Elemental Manual

Release 0.78

Jack Poulson

March 07, 2013

CONTENTS

CHAPTER

ONE

INTRODUCTION

1.1 Overview

Elemental is a library for distributed-memory dense linear algebra that is essentially a careful combination of the following:

- A PLAPACK-like framework of matrix distributions that are trivial for users to redistribute between.
- A FLAME approach to tracking submatrices within (blocked) algorithms.
- Element-wise distribution of matrices. One of the major benefits to this approach is the much more convenient handling of submatrices, relative to block distribution schemes.

Just like ScaLAPACK and PLAPACK, Elemental's primary goal is in extending BLAS and LAPACK-like functionality into distributed-memory environments.

Though Elemental already contains high-quality implementations of a large portion of BLAS and LAPACK-like routines, there are a few important reasons why ScaLAPACK or PLAPACK might be more appropriate:

- Elemental does not yet support non-Hermitian eigenvalue problems, but ScaLAPACK does.
- Elemental does not yet provide routines for narrowly banded linear systems, though ScaLAPACK does (though you may want to consider the sparse-direct solver, Clique, which is built on top of Elemental.
- Some applications exploit the block distribution format used by ScaLAPACK and PLAPACK in order to increase the efficiency of matrix construction. Though it is clearly possible to redistribute the matrix into an element-wise distribution format after construction, this might add an unnecessary level of complexity.

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 exploited in order to simplify the class hierarchy.

1.2 Dependencies

- Functioning C++03 and ANSI C compilers.
- A working MPI implementation.
- BLAS and LAPACK (ideally version 3.3 or greater) implementations. If a sufficiently up-to-date LAPACK implementation is not provided, then a working F90 compiler is required in order to build Elemental's eigensolvers (the tridiagonal eigensolver, PMRRR, requires recent LAPACK routines).
- CMake (version 2.8.5 or later).

Elemental should successfully build on nearly every platform, as it has been verified to build on most major desktop platforms (including Linux, Mac OS X, Microsoft Windows, and Cygwin), as well as a wide variety of Linux clusters (including Blue Gene/P).

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) 2009-2013, Jack Poulson
All rights reserved.
```

For an up-to-date list of contributing authors, please see the AUTHORS file.

CHAPTER

TWO

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) as well as various versions of Microsoft Windows.

Elemental's main dependencies are

1. CMake (required)
2. MPI (required)
3. BLAS and LAPACK (required)
4. PMRRR (required for eigensolvers)
5. libFLAME (recommended for faster SVD's)

Each of these dependencies is discussed in detail below.

2.1 Dependencies

2.1.1 CMake

Elemental uses several new CMake modules, so it is important to ensure that version 2.8.5 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 12.04) 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, especialy for the BLAS. The most commonly used open source versions are ATLAS and OpenBLAS.

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 is available at:

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

Elemental builds a copy of PMRRR by default whenever possible: if an up-to-date non-MKL version of LAPACK is used, then PMRRR only requires a working MPI C compiler, otherwise, a Fortran 90 compiler is needed in order to build several recent LAPACK functions. If these LAPACK routines cannot be made available, then PMRRR is not built and Elemental's eigensolvers are automatically disabled.

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
```

2.2 Getting Elemental's source

There are two basic approaches:

1. Download a tarball of the most recent version from http://code.google.com/p/elemental/downloads/list. A new version is released roughly once a month, on average.
2. Install Mercurial and check out a copy of the repository by running

   ```
   hg clone https://elemental.googlecode.com/hg elemental
   ```

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.

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/elemvariables` can be used to build a simple Makefile:

```
include /usr/local/conf/elemvariables

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 2.8.5)
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 elemental-maint@googlegroups.com 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 elemental-user@googlegroups.com, and development questions to elemental-dev@googlegroups.com.

CHAPTER

THREE

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::`**`Axpy`** (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::Axpy`.

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>::type `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::`**`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::Axpy`, and $\alpha \in T$.

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::Axpy`, 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::Gemv`.

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 $A := \alpha xx^T + A$.

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 $A := \alpha(xy^T + yx^T) + A$.

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::`**`Hetrmm`**(char *uplo*, int *n*, T* *A*, int *lda*)
Form either $A := L^H L$ or $A := UU^H$, 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 `slauum`, `dlauum`, `clauum`, and `zlauum`; it 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 $C := \alpha \mathrm{op}(A) B$ or $C := \alpha B \mathrm{op}(A)$, 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 $C := \alpha \mathrm{op}(A)^{-1} B$ or $C := \alpha B \mathrm{op}(A)^{-1}$, 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*.

R `lapack::`**`MachineEpsilon<R>`**()

Return the relative machine precision.

R `lapack::`**`MachineSafeMin<R>`**()

Return the minimum number which can be inverted without underflow.

R `lapack::`**`MachinePrecision<R>`**()

Return the relative machine precision multiplied by the base.

R `lapack::`**`MachineUnderflowExponent<R>`**()

Return the minimum exponent before (gradual) underflow occurs.

R `lapack::`**`MachineUnderflowThreshold<R>`**()

Return the underflow threshold: `(base)^((underflow exponent)-1)`.

R `lapack::`**`MachineOverflowExponent<R>`**()

Return the largest exponent before overflow.

R `lapack::`**`MachineOverflowThreshold<R>`**()

Return the overflow threshold: `(1-rel. prec.)) * (base)^(overflow exponent)`.

Safe norms

R `lapack::`**`SafeNorm`**(R *alpha*, R *beta*)

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

R `lapack::`**`SafeNorm`**(R *alpha*, R *beta*, R *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 $(\phi, \gamma) \to (c, s, \rho)$ is "as continuous as possible", in the manner described by Kahan and Demmel's "On computing Givens rotations reliably and efficiently".

void `lapack::`**`ComputeGivens`** (R *phi*, R *gamma*, R* *c*, R* *s*, R* *rho*)
Computes a Givens rotation for real ϕ and γ.

void `lapack::`**`ComputeGivens`** (C *phi*, C *gamma*, R* *c*, C* *s*, C* *rho*)
Computes a Givens rotation for complex ϕ and γ.

QR-based SVD

void `lapack::`**`QRSVD`** (int *m*, int *n*, R* *A*, int *lda*, R* *s*, R* *U*, int *ldu*, R* *VTrans*, int *ldvt*)

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

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

void `lapack::`**`SingularValues`** (int *m*, int *n*, R* *A*, int *lda*, R* *s*)

void `lapack::`**`SingularValues`** (int *m*, int *n*, Complex<R>* *A*, int *lda*, R* *s*)

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

Divide-and-conquer SVD

void `lapack::`**`DivideAndConquerSVD`** (int *m*, int *n*, R* *A*, int *lda*, R* *s*, R* *U*, int *ldu*, R* *VTrans*, int *ldvt*)

void `lapack::`**`DivideAndConquerSVD`** (int *m*, int *n*, Complex<R>* *A*, int *lda*, R* *s*, Complex<R>* *U*, int *ldu*, Complex<R>* *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*, R* *d*, R* *e*, R* *VTrans*, int *ldvt*, R* *U*, int *ldu*)

void `lapack::`**`BidiagQRAlg`** (char *uplo*, int *n*, int *numColsVAdj*, int *numRowsU*, R* *d*, R* *e*, Complex<R>* *VAdj*, int *ldva*, Complex<R>* *U*, int *ldu*)

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

Hessenberg QR

void `lapack::`**`HessenbergEig`** (int *n*, R* *H*, int *ldh*, Complex<R>* *w*)

void `lapack::`**`HessenbergEig`** (int *n*, Complex<R>* *H*, int *ldh*, Complex<R>* *w*)

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

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`**
Equivalent to `MPI_Comm`.

type `mpi::`**`Datatype`**
Equivalent to `MPI_Datatype`.

type `mpi::`**`ErrorHandler`**
Equivalent to `MPI_Errhandler`.

type `mpi::`**`Group`**
Equivalent to `MPI_Group`.

type `mpi::`**`Op`**
Equivalent to `MPI_Op`.

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::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 whether or not MPI has been initialized.

bool `mpi::Finalized`()
: Return whether or not MPI has been finalized.

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

void `mpi::OpCreate`(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] op b[i], for i=0,...,length-1`, can be performed in an arbitrary order (for example, using a minimum spanning tree).

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

Communicator manipulation

int `mpi::CommRank`(mpi::Comm *comm*)
: Return our rank in the specified communicator.

int `mpi::CommSize`(mpi::Comm *comm*)
: Return the number of processes in the specified communicator.

void `mpi::CommCreate`(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::CommDup`(mpi::Comm *original*, mpi::Comm& *duplicate*)
: Create a copy of a communicator.

void `mpi::CommSplit`(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::CommFree`(mpi::Comm& *comm*)
: Free the specified communicator.

bool mpi::**CongruentComms**(mpi::Comm *comm1*, mpi::Comm *comm2*)
Return whether or not 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::**GroupRank**(mpi::Group *group*)
Return our rank in the specified group.

int mpi::**GroupSize**(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::**GroupIncl**(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::**GroupDifference**(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::**GroupFree**(mpI::Group& *group*)
Free the specified group.

void mpi::**GroupTranslateRanks**(mpi::Group *origGroup*, int *size*, const int* *origRanks*, mpi::Group *newGroup*, int* *newRanks*)
Return the ranks within *newGroup* of the *size* processes specified by their ranks in the *origGroup* communicator 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 whether or not the specified request has completed.

bool `mpi::IProbe` (int *source*, int *tag*, mpi::Comm *comm*, mpi::Status& *status*)
Return whether or not 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::ISSend` (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*recvCount, (i+1)*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*sendCount, (i+1)*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

$S_{p-1} + (S_{n-2} + \cdots (S_2 + (S_1 + S_0)) \cdots)$, 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 Parallel LCG

Since it is often necessary to generate a large matrix with pseudo-random entries in parallel, a method for ensuring that a large set of processes can each generate independent uniformly random samples is required. The purpose of Parallel LCG (PLCG) is to provide a provably independent generalization of a simple (but well-studied) Linear Congruential Generator. Knuth's constants from The Art of Computer Programming Vol. 2 are used.

The prototypes can be found in include/elemental/core/imports/plcg.hpp, while the implementations are in external/plcg/parallel_lcg.cpp.

Datatypes

type `plcg::`**`UInt32`**

Since the vast majority of modern systems make use of `unsigned` for storing 32-bit unsigned integers, we simply hardcode the type. If your system does not follow this convention, then this typedef will need to be changed!

type struct `plcg::`**`UInt64`**

A custom 64-bit unsigned integer which is simply the concatenation of two 32-bit unsigned integers (`UInt32`).

type struct `plcg::`**`ExpandedUInt64`**

A custom 64-bit unsigned integer which is stores each of the four 16-bit pieces within the first 16 bits of a 32-bit unsigned integer. This is done so that two such expanded 16-bit numbers can be multiplied without any chance of overflow.

LCG primitives

plcg::UInt32 `plcg::`**`Lower16Bits`**(plcg::UInt32 *a*)

Return the lower 16 bits of *a* in the lower 16 bits of the returned 32-bit unsigned integer.

plcg::UInt32 `plcg::Upper16Bits` (plcg::UInt32 *a*)
Return the upper 16 bits of a in the lower 16 bits of the returned 32-bit unsigned integer.

plcg::ExpandedUInt64 `plcg::Expand` (plcg::UInt32 *a*)
Expand a 32-bit unsigned integer into a 64-bit expanded representation.

plcg::ExpandedUInt64 `plcg::Expand` (plcg::UInt64 *a*)
Expand a 64-bit unsigned integer into a 64-bit expanded representation.

plcg::UInt64 `plcg::Deflate` (plcg::ExpandedUInt64 *a*)
Deflate an expanded 64-bit unsigned integer into the standard 64-bit form.

void `plcg::CarryUpper16Bits` (plcg::ExpandedUInt64& *a*)
Carry the results stored in the upper 16-bits of each of the four pieces into the next lower 16 bits.

plcg::ExpandedUInt64 `plcg::AddWith64BitMod` (plcg::ExpandedUInt64 *a*, plcg::ExpandedUInt64 *b*)
Return $a + b \mod 2^{64}$.

plcg::ExpandedUInt64 `plcg::MultiplyWith64BitMod` (plcg::ExpandedUInt64 *a*, plcg::ExpandedUInt64 *b*)
Return $ab \mod 2^{64}$.

plcg::ExpandedUInt64 `plcg::IntegerPowerWith64BitMod` (plcg::ExpandedUInt64 *x*, plcg::ExpandedUInt64 *n*)
Return $x^n \mod 2^{64}$.

void `plcg::Halve` (plcg::ExpandedUInt64& *a*)
$a := a/2$.

void `plcg::SeedSerialLcg` (plcg::UInt64 *globalSeed*)
Set the initial state of the serial Linear Congruential Generator.

void `plcg::SeedParallelLcg` (plcg::UInt32 *rank*, plcg::UInt32 *commSize*, plcg::UInt64 *globalSeed*)
Have our process seed a separate LCG meant for parallel computation, where the calling process has the given rank within a communicator of the specified size.

plcg::UInt64 `plcg::SerialLcg` ()
Return the current state of the serial LCG, and then advance to the next one.

plcg::UInt64 `plcg::ParallelLcg` ()
Return the current state of our process's portion of the parallel LCG, and then advance to our next local state.

void `plcg::ManualLcg` (plcg::ExpandedUInt64 *a*, plcg::ExpandedUInt64 *c*, plcg::ExpandedUInt64& *X*)
$X := aX + c \mod 2^{64}$.

Sampling

R `plcg::SerialUniform` ()
Return a uniform sample from $(0, 1]$ using the serial LCG.

R `plcg::ParallelUniform` ()
Return a uniform sample from $(0, 1]$ using the parallel LCG.

void `plcg::SerialBoxMuller` (R& *X*, R& *Y*)
Return two samples from a normal distribution with mean 0 and standard deviation of 1 using the serial LCG.

void `plcg::ParallelBoxMuller` (R& *X*, R& *Y*)
Return two samples from a normal distribution with mean 0 and standard deviation 1, but using the parallel LCG.

void `plcg::SerialGaussianRandomVariable` (R& *X*)
Return a single sample from a normal distribution with mean 0 and standard deviation 1 using the serial LCG.

void plcg::**ParallelGaussianRandomVariable**(R& *X*)
: Return a single sample from a normal distribution with mean 0 and standard deviation 1, but using the parallel LCG.

3.1.5 PMRRR

Rather than directly using Petschow 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

type struct pmrrr::**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.

type struct pmrrr::**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

pmrrr::Info pmrrr::**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*.

pmrrr::Info pmrrr::**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

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

pmrrr::Info pmrrr::**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.

pmrrr::Estimate pmrrr::**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

pmrrr::Info pmrrr::**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 $0 \leq a \leq b < n$.

pmrrr::Info pmrrr::**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.

3.1.6 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*, Complex<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*, Complex<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).

3.2 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 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.

3.2.2 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.

3.2.3 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 );
```

