
EL

Elemental Manual

Release 0.84

Jack Poulson

November 01, 2015

1	Introduction	1
1.1	Overview	1
1.2	Dependencies	2
1.3	License and copyright	2
2	Build system	5
2.1	Dependencies	5
2.2	Getting Elemental’s source	8
2.3	Building Elemental	8
2.4	Testing the installation	9
2.5	Elemental as a subproject	10
2.6	Troubleshooting	11
3	Core functionality	13
3.1	Imported library routines	13
3.2	Environment	29
3.3	Sequential matrices	37
3.4	Process grids	42
3.5	Distributed matrices	46
3.6	Matrix views	70
3.7	Matrix partitions	72
3.8	Repartitioning matrices	76
3.9	Moving partitions	81
3.10	The “Axy” interface	84
4	Basic linear algebra	89
4.1	Level 1	89
4.2	Level 2	95
4.3	Level 3	98
4.4	Tuning parameters	104
5	High-level linear algebra	105
5.1	Reduction to condensed form	105
5.2	Matrix decompositions	108
5.3	Factorizations	120
5.4	Matrix functions	131
5.5	Permutations	135
5.6	Matrix properties	138
5.7	Linear solvers	149
5.8	Utilities	152

5.9	Tuning parameters	154
6	Convex optimization	155
6.1	ADMM solvers	155
6.2	Utilities	159
7	Control theory	163
7.1	Sylvester	163
7.2	Lyapunov	163
7.3	Algebraic Ricatti	164
8	Special matrices	165
8.1	Deterministic	165
8.2	Random	174
9	Input/output	177
9.1	Display	177
9.2	Print	178
9.3	Spy	178
9.4	Read	178
9.5	Write	178
10	Indices	179
	Index	181

INTRODUCTION

1.1 Overview

Elemental is a library for distributed-memory dense linear algebra that draws heavily from the [PLAPACK](#) approach of building a graph of matrix distributions with a simple interface for redistributions (much of the syntax of the library is also inspired from [FLAME](#)). Elemental is also similar in functionality to [ScaLAPACK](#), which is the very widely used effort towards extending [LAPACK](#) onto distributed-memory architectures. Unlike PLAPACK and ScaLAPACK, Elemental performs virtually all computations using element-wise, rather than block, matrix distributions (please see the first journal publication on Elemental, *Elemental: A new framework for distributed memory dense matrix computations*, for a detailed discussion of this design choice). Some of the unique features of Elemental include distributed implementations of:

- High-performance pseudospectral computation and visualization
- Quadratic-time low-rank Cholesky and LU modifications
- Bunch-Kaufman and Bunch-Parlett for accurate symmetric factorization
- LU and Cholesky with full pivoting
- Column-pivoted QR and interpolative/skeleton decompositions
- Quadratically Weighted Dynamic Halley iteration for the polar decomposition
- Spectral Divide and Conquer Schur decomposition and Hermitian EVD
- Many algorithms for Singular-Value soft-Thresholding (SVT)
- Tall-skinny QR decompositions
- Hermitian matrix functions
- Sign-based Lyapunov/Ricatti/Sylvester solvers

For the sake of objectivity: Elemental is primarily intended to be used from C++11, though interfaces to other languages, such as C, Fortran 90, and Python, are in various stages of development. ScaLAPACK and PLAPACK routines are currently significantly more straightforward to call from C and Fortran.

Note: At this point, the vast majority of Elemental's source code is in header files, so do not be surprised by the sparsity of the `src/` folder; please also look in `include/`. There were essentially two reasons for moving as much of Elemental as possible into header files:

1. In practice, most executables only require a small subset of the library, so avoiding the overhead of compiling the entire library beforehand can be significant. On the other

hand, if one naively builds many such executables with overlapping functionality, then the mainly-header approach becomes slower.

2. Though Elemental does not yet fully support computation over arbitrary fields, the vast majority of its pieces do. Moving templated implementations into header files is a necessary step in the process and also allowed for certain templating techniques to be exploited in order to simplify the class hierarchy.
-

1.2 Dependencies

- Functioning C++11 and ANSI C compilers.
- A working MPI2 implementation.
- BLAS and LAPACK (ideally version 3.3 or greater) implementations.
- CMake (version 2.8.8 or later).

If a sufficiently up-to-date C++11 compiler is used (e.g., recent versions of g++ or clang++), Elemental should be straightforward to build on Unix-like platforms. Building on Microsoft Windows platforms should also be possible with minor effort.

1.3 License and copyright

All files distributed with Elemental are made available under the [New BSD license](#), which states:

```
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 owner 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.
```

Most source files contain the copyright notice:

Copyright (c) 2009-2014, Jack Poulson
All rights reserved.

For an up-to-date list of contributing authors, please see the [AUTHORS file](#).

BUILD SYSTEM

Elemental's build system relies on [CMake](#) in order to manage a large number of configuration options in a platform-independent manner; it can be easily configured to build on Linux and Unix environments (including Darwin), and, at least in theory, various versions of Microsoft Windows. A relatively up-to-date C++11 compiler (e.g., `gcc >= 4.7`) is required in all cases.

Elemental's main dependencies are

1. [CMake](#)
2. [MPI](#)
3. [BLAS](#)
4. [LAPACK](#)

and it includes the package [PMRRR](#), which Elemental uses for its parallel symmetric tridiagonal eigensolver. Furthermore, [libFLAME](#) is recommended for faster SVD's due to its high-performance bidiagonal QR algorithm implementation, and [Qt5](#) is required for matrix visualization.

2.1 Dependencies

2.1.1 CMake

Elemental uses several new CMake modules, so it is important to ensure that version 2.8.8 or later is installed. Thankfully the [installation process](#) is extremely straightforward: either download a platform-specific binary from the [downloads page](#), or instead grab the most recent stable tarball and have CMake bootstrap itself. In the simplest case, the bootstrap process is as simple as running the following commands:

```
./bootstrap
make
make install
```

Note that recent versions of [Ubuntu](#) (e.g., version 13.10) have sufficiently up-to-date versions of CMake, and so the following command is sufficient for installation:

```
sudo apt-get install cmake
```

If you do install from source, there are two important issues to consider

1. By default, `make install` attempts a system-wide installation (e.g., into `/usr/bin`) and will likely require administrative privileges. A different installation folder may be specified with the `--prefix` option to the bootstrap script, e.g.:

```
./bootstrap --prefix=/home/your_username  
make  
make install
```

Afterwards, it is a good idea to make sure that the environment variable `PATH` includes the `bin` subdirectory of the installation folder, e.g., `/home/your_username/bin`.

2. Some highly optimizing compilers will not correctly build CMake, but the GNU compilers nearly always work. You can specify which compilers to use by setting the environment variables `CC` and `CXX` to the full paths to your preferred C and C++ compilers before running the bootstrap script.

Basic usage

Though many configuration utilities, like `autoconf`, are designed such that the user need only invoke `./configure && make && make install` from the top-level source directory, CMake targets *out-of-source* builds, which is to say that the build process occurs away from the source code. The out-of-source build approach is ideal for projects that offer several different build modes, as each version of the project can be built in a separate folder.

A common approach is to create a folder named `build` in the top-level of the source directory and to invoke CMake from within it:

```
mkdir build  
cd build  
cmake ..
```

The last line calls the command line version of CMake, `cmake`, and tells it that it should look in the parent directory for the configuration instructions, which should be in a file named `CMakeLists.txt`. Users that would prefer a graphical interface from the terminal (through `curses`) should instead use `ccmake` (on Unix platforms) or `CMakeSetup` (on Windows platforms). In addition, a GUI version is available through `cmake-gui`.

Though running `make clean` will remove all files generated from running `make`, it will not remove configuration files. Thus, the best approach for completely cleaning a build is to remove the entire build folder. On *nix machines, this is most easily accomplished with:

```
cd ..  
rm -rf build
```

This is a better habit than simply running `rm -rf *` since, if accidentally run from the wrong directory, the former will most likely fail.

2.1.2 MPI

An implementation of the Message Passing Interface (MPI) is required for building Elemental. The two most commonly used implementations are

1. `MPICH2`
2. `OpenMPI`

If your cluster uses [InfiniBand](#) as its interconnect, you may want to look into [MVAPICH2](#).

Each of the respective websites contains installation instructions, but, on recent versions of [Ubuntu](#) (such as version 12.04), [MPICH2](#) can be installed with

```
sudo apt-get install libmpich2-dev
```

and [OpenMPI](#) can be installed with

```
sudo apt-get install libopenmpi-dev
```

2.1.3 BLAS and LAPACK

The Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) are both used heavily within Elemental. On most installations of [Ubuntu](#), the following command should suffice for their installation:

```
sudo apt-get install libatlas-dev liblapack-dev
```

The reference implementation of LAPACK can be found at

<http://www.netlib.org/lapack/>

and the reference implementation of BLAS can be found at

<http://www.netlib.org/blas/>

However, it is better to install an optimized version of these libraries, especially for the BLAS. The most commonly used open source versions are [ATLAS](#) and [OpenBLAS](#). Support for [BLIS](#) is planned in the near future.

2.1.4 PMRRR

PMRRR is a parallel implementation of the MRRR algorithm introduced by [Inderjit Dhillon](#) and [Beresford Parlett](#) for computing k eigenvectors of a tridiagonal matrix of size n in $\mathcal{O}(nk)$ time. PMRRR was written by [Matthias Petschow](#) and [Paolo Bientinesi](#) and, while it is included within Elemental, it is also available at:

<http://code.google.com/p/pmrrr>

Note that PMRRR currently requires support for pthreads.

2.1.5 libFLAME

libFLAME is an open source library made available as part of the FLAME project. Its stated objective is to

...transform the development of dense linear algebra libraries from an art reserved for experts to a science that can be understood by novice and expert alike.

Elemental's current implementation of parallel SVD is dependent upon a serial kernel for the bidiagonal SVD. A high-performance implementation of this kernel was recently introduced in "Restructuring the QR Algorithm for Performance", by Field G. van Zee, Robert A. van de Geijn, and Gregorio Quintana-Orti. It can be found at

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

Installation of *libFLAME* is fairly straightforward. It is recommended that you download the latest nightly snapshot from

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

and then installation should simply be a matter of running:

```
./configure
make
make install
```

2.1.6 Qt5

Qt is an open source cross-platform library for creating Graphical User Interfaces (GUIs) in C++. Elemental currently supports using version 5.1.1 of the library to display and save images of matrices.

Please visit Qt Project's [download page](#) for download and installation instructions. Note that, if Elemental is launched with the *-no-gui* command-line option, then Qt5 will be started without GUI support. This supports using Elemental on clusters whose compute nodes do not run display servers, but PNG's of matrices need to be created using Qt5's simple interface.

2.2 Getting Elemental's source

There are two basic approaches:

1. Download a tarball of the most recent version from libelemental.org/releases. A new version is typically released every one to two months.
2. Install [git](#) and check out a copy of the repository by running

```
git clone git://github.com/elemental/Elemental.git
```

2.3 Building Elemental

On *nix machines with [BLAS](#), [LAPACK](#), and [MPI](#) installed in standard locations, building Elemental can be as simple as:

```
cd elemental
mkdir build
cd build
cmake ..
make
make install
```

As with the installation of CMake, the default install location is system-wide, e.g., `/usr/local`. The installation directory can be changed at any time by running:

```
cmake -D CMAKE_INSTALL_PREFIX=/your/desired/install/path ..
make install
```

Though the above instructions will work on many systems, it is common to need to manually specify several build options, especially when multiple versions of libraries or several different compilers are available on your system. For instance, any C++, C, or Fortran compiler can respectively be set with the `CMAKE_CXX_COMPILER`, `CMAKE_C_COMPILER`, and `CMAKE_Fortran_COMPILER` variables, e.g.,

```
cmake -D CMAKE_CXX_COMPILER=/usr/bin/g++ \
      -D CMAKE_C_COMPILER=/usr/bin/gcc \
      -D CMAKE_Fortran_COMPILER=/usr/bin/gfortran ..
```

It is also common to need to specify which libraries need to be linked in order to provide serial BLAS and LAPACK routines (and, if SVD is important, libFLAME). The `MATH_LIBS` variable was introduced for this purpose and an example usage for configuring with BLAS and LAPACK libraries in `/usr/lib` would be

```
cmake -D MATH_LIBS="-L/usr/lib -llapack -lblas -lm" ..
```

It is important to ensure that if library A depends upon library B, A should be specified to the left of B; in this case, LAPACK depends upon BLAS, so `-llapack` is specified to the left of `-lblas`.

If `libFLAME` is available at `/path/to/libflame.a`, then the above link line should be changed to

```
cmake -D MATH_LIBS="/path/to/libflame.a;-L/usr/lib -llapack -lblas -lm" ..
```

Elemental's performance in Singular Value Decompositions (SVD's) is greatly improved on many architectures when libFLAME is linked.

2.3.1 Build modes

Elemental currently has four different build modes:

- **PureDebug** - An MPI-only build that maintains a call stack and provides more error checking.
- **PureRelease** - An optimized MPI-only build suitable for production use.
- **HybridDebug** - An MPI+OpenMP build that maintains a call stack and provides more error checking.
- **HybridRelease** - An optimized MPI+OpenMP build suitable for production use.

The build mode can be specified with the `CMAKE_BUILD_TYPE` option, e.g., `-D CMAKE_BUILD_TYPE=PureDebug`. If this option is not specified, Elemental defaults to the **PureRelease** build mode.

Once the build mode is selected, one might also want to manually set the optimization level of the compiler, e.g., via the CMake option `-D CXX_FLAGS="-O3"`.

2.4 Testing the installation

Once Elemental has been installed, it is a good idea to verify that it is functioning properly. An example of generating a random distributed matrix, computing its Singular Value Decomposition (SVD), and checking for numerical error is available in [examples/lapack-like/SVD.cpp](#).

As you can see, the only required header is `elemental.hpp`, which must be in the include path when compiling this simple driver, `SVD.cpp`. If Elemental was installed in `/usr/local`, then `/usr/local/conf/ElemVars` can be used to build a simple Makefile:

```
include /usr/local/conf/ElemVars

SVD: SVD.cpp
    ${CXX} ${ELEM_COMPILE_FLAGS} $< -o $@ ${ELEM_LINK_FLAGS} ${ELEM_LIBS}
```

As long as `SVD.cpp` and this Makefile are in the current directory, simply typing `make` should build the driver.

The executable can then typically be run with a single process (generating a 300×300 distributed matrix, using

```
./SVD --height 300 --width 300
```

and the output should be similar to

```
||A||_max = 0.999997
||A||_1   = 165.286
||A||_oo  = 164.116
||A||_F   = 173.012
||A||_2   = 19.7823

||A - U Sigma V^H||_max = 2.20202e-14
||A - U Sigma V^H||_1   = 1.187e-12
||A - U Sigma V^H||_oo  = 1.17365e-12
||A - U Sigma V^H||_F   = 1.10577e-12
||A - U Sigma V_H||_F / (max(m,n) eps ||A||_2) = 1.67825
```

The driver can be run with several processes using the MPI launcher provided by your MPI implementation; a typical way to run the SVD driver on eight processes would be:

```
mpirun -np 8 ./SVD --height 300 --width 300
```

You can also build a wide variety of example and test drivers (unfortunately the line is a little blurred) by using the CMake options:

```
-D ELEM_EXAMPLES=ON
```

and/or

```
-D ELEM_TESTS=ON
```

2.5 Elemental as a subproject

Adding Elemental as a dependency into a project which uses CMake for its build system is relatively straightforward: simply put an entire copy of the Elemental source tree in a subdirectory of your main project folder, say `external/elemental`, and then create a `CMakeLists.txt` file in your main project folder that builds off of the following snippet:

```
cmake_minimum_required(VERSION 2.8.8)
project(Foo)

add_subdirectory(external/elemental)
```

```
include_directories("${PROJECT_BINARY_DIR}/external/elemental/include")
include_directories(${MPI_CXX_INCLUDE_PATH})

# Build your project here
# e.g.,
#   add_library(foo ${LIBRARY_TYPE} ${FOO_SRC})
#   target_link_libraries(foo elemental)
```

2.6 Troubleshooting

If you run into build problems, please email maint@libelemental.org and make sure to attach the file `include/elemental/config.h`, which should be generated within your build directory. Please only direct usage questions to users@libelemental.org, and development questions to dev@libelemental.org.

CORE FUNCTIONALITY

3.1 Imported library routines

Since one of the goals of Elemental is to provide high-performance datatype-independent parallel routines, yet Elemental's dependencies are datatype-dependent, it is convenient to first build a thin datatype-independent abstraction on top of the necessary routines from BLAS, LAPACK, and MPI. The "first-class" datatypes are `float`, `double`, `Complex<float>`, and `Complex<double>`, but `int` and `byte` (unsigned char) are supported for many cases, and support for higher precision arithmetic is in the works.

3.1.1 BLAS

The Basic Linear Algebra Subprograms (BLAS) are heavily exploited within Elemental in order to achieve high performance whenever possible. Since the official BLAS interface uses different routine names for different datatypes, the following interfaces are built directly on top of the datatype-specific versions.

The prototypes can be found in `include/elemental/core/imports/blas.hpp`, while the implementations are in `src/imports/blas.cpp`.

Level 1

`void blas::Axy(int n, T alpha, const T *x, int incx, T *y, int incy)`
Performs $y := \alpha x + y$ for vectors $x, y \in T^n$ and scalar $\alpha \in T$. x and y must be stored such that x_i occurs at `x[i*incx]` (and likewise for y).

`T blas::Dot(int n, const T *x, int incx, T *y, int incy)`
Returns $\alpha := x^H y$, where x and y are stored in the same manner as in `blas::Axy`.

`T blas::Dotc(int n, const T *x, int incx, T *y, int incy)`
Equivalent to `blas::Dot`, but this name is kept for historical purposes (the BLAS provide `?dotc` and `?dotu` for complex datatypes).

`T blas::Dotu(int n, const T *x, int incx, T *y, int incy)`
Similar to `blas::Dot`, but this routine instead returns $\alpha := x^T y$ (x is not conjugated).

`Base<T> blas::Nrm2(int n, const T *x, int incx)`

Return the Euclidean two-norm of the vector x , where $\|x\|_2 = \sqrt{\sum_{i=0}^{n-1} |x_i|^2}$. Note that if T represents a complex field, then the return type is the underlying real field (e.g., `T=Complex<double>` results in a return type of `double`), otherwise T equals the return type.

`void blas::Rot(int n, F *x, int incx, F *y, int incy, Base<F> c, F s)`

Apply a Givens rotation to the length n vectors x and y .

`void blas::Scal(int n, T alpha, T *x, int incx)`

Performs $x := \alpha x$, where $x \in T^n$ is stored in the manner described in `blas::Apy`, and $\alpha \in T$.

`void blas::Swap(int n, T *x, int incx, T *y, int incy)`

Swap the length n vectors x and y .

Level 2

`void blas::Gemv(char trans, int m, int n, T alpha, const T *A, int lda, const T *x, int incx, T beta, T *y, int incy)`

Updates $y := \alpha \text{op}(A)x + \beta y$, where $A \in T^{m \times n}$ and $\text{op}(A) \in \{A, A^T, A^H\}$ is chosen by choosing *trans* from $\{N, T, C\}$, respectively. Note that x is stored in the manner repeatedly described in the Level 1 routines, e.g., `blas::Apy`, but A is stored such that $A(i, j)$ is located at $A[i+j*lda]$.

`void blas::Ger(int m, int n, T alpha, const T *x, int incx, const T *y, int incy, T *A, int lda)`

Updates $A := \alpha xy^H + A$, where $A \in T^{m \times n}$ and x, y , and A are stored in the manner described in `blas::Gev`.

`void blas::Gerc(int m, int n, T alpha, const T *x, int incx, const T *y, int incy, T *A, int lda)`

Equivalent to `blas::Ger`, but the name is provided for historical reasons (the BLAS provides `?gerc` and `?geru` for complex datatypes).

`void blas::Geru(int m, int n, T alpha, const T *x, int incx, const T *y, int incy, T *A, int lda)`

Same as `blas::Ger`, but instead perform $A := \alpha xy^T + A$ (y is not conjugated).

`void blas::Hemv(char uplo, int m, T alpha, const T *A, int lda, const T *x, int incx, T beta, T *y, int incy)`

Performs $y := \alpha Ax + \beta y$, where $A \in T^{m \times n}$ is assumed to be Hermitian with the data stored in either the lower or upper triangle of A (depending upon whether *uplo* is equal to 'L' or 'U', respectively).

`void blas::Her(char uplo, int m, T alpha, const T *x, int incx, T *A, int lda)`

Performs $A := \alpha xx^H + A$, where $A \in T^{m \times m}$ is assumed to be Hermitian, with the data stored in the triangle specified by *uplo* (depending upon whether *uplo* is equal to 'L' or 'U', respectively).

`void blas::Her2(char uplo, int m, T alpha, const T *x, int incx, const T *y, int incy, T *A, int lda)`

Performs $A := \alpha(xy^H + yx^H) + A$, where $A \in T^{m \times m}$ is assumed to be Hermitian, with the data stored in the triangle specified by *uplo* (depending upon whether *uplo* is equal to 'L' or 'U', respectively).

`void blas::Symv(char uplo, int m, T alpha, const T *A, int lda, const T *x, int incx, T beta, T *y, int incy)`

The same as `blas::Hemv`, but $A \in T^{m \times m}$ is instead assumed to be *symmetric*, and the update is $y := \alpha Ax + \beta y$.

Note: The single and double precision complex interfaces, `csymv` and `zsymv`, are technically a part of LAPACK and not BLAS.

```
void blas::Syr(char uplo, int m, T alpha, const T *x, int incx, T *A, int lda)
```

The same as `blas::Her`, but $A \in T^{m \times m}$ is instead assumed to be *symmetric*, and the update is $A := \alpha x x^T + A$.

Note: The single and double precision complex interfaces, `csyr` and `zsyr`, are technically a part of LAPACK and not BLAS.

```
void blas::Syr2(char uplo, int m, T alpha, const T *x, int incx, const T *y, int incy, T *A,
               int lda)
```

The same as `blas::Her2`, but $A \in T^{m \times m}$ is instead assumed to be *symmetric*, and the update is $A := \alpha(x y^T + y x^T) + A$.

Note: The single and double precision complex interfaces do not exist in BLAS or LAPACK, so Elemental instead calls `csyr2k` or `zsyr2k` with $k=1$. This is likely far from optimal, though `Syr2` is not used very commonly in Elemental.

```
void blas::Trmv(char uplo, char trans, char diag, int m, const T *A, int lda, T *x, int incx)
```

Perform the update $x := \alpha \text{op}(A)x$, where $A \in T^{m \times m}$ is assumed to be either lower or upper triangular (depending on whether `uplo` is 'L' or 'U'), unit diagonal if `diag` equals 'U', and $\text{op}(A) \in \{A, A^T, A^H\}$ is determined by `trans` being chosen as 'N', 'T', or 'C', respectively.

```
void blas::Trsv(char uplo, char trans, char diag, int m, const T *A, int lda, T *x, int incx)
```

Perform the update $x := \alpha \text{op}(A)^{-1}x$, where $A \in T^{m \times m}$ is assumed to be either lower or upper triangular (depending on whether `uplo` is 'L' or 'U'), unit diagonal if `diag` equals 'U', and $\text{op}(A) \in \{A, A^T, A^H\}$ is determined by `trans` being chosen as 'N', 'T', or 'C', respectively.

Level 3

```
void blas::Gemm(char transA, char transB, int m, int n, int k, T alpha, const T *A, int lda,
               const T *B, int ldb, T beta, T *C, int ldc)
```

Perform the update $C := \alpha \text{op}_A(A) \text{op}_B(B) + \beta C$, where op_A and op_B are each determined (according to `transA` and `transB`) in the manner described for `blas::Trmv`; it is required that $C \in T^{m \times n}$ and that the inner dimension of $\text{op}_A(A) \text{op}_B(B)$ is k .

```
void blas::Hemm(char side, char uplo, int m, int n, T alpha, const T *A, int lda, const T *B,
               int ldb, T beta, T *C, int ldc)
```

Perform either $C := \alpha AB + \beta C$ or $C := \alpha BA + \beta C$ (depending upon whether `side` is respectively 'L' or 'R') where A is assumed to be Hermitian with its data stored in either the lower or upper triangle (depending upon whether `uplo` is set to 'L' or 'U', respectively) and $C \in T^{m \times n}$.

```
void blas::Her2k(char uplo, char trans, int n, int k, T alpha, const T *A, int lda, const T *B,
                int ldb, T beta, T *C, int ldc)
```

Perform either $C := \alpha(AB^H + BA^H)\beta C$ or $C := \alpha(A^H B + B^H A)\beta C$ (depending upon whether `trans` is respectively 'N' or 'C'), where $C \in T^{n \times n}$ is assumed to be Hermitian, with the data stored in the triangle specified by `uplo` (see `blas::Hemv`) and the inner dimension of AB^H or $A^H B$ is equal to k .

```
void blas::Herk(char uplo, char trans, int n, int k, T alpha, const T *A, int lda, T beta, T *C,
               int ldc)
```

Perform either $C := \alpha AA^H + \beta C$ or $C := \alpha A^H A + \beta C$ (depending upon whether *trans* is respectively 'N' or 'C'), where $C \in T^{n \times n}$ is assumed to be Hermitian with the data stored in the triangle specified by *uplo* (see `blas::Hemv`) and the inner dimension of AA^H or $A^H A$ equal to k .

`void blas::Hetrrmm(char uplo, int n, T *A, int lda)`

Form either $A := L^H L$ or $A := U U^H$, depending upon the choice of *uplo*: if *uplo* equals 'L', then $L \in T^{n \times n}$ is equal to the lower triangle of A , otherwise U is read from the upper triangle of A . In both cases, the relevant triangle of A is overwritten in order to store the Hermitian product.

Note: This routine is built on top of the LAPACK routines `slaaum`, `dlaaum`, `claum`, and `zlaum`; it is in the BLAS section since its functionality is extremely BLAS-like.

`void blas::Symm(char side, char uplo, int m, int n, T alpha, const T *A, int lda, const T *B, int ldb, T beta, T *C, int ldc)`

Perform either $C := \alpha AB + \beta C$ or $C := \alpha BA + \beta C$ (depending upon whether *side* is respectively 'L' or 'R') where A is assumed to be symmetric with its data stored in either the lower or upper triangle (depending upon whether *uplo* is set to 'L' or 'U', respectively) and $C \in T^{m \times n}$.

`void blas::Syr2k(char uplo, char trans, int n, int k, T alpha, const T *A, int lda, const T *B, int ldb, T beta, T *C, int ldc)`

Perform either $C := \alpha (AB^T + BA^T)\beta C$ or $C := \alpha (A^T B + B^T A)\beta C$ (depending upon whether *trans* is respectively 'N' or 'T'), where $C \in T^{n \times n}$ is assumed to be symmetric, with the data stored in the triangle specified by *uplo* (see `blas::Symv`) and the inner dimension of AB^T or $A^T B$ is equal to k .

`void blas::Syrk(char uplo, char trans, int n, int k, T alpha, const T *A, int lda, T beta, T *C, int ldc)`

Perform either $C := \alpha AA^T + \beta C$ or $C := \alpha A^T A + \beta C$ (depending upon whether *trans* is respectively 'N' or 'T'), where $C \in T^{n \times n}$ is assumed to be symmetric with the data stored in the triangle specified by *uplo* (see `blas::Symv`) and the inner dimension of AA^T or $A^T A$ equal to k .

`void blas::Trmm(char side, char uplo, char trans, char unit, int m, int n, T alpha, const T *A, int lda, T *B, int ldb)`

Performs $C := \alpha \text{op}(A)B$ or $C := \alpha B \text{op}(A)$, depending upon whether *side* was chosen as 'L' or 'R', respectively. Whether A is treated as lower or upper triangular is determined by whether *uplo* is 'L' or 'U' (setting *unit* equal to 'U' treats A as unit diagonal, otherwise it should be set to 'N'). *op* is determined in the same manner as in `blas::Trmv`.

`void blas::Trsm(char side, char uplo, char trans, char unit, int m, int n, T alpha, const T *A, int lda, T *B, int ldb)`

Performs $C := \alpha \text{op}(A)^{-1}B$ or $C := \alpha B \text{op}(A)^{-1}$, depending upon whether *side* was chosen as 'L' or 'R', respectively. Whether A is treated as lower or upper triangular is determined by whether *uplo* is 'L' or 'U' (setting *unit* equal to 'U' treats A as unit diagonal, otherwise it should be set to 'N'). *op* is determined in the same manner as in `blas::Trmv`.

3.1.2 LAPACK

A handful of LAPACK routines are currently used by Elemental: a few routines for querying floating point characteristics and a few other utilities. In addition, there are several BLAS-like

routines which are technically part of LAPACK (e.g., `csyr`) which were included in the BLAS imports section.

The prototypes can be found in `include/elemental/core/imports/lapack.hpp`, while the implementations are in `src/imports/lapack.cpp`.

Machine information

In all of the following functions, R can be equal to either *float* or *double*.

`Real lapack::MachineEpsilon<Real>()`

Return the relative machine precision.

`Real lapack::MachineSafeMin<Real>()`

Return the minimum number which can be inverted without underflow.

`Real lapack::MachinePrecision<Real>()`

Return the relative machine precision multiplied by the base.

`Real lapack::MachineUnderflowExponent<Real>()`

Return the minimum exponent before (gradual) underflow occurs.

`Real lapack::MachineUnderflowThreshold<Real>()`

Return the underflow threshold: $(\text{base})^{(\text{underflow exponent})-1}$.

`Real lapack::MachineOverflowExponent<Real>()`

Return the largest exponent before overflow.

`Real lapack::MachineOverflowThreshold<Real>()`

Return the overflow threshold: $(1-\text{rel. prec.}) * (\text{base})^{(\text{overflow exponent})}$.

Safe norms

`Real lapack::SafeNorm(Real alpha, Real beta)`

Return $\sqrt{\alpha^2 + \beta^2}$ in a manner which avoids under/overflow. R can be equal to either *float* or *double*.

`Real lapack::SafeNorm(Real alpha, Real beta, Real gamma)`

Return $\sqrt{\alpha^2 + \beta^2 + \gamma^2}$ in a manner which avoids under/overflow. R can be equal to either *float* or *double*.

Givens rotations

Given $\phi, \gamma \in \mathbb{C}^{n \times n}$, carefully compute $c \in \mathbb{R}$ and $s, \rho \in \mathbb{C}$ such that

$$\begin{bmatrix} c & s \\ -\bar{s} & c \end{bmatrix} \begin{bmatrix} \phi \\ \gamma \end{bmatrix} = \begin{bmatrix} \rho \\ 0 \end{bmatrix},$$

where $c^2 + |s|^2 = 1$ and the mapping from $(\phi, \gamma) \rightarrow (c, s, \rho)$ is “as continuous as possible”, in the manner described by Kahan and Demmel’s “On computing Givens rotations reliably and efficiently”.

`F lapack::Givens(F phi, F gamma, Base<F>*c, F*s)`

Computes a Givens rotation and returns the combined result, ρ .

MRRR-based Hermitian EVP

void lapack::HermitianEig(char uplo, int n, F *A, int lda, Base<F> *w, Base<F> absTol = 0)

void lapack::HermitianEig(char uplo, int n, F *A, int lda, Base<F> *w, F *Z, int ldz, Base<F> absTol = 0)

Compute all eigen-values/pairs of a Hermitian matrix.

void lapack::HermitianEig(char uplo, int n, F *A, int lda, Base<F> *w, int il, int iu, Base<F> absTol = 0)

void lapack::HermitianEig(char uplo, int n, F *A, int lda, Base<F> *w, F *Z, int ldz, int il, int iu, Base<F> absTol = 0)

Compute the il 'th through iu 'th eigen-values/pairs of a Hermitian matrix.

void lapack::HermitianEig(char uplo, int n, F *A, int lda, Base<F> *w, Base<F> vl, Base<F> vu, Base<F> absTol = 0)

void lapack::HermitianEig(char uplo, int n, F *A, int lda, Base<F> *w, F *Z, int ldz, Base<F> vl, Base<F> vu, Base<F> absTol = 0)

Compute the eigen-values/pairs of a Hermitian matrix with eigenvalues in the half-open interval $(v_l, v_u]$.

QR- and DQDS-based SVD

void lapack::QRSVD(int m, int n, F *A, int lda, Base<F> *s, F *U, int ldu, F *VAdj, int ldva)

Computes the singular value decomposition of a general matrix by running the QR algorithm on the condensed bidiagonal matrix.

void lapack::SVD(int m, int n, F *A, int lda, Base<F> *s)

Computes the singular values of a general matrix by running DQDS on the condensed bidiagonal matrix.

Divide-and-conquer SVD

void lapack::DivideAndConquerSVD(int m, int n, F *A, int lda, Base<F> *s, F *U, int ldu, F *VAdj, int ldva)

Computes the SVD of a general matrix using a divide-and-conquer algorithm on the condensed bidiagonal matrix.

Bidiagonal QR

void lapack::BidiagQRAlg(char uplo, int n, int numColsVTrans, int numRowsU, Base<F> *d, Base<F> *e, F *VAdj, int ldva, F *U, int ldu)

Computes the SVD of a bidiagonal matrix using the QR algorithm.

Hessenberg Schur decomposition

void lapack::HessenbergSchur(int n, F *H, int ldh, Complex<Base<F>> *w, bool fullTriangle = false)

Computes the eigenvalues (and possibly the full Schur factor) of an upper Hessenberg matrix using the QR algorithm.

```
void lapack::HessenbergSchur(int n, F *H, int ldh, Complex<Base<F>> *w, F *Q, int
                          ldq, bool fullTriangle = true, bool multiplyQ = false)
    Computes the eigenvalues (and possibly the full Schur factor) as well as the Schur vectors
    of of an upper Hessenberg matrix using the QR algorithm. If multiplyQ is true, then the
    Schur vectors are multiplied against the input matrix from the right.
```

Hessenberg eigenvalues/pairs

```
void lapack::HessenbergEig(int n, F *H, int ldh, Complex<Base<F>> *w)
    Computes the eigenvalues of an upper Hessenberg matrix using the QR algorithm.
```

Note: There are not yet wrappers for computing Hessenberg eigenvectors.

Schur decomposition

```
void lapack::Schur(int n, F *A, int lda, Complex<Base<F>> *w, bool fullTriangle = false)
    Returns the eigenvalues (and possibly also the Schur factor) of a square matrix using the
    QR algorithm.
```

```
void lapack::Schur(int n, F *A, int lda, Complex<Base<F>> *w, F *Q, int ldq, bool full-
                    Triangle = true)
    Returns the Schur decomposition of a square matrix using the QR algorithm.
```

Eigenvalues/pairs

```
void lapack::Eig(int n, F *A, int lda, Complex<Base<F>> *w)
    Returns the eigenvalues of a square matrix using the QR algorithm.
```

Note: There are not yet wrappers for computing general eigenvectors.

3.1.3 MPI

All communication within Elemental is built on top of the Message Passing Interface (MPI). Just like with BLAS and LAPACK, a minimal set of datatype independent abstractions has been built directly on top of the standard MPI interface. This has the added benefit of localizing the changes required for porting Elemental to architectures that do not have full MPI implementations available.

The prototypes can be found in [include/elemental/core/imports/mpi.hpp](#), while the implementations are in [src/imports/mpi.cpp](#).

Datatypes

```
type mpi::Comm
```

```
    MPI_Comm comm
```

```
    Comm(MPI_Comm mpiComm = MPI_COMM_NULL)
```

bool operator==(const mpi::Comm &a, const mpi::Comm &b)

bool operator!=(const mpi::Comm &a, const mpi::Comm &b)

type mpi::Group

MPI_Group group

Group(MPI_Group mpiGroup = MPI_GROUP_NULL)

bool operator==(const mpi::Group &a, const mpi::Group &b)

bool operator!=(const mpi::Group &a, const mpi::Group &b)

type mpi::Op

MPI_Op op

Op(MPI_Op mpiOp = MPI_OP_NULL)

bool operator==(const mpi::Op &a, const mpi::Op &b)

bool operator!=(const mpi::Op &a, const mpi::Op &b)

type mpi::Datatype

Equivalent to MPI_Datatype.

type mpi::ErrorHandler

Equivalent to MPI_Errhandler.

type mpi::Request

Equivalent to MPI_Request.

type mpi::Status

Equivalent to MPI_Status.

type mpi::UserFunction

Equivalent to MPI_User_function.

Constants

const int mpi::ANY_SOURCE

Equivalent to MPI_ANY_SOURCE.

const int mpi::ANY_TAG

Equivalent to MPI_ANY_TAG.

const int mpi::THREAD_SINGLE

Equivalent to MPI_THREAD_SINGLE.

const int mpi::THREAD_FUNNELED

Equivalent to MPI_THREAD_FUNNELED.

const int mpi::THREAD_SERIALIZED

Equivalent to MPI_THREAD_SERIALIZED.

const int mpi::THREAD_MULTIPLE

Equivalent to MPI_THREAD_MULTIPLE.

const int mpi : : UNDEFINED
Equivalent to MPI_UNDEFINED.

const mpi::Group mpi : : GROUP_NULL
Equivalent to MPI_GROUP_NULL.

const mpi::Comm mpi : : COMM_NULL
Equivalent to MPI_COMM_NULL.

const mpi::Comm mpi : : COMM_SELF
Equivalent to MPI_COMM_SELF.

const mpi::Comm mpi : : COMM_WORLD
Equivalent to MPI_COMM_WORLD.

const mpi::ErrorHandler mpi : : ERRORS_RETURN
Equivalent to MPI_ERRORS_RETURN.

const mpi::ErrorHandler mpi : : ERRORS_ARE_FATAL
Equivalent to MPI_ERRORS_ARE_FATAL.

const mpi::Group mpi : : GROUP_EMPTY
Equivalent to MPI_GROUP_EMPTY.

const mpi::Request mpi : : REQUEST_NULL
Equivalent to MPI_REQUEST_NULL.

const mpi::Op mpi : : MAX
Equivalent to MPI_MAX.

const mpi::Op mpi : : MIN
Equivalent to MPI_MIN.

const mpi::Op mpi : : PROD
Equivalent to MPI_PROD.

const mpi::Op mpi : : SUM
Equivalent to MPI_SUM.

const mpi::Op mpi : : LOGICAL_AND
Equivalent to MPI_LAND.

const mpi::Op mpi : : LOGICAL_OR
Equivalent to MPI_LOR.

const mpi::Op mpi : : LOGICAL_XOR
Equivalent to MPI_LXOR.

const mpi::Op mpi : : BINARY_AND
Equivalent to MPI_BAND.

const mpi::Op mpi : : BINARY_OR
Equivalent to MPI_BOR.

const mpi::Op mpi : : BINARY_XOR
Equivalent to MPI_BXOR.

const int mpi : : MIN_COLL_MSG
The minimum message size for collective communication, e.g., the minimum number of elements contributed by each process in an MPI_Allgather. By default, it is hardcoded to

1 in order to avoid problems with MPI implementations that do not support the 0 corner case.

Routines

Environmental

`void mpi::Initialize(int &argc, char **&argv)`
Equivalent of `MPI_Init` (but notice the difference in the calling convention).

```
#include "elemental.hpp"
using namespace elem;

int main( int argc, char* argv[] )
{
    mpi::Initialize( argc, argv );
    ...
    mpi::Finalize();
    return 0;
}
```

`int mpi::InitializeThread(int &argc, char **&argv, int required)`
The threaded equivalent of `mpi::Initialize`; the return integer indicates the level of achieved threading support, e.g., `mpi::THREAD_MULTIPLE`.

`void mpi::Finalize()`
Shut down the MPI environment, freeing all of the allocated resources.

`bool mpi::Initialized()`
Return true if MPI has been initialized.

`bool mpi::Finalized()`
Return true if MPI has been finalized.

`double mpi::Time()`
Return the current wall-time in seconds.

`void mpi::Create(mpi::UserFunction *func, bool commutes, Op &op)`
Create a custom operation for use in reduction routines, e.g., `mpi::Reduce`, `mpi::AllReduce`, and `mpi::ReduceScatter`, where `mpi::UserFunction` could be defined as

```
namespace mpi {
typedef void (UserFunction) ( void* a, void* b, int* length, mpi::Datatype* datatype );
}
```

The *commutes* parameter is also important, as it specifies whether or not the operation $b[i] = a[i] \text{ op } b[i]$, for $i=0, \dots, \text{length}-1$, can be performed in an arbitrary order (for example, using a minimum spanning tree).

`void mpi::Free(mpi::Op &op)`
Free the specified MPI reduction operator.

Communicator manipulation

`int mpi::Rank(mpi::Comm comm)`

Return our rank in the specified communicator.

`int mpi::Size(mpi::Comm comm)`

Return the number of processes in the specified communicator.

`void mpi::Create(mpi::Comm parentComm, mpi::Group subsetGroup, mpi::Comm &subsetComm)`

Create a communicator (*subsetComm*) which is a subset of *parentComm* consisting of the processes specified by *subsetGroup*.

`void mpi::Dup(mpi::Comm original, mpi::Comm &duplicate)`

Create a copy of a communicator.

`void mpi::Split(mpi::Comm comm, int color, int key, mpi::Comm &newComm)`

Split the communicator *comm* into different subcommunicators, where each process specifies the *color* (unique integer) of the subcommunicator it will reside in, as well as its *key* (rank) for the new subcommunicator.

`void mpi::Free(mpi::Comm &comm)`

Free the specified communicator.

`bool mpi::Congruent(mpi::Comm comm1, mpi::Comm comm2)`

Return true if the two communicators consist of the same set of processes (in the same order).

`void mpi::ErrorHandlerSet(mpi::Comm comm, mpi::ErrorHandler errorHandler)`

Modify the specified communicator to use the specified error-handling approach.

Cartesian communicator manipulation

`void mpi::CartCreate(mpi::Comm comm, int numDims, const int *dimensions, const int *periods, bool reorder, mpi::Comm &cartComm)`

Create a Cartesian communicator (*cartComm*) from the specified communicator (*comm*), given the number of dimensions (*numDims*), the sizes of each dimension (*dimensions*), whether or not each dimension is periodic (*periods*), and whether or not the ordering of the processes may be changed (*reorder*).

`void mpi::CartSub(mpi::Comm comm, const int *remainingDims, mpi::Comm &subComm)`

Create this process's subcommunicator of *comm* that results from only keeping the specified dimensions (0 for ignoring and 1 for keeping).

Group manipulation

`int mpi::Rank(mpi::Group group)`

Return our rank in the specified group.

`int mpi::Size(mpi::Group group)`

Return the number of processes in the specified group.

`void mpi::CommGroup(mpi::Comm comm, mpi::Group &group)`

Extract the underlying group from the specified communicator.

`void mpi::Dup(mpi::Group group, mpi::Group &newGroup)`
While `MPI_Group_dup` does not exist, we can mirror its functionality by unioning a group with itself.

`void mpi::Union(mpi::Group groupA, mpi::Group groupB, mpi::Group &newGroup)`
Unions the ranks in groups A and B into a single new group.

`void mpi::Incl(mpi::Group group, int n, const int *ranks, mpi::Group &subGroup)`
Create a subgroup of *group* that consists of the *n* processes whose ranks are specified in the *ranks* array.

`void mpi::Difference(mpi::Group parent, mpi::Group subset, mpi::Group &complement)`
Form a group (*complement*) out of the set of processes which are in the *parent* communicator, but not in the *subset* communicator.

`void mpi::Free(mpi::Group &group)`
Free the specified group.

`int mpi::Translate(mpi::Group origGroup, int origRank, mpi::Group newGroup)`

`int mpi::Translate(mpi::Comm origComm, int origRank, mpi::Group newGroup)`

`int mpi::Translate(mpi::Group origGroup, int origRank, mpi::Comm newComm)`

`int mpi::Translate(mpi::Comm origComm, int origRank, mpi::Comm newComm)`
Return the new rank within *newGroup* (*newComm*) of the process specified by its rank in *origGroup* (*origComm*).

`void mpi::Translate(mpi::Group origGroup, int size, const int *origRanks, mpi::Group newGroup, int *newRanks)`

`void mpi::Translate(mpi::Comm origComm, int size, const int *origRanks, mpi::Group newGroup, int *newRanks)`

`void mpi::Translate(mpi::Group origGroup, int size, const int *origRanks, mpi::Comm newComm, int *newRanks)`

`void mpi::Translate(mpi::Comm origComm, int size, const int *origRanks, mpi::Comm newComm, int *newRanks)`
Return the ranks within *newGroup* (*newComm*) of the *size* processes specified by their ranks in *origGroup* (*origComm*) using the *origRanks* array. The result will be in the *newRanks* array, which must have been preallocated to a length at least as large as *size*.

Utilities

`void mpi::Barrier(mpi::Comm comm)`
Pause until all processes within the *comm* communicator have called this routine.

`void mpi::Wait(mpi::Request &request)`
Pause until the specified request has completed.

`bool mpi::Test(mpi::Request &request)`
Return true if the specified request has completed.

`bool mpi::IProbe(int source, int tag, mpi::Comm comm, mpi::Status &status)`
Return true if there is a message ready which

- is from the process with rank *source* in the communicator *comm* (note that `mpi::ANY_SOURCE` is allowed)

- had the integer tag *tag*

If *true* was returned, then *status* will have been filled with the relevant information, e.g., the source's rank.

```
int mpi::GetCount<T>(mpi::Status &status)
```

Return the number of entries of the specified datatype which are ready to be received.

Point-to-point communication

```
void mpi::Send(const T *buf, int count, int to, int tag, mpi::Comm comm)
```

Send *count* entries of type *T* to the process with rank *to* in the communicator *comm*, and tag the message with the integer *tag*.

```
void mpi::ISend(const T *buf, int count, int to, int tag, mpi::Comm comm, mpi::Request &request)
```

Same as `mpi::Send`, but the call is non-blocking.

```
void mpi::ISend(const T *buf, int count, int to, int tag, mpi::Comm comm, mpi::Request &request)
```

Same as `mpi::ISend`, but the call is in synchronous mode.

```
void mpi::Recv(T *buf, int count, int from, int tag, mpi::Comm comm)
```

Receive *count* entries of type *T* from the process with rank *from* in the communicator *comm*, where the message must have been tagged with the integer *tag*.

```
void mpi::IRecv(T *buf, int count, int from, int tag, mpi::Comm comm, mpi::Request &request)
```

Same as `mpi::Recv`, but the call is non-blocking.

```
void mpi::SendRecv(const T *sendBuf, int sendCount, int to, int sendTag, T *recvBuf, int recvCount, int from, int recvTag, mpi::Comm comm)
```

Send *sendCount* entries of type *T* to process *to*, and simultaneously receive *recvCount* entries of type *T* from process *from*.

Collective communication

```
void mpi::Broadcast(T *buf, int count, int root, mpi::Comm comm)
```

The contents of *buf* (*count* entries of type *T*) on process *root* are duplicated in the local buffers of every process in the communicator.

```
void mpi::Gather(const T *sendBuf, int sendCount, T *recvBuf, int recvCount, int root, mpi::Comm comm)
```

Each process sends an independent amount of data (i.e., *sendCount* entries of type *T*) to the process with rank *root*; the *root* process must specify the maximum number of entries sent from each process, *recvCount*, so that the data received from process *i* lies within the $[i*\text{recvCount}, (i+1)*\text{recvCount})$ range of the receive buffer.

```
void mpi::AllGather(const T *sendBuf, int sendCount, T *recvBuf, int recvCount, mpi::Comm comm)
```

Same as `mpi::Gather`, but every process receives the result.

```
void mpi::Scatter(const T *sendBuf, int sendCount, T *recvBuf, int recvCount, int root, mpi::Comm comm)
```

The same as `mpi::Gather`, but in reverse: the root process starts with an array of data and sends the $[i*\text{sendCount}, (i+1)*\text{sendCount})$ entries to process *i*.

```
void mpi::AllToAll(const T *sendBuf, int sendCount, T *recvBuf, int recvCount,
                  mpi::Comm comm)
```

This can be thought of as every process simultaneously scattering data: after completion, the $[i*recvCount, (i+1)*recvCount)$ portion of the receive buffer on process j will contain the $[j*sendCount, (j+1)*sendCount)$ portion of the send buffer on process i , where $sendCount$ refers to the value specified on process i , and $recvCount$ refers to the value specified on process j .

```
void mpi::AllToAll(const T *sendBuf, const int *sendCounts, const int *sendDispls, T
                  *recvBuf, const int *recvCounts, const int *recvDispls, mpi::Comm
                  comm)
```

Same as previous `mpi::AllToAll`, but the amount of data sent to and received from each process is allowed to vary; after completion, the $[recvDispls[i], recvDispls[i]+recvCounts[i])$ portion of the receive buffer on process j will contain the $[sendDispls[j], sendDispls[j]+sendCounts[j])$ portion of the send buffer on process i .

```
void mpi::Reduce(const T *sendBuf, T *recvBuf, int count, mpi::Op op, int root, mpi::Comm
                 comm)
```

The *root* process receives the result of performing

$S_{p-1} + (S_{n-2} + \dots (S_2 + (S_1 + S_0)) \dots)$, where S_i represents the send buffer of process i , and $+$ represents the operation specified by *op*.

```
void mpi::AllReduce(const T *sendBuf, T *recvBuf, int count, mpi::Op op, mpi::Comm
                   comm)
```

Same as `mpi::Reduce`, but every process receives the result.

```
void mpi::ReduceScatter(const T *sendBuf, T *recvBuf, const int *recvCounts, mpi::Op
                       op, mpi::Comm comm)
```

Same as `mpi::AllReduce`, but process 0 only receives the $[0, recvCounts[0])$ portion of the result, process 1 only receives the $[recvCounts[0], recvCounts[0]+recvCounts[1])$ portion of the result, etc.

3.1.4 PMRRR

Rather than directly using Patschow and Bientinesi's parallel implementation of the Multiple Relatively Robust Representations (MRRR) algorithm, several simplified interfaces have been exposed.

The prototypes can be found in `include/elemental/core/imports/pmrrr.hpp`, while the implementations are in the folder `external/pmrrr/`.

Data structures

```
class herm_tridiag_eig::Estimate
```

For returning upper bounds on the number of local and global eigenvalues with eigenvalues lying in the specified interval, $(a, b]$.

```
int numLocalEigenvalues
```

The upper bound on the number of eigenvalues in the specified interval that our process stores locally.

```
int numGlobalEigenvalues
```

The upper bound on the number of eigenvalues in the specified interval.

```
class herm_tridiag_eig::Info
  For returning information about the computed eigenvalues.

  int numLocalEigenvalues
    The number of computed eigenvalues that our process locally stores.

  int numGlobalEigenvalues
    The number of computed eigenvalues.

  int firstLocalEigenvalue
    The index of the first eigenvalue stored locally on our process.
```

Compute all eigenvalues

```
herm_tridiag_eig::Info herm_tridiag_eig::Eig(int n, double *d, double *e, double *w,
                                             mpi::Comm comm)
  Compute all of the eigenvalues of the real symmetric tridiagonal matrix with diagonal
   $d$  and subdiagonal  $e$ : the eigenvalues will be stored in  $w$  and the work will be divided
  among the processors in  $comm$ .
```

```
herm_tridiag_eig::Info herm_tridiag_eig::Eig(int n, double *d, double *e, double *w,
                                             double *Z, int ldz, mpi::Comm comm)
  Same as above, but also compute the corresponding eigenvectors.
```

Compute eigenvalues within interval

```
herm_tridiag_eig::Info herm_tridiag_eig::Eig(int n, double *d, double *e, double *w,
                                             mpi::Comm comm, double a, double b)
  Only compute the eigenvalues lying within the interval  $(a, b]$ .
```

```
herm_tridiag_eig::Info herm_tridiag_eig::Eig(int n, double *d, double *e, double *w,
                                             double *Z, int ldz, mpi::Comm comm,
                                             double a, double b)
  Same as above, but also compute the corresponding eigenvectors.
```

```
herm_tridiag_eig::Estimate herm_tridiag_eig::EigEstimate(int n, double *d, double *w,
                                                         mpi::Comm comm, double
                                                         a, double b)
  Return upper bounds on the number of local and global eigenvalues lying within the
  specified interval.
```

Compute eigenvalues in index range

```
herm_tridiag_eig::Info herm_tridiag_eig::Eig(int n, double *d, double *e, double *w,
                                             mpi::Comm comm, int a, int b)
  Only compute the eigenvalues with indices ranging from  $a$  to  $b$ , where  $0 \leq a \leq b < n$ .
```

```
herm_tridiag_eig::Info herm_tridiag_eig::Eig(int n, double *d, double *e, double *w,
                                             double *Z, int ldz, mpi::Comm comm, int
                                             a, int b)
  Same as above, but also compute the corresponding eigenvectors.
```

3.1.5 libFLAME

```
int FLA_Bsvd_v_opd_var1 (int k, int mU, int mV, int nGH, int nIterMax, double *d, int dInc,
                        double *e, int eInc, Complex<double> *G, int rsG, int csG, Com-
                        plex<double> *H, int rsH, int csH, double *U, int rsU, int csU,
                        double *V, int rsV, int csV, int nb)
```

```
int FLA_Bsvd_v_opd_var1 (int k, int mU, int mV, int nGH, int nIterMax, double *d, int dInc,
                        double *e, int eInc, Complex<double> *G, int rsG, int csG, Com-
                        plex<double> *H, int rsH, int csH, Complex<double> *U, int
                        rsU, int csU, Complex<double> *V, int rsV, int csV, int nb)
```

Optional high-performance implementations of the bidiagonal QR algorithm. This can lead to substantial improvements in Elemental's distributed-memory SVD on supported architectures (as of now, modern Intel architectures).

3.1.6 BLACS

A handful of Basic Linear Algebra Communication Subprograms (BLACS) routines are currently exposed to Elemental if MATH_LIBS was detected to contain a recent version of ScaLAPACK.

```
int blacs::Handle (MPI_Comm comm)
```

```
int blacs::GridInit (int bhandle, bool colMajor, int gridHeight, int gridWidth)
```

```
int blacs::GridHeight (int context)
```

```
int blacs::GridWidth (int context)
```

```
int blacs::GridRow (int context)
```

```
int blacs::GridCol (int context)
```

```
void blacs::FreeHandle (int bhandle)
```

```
void blacs::FreeGrid (int context)
```

```
void blacs::Exit (bool finished = true)
```

```
type blacs::Desc
```

Currently a typedef to `std::array<int, 9>`.

```
blacs::Desc FillDesc (const BlockDistMatrix<F> &A, int context)
```

Form the descriptor for the distributed matrix *A* using the specified corresponding context.

3.1.7 ScaLAPACK

A handful of ScaLAPACK routines are currently exposed to Elemental if MATH_LIBS was detected to contain a recent version of ScaLAPACK.

