\end{align*}
\]

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 \( \text{uplo} \).

Imp. Notes: \( \text{FLA\_Chol\_blk\_external()} \) and \( \text{FLA\_Chol\_unb\_external()} \) perform their computation by calling external implementations of the LAPACK routines \(?\text{potrf}()\) and \(?\text{potf2}()\), respectively. The algorithmic variants employed by these routines, as well as the blocksize used by \(?\text{potrf}()\), are implementation-dependent.

Caveats: \( \text{FLA\_Chol\_blk\_external()} \) and \( \text{FLA\_Chol\_unb\_external()} \) are available only if external LAPACK interfaces were enabled at configure-time.

More Info: This function is similar to that of \( \text{FLA\_Chol()} \). Please see the description for \( \text{FLA\_Chol()} \) 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 \( \text{uplo} \). The \( \text{diag} \) argument indicates whether the diagonal of \( A \) is unit or non-unit.

Imp. Notes: \( \text{FLA\_Trinv\_blk\_external()} \) and \( \text{FLA\_Trinv\_unb\_external()} \) perform their computation by calling external implementations of the LAPACK routines \(?\text{trtri}()\) and \(?\text{trti2}()\), respectively. The algorithmic variants employed by these routines, as well as the blocksize used by \(?\text{trtri}()\), are implementation-dependent.

Caveats: \( \text{FLA\_Trinv\_blk\_external()} \) and \( \text{FLA\_Trinv\_unb\_external()} \) are available only if external LAPACK interfaces were enabled at configure-time.

More Info: This function is similar to that of \( \text{FLA\_Trinv()} \). Please see the description for \( \text{FLA\_Trinv()} \) 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):

\[
A :\begin{align*}
&= L^T L \\
&= UU^T \\
&= L^H L \\
&= UU^H
\end{align*}
\]

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 \( \text{uplo} \).

**Imp. Notes:** FLA_Ttmm_blk_external() and FLA_Ttmm_unb_external() perform their computation by calling external implementations of the LAPACK routines ?laum() and ?lauu2(), respectively. The algorithmic variants employed by these routines, as well as the blocksize used by ?laum(), are implementation-dependent.

**Caveats:** FLA_Ttmm_blk_external() and FLA_Ttmm_unb_external() are available only if external LAPACK interfaces were enabled at configure-time.

**More Info:** This function 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 \( \text{uplo} \).

**Imp. Notes:** FLA_SPDinv_blk_external() performs its computation by calling external implementations of the LAPACK routines ?potrf(), ?trtri(), and ?laum(). The algorithmic variants and blocksizes used by these routines are implementation-dependent.

**Caveats:** FLA_SPDinv_blk_external() is available only if external LAPACK interfaces were enabled at configure-time.

**More Info:** This function is similar to that of FLA_SPDinv(). Please see the description for FLA_SPDinv() 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.

**Caveats:** FLA_LU_piv_blk_external() and FLA_LU_piv_unb_external() are available only if external LAPACK interfaces were enabled at configure-time.

**More Info:** This function is similar to that of FLA_LU_nopiv(). Please see the description for FLA_LU_nopiv() for further details.
5.7. External wrappers

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:

\[ Q = H(0)H(1)\cdots H(k-1) \]

Each \( H(i) \) has the form

\[ H(i) = I - \tau vv^T \]

where \( \tau \) 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 the upper triangle (or upper trapezoid) of \( A \) with \( R \). However, the matrix \( Q \) is not stored explicitly. Instead, the operation stores the \( \tau \) 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 \( A \).

Caveats: FLA_QR_blk_external() and FLA_QR_unb_external() are available only if external LA-PACK interfaces were enabled at configure-time.

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:

- \( A \) – An FLA_Obj representing matrix \( A \).
- \( t \) – An FLA_Obj 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:

\[ Q = H(k-1) \cdots H(1)H(0) \]

Each \( H(i) \) has the form

\[ H(i) = I - \tau vv^T \]

where \( \tau \) is a scalar and \( v \) is a vector of length \( n \). 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 lower triangle (or lower trapezoid) of \( A \) with \( L \). However, the matrix \( Q \) is not stored explicitly. Instead, the operation stores the \( \tau \) associated with \( H(i) \) to the \( i \)th element of vector \( t \), and also stores the non-unit, non-zero entries \( v_{i+1:n-1} \) of Householder reflectors \( H_0 \) through \( H_k \) row-wise above the diagonal of \( A \). More specifically, entries \( v_{i+1:n-1} \) are stored to elements \( i+1 : n-1 \) of the \( i \)th row of matrix \( A \).