3.2.4 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.

3.2.5 Custom exceptions

type 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();
```

type 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();
```

type 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();
```

3.2.6 Complex data

type struct **Complex<R>**

type R **BaseType**

R **real**

The real part of the complex number

R **imag**

The imaginary part of the complex number

Complex()

This default constructor is a no-op.

Complex(R *a*)

Construction from a real value.

Complex(R *a*, R *b*)

Construction from a complex value.

Complex(const std::complex<R>& *alpha*)

Construction from an `std::complex<R>` instance.

Complex<R>& **operator=**(const R& *alpha*)

Assignment from a real value.

Complex<R>& **operator+=**(const R& *alpha*)

Increment with a real value.

Complex<R>& **operator-=**(const R& *alpha*)

Decrement with a real value.

Complex<R>& **operator*=**(const R& *alpha*)

Scale with a real value.

Complex<R>& **operator/=**(const R& *alpha*)

Divide with a real value.

Complex<R>& **operator=**(const Complex<R>& *alpha*)

Assignment from a complex value.

Complex<R>& **operator+=** (const Complex<R>& *alpha*)
Increment with a complex value.

Complex<R>& **operator-=** (const Complex<R>& *alpha*)
Decrement with a complex value.

Complex<R>& **operator*=** (const Complex<R>& *alpha*)
Scale with a complex value.

Complex<R>& **operator/=** (const Complex<R>& *alpha*)
Divide with a complex value.

type struct **Base<F>**

type type
The underlying real datatype of the (potentially complex) datatype *F*. For example, `typename Base<Complex<double> >::type` and `typename Base<double>::type` 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.

Complex<R> **operator+** (const Complex<R>& *alpha*, const Complex<R>& *beta*)
(complex,complex) addition.

Complex<R> **operator+** (const Complex<R>& *alpha*, const R& *beta*)
(complex,real) addition.

Complex<R> **operator+** (const R& *alpha*, const Complex<R>& *beta*)
(real,complex) addition.

Complex<R> **operator-** (const Complex<R>& *alpha*, const Complex<R>& *beta*)
(complex,complex) subtraction.

Complex<R> **operator-** (const Complex<R>& *alpha*, R& *beta*)
(complex,real) subtraction.

Complex<R> **operator-** (const R& *alpha*, const Complex<R>& *beta*)
(real,complex) subtraction.

Complex<R> **operator*** (const Complex<R>& *alpha*, const Complex<R>& *beta*)
(complex,complex) multiplication.

Complex<R> **operator*** (const Complex<R>& *alpha*, R& *beta*)
(complex,real) multiplication.

Complex<R> **operator*** (const R& *alpha*, const Complex<R>& *beta*)
(real,complex) multiplication.

Complex<R> **operator/** (const Complex<R>& *alpha*, const Complex<R>& *beta*)
(complex,complex) division.

Complex<R> **operator/** (const Complex<R>& *alpha*, const R& *beta*)
(complex,real) division.

Complex<R> **operator/** (const R& *alpha*, const Complex<R>& *beta*)
(real,complex) division.

Complex<R> **operator+** (const Complex<R>& *alpha*)
Returns *alpha*.

Complex<R> **operator-** (const Complex<R>& *alpha*)
Returns negative *alpha*.

bool **operator==** (const Complex<R>& *alpha*, const Complex<R>& *beta*)
    (complex,complex) equality check.

bool **operator==** (const Complex<R>& *alpha*, const R& *beta*)
    (complex,real) equality check.

bool **operator==** (const R& *alpha*, const Complex<R>& *beta*)
    (real,complex) equality check.

bool **operator!=** (const Complex<R>& *alpha*, const Complex<R>& *beta*)
    (complex,complex) inequality check.

bool **operator!=** (const Complex<R>& *alpha*, const R& *beta*)
    (complex,real) inequality check.

bool **operator!=** (const R& *alpha*, const Complex<R>& *beta*)
    (real,complex) inequality check.

std::ostream& **operator<<** (std::ostream& *os*, Complex<R> *alpha*)
    Pretty prints *alpha* in the form `a+bi`.

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

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

3.2.7 Scalar manipulation

typename Base<F>::type **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*)
    Return the real part of the real or complex variable α.

F **ImagPart** (const F& *alpha*)
    Return the imaginary part of the real or complex variable α.

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 or 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.

typename Base<F>::type **Arg** (const F& *alpha*)
Returns the argument of the real or complex variable α.

Complex<R> **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.

3.2.8 Other typedefs and enums

type byte
`typedef unsigned char byte;`

type enum **Conjugation**
An enum which can be set to either `CONJUGATED` or `UNCONJUGATED`.

type enum **Distribution**
An enum for specifying the distribution of a row or column of a distributed matrix:

- `MC`: Column of a standard matrix distribution
- `MD`: Diagonal of a standard matrix distribution
- `MR`: Row of a standard matrix distribution
- `VC`: Column-major vector distribution
- `VR`: Row-major vector distribution
- `STAR`: Redundantly stored

type enum **ForwardOrBackward**
An enum for specifying `FORWARD` or `BACKWARD`.

type enum **GridOrder**
An enum for specifying either a `ROW_MAJOR` or `COLUMN_MAJOR` ordering; it is used to tune 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()`.

type enum **LeftOrRight**
An enum for specifying `LEFT` or `RIGHT`.

type enum **NormType**
An enum that can be set to either

- `ONE_NORM`:

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

- INFINITY_NORM:

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

- MAX_NORM:

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

- NUCLEAR_NORM:

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

- FROBENIUS_NORM:

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

- TWO_NORM:

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

type enum **`Orientation`**
An enum for specifying whether a matrix, say A, should be implicitly treated as A (`NORMAL`), A^H (`ADJOINT`), or A^T (`TRANSPOSE`).

type enum **`UnitOrNonUnit`**
An enum for specifying either `UNIT` or `NON_UNIT`; typically used for stating whether or not a triangular matrix's diagonal is explicitly stored (`NON_UNIT`) or is implicitly unit-diagonal (`UNIT`).

type enum **`UpperOrLower`**
An enum for specifying `LOWER` or `UPPER` (triangular).

type enum **`VerticalOrHorizontal`**
An enum for specifying `VERTICAL` or `HORIZONTAL`.

3.2.9 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 **`LocalLength`** (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 **`LocalLength`** (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*.

3.3 The Matrix class

This is the basic building block of the library: its purpose it to provide convenient mechanisms for performing basic matrix manipulation operations, 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 );
```

The underlying data storage is simply a contiguous buffer that stores entries in a column-major fashion with an arbitrary leading dimension. 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 constant 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"
...
Matrix<Complex<double> > ABR( 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 );`.

type class **`Matrix<T>`**

The most general case, where the underlying datatype T is only assumed to be a ring; that is, it supports multiplication and addition and has the appropriate identities.

Constructors

`Matrix`()

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

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

Matrix (int *height*, int *width*, const T* *buffer*, int *ldim*)
A matrix is built around column-major constant buffer `const T* buffer` with the specified dimensions. The memory pointed to by *buffer* should not be freed until after the `Matrix<T>` object is destructed.

Matrix (int *height*, int *width*, T* *buffer*, int *ldim*)
A matrix is built around the column-major modifiable buffer `T* 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.

Basic information

int **Height** () const
Return the height of the matrix.

int **Width** () const
Return the width of the matrix.

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.

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.

T* **Buffer** ()
Return a pointer to the underlying buffer.

const T* **LockedBuffer** () const
Return a pointer to the underlying buffer that does not allow for modifying the data.

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

const T* **LockedBuffer** (int *i*, int *j*) const
Return a pointer to the portion of the buffer that holds entry (i, j) that does not allow for modifying the data.

I/O

void **Print** (const std::string *msg=""*) const

The matrix is printed to standard output (`std::cout`) with the preceding message *msg* (which is empty if unspecified).

void **Print** (std::ostream& *os*, const std::string *msg*=""") const
: The matrix is printed to the output stream *os* with the preceding message *msg* (which is empty if unspecified).