Hessenberg Schur decomposition

```
void scalapack::HessenbergSchur(int n, F *H, int *desch, Complex<Base<F>> *w, bool
                               fullTriangle = false, bool aed = false)
```

Computes the eigenvalues (and possibly the full Schur factor) of an upper Hessenberg matrix using the QR algorithm. Aggressive Early Deflation may be optionally used for real matrices, but there are known bugs in the ScaLAPACK implementation.

```
void scalapack::HessenbergSchur(int n, F *H, int *desch, Complex<Base<F>> *w, F *Q,
                               int *descq, bool fullTriangle = true, bool multiplyQ =
                               false, bool aed = false)
```

Computes the eigenvalues (and possibly the full Schur factor) as well as the Schur vectors of an upper Hessenberg matrix using the QR algorithm. If *multiplyQ* is true, then the Schur vectors are multiplied against the input matrix from the right. Aggressive Early Deflation may be optionally used for real matrices, but there are known bugs in the ScaLAPACK implementation.

Hessenberg eigenvalues/pairs

```
void scalapack::HessenbergEig(int n, F *H, int *desch, Complex<Base<F>> *w)
```

Computes the eigenvalues of an upper Hessenberg matrix using the QR algorithm.

Note: There are not yet wrappers for computing Hessenberg eigenvectors.

3.2 Environment

This section describes the routines and data structures which help set up Elemental's programming environment: it discusses initialization of Elemental, call stack manipulation, a custom data structure for complex data, many routines for manipulating real and complex data, a litany of custom enums, and a few useful routines for simplifying index calculations.

3.2.1 Build and version information

Every Elemental driver with proper command-line argument processing will run *PrintVersion* if the `--version` argument is used. If `--build` is used, then all of the below information is reported.

```
void PrintVersion(std::ostream &os = std::cout)
```

Prints the Git revision, (pre-)release version, and build type. For example:

```
Elemental version information:
  Git revision: 3c6fbdaad901a554fc27a83378d63dab55af0dd3
  Version:      0.81-dev
  Build type:   PureDebug
```

```
void PrintConfig(std::ostream &os = std::cout)
```

Prints the relevant configuration details. For example:

```
Elemental configuration:
  Math libraries: /usr/lib/liblapack.so;/usr/lib/libblas.so
  HAVE_F90_INTERFACE
```

```
HAVE_MPI_REDUCE_SCATTER_BLOCK
HAVE_MPI_IN_PLACE
USE_BYTE_ALLGATHERS
```

`void PrintCCompilerInfo(std::ostream &os = std::cout)`

Prints the relevant C compilation information. For example:

```
Elemental's C compiler info:
CMAKE_C_COMPILER:    /usr/local/bin/gcc
MPI_C_COMPILER:      /home/poulson/Install/bin/mpicc
MPI_C_INCLUDE_PATH:  /home/poulson/Install/include
MPI_C_COMPILE_FLAGS:
MPI_C_LINK_FLAGS:    -Wl,-rpath -Wl,/home/poulson/Install/lib
MPI_C_LIBRARIES:     /home/poulson/Install/lib/libmpich.so;/home/poulson/Install/lib/libopa.s
```

`void PrintCxxCompilerInfo(std::ostream &os = std::cout)`

Prints the relevant C++ compilation information. For example:

```
Elemental's C++ compiler info:
CMAKE_CXX_COMPILER:  /usr/local/bin/g++
CXX_FLAGS:           -Wall
MPI_CXX_COMPILER:    /home/poulson/Install/bin/mpicxx
MPI_CXX_INCLUDE_PATH: /home/poulson/Install/include
MPI_CXX_COMPILE_FLAGS:
MPI_CXX_LINK_FLAGS:  -Wl,-rpath -Wl,/home/poulson/Install/lib
MPI_CXX_LIBRARIES:   /home/poulson/Install/lib/libmpichcxx.so;/home/poulson/Install/lib/lib
```

3.2.2 Set up and clean up

`void Initialize(int &argc, char **&argv)`

Initializes Elemental and (if necessary) MPI. The usage is very similar to `MPI.Init`, but the `argc` and `argv` can be directly passed in.

```
#include "elemental.hpp"
int main( int argc, char* argv[] )
{
    elem::Initialize( argc, argv );
    // ...
    elem::Finalize();
    return 0;
}
```

`void Finalize()`

Frees all resources allocated by Elemental and (if necessary) MPI.

`bool Initialized()`

Returns whether or not Elemental is currently initialized.

`void ReportException(std::exception &e)`

Used for handling Elemental's various exceptions, e.g.,

```
#include "elemental.hpp"
int main( int argc, char* argv[] )
{
    elem::Initialize( argc, argv );
    try {
```

```

    // ...
} catch( std::exception& e ) { ReportException(e); }
elem::Finalize();
return 0;
}

```

3.2.3 Blocksize manipulation

`int Blocksize()`

Return the currently chosen algorithmic blocksize. The optimal value depends on the problem size, algorithm, and architecture; the default value is 128.

`void SetBlocksize(int blocksize)`

Change the algorithmic blocksize to the specified value.

`void PushBlocksizeStack(int blocksize)`

It is frequently useful to temporarily change the algorithmic blocksize, so rather than having to manually store and reset the current state, one can simply push a new value onto a stack (and later pop the stack to reset the value).

`void PopBlocksizeStack()`

Pops the stack of blocksizes. See above.

`int DefaultBlockHeight()`

`int DefaultBlockWidth()`

Returns the default block height (width) for `BlockDistMatrix<T,U,V>`.

`void SetDefaultBlockHeight(int mb)`

`void SetDefaultBlockWidth(int nb)`

Change the default block height (width) for `BlockDistMatrix<T,U,V>`.

3.2.4 Default process grid

`Grid &DefaultGrid()`

Return a process grid built over `mpi::COMM_WORLD`. This is typically used as a means of allowing instances of the `DistMatrix<T,MC,MR>` class to be constructed without having to manually specify a process grid, e.g.,

```

// Build a 10 x 10 distributed matrix over mpi::COMM_WORLD
elem::DistMatrix<T,MC,MR> A( 10, 10 );

```

3.2.5 Call stack manipulation

Note: The following call stack manipulation routines are only available in non-release builds (i.e., `PureDebug` and `HybridDebug`) and are meant to allow for the call stack to be printed (via `DumpCallStack()`) when an exception is caught.

`void PushCallStack(std::string s)`

Push the given routine name onto the call stack.

void PopCallStack()

Remove the routine name at the top of the call stack.

void DumpCallStack()

Print (and empty) the contents of the call stack.

3.2.6 Custom exceptions

class SingularMatrixException

An extension of `std::runtime_error` which is meant to be thrown when a singular matrix is unexpectedly encountered.

SingularMatrixException(**const** char *msg = "Matrix was singular")

Builds an instance of the exception which allows one to optionally specify the error message.

```
throw elem::SingularMatrixException();
```

class NonHPDMatrixException

An extension of `std::runtime_error` which is meant to be thrown when a non positive-definite Hermitian matrix is unexpectedly encountered (e.g., during Cholesky factorization).

NonHPDMatrixException(**const** char *msg = "Matrix was not HPD")

Builds an instance of the exception which allows one to optionally specify the error message.

```
throw elem::NonHPDMatrixException();
```

class NonHPSDMatrixException

An extension of `std::runtime_error` which is meant to be thrown when a non positive semi-definite Hermitian matrix is unexpectedly encountered (e.g., during computation of the square root of a Hermitian matrix).

NonHPSDMatrixException(**const** char *msg = "Matrix was not HPSD")

Builds an instance of the exception which allows one to optionally specify the error message.

```
throw elem::NonHPSDMatrixException();
```

3.2.7 Complex data

type Complex<Real>

Currently a typedef of `std::complex<Real>`

type Base<F>

The underlying real datatype of the (potentially complex) datatype *F*. For example, `Base<Complex<double>>` and `Base<double>` are both equivalent to `double`. This is often extremely useful in implementing routines which are templated over real and complex datatypes but still make use of real datatypes.

`std::ostream &operator<<(std::ostream &os, Complex<Real> alpha)`

Pretty prints *alpha* in the form *a+bi*.

```
type scomplex
    typedef Complex<float> scomplex;
```

```
type dcomplex
    typedef Complex<double> dcomplex;
```

3.2.8 Scalar manipulation

Base<F> Abs(**const** F &*alpha*)

Return the absolute value of the real or complex variable α .

F FastAbs(**const** F &*alpha*)

Return a cheaper norm of the real or complex α :

$$|\alpha|_{\text{fast}} = |\mathcal{R}(\alpha)| + |\mathcal{I}(\alpha)|$$

F RealPart(**const** F &*alpha*)

F ImagPart(**const** F &*alpha*)

Return the real (imaginary) part of the real or complex variable α .

void SetRealPart(F &*alpha*, *Base*<F> &*beta*)

void SetImagPart(F &*alpha*, *Base*<F> &*beta*)

Set the real (imaginary) part of the real or complex variable α to β . If α has a real type, an error is thrown when an attempt is made to set the imaginary component.

void UpdateRealPart(F &*alpha*, *Base*<F> &*beta*)

void UpdateImagPart(F &*alpha*, *Base*<F> &*beta*)

Update the real (imaginary) part of the real or complex variable α to β . If α has a real type, an error is thrown when an attempt is made to update the imaginary component.

F Conj(**const** F &*alpha*)

Return the complex conjugate of the real or complex variable α .

F Sqrt(**const** F &*alpha*)

Returns the square root of the real or complex variable α .

F Cos(**const** F &*alpha*)

Returns the cosine of the real or complex variable α .

F Sin(**const** F &*alpha*)

Returns the sine of the real or complex variable α .

F Tan(**const** F &*alpha*)

Returns the tangent of the real or complex variable α .

F Cosh(**const** F &*alpha*)

Returns the hyperbolic cosine of the real or complex variable α .

F Sinh(**const** F &*alpha*)

Returns the hyperbolic sine of the real or complex variable α .

Base<F> Arg(**const** F &*alpha*)

Returns the argument of the real or complex variable α .

Complex<Real> Polar(**const** R &*r*, **const** R &*theta* = 0)

Returns the complex variable constructed from the polar coordinates (r, θ) .

`F Exp(const F &alpha)`
Returns the exponential of the real or complex variable α .

`F Pow(const F &alpha, const F &beta)`
Returns α^β for real or complex α and β .

`F Log(const F &alpha)`
Returns the logarithm of the real or complex variable α .

3.2.9 Other typedefs and enums

type `byte`
 typedef unsigned char `byte`;

enum `Conjugation`

enumerator `CONJUGATED`

enumerator `UNCONJUGATED`

enum `Distribution`

For specifying the distribution of a row or column of a distributed matrix:

enumerator `MC`

 Column of a standard matrix distribution

enumerator `MD`

 Diagonal of a standard matrix distribution

enumerator `MR`

 Row of a standard matrix distribution

enumerator `VC`

 Column-major vector distribution

enumerator `VR`

 Row-major vector distribution

enumerator `STAR`

 Redundantly stored on every process

enumerator `CIRC`

 Stored on a single process

enum `ForwardOrBackward`

enumerator `FORWARD`

enumerator `BACKWARD`

enum `GridOrder`

For specifying either a `ROW_MAJOR` or `COLUMN_MAJOR` ordering; it is used to decide how to construct process grids and is also useful for tuning one of the algorithms in *HermitianTridiag()* which requires building a smaller square process grid from a rectangular process grid, as the ordering of the processes can greatly impact performance. See *SetHermitianTridiagGridOrder()*.

enumerator `ROW_MAJOR`

enumerator COLUMN_MAJOR

enum LeftOrRight

enumerator LEFT

enumerator RIGHT

enum SortType

For specifying a sorting strategy:

enumerator UNSORTED

Do not sort

enumerator DESCENDING

Largest values first

enumerator ASCENDING

Smallest values first

enum NormType

enumerator ONE_NORM

$$\|A\|_1 = \max_{\|x\|_1=1} \|Ax\|_1 = \max_j \sum_{i=0}^{m-1} |\alpha_{i,j}|$$

enumerator INFINITY_NORM

$$\|A\|_\infty = \max_{\|x\|_\infty=1} \|Ax\|_\infty = \max_i \sum_{j=0}^{n-1} |\alpha_{i,j}|$$

enumerator ENTRYWISE_ONE_NORM

$$\|\text{vec}(A)\|_1 = \sum_{i,j} |\alpha_{i,j}|$$

enumerator MAX_NORM

$$\|A\|_{\max} = \max_{i,j} |\alpha_{i,j}|$$

enumerator NUCLEAR_NORM

$$\|A\|_* = \sum_{i=0}^{\min(m,n)} \sigma_i(A)$$

enumerator FROBENIUS_NORM

$$\|A\|_F = \sqrt{\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} |\alpha_{i,j}|^2} = \sqrt{\sum_{i=0}^{\min(m,n)} \sigma_i(A)^2}$$

enumerator TWO_NORM

$$\|A\|_2 = \max_i \sigma_i(A)$$

enum Orientation

enumerator NORMAL

Do not transpose or conjugate

enumerator TRANSPOSE

Transpose without conjugation

enumerator ADJOINT

Transpose with conjugation

enum UnitOrNonUnit

enumerator UNIT

enumerator NON_UNIT

enum UpperOrLower

enumerator LOWER

enumerator UPPER

enum VerticalOrHorizontal

enumerator VERTICAL

enumerator HORIZONTAL

3.2.10 Indexing utilities

int Shift(*int rank*, *int firstRank*, *int numProcs*)

Given a element-wise cyclic distribution over *numProcs* processes, where the first entry is owned by the process with rank *firstRank*, this routine returns the first entry owned by the process with rank *rank*.

int Length(*int n*, *int shift*, *int numProcs*)

Given a vector with *n* entries distributed over *numProcs* processes with shift as defined above, this routine returns the number of entries of the vector which are owned by this process.

int Length(*int n*, *int rank*, *int firstRank*, *int numProcs*)

Given a vector with *n* entries distributed over *numProcs* processes, with the first entry owned by process *firstRank*, this routine returns the number of entries locally owned by the process with rank *rank*.

int MaxLength(*int n*, *int numProcs*)

The maximum result of *Length()* with the given parameters. This is useful for padding collective communication routines which are almost regular.

int Mod(*int a*, *int b*)

An extension of C++'s % operator which handles cases where *a* is negative and still returns a result in $[0, b)$.

int GCD(int *a*, int *b*)

Return the greatest common denominator of the integers *a* and *b*.

unsigned Log2(unsigned *n*)

Return the base-two logarithm of a positive integer.

bool PowerOfTwo(unsigned *n*)

Return whether or not a positive integer is an integer power of two.

3.3 Sequential matrices

The `Matrix<T>` class is the building of the library: its purpose is to provide convenient mechanisms for performing basic matrix manipulations, such as setting and querying individual matrix entries, without giving up compatibility with interfaces such as BLAS and LAPACK, which assume column-major storage.

An example of generating an $m \times n$ matrix of real double-precision numbers where the (i, j) entry is equal to $i - j$ would be:

```
#include "elemental.hpp"
using namespace elem;
...
Matrix<double> A( m, n );
for( int j=0; j<n; ++j )
    for( int i=0; i<m; ++i )
        A.Set( i, j, double(i-j) );
```

whereas the complex double-precision equivalent could use `Complex<double>`, which is currently a typedef for `std::complex<double>`.

The underlying data storage for `Matrix<T>` is simply a contiguous buffer that stores entries in a column-major fashion with a *leading dimension* which is only required to be at least as large as the height of the matrix (so that entry (i, j) is located at position $i+j*\text{ldim}$). For modifiable instances of the `Matrix<T>` class, the routine `Matrix<T>::Buffer()` returns a pointer to the underlying buffer, while `Matrix<T>::LDim()` returns the leading dimension; these two routines could be used to directly perform the equivalent of the first code sample as follows:

```
#include "elemental.hpp"
using namespace elem;
...
Matrix<double> A( m, n );
double* buffer = A.Buffer();
const int ldim = A.LDim();
for( int j=0; j<n; ++j )
    for( int i=0; i<m; ++i )
        buffer[i+j*ldim] = double(i-j);
```

For immutable instances of the `Matrix<T>` class, a const pointer to the underlying data can similarly be returned with a call to `Matrix<T>::LockedBuffer()`. In addition, a (const) pointer to the place in the (const) buffer where entry (i, j) resides can be easily retrieved with a call to `Matrix<T>::Buffer()` or `Matrix<T>::LockedBuffer()`.

It is also important to be able to create matrices which are simply *views* of existing (sub)matrices. For example, if *A* is a 10×10 matrix of complex doubles, then a matrix *A_{BR}* can easily be created to view the bottom-right 6×7 submatrix using

```
#include "elemental.hpp"
...
auto ABR = View( A, 1, 2, 3, 4 );
```

since the bottom-right 3×4 submatrix beings at index $(1, 2)$. In general, to view the $M \times N$ submatrix starting at entry (i, j) , one would call `View(ABR, A, i, j, M, N);`.

class `Matrix<T>`

The goal is for the *Matrix* class to support any datatype T which supports both addition and multiplication and has the associated identities (that is, when the datatype T is a *ring*). While there are several barriers to reaching this goal, it is important to keep in mind that, in addition to T being allowed to be a real or complex (single- or double-precision) floating-point type, signed integers are also supported.

Constructors and destructors

Note: Many of the following constructors have the default parameter `bool fixed=false`, which can be changed to `true` in order to produce a *Matrix* whose entries can be modified, but the matrix's dimensions cannot. This is useful for the *DistMatrix<T>* class, which contains a local *Matrix<T>* whose entries can be locally modified in cases where it would not make sense to change the local matrix size (which should instead result from changing the size of the full distributed matrix).

`Matrix(bool fixed = false)`

This simply creates a default 0×0 matrix with a leading dimension of one (BLAS and LAPACK require positive leading dimensions).

`Matrix(int height, int width, bool fixed = false)`

A $height \times width$ matrix is created with an unspecified leading dimension (though it is currently implemented as $\max(height, 1)$).

`Matrix(int height, int width, int ldim, bool fixed = false)`

A $height \times width$ matrix is created with a leading dimension equal to *ldim* (which must be greater than or equal $\max(height, 1)$).

`Matrix(int height, int width, const T *buffer, int ldim, bool fixed = false)`

`Matrix(int height, int width, T *buffer, int ldim, bool fixed = false)`

A matrix is built around a column-major (immutable) buffer with the specified dimensions. The memory pointed to by *buffer* should not be freed until after the *Matrix<T>* object is destructed.

`Matrix(const Matrix<T> &A)`

A copy (not a view) of the matrix *A* is built.

`Matrix(Matrix<T> &&A) noexcept`

A C++11 move constructor which creates a new matrix by moving the metadata from the specified matrix over to the new matrix, which cheaply gives the new matrix control over the resources originally assigned to the input matrix.

`~Matrix()`

Frees all resources owned by the matrix upon destruction.

Assignment and reconfiguration

Matrix<T> &operator=(const *Matrix*<T> &A)

Create a full copy of the specified matrix.

Matrix<T> &operator=(*Matrix*<T> &&A)

A C++11 move assignment which swaps the metadata of two matrices so that the resources owned by the two objects will have been cheaply switched.

void Empty()

Sets the matrix to 0×0 and frees any owned resources.

void Resize(int *height*, int *width*)

Reconfigures the matrix to be *height* \times *width*.

void Resize(int *height*, int *width*, int *ldim*)

Reconfigures the matrix to be *height* \times *width*, but with leading dimension equal to *ldim* (which must be greater than or equal to $\max(\text{height}, 1)$).

void Attach(int *height*, int *width*, T **buffer*, int *ldim*)

void LockedAttach(int *height*, int *width*, const T **buffer*, int *ldim*)

Reconfigure the matrix around the specified (unmodifiable) buffer.

void Control(int *height*, int *width*, T **buffer*, int *ldim*)

Reconfigure the matrix around a specified buffer and give ownership of the resource to the matrix.

Basic queries

int Height() const

int Width() const

Return the height/width of the matrix.

int LDim() const

Return the leading dimension of the underlying buffer.

int MemorySize() const

Return the number of entries of type *T* that this *Matrix*<*T*> instance has allocated space for.

int DiagonalLength(int *offset* = 0) const

Return the length of the specified diagonal of the matrix: an offset of 0 refers to the main diagonal, an offset of 1 refers to the superdiagonal, an offset of -1 refers to the subdiagonal, etc.

T *Buffer()

const T *LockedBuffer() const

Return a pointer to the (immutable) underlying buffer.

T *Buffer(int *i*, int *j*)

const T *LockedBuffer(int *i*, int *j*) const

Return a pointer to the (immutable) portion of the buffer that holds entry (*i*, *j*).

`bool Viewing() const`
Returns *true* if the underlying buffer is merely a pointer into an externally-owned buffer.

`bool FixedSize() const`
Returns *true* if the dimensions of the matrix cannot be changed.

`bool Locked() const`
Returns *true* if the entries of the matrix cannot be changed.

Single-entry manipulation

`T Get(int i, int j) const`

`Base<T> GetRealPart(int i, int j) const`

`Base<T> GetImagPart(int i, int j) const`
Return entry (i, j) (or its real or imaginary part).

`void Set(int i, int j, T alpha)`

`void SetRealPart(int i, int j, Base<T> alpha)`

`void SetImagPart(int i, int j, Base<T> alpha)`
Set entry (i, j) (or its real or imaginary part) to α .

`void Update(int i, int j, T alpha)`

`void UpdateRealPart(int i, int j, Base<T> alpha)`

`void UpdateImagPart(int i, int j, Base<T> alpha)`
Add α to entry (i, j) (or its real or imaginary part).

`void MakeReal(int i, int j)`
Force the (i, j) entry to be real.

`void Conjugate(int i, int j)`
Conjugate the (i, j) entry of the matrix.

Diagonal manipulation

`void GetDiagonal(Matrix<T> &d, int offset = 0) const`

`void GetRealPartOfDiagonal(Matrix<Base<T>> &d, int offset = 0) const`

`void GetImagPartOfDiagonal(Matrix<Base<T>> &d, int offset = 0) const`
Modify d into a column-vector containing the entries (or their real or imaginary parts) lying on the *offset* diagonal of our matrix (for instance, the main diagonal has offset 0, the subdiagonal has offset -1 , and the superdiagonal has offset $+1$).

`Matrix<T> GetDiagonal(int offset = 0) const`

`Matrix<Base<T>> GetRealPartOfDiagonal(int offset = 0) const`

`Matrix<Base<T>> GetImagPartOfDiagonal(int offset = 0) const`
Efficiently construct and return the particular diagonal (or its real or imaginary part) via C++11 move semantics.

`void SetDiagonal(const Matrix<T> &d, int offset = 0)`

void SetRealPartOfDiagonal(**const** *Matrix*<*Base*<T>> &*d*, int *offset* = 0)

void SetImagPartOfDiagonal(**const** *Matrix*<*Base*<T>> &*d*, int *offset* = 0)
Set the entries (or their real or imaginary parts) in the *offset* diagonal entries from the contents of the column-vector *d*.

void UpdateDiagonal(**const** *Matrix*<T> &*d*, int *offset* = 0)

void UpdateRealPartOfDiagonal(**const** *Matrix*<*Base*<T>> &*d*, int *offset* = 0)

void UpdateImagPartOfDiagonal(**const** *Matrix*<*Base*<T>> &*d*, int *offset* = 0)
Add the contents of *d* onto the entries (or the real or imaginary parts) in the *offset* diagonal.

void MakeDiagonalReal(int *offset* = 0)
Force the specified diagonal of the matrix to be real.

void ConjugateDiagonal(int *offset* = 0)
Conjugate the specified diagonal of the matrix.

Arbitrary-submatrix manipulation

void GetSubmatrix(**const** std::vector<int> &*rowInd*, **const** std::vector<int> &*colInd*, *Matrix*<T> &*ASub*) **const**

void GetRealPartOfSubmatrix(**const** std::vector<int> &*rowInd*, **const** std::vector<int> &*colInd*, *Matrix*<*Base*<T>> &*ASub*) **const**

void GetImagPartOfSubmatrix(**const** std::vector<int> &*rowInd*, **const** std::vector<int> &*colInd*, *Matrix*<*Base*<T>> &*ASub*) **const**
Return the submatrix (or its real or imaginary part) with the specified row and column indices via *ASub*.

Matrix<T> GetSubmatrix(**const** std::vector<int> &*rowInd*, **const** std::vector<int> &*colInd*) **const**

Matrix<*Base*<T>> GetRealPartOfSubmatrix(**const** std::vector<int> &*rowInd*, **const** std::vector<int> &*colInd*) **const**

Matrix<*Base*<T>> GetImagPartOfSubmatrix(**const** std::vector<int> &*rowInd*, **const** std::vector<int> &*colInd*) **const**
Return the submatrix (or its real or imaginary part) with the specified row and column indices via C++11 move semantics.

void SetSubmatrix(**const** std::vector<int> &*rowInd*, **const** std::vector<int> &*colInd*, **const** *Matrix*<T> &*ASub*)

void SetRealPartOfSubmatrix(**const** std::vector<int> &*rowInd*, **const** std::vector<int> &*colInd*, **const** *Matrix*<*Base*<T>> &*ASub*)

void SetImagPartOfSubmatrix(**const** std::vector<int> &*rowInd*, **const** std::vector<int> &*colInd*, **const** *Matrix*<*Base*<T>> &*ASub*)
Set the submatrix (or its real or imaginary part) with the specified row and column indices equal to the matrix *ASub*.

```
void UpdateSubmatrix(const std::vector<int> &rowInd, const std::vector<int>
                    &colInd, T alpha, const Matrix<T> &ASub)
```

```
void UpdateRealPartOfSubmatrix(const std::vector<int> &rowInd, const
                               std::vector<int> &colInd, Base<T> alpha,
                               const Matrix<Base<T>> &ASub)
```

```
void UpdateImagPartOfSubmatrix(const std::vector<int> &rowInd, const
                                std::vector<int> &colInd, Base<T> alpha,
                                const Matrix<Base<T>> &ASub)
```

Update the submatrix (or its real or imaginary part) with the specified row and column indices with *alpha* times *ASub*.

```
void MakeSubmatrixReal(const std::vector<int> &rowInd, const std::vector<int>
                      &colInd)
```

Force the submatrix with the specified row and column indices to be real.

```
void ConjugateSubmatrix(const std::vector<int> &rowInd, const std::vector<int>
                       &colInd)
```

Conjugate the entries in the submatrix with the specified row and column indices.

3.3.1 Special cases used in Elemental

This list of special cases is here to help clarify the notation used throughout Elemental's source (as well as this documentation). These are all special cases of *Matrix<T>*.

class Matrix<Real>

Used to denote that the underlying datatype *Real* is real.

class Matrix<Complex<Real>>

Used to denote that the underlying datatype *Complex<Real>* is complex with base type *Real*.

class Matrix<F>

Used to denote that the underlying datatype *F* is a field.

class Matrix<int>

When the underlying datatype is a signed integer.

3.4 Process grids

The *Grid* class is responsible for converting MPI communicators into a two-dimensional process grid meant for distributing matrices (ala the soon-to-be-discussed *DistMatrix<T, U, V>* class).

class Grid

```
Grid(mpi::Comm comm = mpi::COMM_WORLD, GridOrder order = COL-
     UMN_MAJOR)
```

Construct a process grid over the specified communicator and let Elemental decide the process grid dimensions. If no communicator is specified, *mpi::COMM_WORLD* is used. The processes in the chosen communicator are arranged into a two-dimensional grid in a column-major order by default but a row-major ordering can also be specified.

`Grid(mpi::Comm comm, int height, GridOrder order = COLUMN_MAJOR)`

Construct a process grid over the specified communicator with the given height. Note that the size of the communicator must be divisible by *height*. The processes in the chosen communicator are arranged into a two-dimensional grid in a column-major order by default but a row-major ordering can also be specified.

Simple interface (simpler version of distribution-based interface)

`int Row() const`

Return the index of the row of the process grid that this process lies in.

`int Col() const`

Return the index of the column of the process grid that this process lies in.

`int Rank() const`

Return our process's rank in the grid: in the case that the processes in the grid were arranged with a column-major ordering, this is `VCRank()`, otherwise, it is `VRRank()`.

`int Height() const`

Return the height of the process grid.

`int Width() const`

Return the width of the process grid.

`int Size() const`

Return the number of active processes in the process grid. This number is equal to `Height() × Width()`.

`GridOrder Order() const`

Returns whether the processes were ordered in a COLUMN_MAJOR or ROW_MAJOR manner in order to form the process grid.

`mpi::Comm ColComm() const`

Return the communicator for this process's column of the process grid.

`mpi::Comm RowComm() const`

Return the communicator for this process's row of the process grid.

`mpi::Comm Comm() const`

Return the communicator for the process grid: in the case that the processes in the grid were arranged with a column-major ordering, this is `VCComm()`, otherwise, it is `VRComm()`.

Distribution-based interface

`int MCRank() const`

Return our process's rank in the MC (Matrix Column) communicator. This corresponds to our row in the process grid.

`int MRRank() const`

Return our process's rank in the MR (Matrix Row) communicator. This corresponds to our column in the process grid.

`int VCRank() const`

Return our process's rank in the VC (Vector Column) communicator. This corresponds to our rank in a column-major ordering of the process grid.

`int VRRank() const`

Return our process's rank in the VR (Vector Row) communicator. This corresponds to our rank in a row-major ordering of the process grid.

`int MCSize() const`

Return the size of the MC (Matrix Column) communicator, which is equivalent to the height of the process grid.

`int MRSize() const`

Return the size of the MR (Matrix Row) communicator, which is equivalent to the width of the process grid.

`int VCSize() const`

Return the size of the VC (Vector Column) communicator, which is equivalent to the size of the process grid.

`int VRSize() const`

Return the size of the VR (Vector Row) communicator, which is equivalent to the size of the process grid.

`mpi::Comm MCComm() const`

Return the MC (Matrix Column) communicator. This consists of the set of processes within our column of the grid (ordered top-to-bottom).

`mpi::Comm MRComm() const`

Return the MR (Matrix Row) communicator. This consists of the set of processes within our row of the grid (ordered left-to-right).

`mpi::Comm VCComm() const`

Return the VC (Vector Column) communicator. This consists of the entire set of processes in the grid, but ordered in a column-major fashion.

`mpi::Comm VRComm() const`

Return the VR (Vector Row) communicator. This consists of the entire set of processes in the grid, but ordered in a row-major fashion.

Advanced routines

`Grid(mpi::Comm viewingComm, mpi::Group owningGroup, int height, GridOrder order = COLUMN_MAJOR)`

Construct a process grid where only a subset of the participating processes should actively participate in the process grid. In particular, *viewingComm* should consist of the set of all processes constructing this `Grid` instance, and *owningGroup* should define a subset of the processes in *viewingComm*. The height of the process grid is set to the specified value and either a column-major or row-major ordering of the participating processes is used to form the grid. Most users should not call this routine, as this type of grid is only supported for a few `DistMatrix` types. Note that the size of *owningGroup* must be divisible by *height*.

`int GCD() const`

Return the greatest common denominator of the height and width of the process grid.

- `int LCM() const`
Return the lowest common multiple of the height and width of the process grid.
- `bool HaveViewers() const`
Return true if there are processes which constructed this Grid instance but are not a member of the grid.
- `bool InGrid() const`
Return true if our process is actively participating in the process grid.
- `int OwningRank() const`
Return our process's rank within the set of processes that are actively participating in the grid.
- `int ViewingRank() const`
Return our process's rank within the entire set of processes that constructed this grid.
- `int VCToViewingMap() const`
Map the given column-major grid rank to the rank in the (potentially) larger set of processes which constructed the grid.
- `mpi::Group OwningGroup() const`
Return the group of processes which is actively participating in the grid.
- `mpi::Comm OwningComm() const`
Return the communicator for the set of processes actively participating in the grid.
Note that this can only be valid if the calling process is an active member of the grid!
- `mpi::Comm ViewingComm() const`
Return the communicator for the entire set of processes which constructed the grid.
- `int DiagPath() const`
Return our unique diagonal index in an tessellation of the process grid.
- `int DiagPath(int vectorColRank) const`
Return the unique diagonal index of the process with the given column-major vector rank in an tessellation of the process grid.
- `int DiagPathRank() const`
Return our process's rank out of the set of processes lying in our diagonal of the tessellation of the process grid.
- `int DiagPathRank(int vectorColRank) const`
Return the rank of the given process out of the set of processes in its diagonal of the tessellation of the process grid.

Grid comparison functions

- `bool operator==(const Grid &A, const Grid &B)`
Returns true if !A! and !B! are the same process grid.
- `bool operator!=(const Grid &A, const Grid &B)`
Returns true if !A! and !B! are different process grids.

3.5 Distributed matrices

The *DistMatrix* $\langle T, U, V \rangle$ class is meant to provide a distributed-memory analogue of the *Matrix* $\langle T \rangle$ class. In a manner similar to PLAPACK, roughly ten different matrix distributions are provided and it is trivial (in the sense that it requires a single command) to redistribute from one to another: in PLAPACK, one would simply call `PLA_Copy`, whereas, in Elemental, it is handled through overloading the `=` operator (or instead calling the *Copy* $()$ function).

Since it is crucial to know not only how many processes to distribute the data over, but *which* processes, and in what manner they should be decomposed into a logical two-dimensional grid, an instance of the *Grid* class must be passed into the constructor of the *DistMatrix* $\langle T, U, V \rangle$ class.

Note: Since the *DistMatrix* $\langle T, U, V \rangle$ class makes use of MPI for message passing, custom interfaces must be written for nonstandard datatypes. As of now, the following datatypes are fully supported for *DistMatrix* $\langle T, U, V \rangle$: `int`, `float`, `double`, `Complex<float>`, and `Complex<double>`.

The *DistMatrix* $\langle T, U, V \rangle$ class is implemented in a manner which attempts to expose as many symmetries within the various member functions and redistributions as possible. In particular, there is a hierarchy of three increasingly-specific distributed matrix classes: *AbstractDistMatrix* $\langle T \rangle$, which contains every member function whose prototype does not depend upon the exact matrix distribution, *GeneralDistMatrix* $\langle T, U, V \rangle$, which contains every member function whose prototype depends upon the particular matrix distribution but can still be implemented in a generic way, and *DistMatrix* $\langle T, U, V \rangle$, which contains implementations of member functions which are specific to a particular matrix distribution.

3.5.1 AbstractDistMatrix

This abstract class defines the list of member functions that are guaranteed to be available for all matrix distributions and whose prototype does not depend upon the particular matrix distribution; the *GeneralDistMatrix* $\langle T, U, V \rangle$ class exists for general routines whose prototype *does* depend upon the particular matrix distribution.

```
class AbstractDistMatrix<T>
```

Constructors and destructors

```
AbstractDistMatrix(AbstractDistMatrix<T> &&A) noexcept
```

A C++11 move constructor which transfers the metadata from the specified matrix over to the new matrix as a means of cheaply transferring resources.

```
~AbstractDistMatrix()
```

Assignment and reconfiguration

```
AbstractDistMatrix<T> &operator=(AbstractDistMatrix<T> &&A)
```

A C++11 move assignment which swaps the metadata between the two matrices as a means of cheaply swapping the resources assigned to each matrix.

void Empty()
 Empties the data and frees all alignments.

void EmptyData()
 Sets the matrix size to zero and frees associated memory (the alignments are left unchanged).

void SetGrid(const *Grid* &grid)
 Clear the distributed matrix's contents and reconfigure for the new process grid.

void Resize(int *height*, int *width*)

void Resize(int *height*, int *width*, int *ldim*)
 Reconfigure the matrix so that it is $height \times width$. Optionally, the local leading dimension may also be specified.

void MakeConsistent()
 Gives every non-participating process a copy of the metadata stored by the root process in the distribution communicator.

Realignment

void Align(int *colAlign*, int *rowAlign*)

void AlignCols(int *colAlign*)

void AlignRows(int *rowAlign*)
 Aligns the row or column distribution (or both).

void FreeAlignments()
 Free all alignment constraints.

void SetRoot(int *root*)
 For querying and changing the process rank in the cross communicator which owns the data.

void AlignWith(const *DistData* &data)

void AlignColsWith(const *DistData* &data)

void AlignRowsWith(const *DistData* &data)
 Aligns the row or column distribution (or both) as necessary to conform with the specified distribution data.

void AlignAndResize(int *colAlign*, int *rowAlign*, int *height*, int *width*, bool *force* = false)

void AlignColsAndResize(int *colAlign*, int *height*, int *width*, bool *force* = false)

void AlignRowsAndResize(int *rowAlign*, int *height*, int *width*, bool *force* = false)
 Attempt to realign the row or column distribution (or both), with the realignment being optionally *forced*, and then resize the distributed matrix to the specified size.

Buffer attachment

void Attach(int *height*, int *width*, const *Grid* &grid, int *colAlign*, int *rowAlign*, T *buffer*, int *ldim*, int *root* = 0)

`void LockedAttach(int height, int width, const Grid &grid, int colAlign, int rowAlign, const T *buffer, int ldim, int root = 0)`
Reconfigure around the (immutable) buffer of an implicit distributed matrix with the specified dimensions, alignments, process grid, and local leading dimension.

`void Attach(int height, int width, const Grid &grid, int colAlign, int rowAlign, Matrix<T> &A, int root = 0)`

`void LockedAttach(int height, int width, const Grid &grid, int colAlign, int rowAlign, const Matrix<T> &A, int root = 0)`
Reconfigure around the (immutable) local matrix of an implicit distributed matrix with the specified alignments, process grid, and local leading dimension.

Basic queries

`int Height() const`

`int Width() const`

Return the height (width) of the distributed matrix.

`int DiagonalLength(int offset = 0) const`

Return the length of the specified diagonal of the distributed matrix.

`bool Viewing() const`

Return true if this matrix is viewing another.

`bool Locked() const`

Return true if this matrix is viewing another in a manner that does not allow for modifying the viewed data.

`int LocalHeight() const`

`int LocalWidth() const`

Return the height (width) of the local matrix stored by a particular process.

`int LDim() const`

Return the leading dimension of the local matrix stored by a particular process.

`Matrix<T> &Matrix()`

`const Matrix<T> &LockedMatrix() const`

Return an (immutable) reference to the local matrix.

`size_t AllocatedMemory() const`

Return the number of entries of type *T* that we have locally allocated space for.

`T *Buffer()`

`const T *LockedBuffer() const`

Return an (immutable) pointer to the local matrix's buffer.

`T *Buffer(int iLoc, int jLoc)`

`const T *LockedBuffer(int iLoc, int jLoc) const`

Return an (immutable) pointer to the portion of the local buffer that stores entry (*iLoc*, *jLoc*).

Distribution information**const** *Grid* &Grid() **const**

Return the grid that this distributed matrix is distributed over.

bool ColConstrained() **const**bool RowConstrained() **const**

Return true if the column (row) alignment is constrained.

bool RootConstrained() **const**Return if the root (the alignment with respect to the *CrossComm()*) is constrained.int ColAlign() **const**int RowAlign() **const**Return the rank of the member of our *ColComm()* or *RowComm()* assigned to the top-left entry of the matrix.int ColShift() **const**int RowShift() **const**

Return the first row or column to be locally assigned to this process, respectively.

mpi::Comm ColComm() **const**

The communicator used to distribute each column of the matrix.

mpi::Comm RowComm() **const**

The communicator used to distribute each row of the matrix.

mpi::Comm PartialColComm() **const**mpi::Comm PartialUnionColComm() **const**

The *PartialColComm()* is a (not necessarily strict) subset of the *ColComm()*; an element-wise distribution of each column over this communicator can be reached by unioning the local data from a distribution over the *ColComm()* (via an AllGather) over the *PartialUnionColComm()*. One nontrivial example is for *DistMatrix<T, VC, STAR>*, where the column communicator is *Grid::VCComm()*, the partial column communicator is *Grid::MCComm()*, and the partial union column communicator is *Grid::MRComm()*.

mpi::Comm PartialRowComm() **const**mpi::Comm PartialUnionRowComm() **const**

These are the same as *PartialColComm()* and *PartialUnionColComm()*, except that they correspond to distributions of the rows of the matrix.

mpi::Comm DistComm() **const**

The communicator used to distribute the entire set of entries of the matrix (in a particular precise sense, the product of *ColComm()* and *RowComm()*).

mpi::Comm CrossComm() **const**

The orthogonal complement of the product of *DistComm()* and *RedundantComm()* with respect to the process grid. For instance, for *DistMatrix<T, CIRC, CIRC>*, this is *Grid::VCComm()*.

mpi::Comm RedundantComm() **const**

The communicator over which data is redundantly stored. For instance, for *DistMatrix<T, MC, STAR>*, this is *Grid::RowComm()*.

int ColRank() **const**

int RowRank() **const**

int PartialColRank() **const**

int PartialRowRank() **const**

int PartialUnionColRank() **const**

int PartialUnionRowRank() **const**

int DistRank() **const**

int CrossRank() **const**

int RedundantRank() **const**

Return our rank in our *ColComm()*, *RowComm()*, *PartialColComm()*, *PartialRowComm()*, *PartialUnionColComm()*, *PartialUnionRowComm()*, *DistComm()*, *CrossComm()*, or *RedundantComm()*, respectively.

int ColStride() **const**

int RowStride() **const**

int PartialColStride() **const**

int PartialRowStride() **const**

int PartialUnionColStride() **const**

int PartialUnionRowStride() **const**

int DistSize() **const**

int CrossSize() **const**

int RedundantSize() **const**

Return the number of processes within a particular communicator associated with the distributed matrix. For communicators associated with distributions of either the rows or columns of a matrix, the communicator size is equal to the distance (or *stride*) between successive row or column indices assigned to a particular process.

int Root() **const**

Return the rank of the member of our cross communicator (*CrossComm()*) which can store data.

bool Participating() **const**

Return true if this process can be assigned matrix data (that is, if this process is both in the process grid and the root of *CrossComm()*).

int RowOwner(int *i*) **const**

Return the rank (in *ColComm()*) of the process which owns row *i*.

int ColOwner(int *j*) **const**

Return the rank (in *RowComm()*) of the process which owns column *j*.

int Owner(int *i*, int *j*) **const**

Return the rank (in *DistComm()*) of the process which owns entry (*i,j*).

int GlobalRow(int *iLoc*) **const**

`int GlobalCol(int jLoc) const`
 Return the global row (column) index corresponding to the given local row (column) index.

`int LocalRow(int i) const`

`int LocalCol(int j) const`
 Return the local row (column) index for row *i* (*j*); if this process is not assigned row *i* (column *j*), then throw an exception.

`int LocalRowOffset(int i) const`

`int LocalColOffset(int j) const`
 Return the number of local rows (columns) occurring before the given global row (column) index.

`bool IsLocalRow(int i) const`

`bool IsLocalCol(int j) const`

`bool IsLocal(int i, int j) const`
 Return true if the row, column, or entry, respectively, is assigned to this process.

`DistData DistData() const`
 Returns a description of the distribution and alignment information

Single-entry manipulation (global)

`T Get(int i, int j) const`

`Base<T> GetRealPart(int i, int j) const`

`Base<T> GetImagPart(int i, int j) const`
 Return the (*i,j*) entry (or its real or imaginary part) of the global matrix.

`void Set(int i, int j, T alpha)`

`void SetRealPart(int i, int j, Base<T> alpha)`

`void SetImagPart(int i, int j, Base<T> alpha)`
 Set the (*i,j*) entry (or its real or imaginary part) of the global matrix to α .

`void Update(int i, int j, T alpha)`

`void UpdateRealPart(int i, int j, Base<T> alpha)`

`void UpdateImagPart(int i, int j, Base<T> alpha)`
 Add α to the (*i,j*) entry (or its real or imaginary part) of the global matrix.

`void MakeReal(int i, int j)`
 Force the (*i,j*) entry of the global matrix to be real.

`void Conjugate(int i, int j)`
 Conjugate the (*i,j*) entry of the global matrix.

Single-entry manipulation (local)

`T GetLocal(int iLoc, int jLoc) const`

Base<T> GetRealPartLocal (int *iLoc*, int *jLoc*) **const**
Base<T> GetLocalImagPart (int *iLoc*, int *jLoc*) **const**
Return the (*iLoc*, *jLoc*) entry (or its real or imaginary part) of our local matrix.

void SetLocal (int *iLoc*, int *jLoc*, T *alpha*)
void SetLocalRealPart (int *iLoc*, int *jLoc*, *Base*<T> *alpha*)
void SetLocalImagPart (int *iLoc*, int *jLoc*, *Base*<T> *alpha*)
Set the (*iLoc*, *jLoc*) entry (or its real or imaginary part) of our local matrix to α .

void UpdateLocal (int *iLoc*, int *jLoc*, T *alpha*)
void UpdateRealPartLocal (int *iLoc*, int *jLoc*, *Base*<T> *alpha*)
void UpdateLocalImagPart (int *iLoc*, int *jLoc*, *Base*<T> *alpha*)
Add α to the (*iLoc*, *jLoc*) entry (or its real or imaginary part) of our local matrix.

void MakeLocalReal (int *iLoc*, int *jLoc*)
Force the (*iLoc*, *jLoc*) entry of our local matrix to be real.

void ConjugateLocal (int *iLoc*, int *jLoc*)
Conjugate the (*iLoc*, *jLoc*) entry of our local matrix.

Diagonal manipulation

void MakeDiagonalReal (int *offset* = 0)
Force the specified diagonal to be real.

void ConjugateDiagonal (int *offset* = 0)
Conjugate the specified diagonal.

Arbitrary-submatrix manipulation (global)

void GetSubmatrix(**const** std::vector<int> &*rowInd*, **const** std::vector<int> &*colInd*, *DistMatrix*<T, *STAR*, *STAR*> &*ASub*) **const**

void GetRealPartOfSubmatrix(**const** std::vector<int> &*rowInd*, **const** std::vector<int> &*colInd*, *DistMatrix*<*Base*<T>, *STAR*, *STAR*> &*ASub*) **const**

void GetImagPartOfSubmatrix(**const** std::vector<int> &*rowInd*, **const** std::vector<int> &*colInd*, *DistMatrix*<*Base*<T>, *STAR*, *STAR*> &*ASub*) **const**
Return the submatrix (or its real or imaginary part) with the specified row and column indices via *ASub*.

DistMatrix<T, *STAR*, *STAR*> GetSubmatrix(**const** std::vector<int> &*rowInd*, **const** std::vector<int> &*colInd*) **const**

DistMatrix<*Base*<T>, *STAR*, *STAR*> GetRealPartOfSubmatrix(**const** std::vector<int> &*rowInd*, **const** std::vector<int> &*colInd*) **const**

DistMatrix<*Base*<*T*>, *STAR*, *STAR*> GetImagPartOfSubmatrix(const
 std::vector<int>
 &rowInd, const
 std::vector<int>
 &colInd) const

Return the submatrix (or its real or imaginary part) with the specified row and column indices via C++11 move semantics.

void SetSubmatrix(const std::vector<int> &rowInd, const std::vector<int> &colInd, const *DistMatrix*<*T*, *STAR*, *STAR*> &ASub)

void SetRealPartOfSubmatrix(const std::vector<int> &rowInd, const std::vector<int> &colInd, const *DistMatrix*<*Base*<*T*>, *STAR*, *STAR*> &ASub)

void SetImagPartOfSubmatrix(const std::vector<int> &rowInd, const std::vector<int> &colInd, const *DistMatrix*<*Base*<*T*>, *STAR*, *STAR*> &ASub)

Set the submatrix (or its real or imaginary part) with the specified row and column indices equal to the matrix *ASub*.

void UpdateSubmatrix(const std::vector<int> &rowInd, const std::vector<int> &colInd, *T* alpha, const *DistMatrix*<*T*, *STAR*, *STAR*> &ASub)

void UpdateRealPartOfSubmatrix(const std::vector<int> &rowInd, const std::vector<int> &colInd, *Base*<*T*> alpha, const *DistMatrix*<*Base*<*T*>, *STAR*, *STAR*> &ASub)

void UpdateImagPartOfSubmatrix(const std::vector<int> &rowInd, const std::vector<int> &colInd, *Base*<*T*> alpha, const *DistMatrix*<*Base*<*T*>, *STAR*, *STAR*> &ASub)

Update the submatrix (or its real or imaginary part) with the specified row and column indices with *alpha* times *ASub*.

void MakeSubmatrixReal(const std::vector<int> &rowInd, const std::vector<int> &colInd)

Force the submatrix with the specified row and column indices to be real.

void ConjugateSubmatrix(const std::vector<int> &rowInd, const std::vector<int> &colInd)

Conjugate the entries in the submatrix with the specified row and column indices.

Arbitrary-submatrix manipulation (local)

void GetLocalSubmatrix(const std::vector<int> &rowIndLoc, const std::vector<int> &colIndLoc, *Matrix*<*T*> &ASub) const

void GetRealPartOfLocalSubmatrix(const std::vector<int> &rowIndLoc, const std::vector<int> &colIndLoc, *Matrix*<*Base*<*T*>> &ASub) const

void GetImagPartOfLocalSubmatrix(const std::vector<int> &rowIndLoc, const std::vector<int> &colIndLoc, *Matrix*<*Base*<*T*>> &ASub) const

Return the local submatrix (or its real or imaginary part) with the specified row and

column indices via *ASub*.

```
Matrix<T> GetLocalSubmatrix(const std::vector<int> &rowIndLoc, const
                           std::vector<int> &colIndLoc) const
```

```
Matrix<Base<T>> GetRealPartOfLocalSubmatrix(const std::vector<int>
                                             &rowIndLoc, const
                                             std::vector<int> &colInd-
                                             Loc) const
```

```
Matrix<Base<T>> GetImagPartOfLocalSubmatrix(const std::vector<int>
                                             &rowIndLoc, const
                                             std::vector<int> &colInd-
                                             Loc) const
```

Return the local submatrix (or its real or imaginary part) with the specified row and column indices via C++11 move semantics.

```
void SetLocalSubmatrix(const std::vector<int> &rowIndLoc, const
                      std::vector<int> &colIndLoc, const Matrix<T> &ASub)
```

```
void SetRealPartOfLocalSubmatrix(const std::vector<int> &rowIndLoc, const
                                  std::vector<int> &colIndLoc, const Ma-
                                  trix<Base<T>> &ASub)
```

```
void SetImagPartOfLocalSubmatrix(const std::vector<int> &rowIndLoc, const
                                  std::vector<int> &colIndLoc, const Ma-
                                  trix<Base<T>> &ASub)
```

Set the local submatrix (or its real or imaginary part) with the specified row and column indices equal to the matrix *ASub*.

```
void UpdateLocalSubmatrix(const std::vector<int> &rowIndLoc, const
                          std::vector<int> &colIndLoc, T alpha, const Ma-
                          trix<T> &ASub)
```

```
void UpdateRealPartOfLocalSubmatrix(const std::vector<int> &rowIndLoc, const
                                     std::vector<int> &colIndLoc, Base<T> al-
                                     pha, const Matrix<Base<T>> &ASub)
```

```
void UpdateImagPartOfLocalSubmatrix(const std::vector<int> &rowIndLoc, const
                                     std::vector<int> &colIndLoc, Base<T> al-
                                     pha, const Matrix<Base<T>> &ASub)
```

Update the local submatrix (or its real or imaginary part) with the specified row and column indices with *alpha* times *ASub*.

```
void MakeLocalSubmatrixReal(const std::vector<int> &rowIndLoc, const
                             std::vector<int> &colIndLoc)
```

Force the local submatrix with the specified row and column indices to be real.

```
void ConjugateLocalSubmatrix(const std::vector<int> &rowIndLoc, const
                              std::vector<int> &colIndLoc)
```

Conjugate the entries in the local submatrix with the specified row and column indices.

Sum over a specified communicator

```
void SumOver(mpi::Comm comm)
```

Assertions

void ComplainIfReal() **const**

void AssertNotLocked() **const**

void AssertNotStoringData() **const**

void AssertValidEntry(int *i*, int *j*) **const**

void AssertValidSubmatrix(int *i*, int *j*, int *height*, int *width*) **const**

void AssertSameGrid(**const** *Grid* &*grid*) **const**

void AssertSameSize(int *height*, int *width*) **const**

class DistData

Distribution colDist

The *Distribution* scheme used within each column of the matrix.

Distribution rowDist

The *Distribution* scheme used within each row of the matrix.

int colAlign

The rank in the *AbstractDistMatrix*<*T*>::*ColComm*() which is assigned the top-left entry of the matrix.

int rowAlign

The rank in the *AbstractDistMatrix*<*T*>::*RowComm*() which is assigned the top-left entry of the matrix.

int root

The member of the *AbstractDistMatrix*<*T*>::*CrossComm*() which is assigned ownership of the matrix.

const *Grid* **grid*

An immutable pointer to the underlying process grid of the distributed matrix.

DistData(**const** *GeneralDistMatrix*<*T*, *U*, *V*> &*A*)

Construct the distribution data of any instance of *GeneralDistMatrix*<*T*, *U*, *V*>.

3.5.2 GeneralDistMatrix

The *GeneralDistMatrix*<*T*, *U*, *V*> class, which inherits from *AbstractDistMatrix*<*T*>, provides a mechanism for defining member functions of the *DistMatrix*<*T*, *U*, *V*> class which can be implemented in a templated way over the matrix distribution. Since a small number of member functions of *DistMatrix*<*T*, *U*, *V*> must be specialized for each particular distribution (choice of the pair *U* and *V*), a parent class is required in order to support both general and specialized implementations of member functions whose prototype depends upon the matrix distribution.

class GeneralDistMatrix<*T*, *U*, *V*>

Related row and column distributions

Distribution UDiag

Distribution VDiag

The column and row *Distribution*'s of the diagonal of the matrix if it is stored as a column vector.

Distribution UGath

The resulting *Distribution* from unioning distribution U over *AbstractDistMatrix* $\langle T \rangle::ColComm()$. For most matrix distributions this will be STAR, but for *GeneralDistMatrix* $\langle T, CIRC, CIRC \rangle$ it will be CIRC.

Distribution VGath

The resulting *Distribution* from unioning distribution V over *AbstractDistMatrix* $\langle T \rangle::RowComm()$.

Distribution UPart

Distribution VPart

The resulting *Distribution* from unioning distribution U over *AbstractDistMatrix* $\langle T \rangle::PartialUnionColComm()$, or V over *AbstractDistMatrix* $\langle T \rangle::PartialUnionRowComm()$, respectively.

Distribution UScat

Distribution VScat

The result of scattering *Distribution* U over *AbstractDistMatrix* $\langle T \rangle::PartialUnionRowComm()$, or V over *AbstractDistMatrix* $\langle T \rangle::PartialUnionColComm()$, respectively.

Constructors and destructors

GeneralDistMatrix(*GeneralDistMatrix* $\langle T, U, V \rangle$ &&A) **noexcept**

A C++11 move constructor which cheaply transfers resources from A to the new matrix by swapping metadata.

Assignment and reconfiguration

GeneralDistMatrix $\langle T, U, V \rangle$ &operator=(*GeneralDistMatrix* $\langle T, U, V \rangle$ &&A)

A C++11 move assignment which cheaply transfers resources from A to this matrix by swapping metadata.