Caveats: FLA_LQ_blk_external() and FLA_LQ_unb_external() are available only if external LA-PACK interfaces were enabled at configure-time.

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:

\( A \) – An FLA_Obj representing matrix \( A \).
\( t \) – An FLA_Obj representing vector \( t \).
5.7. External wrappers

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 (Hess) via Householder transformations:

\[ A \rightarrow QRQ^H \]

where \( Q \) is an orthogonal matrix (or, a unitary matrix if \( A \) is complex) and \( R \) is an upper Hessenberg matrix (zeroses below the first subdiagonal). Matrix \( Q \) is expressed as a product of \((i_{hi} - i_{lo})\) Householder reflectors:

\[ Q = H(i_{lo})H(i_{lo} + 1)\cdots H(i_{hi} - 1) \]

Each \( H(i) \) has the form

\[ H(i) = I - \tau vv^H \]

where \( \tau \) is a real scalar and \( v \) is a real vector of length \( n \). If \( \nu_j \) 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 \( \nu_{i+1:n-1} \) 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 \( \tau \) 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 ?gedh2(), respectively. The algorithmic variants employed by these routines, as well as the blocksize used by ?gehrd(), are implementation-dependent.

More Info: This function is similar to that of FLA_Hess_UT(). Please see the description for FLA_Hess_UT() for further details.

void FLA_Tridiag_blk_external( FLA_Uplo uplo, FLA_Obj A, FLA_Obj t );
void FLA_Tridiag_unb_external( FLA_Uplo uplo, FLA_Obj A, FLA_Obj t );

Purpose: Perform a reduction to tridiagonal form (tridiag) via Householder transformations:

\[ A \rightarrow QRQ^H \]

where \( Q \) is an orthogonal matrix (or, a unitary matrix if \( A \) is complex) and \( R \) is a tridiagonal matrix (zeros below the first subdiagonal and above the first superdiagonal).

Imp. Notes: FLA_Tridiag_blk_external() and FLA_Tridiag_unb_external() perform their computation by calling external implementations of the LAPACK routines ?sytrd()/?hetrd() and ?sytd2()/?hetd2(), respectively. The algorithmic variants employed by these routines, as well as the blocksize used by ?sytrd() and ?hetrd(), are implementation-dependent.

More Info: This function is similar to that of FLA_Tridiag_UT(). Please see the description for FLA_Tridiag_UT() for further details.
void FLA_Apply_Q_blk_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:

$$
\begin{align*}
B & := QB \\
B & := QTB \\
B & := QTB \\
B & := QHB \\
B & := BQH
\end{align*}
$$

where $Q$ is the orthogonal (or, if $A$ is complex, unitary) matrix implicitly defined by the Householder vectors stored in matrix $A$ and the $\tau$ values stored in vector $t$. The $\text{side}$ argument indicates whether $Q$ is applied to $B$ from the left or the right. The $\text{trans}$ argument indicates whether $Q$ or $Q^T$ (or $Q^H$) is applied to $B$. The $\text{storev}$ 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: $\text{FLA\_Apply\_Q\_blk\_external()}$ performs its computation by calling an external implementation of the LAPACK routines $\text{ormqr()}$/$\text{unmqr()}$/$\text{ormlq()}$/$\text{unmlq()}$. The algorithmic variants employed by these routines, as well as the blockizes used by $\text{ormqr()}$, $\text{unmqr()}$, $\text{ormlq()}$, and $\text{unmlq()}$ are implementation-dependent.

Caveats: $\text{FLA\_Apply\_Q\_blk\_external()}$ is available only if external LAPACK interfaces were enabled at configure-time.

Constraints:

- The numerical datatypes of $A$, $t$, and $B$ must be identical and floating-point, and must not be $\text{FLA\_CONSTANT}$.
- If $\text{side}$ equals $\text{FLA\_LEFT}$, then the number of rows in $B$ and the order of $A$ must be equal; otherwise, if $\text{side}$ equals $\text{FLA\_RIGHT}$, then the number of columns in $B$ and the order of $A$ must be equal.
- If $A$ is real, then $\text{trans}$ must be $\text{FLA\_NO\_TRANSPOSE}$ or $\text{FLA\_TRANSPOSE}$; otherwise if $A$ is complex, then $\text{trans}$ must be $\text{FLA\_NO\_TRANSPOSE}$ or $\text{FLA\_CONJ\_TRANSPOSE}$.
- The length of $t$ must be $\min(m, n)$ where $A$ is $m \times n$.