Entry manipulation

T **Get** (int *i*, int *j*) const
: Return entry (i, j).

void **Set** (int *i*, int *j*, T *alpha*)
: Set entry (i, j) to α.

void **Update** (int *i*, int *j*, T *alpha*)
: Add α to entry (i, j).

void **GetDiagonal** (Matrix<T>& *d*, int *offset=0*) const
: Modify d into a column-vector containing the entries 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 $+1$).

void **SetDiagonal** (const Matrix<T>& *d*, int *offset=0*)
: Set the entries in the *offset* diagonal entries from the contents of the column-vector d.

void **UpdateDiagonal** (const Matrix<T>& *d*, int *offset=0*)
: Add the contents of d onto the entries in the *offset* diagonal.

Note: Many of the following routines are only valid for complex datatypes.

---

typename Base<T>::type **GetRealPart** (int *i*, int *j*) const
: Return the real part of entry (i, j).

typename Base<T>::type **GetImagPart** (int *i*, int *j*) const
: Return the imaginary part of entry (i, j).

void **SetRealPart** (int *i*, int *j*, typename Base<T>::type *alpha*)
: Set the real part of entry (i, j) to α.

void **SetImagPart** (int *i*, int *j*, typename Base<T>::type *alpha*)
: Set the imaginary part of entry (i, j) to α.

void **UpdateRealPart** (int *i*, int *j*, typename Base<T>::type *alpha*)
: Add α to the real part of entry (i, j).

void **UpdateImagPart** (int *i*, int *j*, typename Base<T>::type *alpha*)
: Add α to the imaginary part of entry (i, j).

void **GetRealPartOfDiagonal** (Matrix<typename Base<T>::type>& *d*, int *offset=0*) const
: Modify d into a column-vector containing the real parts of the entries in the *offset* diagonal.

void **GetImagPartOfDiagonal** (Matrix<typename Base<T>::type>& *d*, int *offset=0*) const
: Modify d into a column-vector containing the imaginary parts of the entries in the *offset* diagonal.

void **SetRealPartOfDiagonal** (const Matrix<typename Base<T>::type>& *d*, int *offset=0*)
: Set the real parts of the entries in the *offset* diagonal from the contents of the column-vector d.

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

void **UpdateRealPartOfDiagonal** (const Matrix<typename Base<T>::type>& *d*, int *offset=0*)
: Add the contents of the column-vector d onto the real parts of the entries in the *offset* diagonal.

void **UpdateImagPartOfDiagonal** (const Matrix<typename Base<T>::type>& *d*, int *offset=0*)
Add the contents of the column-vector d onto the imaginary parts of the entries in the *offset* diagonal.

Views

bool **Viewing** () const
Return whether or not this matrix is currently viewing another matrix.

bool **LockedView** () const
Return whether or not we can modify the data we are viewing.

void **Attach** (int *height*, int *width*, T* *buffer*, int *ldim*)
Reconfigure the matrix around the specified buffer.

void **LockedAttach** (int *height*, int *width*, const T* *buffer*, int *ldim*)
Reconfigure the matrix around the specified unmodifiable buffer.

Utilities

const Matrix<T>& **operator=** (const Matrix<T>& *A*)
Create a copy of matrix A.

void **Empty** ()
Sets the matrix to 0×0 and frees the underlying buffer.

void **ResizeTo** (int *height*, int *width*)
Reconfigures the matrix to be *height* $\times$ *width*.

void **ResizeTo** (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 `std::min(height,1)`).

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>`.

type class **Matrix<R>**
Used to denote that the underlying datatype *R* is real.

type class **Matrix<Complex<R>>**
Used to denote that the underlying datatype `Complex<R>` is complex with base type *R*.

type class **Matrix<F>**
Used to denote that the underlying datatype *F* is a field.

3.4 The Grid class

This 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).

type class **Grid**

Grid(mpi::Comm *comm=mpi::COMM_WORLD*)
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.

Grid(mpi::Comm *comm*, int *height*, int *width*)
Construct a process grid over the specified communicator with the given dimensions. Note that the size of the communicator should be *height* $\times$ *width*.

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. The result is equivalent to the `VCRank()` function described below, but this interface is provided for simplicity.

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()` $\times$ `Width()`.

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.

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*)
: 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*. Elemental then chooses the grid dimensions. Most users should not call this routine, as this type of grid is only supported for a few DistMatrix types.

Grid(mpi::Comm *viewingComm*, mpi::Group *owningGroup*, int *height*, int *width*)
: This is the same as the previous routine, but the process grid dimensions are explicitly specified, and it is required that *height* $\times$ *width* equals the size of *owningGroup*. Most users should not call this routine, as it is only supported for a few DistMatrix types.

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 **InGrid**() const
: Return whether or not 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 tesselation 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 tesselation of the process grid.

int **DiagPathRank**() const
: Return our process's rank out of the set of processes lying in our diagonal of the tesselation 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 tesselation of the process grid.

Grid comparison functions

bool **operator==**(const Grid& *A*, const Grid& *B*)
: Returns whether or not !A! and !B! are the same process grid.

bool **operator!=**(const Grid& *A*, const Grid& *B*)
: Returns whether or not !A! and !B! are different process grids.

3.5 The DistMatrix class

The `DistMatrix<T,U,V>` class is meant to provide a distributed-memory analogue of the `Matrix<T>` class. Similarly to PLAPACK, roughly ten different matrix distributions are provided and it is trivial (in the programmability sense) to redistribute from one to another: in PLAPACK, one would simply call `PLA_Copy`, whereas, in Elemental, it is handled through overloading the $=$ operator.

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<T,U,V>` class.

Note: Since the `DistMatrix<T,U,V>` 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<T,U,V>`: `int`, `float`, `double`, `Complex<float>`, and `Complex<double>`.

---

type struct **DistData**

Distribution **colDist**

Distribution **rowDist**

int **colAlignment**

int **rowAlignment**

int **diagPath**

const Grid* **grid**

3.5.1 AbstractDistMatrix

This abstract class defines the list of member functions that are guaranteed to be available for all matrix distributions.

type class **AbstractDistMatrix<T>**
The most general case, where the underlying datatype *T* is only assumed to be a ring; that is, it supports multiplication and addition and has the appropriate identities.

Basic information

int **Height** () const
Return the height of the matrix.

int **Width** () const
Return the width of the matrix.

int **LocalHeight** () const
Return the local height of the matrix.

int **LocalWidth** () const
Return the local width of the matrix.

int **LDim** () const
Return the local leading dimension of the matrix.

size_t **AllocatedMemory** () const
Return the number of entries of type *T* that we have locally allocated space for.

const Grid& **Grid** () const
Return the grid that this distributed matrix is distributed over.

T* **Buffer** (int *iLocal=0*, int *jLocal=0*)
Return a pointer to the portion of the local buffer that stores entry *(iLocal,jLocal)*.

const T* **LockedBuffer** (int *iLocal=0*, int *jLocal=0*) const
Return a pointer to the portion of the local buffer that stores entry *(iLocal,jLocal)*, but do not allow for the data to be modified through the returned pointer.

Matrix<T>& **Matrix** ()
Return a reference to the local matrix.

const Matrix<T>& **LockedMatrix** () const
Return an unmodifiable reference to the local matrix.

I/O

void **Print** (const std::string *msg="“*) const
Print the distributed matrix to standard output (`std::cout`).

void **Print** (std::ostream& *os*, const std::string *msg*=”“) const
Print the distributed matrix to the output stream *os*.

void **Write** (const std::string *filename*, const std::string *msg*=”“) const
Print the distributed matrix to the file named *filename*.

Distribution details

void **FreeAlignments** ()
Free all alignment constaints.

bool **ConstrainedColAlignment** () const
Return whether or not the column alignment is constrained.

bool **ConstrainedRowAlignment** () const
Return whether or not the row alignment is constrained.

int **ColAlignment** () const
Return the alignment of the columns of the matrix.

int **RowAlignment** () const
Return the alignment of the rows of the matrix.

int **ColShift** () const
Return the first global row that our process owns.

int **RowShift** () const
Return the first global column that our process owns.

int **ColStride** () const
Return the number of rows between locally owned entries.

int **RowStride** () const
Return the number of columns between locally owned entries.

elem::DistData **DistData** () const
Returns a description of the distribution and alignment information

Entry manipulation

T **Get** (int *i*, int *j*) const
Return the *(i,j)* entry of the global matrix. This operation is collective.

void **Set** (int *i*, int *j*, T *alpha*)
Set the *(i,j)* entry of the global matrix to α. This operation is collective.

void **Update** (int *i*, int *j*, T *alpha*)
Add α to the *(i,j)* entry of the global matrix. This operation is collective.

T **GetLocal** (int *iLocal*, int *jLocal*) const
Return the *(iLocal,jLocal)* entry of our local matrix.

void **SetLocal** (int *iLocal*, int *jLocal*, T *alpha*)
Set the *(iLocal,jLocal)* entry of our local matrix to α.

void **UpdateLocal** (int *iLoca*, int *jLocal*, T *alpha*)
Add α to the *(iLocal,jLocal)* entry of our local matrix.

Note: Many of the following routines are only valid for complex datatypes.

typename Base<T>::type **GetRealPart** (int *i*, int *j*) const
Return the real part of the *(i,j)* entry of the global matrix. This operation is collective.

typename Base<T>::type **GetImagPart** (int *i*, int *j*) const
Return the imaginary part of the *(i,j)* entry of the global matrix. This operation is collective.

void **SetRealPart** (int *i*, int *j*, typename Base<T>::type *alpha*)
Set the real part of the *(i,j)* entry of the global matrix to α.

void **SetImagPart** (int *i*, int *j*, typename Base<T>::type *alpha*)
Set the imaginary part of the *(i,j)* entry of the global matrix to α.

void **UpdateRealPart** (int *i*, int *j*, typename Base<T>::type *alpha*)
Add α to the real part of the *(i,j)* entry of the global matrix.

void **UpdateImagPart** (int *i*, int *j*, typename Base<T>::type *alpha*)
Add α to the imaginary part of the *(i,j)* entry of the global matrix.

typename Base<T>::type **GetRealPartLocal** (int *iLocal*, int *jLocal*) const
Return the real part of the *(iLocal,jLocal)* entry of our local matrix.

typename Base<T>::type **GetLocalImagPart** (int *iLocal*, int *jLocal*) const
Return the imaginary part of the *(iLocal,jLocal)* entry of our local matrix.

void **SetLocalRealPart** (int *iLocal*, int *jLocal*, typename Base<T>::type *alpha*)
Set the real part of the *(iLocal,jLocal)* entry of our local matrix.

void **SetLocalImagPart** (int *iLocal*, int *jLocal*, typename Base<T>::type *alpha*)
Set the imaginary part of the *(iLocal,jLocal)* entry of our local matrix.

void **UpdateRealPartLocal** (int *iLocal*, int *jLocal*, typename Base<T>::type *alpha*)
Add α to the real part of the *(iLocal,jLocal)* entry of our local matrix.

void **UpdateLocalImagPart** (int *iLocal*, int *jLocal*, typename Base<T>::type *alpha*)
Add α to the imaginary part of the *(iLocal,jLocal)* entry of our local matrix.

Viewing

bool **Viewing** () const
Return whether or not this matrix is viewing another.

bool **LockedView** () const
Return whether or not this matrix is viewing another in a manner that does not allow for modifying the viewed data.