Note: The following routines are for advanced users/developers and can be safely ignored by most users.

void Translate(*DistMatrix* $\langle T, U, V \rangle$ &A) **const**

Handles redistributing between matrices with the same distribution and grid, but possibly different alignments.

void AllGather(*DistMatrix* $\langle T, UGath, VGath \rangle$ &A) **const**

void ColAllGather(*DistMatrix* $\langle T, UGath, V \rangle$ &A) **const**

```
void RowAllGather(DistMatrix<T, U, VGath> &A) const
```

```
void PartialColAllGather(DistMatrix<T, UPart, V> &A) const
```

```
void PartialRowAllGather(DistMatrix<T, U, VPart> &A) const
```

Form the distributed matrix which results from unioning the local data over *AbstractDistMatrix*<T>::*DistComm*(), *AbstractDistMatrix*<T>::*ColComm*(), *AbstractDistMatrix*<T>::*RowComm*(), *AbstractDistMatrix*<T>::*PartialUnionColComm*(), or *AbstractDistMatrix*<T>::*PartialUnionRowComm*(), respectively. Each of these is accomplished via a call to *mpi::AllGather*() over the appropriate communicator.

Note: *ColAllGather*() and (to a lesser degree) *PartialColAllGather*() both require accessing large amounts of data with a non-uniform stride. They therefore do not make efficient usage of cache lines and should be avoided in favor of *TransposeColAllGather*() and *TransposePartialColAllGather*(), respectively.

```
void FilterFrom(const DistMatrix<T, UGath, VGath> &A)
```

```
void ColFilterFrom(const DistMatrix<T, UGath, V> &A)
```

```
void RowFilterFrom(const DistMatrix<T, U, VGath> &A)
```

```
void PartialColFilterFrom(const DistMatrix<T, UPart, V> &A)
```

```
void PartialRowFilterFrom(const DistMatrix<T, U, VPart> &A)
```

Set the current matrix equal to the appropriate subset of a distributed matrix which would have resulted from unioning our local data over a particular communicator.

```
void PartialColAllToAllFrom(const DistMatrix<T, UPart, VScat> &A)
```

```
void PartialRowAllToAllFrom(const DistMatrix<T, UScat, VPart> &A)
```

Set this matrix to the result of scattering columns (rows) and unioning rows (columns) of *A* over *AbstractDistMatrix*<T>::*PartialUnionColComm*() (*AbstractDistMatrix*<T>::*PartialUnionRowComm*()).

```
void PartialColAllToAll(DistMatrix<T, UPart, VScat> &A) const
```

```
void PartialRowAllToAll(DistMatrix<T, UScat, VPart> &A) const
```

Set *A* to the result of unioning columns (rows) and scattering rows (columns) of this matrix over *AbstractDistMatrix*<T>::*PartialUnionColComm*() (*AbstractDistMatrix*<T>::*PartialUnionRowComm*()).

```
void SumScatterFrom(const DistMatrix<T, UGath, VGath> &A)
```

```
void RowSumScatterFrom(const DistMatrix<T, U, VGath> &A)
```

```
void ColSumScatterFrom(const DistMatrix<T, UGath, V> &A)
```

```
void PartialRowSumScatterFrom(const DistMatrix<T, U, VPart> &A)
```

```
void PartialColSumScatterFrom(const DistMatrix<T, UPart, V> &A)
```

Simultaneously sum and scatter each process's local matrix from *A* over the *AbstractDistMatrix*<T>::*DistComm*(), *AbstractDistMatrix*<T>::*RowComm*(), *AbstractDistMatrix*<T>::*ColComm*(), *AbstractDistMatrix*<T>::*PartialRowComm*(), or *AbstractDistMatrix*<T>::*PartialColComm*() communicator, respectively, and set the current matrix equal to the result.

Note: *ColSumScatterFrom()* and (to a lesser degree) *PartialColSumScatterFrom()* both require accessing large amounts of data with a non-uniform stride. They therefore do not make efficient usage of cache lines and should be avoided in favor of *TransposeColSumScatterFrom()* and *TransposePartialColSumScatterFrom()*, respectively.

```
void SumScatterUpdate(T alpha, const DistMatrix<T, UGath, VGath> &A)
```

```
void RowSumScatterUpdate(T alpha, const DistMatrix<T, U, VGath> &A)
```

```
void ColSumScatterUpdate(T alpha, const DistMatrix<T, UGath, V> &A)
```

```
void PartialRowSumScatterUpdate(T alpha, const DistMatrix<T, U, VPart> &A)
```

```
void PartialColSumScatterUpdate(T alpha, const DistMatrix<T, UPart, V> &A)
```

Simultaneously sum and scatter each process's local matrix from A over the *AbstractDistMatrix<T>::DistComm()*, *AbstractDistMatrix<T>::RowComm()*, *AbstractDistMatrix<T>::ColComm()*, *AbstractDistMatrix<T>::PartialRowComm()*, or *AbstractDistMatrix<T>::PartialColComm()* communicator, respectively, and add α times the result to the current matrix.

Note: *ColSumScatterUpdate()* and (to a lesser degree) *PartialColSumScatterUpdate()* both require accessing large amounts of data with a non-uniform stride. They therefore do not make efficient usage of cache lines and should be avoided in favor of *TransposeColSumScatterUpdate()* and *TransposePartialColSumScatterUpdate()*, respectively.

Transpose redistributions

```
void TransposeColAllGather(DistMatrix<T, V, UGath> &A, bool conjugate = false)
    const
```

```
void TransposePartialColAllGather(DistMatrix<T, V, UPart> &A, bool conjugate
    = false) const
```

Since *ColAllGather()* and *PartialColAllGather()* make poor usage of cache lines due to filling up columns of the result in an interleaved manner, forming the transposed result, by transposing the data before calling *mpi::AllGather()*, allows for the expensive unpacking step to copy entire contiguous rows of the input at a time. The optional *conjugate* parameter determines whether or not the result should be conjugated in addition to transposed.

```
void AdjointColAllGather(DistMatrix<T, V, UGath> &A) const
```

```
void AdjointPartialColAllGather(DistMatrix<T, V, UPart> &A) const
```

These routines correspond to *TransposeColAllGather()* and *TransposePartialColAllGather()* with conjugation.

```
void TransposeColFilterFrom(const DistMatrix<T, V, UGath> &A, bool conjugate
    = false)
```

```
void TransposeRowFilterFrom(const DistMatrix<T, VGath, U> &A, bool conjugate
    = false)
```

```
void TransposePartialColFilterFrom(const DistMatrix<T, V, UPart> &A, bool
    conjugate = false)
```

```
void TransposePartialRowFilterFrom(const DistMatrix<T, VPart, U> &A, bool
                                conjugate = false)
```

After performing computations with the results of *TransposeColAllGather()* or *TransposePartialColAllGather()*, it is frequently necessary to form a subset of the transposed result. The result can be optionally conjugated.

```
void AdjointColFilterFrom(const DistMatrix<T, V, UGath> &A)
```

```
void AdjointRowFilterFrom(const DistMatrix<T, VGath, U> &A)
```

```
void AdjointPartialColFilterFrom(const DistMatrix<T, V, UPart> &A)
```

```
void AdjointPartialRowFilterFrom(const DistMatrix<T, VPart, U> &A)
```

These routines correspond to the conjugated versions of *TransposeColFilterFrom()*, *TransposeRowFilterFrom()*, *TransposePartialColFilterFrom()*, and *TransposePartialRowFilterFrom()*.

```
void TransposeColSumScatterFrom(const DistMatrix<T, V, UGath> &A, bool con-
                                jugate = false)
```

```
void TransposePartialColSumScatterFrom(const DistMatrix<T, V, UPart> &A,
                                       bool conjugate = false)
```

Since *ColSumScatterFrom()* and (to a lesser degree) *PartialColSumScatterFrom()* involve accessing large amounts of data with a non-uniform stride, these routines work with the (conjugate-)transposed input in order to form the result in a more efficient manner which primarily moved contiguous chunks of data.

```
void AdjointColSumScatterFrom(const DistMatrix<T, V, UGath> &A)
```

```
void AdjointPartialColSumScatterFrom(const DistMatrix<T, V, UPart> &A)
```

These routines are the conjugated versions of *TransposeColSumScatterFrom()* and *TransposePartialColSumScatterFrom()*.

```
void TransposeColSumScatterUpdate(T alpha, const DistMatrix<T, V, UGath> &A,
                                   bool conjugate = false)
```

```
void TransposePartialColSumScatterUpdate(T alpha, const DistMatrix<T, V, UP-
                                       art> &A, bool conjugate = false)
```

Since *ColSumScatterUpdate()* and (to a lesser degree) *PartialColSumScatterUpdate()* involve accessing large amounts of data with a non-uniform stride, these routines work with the (conjugate-)transposed input in order to form the result in a more efficient manner which primarily moved contiguous chunks of data.

```
void AdjointColSumScatterUpdate(T alpha, const DistMatrix<T, V, UGath> &A)
```

```
void AdjointPartialColSumScatterUpdate(T alpha, const DistMatrix<T, V, UP-
                                       art> &A)
```

These routines are the conjugated versions of *TransposeColSumScatterUpdate()* and *TransposePartialColSumScatterUpdate()*.

3.5.3 DistMatrix

The *DistMatrix<T,U,V>* class, which is the final product from the succession from *AbstractDistMatrix<T>* and *GeneralDistMatrix<T,U,V>*, is specialized for each of the

thirteen different legal distribution pairs. Each specialization involves choosing a sensical pairing of distributions for the rows and columns of the matrix:

- *CIRC* : Only give the data to a single process
- *STAR* : Give the data to every process
- *MC* : Distribute round-robin within each column of the 2D process grid (*M* atrix *C* olumn)
- *MR*: Distribute round-robin within each row of the 2D process grid (*M* atrix *R* ow)
- *VC*: Distribute round-robin within a column-major ordering of the entire 2D process grid (*V* ector *C* olumn)
- *VR*: Distribute round-robin within a row-major ordering of the entire 2D process grid (*V* ector *R* ow)
- *MD*: Distribute round-robin over a diagonal of the tiling of the 2D process grid (*M* atrix *D* iagonal)

The valid pairings are:

ColDist	RowDist	ColComm	RowComm	DistComm	RedundantComm	Cross-Comm
CIRC	CIRC	self	self	self	self	VC
MC	MR	MC	MR	VC	self	self
MC	STAR	MC	self	MC	MR	self
MD	STAR	MD	self	MD	self	MDPerp
MR	MC	MR	MC	VR	self	self
MR	STAR	MR	self	MR	MC	self
STAR	MC	self	MC	MC	MR	self
STAR	MD	self	MD	MD	self	MDPerp
STAR	MR	self	MR	MR	MC	self
STAR	STAR	self	self	self	VC	self
STAR	VC	self	VC	VC	self	self
STAR	VR	self	VR	VR	self	self
VC	STAR	VC	self	VC	self	self
VR	STAR	VR	self	VR	self	self

where *DistComm* refers to the communicator that the entire matrix (rather than just the rows or columns) is distributed over. When the matrix is distributed over a communicator which only involves only a subset of the processes, it is possible to either assign the data to just that subset or redundantly store the entire matrix on each such subset of processes (e.g., within each row of a 2D arrangement of the set of processes). The *RedundantComm* refers to the communicator where each member process stores the same information, and the *CrossComm* is the communicator where only a single process (the *root*) is assigned any data.

To make this discussion more precise, each valid matrix distribution for *DistMatrix* logically arranges the set of p processes of the r by c process grid into a 4D mesh: $ColComm \times RowComm \times RedundantComm \times CrossComm$, where *DistComm* is equal to $ColComm \times RowComm$.

class *DistMatrix*<T, U, V>

The following routines are available for each legal pairing of row and column distributions.

Constructors and destructors

`DistMatrix(const Grid &grid = DefaultGrid(), int root = 0)`

Construct an empty (0×0) distributed matrix.

`DistMatrix(int height, int width, const Grid &grid = DefaultGrid(), int root = 0)`

Create a $height \times width$ distributed matrix.

`DistMatrix(const DistMatrix<T, Y, Z> &A)`

Construct the current matrix to be a redistributed copy of the input matrix.

`DistMatrix(const DistMatrix<T, U, V> &&A) noexcept`

Use C++11 move semantics to construct the current matrix in a way which transfers the resources from the input matrix.

`~DistMatrix()`

All resources owned by the *DistMatrix* are freed upon destruction.

Assignment and reconfiguration

`DistMatrix<T, U, V> &operator=(const DistMatrix<T, Y, Z> &A)`

Set the current distributed matrix equal to the matrix *A* redistributed into the appropriate form.

`DistMatrix<T, U, V> &operator=(DistMatrix<T, U, V> &&A)`

A C++11 move assignment which cheaply transfers the resources from *A* to the current matrix by swapping metadata.

The standard matrix distribution ([MC,MR])

This is by far the most important matrix distribution in Elemental, as the vast majority of parallel routines expect the input to be in this form. For a 7×7 matrix distributed over a 2×3 process grid, individual entries would be owned by the following processes (assuming the column and row alignments are both 0):

$$\begin{pmatrix} 0 & 2 & 4 & 0 & 2 & 4 & 0 \\ 1 & 3 & 5 & 1 & 3 & 5 & 1 \\ 0 & 2 & 4 & 0 & 2 & 4 & 0 \\ 1 & 3 & 5 & 1 & 3 & 5 & 1 \\ 0 & 2 & 4 & 0 & 2 & 4 & 0 \\ 1 & 3 & 5 & 1 & 3 & 5 & 1 \\ 0 & 2 & 4 & 0 & 2 & 4 & 0 \end{pmatrix}$$

Similarly, if the column alignment is kept at 0 and the row alignment is changed to 2 (meaning that the third process column owns the first column of the matrix), the individual entries would be owned as follows:

$$\begin{pmatrix} 4 & 0 & 2 & 4 & 0 & 2 & 4 \\ 5 & 1 & 3 & 5 & 1 & 3 & 5 \\ 4 & 0 & 2 & 4 & 0 & 2 & 4 \\ 5 & 1 & 3 & 5 & 1 & 3 & 5 \\ 4 & 0 & 2 & 4 & 0 & 2 & 4 \\ 5 & 1 & 3 & 5 & 1 & 3 & 5 \\ 4 & 0 & 2 & 4 & 0 & 2 & 4 \end{pmatrix}$$

It should also be noted that this is the default distribution format for the *DistMatrix*<*T*, *U*, *V*> class, as *DistMatrix*<*T*> defaults to *DistMatrix*<*T*, *MC*, *MR*>.

class *DistMatrix*<*T*>

class *DistMatrix*<*T*, *MC*, *MR*>

All public member functions have been described as part of *AbstractDistMatrix*<*T*>, *GeneralDistMatrix*<*T*, *U*, *V*>, and *DistMatrix*<*T*, *U*, *V*>.

[*MC*, *STAR*]

This distribution is often used as part of matrix-matrix multiplication. For a 7×7 matrix distributed over a 2×3 process grid, individual entries would be owned by the following processes (assuming the column alignment is 0):

$$\begin{pmatrix} \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} \\ \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} \\ \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} \\ \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} \\ \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} \\ \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} \\ \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} \end{pmatrix}$$

class *DistMatrix*<*T*, *MC*, *STAR*>

All public member functions have been described as part of *AbstractDistMatrix*<*T*>, *GeneralDistMatrix*<*T*, *U*, *V*>, and *DistMatrix*<*T*, *U*, *V*>.

[*STAR*, *MR*]

This distribution is also frequently used for matrix-matrix multiplication. For a 7×7 matrix distributed over a 2×3 process grid, individual entries would be owned by the following processes (assuming the row alignment is 0):

$$\begin{pmatrix} \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} \\ \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} \\ \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} \\ \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} \\ \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} \\ \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} \\ \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} \end{pmatrix}$$

class *DistMatrix*<*T*, *STAR*, *MR*>

All public member functions have been described as part of *AbstractDistMatrix*<*T*>, *GeneralDistMatrix*<*T*, *U*, *V*>, and *DistMatrix*<*T*, *U*, *V*>.

[*MR*, *MC*]

This is essentially the transpose of the standard matrix distribution, [*MC*, *MR*]. For a 7×7 matrix distributed over a 2×3 process grid, individual entries would be owned by the following

processes (assuming the column and row alignments are both 0):

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 2 & 3 & 2 & 3 & 2 & 3 & 2 \\ 4 & 5 & 4 & 5 & 4 & 5 & 4 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 2 & 3 & 2 & 3 & 2 & 3 & 2 \\ 4 & 5 & 4 & 5 & 4 & 5 & 4 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

class `DistMatrix<T, MR, MC>`

All public member functions have been described as part of *AbstractDistMatrix<T>*, *GeneralDistMatrix<T,U,V>*, and *DistMatrix<T,U,V>*.

[MR, STAR]

This is the transpose of the [`*`, MR] distribution and is, like many of the previous distributions, useful for matrix-matrix multiplication. For a 7×7 matrix distributed over a 2×3 process grid, individual entries would be owned by the following processes (assuming the column alignment is 0):

$$\begin{pmatrix} \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} \\ \{2,3\} & \{2,3\} & \{2,3\} & \{2,3\} & \{2,3\} & \{2,3\} & \{2,3\} \\ \{4,5\} & \{4,5\} & \{4,5\} & \{4,5\} & \{4,5\} & \{4,5\} & \{4,5\} \\ \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} \\ \{2,3\} & \{2,3\} & \{2,3\} & \{2,3\} & \{2,3\} & \{2,3\} & \{2,3\} \\ \{4,5\} & \{4,5\} & \{4,5\} & \{4,5\} & \{4,5\} & \{4,5\} & \{4,5\} \\ \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} \end{pmatrix}$$

class `DistMatrix<T, MR, STAR>`

All public member functions have been described as part of *AbstractDistMatrix<T>*, *GeneralDistMatrix<T,U,V>*, and *DistMatrix<T,U,V>*.

[STAR, MC]

This is the transpose of the [MC, `*`] distribution and is, like many of the previous distributions, useful for matrix-matrix multiplication. For a 7×7 matrix distributed over a 2×3 process grid, individual entries would be owned by the following processes (assuming the column alignment is 0):

$$\begin{pmatrix} \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} \\ \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} \\ \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} \\ \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} \\ \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} \\ \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} \\ \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} \end{pmatrix}$$

class `DistMatrix<T, STAR, MC>`

All public member functions have been described as part of *AbstractDistMatrix<T>*, *GeneralDistMatrix<T,U,V>*, and *DistMatrix<T,U,V>*.

[MD, STAR]

In the case of our 2×3 process grid, each diagonal of the tessellation of the process grid will contain the entire set of processes, for instance, in the order 0,3,4,1,2,5. This would result in the following overlay for the owning processes of the entries of our 7×7 matrix example:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Notice that each column of this matrix is distributed like a diagonal of a [MC, MR] distribution.

class `DistMatrix<T, MD, STAR>`

All public member functions have been described as part of *AbstractDistMatrix<T>*, *GeneralDistMatrix<T, U, V>*, and *DistMatrix<T, U, V>*.

[STAR, MD]

In the case of our 2×3 process grid, each diagonal of the tessellation of the process grid will contain the entire set of processes, for instance, in the order 0,3,4,1,2,5. This would result in the following overlay for the owning processes of the entries of our 7×7 matrix example:

$$\begin{pmatrix} 0 & 3 & 4 & 1 & 2 & 5 & 0 \\ 0 & 3 & 4 & 1 & 2 & 5 & 0 \\ 0 & 3 & 4 & 1 & 2 & 5 & 0 \\ 0 & 3 & 4 & 1 & 2 & 5 & 0 \\ 0 & 3 & 4 & 1 & 2 & 5 & 0 \\ 0 & 3 & 4 & 1 & 2 & 5 & 0 \\ 0 & 3 & 4 & 1 & 2 & 5 & 0 \end{pmatrix}$$

Notice that each *row* of this matrix is distributed like a diagonal of a [MC, MR] distribution.

class `DistMatrix<T, STAR, MD>`

All public member functions have been described as part of *AbstractDistMatrix<T>*, *GeneralDistMatrix<T, U, V>*, and *DistMatrix<T, U, V>*.

[VC, STAR]

This distribution makes use of a 1d distribution which uses a column-major ordering of the entire process grid. Since 1d distributions are useful for distributing *vectors*, and a *column-major* ordering is used, the distribution symbol is VC. Again using the simple 2×3 process grid, with a zero column alignment, each entry of a 7×7 matrix would be owned by the following sets

of processes:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

class `DistMatrix<T, VC, STAR>`

All public member functions have been described as part of *AbstractDistMatrix<T>*, *GeneralDistMatrix<T,U,V>*, and *DistMatrix<T,U,V>*.

[STAR, VC]

This is the transpose of the above [VC, *] distribution. On the standard 2×3 process grid with a row alignment of zero, a 7×7 matrix would be distributed as:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 & 0 \end{pmatrix}$$

class `DistMatrix<T, STAR, VC>`

All public member functions have been described as part of *AbstractDistMatrix<T>*, *GeneralDistMatrix<T,U,V>*, and *DistMatrix<T,U,V>*.

[VR, STAR]

This distribution makes use of a 1d distribution which uses a row-major ordering of the entire process grid. Since 1d distributions are useful for distributing *vectors*, and a *row-major* ordering is used, the distribution symbol is VR. Again using the simple 2×3 process grid, with a zero column alignment, each entry of a 7×7 matrix would be owned by the following sets of processes:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 5 & 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

class `DistMatrix<T, VR, STAR>`

All public member functions have been described as part of *AbstractDistMatrix<T>*, *GeneralDistMatrix<T,U,V>*, and *DistMatrix<T,U,V>*.

[STAR, VR]

This is the transpose of the above [VR, *] distribution. On the standard 2×3 process grid with a row alignment of zero, a 7×7 matrix would be distributed as:

$$\begin{pmatrix} 0 & 2 & 4 & 1 & 3 & 5 & 0 \\ 0 & 2 & 4 & 1 & 3 & 5 & 0 \\ 0 & 2 & 4 & 1 & 3 & 5 & 0 \\ 0 & 2 & 4 & 1 & 3 & 5 & 0 \\ 0 & 2 & 4 & 1 & 3 & 5 & 0 \\ 0 & 2 & 4 & 1 & 3 & 5 & 0 \\ 0 & 2 & 4 & 1 & 3 & 5 & 0 \end{pmatrix}$$

class DistMatrix<T, STAR, VR>

All public member functions have been described as part of *AbstractDistMatrix*<T>, *GeneralDistMatrix*<T,U,V>, and *DistMatrix*<T,U,V>.

[STAR, STAR]

This “distribution” actually redundantly stores every entry of the associated matrix on every process. Again using a 2×3 process grid, the entries of a 7×7 matrix would be owned by the following sets of processes:

$$\begin{pmatrix} \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} \\ \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} \\ \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} \\ \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} \\ \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} \\ \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} \\ \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} & \{0,1,\dots,5\} \end{pmatrix}$$

class DistMatrix<T, STAR, STAR>

All public member functions have been described as part of *AbstractDistMatrix*<T>, *GeneralDistMatrix*<T,U,V>, and *DistMatrix*<T,U,V>.

[CIRC, CIRC]

This distribution stores the entire matrix on a single process. For instance, if the root process is process 0 with respect to a column-major ordering of the process grid, then the corresponding overlay for the owners of each entry of our 7×7 matrix example would be:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

class `DistMatrix<T, CIRC, CIRC>`

Only two public member functions were not described as part of *AbstractDistMatrix<T>*, *GeneralDistMatrix<T, U, V>*, and *DistMatrix<T, U, V>*.

`void CopyFromRoot(const Matrix<T> &A)`

To be called from the root process in order to assign a sequential matrix to a [CIRC, CIRC] “distributed” matrix.

`void CopyFromNonRoot()`

To be called from non-root processes at the same time that the root process is calling *CopyFromRoot()*.

Some special cases used in Elemental

This list of special cases is here to help clarify the notation used throughout Elemental’s source (as well as this documentation). These are all special cases of *DistMatrix<T, U, V>*.

class `DistMatrix<double, U, V>`

class `DistMatrix<double>`

class `DistMatrix<double, CIRC, CIRC>`

class `DistMatrix<double, MC, MR>`

class `DistMatrix<double, MC, STAR>`

class `DistMatrix<double, MD, STAR>`

class `DistMatrix<double, MR, MC>`

class `DistMatrix<double, MR, STAR>`

class `DistMatrix<double, STAR, MC>`

class `DistMatrix<double, STAR, MD>`

class `DistMatrix<double, STAR, MR>`

class `DistMatrix<double, STAR, STAR>`

class `DistMatrix<double, STAR, VC>`

class `DistMatrix<double, STAR, VR>`

class `DistMatrix<double, VC, STAR>`

class `DistMatrix<double, VR, STAR>`

The underlying datatype is the set of double-precision real numbers.

class `DistMatrix<Complex<double>, U, V>`

class `DistMatrix<Complex<double>>`

class `DistMatrix<Complex<double>, CIRC, CIRC>`

class `DistMatrix<Complex<double>, MC, MR>`

class `DistMatrix<Complex<double>, MC, STAR>`

class `DistMatrix<Complex<double>, MD, STAR>`

class `DistMatrix<Complex<double>, MR, MC>`

```
class DistMatrix<Complex<double>, MR, STAR>
class DistMatrix<Complex<double>, STAR, MC>
class DistMatrix<Complex<double>, STAR, MD>
class DistMatrix<Complex<double>, STAR, MR>
class DistMatrix<Complex<double>, STAR, STAR>
class DistMatrix<Complex<double>, STAR, VC>
class DistMatrix<Complex<double>, STAR, VR>
class DistMatrix<Complex<double>, VC, STAR>
class DistMatrix<Complex<double>, VR, STAR>
```

The underlying datatype is the set of double-precision complex numbers.

```
class DistMatrix<Real, U, V>
class DistMatrix<Real>
class DistMatrix<Real, CIRC, CIRC>
class DistMatrix<Real, MC, MR>
class DistMatrix<Real, MC, STAR>
class DistMatrix<Real, MD, STAR>
class DistMatrix<Real, MR, MC>
class DistMatrix<Real, MR, STAR>
class DistMatrix<Real, STAR, MC>
class DistMatrix<Real, STAR, MD>
class DistMatrix<Real, STAR, MR>
class DistMatrix<Real, STAR, STAR>
class DistMatrix<Real, STAR, VC>
class DistMatrix<Real, STAR, VR>
class DistMatrix<Real, VC, STAR>
class DistMatrix<Real, VR, STAR>
```

The underlying datatype, *Real*, is real (as opposed to complex).

```
class DistMatrix<Complex<Real>, U, V>
class DistMatrix<Complex<Real>>
class DistMatrix<Complex<Real>, CIRC, CIRC>
class DistMatrix<Complex<Real>, MC, MR>
class DistMatrix<Complex<Real>, MC, STAR>
class DistMatrix<Complex<Real>, MD, STAR>
class DistMatrix<Complex<Real>, MR, MC>
class DistMatrix<Complex<Real>, MR, STAR>
```



```

class DistMatrix<Complex<Real>, STAR, MC>
class DistMatrix<Complex<Real>, STAR, MD>
class DistMatrix<Complex<Real>, STAR, MR>
class DistMatrix<Complex<Real>, STAR, STAR>
class DistMatrix<Complex<Real>, STAR, VC>
class DistMatrix<Complex<Real>, STAR, VR>
class DistMatrix<Complex<Real>, VC, STAR>
class DistMatrix<Complex<Real>, VR, STAR>

```

The underlying datatype, *Complex<Real>*, is complex with base type *Real*.

```

class DistMatrix<F, U, V>
class DistMatrix<F>
class DistMatrix<F, CIRC, CIRC>
class DistMatrix<F, MC, MR>
class DistMatrix<F, MC, STAR>
class DistMatrix<F, MD, STAR>
class DistMatrix<F, MR, MC>
class DistMatrix<F, MR, STAR>
class DistMatrix<F, STAR, MC>
class DistMatrix<F, STAR, MD>
class DistMatrix<F, STAR, MR>
class DistMatrix<F, STAR, STAR>
class DistMatrix<F, STAR, VC>
class DistMatrix<F, STAR, VR>
class DistMatrix<F, VC, STAR>
class DistMatrix<F, VR, STAR>

```

The underlying datatype, *F*, is a field.

```

class DistMatrix<int, U, V>
class DistMatrix<int>
class DistMatrix<int, CIRC, CIRC>
class DistMatrix<int, MC, MR>
class DistMatrix<int, MC, STAR>
class DistMatrix<int, MD, STAR>
class DistMatrix<int, MR, MC>
class DistMatrix<int, MR, STAR>
class DistMatrix<int, STAR, MC>

```

```
class DistMatrix<int, STAR, MD>
```

```
class DistMatrix<int, STAR, MR>
```

```
class DistMatrix<int, STAR, STAR>
```

```
class DistMatrix<int, STAR, VC>
```

```
class DistMatrix<int, STAR, VR>
```

```
class DistMatrix<int, VC, STAR>
```

```
class DistMatrix<int, VR, STAR>
```

The underlying datatype is a signed integer (of standard size).

3.6 Matrix views

3.6.1 View a full matrix

```
void View(Matrix<T> &A, Matrix<T> &B)
```

```
void View(DistMatrix<T, U, V> &A, DistMatrix<T, U, V> &B)
```

Make *A* a view of the matrix *B*.

```
Matrix<T> View(Matrix<T> &B)
```

```
DistMatrix<T, U, V> View(DistMatrix<T, U, V> &B)
```

Return a view of the matrix *B*.

```
void LockedView(Matrix<T> &A, const Matrix<T> &B)
```

```
void LockedView(DistMatrix<T, U, V> &A, const DistMatrix<T, U, V> &B)
```

Make *A* a non-mutable view of the matrix *B*.

```
Matrix<T> LockedView(const Matrix<T> &B)
```

```
DistMatrix<T, U, V> LockedView(const DistMatrix<T, U, V> &B)
```

Return a view of the matrix *B*.

3.6.2 View a submatrix

```
void View(Matrix<T> &A, Matrix<T> &B, int i, int j, int height, int width)
```

```
void View(DistMatrix<T, U, V> &A, DistMatrix<T, U, V> &B, int i, int j, int height, int  
width)
```

Make *A* a view of the *height* \times *width* submatrix of *B* starting at coordinate (*i*, *j*).

```
Matrix<T> View(Matrix<T> &B, int i, int j, int height, int width)
```

```
DistMatrix<T, U, V> View(DistMatrix<T, U, V> &B, int i, int j, int height, int width)
```

Return a view of the specified submatrix of *B*.

```
void LockedView(Matrix<T> &A, const Matrix<T> &B, int i, int j, int height, int width)
```

```
void LockedView(DistMatrix<T, U, V> &A, const DistMatrix<T, U, V> &B, int i, int j, int  
height, int width)
```

Make *A* a non-mutable view of the *height* \times *width* submatrix of *B* starting at coordinate (*i*, *j*).

Matrix<T> LockedView(const *Matrix*<T> &B, int i, int j, int height, int width)

DistMatrix<T, U, V> LockedView(const *DistMatrix*<T, U, V> &B, int i, int j, int height, int width)

Return an immutable view of the specified submatrix of B.

3.6.3 View 1x2 matrices

void View1x2(*Matrix*<T> &A, *Matrix*<T> &BL, *Matrix*<T> &BR)

void View1x2(*DistMatrix*<T, U, V> &A, *DistMatrix*<T, U, V> &BL, *DistMatrix*<T, U, V> &BR)

Make A a view of the matrix $\begin{pmatrix} B_L & B_R \end{pmatrix}$.

Matrix<T> View1x2(*Matrix*<T> &BL, *Matrix*<T> &BR)

DistMatrix<T, U, V> View1x2(*DistMatrix*<T, U, V> &BL, *DistMatrix*<T, U, V> &BR)

Return a view of the merged matrix.

void LockedView1x2(*Matrix*<T> &A, const *Matrix*<T> &BL, const *Matrix*<T> &BR)

void LockedView1x2(*DistMatrix*<T, U, V> &A, const *DistMatrix*<T, U, V> &BL, const *DistMatrix*<T, U, V> &BR)

Make A a non-mutable view of the matrix $\begin{pmatrix} B_L & B_R \end{pmatrix}$.

Matrix<T> LockedView1x2(const *Matrix*<T> &BL, const *Matrix*<T> &BR)

DistMatrix<T, U, V> LockedView1x2(const *DistMatrix*<T, U, V> &BL, const *DistMatrix*<T, U, V> &BR)

Return an immutable view of the merged matrix.

3.6.4 View 2x1 matrices

void View2x1(*Matrix*<T> &A, *Matrix*<T> &BT, *Matrix*<T> &BB)

void View2x1(*DistMatrix*<T, U, V> &A, *DistMatrix*<T, U, V> &BT, *DistMatrix*<T, U, V> &BB)

Make A a view of the matrix $\begin{pmatrix} B_T \\ B_B \end{pmatrix}$.

Matrix<T> View2x1(*Matrix*<T> &BT, *Matrix*<T> &BB)

DistMatrix<T, U, V> View2x1(*DistMatrix*<T, U, V> &BT, *DistMatrix*<T, U, V> &BB)

Return a view of the merged matrix.

void LockedView2x1(*Matrix*<T> &A, const *Matrix*<T> &BT, const *Matrix*<T> &BB)

void LockedView2x1(*DistMatrix*<T, U, V> &A, const *DistMatrix*<T, U, V> &BT, const *DistMatrix*<T, U, V> &BB)

Make A a non-mutable view of the matrix $\begin{pmatrix} B_T \\ B_B \end{pmatrix}$.

Matrix<T> LockedView2x1(const *Matrix*<T> &BT, const *Matrix*<T> &BB)

DistMatrix<T, U, V> LockedView2x1(const *DistMatrix*<T, U, V> &BT, const *DistMatrix*<T, U, V> &BB)

Return a view of the merged matrix.

3.6.5 View 2x2 matrices

```
void View2x2(Matrix<T> &A, Matrix<T> &BTL, Matrix<T> &BTR, Matrix<T> &BBL,
            Matrix<T> &BBR)
```

```
void View2x2(DistMatrix<T, U, V> &A, DistMatrix<T, U, V> &BTL, DistMatrix<T, U,
            V> &BTR, DistMatrix<T, U, V> &BBL, DistMatrix<T, U, V> &BBR)
```

Make A a view of the matrix $\begin{pmatrix} B_{TL} & B_{TR} \\ B_{BL} & B_{BR} \end{pmatrix}$.

```
Matrix<T> View2x2(Matrix<T> &BTL, Matrix<T> &BTR, Matrix<T> &BBL, Ma-
            trix<T> &BBR)
```

```
DistMatrix<T, U, V> View2x2(DistMatrix<T, U, V> &BTL, DistMatrix<T, U, V> &BTR,
            DistMatrix<T, U, V> &BBL, DistMatrix<T, U, V> &BBR)
```

Return a view of the merged matrix.

```
void LockedView2x2(Matrix<T> &A, const Matrix<T> &BTL, const Matrix<T> &BTR,
            const Matrix<T> &BBL, const Matrix<T> &BBR)
```

```
void LockedView2x2(DistMatrix<T, U, V> &A, const DistMatrix<T, U, V> &BTL, const
            DistMatrix<T, U, V> &BTR, const DistMatrix<T, U, V> &BBL,
            const DistMatrix<T, U, V> &BBR)
```

Make A a non-mutable view of the matrix $\begin{pmatrix} B_{TL} & B_{TR} \\ B_{BL} & B_{BR} \end{pmatrix}$.

```
Matrix<T> LockedView2x2(const Matrix<T> &BTL, const Matrix<T> &BTR, const Ma-
            trix<T> &BBL, const Matrix<T> &BBR)
```

```
DistMatrix<T, U, V> LockedView2x2(const DistMatrix<T, U, V> &BTL, const DistMa-
            trix<T, U, V> &BTR, const DistMatrix<T, U, V>
            &BBL, const DistMatrix<T, U, V> &BBR)
```

Return an immutable view of the merged matrix.

3.7 Matrix partitions

The following routines are slight tweaks of the FLAME project's (as well as LAPACK's) approach to submatrix tracking; the difference is that they have "locked" versions, which are meant for creating partitionings where the submatrices cannot be modified.

3.7.1 PartitionUp

Given an $m \times n$ matrix A , configure A_T and A_B to view the local data of A corresponding to the partition

$$A = \begin{pmatrix} A_T \\ A_B \end{pmatrix},$$

where A_B is of a specified height.

```
void PartitionUp(Matrix<T> &A, Matrix<T> &AT, Matrix<T> &AB, int heightAB =
            Blocksize())
```

```
void LockedPartitionUp(const Matrix<T> &A, Matrix<T> &AT, Matrix<T> &AB, int
            heightAB = Blocksize())
```

Templated over the datatype, T , of the serial matrix A .

```
void PartitionUp(DistMatrix<T, U, V> &A, DistMatrix<T, U, V> &AT, DistMatrix<T,
    U, V> &AB, int heightAB = Blocksize())
```

```
void LockedPartitionUp(const DistMatrix<T, U, V> &A, DistMatrix<T, U, V> &AT,
    DistMatrix<T, U, V> &AB, int heightAB = Blocksize())
```

Templated over the datatype, T , and distribution scheme, (U, V) , of the distributed matrix A .

3.7.2 PartitionDown

Given an $m \times n$ matrix A , configure AT and AB to view the local data of A corresponding to the partition

$$A = \begin{pmatrix} A_T \\ A_B \end{pmatrix},$$

where A_T is of a specified height.

```
void PartitionDown(Matrix<T> &A, Matrix<T> &AT, Matrix<T> &AB, int heightAT =
    Blocksize())
```

```
void LockedPartitionDown(const Matrix<T> &A, Matrix<T> &AT, Matrix<T> &AB,
    int heightAT = Blocksize())
```

Templated over the datatype, T , of the serial matrix A .

```
void PartitionDown(DistMatrix<T, U, V> &A, DistMatrix<T, U, V> &AT, DistMa-
    trix<T, U, V> &AB, int heightAT = Blocksize())
```

```
void LockedPartitionDown(const DistMatrix<T, U, V> &A, DistMatrix<T, U, V> &AT,
    DistMatrix<T, U, V> &AB, int heightAT = Blocksize())
```

Templated over the datatype, T , and distribution scheme, (U, V) , of the distributed matrix A .

3.7.3 PartitionLeft

Given an $m \times n$ matrix A , configure AL and AR to view the local data of A corresponding to the partition

$$A = (A_L \ A_R),$$

where A_R is of a specified width.

```
void PartitionLeft(Matrix<T> &A, Matrix<T> &AL, Matrix<T> &AR, int widthAR =
    Blocksize())
```

```
void LockedPartitionLeft(const Matrix<T> &A, Matrix<T> &AL, Matrix<T> &AR,
    int widthAR = Blocksize())
```

Templated over the datatype, T , of the serial matrix A .

```
void PartitionLeft(DistMatrix<T, U, V> &A, DistMatrix<T, U, V> &AL, DistMa-
    trix<T, U, V> &AR, int widthAR = Blocksize())
```

```
void LockedPartitionLeft(const DistMatrix<T, U, V> &A, DistMatrix<T, U, V> &AL,
    DistMatrix<T, U, V> &AR, int widthAR = Blocksize())
```

Templated over the datatype, T , and the distribution scheme, (U, V) , of the distributed matrix A .

3.7.4 PartitionRight

Given an $m \times n$ matrix A , configure AL and AR to view the local data of A corresponding to the partition

$$A = \left(A_L \quad A_R \right),$$

where A_L is of a specified width.

```
void PartitionRight(Matrix<T> &A, Matrix<T> &AL, Matrix<T> &AR, int widthAL
                  = Blocksize())
```

```
void LockedPartitionRight(const Matrix<T> &A, Matrix<T> &AL, Matrix<T> &AR,
                          int widthAL = Blocksize())
```

Templated over the datatype, T , of the serial matrix A .

```
void PartitionRight(DistMatrix<T, U, V> &A, DistMatrix<T, U, V> &AL, DistMa-
                  trix<T, U, V> &AR, int widthAL = Blocksize())
```

```
void LockedPartitionRight(const DistMatrix<T, U, V> &A, DistMatrix<T, U, V> &AL,
                          DistMatrix<T, U, V> &AR, int widthAL = Blocksize())
```

Templated over the datatype, T , and the distribution scheme, (U,V) , of the distributed matrix A .

3.7.5 PartitionUpDiagonal

Given an $m \times n$ matrix A , configure ATL , ATR , ABL , and ABR to view the local data of A corresponding to the partitioning

$$A = \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix},$$

where the diagonal of A_{BR} lies on the main diagonal (aka, the *left* diagonal) of A and is of the specified height/width.

```
void PartitionUpDiagonal(Matrix<T> &A, Matrix<T> &ATL, Matrix<T> &ATR, Ma-
                        trix<T> &ABL, Matrix<T> &ABR, int diagDist = Block-
                        size())
```

```
void LockedPartitionUpDiagonal(const Matrix<T> &A, Matrix<T> &ATL, Ma-
                               trix<T> &ATR, Matrix<T> &ABL, Matrix<T>
                               &ABR, int diagDist = Blocksize())
```

Templated over the datatype, T , of the serial matrix A .

```
void PartitionUpDiagonal(DistMatrix<T, U, V> &A, DistMatrix<T, U, V> &ATL, Dist-
                        Matrix<T, U, V> &ATR, DistMatrix<T, U, V> &ABL, Dist-
                        Matrix<T, U, V> &ABR, int diagDist = Blocksize())
```

```
void LockedPartitionUpDiagonal(const DistMatrix<T, U, V> &A, DistMatrix<T, U,
                               V> &ATL, DistMatrix<T, U, V> &ATR, DistMa-
                               trix<T, U, V> &ABL, DistMatrix<T, U, V> &ABR,
                               int diagDist = Blocksize())
```

Templated over the datatype, T , and the distribution scheme, (U,V) , of the distributed matrix A .

3.7.6 PartitionUpOffsetDiagonal

Given an $m \times n$ matrix A , configure ATL , ATR , ABL , and ABR to view the local data of A corresponding to the partitioning

$$A = \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix},$$

where the diagonal of A_{BR} lies on the offset diagonal of A , where the main diagonal corresponds to an offset of 0, the subdiagonal is an offset of -1 , the superdiagonal is an offset of 1, etc. The length of the diagonal of A_{BR} is specified as `diagDist`.

```
void PartitionUpOffsetDiagonal(int offset, Matrix<T> &A, Matrix<T> &ATL, Matrix<T> &ATR, Matrix<T> &ABL, Matrix<T> &ABR, int diagDist = Blocksize())
```

```
void LockedPartitionUpOffsetDiagonal(int offset, const Matrix<T> &A, Matrix<T> &ATL, Matrix<T> &ATR, Matrix<T> &ABL, Matrix<T> &ABR, int diagDist = Blocksize())
```

Templated over the datatype, T , of the serial matrix A .

```
void PartitionUpOffsetDiagonal(int offset, DistMatrix<T, U, V> &A, DistMatrix<T, U, V> &ATL, DistMatrix<T, U, V> &ATR, DistMatrix<T, U, V> &ABL, DistMatrix<T, U, V> &ABR, int diagDist = Blocksize())
```

```
void LockedPartitionUpOffsetDiagonal(int offset, const DistMatrix<T, U, V> &A, DistMatrix<T, U, V> &ATL, DistMatrix<T, U, V> &ATR, DistMatrix<T, U, V> &ABL, DistMatrix<T, U, V> &ABR, int diagDist = Blocksize())
```

Templated over the datatype, T , and the distribution scheme, (U,V) , of the distributed matrix A .

3.7.7 PartitionDownDiagonal

Given an $m \times n$ matrix A , configure ATL , ATR , ABL , and ABR to view the local data of A corresponding to the partitioning

$$A = \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix},$$

where the diagonal of A_{TL} is of the specified length and lies on the main diagonal (aka, the *left* diagonal) of A .

```
void PartitionDownDiagonal(Matrix<T> &A, Matrix<T> &ATL, Matrix<T> &ATR, Matrix<T> &ABL, Matrix<T> &ABR, int diagDist = Blocksize())
```

```
void LockedPartitionDownDiagonal(const Matrix<T> &A, Matrix<T> &ATL, Matrix<T> &ATR, Matrix<T> &ABL, Matrix<T> &ABR, int diagDist = Blocksize())
```

Templated over the datatype, T , of the serial matrix A .

```
void PartitionDownDiagonal(DistMatrix<T, U, V> &A, DistMatrix<T, U, V> &ATL, DistMatrix<T, U, V> &ATR, DistMatrix<T, U, V> &ABL, DistMatrix<T, U, V> &ABR, int diagDist = Blocksize())
```

```
void LockedPartitionDownDiagonal(const DistMatrix<T, U, V> &A, DistMatrix<T, U,
                                V> &ATL, DistMatrix<T, U, V> &ATR, DistMa-
                                trix<T, U, V> &ABL, DistMatrix<T, U, V> &ABR,
                                int diagDist = Blocksize())
```

Templated over the datatype, T , and the distribution scheme, (U, V) , of the distributed matrix A .

3.7.8 PartitionDownOffsetDiagonal

Given an $m \times n$ matrix A , configure ATL , ATR , ABL , and ABR to view the local data of A corresponding to the partitioning

$$A = \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix},$$

where the diagonal of A_{BR} lies on the offset diagonal of A , where the main diagonal corresponds to an offset of 0, the subdiagonal is an offset of -1 , the superdiagonal is an offset of 1, etc. The length of the diagonal of A_{TL} is specified as `diagDist`.

```
void PartitionDownOffsetDiagonal(int offset, Matrix<T> &A, Matrix<T> &ATL, Ma-
                                trix<T> &ATR, Matrix<T> &ABL, Matrix<T>
                                &ABR, int diagDist = Blocksize())
```

```
void LockedPartitionDownOffsetDiagonal(int offset, const Matrix<T> &A, Ma-
                                trix<T> &ATL, Matrix<T> &ATR, Ma-
                                trix<T> &ABL, Matrix<T> &ABR, int di-
                                agDist = Blocksize())
```

Templated over the datatype, T , of the serial matrix A .

```
void PartitionDownOffsetDiagonal(int offset, DistMatrix<T, U, V> &A, DistMatrix<T,
                                U, V> &ATL, DistMatrix<T, U, V> &ATR, Dist-
                                Matrix<T, U, V> &ABL, DistMatrix<T, U, V>
                                &ATL, int diagDist = Blocksize())
```

```
void LockedPartitionDownOffsetDiagonal(int offset, const DistMatrix<T, U, V> &A,
                                DistMatrix<T, U, V> &ATL, DistMatrix<T,
                                U, V> &ATR, DistMatrix<T, U, V> &ABL,
                                DistMatrix<T, U, V> &ABR, int diagDist =
                                Blocksize())
```

Templated over the datatype, T , and the distribution scheme, (U, V) , of the distributed matrix A .

3.8 Repartitioning matrices

3.8.1 RepartitionUp

Given the partition

$$A = \begin{pmatrix} A_T \\ A_B \end{pmatrix},$$

and a blocksize, n_b , turn the two-way partition into the three-way partition

$$\left(\begin{array}{c} A_T \\ A_B \end{array} \right) = \left(\begin{array}{c} A_0 \\ A_1 \\ A_2 \end{array} \right),$$

where A_1 is of height n_b and $A_2 = A_B$.

```
void RepartitionUp(Matrix<T> &AT, Matrix<T> &A0, Matrix<T> &A1, Matrix<T>
&AB, Matrix<T> &A2, int bsize = Blocksize())
```

```
void LockedRepartitionUp(const Matrix<T> &AT, Matrix<T> &A0, Matrix<T> &A1,
const Matrix<T> &AB, Matrix<T> &A2, int bsize = Block-
size())
```

Templated over the datatype, T .

```
void RepartitionUp(DistMatrix<T, U, V> &AT, DistMatrix<T, U, V> &A0, DistMa-
trix<T, U, V> &A1, DistMatrix<T, U, V> &AB, DistMatrix<T, U,
V> &A2, int bsize = Blocksize())
```

```
void LockedRepartitionUp(const DistMatrix<T, U, V> &AT, DistMatrix<T, U, V> &A0,
DistMatrix<T, U, V> &A1, const DistMatrix<T, U, V> &AB,
DistMatrix<T, U, V> &A2, int bsize = Blocksize())
```

Templated over the datatype, T , and distribution scheme, (U, V) .

Note that each of the above routines is meant to be used in a manner similar to the following:

```
RepartitionUp( AT,  A0,
               A1,
               /**/ /**/
               AB, A2, blocksize );
```

3.8.2 RepartitionDown

Given the partition

$$A = \left(\begin{array}{c} A_T \\ A_B \end{array} \right),$$

and a blocksize, n_b , turn the two-way partition into the three-way partition

$$\left(\begin{array}{c} A_T \\ A_B \end{array} \right) = \left(\begin{array}{c} A_0 \\ A_1 \\ A_2 \end{array} \right),$$

where A_1 is of height n_b and $A_0 = A_T$.

```
void RepartitionDown(Matrix<T> &AT, Matrix<T> &A0, Matrix<T> &A1, Matrix<T>
&AB, Matrix<T> &A2, int bsize = Blocksize())
```

```
void LockedRepartitionDown(const Matrix<T> &AT, Matrix<T> &A0, Matrix<T>
&A1, const Matrix<T> &AB, Matrix<T> &A2, int bsize
= Blocksize())
```

Templated over the datatype, T .

```
void RepartitionDown(DistMatrix<T, U, V> &AT, DistMatrix<T, U, V> &A0, DistMa-
trix<T, U, V> &A1, DistMatrix<T, U, V> &AB, DistMatrix<T,
U, V> &A2, int bsize = Blocksize())
```

```
void LockedRepartitionDown(const DistMatrix<T, U, V> &AT, DistMatrix<T, U, V>
                          &A0, DistMatrix<T, U, V> &A1, const DistMatrix<T, U,
                          V> &AB, DistMatrix<T, U, V> &A2, int bsize = Block-
                          size())
```

Templated over the datatype, T , and distribution scheme, (U,V) .

Note that each of the above routines is meant to be used in a manner similar to the following:

```
RepartitionDown( AT,  A0,
                /**/ /**/
                A1,
                AB,  A2, blocksize );
```

3.8.3 RepartitionLeft

Given the partition

$$A = (A_L \mid A_R),$$

and a blocksize, n_b , turn the two-way partition into the three-way partition

$$(A_L \mid A_R) = (A_0 \ A_1 \mid A_2),$$

where A_1 is of width n_b and $A_2 = A_R$.

```
void RepartitionLeft(Matrix<T> &AL, Matrix<T> &AR, Matrix<T> &A0, Ma-
                    trix<T> &A1, Matrix<T> &A2, int bsize = Blocksize())
```

```
void LockedRepartitionLeft(const Matrix<T> &AL, const Matrix<T> &AR, Ma-
                           trix<T> &A0, Matrix<T> &A1, Matrix<T> &A2, int
                           bsize = Blocksize())
```

Templated over the datatype, T .

```
void RepartitionLeft(DistMatrix<T, U, V> &AL, DistMatrix<T, U, V> &AR, DistMa-
                    trix<T, U, V> &A0, DistMatrix<T, U, V> &A1, DistMatrix<T, U,
                    V> &A2, int bsize = Blocksize())
```

```
void LockedRepartitionLeft(const DistMatrix<T, U, V> &AL, const DistMatrix<T, U,
                            V> &AR, DistMatrix<T, U, V> &A0, DistMatrix<T, U,
                            V> &A1, DistMatrix<T, U, V> &A2, int bsize = Block-
                            size())
```

Templated over the datatype, T , and distribution scheme, (U,V) .

Note that each of the above routines is meant to be used in a manner similar to the following:

```
RepartitionLeft( AL,    /**/ AR,
                A0, A1, /**/ A2, blocksize );
```

3.8.4 RepartitionRight

Given the partition

$$A = (A_L \mid A_R),$$

and a blocksize, n_b , turn the two-way partition into the three-way partition

$$(A_L \mid A_R) = (A_0 \mid A_1 \ A_2),$$

where A_1 is of width n_b and $A_0 = A_L$.

```
void RepartitionRight(Matrix<T> &AL, Matrix<T> &AR, Matrix<T> &A0, Matrix<T> &A1, Matrix<T> &A2, int bsize = Blocksize())
```

```
void LockedRepartitionRight(const Matrix<T> &AL, const Matrix<T> &AR, Matrix<T> &A0, Matrix<T> &A1, Matrix<T> &A2, int bsize = Blocksize())
```

Templated over the datatype, T .

```
void RepartitionRight(DistMatrix<T, U, V> &AL, DistMatrix<T, U, V> &AR, DistMatrix<T, U, V> &A0, DistMatrix<T, U, V> &A1, DistMatrix<T, U, V> &A2, int bsize = Blocksize())
```

```
void LockedRepartitionRight(const DistMatrix<T, U, V> &AL, const DistMatrix<T, U, V> &AR, DistMatrix<T, U, V> &A0, DistMatrix<T, U, V> &A1, DistMatrix<T, U, V> &A2, int bsize = Blocksize())
```

Templated over the datatype, T , and distribution scheme, (U, V) .

Note that each of the above routines is meant to be used in a manner similar to the following:

```
RepartitionRight( AL, /**/ AR,
                 A0, /**/ A1, A2, blocksize );
```

3.8.5 RepartitionUpDiagonal

Given the partition

$$A = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right),$$

turn the two-by-two partition into the three-by-three partition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right) = \left(\begin{array}{cc|c} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array} \right),$$

where A_{11} is $n_b \times n_b$ and the corresponding quadrants are equivalent.

```
void RepartitionUpDiagonal(Matrix<T> &ATL, Matrix<T> &ATR, Matrix<T> &A00, Matrix<T> &A01, Matrix<T> &A02, Matrix<T> &A10, Matrix<T> &A11, Matrix<T> &A12, Matrix<T> &ABL, Matrix<T> &ABR, Matrix<T> &A20, Matrix<T> &A21, Matrix<T> &A22, int bsize = Blocksize())
```

```
void LockedRepartitionUpDiagonal(const Matrix<T> &ATL, const Matrix<T> &ATR, Matrix<T> &A00, Matrix<T> &A01, Matrix<T> &A02, Matrix<T> &A10, Matrix<T> &A11, Matrix<T> &A12, const Matrix<T> &ABL, const Matrix<T> &ABR, Matrix<T> &A20, Matrix<T> &A21, Matrix<T> &A22, int bsize = Blocksize())
```

Templated over the datatype, T .

```
void RepartitionUpDiagonal(DistMatrix<T, U, V> &ATL, DistMatrix<T, U, V> &ATR,
    DistMatrix<T, U, V> &A00, DistMatrix<T, U, V> &A01,
    DistMatrix<T, U, V> &A02, DistMatrix<T, U, V> &A10,
    DistMatrix<T, U, V> &A11, DistMatrix<T, U, V> &A12,
    DistMatrix<T, U, V> &ABL, DistMatrix<T, U, V> &ABR,
    DistMatrix<T, U, V> &A20, DistMatrix<T, U, V> &A21,
    DistMatrix<T, U, V> &A22, int bsize = Blocksize())
```

```
void LockedRepartitionUpDiagonal(const DistMatrix<T, U, V> &ATL, const DistMatrix<T, U, V> &ATR,
    DistMatrix<T, U, V> &A00, DistMatrix<T, U, V> &A01, DistMatrix<T, U, V> &A02,
    DistMatrix<T, U, V> &A10, DistMatrix<T, U, V> &A11, DistMatrix<T, U, V> &A12,
    const DistMatrix<T, U, V> &ABL, const DistMatrix<T, U, V> &ABR,
    DistMatrix<T, U, V> &A20, DistMatrix<T, U, V> &A21, DistMatrix<T, U, V> &A22,
    int bsize = Blocksize())
```

Templated over the datatype, T , and distribution scheme, (U, V) .