Arguments:

- $\text{side}$ – Indicates whether $Q$ (or $Q^T$ or $Q^H$) is multiplied on the left or right side of $B$.
- $\text{trans}$ – Indicates whether the operation proceeds as if $Q$ were transposed (or conjugate-transposed).
- $\text{storev}$ – Indicates whether the vectors stored within $A$ are stored column-wise or row-wise.
- $A$ – An $\text{FLA\_Obj}$ representing matrix $A$.
- $t$ – An $\text{FLA\_Obj}$ representing vector $t$.
- $B$ – An $\text{FLA\_Obj}$ representing matrix $B$. 
5.7. External wrappers

void FLA_Sylv_blk_external( FLA_Trans transa, FLA_Trans transb, FLA_Obj isgn,
                          FLA_Obj A, FLA_Obj B, FLA_Obj C, FLA_Obj scale );

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):

\[
\begin{align*}
AX \pm XB &= C \\
AX \pm XB^T &= C \\
A^TX \pm XB &= C \\
A^TX \pm XB^T &= C
\end{align*}
\]

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:

\[
\begin{align*}
AX \pm XB &= C \\
AX \pm XB^H &= C \\
A^H X \pm XB &= C \\
A^H X \pm XB^H &= C
\end{align*}
\]

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 \texttt{isgn} argument is a scalar integer object that indicates whether the \( \pm \) sign between terms is a plus or a minus. The \texttt{scale} argument is not referenced and set to 1.0 upon completion.

Imp. Notes: \texttt{FLA_Sylv_blk_external()} and \texttt{FLA_Sylv_unb_external()} perform their computation by calling an external implementation of the LAPACK routine \texttt{?trsyl()}. The algorithmic variant employed by this routine is implementation-dependent.

Imp. Notes: \texttt{FLA_Sylv_blk_external()} is simply a wrapper to \texttt{FLA_Sylv_unb_external()}.

Caveats: \texttt{FLA_Sylv_blk_external()} and \texttt{FLA_Sylv_unb_external()} are available only if external LAPACK interfaces were enabled at configure-time.

More Info: This function is similar to that of \texttt{FLA_Sylv()}. Please see the description for \texttt{FLA_Sylv()} for further details.
void FLA_Eig_gest_blk_external( FLA_Inv inv, FLA_Uplo uplo, FLA_Obj A, FLA_Obj B );
void FLA_Eig_gest_unb_external( FLA_Inv inv, FLA_Uplo uplo, FLA_Obj A, FLA_Obj B );

Purpose: Perform one of the following operations to reduce a symmetric- or Hermitian-definite eigenproblem to standard form (EIGGEST):

\[ \begin{align*}
A & := L^H AL \\
A & := UA^H U \\
A & := LAL^{-H} \\
A & := U^{-H} AU
\end{align*} \]

where \( A \), on input and output, is symmetric (or Hermitian) and \( B \) contains either a lower (\( L \)) or upper (\( U \)) triangular Cholesky factor. The value of \( \text{inv} \) determines whether the operation, as expressed above, requires an inversion of \( L \) or \( U \). The value of \( \text{uplo} \) determines which triangle of \( A \) is read on input, which triangle of the symmetric (or Hermitian) right-hand side is stored, and also which Cholesky factor exists in \( B \).

Imp. Notes: FLA_Eig_gest_blk_external() and FLA_Eig_gest_unb_external() perform their computation by calling external implementations of the LAPACK routines ?sygst()/?hegst() and ?sygs2()/?hegs2(), respectively. The algorithmic variants employed by these routines, as well as the blocksize used by ?sygst()/?hegst(), are implementation-dependent.

Caveats: FLA_Eig_gest_blk_external() and FLA_Eig_gest_unb_external() are available only if external LAPACK interfaces were enabled at configure-time.