Utilities

void **Empty** ()
Resize the distributed matrix so that it is 0×0 and free all allocated storage.

void **ResizeTo** (int *height*, int *width*)
Reconfigure the matrix so that it is *height* $\times$ *width*.

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

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 `AbstractDistMatrix<T>`.

type class **`AbstractDistMatrix<R>`**
Used to denote that the underlying datatype *R* is real.

type class **`AbstractDistMatrix<Complex<R>>`**
Used to denote that the underlying datatype `Complex<R>` is complex with base type *R*.

type class **`AbstractDistMatrix<F>`**
Used to denote that the underlying datatype *F* is a field.

3.5.2 DistMatrix

type class **`DistMatrix<T, U, V>`**
This templated class for manipulating distributed matrices is only defined for the following choices of the column and row `Distribution`'s, *U* and *V* (*T* is a ring in this case).

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>`.

type class **`DistMatrix<double, U, V>`**
The underlying datatype is the set of double-precision real numbers.

type class **`DistMatrix<Complex<double>, U, V>`**
The underlying datatype is the set of double-precision complex numbers.

type class **`DistMatrix<R, U, V>`**
The underlying datatype *R* is real.

type class **`DistMatrix<Complex<R>, U, V>`**
The underlying datatype `Complex<R>` is complex with base type *R*.

type class **`DistMatrix<F, U, V>`**
The underlying datatype *F* is a field.

3.5.3 `[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>`.

type class **DistMatrix<T>**

type class **DistMatrix<T, MC, MR>**

The most general case, where the underlying datatype *T* is only assumed to be a ring.

Constructors

DistMatrix(const Grid& grid=DefaultGrid())

Create a 0×0 distributed matrix over the specified grid.

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

Create a *height* $\times$ *width* distributed matrix over the specified grid.

DistMatrix (int *height*, int *width*, int *colAlignment*, int *rowAlignment*, const Grid& *grid*)

Create a *height* $\times$ *width* distributed matrix distributed over the specified process grid, but with the top-left entry owned by the *colAlignment* process row and the *rowAlignment* process column.

DistMatrix (int *height*, int *width*, int *colAlignment*, int *rowAlignment*, int *ldim*, const Grid& *grid*)

Same as above, but the local leading dimension is also specified.

DistMatrix (int *height*, int *width*, int *colAlignment*, int *rowAlignment*, const T* *buffer*, int *ldim*, const Grid& *grid*)

View a constant distributed matrix's buffer; the buffer must correspond to the local portion of an elemental distributed matrix with the specified row and column alignments and leading dimension, *ldim*.

DistMatrix (int *height*, int *width*, int *colAlignment*, int *rowAlignment*, T* *buffer*, int *ldim*, const Grid& *grid*)

Same as above, but the contents of the matrix are modifiable.

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

Build a copy of the distributed matrix *A*, but force it to be in the `[MC,MR]` distribution.

Redistribution

const DistMatrix<T, MC, MR>& **operator=** (const DistMatrix<T, MC, MR>& *A*)

If this matrix can be properly aligned with *A*, then perform a local copy, otherwise perform an `mpi::SendRecv()` permutation first.

const DistMatrix<T, MC, MR>& **operator=** (const DistMatrix<T, MC, STAR>& *A*)

Perform a local (filtered) copy to form an `[MC,MR ]` distribution and then, if necessary, fix the alignment of the `MC` distribution via an `mpi::SendRecv()` within process columns.

const DistMatrix<T, MC, MR>& **operator=** (const DistMatrix<T, STAR, MR>& *A*)

Perform a local (filtered) copy to form an `[MC,MR ]` distribution and then, if necessary, fix the alignment of the `MR` distribution via an `mpi::SendRecv()` within process rows.

const DistMatrix<T, MC, MR>& **operator=**(const DistMatrix<T, MD, STAR>& *A*)
Since the `[MD,STAR]` distribution is defined such that its columns are distributed like a diagonal of an `[MC,MR]` distributed matrix, this operation is not very common.

Note: This redistribution routine is not yet implemented.

const DistMatrix<T, MC, MR>& **operator=**(const DistMatrix<T, STAR, MD>& *A*)

Note: This redistribution routine is not yet implemented.

const DistMatrix<T, MC, MR>& **operator=**(const DistMatrix<T, MR, MC>& *A*)
This routine serves to transpose the distribution of `A[MR,MC]` into the standard matrix distribution, `A[MC,MR]`. This redistribution is implemented with four different approaches: one for matrices that are taller than they are wide, one for matrices that are wider than they are tall, one for column vectors, and one for row vectors.

const DistMatrix<T, MC, MR>& **operator=**(const DistMatrix<T, MR, STAR>& *A*)
This is similar to the above routine, but with each row of *A* being undistributed, and only one approach is needed: $A[M_C,M_R] \leftarrow A[V_C,\star] \leftarrow A[V_R,\star] \leftarrow A[M_R,\star]$.

const DistMatrix<T, MC, MR>& **operator=**(const DistMatrix<T, STAR, MC>& *A*)
This routine is dual to the $A[M_C,M_R] \leftarrow A[M_R,\star]$ redistribution and is accomplished through the sequence: $A[M_C,M_R] \leftarrow A[\star,V_R] \leftarrow A[\star,V_C] \leftarrow A[\star,M_C]$.

const DistMatrix<T, MC, MR>& **operator=**(const DistMatrix<T, VC, STAR>& *A*)
Perform an `mpi::AllToAll()` within process rows in order to redistribute to the `[MC,MR]` distribution (an `mpi::SendRecv()` within process columns may be required for alignment).

const DistMatrix<T, MC, MR>& **operator=**(const DistMatrix<T, STAR, VC>& *A*)
Accomplished through the sequence $A[M_C,M_R] \leftarrow A[\star,V_R] \leftarrow A[\star,V_C]$.

const DistMatrix<T, MC, MR>& **operator=**(const DistMatrix<T, VR, STAR>& *A*)
Accomplished through the sequence $A[M_C,M_R] \leftarrow A[V_C,\star] \leftarrow A[V_R,\star]$.

const DistMatrix<T, MC, MR>& **operator=**(const DistMatrix<T, STAR, VR>& *A*)
Perform an `mpi::AllToAll()` within process columns in order to redistribute to the `[MC,MR]` distribution (an `mpi::SendRecv()` within process rows may be required for alignment).

const DistMatrix<T, MC, MR>& **operator=**(const DistMatrix<T, STAR, STAR>& *A*)
Perform an `mpi::AllGather()` over the entire grid in order to give every process a full copy of *A*.

Diagonal manipulation

void **GetDiagonal**(DistMatrix<T, MD, STAR>& *d*, int *offset=0*) const
The $[M_D,\star]$ distribution is defined such that its columns are distributed like diagonals of the standard matrix distribution, `[MC,MR]`. Thus, *d* can be formed locally if the distribution can be aligned with that of the *offset* diagonal of $A[M_C,M_R]$.

void **GetDiagonal**(DistMatrix<T, STAR, MD>& *d*, int *offset=0*) const
This is the same as above, but *d* is a row-vector instead of a column-vector.

void **SetDiagonal**(const DistMatrix<T, MD, STAR>& *d*, int *offset=0*)
Same as `DistMatrix<T>::GetDiagonal()`, but in reverse.

void **SetDiagonal**(const DistMatrix<T, STAR, MD>& *d*, int *offset=0*)
Same as `DistMatrix<T>::GetDiagonal()`, but in reverse.

Note: Many of the following routines are only valid for complex datatypes and are analogous to their general counterparts from above in the obvious manner.

void **GetRealPartOfDiagonal** (DistMatrix<typename Base<T>::type, MD, STAR>& *d*, int *offset=0*) const

void **GetImagPartOfDiagonal** (DistMatrix<typename Base<T>::type, MD, STAR>& *d*, int *offset=0*) const

void **GetRealPartOfDiagonal** (DistMatrix<typename Base<T>::type, STAR, MD>& *d*, int *offset=0*) const

void **GetImagPartOfDiagonal** (DistMatrix<typename Base<T>::type, STAR, MD>& *d*, int *offset=0*) const

void **SetRealPartOfDiagonal** (const DistMatrix<typename Base<T>::type, MD, STAR>& *d*, int *offset=0*)

void **SetImagPartOfDiagonal** (const DistMatrix<typename Base<T>::type, MD, STAR>& *d*, int *offset=0*)

void **SetRealPartOfDiagonal** (const DistMatrix<typename Base<T>::type, STAR, MD>& *d*, int *offset=0*)

void **SetImagPartOfDiagonal** (const DistMatrix<typename Base<T>::type, STAR, MD>& *d*, int *offset=0*)

Alignment

All of the following clear the distributed matrix's contents and then reconfigure the alignments as described.

void **AlignWith** (const AbstractDistMatrix<T>& *A*)
: Force the alignments to match those of *A*.

void **AlignWith** (const elem::DistData& *data*)
: A mechanism for aligning with a distributed matrix of a different datatype, via `AlignWith( A.DistData() );`

void **AlignColsWith** (const AbstractDistMatrix<T>& *A*)
: Force the column alignment to match that of *A*.

void **AlignColsWith** (const elem::DistData& *data*)
: A mechanism for aligning with a distributed matrix of a different datatype, via `AlignColsWith( A.DistData() );`

void **AlignRowsWith** (const AbstractDistMatrix<T>& *A*)
: Force the row alignment to match that of *A*.

void **AlignRowsWith** (const elem::DistData& *data*)
: A mechanism for aligning with a distributed matrix of a different datatype, via `AlignRowsWith( A.DistData() );`

Views

void **Attach** (int *height*, int *width*, int *colAlignment*, int *rowAlignment*, T* *buffer*, int *ldim*, const Grid& *grid*)
: Reconfigure this distributed matrix around an implicit `[MC,MR]` distributed matrix of the specified dimensions, alignments, local buffer, local leading dimension, and process grid.

void **LockedAttach** (int *height*, int *width*, int *colAlignment*, int *rowAlignment*, const T* *buffer*, int *ldim*, const Grid& *grid*)
Same as above, but the resulting matrix is "locked", meaning that it cannot modify the underlying local data.

Custom communication routines

The following routines primarily exist as a means of avoiding the poor memory bandwidth which results from packing or unpacking large amounts of data without a unit stride. PLAPACK noticed this issue and avoided the problem by carefully (conjugate-)transposing matrices in strategic places, usually before a gather or after a scatter, and we follow suit.

void **SumScatterFrom** (const DistMatrix<T, MC, STAR>& *A*)
Simultaneously sum $A[M_C, \star]$ within each process row and scatter the entries in each row to form the result in an $[M_C, M_R]$ distribution.

void **SumScatterUpdate** (T *alpha*, const DistMatrix<T, MC, STAR>& *A*)
Same as above, but add α times the result onto the parent distributed matrix rather than simply assigning the result to it.

void **SumScatterFrom** (const DistMatrix<T, STAR, MR>& *A*)
Simultaenously sum $A[\star, M_R]$ within each process column and scatter the entries in each column to form the result in an $[M_C, M_R]$ distribution.

void **SumScatterUpdate** (T *alpha*, const DistMatrix<T, STAR, MR>& *A*)
Same as above, but add α times the result onto the parent distributed matrix rather than simply assigning the result to it.

void **SumScatterFrom** (const DistMatrix<T, STAR, STAR>& *A*)
Simultaneously sum $A[\star, \star]$ over the entire process grid and scatter the entries in each row and column to form the result in an $[M_C, M_R]$ distribution.

void **SumScatterUpdate** (T *alpha*, const DistMatrix<T, STAR, STAR>& *A*)
Same as above, but add α times the result onto the parent distributed matrix rather than simply assigning the result to it.

void **AdjointFrom** (const DistMatrix<T, STAR, MC>& *A*)
Set the parent matrix equal to the redistributed adjoint of $A[\star, M_C]$; in particular, $(A[\star, M_C])^H = A^H[M_C, \star]$, so perform an $[M_C, M_R] \leftarrow [M_C, \star]$ redistribution on the adjoint of *A*, which typically just consists of locally copying (and conjugating) subsets of the data from $A[\star, M_C]$.

void **AdjointFrom** (const DistMatrix<T, MR, STAR>& *A*)
This routine is the dual of the above routine, and performs an $[M_C, M_R] \leftarrow [\star, M_R]$ redistribution on the adjoint of *A*.

void **TransposeFrom** (const DistMatrix<T, STAR, MC>& *A*, bool *conjugate=false*)
Same as the corresponding `DistMatrix<T>::AdjointFrom()`, but with no conjugation by default.

void **TransposeFrom** (const DistMatrix<T, MR, STAR>& *A*, bool *conjugate=false*)
Same as the corresponding `DistMatrix<T>::AdjointFrom()`, but with no conjugation by default.