Note that each of the above routines is meant to be used in a manner similar to the following:

```
RepartitionUpDiagonal( ATL, /**/ ATR,  A00, A01, /**/ A02,
    /**/      A10, A11, /**/ A12,
    /***/***/ /***/***/
    ABL, /**/ ABR,  A20, A21, /**/ A22, blocksize );
```

3.8.6 RepartitionDownDiagonal

Given the partition

$$A = \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right),$$

turn the two-by-two partition into the three-by-three partition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array} \right) = \left(\begin{array}{c|cc} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{array} \right),$$

where A_{11} is $n_b \times n_b$ and the corresponding quadrants are equivalent.

```
void RepartitionDownDiagonal(Matrix<T> &ATL, Matrix<T> &ATR, Matrix<T>
    &A00, Matrix<T> &A01, Matrix<T> &A02, Matrix<T> &A10,
    Matrix<T> &A11, Matrix<T> &A12, Matrix<T> &A12,
    Matrix<T> &ABL, Matrix<T> &ABR, Matrix<T>
    &A20, Matrix<T> &A21, Matrix<T> &A22, int bsize
    = Blocksize())
```

```
void LockedRepartitionDownDiagonal(const Matrix<T> &ATL, const Matrix<T>
    &ATR, Matrix<T> &A00, Matrix<T> &A01,
    Matrix<T> &A02, Matrix<T> &A10, Matrix<T> &A11,
    Matrix<T> &A12, const Matrix<T> &ABL, const Matrix<T> &ABR,
    Matrix<T> &A20, Matrix<T> &A21, Matrix<T> &A22,
    int bsize = Blocksize())
```

Templated over the datatype, T .

```
void RepartitionDownDiagonal(DistMatrix<T, U, V> &ATL, DistMatrix<T, U, V>
&ATR, DistMatrix<T, U, V> &A00, DistMatrix<T, U,
V> &A01, DistMatrix<T, U, V> &A02, DistMatrix<T,
U, V> &A10, DistMatrix<T, U, V> &A11, DistMa-
trix<T, U, V> &A12, DistMatrix<T, U, V> &ABL,
DistMatrix<T, U, V> &ABR, DistMatrix<T, U, V>
&A20, DistMatrix<T, U, V> &A21, DistMatrix<T, U,
V> &A22, int bsize = Blocksize())
```

```
void LockedRepartitionDownDiagonal(const DistMatrix<T, U, V> &ATL, const Dist-
Matrix<T, U, V> &ATR, DistMatrix<T, U, V>
&A00, DistMatrix<T, U, V> &A01, DistMa-
trix<T, U, V> &A02, DistMatrix<T, U, V>
&A10, DistMatrix<T, U, V> &A11, DistMa-
trix<T, U, V> &A12, const DistMatrix<T, U, V>
&ABL, const DistMatrix<T, U, V> &ABR, Dist-
Matrix<T, U, V> &A20, DistMatrix<T, U, V>
&A21, DistMatrix<T, U, V> &A22, int bsize =
Blocksize())
```

Templated over the datatype, T , and distribution scheme, (U, V) .

Note that each of the above routines is meant to be used in a manner similar to the following:

```
RepartitionDownDiagonal( ATL, /**/ ATR, A00, /**/ A01, A02,
/**/ /**/
/**/ A10, /**/ A11, A12,
ABL, /**/ ABR, A20, /**/ A21, A22, blocksize );
```

3.9 Moving partitions

3.9.1 SlidePartitionUp

Simultaneously slide and merge the partition

$$A = \begin{pmatrix} A_0 \\ A_1 \\ A_2 \end{pmatrix},$$

into

$$\begin{pmatrix} A_T \\ A_B \end{pmatrix} = \begin{pmatrix} A_0 \\ A_1 \\ A_2 \end{pmatrix}.$$

```
void SlidePartitionUp(Matrix<T> &AT, Matrix<T> &A0, Matrix<T> &A1, Ma-
trix<T> &AB, Matrix<T> &A2)
```

```
void SlideLockedPartitionUp(Matrix<T> &AT, const Matrix<T> &A0, const Ma-
trix<T> &A1, Matrix<T> &AB, const Matrix<T> &A2)
```

Templated over the datatype, T .

```
void SlidePartitionUp(DistMatrix<T, U, V> &AT, DistMatrix<T, U, V> &A0, DistMa-
trix<T, U, V> &A1, DistMatrix<T, U, V> &AB, DistMatrix<T,
U, V> &A2)
```

```
void SlideLockedPartitionUp(DistMatrix<T, U, V> &AT, const DistMatrix<T, U, V>
    &A0, const DistMatrix<T, U, V> &A1, DistMatrix<T, U,
    V> &AB, const DistMatrix<T, U, V> &A2)
    Templated over the datatype, T, and distribution scheme, (U,V).
```

Note that each of the above routines is meant to be used in a manner similar to the following:

```
SlidePartitionUp( AT, A0,
    /**/ /**/
    A1,
    AB, A2 );
```

3.9.2 SlidePartitionDown

Simultaneously slide and merge the partition

$$A = \begin{pmatrix} A_0 \\ A_1 \\ A_2 \end{pmatrix},$$

into

$$\begin{pmatrix} A_T \\ A_B \end{pmatrix} = \begin{pmatrix} A_0 \\ A_1 \\ A_2 \end{pmatrix}.$$

```
void SlidePartitionDown(Matrix<T> &AT, Matrix<T> &A0, Matrix<T> &A1, Ma-
    trix<T> &AB, Matrix<T> &A2)
```

```
void SlideLockedPartitionDown(Matrix<T> &AT, const Matrix<T> &A0, const Ma-
    trix<T> &A1, Matrix<T> &AB, const Matrix<T>
    &A2)
```

Templated over the datatype, *T*.

```
void SlidePartitionDown(DistMatrix<T, U, V> &AT, DistMatrix<T, U, V> &A0, Dist-
    Matrix<T, U, V> &A1, DistMatrix<T, U, V> &AB, DistMa-
    trix<T, U, V> &A2)
```

```
void SlideLockedPartitionDown(DistMatrix<T, U, V> &AT, const DistMatrix<T, U,
    V> &A0, const DistMatrix<T, U, V> &A1, DistMa-
    trix<T, U, V> &AB, const DistMatrix<T, U, V> &A2)
```

Templated over the datatype, *T*, and distribution scheme, (*U,V*).

Note that each of the above routines is meant to be used in a manner similar to the following:

```
SlidePartitionDown( AT, A0,
    A1,
    /**/ /**/
    AB, A2 );
```

3.9.3 SlidePartitionLeft

Simultaneously slide and merge the partition

$$A = (A_0 \ A_1 \mid A_2)$$

into

$$(A_L \mid A_R) = (A_0 \mid A_1 \ A_2).$$

```
void SlidePartitionLeft(Matrix<T> &AL, Matrix<T> &AR, Matrix<T> &A0, Matrix<T> &A1, Matrix<T> &A2)
```

```
void SlidePartitionLeft(DistMatrix<T, U, V> &AL, DistMatrix<T, U, V> &AR, DistMatrix<T, U, V> &A0, DistMatrix<T, U, V> &A1, DistMatrix<T, U, V> &A2)
```

Templated over the datatype, T .

```
void SlideLockedPartitionLeft(Matrix<T> &AL, Matrix<T> &AR, const Matrix<T> &A0, const Matrix<T> &A1, const Matrix<T> &A2)
```

```
void SlideLockedPartitionLeft(DistMatrix<T, U, V> &AL, DistMatrix<T, U, V> &AR, const DistMatrix<T, U, V> &A0, const DistMatrix<T, U, V> &A1, const DistMatrix<T, U, V> &A2)
```

Templated over the datatype, T , and distribution scheme, (U,V) .

Note that each of the above routines is meant to be used in a manner similar to the following:

```
SlidePartitionLeft( AL, /**/ AR,
                  AO, /**/ A1, A2 );
```

3.9.4 SlidePartitionRight

Simultaneously slide and merge the partition

$$A = (A_0 \mid A_1 \ A_2)$$

into

$$(A_L \mid A_R) = (A_0 \ A_1 \mid A_2).$$

```
void SlidePartitionRight(Matrix<T> &AL, Matrix<T> &AR, Matrix<T> &A0, Matrix<T> &A1, Matrix<T> &A2)
```

```
void SlidePartitionRight(DistMatrix<T, U, V> &AL, DistMatrix<T, U, V> &AR, DistMatrix<T, U, V> &A0, DistMatrix<T, U, V> &A1, DistMatrix<T, U, V> &A2)
```

Templated over the datatype, T .

```
void SlideLockedPartitionRight(Matrix<T> &AL, Matrix<T> &AR, const Matrix<T> &A0, const Matrix<T> &A1, const Matrix<T> &A2)
```

```
void SlideLockedPartitionRight(DistMatrix<T, U, V> &AL, DistMatrix<T, U, V> &AR, const DistMatrix<T, U, V> &A0, const DistMatrix<T, U, V> &A1, const DistMatrix<T, U, V> &A2)
```

Templated over the datatype, T , and distribution scheme, (U,V) .

Note that each of the above routines is meant to be used in a manner similar to the following:

```
SlidePartitionRight( AL, /**/ AR,
                    A0, A1, /**/ A2 );
```

3.9.5 SlidePartitionUpDiagonal

Simultaneously slide and merge the partition

$$A = \left(\begin{array}{cc|c} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{array} \right)$$

into

$$\left(\begin{array}{c|cc} A_{TL} & A_{TR} & \\ \hline A_{BL} & A_{BR} & \end{array} \right) = \left(\begin{array}{c|cc} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{array} \right).$$

Note that the above routines are meant to be used as:

```
SlidePartitionUpDiagonal( ATL, /**/ ATR, A00, /**/ A01, A02,
                        /***/ /***/
                        /**/ A10, /**/ A11, A12,
                        ABL, /**/ ABR, A20, /**/ A21, A22 );
```

3.9.6 SlidePartitionDownDiagonal

Simultaneously slide and merge the partition

$$A = \left(\begin{array}{c|cc} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{array} \right)$$

into

$$\left(\begin{array}{c|cc} A_{TL} & A_{TR} & \\ \hline A_{BL} & A_{BR} & \end{array} \right) = \left(\begin{array}{c|cc} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{array} \right).$$

Note that the above routines are meant to be used as:

```
SlidePartitionDownDiagonal( ATL, /**/ ATR, A00, A01, /**/ A02,
                           /**/ A10, A11, /**/ A12,
                           /***/ /***/
                           ABL, /**/ ABR, A20, A21, /**/ A22 );
```

3.10 The “Axy” interface

The Axy interface is Elemental’s version of the PLAPACK Axy interface, where *Axy* is derived from the BLAS shorthand for $Y := \alpha X + Y$ (Alpha X Plus Y). Rather than always requiring users to manually fill their distributed matrix, this interface provides a mechanism so that individual processes can independently submit local submatrices which will be automatically redistributed and added onto the global distributed matrix (this would be LOCAL_TO_GLOBAL

mode). The interface also allows for the reverse: each process may asynchronously request arbitrary subset of the global distributed matrix (GLOBAL_TO_LOCAL mode).

Note: The catch is that, in order for this behavior to be possible, all of the processes that share a particular distributed matrix must synchronize at the beginning and end of the Axy interface usage (these synchronizations correspond to the Attach and Detach member functions). The distributed matrix should **not** be manually modified between the Attach and Detach calls.

An example usage might be:

```
#include "elemental.hpp"
using namespace elem;
...
// Create an 8 x 8 distributed matrix over the given grid
DistMatrix<double> A( 8, 8, grid );

// Set every entry of A to zero
MakeZeros( A );

// Open up a LOCAL_TO_GLOBAL interface to A
AxyInterface<double> interface;
interface.Attach( LOCAL_TO_GLOBAL, A );

// If we are process 0, then create a 3 x 3 identity matrix, B,
// and Axy it into the bottom-right of A (using alpha=2)
// NOTE: The bottom-right 3 x 3 submatrix starts at the (5,5)
//       entry of A.
// NOTE: Every process is free to Axy as many submatrices as they
//       desire at this point.
if( grid.VCRank() == 0 )
{
    Matrix<double> B;
    Identity( B, 3, 3 );
    interface.Axy( 2.0, B, 5, 5 );
}

// Have all processes collectively detach from A
interface.Detach();

// Print the updated A
Print( A, "Distributed A" );

// Reattach to A, but in the GLOBAL_TO_LOCAL direction
interface.Attach( GLOBAL_TO_LOCAL, A );

// Have process 0 request a copy of the entire distributed matrix
//
// NOTE: Every process is free to Axy as many submatrices as they
//       desire at this point.
Matrix<double> C;
if( grid.VCRank() == 0 )
{
    Zeros( C, 8, 8 );
    interface.Axy( 1.0, C, 0, 0 );
}

// Collectively detach in order to finish filling process 0's request
```

```
interface.Detach();

// Process 0 can now locally print its copy of A
if( g.VCRank() == 0 )
    Print( C, "Process 0's local copy of A" );
```

The output would be

```
Distributed A
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 2 0 0
0 0 0 0 0 0 2 0
0 0 0 0 0 0 0 2

Process 0's local copy of A
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 2 0 0
0 0 0 0 0 0 2 0
0 0 0 0 0 0 0 2
```

enum AxyType

enumerator LOCAL_TO_GLOBAL

enumerator GLOBAL_TO_LOCAL

class AxyInterface<T>

AxyInterface()

Initialize a blank instance of the interface class. It will need to later be attached to a distributed matrix before any Axy's can occur.

AxyInterface(*AxyType* type, *DistMatrix*<T, MC, MR> &Z)

Initialize an interface to the distributed matrix Z, where type can be either LOCAL_TO_GLOBAL or GLOBAL_TO_LOCAL.

AxyInterface(*AxyType* type, **const** *DistMatrix*<T, MC, MR> &Z)

Initialize an interface to the (unmodifiable) distributed matrix Z; since Z cannot be modified, the only sensical AxyType is GLOBAL_TO_LOCAL. The AxyType argument was kept in order to be consistent with the previous routine.

void Attach(*AxyType* type, *DistMatrix*<T, MC, MR> &Z)

Attach to the distributed matrix Z, where type can be either LOCAL_TO_GLOBAL or GLOBAL_TO_LOCAL.

void Attach(*AxyType* type, **const** *DistMatrix*<T, MC, MR> &Z)

Attach to the (unmodifiable) distributed matrix Z; as mentioned above, the only sensical value of type is GLOBAL_TO_LOCAL, but the AxyType argument was kept for

consistency.

void Axy(*T alpha*, *Matrix*<*T*> &*Z*, int *i*, int *j*)

If the interface was previously attached in the LOCAL_TO_GLOBAL direction, then the matrix αZ will be added onto the associated distributed matrix starting at the (i, j) global index; otherwise α times the submatrix of the associated distributed matrix, which starts at index (i, j) and is of the same size as *Z*, will be added onto *Z*.

void Axy(*T alpha*, **const** *Matrix*<*T*> &*Z*, int *i*, int *j*)

Same as above, but since *Z* is unmodifiable, the attachment must have been in the LOCAL_TO_GLOBAL direction.

void Detach()

All processes collectively finish handling each others requests and then detach from the associated distributed matrix.

BASIC LINEAR ALGEBRA

This chapter describes Elemental's support for basic linear algebra routines, such as matrix-matrix multiplication, triangular solves, and matrix-vector multiplication. Most of these routines have counterparts in the Basic Linear Algebra Subprograms (BLAS).

4.1 Level 1

The prototypes for the following routines can be found at [include/elemental/blas-like/decl.hpp](#), while the implementations are in [include/elemental/blas-like/level1/](#).

4.1.1 Adjoint

Note: This is not a standard BLAS routine, but it is BLAS-like.

$B := A^H$.

```
void Adjoint(const Matrix<T> &A, Matrix<T> &B)
```

```
void Adjoint(const DistMatrix<T, U, V> &A, DistMatrix<T, W, Z> &B)
```

4.1.2 Axy

Performs $Y := \alpha X + Y$ (hence the name *axy*).

```
void Axy(T alpha, const Matrix<T> &X, Matrix<T> &Y)
```

```
void Axy(T alpha, const DistMatrix<T, U1, V1> &X, DistMatrix<T, U2, V2> &Y)
```

4.1.3 Conjugate

Note: This is not a standard BLAS routine, but it is BLAS-like.

$B := \bar{A}$. For real datatypes, this is a no-op.

```
void Conjugate(Matrix<T> &A)
```

```
void Conjugate(DistMatrix<T, U, V> &A)
```

$B := \bar{A}$.

```
void Conjugate(const Matrix<T> &A, Matrix<T> &B)
```

```
void Conjugate(const DistMatrix<T, U, V> &A, DistMatrix<T, W, Z> &B)
```

4.1.4 Copy

Sets $Y := X$.

```
void Copy(const Matrix<T> &X, Matrix<T> &Y)
```

```
void Copy(const DistMatrix<T, U, V> &A, DistMatrix<T, W, Z> &B)
```

4.1.5 DiagonalScale

Note: This is not a standard BLAS routine, but it is BLAS-like.

Performs either $X := \text{op}(D)X$ or $X := X\text{op}(D)$, where $\text{op}(D)$ equals $D = D^T$, or $D^H = \bar{D}$, where $D = \text{diag}(d)$ and d is a column vector.

```
void DiagonalScale(LeftOrRight side, Orientation orientation, const Matrix<T> &d, Matrix<T> &X)
```

```
void DiagonalScale(LeftOrRight side, Orientation orientation, const DistMatrix<T, U, V> &d, DistMatrix<T, W, Z> &X)
```

4.1.6 DiagonalScaleTrapezoid

Note: This is not a standard BLAS routine, but it is BLAS-like.

Performs either $A := \text{op}(D)A$ or $A := A\text{op}(D)$, where A is trapezoidal (upper or lower with the boundary diagonal of given offset), $\text{op}(D)$ equals $D = D^T$, or $D^H = \bar{D}$, where $D = \text{diag}(d)$ and d is a column vector.

```
void DiagonalScaleTrapezoid(LeftOrRight side, UpperOrLower uplo, Orientation orientation, const Matrix<T> &d, Matrix<T> &A, int offset = 0)
```

```
void DiagonalScaleTrapezoid(LeftOrRight side, UpperOrLower uplo, Orientation orientation, const DistMatrix<T, U, V> &d, DistMatrix<T, W, Z> &A, int offset = 0)
```

4.1.7 DiagonalSolve

Note: This is not a standard BLAS routine, but it is BLAS-like.

Performs either $X := \text{op}(D)^{-1}X$ or $X := X\text{op}(D)^{-1}$, where $D = \text{diag}(d)$ and d is a column vector.

```
void DiagonalSolve(LeftOrRight side, Orientation orientation, const Matrix<F> &d, Matrix<F> &X, bool checkIfSingular = false)
```

```
void DiagonalSolve(LeftOrRight side, Orientation orientation, const DistMatrix<F, U, V>
                  &d, DistMatrix<F, W, Z> &X, bool checkIfSingular = false)
```

4.1.8 Dot

Returns $(x, y) = x^H y$. x and y are both allowed to be stored as column or row vectors, but will be interpreted as column vectors.

```
T Dot(const Matrix<T> &x, const Matrix<T> &y)
```

```
T Dot(const DistMatrix<T, U, V> &x, const DistMatrix<T, U, V> &y)
```

4.1.9 Dotc

Same as Dot. This routine name is provided since it is the usual BLAS naming convention.

```
T Dotc(const Matrix<T> &x, const Matrix<T> &y)
```

```
T Dotc(const DistMatrix<T, U, V> &x, const DistMatrix<T, U, V> &y)
```

4.1.10 Dotu

Returns $x^T y$, which is **not** an inner product.

```
T Dotu(const Matrix<T> &x, const Matrix<T> &y)
```

```
T Dotu(const DistMatrix<T, U, V> &x, const DistMatrix<T, U, V> &y)
```

4.1.11 EntrywiseMap

```
void EntrywiseMap(Matrix<T> &A, Function func)
```

```
void EntrywiseMap(DistMatrix<T, U, V> &A, Function func)
```

```
void EntrywiseMap(BlockDistMatrix<T, U, V> &A, Function func)
```

Replaces each entry of the passed in matrix with a specified function of the existing entry. *func* will typically be a lambda function which accepts a single argument of type T and returns a value of type T .

4.1.12 Hadamard

Note: This is not a standard BLAS routine, but it is BLAS-like.

The Hadamard product of two $m \times n$ matrices A and B is given entrywise by $\alpha_{i,j}\beta_{i,j}$ and denoted by $C = A \circ B$.

```
void Hadamard(const Matrix<F> &A, const Matrix<F> &B, Matrix<F> &C)
```

```
void Hadamard(const DistMatrix<F, U, V> &A, const DistMatrix<F, U, V> &B, DistMa-
              trix<F, U, V> &C)
```

4.1.13 HilbertSchmidt

Note: This is not a standard BLAS routine, but it is BLAS-like.

The Hilbert-Schmidt inner-product of two $m \times n$ matrices A and B is $\text{tr}(A^H B)$.

`F HilbertSchmidt(const Matrix<F> &A, const Matrix<F> &B)`

`F HilbertSchmidt(const DistMatrix<F, U, V> &A, const DistMatrix<F, U, V> &B)`

4.1.14 MakeTrapezoidal

Note: This is not a standard BLAS routine, but it is BLAS-like.

Sets all entries outside of the specified trapezoidal submatrix to zero. Whether or not the trapezoid is upper or lower (analogous to an upper or lower-triangular matrix) is determined by the `uplo` parameter, and the last diagonal is defined with the `offset` integer.

`void MakeTrapezoidal(UpperOrLower uplo, Matrix<T> &A, int offset = 0)`

`void MakeTrapezoidal(UpperOrLower uplo, DistMatrix<T, U, V> &A, int offset = 0)`

4.1.15 Nrm2

Returns $\|x\|_2 = \sqrt{(x, x)} = \sqrt{x^H x}$. As with most other routines, even if x is stored as a row vector, it will be interpreted as a column vector.

`Base<F> Nrm2(const Matrix<F> &x)`

`Base<F> Nrm2(const DistMatrix<F> &x)`

4.1.16 Scale

$X := \alpha X$.

`void Scale(T alpha, Matrix<T> &X)`

`void Scale(T alpha, DistMatrix<T, U, V> &X)`

4.1.17 ScaleTrapezoid

Note: This is not a standard BLAS routine, but it is BLAS-like.

Scales the entries within the specified trapezoid of a general matrix. The parameter conventions follow those of `MakeTrapezoidal` described above.

`void ScaleTrapezoid(T alpha, UpperOrLower uplo, Matrix<T> &A, int offset = 0)`

`void ScaleTrapezoid(T alpha, UpperOrLower uplo, DistMatrix<T, U, V> &A, int offset = 0)`

4.1.18 Transpose

Note: This is not a standard BLAS routine, but it is BLAS-like.

$B := A^T$ or $B := A^H$.

void Transpose(const *Matrix*<T> &A, *Matrix*<T> &B, bool *conjugate* = false)

void Transpose(const *DistMatrix*<T, U, V> &A, *DistMatrix*<T, W, Z> &B)

4.1.19 Zero

Note: This is not a standard BLAS routine, but it is BLAS-like.

Sets all of the entries of the input matrix to zero.

void Zero(*Matrix*<T> &A)

void Zero(*DistMatrix*<T, U, V> &A)

4.1.20 SetDiagonal

Note: This is not a standard BLAS routine.

Sets all of the diagonal entries of a matrix to a given value.

void SetDiagonal(*Matrix*<T> &A, T *alpha*)

void SetDiagonal(*DistMatrix*<T, U, V> &A, T *alpha*)

void SetDiagonal(*Matrix*<T> &A, T *alpha*, int *offset* = 0, *LeftOrRight side* = LEFT)

void SetDiagonal(*DistMatrix*<T, U, V> &A, T *alpha*, int *offset* = 0, *LeftOrRight side* = LEFT)

4.1.21 Swap

void Swap(*Orientation orientation*, *Matrix*<T> &A, *Matrix*<T> &B)

void Swap(*Orientation orientation*, *DistMatrix*<T, U1, V1> &A, *DistMatrix*<T, U2, V2> &B)

Replace *A* and *B* with each other, their transpose, or their adjoint.

void RowSwap(*Matrix*<T> &A, int *to*, int *from*)

void RowSwap(*DistMatrix*<T, U, V> &A, int *to*, int *from*)

Swap rows *to* and *from* in the matrix.

void ColumnSwap(*Matrix*<T> &A, int *to*, int *from*)

void ColumnSwap(*DistMatrix*<T, U, V> &A, int *to*, int *from*)

Swap columns *to* and *from* in the matrix.

void SymmetricSwap(*UpperOrLower uplo*, *Matrix*<T> &A, int *to*, int *from*, bool *conjugate* = false)

```
void SymmetricSwap(UpperOrLower uplo, DistMatrix<T> &A, int to, int from, bool conjugate = false)
```

Symmetrically permute the *to* and *from* degrees of freedom within the implicitly symmetric (Hermitian) matrix *A* which stores its data in the specified triangle.

4.1.22 QuasiDiagonalScale

Note: This is not a standard BLAS routine.

```
void QuasiDiagonalScale(LeftOrRight side, UpperOrLower uplo, const Matrix<FMain> &d, const Matrix<F> &dSub, Matrix<F> &X, bool conjugate = false)
```

```
void QuasiDiagonalScale(LeftOrRight side, UpperOrLower uplo, const DistMatrix<FMain, U, V> &d, const DistMatrix<F, U, V> &dSub, DistMatrix<F> &X, bool conjugate = false)
```

Apply a symmetric (Hermitian) quasi-diagonal matrix to the matrix *X*.

4.1.23 QuasiDiagonalSolve

Note: This is not a standard BLAS routine.

```
void QuasiDiagonalSolve(LeftOrRight side, UpperOrLower uplo, const Matrix<FMain> &d, const Matrix<F> &dSub, Matrix<F> &X, bool conjugate = false)
```

```
void QuasiDiagonalSolve(LeftOrRight side, UpperOrLower uplo, const DistMatrix<FMain, U, V> &d, const DistMatrix<F, U, V> &dSub, DistMatrix<F> &X, bool conjugate = false)
```

Apply the inverse of a symmetric (Hermitian) quasi-diagonal matrix to the matrix *X*.

4.1.24 Symmetric2x2Scale

Note: This is not a standard BLAS routine.

```
void Symmetric2x2Scale(LeftOrRight side, UpperOrLower uplo, const Matrix<F> &D, Matrix<F> &A, bool conjugate = false)
```

```
void Symmetric2x2Scale(LeftOrRight side, UpperOrLower uplo, const DistMatrix<F, STAR, STAR> &D, DistMatrix<F> &A, bool conjugate = false)
```

Apply a 2x2 symmetric (Hermitian) matrix to the matrix *A*.

4.1.25 Symmetric2x2Solve

Note: This is not a standard BLAS routine.

```
void Symmetric2x2Solve(LeftOrRight side, UpperOrLower uplo, const Matrix<F> &D, Matrix<F> &A, bool conjugate = false)
```

void Symmetric2x2Solve(*LeftOrRight* side, *UpperOrLower* uplo, const *DistMatrix*<F,
STAR, *STAR*> &D, *DistMatrix*<F> &A, bool *conjugate* = false)
 Apply the inverse of a 2x2 symmetric (Hermitian) matrix to the matrix A.

4.1.26 UpdateDiagonal

Note: This is not a standard BLAS routine.

Adds a given value to the diagonal of a matrix.

void UpdateDiagonal (*Matrix*<T> &A, T *alpha*)

void UpdateDiagonal (*DistMatrix*<T, U, V> &A, T *alpha*)

void UpdateDiagonal (*Matrix*<T> &A, T *alpha*, int *offset* = 0, *LeftOrRight* side = LEFT)

void UpdateDiagonal (*DistMatrix*<T, U, V> &A, T *alpha*, int *offset* = 0, *LeftOrRight* side =
 LEFT)

4.2 Level 2

The prototypes for the following routines can be found at [include/elemental/blas-like/decl.hpp](#), while the implementations are in [include/elemental/blas-like/level2/](#).

4.2.1 Gemv

General matrix-vector multiply: $y := \alpha \text{op}(A)x + \beta y$, where $\text{op}(A)$ can be A , A^T , or A^H . Whether or not x and y are stored as row vectors, they will be interpreted as column vectors.

void Gemv (*Orientation* orientation, T *alpha*, const *Matrix*<T> &A, const *Matrix*<T> &x, T
beta, *Matrix*<T> &y)

void Gemv (*Orientation* orientation, T *alpha*, const *DistMatrix*<T> &A, const *DistMa-*
trix<T> &x, T *beta*, *DistMatrix*<T> &y)

4.2.2 Ger

General rank-one update: $A := \alpha xy^H + A$. x and y are free to be stored as either row or column vectors, but they will be interpreted as column vectors.

void Ger (T *alpha*, const *Matrix*<T> &x, const *Matrix*<T> &y, *Matrix*<T> &A)

void Ger (T *alpha*, const *DistMatrix*<T> &x, const *DistMatrix*<T> &y, *DistMatrix*<T>
 &A)

4.2.3 Gerc

This is the same as *Ger()*, but the name is provided because it exists in the BLAS.

void Gerc (T *alpha*, const *Matrix*<T> &x, const *Matrix*<T> &y, *Matrix*<T> &A)

void Gerc (T *alpha*, const *DistMatrix*<T> &x, const *DistMatrix*<T> &y, *DistMatrix*<T>
 &A)

4.2.4 Geru

General rank-one update (unconjugated): $A := \alpha xy^T + A$. x and y are free to be stored as either row or column vectors, but they will be interpreted as column vectors.

```
void Geru(T alpha, const Matrix<T> &x, const Matrix<T> &y, Matrix<T> &A)
```

```
void Geru(T alpha, const DistMatrix<T> &x, const DistMatrix<T> &y, DistMatrix<T> &A)
```

4.2.5 Hemv

Hermitian matrix-vector multiply: $y := \alpha Ax + \beta y$, where A is Hermitian.

```
void Hemv(UpperOrLower uplo, T alpha, const Matrix<T> &A, const Matrix<T> &x, T beta, Matrix<T> &y)
```

```
void Hemv(UpperOrLower uplo, T alpha, const DistMatrix<T> &A, const DistMatrix<T> &x, T beta, DistMatrix<T> &y)
```

Please see `SetLocalSymvBlocksize<T>()` and `LocalSymvBlocksize<T>()` in the *Tuning parameters* section for information on tuning the distributed `Hemv()`.

4.2.6 Her

Hermitian rank-one update: implicitly performs $A := \alpha xx^H + A$, where only the triangle of A specified by `uplo` is updated.

```
void Her(UpperOrLower uplo, T alpha, const Matrix<T> &x, Matrix<T> &A)
```

```
void Her(UpperOrLower uplo, T alpha, const DistMatrix<T> &x, DistMatrix<T> &A)
```

4.2.7 Her2

Hermitian rank-two update: implicitly performs $A := \alpha(xy^H + yx^H) + A$, where only the triangle of A specified by `uplo` is updated.

```
void Her2(UpperOrLower uplo, T alpha, const Matrix<T> &x, const Matrix<T> &y, Matrix<T> &A)
```

```
void Her2(UpperOrLower uplo, T alpha, const DistMatrix<T> &x, const DistMatrix<T> &y, DistMatrix<T> &A)
```

4.2.8 QuasiTrsv

Quasi-triangular solve with a vector: computes $x := \text{op}(A)^{-1}x$, where `op(A)` is either A , A^T , or A^H , and A is treated as either a lower or upper quasi-triangular matrix, depending upon `uplo`.

Note that the term *quasi-triangular* is in the context of real Schur decompositions, which produce triangular matrices with mixes of 1×1 and 2×2 diagonal blocks.

Note: There is no corresponding BLAS routine, but it is a natural extension of `Trsv`.

```
void QuasiTrsv(UpperOrLower uplo, Orientation orientation, const Matrix<F> &A, Matrix<F> &x, bool checkIfSingular = false)
```

```
void QuasiTrsv(UpperOrLower uplo, Orientation orientation, const DistMatrix<F> &A, DistMatrix<F> &x, bool checkIfSingular = false)
```

4.2.9 Symv

Symmetric matrix-vector multiply: $y := \alpha Ax + \beta y$, where A is symmetric.

```
void Symv(UpperOrLower uplo, T alpha, const Matrix<T> &A, const Matrix<T> &x, T beta, Matrix<T> &y, bool conjugate = false)
```

```
void Symv(UpperOrLower uplo, T alpha, const DistMatrix<T> &A, const DistMatrix<T> &x, T beta, DistMatrix<T> &y, bool conjugate = false)
```

Please see *SetLocalSymvBlocksize*<T>() and *LocalSymvBlocksize*<T>() in the *Tuning parameters* section for information on tuning the distributed *Symv*() .

4.2.10 Syr

Symmetric rank-one update: implicitly performs $A := \alpha xx^T + A$, where only the triangle of A specified by *uplo* is updated.

```
void Syr(UpperOrLower uplo, T alpha, const Matrix<T> &x, Matrix<T> &A, bool conjugate = false)
```

```
void Syr(UpperOrLower uplo, T alpha, const DistMatrix<T> &x, DistMatrix<T> &A, bool conjugate = false)
```

4.2.11 Syr2

Symmetric rank-two update: implicitly performs $A := \alpha(xy^T + yx^T) + A$, where only the triangle of A specified by *uplo* is updated.

```
void Syr2(UpperOrLower uplo, T alpha, const Matrix<T> &x, const Matrix<T> &y, Matrix<T> &A, bool conjugate = false)
```

```
void Syr2(UpperOrLower uplo, T alpha, const DistMatrix<T> &x, const DistMatrix<T> &y, DistMatrix<T> &A, bool conjugate = false)
```

4.2.12 Trmv

Not yet written. Please call *Trmm*() for now.

4.2.13 Trsv

Triangular solve with a vector: computes $x := \text{op}(A)^{-1}x$, where $\text{op}(A)$ is either A , A^T , or A^H , and A is treated as either a lower or upper triangular matrix, depending upon *uplo*. A can also be treated as implicitly having a unit-diagonal if *diag* is set to UNIT.

```
void Trsv(UpperOrLower uplo, Orientation orientation, UnitOrNonUnit diag, const Matrix<F> &A, Matrix<F> &x)
```

```
void Trsv(UpperOrLower uplo, Orientation orientation, UnitOrNonUnit diag, const DistMatrix<F> &A, DistMatrix<F> &x)
```

4.3 Level 3

The prototypes for the following routines can be found at `include/elemental/blas-like/decl.hpp`, while the implementations are in `include/elemental/blas-like/level3/`.

4.3.1 Gemm

General matrix-matrix multiplication: updates $C := \alpha A^{\#} B^{\#} + \beta C$, where $M^{\#}$ and $M^{\#}$ are individually defined to be one of $\{M, M^T, M^H\}$.

```
void Gemm(Orientation orientationOfA, Orientation orientationOfB, T alpha, const Matrix<T> &A, const Matrix<T> &B, T beta, Matrix<T> &C)
```

```
void Gemm(Orientation orientationOfA, Orientation orientationOfB, T alpha, const DistMatrix<T> &A, const DistMatrix<T> &B, T beta, DistMatrix<T> &C)
```

4.3.2 Hemm

Hermitian matrix-matrix multiplication: updates $C := \alpha AB + \beta C$, or $C := \alpha BA + \beta C$, depending upon whether *side* is set to LEFT or RIGHT, respectively. In both of these types of updates, A is implicitly Hermitian and only the triangle specified by *uplo* is accessed.

```
void Hemm(LeftOrRight side, UpperOrLower uplo, T alpha, const Matrix<T> &A, const Matrix<T> &B, T beta, Matrix<T> &C)
```

```
void Hemm(LeftOrRight side, UpperOrLower uplo, T alpha, const DistMatrix<T> &A, const DistMatrix<T> &B, T beta, DistMatrix<T> &C)
```

4.3.3 Her2k

Hermitian rank-2K update: updates $C := \alpha(AB^H + BA^H) + \beta C$, or $C := \alpha(A^H B + B^H A) + \beta C$, depending upon whether *orientation* is set to NORMAL or ADJOINT, respectively. Only the triangle of C specified by the *uplo* parameter is modified.

```
void Her2k(UpperOrLower uplo, Orientation orientation, T alpha, const Matrix<T> &A, const Matrix<T> &B, T beta, Matrix<T> &C)
```

```
void Her2k(UpperOrLower uplo, Orientation orientation, T alpha, const DistMatrix<T> &A, const DistMatrix<T> &B, T beta, DistMatrix<T> &C)
```

Please see `SetLocalTrr2kBlocksize<T>()` and `LocalTrr2kBlocksize<T>()` in the *Tuning parameters* section for information on tuning the distributed `Her2k()`.

4.3.4 Herk

Hermitian rank-K update: updates $C := \alpha AA^H + \beta C$, or $C := \alpha A^H A + \beta C$, depending upon whether *orientation* is set to NORMAL or ADJOINT, respectively. Only the triangle of C specified by the *uplo* parameter is modified.

```
void Herk(UpperOrLower uplo, Orientation orientation, T alpha, const Matrix<T> &A, T
         beta, Matrix<T> &C)
```

```
void Herk(UpperOrLower uplo, Orientation orientation, T alpha, const DistMatrix<T> &A,
         T beta, DistMatrix<T> &C)
```

Please see `SetLocalTrrkBlocksize<T>()` and `LocalTrrkBlocksize<T>()` in the *Tuning parameters* section for information on tuning the distributed `Herk()`.

4.3.5 Multi-shift QuasiTrsm

Solve for X in the linear system

$$T^{\#}X - XD^{\#} = Y$$

or

$$XT^{\#} - D^{\#}X = Y$$

where T is *quasi-triangular*, D is diagonal, and $A^{\#}$ is defined to be one of $\{A, A^T, A^H\}$. The data movement requires almost no modification from that of `QuasiTrsm()`.

Note that the term *quasi-triangular* is in the context of real Schur decompositions, which produce triangular matrices with mixes of 1×1 and 2×2 diagonal blocks.

Note: There is no corresponding BLAS routine, but it is a natural extension of `Trsm`.

```
void MultiShiftQuasiTrsm(LeftOrRight side, UpperOrLower uplo, Orientation orientation,
                        F alpha, const Matrix<F> &T, const Matrix<F> &shifts, Ma-
                        trix<F> &X)
```

```
void MultiShiftQuasiTrsm(LeftOrRight side, UpperOrLower uplo, Orientation orientation,
                        F alpha, const DistMatrix<F> &T, const DistMatrix<F, VR,
                        STAR> &shifts, DistMatrix<F> &X)
```

Overwrite the columns of X with the solutions to the shifted linear systems.

```
void MultiShiftQuasiTrsm(LeftOrRight side, UpperOrLower uplo, Orientation orientation,
                        Complex<Real> alpha, const Matrix<Real> &T, const Ma-
                        trix<Complex<Real>> &shifts, Matrix<Real> &XReal, Ma-
                        trix<Real> &XImag)
```

```
void MultiShiftQuasiTrsm(LeftOrRight side, UpperOrLower uplo, Orientation orientation,
                        Complex<Real> alpha, const DistMatrix<Real> &T, const
                        DistMatrix<Complex<Real>, VR, STAR> &shifts, DistMa-
                        trix<Real> &XReal, DistMatrix<Real> &XImag)
```

Overwrite the columns of the real and imaginary parts of X with the solutions to the shifted linear systems.

4.3.6 Multi-shift Trsm

Solve for X in the linear system

$$T^{\#}X - XD^{\#} = Y$$

or

$$XT^{\#} - D^{\#}X = Y$$

where T is triangular, D is diagonal, and $A^{\#}$ is defined to be one of $\{A, A^T, A^H\}$. The data movement requires almost no modification from that of `Trsm()`.

Note: There is no corresponding BLAS routine, but it is a natural modification of `Trsm`.

```
void MultiShiftTrsm(LeftOrRight side, UpperOrLower uplo, Orientation orientation, F alpha,
  const Matrix<F> &T, const Matrix<F> &shifts, Matrix<F> &X)
```

```
void MultiShiftTrsm(LeftOrRight side, UpperOrLower uplo, Orientation orientation, F alpha,
  const DistMatrix<F> &T, const DistMatrix<F, VR, STAR> &shifts, DistMatrix<F> &X)
```

Overwrite the columns of X with the solutions to the shifted linear systems.

```
void MultiShiftTrsm(LeftOrRight side, UpperOrLower uplo, Orientation orientation,
  Complex<Real> alpha, const Matrix<Real> &T, const Matrix<Complex<Real>> &shifts,
  Matrix<Real> &XReal, Matrix<Real> &XImag)
```

```
void MultiShiftTrsm(LeftOrRight side, UpperOrLower uplo, Orientation orientation,
  Complex<Real> alpha, const DistMatrix<Real> &T, const DistMatrix<Complex<Real>,
  VR, STAR> &shifts, DistMatrix<Real> &XReal, DistMatrix<Real> &XImag)
```

Overwrite the columns of the real and imaginary parts of X with the solutions to the shifted linear systems.

4.3.7 QuasiTrsm

Solve for X in the linear system

$$T^{\#}X = Y$$

or

$$XT^{\#} = Y$$

where T is *quasi-triangular* and $A^{\#}$ is defined to be one of $\{A, A^T, A^H\}$. The algorithm is very similar to that of `Trsm()`.

Note that the term *quasi-triangular* is in the context of real Schur decompositions, which produce triangular matrices with mixes of 1×1 and 2×2 diagonal blocks.

Note: There is no corresponding BLAS routine, but it is a natural extension of `Trsm`.

```
void QuasiTrsm(LeftOrRight side, UpperOrLower uplo, Orientation orientation, F alpha,
  const Matrix<F> &T, const Matrix<F> &shifts, Matrix<F> &X, bool
  checkIfSingular = false)
```

```
void QuasiTrsm(LeftOrRight side, UpperOrLower uplo, Orientation orientation, F alpha,
  const DistMatrix<F> &T, const DistMatrix<F, VR, STAR> &shifts, DistMatrix<F> &X,
  bool checkIfSingular = false)
```

Overwrite the columns of X with the solutions to the shifted linear systems.

4.3.8 Symm

Symmetric matrix-matrix multiplication: updates $C := \alpha AB + \beta C$, or $C := \alpha BA + \beta C$, depending upon whether *side* is set to LEFT or RIGHT, respectively. In both of these types of updates, *A* is implicitly symmetric and only the triangle specified by *uplo* is accessed.

```
void Symm(LeftOrRight side, UpperOrLower uplo, T alpha, const Matrix<T> &A, const Matrix<T> &B, T beta, Matrix<T> &C, bool conjugate = false)
```

```
void Symm(LeftOrRight side, UpperOrLower uplo, T alpha, const DistMatrix<T> &A, const DistMatrix<T> &B, T beta, DistMatrix<T> &C, bool conjugate = false)
```

4.3.9 Syr2k

Symmetric rank-2K update: updates $C := \alpha(AB^T + BA^T) + \beta C$, or $C := \alpha(A^T B + B^T A) + \beta C$, depending upon whether *orientation* is set to NORMAL or TRANSPOSE, respectively. Only the triangle of *C* specified by the *uplo* parameter is modified.

```
void Syr2k(UpperOrLower uplo, Orientation orientation, T alpha, const Matrix<T> &A, const Matrix<T> &B, T beta, Matrix<T> &C)
```

```
void Syr2k(UpperOrLower uplo, Orientation orientation, T alpha, const DistMatrix<T> &A, const DistMatrix<T> &B, T beta, DistMatrix<T> &C)
```

Please see `SetLocalTrr2kBlocksize<T>()` and `LocalTrr2kBlocksize<T>()` in the *Tuning parameters* section for information on tuning the distributed `Syr2k()`.

4.3.10 Syrk

Symmetric rank-K update: updates $C := \alpha AA^T + \beta C$, or $C := \alpha A^T A + \beta C$, depending upon whether *orientation* is set to NORMAL or TRANSPOSE, respectively. Only the triangle of *C* specified by the *uplo* parameter is modified.

```
void Syrk(UpperOrLower uplo, Orientation orientation, T alpha, const Matrix<T> &A, T beta, Matrix<T> &C)
```

```
void Syrk(UpperOrLower uplo, Orientation orientation, T alpha, const DistMatrix<T> &A, T beta, DistMatrix<T> &C)
```

Please see `SetLocalTrrkBlocksize<T>()` and `LocalTrrkBlocksize<T>()` in the *Tuning parameters* section for information on tuning the distributed `Syrk()`.

4.3.11 Trmm

Triangular matrix-matrix multiplication: performs $C := \alpha A^\# B$, or $C := \alpha B A^\#$, depending upon whether *side* was chosen to be LEFT or RIGHT, respectively. Whether *A* is treated as lower or upper triangular is determined by *uplo*, and $A^\#$ is defined to be one of $\{A, A^T, A^H\}$ (and *diag* determines whether *A* is treated as unit-diagonal or not).

```
void Trmm(LeftOrRight side, UpperOrLower uplo, Orientation orientation, UnitOrNonUnit diag, T alpha, const Matrix<T> &A, Matrix<T> &B)
```

```
void Trmm(LeftOrRight side, UpperOrLower uplo, Orientation orientation, UnitOrNonUnit diag, T alpha, const DistMatrix<T> &A, DistMatrix<T> &B)
```

4.3.12 Trr2k

Triangular rank-2k update: performs $E := \alpha(A^\#B^\# + C^\diamond D^\triangle) + \beta E$, where only the triangle of E specified by *uplo* is modified, and the orientation of each input matrix, e.g., $A^\# \in \{A, A^T, A^H\}$, is determined by *orientationOfX* for each $X \in \{A, B, C, D\}$.

Note: There is no corresponding BLAS routine, but it is a natural generalization of “symmetric” and “Hermitian” updates.

```
void Trr2k(UpperOrLower uplo, Orientation orientationOfA, Orientation orientationOfB,
           Orientation orientationOfC, Orientation orientationOfD, T alpha, const Matrix<T> &A,
           const Matrix<T> &B, const Matrix<T> &C, const Matrix<T> &D, T beta, Matrix<T> &E)
```

```
void Trr2k(UpperOrLower uplo, Orientation orientationOfA, Orientation orientationOfB,
           Orientation orientationOfC, Orientation orientationOfD, T alpha, const DistMatrix<T> &A,
           const DistMatrix<T> &B, const DistMatrix<T> &C, const DistMatrix<T> &D, T beta, DistMatrix<T> &E)
```

4.3.13 Trrk

Triangular rank-k update: performs $C := \alpha A^\# B^\# + \beta C$, where only the triangle of C specified by *uplo* is modified, and the orientations $A^\#$ and $B^\#$ are determined by *orientationOfA* and *orientationOfB*, respectively.

Note: There is no corresponding BLAS routine, but this type of update is frequently encountered, even in serial. For instance, the symmetric rank-k update performed during an LDL factorization is symmetric but one of the two update matrices is scaled by D .

```
void Trrk(UpperOrLower uplo, Orientation orientationOfA, Orientation orientationOfB, T alpha,
           const Matrix<T> &A, const Matrix<T> &B, T beta, Matrix<T> &C)
```

```
void Trrk(UpperOrLower uplo, Orientation orientationOfA, Orientation orientationOfB, T alpha,
           const DistMatrix<T> &A, const DistMatrix<T> &B, T beta, DistMatrix<T> &C)
```

4.3.14 Trtrmm

Note: This routine loosely corresponds with the LAPACK routines ?lauum.

Symmetric/Hermitian triangular matrix-matrix multiply: performs $L := L^T L$, $L := L^H L$, $U := U U^T$, or $U := U U^H$, depending upon the choice of the *orientation* and *uplo* parameters.

```
void Trtrmm(Orientation orientation, UpperOrLower uplo, Matrix<T> &A)
```

```
void Trtrmm(Orientation orientation, UpperOrLower uplo, DistMatrix<T> &A)
```

4.3.15 Trdtrmm

Note: This is a modification of *Trtrmm* for LDL factorizations.

Symmetric/Hermitian triangular matrix-matrix multiply (with diagonal scaling): performs $L := L^T D^{-1} L$, $L := L^H D^{-1} L$, $U := U D^{-1} U^T$, or $U := U D^{-1} U^H$, depending upon the choice of the *orientation* and *uplo* parameters. Note that L and U are unit-diagonal and their diagonal is overwritten with D .

```
void Trdtrmm(Orientation orientation, UpperOrLower uplo, Matrix<F> &A)
```

```
void Trdtrmm(Orientation orientation, UpperOrLower uplo, DistMatrix<F> &A)
```

4.3.16 Trsm

Triangular solve with multiple right-hand sides: performs $C := \alpha A^{-\#} B$, or $C := \alpha B A^{-\#}$, depending upon whether *side* was chosen to be LEFT or RIGHT, respectively. Whether A is treated as lower or upper triangular is determined by *uplo*, and $A^{-\#}$ can be A^{-1} , A^{-T} , or A^{-H} (and *diag* determines whether A is treated as unit-diagonal or not).

```
void Trsm(LeftOrRight side, UpperOrLower uplo, Orientation orientation, UnitOrNonUnit
         diag, F alpha, const Matrix<F> &A, Matrix<F> &B)
```

```
void Trsm(LeftOrRight side, UpperOrLower uplo, Orientation orientation, UnitOrNonUnit
         diag, F alpha, const DistMatrix<F> &A, DistMatrix<F> &B)
```

4.3.17 Trstrm

Performs a triangular solve against a triangular matrix. Only the Left Lower Normal option is currently supported.

```
void Trstrm(LeftOrRight side, UpperOrLower uplo, Orientation orientation, UnitOrNonUnit
         diag, F alpha, const Matrix<F> &A, Matrix<F> &X, bool checkIfSingular =
         true)
```

```
void Trstrm(LeftOrRight side, UpperOrLower uplo, Orientation orientation, UnitOrNonUnit
         diag, F alpha, const DistMatrix<F> &A, DistMatrix<F> &X, bool checkIfSin-
         gular = true)
```

4.3.18 Two-sided Trmm

Performs a two-sided triangular multiplication with multiple right-hand sides which preserves the symmetry of the input matrix, either $A := L^H A L$ or $A := U A U^H$.

```
void TwoSidedTrmm(UpperOrLower uplo, UnitOrNonUnit diag, Matrix<T> &A, const Ma-
         trix<T> &B)
```

```
void TwoSidedTrmm(UpperOrLower uplo, UnitOrNonUnit diag, DistMatrix<T> &A, const
         DistMatrix<T> &B)
```

4.3.19 Two-sided Trsm

Performs a two-sided triangular solves with multiple right-hand sides which preserves the symmetry of the input matrix, either $A := L^{-1} A L^{-H}$ or $A := U^{-H} A U^{-1}$.

```
void TwoSidedTrsm(UpperOrLower uplo, UnitOrNonUnit diag, Matrix<F> &A, const Ma-
         trix<F> &B)
```

```
void TwoSidedTrsm(UpperOrLower uplo, UnitOrNonUnit diag, DistMatrix<F> &A, const  
                 DistMatrix<F> &B)
```

4.4 Tuning parameters

The following tuning parameters have been exposed since they are system-dependent and can have a large impact on performance.

4.4.1 LocalSymvBlocksize

```
void SetLocalSymvBlocksize<T>(int blocksize)
```

Sets the local blocksize for the distributed *Symv()* routine for datatype T. It is set to 64 by default and is important for the reduction of a real symmetric matrix to symmetric tridiagonal form.

```
int LocalSymvBlocksize<T>()
```

Retrieves the local *Symv()* blocksize for datatype T.

4.4.2 LocalTrrkBlocksize

```
void SetLocalTrrkBlocksize<T>(int blocksize)
```

Sets the local blocksize for the distributed *LocalTrrk* routine for datatype T. It is set to 64 by default and is important for routines that perform distributed *Syrk()* or *Herk()* updates, e.g., Cholesky factorization.

```
int LocalTrrkBlocksize<T>()
```

Retrieves the local blocksize for the distributed *LocalTrrk* routine for datatype T.

4.4.3 LocalTrr2kBlocksize

```
void SetLocalTrr2kBlocksize<T>(int blocksize)
```

Sets the local blocksize for the distributed *LocalTrr2k* routine for datatype T. It is set to 64 by default and is important for routines that perform distributed *Syr2k()* or *Her2k()* updates, e.g., Householder tridiagonalization.

```
int LocalTrr2kBlocksize<T>()
```

Retrieves the local blocksize for the distributed *LocalTrr2k* routine for datatype T.

HIGH-LEVEL LINEAR ALGEBRA

This chapter describes all of the linear algebra operations which are not basic enough to fall within the BLAS (Basic Linear Algebra Subprograms) framework. In particular, algorithms which would traditionally have fallen into the domain of LAPACK (Linear Algebra PACKage), such as factorizations and matrix decompositions, are placed here.

5.1 Reduction to condensed form

5.1.1 Hermitian to tridiagonal

The currently best-known algorithms for computing eigenpairs of dense Hermitian matrices begin by performing a unitary similarity transformation which reduces the matrix to real symmetric tridiagonal form (usually through Householder transformations). This routine performs said reduction on a Hermitian matrix and stores the scaled Householder vectors in place of the introduced zeroes.

[Main source file](#)

[Subroutine implementations](#)

[Test driver](#)

class HermitianTridiagCtrl

HermitianTridiagApproach approach

GridOrder order

HermitianTridiagCtrl()

Sets *approach* to HERMITIAN_TRIDIAG_SQUARE and *order* to ROW_MAJOR.

Note: Please see the [Tuning parameters](#) section for extensive information on maximizing the performance of Householder tridiagonalization.

void HermitianTridiag(*UpperOrLower uplo*, *Matrix*<F> &A, *Matrix*<F> &t)

void HermitianTridiag(*UpperOrLower uplo*, *DistMatrix*<F> &A, *DistMatrix*<F, STAR, STAR> &t, **const** *HermitianTridiagCtrl* &ctrl = HermitianTridiagCtrl())

Overwrites the main and sub (super) diagonal of the real matrix *A* with its similar symmetric tridiagonal matrix and stores the scaled Householder vectors below (above) its

tridiagonal entries. Complex Hermitian reductions have the added complication of needing to also store the scalings for the Householder vectors (the scaling can be inferred since the Householder vectors must be unit length) if they are to be applied (in the column vector t).

```
void HermitianTridiag(UpperOrLower uplo, Matrix<F> &A)
```

```
void HermitianTridiag(UpperOrLower uplo, DistMatrix<F> &A, const HermitianTridiagCtrl &ctrl = HermitianTridiagCtrl())
```

Returns just the (appropriate triangle of the) resulting tridiagonal matrix.

hermitian_tridiag namespace

Implementations

```
void hermitian_tridiag::ApplyQ(LeftOrRight side, UpperOrLower uplo, Orientation orientation, const Matrix<F> &A, const Matrix<F> &t, Matrix<F> &B)
```

```
void hermitian_tridiag::ApplyQ(LeftOrRight side, UpperOrLower uplo, Orientation orientation, const DistMatrix<F> &A, const DistMatrix<F, MD, STAR> &t, DistMatrix<F> &B)
```

```
void hermitian_tridiag::ApplyQ(LeftOrRight side, UpperOrLower uplo, Orientation orientation, const DistMatrix<F> &A, const DistMatrix<F, STAR, STAR> &t, DistMatrix<F> &B)
```

Apply (from the left or right) the implicitly defined unitary matrix (or its adjoint) represented by the Householder transformations stored within the specified triangle of A and their scalings are stored in the vector t .