More Info: This function is similar to that of FLA_Eig_gest(). Please see the description for FLA_Eig_gest() for further details.
 Perform a Hermitian eigenvalue decomposition (HEVD):

\[ A \rightarrow U\Lambda U^H \]

where \( \Lambda \) is a diagonal matrix whose elements contain the eigenvalues of \( A \), and the columns of \( U \) contain the eigenvectors of \( A \). The \texttt{jobz} argument determines whether only eigenvalues (FLA_EVD_WITHOUT_VECTORS) or both eigenvalues and eigenvectors (FLA_EVD_WITH_VECTORS) are computed. The \texttt{uplo} argument determines whether \( A \) is stored in the lower or upper triangle. Upon completion, the eigenvalues are stored to the vector \( l \) in ascending order, and the eigenvectors \( U \), if requested, overwrite matrix \( A \) such that vector element \( l_j \) contains the eigenvalue corresponding to the eigenvector stored in the \( j \)th column of \( U \). If eigenvectors are not requested, then the triangle specified by \texttt{uplo} is destroyed.

\textbf{Returns:} FLA_SUCCESS if the operation is successful; otherwise, \( k \) is returned, where \( k \) is the number of off-diagonal elements of the intermediate tridiagonal matrix that failed to converge.

\textbf{Imp. Notes:} \texttt{FLA_Hevd_external()} performs its computation by calling an external implementation of the LAPACK routines \texttt{?heev()} / \texttt{?syev()}. The algorithmic variants employed by these routines, as well as any blocksizes used by subroutines of \texttt{?heev()} / \texttt{?syev()}, are implementation-dependent.

\textbf{Caveats:} \texttt{FLA_Hevd_external()} is available only if external LAPACK interfaces were enabled at configure-time.

\textbf{Constraints:}

- The numerical datatype of \( A \) must be floating-point and must not be FLA_CONSTANT.
- The numerical datatype of \( l \) must be real and must not be FLA_CONSTANT.
- The precision of the datatype of \( l \) must be equal to that of \( A \).
- \( l \) must be a contiguously-stored vector of length \( n \), where \( A \) is \( n \times n \).

\textbf{Arguments:}

- \texttt{jobz} – Indicates whether only eigenvalues or both eigenvalues and eigenvectors are computed.
- \texttt{uplo} – Indicates whether the lower or upper triangle of \( A \) is read during the operation.
- \texttt{A} – An \texttt{FLA_Obj} representing matrix \( A \).
- \texttt{l} – An \texttt{FLA_Obj} representing vector \( l \).
FLA_Error FLA_Svd_external( FLA_Svd_type jobu, FLA_Svd_type jobv, FLA_Obj A, FLA_Obj s, FLA_Obj U, FLA_Obj V );

Purpose: Perform a singular value decomposition (svd):

\[ A \rightarrow U\Sigma V^H \]

where \( \Sigma \) is an \( m \times n \) diagonal matrix whose elements contain the singular values of \( A \), \( U \) is an \( m \times m \) matrix whose columns contain the left singular vectors of \( A \), and \( V \) is an \( n \times n \) matrix whose rows of \( V \) contain the right singular vectors of \( A \). The \( \text{jobu} \) and \( \text{jobv} \) arguments determine if (and how many of) the left and right singular vectors, respectively, are computed and where they are stored. The \( \text{jobu} \) and \( \text{jobv} \) arguments accept the following values:

- \text{FLA\_SVD\_VECTORS\_ALL}. For \( \text{jobu} \): compute all \( m \) columns of \( U \), storing the result in \( U \). For \( \text{jobv} \): compute all \( n \) columns of \( V \), storing the result in \( V \).
- \text{FLA\_SVD\_VECTORS\_MIN\_COPY}. For \( \text{jobu} \): compute the first \( \min(m, n) \) columns of \( U \) and store them in \( U \). For \( \text{jobv} \): compute the first \( \min(m, n) \) columns of \( V \) and store them in \( V \).
- \text{FLA\_SVD\_VECTORS\_MIN\_OVERWRITE}. For \( \text{jobu} \): compute the first \( \min(m, n) \) columns of \( U \) and store them in \( A \). For \( \text{jobv} \): compute the first \( \min(m, n) \) columns of \( V \) and store them in \( A \). Note that \( \text{jobu} \) and \( \text{jobv} \) cannot both be \text{FLA\_SVD\_VECTORS\_MIN\_OVERWRITE}.
- \text{FLA\_SVD\_VECTORS\_NONE}. For \( \text{jobu} \): no columns of \( U \) are computed. For \( \text{jobv} \): no columns of \( V \) are computed.