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,MC,MR>` = `DistMatrix<T>`.

type class **DistMatrix<double>**

type class **DistMatrix<double, MC, MR>**
The underlying datatype is the set of double-precision real numbers.

type class **DistMatrix<Complex<double>>**

type class **DistMatrix<Complex<double>, MC, MR>**
The underlying datatype is the set of double-precision complex numbers.

type class **DistMatrix<R>**

type class **DistMatrix<R, MC, MR>**
The underlying datatype *R* is real.

type class **DistMatrix<Complex<R>>**

type class **DistMatrix<Complex<R>, MC, MR>**
The underlying datatype `Complex<R>` is complex with base type *R*.

type class **DistMatrix<F>**

type class **DistMatrix<F, MC, MR>**
The underlying datatype *F* is a field.

3.5.4 `[MC,* ]`

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):

$$\left(\begin{array}{ccccccc}
\{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} \\
\{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} \\
\{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} \\
\{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} \\
\{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} \\
\{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} & \{1,3,5\} \\
\{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\} & \{0,2,4\}
\end{array}\right)$$

type class **DistMatrix<T, MC, STAR>**
TODO: Add the member functions.

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,MC,STAR>`.

type class **DistMatrix<double, MC, STAR>**
The underlying datatype is the set of double-precision real numbers.

type class **DistMatrix<Complex<double>, MC, STAR>**
The underlying datatype is the set of double-precision complex numbers.

type class **DistMatrix<R, MC, STAR>**
The underlying datatype *R* is real.

type class **DistMatrix<Complex<R>, MC, STAR>**
The underlying datatype `Complex<R>` is complex with base type *R*.

type class **DistMatrix<F, MC, STAR>**
The underlying datatype *F* is a field.

3.5.5 [* ,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):

$$\begin{pmatrix} \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} \\ \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} \\ \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} \\ \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} \\ \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} \\ \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} \\ \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} & \{2,3\} & \{4,5\} & \{0,1\} \end{pmatrix}$$

type class **DistMatrix<T, STAR, MR>**
    TODO: Add the member functions.

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,STAR,MR>`.

type class **DistMatrix<double, STAR, MR>**
    The underlying datatype is the set of double-precision real numbers.

type class **DistMatrix<Complex<double>, STAR, MR>**
    The underlying datatype is the set of double-precision complex numbers.

type class **DistMatrix<R, STAR, MR>**
    The underlying datatype *R* is real.

type class **DistMatrix<Complex<R>, STAR, MR>**
    The underlying datatype `Complex<R>` is complex with base type *R*.

type class **DistMatrix<F, STAR, MR>**
    The underlying datatype *F* is a field.

3.5.6 [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):

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 2 & 3 & 2 & 3 & 2 & 3 & 2 \\ 4 & 5 & 4 & 5 & 4 & 5 & 4 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 2 & 3 & 2 & 3 & 2 & 3 & 2 \\ 4 & 5 & 4 & 5 & 4 & 5 & 4 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

type class **DistMatrix<T, MR, MC>**
    TODO: Add the member functions.

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,MR,MC>`.

type class **`DistMatrix<double, MR, MC>`**
The underlying datatype is the set of double-precision real numbers.

type class **`DistMatrix<Complex<double>, MR, MC>`**
The underlying datatype is the set of double-precision complex numbers.

type class **`DistMatrix<R, MR, MC>`**
The underlying datatype *R* is real.

type class **`DistMatrix<Complex<R>, MR, MC>`**
The underlying datatype `Complex<R>` is complex with base type *R*.

type class **`DistMatrix<F, MR, MC>`**
The underlying datatype *F* is a field.

3.5.7 `[MR,* ]`

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):

$$\begin{pmatrix}
\{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} \\
\{2,3\} & \{2,3\} & \{2,3\} & \{2,3\} & \{2,3\} & \{2,3\} & \{2,3\} \\
\{4,5\} & \{4,5\} & \{4,5\} & \{4,5\} & \{4,5\} & \{4,5\} & \{4,5\} \\
\{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} \\
\{2,3\} & \{2,3\} & \{2,3\} & \{2,3\} & \{2,3\} & \{2,3\} & \{2,3\} \\
\{4,5\} & \{4,5\} & \{4,5\} & \{4,5\} & \{4,5\} & \{4,5\} & \{4,5\} \\
\{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\} & \{0,1\}
\end{pmatrix}$$

type class **`DistMatrix<T, MR, STAR>`**
TODO: Add the member functions.

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,MR,STAR>`.

type class **`DistMatrix<double, MR, STAR>`**
The underlying datatype is the set of double-precision real numbers.

type class **`DistMatrix<Complex<double>, MR, STAR>`**
The underlying datatype is the set of double-precision complex numbers.

type class **`DistMatrix<R, MR, STAR>`**
The underlying datatype *R* is real.

type class **`DistMatrix<Complex<R>, MR, STAR>`**
The underlying datatype `Complex<R>` is complex with base type *R*.

type class **`DistMatrix<F, MR, STAR>`**
The underlying datatype *F* is a field.

3.5.8 `[* ,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):

$$\begin{pmatrix} \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} \\ \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} \\ \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} \\ \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} \\ \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} \\ \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} \\ \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} & \{1,3,5\} & \{0,2,4\} \end{pmatrix}$$

type class **`DistMatrix<T, STAR, MC>`**
    TODO: Add the member functions.

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,STAR,MC>`.

type class **`DistMatrix<double, STAR, MC>`**
    The underlying datatype is the set of double-precision real numbers.

type class **`DistMatrix<Complex<double>, STAR, MC>`**
    The underlying datatype is the set of double-precision complex numbers.

type class **`DistMatrix<R, STAR, MC>`**
    The underlying datatype *R* is real.

type class **`DistMatrix<Complex<R>, STAR, MC>`**
    The underlying datatype `Complex<R>` is complex with base type *R*.

type class **`DistMatrix<F, STAR, MC>`**
    The underlying datatype *F* is a field.

3.5.9 `[MD,* ]`

TODO, but not as high of a priority since the $[M_D, \star]$ distribution is not as crucial for end users as many other details that have not yet been documented.

type class **`DistMatrix<T, MD, STAR>`**
    TODO: Add the member functions.

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,MD,STAR>`.

type class **`DistMatrix<double, MD, STAR>`**
    The underlying datatype is the set of double-precision real numbers.

type class **`DistMatrix<Complex<double>, MD, STAR>`**
    The underlying datatype is the set of double-precision complex numbers.

type class **DistMatrix<R, MD, STAR>**
The underlying datatype *R* is real.

type class **DistMatrix<Complex<R>, MD, STAR>**
The underlying datatype `Complex<R>` is complex with base type *R*.

type class **DistMatrix<F, MD, STAR>**
The underlying datatype *F* is a field.

3.5.10 [* ,MD]

TODO, but not as high of a priority since the $[\star, M_D]$ distribution is not as crucial for end users as many other details that have not yet been documented.

type class **DistMatrix<T, STAR, MD>**
TODO: Add the member functions.

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,STAR,MD>`.

type class **DistMatrix<double, STAR, MD>**
The underlying datatype is the set of double-precision real numbers.

type class **DistMatrix<Complex<double>, STAR, MD>**
The underlying datatype is the set of double-precision complex numbers.

type class **DistMatrix<R, STAR, MD>**
The underlying datatype *R* is real.

type class **DistMatrix<Complex<R>, STAR, MD>**
The underlying datatype `Complex<R>` is complex with base type *R*.

type class **DistMatrix<F, STAR, MD>**
The underlying datatype *F* is a field.

3.5.11 [VC,*]

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:

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

type class **DistMatrix<T, VC, STAR>**
TODO: Add the member functions.

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,VC,STAR>`.

type class **`DistMatrix<double, VC, STAR>`**
The underlying datatype is the set of double-precision real numbers.

type class **`DistMatrix<Complex<double>, VC, STAR>`**
The underlying datatype is the set of double-precision complex numbers.

type class **`DistMatrix<R, VC, STAR>`**
The underlying datatype *R* is real.

type class **`DistMatrix<Complex<R>, VC, STAR>`**
The underlying datatype `Complex<R>` is complex with base type *R*.

type class **`DistMatrix<F, VC, STAR>`**
The underlying datatype *F* is a field.

3.5.12 `[* ,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:

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 \\ 0 & 1 & 2 & 3 & 4 & 5 & 0 \end{pmatrix}$$

type class **`DistMatrix<T, STAR, VC>`**
TODO: Add the member functions.

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,STAR,VC>`.

type class **`DistMatrix<double, STAR, VC>`**
The underlying datatype is the set of double-precision real numbers.

type class **`DistMatrix<Complex<double>, STAR, VC>`**
The underlying datatype is the set of double-precision complex numbers.

type class **`DistMatrix<R, STAR, VC>`**
The underlying datatype *R* is real.

type class **`DistMatrix<Complex<R>, STAR, VC>`**
The underlying datatype `Complex<R>` is complex with base type *R*.

type class **`DistMatrix<F, STAR, VC>`**
The underlying datatype *F* is a field.

3.5.13 `[VR,* ]`

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:

$$\left(\begin{array}{ccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 2 & 2 & 2 & 2 & 2 & 2 \\ 4 & 4 & 4 & 4 & 4 & 4 & 4 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 3 & 3 & 3 & 3 & 3 & 3 & 3 \\ 5 & 5 & 5 & 5 & 5 & 5 & 5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

type class **`DistMatrix<T, VR, STAR>`**
TODO: Add the member functions.

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,VR,STAR>`.

type class **`DistMatrix<double, VR, STAR>`**
The underlying datatype is the set of double-precision real numbers.

type class **`DistMatrix<Complex<double>, VR, STAR>`**
The underlying datatype is the set of double-precision complex numbers.

type class **`DistMatrix<R, VR, STAR>`**
The underlying datatype *R* is real.

type class **`DistMatrix<Complex<R>, VR, STAR>`**
The underlying datatype `Complex<R>` is complex with base type *R*.

type class **`DistMatrix<F, VR, STAR>`**
The underlying datatype *F* is a field.