5.1.2 Square to Hessenberg

Main header file

Subroutine header files

Test driver

```
void Hessenberg(UpperOrLower uplo, Matrix<F> &A, Matrix<F> &t)
```

```
void Hessenberg(UpperOrLower uplo, DistMatrix<F> &A, DistMatrix<F, STAR, STAR> &t)
```

Returns the in-place reduction of the matrix A to lower-/upper-Hessenberg form. The vector t contains the scalings for the Householder reflectors, which are stored in the locations of the zeros that they introduced.

```
void Hessenberg(UpperOrLower uplo, Matrix<F> &A)
```

```
void Hessenberg(UpperOrLower uplo, DistMatrix<F> &A)
```

Returns just the Hessenberg matrix.

hessenberg namespace

TODO

```
void hessenberg::ApplyQ(UpperOrLower uplo, LeftOrRight side, Orientation orientation, const Matrix<F> &A, const Matrix<F> &t, Matrix<F> &H)
```

```
void hessenberg::ApplyQ(UpperOrLower uplo, LeftOrRight side, Orientation orientation,
    const DistMatrix<F> &A, const DistMatrix<F, MD, STAR>
    &t, DistMatrix<F> &H)
```

```
void hessenberg::ApplyQ(UpperOrLower uplo, LeftOrRight side, Orientation orientation,
    const DistMatrix<F> &A, const DistMatrix<F, STAR, STAR>
    &t, DistMatrix<F> &H)
```

5.1.3 General to bidiagonal

Reduces a general fully-populated $m \times n$ matrix to bidiagonal form through two-sided Householder transformations; when the $m \geq n$, the result is upper bidiagonal, otherwise it is lower bidiagonal. This routine is most commonly used as a preprocessing step in computing the SVD of a general matrix.

Main header file

Subroutine implementations

Test driver

```
void Bidiag(Matrix<F> &A, Matrix<F> &tP, Matrix<F> &tQ)
```

```
void Bidiag(DistMatrix<F> &A, DistMatrix<F, STAR, STAR> &tP, DistMatrix<F, STAR,
    STAR> &tQ)
```

Overwrites the main and sub (or super) diagonal of the real matrix A with the resulting bidiagonal matrix and stores the scaled Householder vectors in the remainder of the matrix. The complex case must also store the scalings of the Householder transformations (in tP and tQ) if they are to be applied.

```
void Bidiag(Matrix<F> &A)
```

```
void Bidiag(DistMatrix<F> &A)
```

Returns just the resulting bidiagonal matrix.

bidiag namespace

TODO

```
void bidiag::ApplyQ(LeftOrRight side, Orientation orientation, const Matrix<F> &A,
    const Matrix<F> &t, Matrix<F> &B)
```

```
void bidiag::ApplyQ(LeftOrRight side, Orientation orientation, const DistMatrix<F> &A,
    const DistMatrix<F, MD, STAR> &t, DistMatrix<F> &B)
```

```
void bidiag::ApplyQ(LeftOrRight side, Orientation orientation, const DistMatrix<F> &A,
    const DistMatrix<F, STAR, STAR> &t, DistMatrix<F> &B)
```

```
void bidiag::ApplyP(LeftOrRight side, Orientation orientation, const Matrix<F> &A,
    const Matrix<F> &t, Matrix<F> &B)
```

```
void bidiag::ApplyP(LeftOrRight side, Orientation orientation, const DistMatrix<F> &A,
    const DistMatrix<F, MD, STAR> &t, DistMatrix<F> &B)
```

```
void bidiag::ApplyP(LeftOrRight side, Orientation orientation, const DistMatrix<F> &A,
    const DistMatrix<F, STAR, STAR> &t, DistMatrix<F> &B)
```

5.2 Matrix decompositions

5.2.1 Hermitian eigensolver

Implementation

Test driver

Example driver

Elemental provides a collection of routines for both full and partial solution of the Hermitian eigenvalue problem

$$AZ = Z\Lambda,$$

where A is the given Hermitian matrix, and unitary Z and real diagonal Λ are sought. In particular, with the eigenvalues and corresponding eigenpairs labeled in non-decreasing order, the three basic modes are:

1. Compute all eigenvalues or eigenpairs, $\{\lambda_i\}_{i=0}^{n-1}$ or $\{(z_i, \lambda_i)\}_{i=0}^{n-1}$.
2. Compute the eigenvalues or eigenpairs with a given range of indices, say $\{\lambda_i\}_{i=a}^b$ or $\{(z_i, \lambda_i)\}_{i=a}^b$, with $0 \leq a \leq b < n$.
3. Compute all eigenpairs (or just eigenvalues) with eigenvalues lying in a particular half-open interval, either $\{\lambda_i \mid \lambda_i \in (a, b]\}$ or $\{(z_i, \lambda_i) \mid \lambda_i \in (a, b]\}$.

As of now, all three approaches start with Householder tridiagonalization (ala *HermitianTridiag()*) and then call Matthias Patschow and Paolo Bientinesi's PMRRR for the tridiagonal eigenvalue problem.

Note: Unfortunately, PMRRR currently only supports double-precision problems, and so the parallel versions of these routines are limited to real and complex double-precision matrices.

```
class HermitianEigCtrl<Real>
```

```
    HermitianTridiagCtrl tridiagCtrl
```

```
    HermitianSdcCtrl<Real> sdcCtrl
```

```
    bool useSdc
```

```
    HermitianEigCtrl()
```

```
        Initializes tridiagCtrl and sdcCtrl to their defaults and sets useSdc to false.
```

Note: Please see the *Tuning parameters* section for extensive information on maximizing the performance of Householder tridiagonalization.

Full spectrum computation

```
void HermitianEig(UpperOrLower uplo, Matrix<F> &A, Matrix<Base<F>> &w, Sort-  
    Type sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl =  
    HermitianEigCtrl<Base<F>>())
```



```
void HermitianEig(UpperOrLower uplo, DistMatrix<F> &A, DistMatrix<Base<F>, VR,
STAR> &w, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

Compute the full set of eigenvalues of the Hermitian matrix A .

```
void HermitianEig(UpperOrLower uplo, Matrix<F> &A, Matrix<Base<F>> &w, Matrix<F> &Z, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

```
void HermitianEig(UpperOrLower uplo, DistMatrix<F> &A, DistMatrix<Base<F>, VR, STAR> &w, DistMatrix<F> &Z, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

Compute the full set of eigenpairs of the Hermitian matrix A .

Index-based subset computation

```
void HermitianEig(UpperOrLower uplo, Matrix<F> &A, Matrix<Base<F>> &w, int a, int b, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

```
void HermitianEig(UpperOrLower uplo, DistMatrix<F> &A, DistMatrix<Base<F>, VR, STAR> &w, int a, int b, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

Compute the eigenvalues of a Hermitian matrix A with indices in the range $a, a + 1, \dots, b$.

```
void HermitianEig(UpperOrLower uplo, Matrix<F> &A, Matrix<Base<F>> &w, Matrix<F> &Z, int a, int b, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

```
void HermitianEig(UpperOrLower uplo, DistMatrix<F> &A, DistMatrix<Base<F>, VR, STAR> &w, DistMatrix<F> &Z, int a, int b, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

Compute the eigenpairs of a Hermitian matrix A with indices in the range $a, a + 1, \dots, b$.

Range-based subset computation

```
void HermitianEig(UpperOrLower uplo, Matrix<F> &A, Matrix<Base<F>> &w, Base<F> a, Base<F> b, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

```
void HermitianEig(UpperOrLower uplo, DistMatrix<F, STAR, STAR> &A, DistMatrix<Base<F>, STAR, STAR> &w, Base<F> a, Base<F> b, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

```
void HermitianEig(UpperOrLower uplo, DistMatrix<F> &A, DistMatrix<Base<F>, VR, STAR> &w, Base<F> a, Base<F> b, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

Compute the eigenvalues of a Hermitian matrix A lying in the half-open interval $(a, b]$.

```
void HermitianEig(UpperOrLower uplo, Matrix<F> &A, Matrix<Base<F>> &w, Matrix<F> &Z, Base<F> a, Base<F> b, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

```
void HermitianEig(UpperOrLower uplo, DistMatrix<F, STAR, STAR> &A, DistMatrix<Base<F>, STAR, STAR> &w, DistMatrix<F, STAR, STAR> &Z, Base<F> a, Base<F> b, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

```
void HermitianEig(UpperOrLower uplo, DistMatrix<F> &A, DistMatrix<Base<F>, VR, STAR> &w, DistMatrix<F> &Z, Base<F> a, Base<F> b, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

Compute the eigenpairs of a Hermitian matrix A with eigenvalues lying in the half-open interval $(a, b]$.

Spectral divide and conquer

Implementation

The primary references for this approach is Demmel et al.'s *Fast linear algebra is stable* and Nakatsukasa et al.'s *Stable and efficient spectral divide and conquer algorithms for the symmetric eigenvalue problem*.

```
class HermitianSdcCtrl<Real>
```

```
    int cutoff
```

```
    int maxInnerIts
```

```
    int maxOuterIts
```

```
    Real tol
```

```
    Real spreadFactor
```

```
    bool random
```

```
    bool progress
```

```
type HermitianSdcCtrl<Base<F>>
```

A particular case where the datatype is the base of the potentially complex type F .

```
void herm_eig::SDC(Matrix<F> &A, Matrix<Base<F>> &w, HermitianSdcCtrl<Base<F>> sdcCtrl = HermitianSdcCtrl<Base<F>>())
```

```
void herm_eig::SDC(DistMatrix<F> &A, DistMatrix<Base<F>, VR, STAR> &w, HermitianSdcCtrl<Base<F>> sdcCtrl = HermitianSdcCtrl<Base<F>>())
```

Compute the eigenvalues of the matrix A via a QDWH-based spectral divide and conquer process.

```
void herm_eig::SDC(Matrix<F> &A, Matrix<Base<F>> &w, Matrix<F> &Q, HermitianSdcCtrl<Base<F>> sdcCtrl = HermitianSdcCtrl<Base<F>>())
```

```
void herm_eig::SDC(DistMatrix<F> &A, DistMatrix<Base<F>, VR, STAR> &w, DistMatrix<F> &Q, HermitianSdcCtrl<Base<F>> sdcCtrl = HermitianSdcCtrl<Base<F>>())
```

Attempt to also compute the eigenvectors.

5.2.2 Skew-Hermitian eigensolver

Essentially identical to the Hermitian eigensolver, *HermitianEig()*; for any skew-Hermitian matrix G , iG is Hermitian, as

$$(iG)^H = -iG^H = iG.$$

This fact implies a fast method for solving skew-Hermitian eigenvalue problems:

1. Form iG in $O(n^2)$ work (switching to complex arithmetic in the real case)
2. Run a Hermitian eigensolve on iG , yielding $iG = Z\Lambda Z^H$.
3. Recognize that $G = Z(-i\Lambda)Z^H$ provides an EVD of G .

Please see the *HermitianEig()* documentation for more details.

Note: Unfortunately, PMRRR currently only supports double-precision problems, and so the parallel versions of these routines are limited to real and complex double-precision matrices.

Implementation

Full spectrum computation

```
void SkewHermitianEig(UpperOrLower uplo, Matrix<F> &G, Matrix<Base<F>>
    &wImag, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>>
    &ctrl = HermitianEigCtrl<Base<F>>())
```

```
void SkewHermitianEig(UpperOrLower uplo, DistMatrix<F> &G, DistMatrix<Base<F>,
    VR, STAR> &wImag, SortType sort = UNSORTED, const
    HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

Compute the full set of eigenvalues of the skew-Hermitian matrix G .

```
void SkewHermitianEig(UpperOrLower uplo, Matrix<F> &G, Matrix<Base<F>>
    &wImag, Matrix<Complex<Base<F>>> &Z, SortType sort
    = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl =
    HermitianEigCtrl<Base<F>>())
```

```
void SkewHermitianEig(UpperOrLower uplo, DistMatrix<F> &G, DistMatrix<Base<F>,
    VR, STAR> &wImag, DistMatrix<Complex<Base<F>>>
    &Z, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>>
    &ctrl = HermitianEigCtrl<Base<F>>())
```

Compute the full set of eigenpairs of the skew-Hermitian matrix G .

Index-based subset computation

```
void SkewHermitianEig(UpperOrLower uplo, Matrix<F> &G, Matrix<Base<F>>
    &wImag, int a, int b, SortType sort = UNSORTED, const
    HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

```
void SkewHermitianEig(UpperOrLower uplo, DistMatrix<F> &G, DistMatrix<Base<F>,
    VR, STAR> &wImag, int a, int b, SortType sort = UNSORTED,
    const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

Compute the eigenvalues of a skew-Hermitian matrix G with indices in the range $a, a + 1, \dots, b$.

```
void SkewHermitianEig(UpperOrLower uplo, Matrix<F> &G, Matrix<Base<F>>
    &wImag, Matrix<Complex<Base<F>>> &Z, int a, int b, SortType sort = UNSORTED,
    const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

```
void SkewHermitianEig(UpperOrLower uplo, DistMatrix<F> &G, DistMatrix<Base<F>,
    VR, STAR> &wImag, DistMatrix<Complex<Base<F>>> &Z,
    int a, int b, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>>
    &ctrl = HermitianEigCtrl<Base<F>>())
```

Compute the eigenpairs of a skew-Hermitian matrix G with indices in the range $a, a + 1, \dots, b$.

Range-based subset computation

```
void SkewHermitianEig(UpperOrLower uplo, Matrix<F> &G, Matrix<Base<F>>
    &wImag, Base<F> a, Base<F> b, SortType sort = UNSORTED,
    const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

```
void SkewHermitianEig(UpperOrLower uplo, DistMatrix<F> &G, DistMatrix<Base<F>,
    VR, STAR> &wImag, Base<F> a, Base<F> b, SortType sort =
    UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

Compute the eigenvalues of a skew-Hermitian matrix G lying in the half-open interval $(a, b]i$.

```
void SkewHermitianEig(UpperOrLower uplo, Matrix<F> &G, Matrix<Base<F>>
    &wImag, Matrix<F> &Z, Base<F> a, Base<F> b, SortType sort =
    UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl =
    HermitianEigCtrl<Base<F>>())
```

```
void SkewHermitianEig(UpperOrLower uplo, DistMatrix<F> &G, DistMatrix<Base<F>,
    VR, STAR> &wImag, DistMatrix<F> &Z, Base<F> a,
    Base<F> b, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>>
    &ctrl = HermitianEigCtrl<Base<F>>())
```

Compute the eigenpairs of a skew-Hermitian matrix G with eigenvalues lying in the half-open interval $(a, b]i$.

5.2.3 Hermitian generalized-definite eigensolvers

The following Hermitian generalized-definite eigenvalue problems frequently appear, where both A and B are Hermitian, and B is additionally positive-definite.

```
enum HermitianGenDefiniteEigType
```

enumerator ABX

$$ABx = \lambda x$$

enumerator BAX

$$BAx = \lambda x$$

enumerator AXBX

$$Ax = Bx\lambda$$

Implementation

Test driver

Full spectrum computation

```
void HermitianGenDefiniteEig(HermitianGenDefiniteEigType type, UpperOrLower uplo,
Matrix<F> &A, Matrix<F> &B, Matrix<Base<F>>
&w, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>>
&ctrl = HermitianEigCtrl<Base<F>>())
```

```
void HermitianGenDefiniteEig(HermitianGenDefiniteEigType type, UpperOrLower uplo,
DistMatrix<F> &A, DistMatrix<F> &B, DistMatrix<Base<F>, VR, STAR>
&w, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>>
&ctrl = HermitianEigCtrl<Base<F>>())
```

Compute the full set of eigenvalues of a generalized EVP involving the Hermitian matrices A and B , where B is also positive-definite.

```
void HermitianGenDefiniteEig(HermitianGenDefiniteEigType type, UpperOrLower uplo,
Matrix<F> &A, Matrix<F> &B, Matrix<Base<F>>
&w, Matrix<Base<F>> &Z, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>>
&ctrl = HermitianEigCtrl<Base<F>>())
```

```
void HermitianGenDefiniteEig(HermitianGenDefiniteEigType type, UpperOrLower uplo,
DistMatrix<F> &A, DistMatrix<F> &B,
DistMatrix<Base<F>, VR, STAR> &w, DistMatrix<Base<F>>
&Z, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>>
&ctrl = HermitianEigCtrl<Base<F>>())
```

Compute the full set of eigenpairs of a generalized EVP involving the Hermitian matrices A and B , where B is also positive-definite.

Index-based subset computation

```
void HermitianGenDefiniteEig(HermitianGenDefiniteEigType type, UpperOrLower uplo,
Matrix<F> &A, Matrix<F> &B, Matrix<Base<F>>
&w, int a, int b, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>>
&ctrl = HermitianEigCtrl<Base<F>>())
```

```
void HermitianGenDefiniteEig(HermitianGenDefiniteEigType type, UpperOrLower uplo,
    DistMatrix<F> &A, DistMatrix<F> &B, DistMatrix<Base<F>, VR, STAR> &w, int a, int b, SortType
    sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

Compute the eigenvalues with indices in the range $a, a + 1, \dots, b$ of a generalized EVP involving the Hermitian matrices A and B , where B is also positive-definite.

```
void HermitianGenDefiniteEig(HermitianGenDefiniteEigType type, UpperOrLower uplo,
    Matrix<F> &A, Matrix<F> &B, Matrix<Base<F>> &w, Matrix<F> &Z, int a, int b, SortType sort = UN-
    SORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

```
void HermitianGenDefiniteEig(HermitianGenDefiniteEigType type, UpperOrLower uplo,
    DistMatrix<F> &A, DistMatrix<F> &B, DistMatrix<Base<F>, VR, STAR> &w, DistMatrix<F> &Z,
    int a, int b, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigC-
    ctrl<Base<F>>())
```

Compute the eigenpairs with indices in the range $a, a + 1, \dots, b$ of a generalized EVP involving the Hermitian matrices A and B , where B is also positive-definite.

Range-based subset computation

```
void HermitianGenDefiniteEig(HermitianGenDefiniteEigType type, UpperOrLower uplo,
    Matrix<F> &A, Matrix<F> &B, Matrix<Base<F>> &w, Base<F> a, Base<F> b, SortType sort = UN-
    SORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

```
void HermitianGenDefiniteEig(HermitianGenDefiniteEigType type, UpperOrLower
    uplo, DistMatrix<F> &A, DistMatrix<F> &B, DistMatrix<Base<F>, VR, STAR> &w, Base<F> a,
    Base<F> b, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigC-
    ctrl<Base<F>>())
```

Compute the eigenvalues lying in the half-open interval $(a, b]$ of a generalized EVP involving the Hermitian matrices A and B , where B is also positive-definite.

```
void HermitianGenDefiniteEig(HermitianGenDefiniteEigType type, UpperOrLower uplo,
    Matrix<F> &A, Matrix<F> &B, Matrix<Base<F>> &w, Matrix<F> &Z, Base<F> a, Base<F> b, SortType
    sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = HermitianEigCtrl<Base<F>>())
```

```
void HermitianGenDefiniteEig(HermitianGenDefiniteEigType type, UpperOrLower uplo,
    DistMatrix<F> &A, DistMatrix<F> &B, DistMatrix<Base<F>, VR, STAR> &w, DistMatrix<F> &Z,
    Base<F> a, Base<F> b, SortType sort = UNSORTED, const HermitianEigCtrl<Base<F>> &ctrl = Hermi-
    tianEigCtrl<Base<F>>())
```

Compute the eigenpairs whose eigenvalues lie in the half-open interval $(a, b]$ of a generalized EVP involving the Hermitian matrices A and B , where B is also positive-definite.

5.2.4 Unitary eigensolver

Not yet written, will likely be based on Ming Gu's unitary Divide and Conquer algorithm for unitary Hessenberg matrices. The spectral divide and conquer technique described below should suffice in the meantime.

5.2.5 Normal eigensolver

Not yet written, will likely be based on Angelika Bunse-Gerstner et al.'s Jacobi-like method for simultaneous diagonalization of the commuting Hermitian and skew-Hermitian portions of the matrix. The spectral divide and conquer scheme described below should suffice in the meantime.

5.2.6 Schur decomposition

Elemental contains a native prototype implementation of a spectral divide and conquer scheme for the Schur decomposition, but it is not yet robust enough to handle general matrices. For local matrices, Elemental defaults to calling LAPACK's Hessenberg QR algorithm (with Aggressive Early Deflation); if support for ScaLAPACK was detected during configuration, Elemental defaults to ScaLAPACK's Hessenberg QR algorithm (without deflation), otherwise the Spectral Divide and Conquer approach is attempted.

Main header file

```
void Schur(Matrix<F> &A)
void Schur(DistMatrix<F> &A)
void Schur(Matrix<F> &A, Matrix<F> &Q)
void Schur(DistMatrix<F> &A, DistMatrix<F> &Q)
```

Hessenberg QR algorithm

QR algorithm header file

```
void schur::QR(Matrix<F> &A, Matrix<Complex<Base<F>>> &w, bool fullTriangle =
    false)
void schur::QR(Matrix<F> &A, Matrix<Complex<Base<F>>> &w, Matrix<F> &Q,
    bool fullTriangle = true)
    Condense the matrix to upper-Hessenberg form and then use a sequential QR algorithm
    to compute a (partial) Schur decomposition. It is optional whether or not the full Schur
    factor is computed.
void schur::QR(DistMatrix<F> &A, DistMatrix<Complex<Base<F>>, VR, STAR> &w,
    bool fullTriangle = false, bool aed = false)
void schur::QR(BlockDistMatrix<F> &A, DistMatrix<Complex<Base<F>>, VR,
    STAR> &w, bool fullTriangle = false, bool aed = false)
void schur::QR(DistMatrix<F> &A, DistMatrix<Complex<Base<F>>, VR, STAR> &w,
    DistMatrix<F> &Q, bool fullTriangle = true, bool aed = false)
```

```
void schur::QR(BlockDistMatrix<F> &A, DistMatrix<Complex<Base<F>>, VR,
              STAR> &w, BlockDistMatrix<F> &Q, bool fullTriangle = true, bool aed
              = false)
```

Condense the matrix to upper-Hessenberg form and then use ScaLAPACK's parallel QR algorithm to compute a (partial) Schur decomposition. It is optional whether or not the full Schur factor is computed, and Aggressive Early Deflation is also optional for real matrices (as of now, its usage is not recommended due to known bugs in the implementation).

Quasi-triangular manipulation

```
void schur::QuasiTriangEig(const Matrix<F> &U, Matrix<Complex<Base<F>>>
                          &w)
```

```
void schur::QuasiTriangEig(const DistMatrix<F> &U, DistMa-
                          trix<Complex<Base<F>>, colDist, rowDist> &w)
```

Return the eigenvalues of the upper quasi-triangular matrix U in the vector w .

```
Matrix<Complex<Base<F>>> schur::QuasiTriangEig(const Matrix<F> &U)
```

```
DistMatrix<Complex<Base<F>>, VR, STAR> schur::QuasiTriangEig(const Dist-
                                                            Matrix<F>
                                                            &U)
```

Return the eigenvalues of the upper quasi-triangular matrix U .

```
void schur::QuasiTriangEig(const Matrix<F> &dMain, const Matrix<F> &dSub,
                          const Matrix<F> &dSup, Matrix<Complex<Base<F>>>
                          &w)
```

The underlying computation routine for computing the eigenvalues of quasi-triangular matrices. The vectors $dMain$, $dSub$, and $dSup$ should respectively contain the main, sub, and super-diagonals of the upper quasi-triangular matrix.

```
void schur::RealToComplex(const Matrix<Real> &UQuasi, Matrix<Complex<Real>>
                          &U)
```

```
void schur::RealToComplex(const DistMatrix<Real> &UQuasi, DistMa-
                          trix<Complex<Real>> &U)
```

Rotate a real upper quasi-triangular matrix into a complex upper triangular matrix.

```
void schur::CheckRealSchur(const Matrix<Real> &U, bool standardForm = false)
```

```
void schur::CheckRealSchur(const DistMatrix<Real> &U, bool standardForm = false)
```

Check whether or not the largest diagonal blocks of the upper quasi-triangular matrix are at most 2×2 and, optionally, check if the 2×2 diagonal blocks are in standard form (if so, their diagonal must be constant and the product of the off-diagonal entries should be negative).

Spectral divide and conquer

The primary reference for this approach is Demmel et al.'s *Fast linear algebra is stable*. While the current implementation needs a large number of algorithmic improvements, especially with respect to choosing the Mobius transformations, it tends to succeed on random matrices.

SDC header file

```
class SdcCtrl<Real>
```



```

int cutoff
int maxInnerIts
int maxOuterIts
Real tol
Real spreadFactor
bool random
bool progress

```

type SdcCtrl<Base<F>>

A particular case where the datatype is the base of the potentially complex type F.

```
void schur::SDC(Matrix<F> &A, Matrix<Complex<Base<F>>> &w, bool formATR =
    false, SdcCtrl<Base<F>> sdcCtrl = SdcCtrl<Base<F>>())
```

```
void schur::SDC(DistMatrix<F> &A, DistMatrix<Complex<Base<F>>, VR, STAR>
    &w, bool formATR = false, SdcCtrl<Base<F>> sdcCtrl = SdcCtrl<Base<F>>())
```

Compute the eigenvalues of the matrix A via a spectral divide and conquer process. On exit, the eigenvalues of A will be stored on its diagonal, and, if `formATR` was set to true, the upper triangle of A will be its corresponding upper-triangular Schur factor.

```
void schur::SDC(Matrix<F> &A, Matrix<Complex<Base<F>>> &w, Matrix<F>
    &Q, bool formATR = true, SdcCtrl<Base<F>> sdcCtrl = SdcCtrl<Base<F>>())
```

```
void schur::SDC(DistMatrix<F> &A, DistMatrix<Complex<Base<F>>, VR, STAR>
    &w, DistMatrix<F> &Q, bool formATR = true, SdcCtrl<Base<F>> sdcCtrl = SdcCtrl<Base<F>>())
```

Attempt to also compute the Schur vectors.

5.2.7 Hermitian SVD

Given an eigenvalue decomposition of a Hermitian matrix A , say

$$A = V\Lambda V^H,$$

where V is unitary and Λ is diagonal and real. Then an SVD of A can easily be computed as

$$A = U|\Lambda|V^H,$$

where the columns of U equal the columns of V , modulo sign flips introduced by negative eigenvalues.

Implementation

```
void HermitianSVD(UpperOrLower uplo, Matrix<F> &A, Matrix<Base<F>> &s, Matrix<F> &U, Matrix<F> &V)
```

```
void HermitianSVD(UpperOrLower uplo, DistMatrix<F> &A, DistMatrix<Base<F>, VR, STAR> &s, DistMatrix<F> &U, DistMatrix<F> &V)
```

Return a vector of singular values, s , and the left and right singular vector matrices, U and V , such that $A = U\text{diag}(s)V^H$.

```
void HermitianSVD(UpperOrLower uplo, Matrix<F> &A, Matrix<Base<F>> &s)
```

```
void HermitianSVD(UpperOrLower uplo, DistMatrix<F> &A, DistMatrix<Base<F>, VR,
                 STAR> &s)
    Return the singular values of  $A$  in  $s$ . Note that the appropriate triangle of  $A$  is overwritten
    during computation.
```

5.2.8 Polar decomposition

Every matrix A can be written as $A = QP$, where Q is unitary and P is Hermitian and positive semi-definite. This is known as the *polar decomposition* of A and can be constructed as $Q := UV^H$ and $P := V\Sigma V^H$, where $A = U\Sigma V^H$ is the SVD of A . Alternatively, it can be computed through (a dynamically-weighted) Halley iteration.

Implementation

SVD approach

QWDH approach

```
void Polar(Matrix<F> &A)
```

```
void Polar(DistMatrix<F> &A)
```

```
void Polar(Matrix<F> &A, Matrix<F> &P)
```

```
void Polar(DistMatrix<F> &A, DistMatrix<F> &P)
```

Compute the polar decomposition of A , $A = QP$, returning Q within A and P within P . The current implementation first computes the SVD.

```
void HermitianPolar(UpperOrLower uplo, Matrix<F> &A)
```

```
void HermitianPolar(UpperOrLower uplo, DistMatrix<F> &A)
```

```
void HermitianPolar(UpperOrLower uplo, Matrix<F> &A, Matrix<F> &P)
```

```
void HermitianPolar(UpperOrLower uplo, DistMatrix<F> &A, DistMatrix<F> &P)
```

Compute the polar decomposition through a Hermitian EVD. Since this is equivalent to a Hermitian sign decomposition (if $\text{sgn}(0)$ is set to 1), these routines are equivalent to *HermitianSign*.

polar namespace

```
int polar::QDWH(Matrix<F> &A, bool colPiv = false, int maxIts = 20)
```

```
int polar::QDWH(DistMatrix<F> &A, bool colPiv = false, int maxIts = 20)
```

```
int hermitian_polar::QDWH(UpperOrLower uplo, Matrix<F> &A, bool colPiv = false, int
                        maxIts = 20)
```

```
int hermitian_polar::QDWH(UpperOrLower uplo, DistMatrix<F> &A, bool colPiv = false,
                        int maxIts = 20)
```

Overwrites A with the Q from the polar decomposition using a QR-based dynamically weighted Halley iteration. The number of iterations used is returned upon completion.

TODO: reference to Yuji's paper

```
int polar::QDWH(Matrix<F> &A, Matrix<F> &P, bool colPiv = false, int maxIts = 20)
```

```
int polar::QDWH(DistMatrix<F> &A, DistMatrix<F> &P, bool colPiv = false, int maxIts =
                20)
```

```
int hermitian_polar::QDWH(UpperOrLower uplo, Matrix<F> &A, Matrix<F> &P, bool
    colPiv = false, int maxIts = 20)
```

```
int hermitian_polar::QDWH(UpperOrLower uplo, DistMatrix<F> &A, DistMatrix<F>
    &P, bool colPiv = false, int maxIts = 20)
```

Return the full polar decomposition, where P is HPD.

5.2.9 SVD

Given a general matrix A , the *Singular Value Decomposition* is the triplet (U, Σ, V) such that

$$A = U\Sigma V^H,$$

where U and V are unitary, and Σ is diagonal with non-negative entries.

Implementation

Subroutines

```
void SVD(Matrix<F> &A, Matrix<Base<F>> &s, Matrix<F> &V)
```

```
void SVD(DistMatrix<F> &A, DistMatrix<Base<F>, VR, STAR> &s, DistMatrix<F>
    &V)
```

Overwrites A with U , s with the diagonal entries of Σ , and V with V .

```
void SVD(Matrix<F> &A, Matrix<Base<F>> &s)
```

```
void SVD(DistMatrix<F> &A, DistMatrix<Base<F>, VR, STAR> &s)
```

Forms the singular values of A in s . Note that A is overwritten in order to compute the singular values.

svd namespace

```
void svd::QRSVD(Matrix<F> &A, Matrix<Base<F>> &s, Matrix<F> &V)
```

SVD which uses bidiagonal QR algorithm.

```
void svd::DivideAndConquerSVD(Matrix<F> &A, Matrix<Base<F>> &s, Matrix<F>
    &V)
```

SVD which uses a bidiagonal divide-and-conquer algorithm.

```
void svd::Chan(DistMatrix<F> &A, DistMatrix<Base<F>, VR, STAR> &s, double
    heightRatio = 1.2)
```

```
void svd::Chan(DistMatrix<F> &A, DistMatrix<Base<F>, VR, STAR> &s, DistMa-
    trix<F> &V, double heightRatio = 1.5)
```

SVD which preprocesses with an initial QR decomposition if the matrix is sufficiently tall relative to its width.

```
void svd::GolubReinschUpper(DistMatrix<F> &A, DistMatrix<Base<F>, VR, STAR>
    &s)
```

```
void svd::GolubReinschUpper(DistMatrix<F> &A, DistMatrix<Base<F>, VR, STAR>
    &s, DistMatrix<F> &V)
```

Computes the singular values (and vectors) of a matrix which is taller than it is wide using the Golub-Reinsch algorithm, though DQDS is used when only the singular values are sought.

```
void svd::Thresholded(Matrix<F> &A, Matrix<Base<F>> &s, Matrix<F> &V,
    Base<F> tol = 0, bool relative = false)
```

void svd::Thresholded(*DistMatrix*<F> &A, *DistMatrix*<Base<F>, VR, STAR> &s,
DistMatrix<F> &V, *Base*<F> tol = 0, bool relative = false)
 Computes the singular triplets whose singular values are larger than a specified tolerance using the cross-product algorithm. This is often advantageous because tridiagonal eigensolvers tend to enjoy better parallel implementations than bidiagonal SVD's.

5.3 Factorizations

5.3.1 Cholesky factorization

It is well-known that Hermitian positive-definite (HPD) matrices can be decomposed into the form $A = LL^H$ or $A = U^H U$, where $L = U^H$ is lower triangular, and Cholesky factorization provides such an L (or U) given an HPD A . If A is found to be numerically indefinite, then a *NonHPDMatrixException* will be thrown.

Main header file

Subroutines

Test driver

void Cholesky(*UpperOrLower* uplo, *Matrix*<F> &A)

void Cholesky(*UpperOrLower* uplo, *DistMatrix*<F> &A)

Overwrite the *uplo* triangle of the HPD matrix A with its Cholesky factor.

void Cholesky(*UpperOrLower* uplo, *Matrix*<F> &A, *Matrix*<int> &p)

void Cholesky(*UpperOrLower* uplo, *DistMatrix*<F> &A, *DistMatrix*<int, UPerm, STAR>
 &p)

Performs Cholesky factorization with full (diagonal) pivoting. On exit, the vector p consists of the nonzero column indices of the permutation matrix P such that $PAP^T = LL^H = U^H U$.

void CholeskyMod(*UpperOrLower* uplo, *Matrix*<F> &T, *Base*<F> &alpha, *Matrix*<F>
 &V)

void CholeskyMod(*UpperOrLower* uplo, *DistMatrix*<F> &T, *Base*<F> &alpha, *DistMatrix*<F> &V)

Updates the Cholesky factorization to incorporate the modification αVV^H to the original matrix. The current algorithm uses Householder transformations for updates ($\alpha \geq 0$) and hyperbolic Householder transformations for downdates.

It is possible to compute the Cholesky factor of a Hermitian positive semi-definite (HPSD) matrix through its eigenvalue decomposition, though it is significantly more expensive than the HPD case: Let $A = U\Lambda U^H$ be the eigenvalue decomposition of A , where all entries of Λ are non-negative. Then $B = U\sqrt{\Lambda}U^H$ is the matrix square root of A , i.e., $BB = A$, and it follows that the QR and LQ factorizations of B yield Cholesky factors of A :

$$A = BB = B^H B = (QR)^H (QR) = R^H Q^H QR = R^H R,$$

and

$$A = BB = BB^H = (LQ)(LQ)^H = LQQ^H L^H = LL^H.$$

If A is found to have eigenvalues less than $-\epsilon \|A\|_2$, then a *NonHPSDMatrixException* will be thrown.

Example driver

```
void HPSDCholesky(UpperOrLower uplo, Matrix<F> &A)
```

```
void HPSDCholesky(UpperOrLower uplo, DistMatrix<F> &A)
```

Overwrite the *uplo* triangle of the potentially singular matrix *A* with its Cholesky factor.

cholesky namespace

```
void cholesky::SolveAfter(UpperOrLower uplo, Orientation orientation, const Ma-  
trix<F> &A, Matrix<F> &B)
```

```
void cholesky::SolveAfter(UpperOrLower uplo, Orientation orientation, const DistMa-  
trix<F> &A, DistMatrix<F> &B)
```

Solve linear systems using an unpivoted Cholesky factorization.

```
void cholesky::SolveAfter(UpperOrLower uplo, Orientation orientation, const Ma-  
trix<F> &A, Matrix<F> &B, Matrix<int> &perm)
```

```
void cholesky::SolveAfter(UpperOrLower uplo, Orientation orientation, const DistMa-  
trix<F> &A, DistMatrix<F> &B, DistMatrix<int, UPerm,  
STAR> &perm)
```

Solve linear systems using a pivoted Cholesky factorization.

5.3.2 LDL factorization

enum LDLPivotType

For specifying a symmetric pivoting strategy. The current (not yet all supported) options include:

enumerator BUNCH_KAUFMAN_A

enumerator BUNCH_KAUFMAN_C

Not yet supported.

enumerator BUNCH_KAUFMAN_D

enumerator BUNCH_KAUFMAN_BOUNDED

Not yet supported.

enumerator BUNCH_PARLETT

class LDLPivot

int nb

int from[2]

Main header file

Subroutines

Test driver

Example driver

```
void LDLH(Matrix<F> &A, Matrix<F> &dSub, Matrix<int> &p, LDLPivotType pivotType  
= BUNCH_KAUFMAN_A)
```

```
void LDLT(Matrix<F> &A, Matrix<F> &dSub, Matrix<int> &p, LDLPivotType pivotType
        = BUNCH_KAUFMAN_A)
```

```
void LDLH(DistMatrix<F> &A, DistMatrix<F, MD, STAR> &dSub, DistMatrix<int, UP-
        erm, STAR> &p, LDLPivotType pivotType = BUNCH_KAUFMAN_A)
```

```
void LDLT(DistMatrix<F> &A, DistMatrix<F, MD, STAR> &dSub, DistMatrix<int, UP-
        erm, STAR> &p, LDLPivotType pivotType = BUNCH_KAUFMAN_A)
```

Returns a pivoted LDL factorization, where the vector p contains the column indices of the nonzero entries of the permutation matrix P such that PAP^T equals either LDL^T or LDL^H , where D is quasi-diagonal. The Bunch-Kaufman pivoting rules are used within a higher-performance blocked algorithm, whereas the Bunch-Parlett strategy uses an unblocked algorithm.

Though the Cholesky factorization is ideal for most HPD matrices, the unpivoted LDL factorizations exist as slight relaxation of the Cholesky factorization and compute lower-triangular (with unit diagonal) L and diagonal D such that $A = LDL^H$ or $A = LDL^T$. If a zero pivot is attempted, then a `ZeroPivotException` will be thrown.

Warning: The following routines do not pivot, so please use with caution.

```
void LDLH(Matrix<F> &A)
```

```
void LDLT(Matrix<F> &A)
```

```
void LDLH(DistMatrix<F> &A)
```

```
void LDLT(DistMatrix<F> &A)
```

Overwrite the strictly lower triangle of A with the strictly lower portion of L (L implicitly has ones on its diagonal) and the diagonal with D .

ldl namespace

```
void ldl::SolveAfter(const Matrix<F> &A, Matrix<F> &B, bool conjugated = false)
```

```
void ldl::SolveAfter(const DistMatrix<F> &A, DistMatrix<F> &B, bool conjugated =
        false)
```

Solve linear systems using an unpivoted LDL factorization.

```
void ldl::SolveAfter(const Matrix<F> &A, const Matrix<F> &dSub, const Ma-
        trix<int> &p, Matrix<F> &B, bool conjugated = false)
```

```
void ldl::SolveAfter(const DistMatrix<F> &A, const DistMatrix<F, MD, STAR>
        &dSub, const DistMatrix<int, UPerm, STAR> &p, DistMa-
        trix<F> &B, bool conjugated = false)
```

Solve linear systems using a pivoted LDL factorization.

5.3.3 LU factorization

Main header file

Subroutines

Test driver

Example driver

Given $A \in \mathbb{F}^{m \times n}$, an LU factorization (without pivoting) finds a unit lower-trapezoidal $L \in \mathbb{F}^{m \times \min(m,n)}$ and upper-trapezoidal $U \in \mathbb{F}^{\min(m,n) \times n}$ such that $A = LU$. Since L is required to have its diagonal entries set to one: the upper portion of A can be overwritten with U , and the strictly lower portion of A can be overwritten with the strictly lower portion of L . If A is found to be numerically singular, then a *SingularMatrixException* will be thrown.

```
void LU(Matrix<F> &A)
```

```
void LU(DistMatrix<F> &A)
```

Overwrites A with its LU decomposition.

Since LU factorization without pivoting is known to be unstable for general matrices, it is standard practice to pivot the rows of A during the factorization (this is called partial pivoting since the columns are not also pivoted). An LU factorization with partial pivoting therefore computes P , L , and U such that $PA = LU$, where L and U are as described above and P is a permutation matrix.

```
void LU(Matrix<F> &A, Matrix<int> &p)
```

```
void LU(DistMatrix<F> &A, DistMatrix<F, UPerm, STAR> &p)
```

Overwrites the matrix A with the LU decomposition of PA , where P is represented by the permutation vector p , which consists of the column indices of the nonzero entry in each row of P .

```
void LU(Matrix<F> &A, Matrix<int> &p, Matrix<int> &q)
```

```
void LU(DistMatrix<F> &A, DistMatrix<F, UPerm, STAR> &p, DistMatrix<F, UPerm, STAR> &q)
```

Overwrites the matrix A with the LU decomposition of PAQ^T , where P and Q are represented by the permutation vectors p and q , which consist of the column indices of the nonzero entry in each row of P and Q , respectively.

```
void LUMod(Matrix<F> &A, Matrix<int> &p, const Matrix<F> &u, const Matrix<F> &v, bool conjugate = true, Base<F> tau = 0.1)
```

```
void LUMod(DistMatrix<F> &A, DistMatrix<int, UPerm, STAR> &p, const DistMatrix<F> &u, const DistMatrix<F> &v, bool conjugate = true, Base<F> tau = 0.1)
```

Modify an existing LU factorization, $A = P^T L U$, to incorporate the rank-one update $A + uv^T$ or $A + uv^H$. This algorithm only requires a quadratic number of operations.

Note: The current implementation has only been tested for square matrices.

lu namespace

```
void lu::SolveAfter(Orientation orientation, const Matrix<F> &A, Matrix<F> &B)
```

```
void lu::SolveAfter(Orientation orientation, const DistMatrix<F> &A, DistMatrix<F> &B)
```

Update $B := A^{-1}B$, $B := A^{-T}B$, or $B := A^{-H}B$, where A has been overwritten with its LU factors (without partial pivoting).

```
void lu::SolveAfter(Orientation orientation, const Matrix<F> &A, const Matrix<int> &p, Matrix<F> &B)
```

```
void lu::SolveAfter(Orientation orientation, const DistMatrix<F> &A, const DistMatrix<int, UPerm, STAR> &p, DistMatrix<F> &B)
```

HERE Update $B := A^{-1}B$, $B := A^{-T}B$, or $B := A^{-H}B$, where A has been overwritten with its LU factors with partial pivoting, which satisfy $PA = LU$, where the permutation matrix P is represented by the pivot vector p .

```
void lu::SolveAfter(Orientation orientation, const Matrix<F> &A, const Matrix<int> &p, const Matrix<int> &q, Matrix<F> &B)
```

```
void lu::SolveAfter(Orientation orientation, const DistMatrix<F> &A, const DistMatrix<int, UPerm, STAR> &p, const DistMatrix<int, UPerm, STAR> &q, DistMatrix<F> &B)
```

Update $B := A^{-1}B$, $B := A^{-T}B$, or $B := A^{-H}B$, where A has been overwritten with its LU factors with full pivoting, which satisfy $PAQ = LU$, where the permutation matrices P and Q are represented by the pivot vector p and q , respectively.

5.3.4 LQ factorization

Main header file

Subroutines

Test driver

Given $A \in \mathbb{F}^{m \times n}$, an LQ factorization typically computes an implicit unitary matrix $\hat{Q} \in \mathbb{F}^{n \times n}$ such that $\hat{L} \equiv A\hat{Q}^H$ is lower trapezoidal. One can then form the thin factors $L \in \mathbb{F}^{m \times \min(m,n)}$ and $Q \in \mathbb{F}^{\min(m,n) \times n}$ by setting L and Q to first $\min(m,n)$ columns and rows of \hat{L} and \hat{Q} , respectively. Upon completion L is stored in the lower trapezoid of A and the Householder reflectors (and preceding unitary diagonal matrix forcing L to have a positive diagonal, defined by the vector d) representing \hat{Q} are stored within the rows of the strictly upper trapezoid.

```
void LQ(Matrix<F> &A)
```

```
void LQ(DistMatrix<F> &A)
```

```
void LQ(Matrix<F> &A, Matrix<F> &t, Matrix<Base<F>> &d)
```

```
void LQ(DistMatrix<F> &A, DistMatrix<F, MD, STAR> &t, DistMatrix<Base<F>, MD, STAR> &d)
```

Overwrite the matrix A with L and the Householder reflectors representing \hat{Q} . The scalings for the Householder reflectors are stored in the vector t and the diagonal matrix which forces L to be positive in d .

lq namespace

```
void lq::ApplyQ(LeftOrRight side, Orientation orientation, const Matrix<F> &A, const Matrix<F> &t, const Matrix<Base<F>> &d, Matrix<F> &B)
```

```
void lq::ApplyQ(LeftOrRight side, Orientation orientation, const DistMatrix<F> &A, const DistMatrix<F, Ut, Vt> &t, const DistMatrix<Base<F>, Ud, Vd> &d, DistMatrix<F> &B)
```

Applies the implicitly-defined Q (or its adjoint) stored within A , t , and d from either the left or the right to B .


```
void lq::SolveAfter(Orientation orientation, const Matrix<F> &A, const Matrix<F> &t,
                  const Matrix<Base<F>> &d, const Matrix<F> &B, Matrix<F>
                  &X)
```

```
void lq::SolveAfter(Orientation orientation, const DistMatrix<F> &A, const DistMa-
                  trix<F, MD, STAR> &t, const DistMatrix<Base<F>, MD, STAR>
                  &d, const DistMatrix<F> &B, DistMatrix<F> &X)
```

Solves a set of linear systems using an existing packed LQ factorization given by A and the vectors t and d . B is the matrix of input vectors and X is the matrix of solutions.

5.3.5 QR factorization

Main header file

Subroutines

Test driver

Example driver

Given $A \in \mathbb{F}^{m \times n}$, a QR factorization typically computes an implicit unitary matrix $\hat{Q} \in \mathbb{F}^{m \times m}$ such that $\hat{R} \equiv \hat{Q}^H A$ is upper trapezoidal. One can then form the thin factors $Q \in \mathbb{F}^{m \times \min(m,n)}$ and $R \in \mathbb{F}^{\min(m,n) \times n}$ by setting Q and R to first $\min(m,n)$ columns and rows of \hat{Q} and \hat{R} , respectively. Upon completion R is stored in the upper trapezoid of A and the Householder reflectors representing \hat{Q} are stored within the columns of the strictly lower trapezoid (this unitary matrix is scaled from the right by a unitary diagonal matrix with entries given by d so that R has a positive diagonal).

```
void QR(Matrix<F> &A)
```

```
void QR(DistMatrix<F> &A)
    Overwrite  $A$  with  $R$ .
```

```
void QR(Matrix<F> &A, Matrix<F> &t, Matrix<Base<F>> &d)
```

```
void QR(DistMatrix<F> &A, DistMatrix<F, MD, STAR> &t, DistMatrix<Base<F>, MD,
        STAR> &d)
    Overwrite the matrix  $A$  with both  $R$  and the Householder reflectors (and subsequent unitary diagonal matrix defined by the vector,  $d$ ) representing  $\hat{Q}$ . The scalings for the Householder reflectors are stored in the vector  $t$ .
```

```
void QR(Matrix<F> &A, Matrix<int> &p)
```

```
void QR(DistMatrix<F> &A, DistMatrix<int, UPerm, STAR> &p)
    Overwrite  $A$  with the  $R$  from a column-pivoted QR factorization,  $AP = QR$ . The permutation matrix  $P$  is represented via the permutation vector  $p$ , which contains the column indices of the nonzero entry in each row of  $P$ .
```

```
void QR(Matrix<F> &A, Matrix<F> &t, Matrix<Base<F>> &d, Matrix<int> &p)
```

```
void QR(DistMatrix<F> &A, DistMatrix<F, MD, STAR> &t, DistMatrix<Base<F>, MD,
        STAR> &d, DistMatrix<int, UPerm, STAR> &p)
    Overwrite  $A$  with both the  $R$  and (scaled) Householder reflectors from a column-pivoted QR factorization.
```

qr namespace

```
void qr::Explicit(Matrix<F> &A, bool colPiv = false)
```

```
void qr::Explicit(DistMatrix<F> &A, bool colPiv = false)
```

Overwrite A with the orthogonal matrix from its QR factorization (with or without column pivoting).

```
void qr::Explicit(Matrix<F> &A, Matrix<F> &R, bool colPiv = false)
```

```
void qr::Explicit(DistMatrix<F> &A, DistMatrix<F> &R, bool colPiv = false)
```

Additionally explicitly return the R from the QR factorization.

```
void qr::Explicit(Matrix<F> &A, Matrix<F> &R, Matrix<int> &p)
```

```
void qr::Explicit(DistMatrix<F> &A, DistMatrix<F> &R, DistMatrix<int, UPerm, STAR> &p)
```

Return representations of all matrices of the pivoted QR factorization (note that the pivot *vector* is returned, not the full pivot matrix).

```
void qr::ApplyQ(LeftOrRight side, Orientation orientation, const Matrix<F> &A, const Matrix<F> &t, const Matrix<Base<F>> &d, Matrix<F> &B)
```

```
void qr::ApplyQ(LeftOrRight side, Orientation orientation, const DistMatrix<F> &A, const DistMatrix<F, Ut, Vt> &t, const DistMatrix<Base<F>, Ud, Vd> &d, DistMatrix<F> &B)
```

Applies the implicitly-defined Q (or its adjoint) stored within A , t , and d from either the left or the right to B .

```
int qr::BusingerGolub(Matrix<F> &A, Matrix<int> &p)
```

```
int qr::BusingerGolub(DistMatrix<F> &A, DistMatrix<int, UPerm, STAR> &p)
```

```
int qr::BusingerGolub(Matrix<F> &A, Matrix<F> &t, Matrix<Base<F>> &d, Matrix<int> &p)
```

```
int qr::BusingerGolub(DistMatrix<F> &A, DistMatrix<F, MD, STAR> &t, DistMatrix<Base<F>, MD, STAR> &d, DistMatrix<int, UPerm, STAR> &p)
```

Column-pivoted versions of the above routines which use the Businger/Golub strategy, i.e., the pivot is chosen as the remaining column with maximum two norm. The return value is the number of performed iterations.

```
int qr::BusingerGolub(Matrix<F> &A, Matrix<int> &p, int numSteps)
```

```
int qr::BusingerGolub(DistMatrix<F> &A, DistMatrix<int, UPerm, STAR> &p, int numSteps)
```

```
int qr::BusingerGolub(Matrix<F> &A, Matrix<F> &t, Matrix<Base<F>> &d, Matrix<int> &p, int numSteps)
```

```
int qr::BusingerGolub(DistMatrix<F> &A, DistMatrix<F, MD, STAR> &t, DistMatrix<Base<F>, MD, STAR> &d, DistMatrix<int, UPerm, STAR> &p, int numSteps)
```

Same as above, but only execute a fixed number of steps of the rank-revealing factorization. The return value is the number of performed iterations.

```
int qr::BusingerGolub(Matrix<F> &A, Matrix<int> &p, int maxSteps, Base<F> tol)
```

```
int qr::BusingerGolub(DistMatrix<F> &A, DistMatrix<int, UPerm, STAR> &p, int maxSteps, Base<F> tol)
```

```
int qr::BusingerGolub(Matrix<F> &A, Matrix<F> &t, Matrix<int> &p, int maxSteps,
                    Base<F> tol)
```

```
int qr::BusingerGolub(DistMatrix<F> &A, DistMatrix<F, MD, STAR> &t, DistMa-
                    trix<int, UPerm, STAR> &p, int maxSteps, Base<F> tol)
```

Either execute *maxSteps* iterations or stop after the maximum remaining column norm is less than or equal to *tol* times the maximum original column norm. The return value is the number of performed iterations.

```
void qr::SolveAfter(Orientation orientation, const Matrix<F> &A, const Matrix<F> &t,
                  const Matrix<Base<F>> &d, const Matrix<F> &B, Matrix<F>
                  &X)
```

```
void qr::SolveAfter(Orientation orientation, const DistMatrix<F> &A, const DistMa-
                  trix<F, MD, STAR> &t, const DistMatrix<Base<F>, MD, STAR>
                  &d, const DistMatrix<F> &B, DistMatrix<F> &X)
```

Solves a set of linear systems using an existing packed QR factorization given by *A* and the vectors *t* and *d*. *B* is the matrix of input vectors and *X* is the matrix of solutions.

```
type TreeData<F>
```

```
Matrix<F> QR0
    Initial QR factorization
```

```
Matrix<F> t0
    Phases from initial QR factorization
```

```
Matrix<Base<F>> d0
    Signature (-1,+1) which scales the Householder matrix from the right.
```

```
std::vector<Matrix<F>> QRList
    Factorizations within reduction tree
```

```
std::vector<Matrix<F>> tList
    Phases within reduction tree
```

```
std::vector<Matrix<Base<F>>> dList
    Signatures within reduction tree
```

```
qr::TreeData<F> qr::TS(const DistMatrix<F, U, STAR> &A)
    Forms an implicit tall-skinny QR decomposition.
```

```
void qr::ExplicitTS(DistMatrix<F, U, STAR> &A, DistMatrix<F, STAR, STAR> &R)
    Forms an explicit QR decomposition using a tall-skinny algorithm: A is overwritten with Q.
```

```
qr::ts namespace
```

```
DistMatrix<F, STAR, STAR> qr::ts::FormR(const DistMatrix<F, U, STAR> &A, const
                                      qr::TreeData<F> &treeData)
    Return the R from the QR decomposition.
```

```
void qr::ts::FormQ(DistMatrix<F, U, STAR> &A, qr::TreeData<F> &treeData)
    Overwrite A with the Q from the QR decomposition.
```

5.3.6 Generalized QR factorization

Implementation

The *generalized QR factorization* of a pair of matrices (A, B) is analogous to a QR factorization of $B^{-1}A$ but does not require that B is square or invertible: unitary matrices Q and Z , and (right) upper-triangular matrices R and T , are computed such that

$$A = QR$$

and

$$B = QTZ.$$

Thus, if B was square and invertible, then the QR factorization of $B^{-1}A$ would be given by $Z^H(T^{-1}R)$.

```
void GQR(Matrix<F> &A, Matrix<F> &B)
```

```
void GQR(DistMatrix<F> &A, DistMatrix<F> &B)
    Overwrite A with R and B with T.
```

```
void GQR(Matrix<F> &A, Matrix<F> &tA, Matrix<Base<F>> &dA, Matrix<F> &B, Matrix<F> &tB, Matrix<Base<F>> &dB)
```

```
void GQR(DistMatrix<F> &A, DistMatrix<F, MD, STAR> &tA, DistMatrix<Base<F>, MD, STAR> &dA, DistMatrix<F> &B, DistMatrix<F, MD, STAR> &tB, DistMatrix<Base<F>, MD, STAR> &dB)
```

Overwrite A with both R and the (scaled) Householder vectors which, along with the scalings tA and sign changes dA , define Q . Likewise, B is overwritten with both T and the (scaled) Householder vectors which define Z .