Upon completion, the \( \min(m, n) \) singular values of \( A \) are stored to \( s \), sorted in descending order and singular vectors, if computed, are stored to either \( A \) or \( U \) and \( V \), depending on the values of \( \text{jobu} \) and \( \text{jobv} \). If neither \( \text{jobu} \) nor \( \text{jobv} \) is \text{FLA\_SVD\_VECTORS\_MIN\_OVERWRITE}, then \( A \) is destroyed.

Returns: \text{FLA\_SUCCESS} if the operation is successful; otherwise, \( k \) is returned, where \( k \) is the number of superdiagonal elements of the intermediate bidiagonal matrix that failed to converge.

Notes: If right singular vectors are requested (ie: \( \text{jobv} \) is not \text{FLA\_SVD\_VECTORS\_NONE}) then \( V^H \) is actually stored rather than \( V \).

Imp. Notes: \text{FLA\_Svd\_external()} performs its computation by calling an external implementation of the LAPACK routines \text{?gesvd()} / \text{?gesvd()}. The algorithmic variants employed by these routines, as well as any blocksizes used by subroutines of \text{?gesvd()} / \text{?gesvd()}, are implementation-dependent.

Caveats: \text{FLA\_Svd\_external()} is available only if external LAPACK interfaces were enabled at configure-time.

Constraints:
- The numerical datatypes of \( A \), \( U \), and \( V \) must be identical and floating-point, and must not be \text{FLA\_CONSTANT}.
- The numerical datatype of \( s \) must be real and must not be \text{FLA\_CONSTANT}.
- The precision of the datatype of \( s \) must be equal to that of \( A \).
- \( e \) must be a contiguously-stored vector of length \( \min(m, n) \), where \( A \) is \( m \times n \).
- \( U \) and \( V \) must be square.
- The order of \( U \) and the order of \( V \) must be equal to the the number of rows in \( A \) and the number of columns in \( A \), respectively.

Arguments:
- \( \text{jobu} \) – Indicates whether the left singular vectors are computed, how many are computed, and where they are stored.
- \( \text{jobv} \) – Indicates whether the right singular vectors are computed, how many are computed, and where they are stored.
- \( A \) – An \text{FLA\_Obj} representing matrix \( A \).
- \( s \) – An \text{FLA\_Obj} representing vector \( s \).
- \( U \) – An \text{FLA\_Obj} representing matrix \( U \).
5.7.3 LAPACK-related utility functions

```c
void FLA_Apply_pivots_unb_external( FLA_Side side, FLA_Trans trans, FLA_Obj p, FLA_Obj A);
```

**Purpose:** Apply a permutation matrix $P$ (or $P^T$) from either the left or the right to a matrix $A$. The permutation matrix $P$, which is not explicitly formed, is encoded by the integer values stored in 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_Sift_pivots_to()` before it may be used with `FLA_Apply_pivots_unb_external()`. Please see the description for `FLA_LU_piv()` in Section 5.6.2 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()`.

**Caveats:** `FLA_Apply_pivots_unb_external()` is only implemented for the case where `side` is left and `trans` is `FLA_NO_TRANSPOSE`.

**Caveats:** `FLA_Apply_pivots_unb_external()` is available only if external LAPACK interfaces were enabled at configure-time.

**Arguments:**
- `side` – Indicates whether the permutation matrix $P$ is applied from the left or the right.
- `trans` – Indicates whether the operation proceeds as if the permutation matrix $P$ were transposed.
- `A` – An FLA_Obj representing matrix $A$.

5.8 LAPACK compatibility (lapack2flame)

As part of the libflame package we provide an LAPACK compatibility layer, which we call lapack2flame, 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 API. More specifically, lapack2flame consists of interfaces that map LAPACK routine invocations to their corresponding libflame implementations. For example, linking your application to an lapack2flame-enabled build of libflame would cause any invocation of `dpotrf()` to invoke the Cholesky factorization implemented by `FLA_Chol()`. The amount of overhead incurred when interfacing to libflame via this compatibility layer is typically small. Much of this overhead stems from needing to initialize and finalize the library within each LAPACK interface routine implementation. If the user initializes libflame a priori, the overhead incurred within the LAPACK interface routine is usually much lower. That said, even in under this ideal usage scenario, the overhead may still be noticeable for very small matrices (ie: those smaller than approximately $100 \times 100$).