3.5.14 `[* ,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:

$$\left(\begin{array}{ccccccc} 0 & 2 & 4 & 1 & 3 & 5 & 0 \\ 0 & 2 & 4 & 1 & 3 & 5 & 0 \\ 0 & 2 & 4 & 1 & 3 & 5 & 0 \\ 0 & 2 & 4 & 1 & 3 & 5 & 0 \\ 0 & 2 & 4 & 1 & 3 & 5 & 0 \\ 0 & 2 & 4 & 1 & 3 & 5 & 0 \\ 0 & 2 & 4 & 1 & 3 & 5 & 0 \end{array}\right)$$

type class **`DistMatrix<T, STAR, VR>`**
TODO: Add the member functions.

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,STAR,VR>`.

type class **`DistMatrix<double, STAR, VR>`**
: The underlying datatype is the set of double-precision real numbers.

type class **`DistMatrix<Complex<double>, STAR, VR>`**
: The underlying datatype is the set of double-precision complex numbers.

type class **`DistMatrix<R, STAR, VR>`**
: The underlying datatype *R* is real.

type class **`DistMatrix<Complex<R>, STAR, VR>`**
: The underlying datatype `Complex<R>` is complex with base type *R*.

type class **`DistMatrix<F, STAR, VR>`**
: The underlying datatype *F* is a field.

3.5.15 `[* ,* ]`

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:

$$\left(\begin{array}{ccccccc}
\{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} \\
\{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} \\
\{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} \\
\{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} \\
\{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} \\
\{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} \\
\{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\} & \{0,1,...,5\}
\end{array}\right)$$

type class **`DistMatrix<T, STAR, STAR>`**
: **TODO:** Add the member functions.

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,STAR,STAR>`.

type class **`DistMatrix<double, STAR, STAR>`**
: The underlying datatype is the set of double-precision real numbers.

type class **`DistMatrix<Complex<double>, STAR, STAR>`**
: The underlying datatype is the set of double-precision complex numbers.

type class **`DistMatrix<R, STAR, STAR>`**
: The underlying datatype *R* is real.

type class **`DistMatrix<Complex<R>, STAR, STAR>`**
: The underlying datatype `Complex<R>` is complex with base type *R*.

type class **`DistMatrix<F, STAR, STAR>`**
: The underlying datatype *F* is a field.

3.6 Viewing

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*.

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*.

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 x width* submatrix of *B* starting at coordinate `(i,j)`.

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 x width* submatrix of *B* starting at coordinate `(i,j)`.

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 $\begin{pmatrix} B_L & B_R \end{pmatrix}$.

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 $\begin{pmatrix} B_L & B_R \end{pmatrix}$.

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 $\begin{pmatrix} B_T \\ B_B \end{pmatrix}$.

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 $\begin{pmatrix} B_T \\ B_B \end{pmatrix}$.

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 $\left(\begin{array}{cc} B_{TL} & B_{TR} \\ B_{BB} & B_{BR} \end{array}\right)$.

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 $\left(\begin{array}{cc} B_{TL} & B_{TR} \\ B_{BB} & B_{BR} \end{array}\right)$.

3.7 Partitioning

The following routines are slight tweaks of the FLAME project's (as well as PLAPACK'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 *AT* and *AB* to view the local data of *A* corresponding to the partition

$$A = \left(\begin{array}{c} A_T \\ A_B \end{array}\right),$$

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=Bloc

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

void LockedPartitionUp(const DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& AT, DistMatrix<T,U,V

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

$$A = \left(\begin{array}{c} A_T \\ A_B \end{array}\right),$$

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=Bl

Templated over the datatype, *T*, of the serial matrix *A*.

void PartitionDown(DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& AT, DistMatrix<T,U,V>& AB, int

`void LockedPartitionDown( const DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& AT, DistMatrix<T,U`
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

$$A = \begin{pmatrix} A_L & A_R \end{pmatrix},$$

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=Blo`
Templated over the datatype, *T*, of the serial matrix *A*.

`void PartitionLeft( DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& AL, DistMatrix<T,U,V>& AR, int`

`void LockedPartitionLeft( const DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& AL, DistMatrix<T,U`
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

$$A = \begin{pmatrix} A_L & A_R \end{pmatrix},$$

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=Bl`
Templated over the datatype, *T*, of the serial matrix *A*.

`void PartitionRight( DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& AL, DistMatrix<T,U,V>& AR, in`

`void LockedPartitionRight( const DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& AL, DistMatrix<T,`
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, Matrix<T>& ABL, Mat`

`void LockedPartitionUpDiagonal( const Matrix<T>& A, Matrix<T>& ATL, Matrix<T>& ATR, Matrix<`
Templated over the datatype, *T*, of the serial matrix *A*.

`void PartitionUpDiagonal( DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& ATL, DistMatrix<T,U,V>&`

`void LockedPartitionUpDiagonal( const DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& ATL, DistMat`
Templated over the datatype, *T*, and the distribution scheme, *(U,V)*, of the distributed matrix *A*.

3.7.6 PartitionUpLeftDiagonal

Same as `PartitionUpDiagonal`.

```
void PartitionUpLeftDiagonal( Matrix<T>& A, Matrix<T>& ATL, Matrix<T>& ATR, Matrix<T>& ABL,
```

```
void LockedPartitionUpLeftDiagonal( const Matrix<T>& A, Matrix<T>& ATL, Matrix<T>& ATR, Mat
```

Templated over the datatype, *T*, of the serial matrix *A*.

```
void PartitionUpLeftDiagonal( DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& ATL, DistMatrix<T,U,
```

```
void LockedPartitionUpLeftDiagonal( const DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& ATL, Dis
```

Templated over the datatype, *T*, and the distribution scheme, *(U,V)*, of the distributed matrix *A*.

3.7.7 PartitionUpRightDiagonal

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 *right* diagonal of A, which is defined to include the bottom-right entry of A; the length of the diagonal of A_{BR} is specified as a parameter in all of the following routines.

```
void PartitionUpRightDiagonal( Matrix<T>& A, Matrix<T>& ATL, Matrix<T>& ATR, Matrix<T>& AB
```

```
void LockedPartitionUpRightDiagonal( const Matrix<T>& A, Matrix<T>& ATL, Matrix<T>& ATR, Ma
```

Templated over the datatype, *T*, of the serial matrix *A*.

```
void PartitionUpRightDiagonal( DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& ATL, DistMatrix<T,U
```

```
void LockedPartitionUpRightDiagonal( const DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& ATL, Di
```

Templated over the datatype, *T*, and the distribution scheme, *(U,V)*, of the distributed matrix *A*.

3.7.8 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, M
```

```
void LockedPartitionDownDiagonal( const Matrix<T>& A, Matrix<T>& ATL, Matrix<T>& ATR, Matri
```

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>
```

```
void LockedPartitionDownDiagonal( const DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& ATL, DistM
```

Templated over the datatype, *T*, and the distribution scheme, *(U,V)*, of the distributed matrix *A*.

3.7.9 PartitionDownLeftDiagonal

Same as `PartitionDownDiagonal`.

`void PartitionDownLeftDiagonal( Matrix<T>& A, Matrix<T>& ATL, Matrix<T>& ATR, Matrix<T>& AB`

`void LockedPartitionDownLeftDiagonal( const Matrix<T>& A, Matrix<T>& ATL, Matrix<T>& ATR, M`

Templated over the datatype, *T*, of the serial matrix *A*.

`void PartitionDownLeftDiagonal( DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& ATL, DistMatrix<T,`

`void LockedPartitionDownLeftDiagonal( const DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& ATL, D`

Templated over the datatype, *T*, and the distribution scheme, *(U,V)*, of the distributed matrix *A*.

3.7.10 PartitionDownRightDiagonal

Given an $m \times n$ matrix *A*, configure *ATL*, *ATR*, *ABL*, and *ABR* to view the local data 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 *right* diagonal of A, which includes the bottom-right entry of A.

`void PartitionDownLeftDiagonal( Matrix<T>& A, Matrix<T>& ATL, Matrix<T>& ATR, Matrix<T>& AB`

`void LockedPartitionDownLeftDiagonal( const Matrix<T>& A, Matrix<T>& ATL, Matrix<T>& ATR, M`

Templated over the datatype, *T*, of the serial matrix *A*.

`void PartitionDownLeftDiagonal( DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& ATL, DistMatrix<T,`

`void LockedPartitionDownLeftDiagonal( const DistMatrix<T,U,V>& A, DistMatrix<T,U,V>& ATL, D`

Templated over the datatype, *T*, and the distribution scheme, *(U,V)*, of the distributed matrix *A*.

3.8 Repartitioning

3.8.1 RepartitionUp

Given the partition

$$A = \left(\frac{A_T}{A_B}\right),$$

and a blocksize, n_b, turn the two-way partition into the three-way partition

$$\left(\frac{A_T}{A_B}\right) = \begin{pmatrix} A_0 \\ A_1 \\ \hline A_2 \end{pmatrix},$$

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>&`

`void LockedRepartitionUp( const Matrix<T>& AT, Matrix<T>& A0, Matrix<T>& A1, const Matrix<T`

Templated over the datatype, *T*.

`void RepartitionUp( DistMatrix<T,U,V>& AT, DistMatrix<T,U,V>& A0, DistMatrix<T,U,V>& A1, Di`

void LockedRepartitionUp(const DistMatrix<T,U,V>& AT, DistMatrix<T,U,V>& A0, DistMatrix<T,

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

$$A = \left(\frac{A_T}{A_B} \right),$$

and a blocksize, n_b, turn the two-way partition into the three-way partition

$$\left(\frac{A_T}{A_B} \right) = \left(\begin{array}{c} A_0 \\ \hline A_1 \\ A_2 \end{array} \right),$$

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>

void LockedRepartitionDown(const Matrix<T>& AT, Matrix<T>& A0, Matrix<T>& A1, const Matrix

Templated over the datatype, *T*.

void RepartitionDown(DistMatrix<T,U,V>& AT, DistMatrix<T,U,V>& A0, DistMatrix<T,U,V>& A1,

void LockedRepartitionDown(const DistMatrix<T,U,V>& AT, DistMatrix<T,U,V>& A0, DistMatrix

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

$$A = \left(\begin{array}{c|c} A_L & A_R \end{array} \right),$$

and a blocksize, n_b, turn the two-way partition into the three-way partition

$$\left(\begin{array}{c|c} A_L & A_R \end{array} \right) = \left(\begin{array}{cc|c} A_0 & A_1 & A_2 \end{array} \right),$$

where A_1 is of width n_b and $A_2 = A_R$.

void RepartitionLeft(Matrix<T>& AL, Matrix<T>& AR, Matrix<T>& A0, Matrix<T>& A1, Matrix<T>

void LockedRepartitionLeft(const Matrix<T>& AL, const Matrix<T>& AR, Matrix<T>& A0, Matrix

Templated over the datatype, *T*.

void RepartitionLeft(DistMatrix<T,U,V>& AL, DistMatrix<T,U,V>& AR, DistMatrix<T,U,V>& A0,

void LockedRepartitionLeft(const DistMatrix<T,U,V>& AL, const DistMatrix<T,U,V>& AR, DistM

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

$$A = \left(\begin{array}{c|c} A_L & A_R \end{array} \right),$$

and a blocksize, n_b, turn the two-way partition into the three-way partition

$$\left(\begin{array}{c|c} A_L & A_R \end{array} \right) = \left(\begin{array}{c|cc} A_0 & A_1 & A_2 \end{array} \right),$$

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

void LockedRepartitionRight(const Matrix<T>& AL, const Matrix<T>& AR, Matrix<T>& A0, Matri

Templated over the datatype, *T*.