5.3.7 RQ factorization

Main header file

Subroutines

Test driver

Just like an LQ factorization, but the orthogonalization process starts from the bottom row and produces a much sparser triangular factor when the matrix is wider than it is tall.

```
void RQ(Matrix<F> &A)
```

```
void RQ(DistMatrix<F> &A)
```

```
void RQ(Matrix<F> &A, Matrix<F> &t, Matrix<Base<F>> &d)
```

```
void RQ(DistMatrix<F> &A, DistMatrix<F, MD, STAR> &t, DistMatrix<Base<F>, MD, STAR> &d)
```

Overwrite the matrix A with R and the Householder reflectors representing \hat{Q} . The scalings for the Householder reflectors are stored in the vector t and the unitary diagonal matrix which forces R to be positive is defined by the vector d .

rq namespace

```
void rq::ApplyQ(LeftOrRight side, Orientation orientation, const Matrix<F> &A, const
                Matrix<F> &t, const Matrix<Base<F>> &d, Matrix<F> &B)
```

```
void rq::ApplyQ(LeftOrRight side, Orientation orientation, const DistMatrix<F> &A,
                const DistMatrix<F, Ut, Vt> &t, const DistMatrix<Base<F>, Ud, Vd>
                &d, DistMatrix<F> &B)
```

Applies the implicitly-defined Q (or its adjoint) stored within A , t , and d from either the left or the right to B .

```
void rq::SolveAfter(Orientation orientation, const Matrix<F> &A, const Matrix<F> &t,
                   const Matrix<Base<F>> &d, const Matrix<F> &B, Matrix<F>
                   &X)
```

```
void rq::SolveAfter(Orientation orientation, const DistMatrix<F> &A, const DistMa-
                    trix<F, MD, STAR> &t, const DistMatrix<Base<F>, MD, STAR>
                    &d, const DistMatrix<F> &B, DistMatrix<F> &X)
```

Solves a set of linear systems using an existing packed RQ factorization given by A and the vectors t and d . B is the matrix of input vectors and X is the matrix of solutions.

5.3.8 Generalized RQ factorization**Implementation**

The *generalized RQ factorization* of a pair of matrices (A, B) is analogous to an RQ factorization of AB^{-1} but does not require that B is square or invertible: unitary matrices Q and Z , and (right) upper-triangular matrices R and T , are computed such that

$$A = RQ$$

and

$$B = ZTQ.$$

Thus, if B was square and invertible, then the RQ factorization of AB^{-1} would be given by $(RT^{-1})Z^H$.

```
void GRQ(Matrix<F> &A, Matrix<F> &B)
```

```
void GRQ(DistMatrix<F> &A, DistMatrix<F> &B)
```

Overwrite A with R and B with T .

```
void GRQ(Matrix<F> &A, Matrix<F> &tA, Matrix<Base<F>> &dA, Matrix<F> &B, Ma-
        trix<F> &tB, Matrix<Base<F>> &dB)
```

```
void GRQ(DistMatrix<F> &A, DistMatrix<F, MD, STAR> &tA, DistMatrix<Base<F>,
        MD, STAR> &dA, DistMatrix<F> &B, DistMatrix<F, MD, STAR> &tB, Dist-
        Matrix<Base<F>, MD, STAR> &dB)
```

Overwrite A with both R and the (scaled) Householder vectors which, along with the scalings tA and sign changes dA , define Q . Likewise, B is overwritten with both T and the (scaled) Householder vectors which define Z .

5.3.9 Interpolative Decomposition (ID)**Implementation**

Example driver

Interpolative Decompositions (ID's) are closely related to pivoted QR factorizations and are useful for representing (approximately) low-rank matrices in terms of linear combinations of a few of their columns, i.e.,

$$AP = \hat{A} (I \ Z),$$

where P is a permutation matrix, \hat{A} is a small set of columns of A , and Z is an interpolation matrix responsible for representing the remaining columns in terms of the selected columns of A .

```
void ID(const Matrix<F> &A, Matrix<int> &p, Matrix<F> &Z, int numSteps)
```

```
void ID(const DistMatrix<F> &A, DistMatrix<int, UPerm, STAR> &p, DistMatrix<F,
    STAR, VR> &Z, int numSteps)
    numSteps steps of a pivoted QR factorization are used to return an Interpolative Decomposition of A.
```

```
void ID(const Matrix<F> &A, Matrix<int> &p, Matrix<F> &Z, int maxSteps, Base<F>
    tol)
```

```
void ID(const DistMatrix<F> &A, DistMatrix<int, UPerm, STAR> &p, DistMatrix<F,
    STAR, VR> &Z, int maxSteps, Base<F> tol)
    Either maxSteps steps of a pivoted QR factorization are used, or execution stopped after the maximum remaining column norm was less than or equal to tol times the maximum original column norm.
```

5.3.10 Skeleton decomposition

Implementation

Example driver

Skeleton decompositions are essentially two-sided interpolative decompositions, but the terminology is unfortunately extremely contested. We follow the convention that a skeleton decomposition is an approximation

$$A \approx A_C Z A_R,$$

where A_C is a (small) selection of columns of A , A_R is a (small) selection of rows of A , and Z is a (small) square matrix. When Z is allowed to be rectangular, it is more common to call this a CUR decomposition.

```
void Skeleton(const Matrix<F> &A, Matrix<int> &pR, Matrix<int> &pC, Matrix<F>
    &Z, int maxSteps, Base<F> tol)
```

```
void Skeleton(const DistMatrix<F> &A, DistMatrix<int, UPerm, STAR> &pR, DistMa-
    trix<int, UPerm, STAR> &pC, int maxSteps, Base<F> tol)
```

Rather than returning A_R and A_C , the permutation matrices which implicitly define them are returned instead. At most $maxSteps$ steps of a pivoted QR decomposition will be used in order to generate the row / column subsets, and less steps will be taken if a pivot norm is less than or equal to $tolerance$ times the first pivot norm.

5.4 Matrix functions

5.4.1 Direct inversion

Triangular

Inverts a (possibly unit-diagonal) triangular matrix in-place.

```
void TriangularInverse(UpperOrLower uplo, UnitOrNonUnit diag, Matrix<F> &A)
```

```
void TriangularInverse(UpperOrLower uplo, UnitOrNonUnit diag, DistMatrix<F> &A)
```

Inverts the triangle of A specified by the parameter *uplo*; if *diag* is set to *UNIT*, then A is treated as unit-diagonal.

General

This routine computes the in-place inverse of a general fully-populated (invertible) matrix A as

$$A^{-1} = U^{-1}L^{-1}P,$$

where $PA = LU$ is the result of LU factorization with partial pivoting. The algorithm essentially factors A , inverts U in place, solves against L one block column at a time, and then applies the row pivots in reverse order to the columns of the result.

```
void Inverse(Matrix<F> &A)
```

```
void Inverse(DistMatrix<F> &A)
```

Overwrites the general matrix A with its inverse.

Symmetric/Hermitian

```
void SymmetricInverse(UpperOrLower uplo, Matrix<F> &A, bool conjugate = false,
                     LDLPivotType pivotType = BUNCH_KAUFMAN_A)
```

```
void SymmetricInverse(UpperOrLower uplo, DistMatrix<F> &A, bool conjugate = false,
                     LDLPivotType pivotType = BUNCH_KAUFMAN_A)
```

Invert a symmetric or Hermitian matrix using a pivoted LDL factorization.

```
void HermitianInverse(UpperOrLower uplo, Matrix<F> &A, bool conjugate = false,
                     LDLPivotType pivotType = BUNCH_KAUFMAN_A)
```

```
void HermitianInverse(UpperOrLower uplo, DistMatrix<F> &A, bool conjugate = false,
                     LDLPivotType pivotType = BUNCH_KAUFMAN_A)
```

Invert a Hermitian matrix using a pivoted LDL factorization.

HPD

This routine uses a custom algorithm for computing the inverse of a Hermitian positive-definite matrix A as

$$A^{-1} = (LL^H)^{-1} = L^{-H}L^{-1},$$

where L is the lower Cholesky factor of A (the upper Cholesky factor is computed in the case of upper-triangular storage). Rather than performing Cholesky factorization, triangular inversion, and then the Hermitian triangular outer product in sequence, this routine makes use of the single-sweep algorithm described in Bientinesi et al.'s "Families of algorithms related to the inversion of a symmetric positive definite matrix", in particular, the variant 2 algorithm from Fig. 9.

If the matrix is found to not be HPD, then a *NonHPDMatrixException* is thrown.

```
void HPDInverse(UpperOrLower uplo, Matrix<F> &A)
```

```
void HPDInverse(UpperOrLower uplo, DistMatrix<F> &A)
```

Overwrite the *uplo* triangle of the HPD matrix A with the same triangle of the inverse of A .

5.4.2 Hermitian functions

Reform the matrix with the eigenvalues modified by a user-defined function. When the user-defined function is real-valued, the result will remain Hermitian, but when the function is complex-valued, the result is best characterized as normal.

When the user-defined function, say f , is analytic, we can say much more about the result: if the eigenvalue decomposition of the Hermitian matrix A is $A = Z\Lambda Z^H$, then

$$f(A) = f(Z\Lambda Z^H) = Zf(\Lambda)Z^H.$$

Two important special cases are $f(\lambda) = \exp(\lambda)$ and $f(\lambda) = \exp(i\lambda)$, where the former results in a Hermitian matrix and the latter in a normal (in fact, unitary) matrix.

Note: Since Elemental currently depends on PMRRR for its tridiagonal eigensolver, only double-precision results are supported as of now.

```
void RealHermitianFunction(UpperOrLower uplo, Matrix<F> &A, const RealFunc-  
tor &f)
```

```
void RealHermitianFunction(UpperOrLower uplo, DistMatrix<F> &A, const RealFunc-  
tor &f)
```

Modifies the eigenvalues of the passed-in Hermitian matrix by replacing each eigenvalue λ_i with $f(\lambda_i) \in \mathbb{R}$. *RealFunc* is any class which has the member function `Real operator()(Real omega) const`.

```
void ComplexHermitianFunction(UpperOrLower uplo, Matrix<Complex<Real>> &A,  
const ComplexFunc-  
tor &f)
```

```
void ComplexHermitianFunction(UpperOrLower uplo, DistMatrix<Complex<Real>>  
&A, const ComplexFunc-  
tor &f)
```

Modifies the eigenvalues of the passed-in complex Hermitian matrix by replacing each eigenvalue λ_i with $f(\lambda_i) \in \mathbb{C}$. *ComplexFunc* can be any class which has the member function `Complex<Real> operator()(Real omega) const`.

TODO: A version of `ComplexHermitianFunction` which begins with a real matrix

5.4.3 Pseudoinverse

```
Pseudoinverse(Matrix<F> &A, Base<F> tolerance = 0)
```


`Pseudoinverse(DistMatrix<F> &A, Base<F> tolerance = 0)`

Computes the pseudoinverse of a general matrix through computing its SVD, modifying the singular values with the function

$$f(\sigma) = \begin{cases} 1/\sigma, & \sigma \geq \epsilon n \|A\|_2 \\ 0, & \text{otherwise} \end{cases},$$

where ϵ is the relative machine precision, n is the height of A , and $\|A\|_2$ is the maximum singular value. If a nonzero value for *tolerance* was specified, it is used instead of $\epsilon n \|A\|_2$.

`HermitianPseudoinverse(UpperOrLower uplo, Matrix<F> &A, Base<F> tolerance = 0)`

`HermitianPseudoinverse(UpperOrLower uplo, DistMatrix<F> &A, Base<F> tolerance = 0)`

Computes the pseudoinverse of a Hermitian matrix through a customized version of `RealHermitianFunction()` which used the eigenvalue mapping function

$$f(\omega) = \begin{cases} 1/\omega, & |\omega| \geq \epsilon n \|A\|_2 \\ 0, & \text{otherwise} \end{cases},$$

where ϵ is the relative machine precision, n is the height of A , and $\|A\|_2$ can be computed as the maximum absolute value of the eigenvalues of A . If a nonzero value for *tolerance* is specified, it is used instead of $\epsilon n \|A\|_2$.

5.4.4 Square root

A matrix B satisfying

$$B^2 = A$$

is referred to as the *square-root* of the matrix A . Such a matrix is guaranteed to exist as long as A is diagonalizable: if $A = X\Lambda X^{-1}$, then we may put

$$B = X\sqrt{\Lambda}X^{-1},$$

where each eigenvalue $\lambda = re^{i\theta}$ maps to $\sqrt{\lambda} = \sqrt{r}e^{i\theta/2}$.

`void SquareRoot(Matrix<F> &A)`

`void SquareRoot(DistMatrix<F> &A)`

Currently uses a Newton iteration to compute the general matrix square-root. See `square_root::Newton` for the more detailed interface.

`void HPSDSquareRoot(UpperOrLower uplo, Matrix<F> &A)`

`void HPSDSquareRoot(UpperOrLower uplo, DistMatrix<F> &A)`

Computes the Hermitian EVD, square-roots the eigenvalues, and then reforms the matrix. If any of the eigenvalues were sufficiently negative, a `NonHPSDMatrixException` is thrown.

TODO: HermitianSquareRoot

square_root namespace

`int square_root::Newton(Matrix<F> &A, int maxIts = 100, Base<F> tol = 0)`

`int square_root::Newton(DistMatrix<F> &A, int maxIts = 100, Base<F> tol = 0)`

Performs at most `maxIts` Newton steps in an attempt to compute the matrix square-root within the specified tolerance, which defaults to $n\epsilon$, where n is the matrix height and ϵ is the machine precision.

5.4.5 Sign

The matrix sign function can be written as

$$\text{sgn}(A) = A(A^2)^{-1/2},$$

as long as A does not have any pure-imaginary eigenvalues.

type SignScaling

An enum which can be set to one of

- SIGN_SCALE_NONE
- SIGN_SCALE_DET
- SIGN_SCALE_FROB

type SignCtrl<Real>

int maxIts

Real tol

Real power

SignScaling scaling

bool progress

type SignCtrl<Base<F>>

A particular case where the datatype is the base of the potentially complex datatype F .

`void Sign(Matrix<F> &A, SignCtrl<Base<F>> signCtrl = SignCtrl<Base<F>>())`

`void Sign(DistMatrix<F> &A, SignCtrl<Base<F>> signCtrl = SignCtrl<Base<F>>())`

`void Sign(Matrix<F> &A, Matrix<F> &N, SignCtrl<Base<F>> signCtrl = SignCtrl<Base<F>>())`

`void Sign(DistMatrix<F> &A, DistMatrix<F> &N, SignCtrl<Base<F>> signCtrl = SignCtrl<Base<F>>())`

Compute the matrix sign through a globally-convergent Newton iteration scaled with the Frobenius norm of the iterate and its inverse. Optionally return the full decomposition, $A = SN$, where A is overwritten by S .

`void HermitianSign(UpperOrLower uplo, Matrix<F> &A)`

`void HermitianSign(UpperOrLower uplo, DistMatrix<F> &A)`

`void HermitianSign(UpperOrLower uplo, Matrix<F> &A, Matrix<F> &N)`

`void HermitianSign(UpperOrLower uplo, DistMatrix<F> &A, DistMatrix<F> &N)`

Compute the Hermitian EVD, replace the eigenvalues with their sign, and then reform the matrix. Optionally return the full decomposition, $A = SN$, where A is overwritten by S . Note that this will also be a polar decomposition.

5.5 Permutations

5.5.1 Permutation vectors

In order to represent arbitrary permutations, Elemental recently switched from LAPACK-style pivot sequence representations to general permutation vectors, where entry i of the *permutation vector* p contains the column index of the nonzero entry in row i of the corresponding permutation matrix.

PermuteCols

```
void PermuteCols(Matrix<T> &A, const Matrix<int> &perm)
```

```
void PermuteCols(DistMatrix<T, U, V> &A, const DistMatrix<int, UPerm, STAR>
                 &perm)
```

```
void PermuteCols(Matrix<T> &A, const Matrix<int> &perm, const Matrix<int> &invPerm)
```

```
void PermuteCols(DistMatrix<T, U, V> &A, const DistMatrix<int, UPerm, STAR>
                 &perm, const DistMatrix<int, UPerm, STAR> &invPerm)
```

Provide the inverse permutations so that they do not need to be internally computed and discarded.

PermuteRows

```
void PermuteRows(Matrix<T> &A, const Matrix<int> &perm)
```

```
void PermuteRows(DistMatrix<T, U, V> &A, const DistMatrix<int, UPerm, STAR>
                 &perm)
```

```
void PermuteRows(Matrix<T> &A, const Matrix<int> &perm, const Matrix<int> &invPerm)
```

```
void PermuteRows(DistMatrix<T, U, V> &A, const DistMatrix<int, UPerm, STAR>
                 &perm, const DistMatrix<int, UPerm, STAR> &invPerm)
```

Provide the inverse permutations so that they do not need to be internally computed and discarded.

InversePermuteCols

```
void InversePermuteCols(Matrix<T> &A, const Matrix<int> &perm)
```

```
void InversePermuteCols(DistMatrix<T, U, V> &A, const DistMatrix<int, UPerm,
                       STAR> &perm)
```

InversePermuteRows

```
void InversePermuteRows(Matrix<T> &A, const Matrix<int> &perm)
```

```
void InversePermuteRows(DistMatrix<T, U, V> &A, const DistMatrix<int, UPerm,
                       STAR> &perm)
```

InvertPermutation

```
void InvertPermutation(const Matrix<int> &perm, Matrix<int> &invPerm)
```

```
void InvertPermutation(const DistMatrix<int, UPerm, STAR> &perm, DistMatrix<int, UPerm, STAR> &invPerm)
```

Compute the inverse permutation matrix in compressed (vector) form.

ExplicitPermutation

```
void ExplicitPermutation(const Matrix<int> &perm, Matrix<int> &P)
```

```
void ExplicitPermutation(const DistMatrix<int, UPerm, STAR> &perm, DistMatrix<int, U, V> &P)
```

Return the full permutation matrix, P , represented by the permutation vector $perm$.

PermutationParity

```
bool PermutationParity(const Matrix<int> &perm)
```

```
bool PermutationParity(const DistMatrix<int, UPerm, STAR> &perm)
```

Return true if the given permutation is odd. The parity is determined in linear time by finding the decomposition of the inverse of the permutation as a product of transpositions.

PermutationMeta

```
type PermutationMeta
```

```
int align
```

```
mpi::Comm comm
```

```
std::vector<int> sendCounts
```

```
std::vector<int> sendDispls
```

```
std::vector<int> sendIdx
```

```
std::vector<int> sendRanks
```

```
std::vector<int> recvCounts
```

```
std::vector<int> recvDispls
```

```
std::vector<int> recvIdx
```

```
std::vector<int> recvRanks
```

```
int TotalSend() const
```

```
int TotalRecv() const
```

```
void ScaleUp(int length)
```

```
void ScaleDown(int length)
```

```
PermutationMeta(const DistMatrix<int, U, STAR> &perm, const DistMatrix<int,
                U, STAR> &invPerm)
```

5.5.2 Pivot sequences

These routines make use of LAPACK-style pivot sequence vectors, where the *pivot sequence vector* p implies the sequence of swaps $(0, p_0), (1, p_1), \dots, (n-1, p_{n-1})$. Elemental used to follow this convention when returning permutations from factorizations, but clearly this representation is somewhat restrictive, as routines which perform multiple swaps for each pivot (e.g., some variants of Bunch-Kaufman) cannot be handled.

ApplyColPivots

```
void ApplyColPivots(Matrix<T> &A, const Matrix<int> &pivots)
void ApplyColPivots(DistMatrix<T, U1, V1> &A, const DistMatrix<int, U2, V2> &pivots)
```

ApplyInverseColPivots

```
void ApplyInverseColPivots(Matrix<T> &A, const Matrix<int> &pivots)
void ApplyInverseColPivots(DistMatrix<T, U1, V1> &A, const DistMatrix<int, U2,
                          V2> &pivots)
```

ApplyRowPivots

```
void ApplyRowPivots(Matrix<T> &A, const Matrix<int> &pivots)
void ApplyRowPivots(DistMatrix<T, U1, V1> &A, const DistMatrix<int, U2, V2> &pivots)
```

ApplyInverseRowPivots

```
void ApplyInverseRowPivots(Matrix<T> &A, const Matrix<int> &pivots)
void ApplyInverseRowPivots(DistMatrix<T, U1, V1> &A, const DistMatrix<int, U2,
                          V2> &pivots)
```

ApplySymmetricPivots

```
void ApplySymmetricPivots(UpperOrLower uplo, Matrix<T> &A, const Matrix<int>
                          &pivots, bool conjugate = false)
void ApplySymmetricPivots(UpperOrLower uplo, DistMatrix<T> &A, const DistMatrix<int, UPerm,
                          STAR> &pivots, bool conjugate = false)
```

ApplyInverseSymmetricPivots

```
void ApplyInverseSymmetricPivots(UpperOrLower uplo, Matrix<T> &A, const Ma-  
trix<int> &pivots, bool conjugate = false)
```

```
void ApplyInverseSymmetricPivots(UpperOrLower uplo, DistMatrix<T> &A, const  
DistMatrix<int, UPerm, STAR> &pivots, bool con-  
jugate = false)
```

PivotParity

```
bool PivotParity(const Matrix<int> &pivots)
```

```
bool PivotParity(const DistMatrix<int, UPerm, STAR> &pivots)
```

Return true if the permutation implied by the pivot sequence is odd. This is determined by determining if an odd number of non-trivial transpositions are performed.

5.5.3 Converting pivot sequences to permutations

PivotsToPermutation

```
void PivotsToPermutation(const Matrix<int> &pivots, Matrix<int> &perm, int offset =  
0)
```

```
void PivotsToPermutation(const DistMatrix<int, U, STAR> &pivots, DistMatrix<int,  
UPerm, STAR> &perm, int offset = 0)
```

PivotsToInversePermutation

```
void PivotsToInversePermutation(const Matrix<int> &pivots, Matrix<int> &in-  
vPerm, int offset = 0)
```

```
void PivotsToInversePermutation(const DistMatrix<int, U, STAR> &pivots, DistMa-  
trix<int, UPerm, STAR> &invPerm, int offset = 0)
```

PivotsToPartialPermutation

```
void PivotsToPartialPermutation(const Matrix<int> &pivots, Matrix<int> &perm,  
Matrix<int> &invPerm, int offset = 0)
```

```
void PivotsToPartialPermutation(const DistMatrix<int, U, STAR> &pivots, DistMa-  
trix<int, UPerm, STAR> &perm, DistMatrix<int,  
UPerm, STAR> &invPerm, int offset = 0)
```

5.6 Matrix properties

5.6.1 Condition number

Main header file

Subroutines

The condition number of a matrix with respect to a particular norm is

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

with the most common choice being the matrix two-norm.

Base<F> `Condition(const Matrix<F> &A, NormType type = TWO_NORM)`

Base<F> `Condition(const DistMatrix<F, U, V> &A, NormType type = TWO_NORM)`
Returns the condition number with respect to the specified norm (one, two, or Frobenius).

Base<F> `FrobeniusCondition(const Matrix<F> &A)`

Base<F> `FrobeniusCondition(const DistMatrix<F, U, V> &A)`
Returns the condition number with respect to the Frobenius norm.

Base<F> `InfinityCondition(const Matrix<F> &A)`

Base<F> `InfinityCondition(const DistMatrix<F, U, V> &A)`
Returns the condition number with respect to the infinity norm.

Base<F> `MaxCondition(const Matrix<F> &A)`

Base<F> `MaxCondition(const DistMatrix<F, U, V> &A)`
Returns the condition number with respect to the entrywise maximum norm.

Base<F> `OneCondition(const Matrix<F> &A)`

Base<F> `OneCondition(const DistMatrix<F, U, V> &A)`
Returns the condition number with respect to the one norm.

Base<F> `TwoCondition(const Matrix<F> &A)`

Base<F> `TwoCondition(const DistMatrix<F, U, V> &A)`
Returns the condition number with respect to the two norm.

5.6.2 Determinant

Main header file

Subroutines

Though there are many different possible definitions of the determinant of a matrix $A \in \mathbb{F}^{n \times n}$, the simplest one is in terms of the product of the eigenvalues (including multiplicity):

$$\det(A) = \prod_{i=0}^{n-1} \lambda_i.$$

General

Since $\det(AB) = \det(A)\det(B)$, we can compute the determinant of an arbitrary matrix in $\mathcal{O}(n^3)$ work by computing its LU decomposition (with partial pivoting), $PA = LU$, recognizing that $\det(P) = \pm 1$ (the *signature* of the permutation), and computing

$$\det(A) = \det(P)\det(L)\det(U) = \det(P) \prod_{i=0}^{n-1} v_{i,i} = \pm \prod_{i=0}^{n-1} v_{i,i},$$

where $v_{i,i}$ is the i 'th diagonal entry of U .

F Determinant(**const** *Matrix*<F> &A)

F Determinant(**const** *DistMatrix*<F> &A)

F Determinant(*Matrix*<F> &A, bool *canOverwrite* = false)

F Determinant(*DistMatrix*<F> &A, bool *canOverwrite* = false)

The determinant of the (fully populated) square matrix *A*. Some of the variants allow for overwriting the input matrix in order to avoid forming another temporary matrix.

type SafeProduct<F>

Represents the product of *n* values as $\rho \exp(\kappa n)$, where $|\rho| \leq 1$ and $\kappa \in \mathbb{R}$.

F rho

For nonzero values, *rho* is the modulus and lies *on* the unit circle; in order to represent a value that is precisely zero, *rho* is set to zero.

Base<F> kappa

kappa can be an arbitrary real number.

int n

The number of values in the product.

SafeProduct<F> SafeDeterminant(**const** *Matrix*<F> &A)

SafeProduct<F> SafeDeterminant(**const** *DistMatrix*<F> &A)

SafeProduct<F> SafeDeterminant(*Matrix*<F> &A, bool *canOverwrite* = false)

SafeProduct<F> SafeDeterminant(*DistMatrix*<F> &A, bool *canOverwrite* = false)

The determinant of the square matrix *A* in an expanded form which is less likely to over/under-flow.

HPD

A version of the above determinant specialized for Hermitian positive-definite matrices (which will therefore have all positive eigenvalues and a positive determinant).

Base<F> HPDDeterminant(*UpperOrLower* *uplo*, **const** *Matrix*<F> &A)

Base<F> HPDDeterminant(*UpperOrLower* *uplo*, **const** *DistMatrix*<F> &A)

Base<F> HPDDeterminant(*UpperOrLower* *uplo*, *Matrix*<F> &A, bool *canOverwrite* = false)

Base<F> HPDDeterminant(*UpperOrLower* *uplo*, *DistMatrix*<F> &A, bool *canOverwrite* = false)

The determinant of the (fully populated) Hermitian positive-definite matrix *A*. Some of the variants allow for overwriting the input matrix in order to avoid forming another temporary matrix.

SafeProduct<F> SafeHPDDeterminant(*UpperOrLower* *uplo*, **const** *Matrix*<F> &A)

SafeProduct<F> SafeHPDDeterminant(*UpperOrLower* *uplo*, **const** *DistMatrix*<F> &A)

SafeProduct<F> SafeHPDDeterminant(*UpperOrLower* *uplo*, *Matrix*<F> &A, bool *canOverwrite* = false)

SafeProduct<F> *SafeHPDDeterminant*(*UpperOrLower* *uplo*, *DistMatrix*<F> &*A*, bool
canOverwrite = false)

The determinant of the Hermitian positive-definite matrix *A* in an expanded form which is less likely to over/under-flow.

5.6.3 Matrix norms

Main header file

Subroutines

The following routines can return either $\|A\|_1$, $\|A\|_\infty$, $\|A\|_F$ (the Frobenius norm), the maximum entrywise norm, $\|A\|_2$, or $\|A\|_*$ (the nuclear/trace norm) of fully-populated matrices.

Base<F> *Norm*(const *Matrix*<F> &*A*, *NormType* *type* = FROBENIUS_NORM)

Base<F> *Norm*(const *DistMatrix*<F, U, V> &*A*, *NormType* *type* = FROBENIUS_NORM)

Base<F> *HermitianNorm*(*UpperOrLower* *uplo*, const *Matrix*<F> &*A*, *NormType* *type* = FROBENIUS_NORM)

Base<F> *HermitianNorm*(*UpperOrLower* *uplo*, const *DistMatrix*<F, U, V> &*A*, *NormType* *type* = FROBENIUS_NORM)

Base<F> *SymmetricNorm*(*UpperOrLower* *uplo*, const *Matrix*<F> &*A*, *NormType* *type* = FROBENIUS_NORM)

Base<F> *SymmetricNorm*(*UpperOrLower* *uplo*, const *DistMatrix*<F, U, V> &*A*, *NormType* *type* = FROBENIUS_NORM)

Compute a norm of a fully-populated or implicitly symmetric/Hermitian (with the data stored in the specified triangle) matrix.

Alternatively, one may directly call the following routines (note that the entrywise, KyFan, and Schatten norms have an extra parameter and must be called directly).

Base<F> *EntrywiseNorm*(const *Matrix*<F> &*A*, *Base*<F> *p*)

Base<F> *EntrywiseNorm*(const *DistMatrix*<F, U, V> &*A*, *Base*<F> *p*)

Base<F> *HermitianEntrywiseNorm*(*UpperOrLower* *uplo*, const *Matrix*<F> &*A*, *Base*<F> *p*)

Base<F> *HermitianEntrywiseNorm*(*UpperOrLower* *uplo*, const *DistMatrix*<F, U, V> &*A*, *Base*<F> *p*)

Base<F> *SymmetricEntrywiseNorm*(*UpperOrLower* *uplo*, const *Matrix*<F> &*A*, *Base*<F> *p*)

Base<F> *SymmetricEntrywiseNorm*(*UpperOrLower* *uplo*, const *DistMatrix*<F, U, V> &*A*, *Base*<F> *p*)

The ℓ_p norm of the columns of *A* stacked into a single vector. Note that the Frobenius norm corresponds to the $p = 2$ case.

Base<F> *EntrywiseOneNorm*(const *Matrix*<F> &*A*)

Base<F> *EntrywiseOneNorm*(const *DistMatrix*<F, U, V> &*A*)

Base<F> *HermitianEntrywiseOneNorm*(*UpperOrLower* *uplo*, const *Matrix*<F> &*A*)

Base<F> *HermitianEntrywiseOneNorm*(*UpperOrLower* *uplo*, const *DistMatrix*<F, U, V> &*A*)

Base<F> *SymmetricEntrywiseOneNorm*(*UpperOrLower uplo*, **const** *Matrix*<F> &A)

Base<F> *SymmetricEntrywiseOneNorm*(*UpperOrLower uplo*, **const** *DistMatrix*<F, U, V> &A)

The ℓ_1 norm of the columns of A stacked into a single vector.

Base<F> *FrobeniusNorm*(**const** *Matrix*<F> &A)

Base<F> *FrobeniusNorm*(**const** *DistMatrix*<F, U, V> &A)

Base<F> *HermitianFrobeniusNorm*(*UpperOrLower uplo*, **const** *Matrix*<F> &A)

Base<F> *HermitianFrobeniusNorm*(*UpperOrLower uplo*, **const** *DistMatrix*<F, U, V> &A)

Base<F> *SymmetricFrobeniusNorm*(*UpperOrLower uplo*, **const** *Matrix*<F> &A)

Base<F> *SymmetricFrobeniusNorm*(*UpperOrLower uplo*, **const** *DistMatrix*<F, U, V> &A)

The ℓ_2 norm of the singular values (the Schatten norm with $p = 2$), which can be cheaply computed as the ℓ_2 norm of $\text{vec}(A)$.

Base<F> *KyFanNorm*(**const** *Matrix*<F> &A, int k)

Base<F> *KyFanNorm*(**const** *DistMatrix*<F, U, V> &A, int k)

Base<F> *HermitianKyFanNorm*(*UpperOrLower uplo*, **const** *Matrix*<F> &A, int k)

Base<F> *HermitianKyFanNorm*(*UpperOrLower uplo*, **const** *DistMatrix*<F, U, V> &A, int k)

Base<F> *SymmetricKyFanNorm*(*UpperOrLower uplo*, **const** *Matrix*<F> &A, int k)

Base<F> *SymmetricKyFanNorm*(*UpperOrLower uplo*, **const** *DistMatrix*<F, U, V> &A, int k)

The sum of the largest k singular values.

Base<F> *InfinityNorm*(**const** *Matrix*<F> &A)

Base<F> *InfinityNorm*(**const** *DistMatrix*<F, U, V> &A)

Base<F> *HermitianInfinityNorm*(*UpperOrLower uplo*, **const** *Matrix*<F> &A)

Base<F> *HermitianInfinityNorm*(*UpperOrLower uplo*, **const** *DistMatrix*<F, U, V> &A)

Base<F> *SymmetricInfinityNorm*(*UpperOrLower uplo*, **const** *Matrix*<F> &A)

Base<F> *SymmetricInfinityNorm*(*UpperOrLower uplo*, **const** *DistMatrix*<F, U, V> &A)

The maximum ℓ_1 norm of the rows of A . In the symmetric and Hermitian cases, this is equivalent to the $\|\cdot\|_1$ norm.

Base<F> *MaxNorm*(**const** *Matrix*<F> &A)

Base<F> *MaxNorm*(**const** *DistMatrix*<F, U, V> &A)

Base<F> *HermitianMaxNorm*(*UpperOrLower uplo*, **const** *Matrix*<F> &A)

Base<F> *HermitianMaxNorm*(*UpperOrLower uplo*, **const** *DistMatrix*<F, U, V> &A)

Base<F> *SymmetricMaxNorm*(*UpperOrLower uplo*, **const** *Matrix*<F> &A)

Base<F> *SymmetricMaxNorm*(*UpperOrLower uplo*, **const** *DistMatrix*<F, U, V> &A)

The maximum absolute value of the matrix entries.

Base<F> *NuclearNorm*(**const** *Matrix*<F> &A)

Base<F> *NuclearNorm*(**const** *DistMatrix*<F, U, V> &A)

Base<F> HermitianNuclearNorm(*UpperOrLower uplo*, **const Matrix**<F> &A)

Base<F> HermitianNuclearNorm(*UpperOrLower uplo*, **const DistMatrix**<F, U, V> &A)

Base<F> SymmetricNuclearNorm(*UpperOrLower uplo*, **const Matrix**<F> &A)

Base<F> SymmetricNuclearNorm(*UpperOrLower uplo*, **const DistMatrix**<F, U, V> &A)

The sum of the singular values. This is equivalent to both the KyFan norm with $k = n$ and the Schatten norm with $p = 1$. Note that the nuclear norm is dual to the two-norm, which is the Schatten norm with $p = \infty$.

Base<F> OneNorm(**const Matrix**<F> &A)

Base<F> OneNorm(**const DistMatrix**<F, U, V> &A)

Base<F> HermitianOneNorm(*UpperOrLower uplo*, **const Matrix**<F> &A)

Base<F> HermitianOneNorm(*UpperOrLower uplo*, **const DistMatrix**<F, U, V> &A)

Base<F> SymmetricOneNorm(*UpperOrLower uplo*, **const Matrix**<F> &A)

Base<F> SymmetricOneNorm(*UpperOrLower uplo*, **const DistMatrix**<F, U, V> &A)

The maximum ℓ_1 norm of the columns of A . In the symmetric and Hermitian cases, this is equivalent to the $\|\cdot\|_\infty$ norm.

Base<F> SchattenNorm(**const Matrix**<F> &A, *Base*<F> p)

Base<F> SchattenNorm(**const DistMatrix**<F, U, V> &A, *Base*<F> p)

Base<F> HermitianSchattenNorm(*UpperOrLower uplo*, **const Matrix**<F> &A, *Base*<F> p)

Base<F> HermitianSchattenNorm(*UpperOrLower uplo*, **const DistMatrix**<F, U, V> &A, *Base*<F> p)

Base<F> SymmetricSchattenNorm(*UpperOrLower uplo*, **const Matrix**<F> &A, *Base*<F> p)

Base<F> SymmetricSchattenNorm(*UpperOrLower uplo*, **const DistMatrix**<F, U, V> &A, *Base*<F> p)

The ℓ_p norm of the singular values.

Base<F> TwoNorm(**const Matrix**<F> &A)

Base<F> TwoNorm(**const DistMatrix**<F, U, V> &A)

Base<F> HermitianTwoNorm(*UpperOrLower uplo*, **const Matrix**<F> &A)

Base<F> HermitianTwoNorm(*UpperOrLower uplo*, **const DistMatrix**<F, U, V> &A)

Base<F> SymmetricTwoNorm(*UpperOrLower uplo*, **const Matrix**<F> &A)

Base<F> SymmetricTwoNorm(*UpperOrLower uplo*, **const DistMatrix**<F, U, V> &A)

The maximum singular value. This is equivalent to the KyFan norm with k equal to one and the Schatten norm with $p = \infty$.

int ZeroNorm(**const Matrix**<F> &A)

int ZeroNorm(**const DistMatrix**<F, U, V> &A)

Return the number of nonzero entries in the matrix.

Note: Symmetric/Hermitian versions are not yet written

Two-norm estimates

`Base<F> TwoNormEstimate(Matrix<F> &A, Base<F> tol = 1e-6)`

`Base<F> TwoNormEstimate(DistMatrix<F> &A, Base<F> tol = 1e-6)`

`Base<F> HermitianTwoNormEstimate(Matrix<F> &A, Base<F> tol = 1e-6)`

`Base<F> HermitianTwoNormEstimate(DistMatrix<F> &A, Base<F> tol = 1e-6)`

`Base<F> SymmetricTwoNormEstimate(Matrix<F> &A, Base<F> tol = 1e-6)`

`Base<F> SymmetricTwoNormEstimate(DistMatrix<F> &A, Base<F> tol = 1e-6)`

Return an estimate for the two-norm which should be accurate within a factor of n times the specified tolerance.

5.6.4 Pseudospectra

Main header file

Subroutines

Pseudospectrum example driver

ChunkedPseudospectrum example driver

TriangularPseudospectrum example driver

ChunkedTriangularPseudospectrum example driver

The ϵ -*pseudospectrum* of a square matrix A is the set of all shifts z such that $\hat{A} - z$ is singular for some \hat{A} such that $\|\hat{A} - A\|_2 < \epsilon$. In other words, z is in the ϵ -pseudospectrum of A if the smallest singular value of $A - z$ is less than ϵ .

Elemental currently supports two methods for computing pseudospectra: the first is a high-performance improvement of Shiu-Hong Lui's triangularization followed by inverse iteration approach suggested in *Computation of pseudospectra by continuation* (please see Trefethen's *Computation of pseudospectra* for a comprehensive review). In particular, Elemental begins by computing the Schur decomposition of the given matrix, which preserves the ϵ -pseudospectrum, up to round-off error, and then simultaneously performs many Implicitly Restarted Arnoldi (IRA) iterations with the inverse normal matrix for each shift in a manner which communicates no more data than a standard triangular solve with many right-hand sides. Converged pseudospectrum estimates are deflated after convergence.

The second approach is quite similar and, instead of reducing to triangular form, reduces to Hessenberg form and performs multi-shift triangular solves in a manner similar to Greg Henry's *The shifted Hessenberg system solve computation* and Purkayastha et al.'s *A parallel algorithm for the Sylvester-Observer Equation*. This algorithm is not yet performance-tuned in Elemental, but should be preferred in massively-parallel situations where the Schur decomposition is considered infeasible.

```
class SnapshotCtrl
```

```
    int realSize
```

```
    int imagSize
```

```
    int imgSaveFreq
```

```

int numSaveFreq
int imgDispFreq
    Negative if no snapshots should be saved/displayed, zero if only a final snapshot
    should be saved/displayed, and equal to  $n > 0$  if, in addition to a final snapshot,
    the partial results should be output roughly every  $n$  iterations (there is no output in
    the middle of Implicitly Restarted Arnoldi cycles).

int imgSaveCount
int numSaveCount
int imgDispCount
std::string imgBase
std::string numBase
FileFormat imgFormat
FileFormat numFormat
SnapshotCtrl()
    All counters and dimensions are initially zero, all save/display "frequencies" are
    set to -1 (no output), the basename strings are initialized to "ps", the image format
    to PNG, and the numerical format to ASCII_MATLAB.

void ResetCounts()
    Resets all counters to zero

void Iterate()
    Increments all counters by one

```

```

class PseudospecCtrl<Real>

```

```

    bool forceComplexSchur
    bool forceComplexPs
    SdcCtrl<Real> sdcCtrl
    int maxIts
    Real tol
    bool deflate
    bool arnoldi
    int basisSize
    bool reorthog
    bool progress
    SnapshotCtrl snapCtrl

```

```

class PseudospecCtrl<Base<F>>

```

A particular case where the datatype is the base of the potentially complex type F.

```
Matrix<int> Pseudospectrum(const Matrix<F> &A, const Matrix<Complex<Base<F>>> &shifts, Matrix<Base<F>> &invNorms, PseudospecCtrl<Base<F>> psCtrl = PseudospecCtrl<Base<F>>())
```

```
DistMatrix<int, VR, STAR> Pseudospectrum(const DistMatrix<F> &A, const DistMatrix<Complex<Base<F>>, VR, STAR> &shifts, DistMatrix<Base<F>, VR, STAR> &invNorms, PseudospecCtrl<Base<F>> psCtrl = PseudospecCtrl<Base<F>>())
```

```
Matrix<int> TriangularPseudospectrum(const Matrix<F> &U, const Matrix<Complex<Base<F>>> &shifts, Matrix<Base<F>> &invNorms, PseudospecCtrl<Base<F>> psCtrl = PseudospecCtrl<Base<F>>())
```

```
DistMatrix<int, VR, STAR> TriangularPseudospectrum(const DistMatrix<F> &U, const DistMatrix<Complex<Base<F>>, VR, STAR> &shifts, DistMatrix<Base<F>, VR, STAR> &invNorms, PseudospecCtrl<Base<F>> psCtrl = PseudospecCtrl<Base<F>>())
```

```
DistMatrix<int, VR, STAR> QuasiTriangularPseudospectrum(const DistMatrix<Real> &U, const DistMatrix<Complex<Real>, VR, STAR> &shifts, DistMatrix<Real, VR, STAR> &invNorms, PseudospecCtrl<Real> psCtrl = PseudospecCtrl<Real>())
```

```
Matrix<int> HessenbergPseudospectrum(const Matrix<F> &H, const Matrix<Complex<Base<F>>> &shifts, Matrix<Base<F>> &invNorms, PseudospecCtrl<Base<F>> psCtrl = PseudospecCtrl<Base<F>>())
```

```
DistMatrix<int, VR, STAR> HessenbergPseudospectrum(const DistMatrix<F> &H, const DistMatrix<Complex<Base<F>>, VR, STAR> &shifts, DistMatrix<Base<F>, VR, STAR> &invNorms, PseudospecCtrl<Base<F>> psCtrl = PseudospecCtrl<Base<F>>())
```

Returns the norms of the shifted inverses in the vector `invNorms` for a given set of shifts. The returned integer vector is a list of the number of iterations required for convergence of each shift.

Matrix<int> Pseudospectrum(**const** *Matrix*<F> &A, *Matrix*<Base<F>> &invNormMap, *Complex*<Base<F>> center, int realSize, int imagSize, *PseudospecCtrl*<Base<F>> psCtrl = *PseudospecCtrl*<Base<F>>())

DistMatrix<int> Pseudospectrum(**const** *DistMatrix*<F> &A, *DistMatrix*<Base<F>> &invNormMap, *Complex*<Base<F>> center, int realSize, int imagSize, *PseudospecCtrl*<Base<F>> psCtrl = *PseudospecCtrl*<Base<F>>())

Matrix<int> TriangularPseudospectrum(**const** *Matrix*<F> &U, *Matrix*<Base<F>> &invNormMap, *Complex*<Base<F>> center, int realSize, int imagSize, *PseudospecCtrl*<Base<F>> psCtrl = *PseudospecCtrl*<Base<F>>())

DistMatrix<int> TriangularPseudospectrum(**const** *DistMatrix*<F> &U, *DistMatrix*<Base<F>> &invNormMap, *Complex*<Base<F>> center, int realSize, int imagSize, *PseudospecCtrl*<Base<F>> psCtrl = *PseudospecCtrl*<Base<F>>())

Matrix<int> QuasiTriangularPseudospectrum(**const** *Matrix*<Real> &U, *Matrix*<Real> &invNormMap, *Complex*<Real> center, int realSize, int imagSize, *PseudospecCtrl*<Real> psCtrl = *PseudospecCtrl*<Real>())

DistMatrix<int> QuasiTriangularPseudospectrum(**const** *DistMatrix*<Real> &U, *DistMatrix*<Real> &invNormMap, *Complex*<Real> center, int realSize, int imagSize, *PseudospecCtrl*<Real> psCtrl = *PseudospecCtrl*<Real>())

Matrix<int> HessenbergPseudospectrum(**const** *Matrix*<F> &H, *Matrix*<Base<F>> &invNormMap, *Complex*<Base<F>> center, int realSize, int imagSize, *PseudospecCtrl*<Base<F>> psCtrl = *PseudospecCtrl*<Base<F>>())

DistMatrix<int> HessenbergPseudospectrum(**const** *DistMatrix*<F> &H, *DistMatrix*<Base<F>> &invNormMap, *Complex*<Base<F>> center, int realSize, int imagSize, *PseudospecCtrl*<Base<F>> psCtrl = *PseudospecCtrl*<Base<F>>())

Returns the norms of the shifted inverses over a 2D grid (in the matrix *invNormMap*) with the specified x and y resolutions. The width of the grid in the complex plane is determined based upon the one and two norms of the Schur factor. The returned integer matrix corresponds to the number of iterations required for convergence at each shift in the 2D grid.

Matrix<int> Pseudospectrum(**const** *Matrix*<F> &A, *Matrix*<Base<F>> &invNormMap, *Complex*<Base<F>> center, *Base*<F> realWidth, *Base*<F> imagWidth, int realSize, int imagSize, *PseudospecCtrl*<Base<F>> psCtrl = *PseudospecCtrl*<Base<F>>())

DistMatrix<int> Pseudospectrum(**const** *DistMatrix*<F> &A, *DistMatrix*<Base<F>> &invNormMap, *Complex*<Base<F>> center, *Base*<F> realWidth, *Base*<F> imagWidth, int realSize, int imagSize, *PseudospecCtrl*<Base<F>> psCtrl = PseudospecCtrl<Base<F>>())

Matrix<int> TriangularPseudospectrum(**const** *Matrix*<F> &U, *Matrix*<Base<F>> &invNormMap, *Complex*<Base<F>> center, *Base*<F> realWidth, *Base*<F> imagWidth, int realSize, int imagSize, *PseudospecCtrl*<Base<F>> psCtrl = PseudospecCtrl<Base<F>>())

DistMatrix<int> TriangularPseudospectrum(**const** *DistMatrix*<F> &U, *DistMatrix*<Base<F>> &invNormMap, *Complex*<Base<F>> center, *Base*<F> realWidth, *Base*<F> imagWidth, int realSize, int imagSize, *PseudospecCtrl*<Base<F>> psCtrl = PseudospecCtrl<Base<F>>())

Matrix<int> QuasiTriangularPseudospectrum(**const** *Matrix*<Real> &U, *Matrix*<Real> &invNormMap, *Complex*<Real> center, Real realWidth, Real imagWidth, int realSize, int imagSize, *PseudospecCtrl*<Real> psCtrl = PseudospecCtrl<Real>())

DistMatrix<int> QuasiTriangularPseudospectrum(**const** *DistMatrix*<Real> &U, *DistMatrix*<Real> &invNormMap, *Complex*<Real> center, Real realWidth, Real imagWidth, int realSize, int imagSize, *PseudospecCtrl*<Real> psCtrl = PseudospecCtrl<Real>())

Matrix<int> HessenbergPseudospectrum(**const** *Matrix*<F> &H, *Matrix*<Base<F>> &invNormMap, *Complex*<Base<F>> center, *Base*<F> realWidth, *Base*<F> imagWidth, int realSize, int imagSize, *PseudospecCtrl*<Base<F>> psCtrl = PseudospecCtrl<Base<F>>())

DistMatrix<int> HessenbergPseudospectrum(**const** *DistMatrix*<F> &H, *DistMatrix*<Base<F>> &invNormMap, *Complex*<Base<F>> center, *Base*<F> realWidth, *Base*<F> imagWidth, int realSize, int imagSize, *PseudospecCtrl*<Base<F>> psCtrl = PseudospecCtrl<Base<F>>())

Same as above, but the real and imaginary widths of the 2D grid in the complex plane are manually specified.

5.6.5 Trace

Implementation

The two equally useful definitions of the trace of a square matrix $A \in \mathbb{F}^{n \times n}$ are

$$\text{tr}(A) = \sum_{i=0}^{n-1} \alpha_{i,i} = \sum_{i=0}^{n-1} \lambda_i,$$

where $\alpha_{i,i}$ is the i 'th diagonal entry of A and λ_i is the i 'th eigenvalue (counting multiplicity) of A .

Clearly the former equation is easier to compute, but the latter is an important characterization.

F Trace(**const** *Matrix*<F> &A)

F Trace(**const** *DistMatrix*<F> &A)

Return the trace of the square matrix A .

5.6.6 Inertia

Implementation

type InertiaType

int numPositive

int numNegative

int numZero

The basic eigenvalue structure of a Hermitian matrix (which can be quickly computed from the quasi-diagonal matrix produced by Bunch-Kaufman).

InertiaType Inertia(*UpperOrLower* uplo, *Matrix*<F> &A, *LDLPivotType* pivotType = BUNCH.PARLETT)

InertiaType Inertia(*UpperOrLower* uplo, *DistMatrix*<F> &A, *LDLPivotType* pivotType = BUNCH.PARLETT)

Returns the triplet containing the number of positive, negative, and zero eigenvalues of the Hermitian matrix by analyzing the block diagonal resulting from a pivoted LDL factorization.

5.7 Linear solvers

5.7.1 HPD solve

Implementation

Solves $AX = B$, $A^T X = B$, or $A^H X = B$ for X given Hermitian positive-definite (HPD) A and right-hand side matrix B (note that these options are all identical except for when A is complex). The solution is computed by first finding the Cholesky factorization of A and then performing two successive triangular solves against B .

void HPDSolve(*UpperOrLower* uplo, *Orientation* orientation, *Matrix*<F> &A, *Matrix*<F> &B)

```
void HPDSolve(UpperOrLower uplo, Orientation orientation, DistMatrix<F> &A, DistMa-  
trix<F> &B)
```

Overwrite B with the solution to $AX = B$ or $A^T X = B$, where A is Hermitian positive-definite and only the triangle of A specified by $uplo$ is accessed.

5.7.2 Symmetric solve

Implementation

Solve $AX = B$, $A^T X = B$, or $A^H X = B$ for X given a symmetric or Hermitian matrix A and a right-hand side matrix B using Bunch-Kaufman.

```
void SymmetricSolve(UpperOrLower uplo, Orientation orientation, Matrix<F> &A, Ma-  
trix<F> &B, bool conjugate = false, LDLPivotType pivotType =  
BUNCH.KAUFMAN_A)
```

```
void SymmetricSolve(UpperOrLower uplo, Orientation orientation, DistMatrix<F> &A,  
DistMatrix<F> &B, bool conjugate = false, LDLPivotType pivotType  
= BUNCH.KAUFMAN_A)
```

Overwrites B with the solution to the linear system. A is assumed symmetric if $conjugate$ is false, and Hermitian otherwise.

Note: Only the lower-storage case is currently supported.

5.7.3 Hermitian solve

Implementation

Solve $AX = B$, $A^T X = B$, or $A^H X = B$ for X given a Hermitian matrix A and a right-hand side matrix B using Bunch-Kaufman.

```
void HermitianSolve(UpperOrLower uplo, Orientation orientation, Matrix<F> &A, Ma-  
trix<F> &B, LDLPivotType pivotType = BUNCH.KAUFMAN_A)
```

```
void HermitianSolve(UpperOrLower uplo, Orientation orientation, DistMa-  
trix<F> &A, DistMatrix<F> &B, LDLPivotType pivotType =  
BUNCH.KAUFMAN_A)
```

Overwrites B with the solution to the linear system.

Note: Only the lower-storage case is currently supported, as this is a simple wrapper around `SymmetricSolve()`.

5.7.4 Gaussian elimination

Implementation

Solves $AX = B$ for X given a general square nonsingular matrix A and right-hand side matrix B . The solution is computed through (partially pivoted) Gaussian elimination.

```
void GaussianElimination(Matrix<F> &A, Matrix<F> &B)
```

```
void GaussianElimination(DistMatrix<F> &A, DistMatrix<F> &B)
```

Upon completion, A will have been overwritten with Gaussian elimination and B will be overwritten with X .

5.7.5 Least Squares

Implementation

Given $A \in \mathbb{F}^{m \times n}$ and a right-hand side $b \in \mathbb{F}^m$, a *least-squares* method solves $Ax \approx b$ differently depending upon whether $m \geq n$.

When $m \geq n$, there are at least as many constraints as degrees of freedom, and so a solution is sought for

$$\min_x \|Ax - b\|_2$$

This problem is solved through the use of `QR()`.

When $m < n$, the problem is under-constrained and a solution is sought for the problem

$$\min_x \|x\|_2 \quad \text{such that } Ax = b.$$

This problem is solved through the use of `LQ()`.

The above optimization problems can be readily generalized to multiple right-hand sides by switching to Frobenius norms.

```
void LeastSquares(Orientation orientation, Matrix<F> &A, const Matrix<F> &B, Matrix<F> &X)
```

```
void LeastSquares(Orientation orientation, DistMatrix<F> &A, const DistMatrix<F> &B, DistMatrix<F> &X)
```

If *orientation* is set to `NORMAL`, then solve $AX = B$, otherwise *orientation* must be equal to `ADJOINT` and $A^H X = B$ will be solved. Upon completion, *A* is overwritten with its QR or LQ factorization, and each column of *X* is overwritten with a solution vector.