Footnote: There are, however, operations for which the overhead is noticeable even at larger problem sizes. This typically is due to libflame needing to recompute intermediate data products that LAPACK routines discard. A noteworthy example is the routine family `?ormqr`/`?unmqr` and `?ormlq`/`?unmlq`, which apply the orthogonal (or unitary) matrix $Q$ that was previously computed via a QR or LQ factorization. The QR and LQ factorizations in LAPACK were designed to preserve only the vector of the $\tau$ values that form the individual Householder transformations. By contrast, the corresponding QR and LQ factorizations in libflame preserve the $b \times b$ triangular factors of the block Householder transformations applied at each step of the blocked factorization algorithm, which are then reused when applying $Q$ or $Q^H$. But since the LAPACK interface does not allow the user to pass in the full triangular factors, the lapack2flame implementation must re-compute the factors on-the-fly before continuing with the application of $Q$. 

5.8.1 Supported routines

This section summarizes the LAPACK interfaces currently supported within `lapack2flame`. Table 5.2 lists all LAPACK interfaces which map directly to functionality implemented within `libflame`.

<table>
<thead>
<tr>
<th>Operation</th>
<th>datatypes supported</th>
<th>LAPACK interface</th>
<th>Maps to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cholesky factorization</td>
<td>sdcz</td>
<td>?potrf()</td>
<td>FLA_Chol()</td>
</tr>
<tr>
<td>LU factorization with partial row pivoting</td>
<td>sdcz</td>
<td>?getrf()</td>
<td>FLA_LU_piv()</td>
</tr>
<tr>
<td>QR factorization</td>
<td>sdcz</td>
<td>?geqrf()</td>
<td>FLA_QR()</td>
</tr>
<tr>
<td>LQ factorization</td>
<td>sdcz</td>
<td>?gelqf()</td>
<td>FLA_LQ()</td>
</tr>
<tr>
<td>Apply $Q$ or $Q^H$ from a QR factorization</td>
<td>sdcz</td>
<td>?ormqr()</td>
<td>FLA_Apply_Q_UT()</td>
</tr>
<tr>
<td>Apply $Q$ or $Q^H$ from an LQ factorization</td>
<td>sdcz</td>
<td>?ormql()</td>
<td>FLA_Apply_Q_UT()</td>
</tr>
<tr>
<td>Triangular matrix inversion</td>
<td>sdcz</td>
<td>?trtri()</td>
<td>FLA_Trinv()</td>
</tr>
<tr>
<td>SPD/HPD matrix inversion</td>
<td>sdcz</td>
<td>?potri()</td>
<td>FLA_Trinv(); FLA_Ttmm()</td>
</tr>
<tr>
<td>Triangular-transpose matrix multiply</td>
<td>sdcz</td>
<td>?lauum()</td>
<td>FLA_Ttmm()</td>
</tr>
<tr>
<td>Triangular Sylvester equation solve</td>
<td>sdcz</td>
<td>?trsyl()</td>
<td>FLA_Sylv()</td>
</tr>
</tbody>
</table>

Table 5.2: A list of LAPACK interfaces supported directly by the `lapack2flame` compatibility layer.
Bibliography


Index

libflame
features, 1
for GNU/Linux and UNIX
configure options, 12
compiling, 20
configuration, 12
hardware support, 10
installing, 21
linking against, 22
linking with lapack2flame, 24
running configure, 18
software requirements, 9
source code, 11
for Microsoft Windows
compiling, 30
configuration, 28
hardware support, 26
installing, 32
linking against, 34
running configure.cmd, 30
software requirements, 25
source code, 26
license, 10, 26
obtaining, 11
version numbering, 11, 26