void RepartitionRight(DistMatrix<T,U,V>& AL, DistMatrix<T,U,V>& AR, DistMatrix<T,U,V>& A0,

void LockedRepartitionRight(const DistMatrix<T,U,V>& AL, const DistMatrix<T,U,V>& AR, Dist

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,

void LockedRepartitionUpDiagonal(const Matrix<T>& ATL, const Matrix<T>& ATR, Matrix<T>& A0

Templated over the datatype, *T*.

void RepartitionUpDiagonal(DistMatrix<T,U,V>& ATL, DistMatrix<T,U,V>& ATR, DistMatrix<T,U,

void LockedRepartitionUpDiagonal(const DistMatrix<T,U,V>& ATL, const DistMatrix<T,U,V>& AT

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>& A0

void LockedRepartitionDownDiagonal(const Matrix<T>& ATL, const Matrix<T>& ATR, Matrix<T>&
Templated over the datatype, *T*.

void RepartitionDownDiagonal(DistMatrix<T,U,V>& ATL, DistMatrix<T,U,V>& ATR, DistMatrix<T,

void LockedRepartitionDownDiagonal(const DistMatrix<T,U,V>& ATL, const DistMatrix<T,U,V>&
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 Sliding partitions

3.9.1 SlidePartitionUp

Simultaneously slide and merge the partition

$$A = \left(\begin{array}{c} A_0 \\ A_1 \\ \hline A_2 \end{array}\right),$$

into

$$\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) = \left(\begin{array}{c} A_0 \\ \hline A_1 \\ A_2 \end{array}\right).$$

void **SlidePartitionUp**(Matrix<T>& *AT*, Matrix<T>& *A0*, Matrix<T>& *A1*, Matrix<T>& *AB*, Matrix<T>& *A2*)

void **SlideLockedPartitionUp**(Matrix<T>& *AT*, const Matrix<T>& *A0*, const Matrix<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*, DistMatrix<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

$$A = \left(\begin{array}{c} A_0 \\ \hline A_1 \\ A_2 \end{array}\right),$$

into

$$\left(\begin{array}{c} A_T \\ \hline A_B \end{array}\right) = \left(\begin{array}{c} A_0 \\ A_1 \\ \hline A_2 \end{array}\right).$$

void **SlidePartitionDown**(Matrix<T>& *AT*, Matrix<T>& *A0*, Matrix<T>& *A1*, Matrix<T>& *AB*, Matrix<T>& *A2*)

void **SlideLockedPartitionDown**(Matrix<T>& *AT*, const Matrix<T>& *A0*, const Matrix<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*, DistMatrix<T, U, V>& *A1*, DistMatrix<T, U, V>& *AB*, DistMatrix<T, U, V>& *A2*)

void **SlideLockedPartitionDown**(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:

```
SlidePartitionDown( AT,  A0,
                         A1,
                   /**/ /**/
                    AB,  A2 );
```

3.9.3 SlidePartitionLeft

Simultaneously slide and merge the partition

$$A = \left(\begin{array}{cc|c} A_0 & A_1 & A_2 \end{array} \right)$$

into

$$\left(\begin{array}{c|c} A_L & A_R \end{array} \right) = \left(\begin{array}{c|cc} A_0 & A_1 & A_2 \end{array} \right).$$

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,
                    A0, /**/ A1, A2 );
```

3.9.4 SlidePartitionRight

Simultaneously slide and merge the partition

$$A = \left(\begin{array}{c|cc} A_0 & A_1 & A_2 \end{array} \right)$$

into

$$\left(\begin{array}{c|c} A_L & A_R \end{array} \right) = \left(\begin{array}{cc|c} A_0 & A_1 & A_2 \end{array} \right).$$

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

$$A = \left(\begin{array}{cc|c} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$$

into

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|cc} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{array}\right).$$

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

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{cc|c} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right).$$

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 Axpy interface

The Axpy interface is Elemental's version of the PLAPACK Axpy interface, where *Axpy* 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 Axpy 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,MC,MR> A( 8, 8, grid );

// Set every entry of A to zero
A.SetToZero();

// Print the original A
A.Print("Original distributed A");

// Open up a LOCAL_TO_GLOBAL interface to A
AxpyInterface<double> interface;
interface.Attach( LOCAL_TO_GLOBAL, A );

// If we are process 0, then create a 3 x 3 identity matrix, B,
// and Axpy 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 Axpy as many submatrices as they
//       desire at this point.
if( grid.VCRank() == 0 )
{
    Matrix<double> B( 3, 3 );
    B.SetToIdentity();
    interface.Axpy( 2.0, B, 5, 5 );
}

// Have all processes collectively detach from A
interface.Detach();

// Print the updated A
A.Print("Updated 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 Axpy as many submatrices as they
//       desire at this point.
Matrix<double> C;
if( grid.VCRank() == 0 )
{
    C.ResizeTo( 8, 8 );
    C.SetToZero();
    interface.Axpy( 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 )
    C.Print("Process 0's local copy of A");
```

The output would be

```
Original 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 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

Updated 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
```

type AxpyType

An enum that can take on the value of either `LOCAL_TO_GLOBAL` or `GLOBAL_TO_LOCAL`, with the meanings described above.

type class **AxpyInterface<T>**

AxpyInterface()

Initialize a blank instance of the interface class. It will need to later be attached to a distributed matrix before any Axpy's can occur.

AxpyInterface(AxpyType *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`.

AxpyInterface(AxpyType *type*, const DistMatrix<T, MC, MR>& *Z*)

Initialize an interface to the (unmodifiable) distributed matrix `Z`; since `Z` cannot be modified, the only sensical `AxpyType` is `GLOBAL_TO_LOCAL`. The `AxpyType` argument was kept in order to be consistent with the previous routine.

void **Attach**(AxpyType *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`**(AxpyType *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 `AxpyType` argument was kept for consistency.

void **`Axpy`**(T *alpha*, Matrix<T>& *Z*, int *i*, int *j*)
If the interface was previously attached in the `LOCAL_TO_GLOBAL` direction, then the matrix `\alpha Z` 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 **`Axpy`**(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.

CHAPTER

FOUR

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 Axpy

Performs $Y := \alpha X + Y$ (hence the name *axpy*).

void **`Axpy`** (T *alpha*, const Matrix<T>& *X*, Matrix<T>& *Y*)

void **`Axpy`** (T *alpha*, const DistMatrix<T, U, V>& *X*, DistMatrix<T, U, V>& *Y*)

4.1.3 Conjugate

Note: This is not a standard BLAS routine, but it is BLAS-like.

$A := \bar{A}$. 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 *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 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*)

4.1.7 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*)

4.1.8 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*)

4.1.9 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*)

4.1.10 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. The diagonal of the trapezoidal matrix is defined relative to either the upper-left or bottom-right corner of the matrix, depending on the value of `side`; 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** (LeftOrRight *side*, UpperOrLower *uplo*, int *offset*, Matrix<T>& *A*)

void **MakeTrapezoidal** (LeftOrRight *side*, UpperOrLower *uplo*, int *offset*, DistMatrix<T, U, V>& *A*)

4.1.11 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.

typename Base<F>::type **Nrm2** (const Matrix<F>& *x*)

typename Base<F>::type **Nrm2** (const DistMatrix<F>& *x*)

4.1.12 Scal

$X := \alpha X$.

void **Scal** (T *alpha*, Matrix<T>& *X*)

void **Scal** (T *alpha*, DistMatrix<T, U, V>& *X*)

4.1.13 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*, LeftOrRight *side*, UpperOrLower *uplo*, int *offset*, Matrix<T>& *A*)

void **ScaleTrapezoid** (T *alpha*, LeftOrRight *side*, UpperOrLower *uplo*, int *offset*, DistMatrix<T, U, V>& *A*)

4.1.14 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*)

4.1.15 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*)

4.1.16 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** (LeftOrRight *side*, int *offset*, Matrix<T>& *A*, T *alpha*)

void **SetDiagonal** (LeftOrRight *side*, int *offset*, DistMatrix<T, U, V>& *A*, T *alpha*)

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 DistMatrix<T>& *x*, T *beta*, DistMatrix<T>& *y*)

4.2.2 Ger

General rank-one update: $A := \alpha x y^H + A$. 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): $A := \alpha x y^T + A$. 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: $y := \alpha A x + \beta y$, 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 $A := \alpha x x^H + A$, 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 $A := \alpha(x y^H + y x^H) + A$, 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 Symv

Symmetric matrix-vector multiply: $y := \alpha A x + \beta y$, 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()`.

4.2.9 Syr

Symmetric rank-one update: implicitly performs $A := \alpha x x^T + A$, 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*)

4.2.10 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*)

4.2.11 Trmv

Not yet written. Please call `Trmm()` for now.

4.2.12 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 an 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 $C := \alpha \text{op}_A(A)\text{op}_B(B) + \beta C$, where $\text{op}_A(M)$ and $\text{op}_B(M)$ can each be chosen from M, M^T, and 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 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*)

4.3.6 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()`.

4.3.7 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()`.

4.3.8 Trmm

Triangular matrix-matrix multiplication: performs $C := \alpha \text{op}(A)B$, or $C := \alpha B\text{op}(A)$, 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 $\text{op}(A)$ can be any of A, A^T, and 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*)

4.3.9 Trr2k

Triangular rank-2k update: performs $E := \alpha(\text{op}(A)\text{op}(B) + \text{op}(C)\text{op}(D)) + \beta E$, where only the triangle of E specified by *uplo* is modified, and $\text{op}(X)$ 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*)

4.3.10 Trrk

Triangular rank-k update: performs $C := \alpha \text{op}(A)\text{op}(B) + \beta C$, where only the triangle of *C* specified by *uplo* is modified, and $\text{op}(A)$ and $\text{op}(B)$ 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*)

4.3.11 Trtrmm

Note: This routine loosely corresponds with the LAPACK routines ?lauum.

Symmetric/Hermitian triangular matrix-matrix multiply: performs $L := L^T L$, $L := L^H L$, $U := UU^T$, or $U := UU^H$, 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*)

4.3.12 Trdtrmm

Note: This is a modification of Trtrmm for LDL factorizations.

Symmetric/Hermitian triangular matrix-matrix multiply (with diagonal scaling): performs $L := L^T D^{-1} L$, $L := L^H D^{-1} L$, $U := UD^{-1}U^T$, or $U := UD^{-1}U^H$, 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*)

4.3.13 Trsm

Triangular solve with multiple right-hand sides: performs $C := \alpha \text{op}(A)^{-1} B$, or $C := \alpha B \text{op}(A)^{-1}$, 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 op(A) can be any of A, A^T, and 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*)

4.3.14 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 Matrix<T>& *B*)

void **TwoSidedTrmm**(UpperOrLower *uplo*, UnitOrNonUnit *diag*, DistMatrix<T>& *A*, const DistMatrix<T>& *B*)

4.3.15 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 Matrix<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 `internal::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 `internal::LocalTrrk` routine for datatype `T`.

4.4.3 LocalTrr2kBlocksize

void **SetLocalTrr2kBlocksize<T>**(int *blocksize*)

Sets the local blocksize for the distributed `internal::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 `internal::LocalTrr2k` routine for datatype `T`.

CHAPTER

FIVE

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 eigensolvers, are placed here.