5.7.6 General (Gauss-Markov) Linear Model (GLM)

Implementation

Example driver

$$\min_{X,Y} \|Y\|_F \quad \text{subject to } AX + BY = D.$$

```
void GLM(Matrix<F> &A, Matrix<F> &B, Matrix<F> &D, Matrix<F> &Y)
```

```
void GLM(DistMatrix<F> &A, DistMatrix<F> &B, DistMatrix<F> &D, DistMatrix<F> &Y)
```

5.7.7 Equality-constrained Least Squares (LSE)

Implementation

Example driver

$$\min_X \|AX - C\|_F \quad \text{subject to } BX = D.$$

```
void LSE(Matrix<F> &A, Matrix<F> &B, Matrix<F> &C, Matrix<F> &D, Matrix<F>
    &X, bool computeResidual = false)
```

```
void LSE(DistMatrix<F> &A, DistMatrix<F> &B, DistMatrix<F> &C, DistMatrix<F>
    &D, DistMatrix<F> &X, bool computeResidual = false)
```

5.7.8 Multi-shift Hessenberg solves

Implementation

Solve for X in the system

$$H^\# X - XD^\# = Y$$

where H is Hessenberg, D is diagonal, and $A^\#$ is defined to be one of $\{A, A^T, A^H\}$.

```
void MultiShiftHessSolve(UpperOrLower uplo, Orientation orientation, F alpha, const
    Matrix<F> &H, const Matrix<F> &shifts, Matrix<F> &X)
```

```
void MultiShiftHessSolve(UpperOrLower uplo, Orientation orientation, F alpha, const
    DistMatrix<F, UH, VH> &H, const DistMatrix<F, VX,
    STAR> &shifts, DistMatrix<F, STAR, VX> &X)
```

Overwrite the columns of X with the solutions to shifted linear systems.

Note: Only a few subcases are currently supported, as this was added as part of `HessenbergPseudospectrum()`

5.8 Utilities

5.8.1 Householder reflectors

TODO: Describe major difference from LAPACK's conventions (i.e., we do not treat the identity matrix as a Householder transform since it requires the u in $H = I - 2uu'$ to have norm zero rather than one).

```
F LeftReflector(F &chi, Matrix<F> &x)
```

```
F LeftReflector(Matrix<F> &chi, Matrix<F> &x)
```

```
F LeftReflector(F &chi, DistMatrix<F> &x)
```

```
F LeftReflector(DistMatrix<F> &chi, DistMatrix<F> &x)
```

```
F reflector::Col(F &chi, DistMatrix<F> &x)
```

```
F reflector::Col(DistMatrix<F> &chi, DistMatrix<F> &x)
```

```
F RightReflector(F &chi, Matrix<F> &x)
```

```
F RightReflector(Matrix<F> &chi, Matrix<F> &x)
```

```
F RightReflector(F &chi, DistMatrix<F> &x)
```

```
F RightReflector(DistMatrix<F> &chi, DistMatrix<F> &x)
```

```
F reflector::Row(F &chi, DistMatrix<F> &x)
```

```

F reflector::Row(DistMatrix<F> &chi, DistMatrix<F> &x)
F LeftHyperbolicReflector(F &chi, Matrix<F> &x)
F LeftHyperbolicReflector(Matrix<F> &chi, Matrix<F> &x)
F LeftHyperbolicReflector(F &chi, DistMatrix<F> &x)
F LeftHyperbolicReflector(DistMatrix<F> &chi, DistMatrix<F> &x)
F hyp_reflector::Col(F &chi, DistMatrix<F> &x)
F hyp_reflector::Col(DistMatrix<F> &chi, DistMatrix<F> &x)
F RightHyperbolicReflector(F &chi, Matrix<F> &x)
F RightHyperbolicReflector(Matrix<F> &chi, Matrix<F> &x)
F RightHyperbolicReflector(F &chi, DistMatrix<F> &x)
F RightHyperbolicReflector(DistMatrix<F> &chi, DistMatrix<F> &x)
F hyp_reflector::Row(F &chi, DistMatrix<F> &x)
F hyp_reflector::Row(DistMatrix<F> &chi, DistMatrix<F> &x)
void ApplyPackedReflectors(LeftOrRight side, UpperOrLower uplo, VerticalOrHorizontal dir, ForwardOrBackward order, Conjugation conjugation, int offset, const Matrix<F> &H, const Matrix<F> &t, Matrix<F> &A)
void ApplyPackedReflectors(LeftOrRight side, UpperOrLower uplo, VerticalOrHorizontal dir, ForwardOrBackward order, Conjugation conjugation, int offset, const DistMatrix<F> &H, const DistMatrix<F, MD, STAR> &t, DistMatrix<F> &A)
void ApplyPackedReflectors(LeftOrRight side, UpperOrLower uplo, VerticalOrHorizontal dir, ForwardOrBackward order, Conjugation conjugation, int offset, const DistMatrix<F> &H, const DistMatrix<F, STAR, STAR> &t, DistMatrix<F> &A)
void ExpandPackedReflectors(UpperOrLower uplo, VerticalOrHorizontal dir, Conjugation conjugation, int offset, Matrix<F> &H, const Matrix<F> &t)
void ExpandPackedReflectors(UpperOrLower uplo, VerticalOrHorizontal dir, Conjugation conjugation, int offset, DistMatrix<F> &H, const DistMatrix<F, MD, STAR> &t)
void ExpandPackedReflectors(UpperOrLower uplo, VerticalOrHorizontal dir, Conjugation conjugation, int offset, DistMatrix<F> &H, const DistMatrix<F, STAR, STAR> &t)

```

5.8.2 Sorting

```

void Sort(Matrix<Real> &X, SortType sort = ASCENDING)
void Sort(DistMatrix<Real, U, V> &X, SortType sort = ASCENDING)
std::vector<ValueInt<Real>> TaggedSort(const Matrix<Real> &X, SortType sort = ASCENDING)

```

```
std::vector<ValueInt<Real>> TaggedSort(const DistMatrix<Real, U, V> &X, SortType  
                                     sort = ASCENDING)
```

```
ValueInt<Real> Median(const Matrix<Real> &x)
```

```
ValueInt<Real> Median(const DistMatrix<Real, U, V> &x)
```

5.9 Tuning parameters

5.9.1 Hermitian to tridiagonal

Two different basic strategies are available for the reduction to tridiagonal form:

1. Run a pipelined algorithm designed for general (rectangular) process grids.
2. Redistribute the matrix so that it is owned by a perfect square number of processes, perform a fast reduction to tridiagonal form, and redistribute the data back to the original process grid. This algorithm is essentially an evolution of the HJS tridiagonalization approach (see “Towards an efficient parallel eigensolver for dense symmetric matrices” by Bruce Hendrickson, Elizabeth Jessup, and Christopher Smith) which is described in detail in Ken Stanley’s dissertation, “Execution time of symmetric eigensolvers”.

There is clearly a small penalty associated with the extra redistributions necessary for the second approach, but the benefit from using a square process grid is usually quite significant. By default, *HermitianTridiag()* will run the standard algorithm (approach 1) unless the matrix is already distributed over a square process grid. The reasoning is that good performance depends upon a “good” ordering of the square (say, $\hat{p} \times \hat{p}$) subgrid, though usually either a row-major or column-major ordering of the first \hat{p}^2 processes suffices.

type HermitianTridiagApproach

- HERMITIAN_TRIDIAG_NORMAL: Run the pipelined rectangular algorithm.
- HERMITIAN_TRIDIAG_SQUARE: Run the square grid algorithm on the largest possible square process grid.
- HERMITIAN_TRIDIAG_DEFAULT: If the given process grid is already square, run the square grid algorithm, otherwise use the pipelined non-square approach.

Note: A properly tuned HERMITIAN_TRIDIAG_SQUARE approach is almost always fastest, so it is worthwhile to test it with both the COLUMN_MAJOR and ROW_MAJOR subgrid orderings, as described below.

Note: The first algorithm heavily depends upon the performance of distributed *Symv()*, so users interested in maximizing the performance of the first algorithm will likely want to investigate different values for the local blocksize through the routine `SetLocalSymvBlocksize<T>(int blocksize);` the default value is 64.

 CONVEX OPTIMIZATION

6.1 ADMM solvers

The following functions are inspired by the MATLAB scripts associated with the paper:

S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, Distributed optimization and statistical learning via the Alternating Direction Method of Multipliers,

which describes a wide variety of optimization schemes using the Alternating Direction Method of Multipliers (ADMM).

6.1.1 BasisPursuit

Attempt to minimize $\|x\|_1$ such that $Ax = b$ using ADMM.

```
int BasisPursuit(const Matrix<F> &A, const Matrix<F> &b, Matrix<F> &x, Matrix<F> &z, Matrix<F> &u, Base<F> rho = 1, Base<F> alpha = 1.2, Int maxIter = 500, Base<F> absTol = 1e-6, Base<F> relTol = 1e-4, bool usePinv = true, Base<F> pinvTol = 0, bool progress = false)
```

```
int BasisPursuit(const DistMatrix<F> &A, const DistMatrix<F> &b, DistMatrix<F> &x, DistMatrix<F> &z, DistMatrix<F> &u, Base<F> rho = 1, Base<F> alpha = 1.2, Int maxIter = 500, Base<F> absTol = 1e-6, Base<F> relTol = 1e-4, bool usePinv = true, Base<F> pinvTol = 0, bool progress = false)
```

[Implementations on GitHub](#)

[Example driver on GitHub](#)

These functions are inspired by [this basis pursuit ADMM implementation](#). Elemental's implementations use parallel (dense) linear algebra and allows the user to choose between either directly computing a (thresholded-)pseudoinverse or an LQ factorization. The return value is the number of performed ADMM iterations.

Parameters	Input/Output	Explanation
A	Input	part of constraint $Ax = b$
b	Input	part of constraint $Ax = b$
x	Output	primal solution (first term)
z	Output	primal solution (second term)
u	Output	dual solution
ρ	Input	augmented-Lagrangian parameter
α	Input	over-relaxation parameter (usually in $[1, 1.8]$)
$maxIter$	Input	maximum number of ADMM iterations
$absTol$	Input	absolute convergence tolerance
$relTol$	Input	relative convergence tolerance
$usePinv$	Input	directly compute a pseudoinverse?
$pinvTol$	Input	the pseudoinverse inversion tolerance
$progress$	Input	display detailed progress information?

6.1.2 LinearProgram

Attempt to solve the linear program

$$\min c^T x \quad \text{such that } Ax = b, x \geq 0.$$

using ADMM.

```
int LinearProgram(const Matrix<Real> &A, const Matrix<Real> &b, const Matrix<Real> &c, Matrix<Real> &x, Matrix<Real> &z, Matrix<Real> &u, Real rho = 1., Real alpha = 1.2, Int maxIter = 500, Real absTol = 1e-6, Real relTol = 1e-4, bool inv = false, bool progress = true)
```

```
int LinearProgram(const DistMatrix<Real> &A, const DistMatrix<Real> &b, const DistMatrix<Real> &c, DistMatrix<Real> &x, DistMatrix<Real> &z, DistMatrix<F> &u, Real rho = 1., Real alpha = 1.2, Int maxIter = 500, Real absTol = 1e-6, Real relTol = 1e-4, bool inv = true, bool progress = true)
```

[Implementations on GitHub](#)

[Example driver on GitHub](#)

These functions are inspired by [this ADMM linear program solver](#). Elemental's implementations make use of parallel (dense) linear algebra, a custom structured factorization, and allows the user to optionally directly invert the (pivoted) Schur complement to accelerate its application. The return value is the number of performed ADMM iterations.

Parameters	Input/Output	Explanation
A	Input	part of constraints, $Ax = b$
b	Input	part of constraints, $Ax = b$
c	Input	part of the objective, $c^T x$
x	Output	primal solution (first term)
z	Output	primal solution (second term)
u	Output	dual solution
ρ	Input	augmented-Lagrangian parameter
α	Input	over-relaxation parameter (usually in $[1, 1.8]$)
maxIter	Input	maximum number of ADMM iterations
absTol	Input	absolute convergence tolerance
relTol	Input	relative convergence tolerance
inv	Input	directly compute Schur-complement inverse?
progress	Input	display detailed progress information?

6.1.3 QuadraticProgram

Attempt to solve the quadratic program

$$\min \frac{1}{2} x^T P x + q^T x \quad \text{such that } l_b \leq x \leq u_b$$

using ADMM.

```
int QuadraticProgram(const Matrix<Real> &P, const Matrix<Real> &q, Real lb, Real
    ub, Matrix<Real> &x, Matrix<Real> &z, Matrix<Real> &u, Real
    rho = 1., Real alpha = 1.2, Int maxIter = 500, Real absTol = 1e-6, Real
    relTol = 1e-4, bool inv = false, bool progress = true)
```

```
int QuadraticProgram(const DistMatrix<Real> &P, const DistMatrix<Real> &q, Real
    lb, Real ub, DistMatrix<Real> &x, DistMatrix<Real> &z, DistMa-
    trix<Real> &u, Real rho = 1., Real alpha = 1.2, Int maxIter = 500,
    Real absTol = 1e-6, Real relTol = 1e-4, bool inv = true, bool progress
    = true)
```

[Implementations on GitHub](#)

[Example driver on GitHub](#)

These functions are inspired by [this ADMM quadratic program solver](#). Elemental's implementations make use of parallel (dense) linear algebra and allows the user to optionally directly invert the Cholesky factor to improve the parallel performance of the application of its inverse.

Parameters	Input/Output	Explanation
P	Input	SPD and part of objective, $\frac{1}{2}x^T Px + q^T x$
q	Input	part of objective
lb	Input	lower-bound of constraints, $l_b \leq x \leq u_b$
ub	Input	upper-bound of constraints, $l_b \leq x \leq u_b$
x	Output	primal solution (first term)
z	Output	primal solution (second term)
u	Output	dual solution
ρ	Input	augmented-Lagrangian parameter
α	Input	over-relaxation parameter (usually in $[1, 1.8]$)
$maxIter$	Input	maximum number of ADMM iterations
$absTol$	Input	absolute convergence tolerance
$relTol$	Input	relative convergence tolerance
inv	Input	compute inverse of Cholesky factor?
$progress$	Input	display detailed progress information?

6.1.4 SparseInvCov

Attempt to find a sparse inverse covariance matrix which generated the given observations by solving the program

$$\text{mintrace}(SX) - \log \det X + \lambda \|\text{vec}(X)\|_1$$

using ADMM.

```
int SparseInvCov(const Matrix<Real> &D, Matrix<Real> &X, Matrix<Real> &Z, Matrix<Real> &U, Real lambda, Real rho = 1., Real alpha = 1.2, Int maxIter = 500, Real absTol = 1e-6, Real relTol = 1e-4, bool progress = true)
```

```
int SparseInvCov(const DistMatrix<Real> &D, DistMatrix<Real> &X, DistMatrix<Real> &Z, DistMatrix<Real> &U, Real lambda, Real rho = 1., Real alpha = 1.2, Int maxIter = 500, Real absTol = 1e-6, Real relTol = 1e-4, bool progress = true)
```

[Implementations on GitHub](#)

[Example driver on GitHub](#)

These functions are inspired by [this ADMM solver](#). Elemental's implementations make use of parallel (dense) linear algebra (including PMRRR for the symmetric tridiagonal eigensolver).

Parameters	Input/Output	Explanation
D	Input	Observations
X	Output	primal solution (first term)
Z	Output	primal solution (second term)
U	Output	dual solution
λ	Input	coefficient for vector-l1 penalty
ρ	Input	augmented-Lagrangian parameter
α	Input	over-relaxation parameter (usually in $[1, 1.8]$)
$maxIter$	Input	maximum number of ADMM iterations
$absTol$	Input	absolute convergence tolerance
$relTol$	Input	relative convergence tolerance
$progress$	Input	display detailed progress information?

6.2 Utilities

The following utility routines are widely-used within first-order optimization methods, and some, such as Singular Value soft-Thresholding (SVT), warrant non-trivial implementations.

6.2.1 Clip

Force every entry of a matrix to lie within a given (half-)interval.

Implementations on GitHub

```
void LowerClip(Matrix<Real> &X, Real lowerBound = 0)
```

```
void LowerClip(DistMatrix<Real> &X, Real lowerBound = 0)
    Force every entry to be at least lowerBound.
```

```
void UpperClip(Matrix<Real> &X, Real upperBound = 0)
```

```
void UpperClip(DistMatrix<Real> &X, Real upperBound = 0)
    Force every entry to be at most upperBound.
```

```
void Clip(Matrix<Real> &X, Real lowerBound = 0, Real upperBound = 1)
```

```
void Clip(DistMatrix<Real> &X, Real lowerBound = 0, Real upperBound = 1)
    Force every entry to lie within the interval defined by lowerBound and upperBound.
```

6.2.2 LogBarrier

Uses a careful calculation of the log of the determinant in order to return the *log barrier* of a Hermitian positive-definite matrix A , $-\log(\det(A))$.

Implementations on GitHub

```
Base<F> LogBarrier(UpperOrLower uplo, const Matrix<F> &A)
```

```
Base<F> LogBarrier(UpperOrLower uplo, const DistMatrix<F> &A)
```

```
Base<F> LogBarrier(UpperOrLower uplo, Matrix<F> &A, bool canOverwrite = false)
```

```
Base<F> LogBarrier(UpperOrLower uplo, DistMatrix<F> &A, bool canOverwrite = false)
```

6.2.3 LogDetDiv

The *log-det divergence* of a pair of $n \times n$ Hermitian positive-definite matrices A and B is

$$D_{ld}(A, B) = \text{tr}(AB^{-1}) - \log(\det(AB^{-1})) - n,$$

which can be greatly simplified using the Cholesky factors of A and B . In particular, if we set $Z = L_B^{-1}L_A$, where $A = L_A L_A^H$ and $B = L_B L_B^H$ are Cholesky factorizations, then

$$D_{ld}(A, B) = \|Z\|_F^2 - 2 \log(\det(Z)) - n.$$

Implementations on GitHub

Example driver on GitHub

```
Base<F> LogDetDivergence(UpperOrLower uplo, const Matrix<F> &A, const Ma-
    trix<F> &B)
```

Base<F> LogDetDivergence(*UpperOrLower* *uplo*, **const** *DistMatrix*<F> &A, **const** *DistMatrix*<F> &B)

6.2.4 Singular-value soft-thresholding

Overwrites A with $US_{\tau}(\Sigma)V^H$, where $U\Sigma V^H$ is the singular-value decomposition of A upon input and S_{τ} performs soft-thresholding with parameter τ . The return value is the rank of the soft-thresholded matrix.

Implementations on GitHub

int SVT(*Matrix*<F> &A, *Base*<F> *tau*, bool *relative* = false)

int SVT(*DistMatrix*<F> &A, *Base*<F> *tau*, bool *relative* = false)

Runs the default SVT algorithm. In the sequential case, this is currently *svt::Normal*, and, in the parallel case, it is *svt::Cross*.

int SVT(*Matrix*<F> &A, *Base*<F> *tau*, int *relaxedRank*, bool *relative* = false)

int SVT(*DistMatrix*<F> &A, *Base*<F> *tau*, int *relaxedRank*, bool *relative* = false)

Runs a faster (for small ranks), but less accurate, algorithm given an upper bound on the rank of the soft-thresholded matrix. The current implementation preprocesses via *relaxedRank* steps of (Businger-Golub) column-pivoted QR via the routine *svt::PivotedQR*.

int SVT(*DistMatrix*<F, U, STAR> &A, *Base*<F> *tau*, bool *relative* = false)

Runs an SVT algorithm designed for tall-skinny matrices. The current implementation is based on TSQR factorization and is *svt::TSQR*.

namespace svt

int svt::Normal(*Matrix*<F> &A, *Base*<F> *tau*, bool *relative* = false)

int svt::Normal(*DistMatrix*<F> &A, *Base*<F> *tau*, bool *relative* = false)

Runs a standard SVD, soft-thresholds the singular values, and then reforms the matrix.

int svt::Cross(*Matrix*<F> &A, *Base*<F> *tau*, bool *relative* = false)

int svt::Cross(*DistMatrix*<F> &A, *Base*<F> *tau*, bool *relative* = false)

Forms the normal matrix, computes its Hermitian EVD, soft-thresholds the eigenvalues, and then reforms the matrix. Note that Elemental's parallel Hermitian EVD is much faster than its parallel SVD; this is typically worth the loss of accuracy in the computed small (truncated) singular values and is therefore the default choice for parallel SVT.

int svt::PivotedQR(*Matrix*<F> &A, *Base*<F> *tau*, int *numStepsQR*, bool *relative* = false)

int svt::PivotedQR(*DistMatrix*<F> &A, *Base*<F> *tau*, int *numStepsQR*, bool *relative* = false)

Computes an approximate SVT by first approximating A as the rank-*numSteps* approximation produced by *numSteps* iterations of column-pivoted QR.

int svt::TSQR(*DistMatrix*<F, U, STAR> &A, *Base*<F> *tau*, bool *relative* = false)

Since the majority of the work in a tall-skinny SVT will be in the initial QR factorization, this algorithm runs a TSQR factorization and then computes the SVT of the small R factor using a single process.

6.2.5 Soft-thresholding

Overwrites each entry of A with its soft-thresholded value.

[Implementations on GitHub](#)

```
void SoftThreshold(Matrix<F> &A, Base<F> tau, bool relative = false)
```

```
void SoftThreshold(DistMatrix<F> &A, Base<F> tau, bool relative = false)
```


CONTROL THEORY

The following algorithms draw heavily from the second chapter of Nicholas J. Higham's "Functions of Matrices: Theory and Computation" and depend heavily on the matrix sign function. They have only undergone cursory testing.

7.1 Sylvester

As long as both A and B only have eigenvalues in the open right-half plane, the following routines solve for X in the *Sylvester equation*

$$AX + XB = C$$

via computing $\text{sgn}(W)$, where

$$W = \begin{pmatrix} A & -C \\ 0 & -B \end{pmatrix}.$$

```
void Sylvester(const Matrix<F> &A, const Matrix<F> &B, const Matrix<F> &C, Ma-
               trix<F> &X, SignCtrl<Base<F>> signCtrl = SignCtrl<Base<F>>())
```

```
void Sylvester(const DistMatrix<F> &A, const DistMatrix<F> &B, const DistMa-
               trix<F> &C, DistMatrix<F> &X, SignCtrl<Base<F>> signCtrl = SignC-
               trl<Base<F>>())
```

One may also directly pass in W in order to save space.

```
void Sylvester(int m, Matrix<F> &W, Matrix<F> &X, SignCtrl<Base<F>> signCtrl =
               SignCtrl<Base<F>>())
```

```
void Sylvester(int m, DistMatrix<F> &W, DistMatrix<F> &X, SignCtrl<Base<F>>
               signCtrl = SignCtrl<Base<F>>())
```

7.2 Lyapunov

A special case of the Sylvester solver, where $B = A^H$.

```
void Lyapunov(const Matrix<F> &A, const Matrix<F> &C, Matrix<F> &X, SignC-
               trl<Base<F>> signCtrl = SignCtrl<Base<F>>())
```

```
void Lyapunov(const DistMatrix<F> &A, const DistMatrix<F> &C, DistMatrix<F> &X,
               SignCtrl<Base<F>> signCtrl = SignCtrl<Base<F>>())
```

7.3 Algebraic Ricatti

Under suitable conditions, the following routines solve for X in the *algebraic Ricatti equation*

$$XKX - A^H X - XA = L,$$

where both K and L are Hermitian.

```
void Ricatti(UpperOrLower uplo, const Matrix<F> &A, const Matrix<F> &K, const
            Matrix<F> &L, Matrix<F> &X, SignCtrl<Base<F>> signCtrl = SignC-
            trl<Base<F>>())
```

```
void Ricatti(UpperOrLower uplo, const DistMatrix<F> &A, const DistMatrix<F> &K,
            const DistMatrix<F> &L, DistMatrix<F> &X, SignCtrl<Base<F>> signC-
            trl = SignCtrl<Base<F>>())
```

Alternatively, one may directly fill the matrix W .

```
void Ricatti(Matrix<F> &W, Matrix<F> &X, SignCtrl<Base<F>> signCtrl = SignC-
            trl<Base<F>>())
```

```
void Ricatti(DistMatrix<F> &W, DistMatrix<F> &X, SignCtrl<Base<F>> signCtrl =
            SignCtrl<Base<F>>())
```


SPECIAL MATRICES

It is frequently useful to test algorithms on well-known, trivial, and random matrices, such as

1. matrices with entries sampled from a uniform distribution,
2. matrices with spectrum sampled from a uniform distribution,
3. Wilkinson matrices,
4. identity matrices,
5. matrices of all ones, and
6. matrices of all zeros.

Elemental therefore provides utilities for generating many such matrices.

8.1 Deterministic

8.1.1 Cauchy

An $m \times n$ matrix A is called *Cauchy* if there exist vectors x and y such that

$$\alpha_{i,j} = \frac{1}{\chi_i - \eta_j},$$

where χ_i is the i 'th entry of x and η_j is the j 'th entry of y .

```
void Cauchy(Matrix<F> &A, const std::vector<F> &x, const std::vector<F> &y)
```

```
void Cauchy(DistMatrix<F, U, V> &A, const std::vector<F> &x, const std::vector<F>  
&y)
```

Generate a Cauchy matrix using the defining vectors, x and y .

8.1.2 Cauchy-like

An $m \times n$ matrix A is called *Cauchy-like* if there exist vectors r , s , x , and y such that

$$\alpha_{i,j} = \frac{\rho_i \psi_j}{\chi_i - \eta_j},$$

where ρ_i is the i 'th entry of r , ψ_j is the j 'th entry of s , χ_i is the i 'th entry of x , and η_j is the j 'th entry of y .

```
void CauchyLike(Matrix<F> &A, const std::vector<F> &r, const std::vector<F> &s,  
               const std::vector<F> &x, const std::vector<F> &y)
```

```
void CauchyLike(DistMatrix<F, U, V> &A, const std::vector<F> &r, const  
               std::vector<F> &s, const std::vector<F> &x, const std::vector<F>  
               &y)
```

Generate a Cauchy-like matrix using the defining vectors: r , s , x , and y .

8.1.3 Circulant

An $n \times n$ matrix A is called *circulant* if there exists a vector b such that

$$\alpha_{i,j} = \beta_{(i-j) \bmod n},$$

where β_k is the k 'th entry of vector b .

```
void Circulant(Matrix<T> &A, const std::vector<T> &a)
```

```
void Circulant(DistMatrix<T, U, V> &A, const std::vector<T> &a)
```

Generate a circulant matrix using the vector a .

8.1.4 Diagonal

An $n \times n$ matrix A is called *diagonal* if each entry (i, j) , where $i \neq j$, is 0. They are therefore defined by the *diagonal* values, where $i = j$.

```
void Diagonal(Matrix<T> &D, const std::vector<T> &d)
```

```
void Diagonal(DistMatrix<T, U, V> &D, const std::vector<T> &d)
```

Construct a diagonal matrix from the vector of diagonal values, d .

8.1.5 Egorov

Sets A to an $n \times n$ matrix with the (i, j) entry equal to

$$\exp(i\phi(i, j)).$$

```
void Egorov(Matrix<Complex<Real>> &A, const RealFunctor &phase, int n)
```

```
void Egorov(DistMatrix<Complex<Real>, U, V> &A, const RealFunctor &phase, int n)
```

8.1.6 Extended Kahan

TODO

```
void ExtendedKahan(Matrix<F> &A, int k, Base<F> phi, Base<F> mu)
```

```
void ExtendedKahan(DistMatrix<F, U, V> &A, int k, Base<F> phi, Base<F> mu)
```

8.1.7 Fiedler

TODO

```
void Fiedler(Matrix<F> &A, const std::vector<F> &c)
```

```
void Fiedler(DistMatrix<F, U, V> &A, const std::vector<F> &c)
```

8.1.8 Forsythe

TODO

```
void Forsythe(Matrix<T> &J, int n, T alpha, T lambda)
```

```
void Forsythe(DistMatrix<T, U, V> &J, int n, T alpha, T lambda)
```

8.1.9 Fourier

The $n \times n$ Discrete Fourier Transform (DFT) matrix, say A , is given by

$$\alpha_{i,j} = \frac{e^{-2\pi i j/n}}{\sqrt{n}}.$$

```
void Fourier(Matrix<Complex<Real>> &A, int n)
```

```
void Fourier(DistMatrix<Complex<Real>, U, V> &A, int n)  
    Set the matrix A equal to the  $n \times n$  DFT matrix.
```

```
void MakeFourier(Matrix<Complex<Real>> &A)
```

```
void MakeFourier(DistMatrix<Complex<Real>, U, V> &A)  
    Turn the existing  $n \times n$  matrix A into a DFT matrix.
```

8.1.10 GCDMatrix

TODO

```
void GCDMatrix(Matrix<T> &G, int m, int n)
```

```
void GCDMatrix(DistMatrix<T, U, V> &G, int m, int n)
```

8.1.11 Gear

TODO

```
void Gear(Matrix<T> &G, int n, int s, int t)
```

```
void Gear(DistMatrix<T, U, V> &G, int n, int s, int t)
```

8.1.12 Golub/Klema/Stewart

TODO

void GKS(*Matrix*<F> &A, int *n*)

void GKS(*DistMatrix*<F, U, V> &A, int *n*)

8.1.13 Grcar

TODO

void Grcar(*Matrix*<T> &A, int *n*, int *k* = 3)

void Grcar(*DistMatrix*<T, U, V> &A, int *n*, int *k* = 3)

8.1.14 Hankel

An $m \times n$ matrix A is called a *Hankel matrix* if there exists a vector b such that

$$\alpha_{i,j} = \beta_{i+j},$$

where $\alpha_{i,j}$ is the (i, j) entry of A and β_k is the k 'th entry of the vector b .

void Hankel(*Matrix*<T> &A, int *m*, int *n*, **const** std::vector<T> &*b*)

void Hankel(*DistMatrix*<T, U, V> &A, int *m*, int *n*, **const** std::vector<T> &*b*)

Create an $m \times n$ Hankel matrix from the generate vector, b .

8.1.15 Hanowa

TODO

void Hanowa(*Matrix*<T> &A, int *n*, T *mu*)

void Hanowa(*DistMatrix*<T, U, V> &A, int *n*, T *mu*)

8.1.16 Helmholtz

TODO

void Helmholtz(*Matrix*<F> &H, int *n*, F *shift*)

void Helmholtz(*DistMatrix*<F, U, V> &H, int *n*, F *shift*)

1D Helmholtz: **TODO**

void Helmholtz(*Matrix*<F> &H, int *nx*, int *ny*, F *shift*)

void Helmholtz(*DistMatrix*<F, U, V> &H, int *nx*, int *ny*, F *shift*)

2D Helmholtz: **TODO**

void Helmholtz(*Matrix*<F> &H, int *nx*, int *ny*, int *nz*, F *shift*)

void Helmholtz(*DistMatrix*<F, U, V> &H, int *nx*, int *ny*, int *nz*, F *shift*)

3D Helmholtz: **TODO**

8.1.17 Hilbert

The Hilbert matrix of order n is the $n \times n$ matrix where entry (i, j) is equal to $1/(i + j + 1)$.

```
void Hilbert(Matrix<F> &A, int n)
```

```
void Hilbert(DistMatrix<F, U, V> &A, int n)
```

Generate the $n \times n$ Hilbert matrix A.

```
void MakeHilbert(Matrix<F> &A)
```

```
void MakeHilbert(DistMatrix<F, U, V> &A)
```

Turn the square matrix A into a Hilbert matrix.

8.1.18 HermitianFromEVD

Form

$$A := Z\Omega Z^H,$$

where Ω is a real diagonal matrix.

```
void HermitianFromEVD(UpperOrLower uplo, Matrix<F> &A, const Matrix<Base<F>>
    &w, const Matrix<F> &Z)
```

```
void HermitianFromEVD(UpperOrLower uplo, DistMatrix<F> &A, const DistMa-
    trix<Base<F>, VR, STAR> &w, const DistMatrix<F> &Z)
```

The diagonal entries of Ω are given by the vector w .

8.1.19 Identity

The $n \times n$ *identity matrix* is simply defined by setting entry (i, j) to one if $i = j$, and zero otherwise. For various reasons, we generalize this definition to nonsquare, $m \times n$, matrices.

```
void Identity(Matrix<T> &A, int m, int n)
```

```
void Identity(DistMatrix<T, U, V> &A, int m, int n)
```

Set the matrix A equal to the $m \times n$ identity(-like) matrix.

```
void MakeIdentity(Matrix<T> &A)
```

```
void MakeIdentity(DistMatrix<T, U, V> &A)
```

Set the matrix A to be identity-like.

8.1.20 Jordan

TODO

```
void Jordan(Matrix<T> &J, int n, T lambda)
```

```
void Jordan(DistMatrix<T, U, V> &J, int n, T lambda)
```

8.1.21 Kahan

For any pair (ϕ, ζ) such that $|\phi|^2 + |\zeta|^2 = 1$, the corresponding $n \times n$ Kahan matrix is given by:

$$K = \text{diag}(1, \phi, \dots, \phi^{n-1}) \begin{pmatrix} 1 & -\zeta & -\zeta & \cdots & -\zeta \\ 0 & 1 & -\zeta & \cdots & -\zeta \\ & \ddots & & \vdots & \vdots \\ \vdots & & & 1 & -\zeta \\ 0 & \cdots & & & 1 \end{pmatrix}$$

`void Kahan(Matrix<F> &A, int n, F phi)`

`void Kahan(DistMatrix<F> &A, int n, F phi)`

Sets the matrix A equal to the $n \times n$ Kahan matrix with the specified value for ϕ .

8.1.22 KMS

TODO

`void KMS(Matrix<T> &K, int n, T rho)`

`void KMS(DistMatrix<T, U, V> &K, int n, T rho)`

8.1.23 Laplacian

TODO

`void Laplacian(Matrix<F> &L, int n)`

`void Laplacian(DistMatrix<F, U, V> &L, int n)`

1D Laplacian: **TODO**

`void Laplacian(Matrix<F> &L, int nx, int ny)`

`void Laplacian(DistMatrix<F, U, V> &L, int nx, int ny)`

2D Laplacian: **TODO**

`void Laplacian(Matrix<F> &L, int nx, int ny, int nz)`

`void Laplacian(DistMatrix<F, U, V> &L, int nx, int ny, int nz)`

3D Laplacian: **TODO**

8.1.24 Lauchli

TODO

`void Lauchli(Matrix<T> &A, int n, T mu)`

`void Lauchli(DistMatrix<T, U, V> &A, int n, T mu)`

8.1.25 Legendre

The $n \times n$ tridiagonal Jacobi matrix associated with the Legendre polynomials. Its main diagonal is zero, and the off-diagonal terms are given by

$$\beta_j = \frac{1}{2} (1 - (2(j+1))^{-2})^{-1/2},$$

where β_j connects the j 'th degree of freedom to the $j + 1$ 'th degree of freedom, counting from zero. The eigenvalues of this matrix lie in $[-1, 1]$ and are the locations for Gaussian quadrature of order n . The corresponding weights may be found by doubling the square of the first entry of the corresponding normalized eigenvector.

void Legendre(*Matrix*<F> &A, int n)

void Legendre(*DistMatrix*<F, U, V> &A, int n)

Sets the matrix A equal to the $n \times n$ Jacobi matrix.

8.1.26 Lehmer

TODO

void Lehmer(*Matrix*<F> &L, int n)

void Lehmer(*DistMatrix*<F, U, V> &L, int n)

8.1.27 Lotkin

TODO

void Lotkin(*Matrix*<F> &A, int n)

void Lotkin(*DistMatrix*<F, U, V> &A, int n)

8.1.28 MinIJ

TODO

void MinIJ(*Matrix*<T> &M, int n)

void MinIJ(*DistMatrix*<T, U, V> &M, int n)

8.1.29 NormalFromEVD

Form

$$A := Z\Omega Z^H,$$

where Ω is a complex diagonal matrix.

void NormalFromEVD(*Matrix*<Complex<Real>> &A, const *Matrix*<Complex<Real>> & w , const *Matrix*<Complex<Real>> &Z)

void NormalFromEVD(*DistMatrix*<Complex<Real>> &A, const *DistMatrix*<Complex<Real>, VR, STAR> & w , const *DistMatrix*<Complex<Real>> &Z)

The diagonal entries of Ω are given by the vector w .

8.1.30 Ones

Create an $m \times n$ matrix of all ones.

```
void Ones(Matrix<T> &A, int m, int n)
```

```
void Ones(DistMatrix<T, U, V> &A, int m, int n)  
    Set the matrix A to be an  $m \times n$  matrix of all ones.
```

Change all entries of the matrix A to one.

```
void MakeOnes(Matrix<T> &A)
```

```
void MakeOnes(DistMatrix<T, U, V> &A)  
    Change the entries of the matrix to ones.
```

8.1.31 OneTwoOne

A “1-2-1” matrix is tridiagonal with a diagonal of all twos and sub- and super-diagonals of all ones.

```
void OneTwoOne(Matrix<T> &A, int n)
```

```
void OneTwoOne(DistMatrix<T, U, V> &A, int n)  
    Set A to a  $n \times n$  “1-2-1” matrix.
```

```
void MakeOneTwoOne(Matrix<T> &A)
```

```
void MakeOneTwoOne(DistMatrix<T, U, V> &A)  
    Modify the entries of the square matrix A to be “1-2-1”.
```

8.1.32 Parter

TODO

```
void Parter(Matrix<F> &P, int n)
```

```
void Parter(DistMatrix<F, U, V> &P, int n)
```

8.1.33 Pei

TODO

```
void Pei(Matrix<T> &P, int n, T alpha)
```

```
void Pei(DistMatrix<T, U, V> &P, int n, T alpha)
```

8.1.34 Redheffer

TODO

```
void Redheffer(Matrix<T> &R, int n)
```

```
void Redheffer(DistMatrix<T, U, V> &R, int n)
```


8.1.35 Riemann

TODO

```
void Riemann(Matrix<T> &R, int n)
```

```
void Riemann(DistMatrix<T, U, V> &R, int n)
```

8.1.36 Ris

TODO

```
void Ris(Matrix<F> &R, int n)
```

```
void Ris(DistMatrix<F, U, V> &R, int n)
```

8.1.37 Toeplitz

An $m \times n$ matrix is *Toeplitz* if there exists a vector b such that, for each entry $\alpha_{i,j}$ of A ,

$$\alpha_{i,j} = \beta_{i-j+(n-1)},$$

where β_k is the k 'th entry of b .

```
void Toeplitz(Matrix<T> &A, int m, int n, const std::vector<T> &b)
```

```
void Toeplitz(DistMatrix<T, U, V> &A, int m, int n, const std::vector<T> &b)
```

Build the matrix A using the generating vector b .

8.1.38 TriW

TODO

```
void TriW(Matrix<T> &A, int m, int n, T alpha, int k)
```

```
void TriW(DistMatrix<T, U, V> &A, int m, int n, T alpha, int k)
```

8.1.39 Walsh

The Walsh matrix of order k is a $2^k \times 2^k$ matrix, where

$$W_1 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix},$$

and

$$W_k = \begin{pmatrix} W_{k-1} & W_{k-1} \\ W_{k-1} & -W_{k-1} \end{pmatrix}.$$

A *binary* Walsh matrix changes the bottom-right entry of W_1 from -1 to 0 .

```
void Walsh(Matrix<T> &W, int k, bool binary = false)
```

```
void Walsh(DistMatrix<T, U, V> &W, int k, bool binary = false)
```

Set the matrix W equal to the k 'th (possibly binary) Walsh matrix.

8.1.40 Wilkinson

A *Wilkinson matrix* of order k is a tridiagonal matrix with diagonal

$$[k, k - 1, k - 2, \dots, 1, 0, 1, \dots, k - 2, k - 1, k],$$

and sub- and super-diagonals of all ones.

```
void Wilkinson(Matrix<T> &W, int k)
```

```
void Wilkinson(DistMatrix<T, U, V> &W, int k)
```

Set the matrix W equal to the k 'th Wilkinson matrix.

8.1.41 Zeros

Create an $m \times n$ matrix of all zeros.

```
void Zeros(Matrix<T> &A, int m, int n)
```

```
void Zeros(DistMatrix<T, U, V> &A, int m, int n)
```

Set the matrix A to be an $m \times n$ matrix of all zeros.

Change all entries of the matrix A to zero.

```
void MakeZeros(Matrix<T> &A)
```

```
void MakeZeros(DistMatrix<T, U, V> &A)
```

Change the entries of the matrix to zero.

8.2 Random

8.2.1 Gaussian

An $m \times n$ matrix is Gaussian if each entry is independently drawn from a normal distribution.

```
void Gaussian(Matrix<T> &A, int m, int n, T mean = 0, Base<T> stddev = 1)
```

```
void Gaussian(DistMatrix<T, U, V> &A, int m, int n, T mean = 0, Base<T> stddev = 1)
```

Sets the matrix A to an $m \times n$ Gaussian matrix with the specified mean and standard deviation.

```
void MakeGaussian(Matrix<T> &A, T mean = 0, Base<T> stddev = 1)
```

```
void MakeGaussian(DistMatrix<T, U, V> &A, T mean = 0, Base<T> stddev = 1)
```

Changes each entry to an independent sample from the specified normal distribution.

8.2.2 Wigner

A Hermitian matrix whose entries in one triangle are all independent samples from a normal distribution. The spectra of these matrices are well-studied.

```
void Wigner(Matrix<T> &A, int n, T mean = 0, Base<T> stddev = 1)
```

`void Wigner(DistMatrix<T, U, V> &A, int n, T mean = 0, Base<T> stddev = 1)`
 Sets the matrix A to an $n \times n$ Wigner matrix with the specified mean and standard deviation.

8.2.3 Haar

The Haar distribution is the uniform distribution over the space of real or complex unitary matrices.

`void Haar(Matrix<F> &A, int n)`

`void Haar(DistMatrix<F> &A, int n)`

Draws A from the Haar distribution. The current scheme performs a QR factorization of a Gaussian matrix, but Stewart introduced a well-known scheme which only requires quadratic work for the implicit representation as a product of random Householder reflectors.

`void ImplicitHaar(Matrix<F> &A, Matrix<F> &t, int n)`

`void ImplicitHaar(DistMatrix<F> &A, DistMatrix<F, MD, STAR> &t, int n)`

Sets A to a set of Householder reflectors with the same structure as the result of a QR decomposition. The product of these reflectors is a sample from the Haar distribution.

8.2.4 Uniform

We call an $m \times n$ matrix is uniformly random if each entry is drawn from a uniform distribution over some ball $B_r(x)$, which is centered around some point x and of radius r .

`void Uniform(Matrix<T> &A, int m, int n, T center = 0, Base<T> radius = 1)`

`void Uniform(DistMatrix<T, U, V> &A, int m, int n, T center = 0, Base<T> radius = 1)`

Set the matrix A to an $m \times n$ matrix with each entry sampled from the uniform distribution centered at $center$ with radius $radius$.

`void MakeUniform(Matrix<T> &A, T center = 0, Base<T> radius = 1)`

`void MakeUniform(DistMatrix<T, U, V> &A, T center = 0, Base<T> radius = 1)`

Sample each entry of A from $U(B_r(x))$, where r is given by $radius$ and x is given by $center$.

8.2.5 HermitianUniformSpectrum

These routines sample a diagonal matrix from the specified interval of the real line and then perform a similarity transformation using a random Householder transform.

`void HermitianUniformSpectrum(Matrix<F> &A, int n, Base<F> lower = 0, Base<F> upper = 1)`

`void HermitianUniformSpectrum(DistMatrix<F, U, V> &A, int n, Base<F> lower = 0, Base<F> upper = 1)`

Build the $n \times n$ matrix A with a spectrum sampled uniformly from the interval $[lower, upper]$.

`void MakeHermitianUniformSpectrum(Matrix<F> &A, Base<F> lower = 0, Base<F> upper = 1)`

```
void MakeHermitianUniformSpectrum(DistMatrix<F, U, V> &A, Base<F> lower = 0,  
                                Base<F> upper = 1)
```

Sample the entries of the square matrix A from the interval (*lower*, *upper*].

8.2.6 NormalUniformSpectrum

These routines sample a diagonal matrix from the specified ball in the complex plane and then perform a similarity transformation using a random Householder transform.

```
void NormalUniformSpectrum(Matrix<Complex<Real>> &A, int n, Complex<Real> cen-  
                           ter = 0, Real radius = 1)
```

```
void NormalUniformSpectrum(DistMatrix<Complex<Real>, U, V> &A, int n, Com-  
                           plex<Real> center = 0, Real radius = 1)
```

Build the $n \times n$ matrix A with a spectrum sampled uniformly from the ball $B_{\text{radius}}(\text{center})$.

```
void MakeNormalUniformSpectrum(Matrix<Complex<Real>> &A, Complex<Real> cen-  
                               ter = 0, Real radius = 1)
```

```
void MakeNormalUniformSpectrum(DistMatrix<Complex<Real>, U, V> &A, Com-  
                               plex<Real> center = 0, Real radius = 1)
```

Sample the entries of the square matrix A from the ball in the complex plane centered at center with radius radius.

INPUT/OUTPUT

enum FileFormat

enumerator AUTO

Attempt to detect format from filename extension

enumerator ASCII

Simple ASCII text file

enumerator ASCII_MATLAB

MATLAB-ready ASCII text file

enumerator BINARY

Column-major binary file with integer height and width header

enumerator BINARY_FLAT

Column-major binary file with no header data

enumerator BMP

Bitmap image format (requires Qt5)

enumerator MATRIX_MARKET

Matrix Market format

enumerator JPG

JPG image format (requires Qt5)

enumerator JPEG

JPEG image format (requires Qt5)

enumerator PNG

PNG image format (requires Qt5)

enumerator PPM

PPM image format (requires Qt5)

enumerator XBM

XBM image format (requires Qt5)

enumerator XPM

XPM image format (requires Qt5)

9.1 Display

```
void Display(const Matrix<T> &A, std::string title = "Default")
```

void Display(const *DistMatrix*<T, U, V> &A, std::string title = "Default")

If Qt5 was detected during configuration, display the matrix on screen. Otherwise, print it to the console.

9.2 Print

void Print(const *Matrix*<T> &A, std::string title = "", std::ostream &os = std::cout)

void Print(const *DistMatrix*<T, U, V> &A, std::string title = "", std::ostream &os = std::cout)
Prints the matrix to the console.

9.3 Spy

void Spy(const *Matrix*<T> &A, std::string title = "Default", *Base*<T> tol = 0)

void Spy(const *DistMatrix*<T, U, V> &A, std::string title = "Default", *Base*<T> tol = 0)

Only available if Qt5 was detected during configuration. A spy plot of the elements with absolute values greater than or equal to *tol* is displayed on screen.

9.4 Read

void Read(*Matrix*<T> &A, std::string filename, *FileFormat* format = AUTO)

void Read(*DistMatrix*<T, U, V> &A, std::string filename, *FileFormat* format = AUTO, bool sequential = false)

Read the specified matrix from a file with a supported format. As of now, only the ASCII, BINARY, BINARY_FLAT and MATRIX_MARKET formats are supported. In the distributed case, the `sequential` flag determines whether or not the data should be read from a single process and then scattered to the other processes as necessary.

Note: In the case that a BINARY_FLAT file is read, *A* should already have the correct dimensions.

9.5 Write

void Write(const *Matrix*<T> &A, std::string basename = "matrix", *FileFormat* format = BINARY, std::string title = "")

void Write(const *DistMatrix*<T, U, V> &A, std::string basename = "matrix", *FileFormat* format = BINARY, std::string title = "")

The matrix is written to a file (the given basename plus the appropriate extension) in the specified format. Note that Qt5 is required for the image formats.