API
section descriptions, 48

constant
types, see types
values, see types

FLAME/C
notational conventions, 46
object concepts, 49
terminology, 45

FLAME/C functions
FLA_Abort(), 81
FLA_Absolute_square(), 70
FLA_Absolute_value(), 70
FLA_Accum_T_UT(), 196
FLA_Add_to_diag(), 68
FLA_Amax_external(), 214
FLA_Amax(), 115
FLA_Apply_diag_matrix(), 191
FLA_Apply_H2_UT(), 197
FLA_Apply_pivots(), 159
FLA_Apply_pivots_unb_external(), 249
FLA_Apply_Q_blk_external(), 244
FLA_Apply_QUD_UT_create_workspace(), 206
FLA_Apply_QUD_UT(), 181
FLA_Apply_Q_UT_create_workspace(), 205
FLA_Apply_Q_UT(), 179
FLA_Asum_external(), 214
FLA_Asum(), 116
FLA_Axpy_buffer_to_object(), 61
FLA_Axpy external(), 214
FLA_Axpy(), 116
FLA_Axpy object_to_buffer(), 62
FLA_Axpyr external(), 215
FLA_Axpyr(), 118
FLA_Axpyrs external(), 216
FLA_Axpyrs(), 119
FLA_Axpyt external(), 215
FLA_Axpyt(), 117
FLA_Bidiag_UT_create_T(), 204
FLA_Bidiag_UT(), 178
FLA_Bidiag_UT_realify(), 205
FLA_Bidiag_UT recover tau(), 204
FLA_Check_error_level(), 80
FLA_Check_error_level_set(), 80
FLA_Chol blk_external(), 238
FLA_Chol(), 154
FLA_Chol solve(), 155
FLA_Chol unb external(), 238
FLA_Clock(), 82
FLA_Conjugate(), 71
FLA_Conjugate r(), 71
FLA_Cont with 1x3 to 1x2(), 86
FLA_Cont with 3x1 to 2x1(), 85
FLA_Cont with 3x3 to 2x2(), 87
FLA_Copy buffer_to_object(), 59
FLA_Copy external(), 216
FLA_Copy(), 119
FLA_Copy object_to_buffer(), 60
FLA_Copyr external(), 216
FLA_Copyr(), 120
FLA_Copyrt external(), 217
FLA_Copyrt(), 120
FLA_Copyt_external(), 217
FLA_Copyt(), 121
FLA_Dotc_external(), 218
FLA_Dotc(), 122
FLA_Dotcs_external(), 219
FLA_Dotcs(), 124
FLA_Dotc_external(), 218
FLA_Dotc(), 122
FLA_Dots_external(), 220
FLA_Dots(), 123
FLA_Dot2cs_external(), 220
FLA_Dot2cs(), 126
FLA_Dot2s_external(), 220
FLA_Dot2s(), 125
FLA_Eig_gest_blk_external(), 246
FLA_Eig_gest(), 186
FLA_Eig_gest_unb_external(), 246
FLA_Finalize(), 50
FLA_Form_perm_matrix(), 192
FLA_Gemm_external(), 231
FLA_Gemm(), 143
FLA_Gemvc_external(), 224
FLA_Gemvc(), 131
FLA_Gemv_external(), 223
FLA_Gemv(), 130
FLA_Gerc_external(), 225
FLA_Gerc(), 132
FLA_Ger_external(), 224
FLA_Ger(), 132
FLA_Hemm_external(), 232
FLA_Hemm(), 144
FLA_Hemvc_external(), 226
FLA_Hemvc(), 134
FLA_Hemv_external(), 225
FLA_Hemv(), 133
FLA_Herc_external(), 227
FLA_Herc(), 135
FLA_Her_external(), 226
FLA_Her2k_external(), 232
FLA_Herk(), 145
FLA_Hermitianize(), 79
FLA_Her(), 135
FLA_Her2c_external(), 228
FLA_Her2c(), 137
FLA_Her2k_external(), 227
FLA_Her2k(), 146
FLA_Her2(), 136
FLA_Hess_blk_external(), 243
FLA_Hess_unb_external(), 243
FLA_Hess_UT_create_T(), 201
FLA_Hess_UT(), 176
FLA_Hess_UT_recover_tau(), 202
FLA_Hevd_external(), 247
FLA_Hevd(), 189
FLA_Househ2s_UT(), 194
FLA_Househ3D_UT(), 195
FLA_Househ2_UT(), 193
FLA_Initialized(), 50
FLA_Init(), 50
FLA_Invert(), 72
FLA_Inv_scalc_external(), 221
FLA_Inv_scalc(), 127
FLA_Inv_scalc_external(), 221
FLA_Inv_scalc(), 126
FLA_LQ_blk_external(), 242
FLA_LQ_unb_external(), 242
FLA_LQ_UT_create_T(), 199
FLA_LQ_UT_form_Q(), 201
FLA_LQ_UT(), 167
FLA_LQ_UT_recover_tau(), 200
FLA_LQ_UT_solve(), 168
FLA_LUnopiv(), 156
FLA_LU_piv_blk_external(), 240
FLA_LU_piv(), 157
FLA_LU_piv_solve(), 158
FLA_LU_piv_unb_external(), 240
FLA_Lyap(), 188
FLA_Max_abs_value(), 72
FLA_Max_elemwise_diff(), 73
FLA_Memory_leak_counter_set(), 81
FLA_Merge_1x2(), 88
FLA_Merge_2x1(), 88
FLA_Merge_2x2(), 89
FLA_Mult_add(), 73
FLA_Negate(), 73
FLA_Norm_frob(), 75
FLA_Norm_inf(), 74
FLA_Norm1(), 74
FLA_Nrm2_external(), 221
FLA_Nrm2(), 127
FLA_Obj_attach_buffer(), 57
FLA_Obj_base_buffer(), 83
FLA_Obj_base_length(), 82
FLA_Obj_base_width(), 83
FLA_Obj_buffer_at_view(), 57
FLA_Obj_buffer_is_null(), 67
FLA_Obj_col_offset(), 82, 107
FLA_Obj_col_stride(), 58
FLA_Obj_create_buffer(), 56
FLA_Obj_create_conf_to(), 52
FLA_Obj_create_copy_of(), 52
FLA_Obj_create(), 51
FLA_Obj_create_without_buffer(), 55
FLA_Obj_datatype(), 53
FLA_Obj_datatype_proj_to_complex(), 63
FLA_Obj_datatype_proj_to_real(), 62
FLASH_Random_matrix(), 76
FLASH_Random_spd_matrix(), 78
FLASH_Shift_diag(), 69
FLASH_SPDinv(), 185
FLASH_Sylv(), 187
FLASH_Symp(), 147
FLASH_Syrk(), 148
FLASH_Syr2k(), 149
FLASH_Trinv(), 184
FLASH_Trmm(), 150
FLASH_Trsm(), 152
FLASH_Trsv(), 141
FLASH_Ttmm(), 183
FLASH_UDdate_UT_inc_create_hier_matrices(), 212
FLASH_UDdate_UT_inc(), 174
FLASH_UDdate_UT_inc_solve(), 175
FLASH_UDdate_UT_inc_update_rhs(), 175