**CHAPTER
TEN**

INDICES

- [genindex](#)
- [search](#)

A

- Abs (C++ function), 33
- AbstractDistMatrix<T> (C++ class), 46
- AbstractDistMatrix<T>::~AbstractDistMatrix (C++ function), 46
- AbstractDistMatrix<T>::~AbstractDistMatrix (C++ function), 46
- AbstractDistMatrix<T>::Align (C++ function), 47
- AbstractDistMatrix<T>::AlignAndResize (C++ function), 47
- AbstractDistMatrix<T>::AlignCols (C++ function), 47
- AbstractDistMatrix<T>::AlignColsAndResize (C++ function), 47
- AbstractDistMatrix<T>::AlignColsWith (C++ function), 47
- AbstractDistMatrix<T>::AlignRows (C++ function), 47
- AbstractDistMatrix<T>::AlignRowsAndResize (C++ function), 47
- AbstractDistMatrix<T>::AlignRowsWith (C++ function), 47
- AbstractDistMatrix<T>::AlignWith (C++ function), 47
- AbstractDistMatrix<T>::AllocatedMemory (C++ function), 48
- AbstractDistMatrix<T>::AssertNotLocked (C++ function), 55
- AbstractDistMatrix<T>::AssertNotStoringData (C++ function), 55
- AbstractDistMatrix<T>::AssertSameGrid (C++ function), 55
- AbstractDistMatrix<T>::AssertSameSize (C++ function), 55
- AbstractDistMatrix<T>::AssertValidEntry (C++ function), 55
- AbstractDistMatrix<T>::AssertValidSubmatrix (C++ function), 55
- AbstractDistMatrix<T>::Attach (C++ function), 47, 48
- AbstractDistMatrix<T>::Buffer (C++ function), 48
- AbstractDistMatrix<T>::ColAlign (C++ function), 49
- AbstractDistMatrix<T>::ColComm (C++ function), 49
- AbstractDistMatrix<T>::ColConstrained (C++ function), 49
- AbstractDistMatrix<T>::ColOwner (C++ function), 50
- AbstractDistMatrix<T>::ColRank (C++ function), 49
- AbstractDistMatrix<T>::ColShift (C++ function), 49
- AbstractDistMatrix<T>::ColStride (C++ function), 50
- AbstractDistMatrix<T>::ComplainIfReal (C++ function), 55
- AbstractDistMatrix<T>::Conjugate (C++ function), 51
- AbstractDistMatrix<T>::ConjugateDiagonal (C++ function), 52
- AbstractDistMatrix<T>::ConjugateLocal (C++ function), 52
- AbstractDistMatrix<T>::ConjugateLocalSubmatrix (C++ function), 54
- AbstractDistMatrix<T>::ConjugateSubmatrix (C++ function), 53
- AbstractDistMatrix<T>::CrossComm (C++ function), 49
- AbstractDistMatrix<T>::CrossRank (C++ function), 50
- AbstractDistMatrix<T>::CrossSize (C++ function), 50
- AbstractDistMatrix<T>::DiagonalLength (C++ function), 48
- AbstractDistMatrix<T>::DistComm (C++ function), 49
- AbstractDistMatrix<T>::DistData (C++ function), 51

AbstractDistMatrix<T>::DistRank (C++ function), 50	AbstractDistMatrix<T>::LocalCol (C++ function), 51
AbstractDistMatrix<T>::DistSize (C++ function), 50	AbstractDistMatrix<T>::LocalColOffset (C++ function), 51
AbstractDistMatrix<T>::Empty (C++ function), 46	AbstractDistMatrix<T>::LocalHeight (C++ function), 48
AbstractDistMatrix<T>::EmptyData (C++ function), 47	AbstractDistMatrix<T>::LocalRow (C++ function), 51
AbstractDistMatrix<T>::FreeAlignments (C++ function), 47	AbstractDistMatrix<T>::LocalRowOffset (C++ function), 51
AbstractDistMatrix<T>::Get (C++ function), 51	AbstractDistMatrix<T>::LocalWidth (C++ function), 48
AbstractDistMatrix<T>::GetImagPart (C++ function), 51	AbstractDistMatrix<T>::Locked (C++ function), 48
AbstractDistMatrix<T>::GetImagPartOfLocalSubmatrix (C++ function), 53, 54	AbstractDistMatrix<T>::LockedAttach (C++ function), 47, 48
AbstractDistMatrix<T>::GetImagPartOfSubmatrix (C++ function), 52	AbstractDistMatrix<T>::LockedBuffer (C++ function), 48
AbstractDistMatrix<T>::GetLocal (C++ function), 51	AbstractDistMatrix<T>::LockedMatrix (C++ function), 48
AbstractDistMatrix<T>::GetLocalImagPart (C++ function), 52	AbstractDistMatrix<T>::MakeConsistent (C++ function), 47
AbstractDistMatrix<T>::GetLocalSubmatrix (C++ function), 53, 54	AbstractDistMatrix<T>::MakeDiagonalReal (C++ function), 52
AbstractDistMatrix<T>::GetRealPart (C++ function), 51	AbstractDistMatrix<T>::MakeLocalReal (C++ function), 52
AbstractDistMatrix<T>::GetRealPartLocal (C++ function), 51	AbstractDistMatrix<T>::MakeLocalSubmatrixReal (C++ function), 54
AbstractDistMatrix<T>::GetRealPartOfLocalSubmatrix (C++ function), 53, 54	AbstractDistMatrix<T>::MakeReal (C++ function), 51
AbstractDistMatrix<T>::GetRealPartOfSubmatrix (C++ function), 52	AbstractDistMatrix<T>::MakeSubmatrixReal (C++ function), 53
AbstractDistMatrix<T>::GetSubmatrix (C++ function), 52	AbstractDistMatrix<T>::Matrix (C++ function), 48
AbstractDistMatrix<T>::GlobalCol (C++ function), 50	AbstractDistMatrix<T>::operator= (C++ function), 46
AbstractDistMatrix<T>::GlobalRow (C++ function), 50	AbstractDistMatrix<T>::Owner (C++ function), 50
AbstractDistMatrix<T>::Grid (C++ function), 49	AbstractDistMatrix<T>::PartialColComm (C++ function), 49
AbstractDistMatrix<T>::Height (C++ function), 48	AbstractDistMatrix<T>::PartialColRank (C++ function), 50
AbstractDistMatrix<T>::IsLocal (C++ function), 51	AbstractDistMatrix<T>::PartialColStride (C++ function), 50
AbstractDistMatrix<T>::IsLocalCol (C++ function), 51	AbstractDistMatrix<T>::PartialRowComm (C++ function), 49
AbstractDistMatrix<T>::IsLocalRow (C++ function), 51	AbstractDistMatrix<T>::PartialRowRank (C++ function), 50
AbstractDistMatrix<T>::LDim (C++ function), 48	AbstractDistMatrix<T>::PartialRowStride (C++ function), 50

AbstractDistMatrix<T>::PartialUnionColComm (C++ function), 49
 AbstractDistMatrix<T>::PartialUnionColRank (C++ function), 50
 AbstractDistMatrix<T>::PartialUnionColStride (C++ function), 50
 AbstractDistMatrix<T>::PartialUnionRowComm (C++ function), 49
 AbstractDistMatrix<T>::PartialUnionRowRank (C++ function), 50
 AbstractDistMatrix<T>::PartialUnionRowStride (C++ function), 50
 AbstractDistMatrix<T>::Participating (C++ function), 50
 AbstractDistMatrix<T>::RedundantComm (C++ function), 49
 AbstractDistMatrix<T>::RedundantRank (C++ function), 50
 AbstractDistMatrix<T>::RedundantSize (C++ function), 50
 AbstractDistMatrix<T>::Resize (C++ function), 47
 AbstractDistMatrix<T>::Root (C++ function), 50
 AbstractDistMatrix<T>::RootConstrained (C++ function), 49
 AbstractDistMatrix<T>::RowAlign (C++ function), 49
 AbstractDistMatrix<T>::RowComm (C++ function), 49
 AbstractDistMatrix<T>::RowConstrained (C++ function), 49
 AbstractDistMatrix<T>::RowOwner (C++ function), 50
 AbstractDistMatrix<T>::RowRank (C++ function), 50
 AbstractDistMatrix<T>::RowShift (C++ function), 49
 AbstractDistMatrix<T>::RowStride (C++ function), 50
 AbstractDistMatrix<T>::Set (C++ function), 51
 AbstractDistMatrix<T>::SetGrid (C++ function), 47
 AbstractDistMatrix<T>::SetImagPart (C++ function), 51
 AbstractDistMatrix<T>::SetImagPartOfLocalSubmatrix (C++ function), 54
 AbstractDistMatrix<T>::SetImagPartOfSubmatrix (C++ function), 53
 AbstractDistMatrix<T>::SetLocal (C++ function), 52
 AbstractDistMatrix<T>::SetLocalImagPart (C++ function), 52
 AbstractDistMatrix<T>::SetLocalRealPart (C++ function), 52
 AbstractDistMatrix<T>::SetLocalSubmatrix (C++ function), 54
 AbstractDistMatrix<T>::SetRealPart (C++ function), 51
 AbstractDistMatrix<T>::SetRealPartOfLocalSubmatrix (C++ function), 54
 AbstractDistMatrix<T>::SetRealPartOfSubmatrix (C++ function), 53
 AbstractDistMatrix<T>::SetRoot (C++ function), 47
 AbstractDistMatrix<T>::SetSubmatrix (C++ function), 53
 AbstractDistMatrix<T>::SumOver (C++ function), 54
 AbstractDistMatrix<T>::Update (C++ function), 51
 AbstractDistMatrix<T>::UpdateImagPart (C++ function), 51
 AbstractDistMatrix<T>::UpdateImagPartOfLocalSubmatrix (C++ function), 54
 AbstractDistMatrix<T>::UpdateImagPartOfSubmatrix (C++ function), 53
 AbstractDistMatrix<T>::UpdateLocal (C++ function), 52
 AbstractDistMatrix<T>::UpdateLocalImagPart (C++ function), 52
 AbstractDistMatrix<T>::UpdateLocalSubmatrix (C++ function), 54
 AbstractDistMatrix<T>::UpdateRealPart (C++ function), 51
 AbstractDistMatrix<T>::UpdateRealPartLocal (C++ function), 52
 AbstractDistMatrix<T>::UpdateRealPartOfLocalSubmatrix (C++ function), 54
 AbstractDistMatrix<T>::UpdateRealPartOfSubmatrix (C++ function), 53
 AbstractDistMatrix<T>::UpdateSubmatrix (C++ function), 53
 AbstractDistMatrix<T>::Viewing (C++ function), 48
 AbstractDistMatrix<T>::Width (C++ function), 48
 Adjoint (C++ function), 89
 align (C++ member), 136
 ApplyColPivots (C++ function), 137

- ApplyInverseColPivots (C++ function), 137
 ApplyInverseRowPivots (C++ function), 137
 ApplyInverseSymmetricPivots (C++ function), 138
 ApplyPackedReflectors (C++ function), 153
 ApplyRowPivots (C++ function), 137
 ApplySymmetricPivots (C++ function), 137
 Arg (C++ function), 33
 Axy (C++ function), 89
 AxyInterface<T> (C++ class), 86
 AxyInterface<T>::Attach (C++ function), 86
 AxyInterface<T>::Axy (C++ function), 87
 AxyInterface<T>::AxyInterface (C++ function), 86
 AxyInterface<T>::Detach (C++ function), 87
 AxyType (C++ enum), 86
 AxyType::GLOBAL_TO_LOCAL (C++ enumerator), 86
 AxyType::LOCAL_TO_GLOBAL (C++ enumerator), 86
- ## B
- Base<F> (C++ type), 32
 BasisPursuit (C++ function), 155
 Bidiag (C++ function), 107
 bidiag::ApplyP (C++ function), 107
 bidiag::ApplyQ (C++ function), 107
 blacs::Desc (C++ type), 28
 blacs::Exit (C++ function), 28
 blacs::FreeGrid (C++ function), 28
 blacs::FreeHandle (C++ function), 28
 blacs::GridCol (C++ function), 28
 blacs::GridHeight (C++ function), 28
 blacs::GridInit (C++ function), 28
 blacs::GridRow (C++ function), 28
 blacs::GridWidth (C++ function), 28
 blacs::Handle (C++ function), 28
 blas::Axy (C++ function), 13
 blas::Dot (C++ function), 13
 blas::Dotc (C++ function), 13
 blas::Dotu (C++ function), 13
 blas::Gemm (C++ function), 15
 blas::Gevv (C++ function), 14
 blas::Ger (C++ function), 14
 blas::Gerc (C++ function), 14
 blas::Geru (C++ function), 14
 blas::Hemm (C++ function), 15
 blas::Hemv (C++ function), 14
 blas::Her (C++ function), 14
 blas::Her2 (C++ function), 14
 blas::Her2k (C++ function), 15
 blas::Herk (C++ function), 15
 blas::Hetrm (C++ function), 16
 blas::Nrm2 (C++ function), 13
 blas::Rot (C++ function), 14
 blas::Scal (C++ function), 14
 blas::Swap (C++ function), 14
 blas::Symm (C++ function), 16
 blas::Symv (C++ function), 14
 blas::Syr (C++ function), 15
 blas::Syr2 (C++ function), 15
 blas::Syr2k (C++ function), 16
 blas::Syrk (C++ function), 16
 blas::Trmm (C++ function), 16
 blas::Trmv (C++ function), 15
 blas::Trsm (C++ function), 16
 blas::Trsv (C++ function), 15
 Blocksize (C++ function), 31
 byte (C++ type), 34
- ## C
- Cauchy (C++ function), 165
 CauchyLike (C++ function), 165, 166
 Cholesky (C++ function), 120
 cholesky::SolveAfter (C++ function), 121
 CholeskyMod (C++ function), 120
 Circulant (C++ function), 166
 Clip (C++ function), 159
 ColumnSwap (C++ function), 93
 Comm (C++ function), 19
 comm (C++ member), 19, 136
 Complex<Real> (C++ type), 32
 ComplexHermitianFunction (C++ function), 132
 Condition (C++ function), 139
 Conj (C++ function), 33
 Conjugate (C++ function), 89, 90
 Conjugation (C++ enum), 34
 Conjugation::CONJUGATED (C++ enumerator), 34
 Conjugation::UNCONJUGATED (C++ enumerator), 34
 Copy (C++ function), 90
 Cos (C++ function), 33
 Cosh (C++ function), 33
- ## D
- d0 (C++ member), 127
 dcomplex (C++ type), 33
 DefaultBlockHeight (C++ function), 31
 DefaultBlockWidth (C++ function), 31
 DefaultGrid (C++ function), 31
 Determinant (C++ function), 139, 140

- Diagonal (C++ function), 166
- DiagonalScale (C++ function), 90
- DiagonalScaleTrapezoid (C++ function), 90
- DiagonalSolve (C++ function), 90
- Display (C++ function), 177, 178
- DistData (C++ class), 55
- DistData::colAlign (C++ member), 55
- DistData::colDist (C++ member), 55
- DistData::DistData (C++ function), 55
- DistData::grid (C++ member), 55
- DistData::root (C++ member), 55
- DistData::rowAlign (C++ member), 55
- DistData::rowDist (C++ member), 55
- DistMatrix<Complex<double>, CIRC, CIRC> (C++ class), 67
- DistMatrix<Complex<double>, MC, MR> (C++ class), 67
- DistMatrix<Complex<double>, MC, STAR> (C++ class), 67
- DistMatrix<Complex<double>, MD, STAR> (C++ class), 67
- DistMatrix<Complex<double>, MR, MC> (C++ class), 67
- DistMatrix<Complex<double>, MR, STAR> (C++ class), 67
- DistMatrix<Complex<double>, STAR, MC> (C++ class), 68
- DistMatrix<Complex<double>, STAR, MD> (C++ class), 68
- DistMatrix<Complex<double>, STAR, MR> (C++ class), 68
- DistMatrix<Complex<double>, STAR, STAR> (C++ class), 68
- DistMatrix<Complex<double>, STAR, VC> (C++ class), 68
- DistMatrix<Complex<double>, STAR, VR> (C++ class), 68
- DistMatrix<Complex<double>, U, V> (C++ class), 67
- DistMatrix<Complex<double>, VC, STAR> (C++ class), 68
- DistMatrix<Complex<double>, VR, STAR> (C++ class), 68
- DistMatrix<Complex<double>> (C++ class), 67
- DistMatrix<Complex<Real>, CIRC, CIRC> (C++ class), 68
- DistMatrix<Complex<Real>, MC, MR> (C++ class), 68
- DistMatrix<Complex<Real>, MC, STAR> (C++ class), 68
- DistMatrix<Complex<Real>, MD, STAR> (C++ class), 68
- DistMatrix<Complex<Real>, MR, MC> (C++ class), 68
- DistMatrix<Complex<Real>, MR, STAR> (C++ class), 68
- DistMatrix<Complex<Real>, STAR, MC> (C++ class), 68
- DistMatrix<Complex<Real>, STAR, MD> (C++ class), 69
- DistMatrix<Complex<Real>, STAR, MR> (C++ class), 69
- DistMatrix<Complex<Real>, STAR, STAR> (C++ class), 69
- DistMatrix<Complex<Real>, STAR, VC> (C++ class), 69
- DistMatrix<Complex<Real>, STAR, VR> (C++ class), 69
- DistMatrix<Complex<Real>, U, V> (C++ class), 68
- DistMatrix<Complex<Real>, VC, STAR> (C++ class), 69
- DistMatrix<Complex<Real>, VR, STAR> (C++ class), 69
- DistMatrix<Complex<Real>> (C++ class), 68
- DistMatrix<double, CIRC, CIRC> (C++ class), 67
- DistMatrix<double, MC, MR> (C++ class), 67
- DistMatrix<double, MC, STAR> (C++ class), 67
- DistMatrix<double, MD, STAR> (C++ class), 67
- DistMatrix<double, MR, MC> (C++ class), 67
- DistMatrix<double, MR, STAR> (C++ class), 67
- DistMatrix<double, STAR, MC> (C++ class), 67
- DistMatrix<double, STAR, MD> (C++ class), 67
- DistMatrix<double, STAR, MR> (C++ class), 67
- DistMatrix<double, STAR, STAR> (C++ class), 67
- DistMatrix<double, STAR, VC> (C++ class), 67
- DistMatrix<double, STAR, VR> (C++ class), 67
- DistMatrix<double, U, V> (C++ class), 67
- DistMatrix<double, VC, STAR> (C++ class), 67

- DistMatrix<double, VR, STAR> (C++ class), 67
- DistMatrix<double> (C++ class), 67
- DistMatrix<F, CIRC, CIRC> (C++ class), 69
- DistMatrix<F, MC, MR> (C++ class), 69
- DistMatrix<F, MC, STAR> (C++ class), 69
- DistMatrix<F, MD, STAR> (C++ class), 69
- DistMatrix<F, MR, MC> (C++ class), 69
- DistMatrix<F, MR, STAR> (C++ class), 69
- DistMatrix<F, STAR, MC> (C++ class), 69
- DistMatrix<F, STAR, MD> (C++ class), 69
- DistMatrix<F, STAR, MR> (C++ class), 69
- DistMatrix<F, STAR, STAR> (C++ class), 69
- DistMatrix<F, STAR, VC> (C++ class), 69
- DistMatrix<F, STAR, VR> (C++ class), 69
- DistMatrix<F, U, V> (C++ class), 69
- DistMatrix<F, VC, STAR> (C++ class), 69
- DistMatrix<F, VR, STAR> (C++ class), 69
- DistMatrix<F> (C++ class), 69
- DistMatrix<int, CIRC, CIRC> (C++ class), 69
- DistMatrix<int, MC, MR> (C++ class), 69
- DistMatrix<int, MC, STAR> (C++ class), 69
- DistMatrix<int, MD, STAR> (C++ class), 69
- DistMatrix<int, MR, MC> (C++ class), 69
- DistMatrix<int, MR, STAR> (C++ class), 69
- DistMatrix<int, STAR, MC> (C++ class), 69
- DistMatrix<int, STAR, MD> (C++ class), 69
- DistMatrix<int, STAR, MR> (C++ class), 70
- DistMatrix<int, STAR, STAR> (C++ class), 70
- DistMatrix<int, STAR, VC> (C++ class), 70
- DistMatrix<int, STAR, VR> (C++ class), 70
- DistMatrix<int, U, V> (C++ class), 69
- DistMatrix<int, VC, STAR> (C++ class), 70
- DistMatrix<int, VR, STAR> (C++ class), 70
- DistMatrix<int> (C++ class), 69
- DistMatrix<Real, CIRC, CIRC> (C++ class), 68
- DistMatrix<Real, MC, MR> (C++ class), 68
- DistMatrix<Real, MC, STAR> (C++ class), 68
- DistMatrix<Real, MD, STAR> (C++ class), 68
- DistMatrix<Real, MR, MC> (C++ class), 68
- DistMatrix<Real, MR, STAR> (C++ class), 68
- DistMatrix<Real, STAR, MC> (C++ class), 68
- DistMatrix<Real, STAR, MD> (C++ class), 68
- DistMatrix<Real, STAR, MR> (C++ class), 68
- DistMatrix<Real, STAR, STAR> (C++ class), 68
- DistMatrix<Real, STAR, VC> (C++ class), 68
- DistMatrix<Real, STAR, VR> (C++ class), 68
- DistMatrix<Real, U, V> (C++ class), 68
- DistMatrix<Real, VC, STAR> (C++ class), 68
- DistMatrix<Real, VR, STAR> (C++ class), 68
- DistMatrix<Real> (C++ class), 68
- DistMatrix<T, CIRC, CIRC> (C++ class), 66
- DistMatrix<T, CIRC>::CopyFromNonRoot (C++ function), 67
- DistMatrix<T, CIRC, CIRC>::CopyFromRoot (C++ function), 67
- DistMatrix<T, MC, MR> (C++ class), 62
- DistMatrix<T, MC, STAR> (C++ class), 62
- DistMatrix<T, MD, STAR> (C++ class), 64
- DistMatrix<T, MR, MC> (C++ class), 63
- DistMatrix<T, MR, STAR> (C++ class), 63
- DistMatrix<T, STAR, MC> (C++ class), 63
- DistMatrix<T, STAR, MD> (C++ class), 64
- DistMatrix<T, STAR, MR> (C++ class), 62
- DistMatrix<T, STAR, STAR> (C++ class), 66
- DistMatrix<T, STAR, VC> (C++ class), 65
- DistMatrix<T, STAR, VR> (C++ class), 66
- DistMatrix<T, U, V> (C++ class), 60
- DistMatrix<T, U, V>::~DistMatrix (C++ function), 61
- DistMatrix<T, U, V>::DistMatrix (C++ function), 61
- DistMatrix<T, U, V>::operator= (C++ function), 61
- DistMatrix<T, VC, STAR> (C++ class), 65
- DistMatrix<T, VR, STAR> (C++ class), 65
- DistMatrix<T> (C++ class), 62
- Distribution (C++ enum), 34
- Distribution::CIRC (C++ enumerator), 34
- Distribution::MC (C++ enumerator), 34
- Distribution::MD (C++ enumerator), 34
- Distribution::MR (C++ enumerator), 34
- Distribution::STAR (C++ enumerator), 34
- Distribution::VC (C++ enumerator), 34
- Distribution::VR (C++ enumerator), 34
- dList (C++ member), 127
- Dot (C++ function), 91
- Dotc (C++ function), 91
- Dotu (C++ function), 91
- DumpCallStack (C++ function), 32
- ## E
- Egorov (C++ function), 166
- EntrywiseMap (C++ function), 91
- EntrywiseNorm (C++ function), 141
- EntrywiseOneNorm (C++ function), 141
- Exp (C++ function), 34
- ExpandPackedReflectors (C++ function), 153
- ExplicitPermutation (C++ function), 136

- ExtendedKahan (C++ function), 166
- ## F
- FastAbs (C++ function), 33
 Fiedler (C++ function), 167
 FileFormat (C++ enum), 177
 FileFormat::ASCII (C++ enumerator), 177
 FileFormat::ASCII_MATLAB (C++ enumerator), 177
 FileFormat::AUTO (C++ enumerator), 177
 FileFormat::BINARY (C++ enumerator), 177
 FileFormat::BINARY_FLAT (C++ enumerator), 177
 FileFormat::BMP (C++ enumerator), 177
 FileFormat::JPEG (C++ enumerator), 177
 FileFormat::JPG (C++ enumerator), 177
 FileFormat::MATRIX_MARKET (C++ enumerator), 177
 FileFormat::PNG (C++ enumerator), 177
 FileFormat::PPM (C++ enumerator), 177
 FileFormat::XBM (C++ enumerator), 177
 FileFormat::XPM (C++ enumerator), 177
 FillDesc (C++ function), 28
 Finalize (C++ function), 30
 FLA_Bsvd_v_opd_var1 (C++ function), 28
 Forsythe (C++ function), 167
 ForwardOrBackward (C++ enum), 34
 ForwardOrBackward::BACKWARD (C++ enumerator), 34
 ForwardOrBackward::FORWARD (C++ enumerator), 34
 Fourier (C++ function), 167
 FrobeniusCondition (C++ function), 139
 FrobeniusNorm (C++ function), 142
- ## G
- Gaussian (C++ function), 174
 GaussianElimination (C++ function), 150
 GCD (C++ function), 36
 GCDMatrix (C++ function), 167
 Gear (C++ function), 167
 Gemm (C++ function), 98
 Gemv (C++ function), 95
 GeneralDistMatrix<T, U, V> (C++ class), 55
 GeneralDistMatrix<T, U, V>::AdjointColAllGather (C++ function), 58
 GeneralDistMatrix<T, U, V>::AdjointColFilterFrom (C++ function), 59
 GeneralDistMatrix<T, U, V>::AdjointColSumScatterFrom (C++ function), 59
 GeneralDistMatrix<T, U, V>::AdjointColSumScatterUpdate (C++ function), 59
 GeneralDistMatrix<T, U, V>::AdjointPartialColAllGather (C++ function), 58
 GeneralDistMatrix<T, U, V>::AdjointPartialColFilterFrom (C++ function), 59
 GeneralDistMatrix<T, U, V>::AdjointPartialColSumScatterFrom (C++ function), 59
 GeneralDistMatrix<T, U, V>::AdjointPartialColSumScatterUpdate (C++ function), 59
 GeneralDistMatrix<T, U, V>::AdjointPartialRowFilterFrom (C++ function), 59
 GeneralDistMatrix<T, U, V>::AdjointRowFilterFrom (C++ function), 59
 GeneralDistMatrix<T, U, V>::AllGather (C++ function), 56
 GeneralDistMatrix<T, U, V>::ColAllGather (C++ function), 56
 GeneralDistMatrix<T, U, V>::ColFilterFrom (C++ function), 57
 GeneralDistMatrix<T, U, V>::ColSumScatterFrom (C++ function), 57
 GeneralDistMatrix<T, U, V>::ColSumScatterUpdate (C++ function), 58
 GeneralDistMatrix<T, U, V>::FilterFrom (C++ function), 57
 GeneralDistMatrix<T, U, V>::GeneralDistMatrix (C++ function), 56
 GeneralDistMatrix<T, U, V>::operator= (C++ function), 56
 GeneralDistMatrix<T, U, V>::PartialColAllGather (C++ function), 57
 GeneralDistMatrix<T, U, V>::PartialColAllToAll (C++ function), 57
 GeneralDistMatrix<T, U, V>::PartialColAllToAllFrom (C++ function), 57

			function), 57						function), 58
GeneralDistMatrix<T,		U,	V>::PartialColFilterFrom	(C++	GeneralDistMatrix<T,		U,	V>::TransposeColSumScatterFrom	(C++ function), 59
			function), 57						
GeneralDistMatrix<T,		U,	V>::PartialColSumScatterFrom	(C++ function), 57	GeneralDistMatrix<T,		U,	V>::TransposeColSumScatterUpdate	(C++ function), 59
GeneralDistMatrix<T,		U,	V>::PartialColSumScatterUpdate	(C++ function), 58	GeneralDistMatrix<T,		U,	V>::TransposePartialColAllGather	(C++ function), 58
GeneralDistMatrix<T,		U,	V>::PartialRowAllGather	(C++	GeneralDistMatrix<T,		U,	V>::TransposePartialColFilterFrom	(C++ function), 58
			function), 57						
GeneralDistMatrix<T,		U,	V>::PartialRowAllToAll	(C++ func-	GeneralDistMatrix<T,		U,	V>::TransposePartialColSumScatterFrom	(C++ function), 59
			tion), 57						
GeneralDistMatrix<T,		U,	V>::PartialRowAllToAllFrom	(C++	GeneralDistMatrix<T,		U,	V>::TransposePartialColSumScatterUpdate	(C++ function), 59
			function), 57						
GeneralDistMatrix<T,		U,	V>::PartialRowFilterFrom	(C++	GeneralDistMatrix<T,		U,	V>::TransposePartialRowFilterFrom	(C++ function), 58
			function), 57						
GeneralDistMatrix<T,		U,	V>::PartialRowSumScatterFrom	(C++	GeneralDistMatrix<T,		U,	V>::TransposeRowFilterFrom	(C++
			function), 57						function), 58
GeneralDistMatrix<T,		U,	V>::PartialRowSumScatterUpdate	(C++	GeneralDistMatrix<T,	U,	V>::UDiag	(C++	member), 56
			function), 58						
GeneralDistMatrix<T,	U,	V>::RowAllGather	(C++ function), 56		GeneralDistMatrix<T,	U,	V>::UGath	(C++	member), 56
GeneralDistMatrix<T,	U,	V>::RowFilterFrom	(C++ function), 57		GeneralDistMatrix<T,	U,	V>::UPart	(C++	member), 56
GeneralDistMatrix<T,	U,	V>::RowSumScatterFrom	(C++	GeneralDistMatrix<T,	U,	V>::UScat	(C++	member), 56	
			function), 57						
GeneralDistMatrix<T,	U,	V>::RowSumScatterUpdate	(C++	GeneralDistMatrix<T,	U,	V>::VDiag	(C++	member), 56	
			function), 58						
GeneralDistMatrix<T,	U,	V>::SumScatterFrom	(C++ func-	GeneralDistMatrix<T,	U,	V>::VPart	(C++	member), 56	
			tion), 57						
GeneralDistMatrix<T,	U,	V>::SumScatterUpdate	(C++ func-	GeneralDistMatrix<T,	U,	V>::VScat	(C++	member), 56	
			tion), 58						
GeneralDistMatrix<T,	U,	V>::Translate	(C++	Ger	(C++ function), 95				
			function), 56		Gerc	(C++ function), 95			
GeneralDistMatrix<T,	U,	V>::TransposeColAllGather	(C++	Geru	(C++ function), 96				
			function), 58		GKS	(C++ function), 168			
GeneralDistMatrix<T,	U,	V>::TransposeColFilterFrom	(C++	GLM	(C++ function), 151				
			function), 58		GQR	(C++ function), 128			
GeneralDistMatrix<T,	U,	V>::TransposeColFilterFrom	(C++	Grcar	(C++ function), 168				
			function), 58		Grid	(C++ class), 42			
GeneralDistMatrix<T,	U,	V>::TransposeColFilterFrom	(C++	Grid::Col	(C++ function), 43				
			function), 58		Grid::ColComm	(C++ function), 43			

- Grid::Comm (C++ function), 43
 Grid::DiagPath (C++ function), 45
 Grid::DiagPathRank (C++ function), 45
 Grid::GCD (C++ function), 44
 Grid::Grid (C++ function), 42, 44
 Grid::HaveViewers (C++ function), 45
 Grid::Height (C++ function), 43
 Grid::InGrid (C++ function), 45
 Grid::LCM (C++ function), 45
 Grid::MCComm (C++ function), 44
 Grid::MCRank (C++ function), 43
 Grid::MCSize (C++ function), 44
 Grid::MRComm (C++ function), 44
 Grid::MRRank (C++ function), 43
 Grid::MRSIZE (C++ function), 44
 Grid::Order (C++ function), 43
 Grid::OwningComm (C++ function), 45
 Grid::OwningGroup (C++ function), 45
 Grid::OwningRank (C++ function), 45
 Grid::Rank (C++ function), 43
 Grid::Row (C++ function), 43
 Grid::RowComm (C++ function), 43
 Grid::Size (C++ function), 43
 Grid::VCComm (C++ function), 44
 Grid::VCRank (C++ function), 43
 Grid::VCSIZE (C++ function), 44
 Grid::VCToViewingMap (C++ function), 45
 Grid::ViewingComm (C++ function), 45
 Grid::ViewingRank (C++ function), 45
 Grid::VRComm (C++ function), 44
 Grid::VRRank (C++ function), 44
 Grid::VRSize (C++ function), 44
 Grid::Width (C++ function), 43
 GridOrder (C++ enum), 34
 GridOrder::COLUMN_MAJOR (C++ enumerator), 34
 GridOrder::ROW_MAJOR (C++ enumerator), 34
 Group (C++ function), 20
 group (C++ member), 20
 GRQ (C++ function), 129
- ## H
- Haar (C++ function), 175
 Hadamard (C++ function), 91
 Hankel (C++ function), 168
 Hanowa (C++ function), 168
 Helmholtz (C++ function), 168
 Hemm (C++ function), 98
 Hemv (C++ function), 96
 Her (C++ function), 96
 Her2 (C++ function), 96
 Her2k (C++ function), 98
 Herk (C++ function), 98, 99
 herm_eig::SDC (C++ function), 110
 herm_tridiag_eig::Eig (C++ function), 27
 herm_tridiag_eig::EigEstimate (C++ function), 27
 herm_tridiag_eig::Estimate (C++ class), 26
 herm_tridiag_eig::Estimate::numGlobalEigenvalues (C++ member), 26
 herm_tridiag_eig::Estimate::numLocalEigenvalues (C++ member), 26
 herm_tridiag_eig::Info (C++ class), 26
 herm_tridiag_eig::Info::firstLocalEigenvalue (C++ member), 27
 herm_tridiag_eig::Info::numGlobalEigenvalues (C++ member), 27
 herm_tridiag_eig::Info::numLocalEigenvalues (C++ member), 27
 hermitian_polar::QDWH (C++ function), 118, 119
 hermitian_tridiag::ApplyQ (C++ function), 106
 HermitianEig (C++ function), 108–110
 HermitianEigCtrl<Real> (C++ class), 108
 HermitianEigCtrl<Real>::HermitianEigCtrl (C++ function), 108
 HermitianEigCtrl<Real>::sdcCtrl (C++ member), 108
 HermitianEigCtrl<Real>::tridiagCtrl (C++ member), 108
 HermitianEigCtrl<Real>::useSdc (C++ member), 108
 HermitianEntrywiseNorm (C++ function), 141
 HermitianEntrywiseOneNorm (C++ function), 141
 HermitianFrobeniusNorm (C++ function), 142
 HermitianFromEVD (C++ function), 169
 HermitianGenDefiniteEig (C++ function), 113, 114
 HermitianGenDefiniteEigType (C++ enum), 112
 HermitianGenDefiniteEigType::ABX (C++ enumerator), 112
 HermitianGenDefiniteEigType::AXBX (C++ enumerator), 113
 HermitianGenDefiniteEigType::BAX (C++ enumerator), 113
 HermitianInfinityNorm (C++ function), 142

- HermitianInverse (C++ function), 131
HermitianKyFanNorm (C++ function), 142
HermitianMaxNorm (C++ function), 142
HermitianNorm (C++ function), 141
HermitianNuclearNorm (C++ function), 142, 143
HermitianOneNorm (C++ function), 143
HermitianPolar (C++ function), 118
HermitianPseudoinverse (C++ function), 133
HermitianSchattenNorm (C++ function), 143
HermitianSdcCtrl<Base<F>> (C++ type), 110
HermitianSdcCtrl<Real> (C++ class), 110
HermitianSdcCtrl<Real>::cutoff (C++ member), 110
HermitianSdcCtrl<Real>::maxInnerIts (C++ member), 110
HermitianSdcCtrl<Real>::maxOuterIts (C++ member), 110
HermitianSdcCtrl<Real>::progress (C++ member), 110
HermitianSdcCtrl<Real>::random (C++ member), 110
HermitianSdcCtrl<Real>::spreadFactor (C++ member), 110
HermitianSdcCtrl<Real>::tol (C++ member), 110
HermitianSign (C++ function), 134
HermitianSolve (C++ function), 150
HermitianSVD (C++ function), 117
HermitianTridiag (C++ function), 105, 106
HermitianTridiagApproach (C++ type), 154
HermitianTridiagCtrl (C++ class), 105
HermitianTridiagCtrl::approach (C++ member), 105
HermitianTridiagCtrl::HermitianTridiagCtrl (C++ function), 105
HermitianTridiagCtrl::order (C++ member), 105
HermitianTwoNorm (C++ function), 143
HermitianTwoNormEstimate (C++ function), 144
HermitianUniformSpectrum (C++ function), 175
Hessenberg (C++ function), 106
hessenberg::ApplyQ (C++ function), 106, 107
HessenbergPseudospectrum (C++ function), 146–148
Hilbert (C++ function), 169
HilbertSchmidt (C++ function), 92
HPDDeterminant (C++ function), 140
HPDInverse (C++ function), 132
HPDSolve (C++ function), 149
HPSDCholesky (C++ function), 121
HPSDSquareRoot (C++ function), 133
hyp_reflector::Col (C++ function), 153
hyp_reflector::Row (C++ function), 153
- I
- ID (C++ function), 130
Identity (C++ function), 169
ImagPart (C++ function), 33
ImplicitHaar (C++ function), 175
Inertia (C++ function), 149
InertiaType (C++ type), 149
InfinityCondition (C++ function), 139
InfinityNorm (C++ function), 142
Initialize (C++ function), 30
Initialized (C++ function), 30
Inverse (C++ function), 131
InversePermuteCols (C++ function), 135
InversePermuteRows (C++ function), 135
InvertPermutation (C++ function), 136
- J
- Jordan (C++ function), 169
- K
- Kahan (C++ function), 170
kappa (C++ member), 140
KMS (C++ function), 170
KyFanNorm (C++ function), 142
- L
- lapack::BidiagQRAlg (C++ function), 18
lapack::DivideAndConquerSVD (C++ function), 18
lapack::Eig (C++ function), 19
lapack::Givens (C++ function), 17
lapack::HermitianEig (C++ function), 18
lapack::HessenbergEig (C++ function), 19
lapack::HessenbergSchur (C++ function), 18
lapack::MachineEpsilon<Real> (C++ function), 17
lapack::MachineOverflowExponent<Real> (C++ function), 17
lapack::MachineOverflowThreshold<Real> (C++ function), 17
lapack::MachinePrecision<Real> (C++ function), 17
lapack::MachineSafeMin<Real> (C++ function), 17

- lapack::MachineUnderflowExponent<Real> (C++ function), 17
- lapack::MachineUnderflowThreshold<Real> (C++ function), 17
- lapack::QRSVD (C++ function), 18
- lapack::SafeNorm (C++ function), 17
- lapack::Schur (C++ function), 19
- lapack::SVD (C++ function), 18
- Laplacian (C++ function), 170
- Lauchli (C++ function), 170
- ldl::SolveAfter (C++ function), 122
- LDLH (C++ function), 121, 122
- LDLPivot (C++ class), 121
- LDLPivot::from (C++ member), 121
- LDLPivot::nb (C++ member), 121
- LDLPivotType (C++ enum), 121
- LDLPivotType::BUNCH.KAUFMAN_A (C++ enumerator), 121
- LDLPivotType::BUNCH.KAUFMAN_BOUNDED (C++ enumerator), 121
- LDLPivotType::BUNCH.KAUFMAN_C (C++ enumerator), 121
- LDLPivotType::BUNCH.KAUFMAN_D (C++ enumerator), 121
- LDLPivotType::BUNCH.PARLETT (C++ enumerator), 121
- LDLT (C++ function), 121, 122
- LeastSquares (C++ function), 151
- LeftHyperbolicReflector (C++ function), 153
- LeftOrRight (C++ enum), 35
- LeftOrRight::LEFT (C++ enumerator), 35
- LeftOrRight::RIGHT (C++ enumerator), 35
- LeftReflector (C++ function), 152
- Legendre (C++ function), 171
- Lehmer (C++ function), 171
- Length (C++ function), 36
- LinearProgram (C++ function), 156
- LocalSymvBlocksize<T> (C++ function), 104
- LocalTrr2kBlocksize<T> (C++ function), 104
- LocalTrrkBlocksize<T> (C++ function), 104
- LockedPartitionDown (C++ function), 73
- LockedPartitionDownDiagonal (C++ function), 75, 76
- LockedPartitionDownOffsetDiagonal (C++ function), 76
- LockedPartitionLeft (C++ function), 73
- LockedPartitionRight (C++ function), 74
- LockedPartitionUp (C++ function), 72, 73
- LockedPartitionUpDiagonal (C++ function), 74
- LockedPartitionUpOffsetDiagonal (C++ function), 75
- LockedRepartitionDown (C++ function), 77, 78
- LockedRepartitionDownDiagonal (C++ function), 80, 81
- LockedRepartitionLeft (C++ function), 78
- LockedRepartitionRight (C++ function), 79
- LockedRepartitionUp (C++ function), 77
- LockedRepartitionUpDiagonal (C++ function), 79, 80
- LockedView (C++ function), 70, 71
- LockedView1x2 (C++ function), 71
- LockedView2x1 (C++ function), 71
- LockedView2x2 (C++ function), 72
- Log (C++ function), 34
- Log2 (C++ function), 37
- LogBarrier (C++ function), 159
- LogDetDivergence (C++ function), 159, 160
- Lotkin (C++ function), 171
- LowerClip (C++ function), 159
- LQ (C++ function), 124
- lq::ApplyQ (C++ function), 124
- lq::SolveAfter (C++ function), 124, 125
- LSE (C++ function), 151, 152
- LU (C++ function), 123
- lu::SolveAfter (C++ function), 123, 124
- LUMod (C++ function), 123
- Lyapunov (C++ function), 163
- ## M
- MakeFourier (C++ function), 167
- MakeGaussian (C++ function), 174
- MakeHermitianUniformSpectrum (C++ function), 175
- MakeHilbert (C++ function), 169
- MakeIdentity (C++ function), 169
- MakeNormalUniformSpectrum (C++ function), 176
- MakeOnes (C++ function), 172
- MakeOneTwoOne (C++ function), 172
- MakeTrapezoidal (C++ function), 92
- MakeUniform (C++ function), 175
- MakeZeros (C++ function), 174
- Matrix<Complex<Real>> (C++ class), 42
- Matrix<F> (C++ class), 42
- Matrix<int> (C++ class), 42
- Matrix<Real> (C++ class), 42
- Matrix<T> (C++ class), 38
- Matrix<T>::~Matrix (C++ function), 38
- Matrix<T>::~Attach (C++ function), 39

- Matrix<T>::Buffer (C++ function), 39
 Matrix<T>::Conjugate (C++ function), 40
 Matrix<T>::ConjugateDiagonal (C++ function), 41
 Matrix<T>::ConjugateSubmatrix (C++ function), 42
 Matrix<T>::Control (C++ function), 39
 Matrix<T>::DiagonalLength (C++ function), 39
 Matrix<T>::Empty (C++ function), 39
 Matrix<T>::FixedSize (C++ function), 40
 Matrix<T>::Get (C++ function), 40
 Matrix<T>::GetDiagonal (C++ function), 40
 Matrix<T>::GetImagPart (C++ function), 40
 Matrix<T>::GetImagPartOfDiagonal (C++ function), 40
 Matrix<T>::GetImagPartOfSubmatrix (C++ function), 41
 Matrix<T>::GetRealPart (C++ function), 40
 Matrix<T>::GetRealPartOfDiagonal (C++ function), 40
 Matrix<T>::GetRealPartOfSubmatrix (C++ function), 41
 Matrix<T>::GetSubmatrix (C++ function), 41
 Matrix<T>::Height (C++ function), 39
 Matrix<T>::LDim (C++ function), 39
 Matrix<T>::Locked (C++ function), 40
 Matrix<T>::LockedAttach (C++ function), 39
 Matrix<T>::LockedBuffer (C++ function), 39
 Matrix<T>::MakeDiagonalReal (C++ function), 41
 Matrix<T>::MakeReal (C++ function), 40
 Matrix<T>::MakeSubmatrixReal (C++ function), 42
 Matrix<T>::Matrix (C++ function), 38
 Matrix<T>::MemorySize (C++ function), 39
 Matrix<T>::operator= (C++ function), 39
 Matrix<T>::Resize (C++ function), 39
 Matrix<T>::Set (C++ function), 40
 Matrix<T>::SetDiagonal (C++ function), 40
 Matrix<T>::SetImagPart (C++ function), 40
 Matrix<T>::SetImagPartOfDiagonal (C++ function), 41
 Matrix<T>::SetImagPartOfSubmatrix (C++ function), 41
 Matrix<T>::SetRealPart (C++ function), 40
 Matrix<T>::SetRealPartOfDiagonal (C++ function), 41
 Matrix<T>::SetRealPartOfSubmatrix (C++ function), 41
 Matrix<T>::SetSubmatrix (C++ function), 41
 Matrix<T>::Update (C++ function), 40
 Matrix<T>::UpdateDiagonal (C++ function), 41
 Matrix<T>::UpdateImagPart (C++ function), 40
 Matrix<T>::UpdateImagPartOfDiagonal (C++ function), 41
 Matrix<T>::UpdateImagPartOfSubmatrix (C++ function), 42
 Matrix<T>::UpdateRealPart (C++ function), 40
 Matrix<T>::UpdateRealPartOfDiagonal (C++ function), 41
 Matrix<T>::UpdateRealPartOfSubmatrix (C++ function), 42
 Matrix<T>::UpdateSubmatrix (C++ function), 41
 Matrix<T>::Viewing (C++ function), 39
 Matrix<T>::Width (C++ function), 39
 MaxCondition (C++ function), 139
 maxIts (C++ member), 134
 MaxLength (C++ function), 36
 MaxNorm (C++ function), 142
 Median (C++ function), 154
 MinIJ (C++ function), 171
 Mod (C++ function), 36
 mpi::AllGather (C++ function), 25
 mpi::AllReduce (C++ function), 26
 mpi::AllToAll (C++ function), 26
 mpi::ANY_SOURCE (C++ member), 20
 mpi::ANY_TAG (C++ member), 20
 mpi::Barrier (C++ function), 24
 mpi::BINARY_AND (C++ member), 21
 mpi::BINARY_OR (C++ member), 21
 mpi::BINARY_XOR (C++ member), 21
 mpi::Broadcast (C++ function), 25
 mpi::CartCreate (C++ function), 23
 mpi::CartSub (C++ function), 23
 mpi::Comm (C++ type), 19
 mpi::COMM_NULL (C++ member), 21
 mpi::COMM_SELF (C++ member), 21
 mpi::COMM_WORLD (C++ member), 21
 mpi::CommGroup (C++ function), 23
 mpi::Congruent (C++ function), 23
 mpi::Create (C++ function), 22, 23
 mpi::Datatype (C++ type), 20
 mpi::Difference (C++ function), 24
 mpi::Dup (C++ function), 23
 mpi::ErrorHandler (C++ type), 20
 mpi::ErrorHandlerSet (C++ function), 23
 mpi::ERRORS_ARE_FATAL (C++ member), 21

- mpi::ERRORS_RETURN (C++ member), 21
 mpi::Finalize (C++ function), 22
 mpi::Finalized (C++ function), 22
 mpi::Free (C++ function), 22–24
 mpi::Gather (C++ function), 25
 mpi::GetCount<T> (C++ function), 25
 mpi::Group (C++ type), 20
 mpi::GROUP_EMPTY (C++ member), 21
 mpi::GROUP_NULL (C++ member), 21
 mpi::Incl (C++ function), 24
 mpi::Initialize (C++ function), 22
 mpi::Initialized (C++ function), 22
 mpi::InitializeThread (C++ function), 22
 mpi::IProbe (C++ function), 24
 mpi::IRecv (C++ function), 25
 mpi::ISend (C++ function), 25
 mpi::ISSend (C++ function), 25
 mpi::LOGICAL_AND (C++ member), 21
 mpi::LOGICAL_OR (C++ member), 21
 mpi::LOGICAL_XOR (C++ member), 21
 mpi::MAX (C++ member), 21
 mpi::MIN (C++ member), 21
 mpi::MIN_COLL_MSG (C++ member), 21
 mpi::Op (C++ type), 20
 mpi::PROD (C++ member), 21
 mpi::Rank (C++ function), 23
 mpi::Recv (C++ function), 25
 mpi::Reduce (C++ function), 26
 mpi::ReduceScatter (C++ function), 26
 mpi::Request (C++ type), 20
 mpi::REQUEST_NULL (C++ member), 21
 mpi::Scatter (C++ function), 25
 mpi::Send (C++ function), 25
 mpi::SendRecv (C++ function), 25
 mpi::Size (C++ function), 23
 mpi::Split (C++ function), 23
 mpi::Status (C++ type), 20
 mpi::SUM (C++ member), 21
 mpi::Test (C++ function), 24
 mpi::THREAD_FUNNELED (C++ member), 20
 mpi::THREAD_MULTIPLE (C++ member), 20
 mpi::THREAD_SERIALIZED (C++ member), 20
 mpi::THREAD_SINGLE (C++ member), 20
 mpi::Time (C++ function), 22
 mpi::Translate (C++ function), 24
 mpi::UNDEFINED (C++ member), 20
 mpi::Union (C++ function), 24
 mpi::UserFunction (C++ type), 20
 mpi::Wait (C++ function), 24
 MultiShiftHessSolve (C++ function), 152
 MultiShiftQuasiTrsm (C++ function), 99
 MultiShiftTrsm (C++ function), 100
- ## N
- n (C++ member), 140
 NonHPDMatrixException (C++ class), 32
 NonHPDMatrixException::NonHPDMatrixException (C++ function), 32
 NonHPSDMatrixException (C++ class), 32
 NonHPSDMatrixException::NonHPSDMatrixException (C++ function), 32
 Norm (C++ function), 141
 NormalFromEVD (C++ function), 171
 NormalUniformSpectrum (C++ function), 176
 NormType (C++ enum), 35
 NormType::ENTRYWISE_ONE_NORM (C++ enumerator), 35
 NormType::FROBENIUS_NORM (C++ enumerator), 35
 NormType::INFINITY_NORM (C++ enumerator), 35
 NormType::MAX_NORM (C++ enumerator), 35
 NormType::NUCLEAR_NORM (C++ enumerator), 35
 NormType::ONE_NORM (C++ enumerator), 35
 NormType::TWO_NORM (C++ enumerator), 35
 Nrm2 (C++ function), 92
 NuclearNorm (C++ function), 142
 numNegative (C++ member), 149
 numPositive (C++ member), 149
 numZero (C++ member), 149
- ## O
- OneCondition (C++ function), 139
 OneNorm (C++ function), 143
 Ones (C++ function), 172
 OneTwoOne (C++ function), 172
 Op (C++ function), 20
 op (C++ member), 20
 operator
 = (C++ function), 20, 45
 operator== (C++ function), 19, 20, 45
 operator<< (C++ function), 32
 Orientation (C++ enum), 36
 Orientation::ADJOINT (C++ enumerator), 36
 Orientation::NORMAL (C++ enumerator), 36
 Orientation::TRANSPOSE (C++ enumerator), 36

P

Parter (C++ function), 172
 PartitionDown (C++ function), 73
 PartitionDownDiagonal (C++ function), 75
 PartitionDownOffsetDiagonal (C++ function), 76
 PartitionLeft (C++ function), 73
 PartitionRight (C++ function), 74
 PartitionUp (C++ function), 72
 PartitionUpDiagonal (C++ function), 74
 PartitionUpOffsetDiagonal (C++ function), 75
 Pei (C++ function), 172
 PermutationMeta (C++ function), 136
 PermutationMeta (C++ type), 136
 PermutationParity (C++ function), 136
 PermuteCols (C++ function), 135
 PermuteRows (C++ function), 135
 PivotParity (C++ function), 138
 PivotsToInversePermutation (C++ function), 138
 PivotsToPartialPermutation (C++ function), 138
 PivotsToPermutation (C++ function), 138
 Polar (C++ function), 33, 118
 polar::QDWH (C++ function), 118
 PopBlocksizeStack (C++ function), 31
 PopCallStack (C++ function), 31
 Pow (C++ function), 34
 power (C++ member), 134
 PowerOfTwo (C++ function), 37
 Print (C++ function), 178
 PrintCCompilerInfo (C++ function), 30
 PrintConfig (C++ function), 29
 PrintCxxCompilerInfo (C++ function), 30
 PrintVersion (C++ function), 29
 progress (C++ member), 134
 Pseudoinverse (C++ function), 132
 PseudospecCtrl<Base<F>> (C++ class), 145
 PseudospecCtrl<Real> (C++ class), 145
 PseudospecCtrl<Real>::arnoldi (C++ member), 145
 PseudospecCtrl<Real>::basisSize (C++ member), 145
 PseudospecCtrl<Real>::deflate (C++ member), 145
 PseudospecCtrl<Real>::forceComplexPs (C++ member), 145
 PseudospecCtrl<Real>::forceComplexSchur (C++ member), 145
 PseudospecCtrl<Real>::maxIts (C++ member), 145

PseudospecCtrl<Real>::progress (C++ member), 145
 PseudospecCtrl<Real>::reorthog (C++ member), 145
 PseudospecCtrl<Real>::sdcCtrl (C++ member), 145
 PseudospecCtrl<Real>::snapCtrl (C++ member), 145
 PseudospecCtrl<Real>::tol (C++ member), 145
 Pseudospectrum (C++ function), 145–147
 PushBlocksizeStack (C++ function), 31
 PushCallStack (C++ function), 31

Q

QR (C++ function), 125
 QR0 (C++ member), 127
 qr::ApplyQ (C++ function), 126
 qr::BusingerGolub (C++ function), 126, 127
 qr::Explicit (C++ function), 126
 qr::ExplicitTS (C++ function), 127
 qr::SolveAfter (C++ function), 127
 qr::TS (C++ function), 127
 qr::ts::FormQ (C++ function), 127
 qr::ts::FormR (C++ function), 127
 QRList (C++ member), 127
 QuadraticProgram (C++ function), 157
 QuasiDiagonalScale (C++ function), 94
 QuasiDiagonalSolve (C++ function), 94
 QuasiTriangularPseudospectrum (C++ function), 146–148
 QuasiTrsm (C++ function), 100
 QuasiTrsv (C++ function), 96, 97

R

Read (C++ function), 178
 RealHermitianFunction (C++ function), 132
 RealPart (C++ function), 33
 recvCounts (C++ member), 136
 recvDispls (C++ member), 136
 recvIdx (C++ member), 136
 recvRanks (C++ member), 136
 Redheffer (C++ function), 172
 reflector::Col (C++ function), 152
 reflector::Row (C++ function), 152
 RepartitionDown (C++ function), 77
 RepartitionDownDiagonal (C++ function), 80, 81
 RepartitionLeft (C++ function), 78
 RepartitionRight (C++ function), 78, 79
 RepartitionUp (C++ function), 77
 RepartitionUpDiagonal (C++ function), 79

- ReportException (C++ function), 30
rho (C++ member), 140
Ricatti (C++ function), 164
Riemann (C++ function), 173
RightHyperbolicReflector (C++ function), 153
RightReflector (C++ function), 152
Ris (C++ function), 173
RowSwap (C++ function), 93
RQ (C++ function), 128
rq::ApplyQ (C++ function), 129
rq::SolveAfter (C++ function), 129
- ## S
- SafeDeterminant (C++ function), 140
SafeHPDDeterminant (C++ function), 140
SafeProduct<F> (C++ type), 140
scalapack::HessenbergEig (C++ function), 29
scalapack::HessenbergSchur (C++ function), 29
Scale (C++ function), 92
ScaleDown (C++ function), 136
ScaleTrapezoid (C++ function), 92
ScaleUp (C++ function), 136
scaling (C++ member), 134
SchattenNorm (C++ function), 143
Schur (C++ function), 115
schur::CheckRealSchur (C++ function), 116
schur::QR (C++ function), 115
schur::QuasiTriangEig (C++ function), 116
schur::RealToComplex (C++ function), 116
schur::SDC (C++ function), 117
scomplex (C++ type), 32
SdcCtrl<Base<F>> (C++ type), 117
SdcCtrl<Real> (C++ class), 116
SdcCtrl<Real>::cutoff (C++ member), 116
SdcCtrl<Real>::maxInnerIts (C++ member), 117
SdcCtrl<Real>::maxOuterIts (C++ member), 117
SdcCtrl<Real>::progress (C++ member), 117
SdcCtrl<Real>::random (C++ member), 117
SdcCtrl<Real>::spreadFactor (C++ member), 117
SdcCtrl<Real>::tol (C++ member), 117
sendCounts (C++ member), 136
sendDispls (C++ member), 136
sendIdx (C++ member), 136
sendRanks (C++ member), 136
SetBlocksize (C++ function), 31
SetDefaultBlockHeight (C++ function), 31
SetDefaultBlockWidth (C++ function), 31
SetDiagonal (C++ function), 93
SetImagPart (C++ function), 33
SetLocalSymvBlocksize<T> (C++ function), 104
SetLocalTrr2kBlocksize<T> (C++ function), 104
SetLocalTrrkBlocksize<T> (C++ function), 104
SetRealPart (C++ function), 33
Shift (C++ function), 36
Sign (C++ function), 134
SignCtrl<Base<F>> (C++ type), 134
SignCtrl<Real> (C++ type), 134
SignScaling (C++ type), 134
Sin (C++ function), 33
SingularMatrixException (C++ class), 32
SingularMatrixException::SingularMatrixException (C++ function), 32
Sinh (C++ function), 33
Skeleton (C++ function), 130
SkewHermitianEig (C++ function), 111, 112
SlideLockedPartitionDown (C++ function), 82
SlideLockedPartitionLeft (C++ function), 83
SlideLockedPartitionRight (C++ function), 83
SlideLockedPartitionUp (C++ function), 81, 82
SlidePartitionDown (C++ function), 82
SlidePartitionLeft (C++ function), 83
SlidePartitionRight (C++ function), 83
SlidePartitionUp (C++ function), 81
SnapshotCtrl (C++ class), 144
SnapshotCtrl::imagSize (C++ member), 144
SnapshotCtrl::imgBase (C++ member), 145
SnapshotCtrl::imgDispCount (C++ member), 145
SnapshotCtrl::imgDispFreq (C++ member), 145
SnapshotCtrl::imgFormat (C++ member), 145
SnapshotCtrl::imgSaveCount (C++ member), 145
SnapshotCtrl::imgSaveFreq (C++ member), 144
SnapshotCtrl::Iterate (C++ function), 145
SnapshotCtrl::numBase (C++ member), 145
SnapshotCtrl::numFormat (C++ member), 145
SnapshotCtrl::numSaveCount (C++ member), 145
SnapshotCtrl::numSaveFreq (C++ member), 144
SnapshotCtrl::realSize (C++ member), 144

- SnapshotCtrl::ResetCounts (C++ function), 145
- SnapshotCtrl::SnapshotCtrl (C++ function), 145
- SoftThreshold (C++ function), 161
- Sort (C++ function), 153
- SortType (C++ enum), 35
- SortType::ASCENDING (C++ enumerator), 35
- SortType::DESCENDING (C++ enumerator), 35
- SortType::UNSORTED (C++ enumerator), 35
- SparseInvCov (C++ function), 158
- Spy (C++ function), 178
- Sqrt (C++ function), 33
- square_root::Newton (C++ function), 133
- SquareRoot (C++ function), 133
- SVD (C++ function), 119
- svd::Chan (C++ function), 119
- svd::DivideAndConquerSVD (C++ function), 119
- svd::GolubReinschUpper (C++ function), 119
- svd::QRSVD (C++ function), 119
- svd::Thresholded (C++ function), 119
- SVT (C++ function), 160
- svt::Cross (C++ function), 160
- svt::Normal (C++ function), 160
- svt::PivotedQR (C++ function), 160
- svt::TSQR (C++ function), 160
- Swap (C++ function), 93
- Sylvester (C++ function), 163
- Symm (C++ function), 101
- Symmetric2x2Scale (C++ function), 94
- Symmetric2x2Solve (C++ function), 94
- SymmetricEntrywiseNorm (C++ function), 141
- SymmetricEntrywiseOneNorm (C++ function), 142
- SymmetricFrobeniusNorm (C++ function), 142
- SymmetricInfinityNorm (C++ function), 142
- SymmetricInverse (C++ function), 131
- SymmetricKyFanNorm (C++ function), 142
- SymmetricMaxNorm (C++ function), 142
- SymmetricNorm (C++ function), 141
- SymmetricNuclearNorm (C++ function), 143
- SymmetricOneNorm (C++ function), 143
- SymmetricSchattenNorm (C++ function), 143
- SymmetricSolve (C++ function), 150
- SymmetricSwap (C++ function), 93
- SymmetricTwoNorm (C++ function), 143
- SymmetricTwoNormEstimate (C++ function), 144
- Symv (C++ function), 97
- Syr (C++ function), 97
- Syr2 (C++ function), 97
- Syr2k (C++ function), 101
- Syrk (C++ function), 101
- ## T
- t0 (C++ member), 127
- TaggedSort (C++ function), 153
- Tan (C++ function), 33
- tList (C++ member), 127
- Toeplitz (C++ function), 173
- tol (C++ member), 134
- TotalRecv (C++ function), 136
- TotalSend (C++ function), 136
- Trace (C++ function), 149
- Transpose (C++ function), 93
- Trdtrmm (C++ function), 103
- TreeData<F> (C++ type), 127
- TriangularInverse (C++ function), 131
- TriangularPseudospectrum (C++ function), 146–148
- TriW (C++ function), 173
- Trmm (C++ function), 101
- Trr2k (C++ function), 102
- Trrk (C++ function), 102
- Trsm (C++ function), 103
- Trstrm (C++ function), 103
- Trsv (C++ function), 97
- Trtrmm (C++ function), 102
- TwoCondition (C++ function), 139
- TwoNorm (C++ function), 143
- TwoNormEstimate (C++ function), 144
- TwoSidedTrmm (C++ function), 103
- TwoSidedTrsm (C++ function), 103, 104
- ## U
- Uniform (C++ function), 175
- UnitOrNonUnit (C++ enum), 36
- UnitOrNonUnit::NON_UNIT (C++ enumerator), 36
- UnitOrNonUnit::UNIT (C++ enumerator), 36
- UpdateDiagonal (C++ function), 95
- UpdateImagPart (C++ function), 33
- UpdateRealPart (C++ function), 33
- UpperClip (C++ function), 159
- UpperOrLower (C++ enum), 36
- UpperOrLower::LOWER (C++ enumerator), 36

UpperOrLower::UPPER (C++ enumerator),
36

V

VerticalOrHorizontal (C++ enum), 36

VerticalOrHorizontal::HORIZONTAL (C++
enumerator), 36

VerticalOrHorizontal::VERTICAL (C++ enu-
merator), 36

View (C++ function), 70

View1x2 (C++ function), 71

View2x1 (C++ function), 71

View2x2 (C++ function), 72

W

Walsh (C++ function), 173

Wigner (C++ function), 174

Wilkinson (C++ function), 174

Write (C++ function), 178

Z

Zero (C++ function), 93

ZeroNorm (C++ function), 143

Zeros (C++ function), 174