List of Contributors, v

routines
   FLAME/C, see FLAME/C functions
   FLASH, see FLASH functions
   SuperMatrix, see SuperMatrix functions

SuperMatrix
   description, 110
   integration with FLASH, 115
   preconditions for enabling, 115

SuperMatrix functions
   FLASH_Queue_begin(), 112
   FLASH_Queue_disable_gpu(), 114
   FLASH_Queue_disable(), 111
   FLASH_Queue_enable_gpu(), 114
   FLASH_Queue_enable(), 111
   FLASH_Queue_end(), 112
   FLASH_Queue_get_data_affinity(), 114
   FLASH_Queue_get_enabled_gpu(), 114
   FLASH_Queue_get_enabled(), 111
   FLASH_Queue_get_gpu_num_blocks(), 114
   FLASH_Queue_get_num_threads(), 112
   FLASH_Queue_get_sorting(), 113
   FLASH_Queue_get_verbose_output(), 113
   FLASH_Queue_set_data_affinity(), 113
   FLASH_Queue_set_gpu_num_blocks(), 114
   FLASH_Queue_set_num_threads(), 112
   FLASH_Queue_set_sorting(), 113
   FLASH_Queue_set_verbose_output(), 112

types
   constant-valued, 47
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


A.2 Dissertations


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.

D3 Jack Poulson. Formalized Parallel Dense Linear Algebra and its Application to the Generalized Eigenvalue Problem. Masters Thesis. The University of Texas at Austin, Department of Aerospace Engineering. May 2009. (Supervised by Prof. Jeffrey K. Bennighof)

A.3 Journal Articles


### A.4 Conference Papers


### A.5 FLAME Working Notes


W37 Francisco D. Igual, Gregorio Quintana-Ortí, and Robert van de Geijn. "Level-3 BLAS on a GPU: Picking the Low Hanging Fruit " FLAME Working Note #37. Universidad Jaume I, Depto. de


A.6 Other Technical Reports


Appendix B

License

B.1

BSD 3-CLAUSE LICENSE

libflame is available as free software under the following “3-clause” (also known as “new” or “modified”) BSD license.

Copyright © 2014, The University of Texas at Austin

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

• Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

• Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

• Neither the name of The University of Texas at Austin nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.