5.1 Invariants, inner products, norms, etc.

5.1.1 Condition number

The two-norm condition number

$$\kappa_2(A) = \|A\|_2 \|A^{-1}\|_2.$$

typename Base<F>::type **ConditionNumber** (const Matrix<F>& *A*)

typename Base<F>::type **ConditionNumber** (const DistMatrix<F, U, V>& *A*)

5.1.2 Determinant

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):

$$\det(A) = \prod_{i=0}^{n-1} \lambda_i.$$

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

$$\det(A) = \det(P)\det(L)\det(U) = \det(P) \prod_{i=0}^{n-1} \upsilon_{i,i} = \pm \prod_{i=0}^{n-1} \upsilon_{i,i},$$

where $\upsilon_{i,i}$ 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 struct **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.

typename Base<F>::type **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.

5.1.3 HPDDeterminant

A version of the above determinant specialized for Hermitian positive-definite matrices (which will therefore have all positive eigenvalues and a positive determinant).

typename Base<F>::type **HPDDeterminant** (UpperOrLower *uplo*, const Matrix<F>& *A*)

typename Base<F>::type **HPDDeterminant** (UpperOrLower *uplo*, const DistMatrix<F>& *A*)

typename Base<F>::type **HPDDeterminant** (UpperOrLower *uplo*, Matrix<F>& *A*, bool *canOverwrite=false*)

typename Base<F>::type **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.

5.1.4 Norm

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.

typename Base<F>::type **Norm** (const Matrix<F>& *A*, NormType *type=FROBENIUS_NORM*)

typename Base<F>::type **Norm** (const DistMatrix<F, U, V>& *A*, NormType *type=FROBENIUS_NORM*)
Assumes that the input is a fully-specified matrix.

typename Base<F>::type **HermitianNorm** (UpperOrLower *uplo*, const Matrix<F>& *A*, NormType *type=FROBENIUS_NORM*)

typename Base<F>::type **HermitianNorm** (UpperOrLower *uplo*, const DistMatrix<F>& *A*, NormType *type=FROBENIUS_NORM*)

typename Base<F>::type **SymmetricNorm** (UpperOrLower *uplo*, const Matrix<F>& *A*, NormType *type=FROBENIUS_NORM*)

typename Base<F>::type **SymmetricNorm** (UpperOrLower *uplo*, const DistMatrix<F>& *A*, NormType *type=FROBENIUS_NORM*)
Same as `Norm()`, but the matrix is implicitly Hermitian/symmetric with the data stored in the triangle specified by `UpperOrLower`. Also, while `Norm()` supports every type of distribution, `HermitianNorm()`/`SymmetricNorm()` currently only supports the standard matrix distribution.

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).

typename Base<F>::type **EntrywiseNorm** (const Matrix<F>& *A*, typename Base<F>::type *p*)

typename Base<F>::type **EntrywiseNorm** (const DistMatrix<F, U, V>& *A*, typename Base<F>::type *p*)

typename Base<F>::type **HermitianEntrywiseNorm** (UpperOrLower *uplo*, const Matrix<F>& *A*, typename Base<F>::type *p*)

typename Base<F>::type **HermitianEntrywiseNorm** (UpperOrLower *uplo*, const DistMatrix<F>& *A*, typename Base<F>::type *p*)

typename Base<F>::type **SymmetricEntrywiseNorm** (UpperOrLower *uplo*, const Matrix<F>& *A*, typename Base<F>::type *p*)

typename Base<F>::type **SymmetricEntrywiseNorm** (UpperOrLower *uplo*, const DistMatrix<F>& *A*, typename Base<F>::type *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.

typename Base<F>::type **EntrywiseOneNorm** (const Matrix<F>& *A*)

typename Base<F>::type **EntrywiseOneNorm** (const DistMatrix<F, U, V>& *A*)

typename Base<F>::type **HermitianEntrywiseOneNorm** (UpperOrLower *uplo*, const Matrix<F>& *A*)

typename Base<F>::type **HermitianEntrywiseOneNorm** (UpperOrLower *uplo*, const DistMatrix<F>& *A*)

typename Base<F>::type **SymmetricEntrywiseOneNorm** (UpperOrLower *uplo*, const Matrix<F>& *A*)

typename Base<F>::type **SymmetricEntrywiseOneNorm** (UpperOrLower *uplo*, const DistMatrix<F>& *A*)
The ℓ_1 norm of the columns of A stacked into a single vector.

typename Base<F>::type **FrobeniusNorm** (const Matrix<F>& *A*)

typename Base<F>::type **FrobeniusNorm** (const DistMatrix<F, U, V>& *A*)

typename Base<F>::type **HermitianFrobeniusNorm** (UpperOrLower *uplo*, const Matrix<F>& *A*)

typename Base<F>::type **HermitianFrobeniusNorm** (UpperOrLower *uplo*, const DistMatrix<F>& *A*)

typename Base<F>::type **SymmetricFrobeniusNorm** (UpperOrLower *uplo*, const Matrix<F>& *A*)

typename Base<F>::type **SymmetricFrobeniusNorm** (UpperOrLower *uplo*, const DistMatrix<F>& *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 $\text{vec}(A)$.

typename Base<F>::type **KyFanNorm**(const Matrix<F>& *A*, int *k*)

typename Base<F>::type **KyFanNorm**(const DistMatrix<F, U, V>& *A*, int *k*)

typename Base<F>::type **HermitianKyFanNorm**(UpperOrLower *uplo*, const Matrix<F>& *A*, int *k*)

typename Base<F>::type **HermitianKyFanNorm**(UpperOrLower *uplo*, const DistMatrix<F, U, V>& *A*, int *k*)

typename Base<F>::type **SymmetricKyFanNorm**(UpperOrLower *uplo*, const Matrix<F>& *A*, int *k*)

typename Base<F>::type **SymmetricKyFanNorm**(UpperOrLower *uplo*, const DistMatrix<F, U, V>& *A*, int *k*)

The sum of the largest *k* singular values.

typename Base<F>::type **InfinityNorm**(const Matrix<F>& *A*)

typename Base<F>::type **InfinityNorm**(const DistMatrix<F, U, V>& *A*)

typename Base<F>::type **HermitianInfinityNorm**(UpperOrLower *uplo*, const Matrix<F>& *A*)

typename Base<F>::type **HermitianInfinityNorm**(UpperOrLower *uplo*, const DistMatrix<F>& *A*)

typename Base<F>::type **SymmetricInfinityNorm**(UpperOrLower *uplo*, const Matrix<F>& *A*)

typename Base<F>::type **SymmetricInfinityNorm**(UpperOrLower *uplo*, const DistMatrix<F>& *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.

typename Base<F>::type **MaxNorm**(const Matrix<F>& *A*)

typename Base<F>::type **MaxNorm**(const DistMatrix<F, U, V>& *A*)

typename Base<F>::type **HermitianMaxNorm**(UpperOrLower *uplo*, const Matrix<F>& *A*)

typename Base<F>::type **HermitianMaxNorm**(UpperOrLower *uplo*, const DistMatrix<F>& *A*)

typename Base<F>::type **SymmetricMaxNorm**(UpperOrLower *uplo*, const Matrix<F>& *A*)

typename Base<F>::type **SymmetricMaxNorm**(UpperOrLower *uplo*, const DistMatrix<F>& *A*)

The maximum absolute value of the matrix entries.

typename Base<F>::type **NuclearNorm**(const Matrix<F>& *A*)

typename Base<F>::type **NuclearNorm**(const DistMatrix<F, U, V>& *A*)

typename Base<F>::type **HermitianNuclearNorm**(UpperOrLower *uplo*, const Matrix<F>& *A*)

typename Base<F>::type **HermitianNuclearNorm**(UpperOrLower *uplo*, const DistMatrix<F, U, V>& *A*)

typename Base<F>::type **SymmetricNuclearNorm**(UpperOrLower *uplo*, const Matrix<F>& *A*)

typename Base<F>::type **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 $p = \infty$.

typename Base<F>::type **OneNorm**(const Matrix<F>& *A*)

typename Base<F>::type **OneNorm**(const DistMatrix<F, U, V>& *A*)

typename Base<F>::type **HermitianOneNorm**(UpperOrLower *uplo*, const Matrix<F>& *A*)

typename Base<F>::type **HermitianOneNorm**(UpperOrLower *uplo*, const DistMatrix<F>& *A*)

typename Base<F>::type **SymmetricOneNorm**(UpperOrLower *uplo*, const Matrix<F>& *A*)

typename Base<F>::type **SymmetricOneNorm**(UpperOrLower *uplo*, const DistMatrix<F>& *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.

typename Base<F>::type **SchattenNorm**(const Matrix<F>& *A*, typename Base<F>::type *p*)

typename Base<F>::type **SchattenNorm**(const DistMatrix<F, U, V>& *A*, typename Base<F>::type *p*)

typename Base<F>::type **HermitianSchattenNorm**(UpperOrLower *uplo*, const Matrix<F>& *A*, typename Base<F>::type *p*)

typename Base<F>::type **HermitianSchattenNorm**(UpperOrLower *uplo*, const DistMatrix<F, U, V>& *A*, typename Base<F>::type *p*)

typename Base<F>::type **SymmetricSchattenNorm**(UpperOrLower *uplo*, const Matrix<F>& *A*, typename Base<F>::type *p*)

typename Base<F>::type **SymmetricSchattenNorm**(UpperOrLower *uplo*, const DistMatrix<F, U, V>& *A*, typename Base<F>::type *p*)

The ℓ_p norm of the singular values.

typename Base<F>::type **TwoNorm**(const Matrix<F>& *A*)

typename Base<F>::type **TwoNorm**(const DistMatrix<F, U, V>& *A*)

typename Base<F>::type **HermitianTwoNorm**(UpperOrLower *uplo*, const Matrix<F>& *A*)

typename Base<F>::type **HermitianTwoNorm**(UpperOrLower *uplo*, const DistMatrix<F, U, V>& *A*)

typename Base<F>::type **SymmetricTwoNorm**(UpperOrLower *uplo*, const Matrix<F>& *A*)

typename Base<F>::type **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 $p = \infty$.

5.1.5 Two-norm estimates

Since the two-norm is extremely useful, but expensive to compute, it is useful to be able to compute rough lower and upper bounds for it. The following routines provide cheap, rough estimates. The ability to compute sharper estimates will likely be added later.

typename Base<F>::type **TwoNormLowerBound**(const Matrix<F>& *A*)

typename Base<F>::type **TwoNormLowerBound**(const DistMatrix<F>& *A*)

Return the tightest lower bound on $\|A\|_2$ implied by the following inequalities:

$$\|A\|_2 \geq \|A\|_{\max},$$

$$\|A\|_2 \geq \frac{1}{\sqrt{n}}\|A\|_\infty,$$

$$\|A\|_2 \geq \frac{1}{\sqrt{m}}\|A\|_1, \quad \text{and}$$

$$\|A\|_2 \geq \frac{1}{\min(m,n)}\|A\|_F.$$

typename Base<F>::type **TwoNormUpperBound**(const Matrix<F>& *A*)

typename Base<F>::type **TwoNormUpperBound**(const DistMatrix<F>& *A*)

Return the tightest upper bound on $\|A\|_2$ implied by the following inequalities:

$$\|A\|_2 \leq \sqrt{mn}\|A\|_{\max},$$

$$\|A\|_2 \leq \sqrt{m}\|A\|_\infty,$$

$$\|A\|_2 \leq \sqrt{n}\|A\|_1, \quad \text{and}$$

$$\|A\|_2 \leq \sqrt{\|A\|_1 \|A\|_\infty}.$$

5.1.6 Trace

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*.

5.1.7 Hadamard

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*, DistMatrix<F, U, V>& *C*)

5.1.8 HilbertSchmidt

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*)

5.2 Factorizations

5.2.1 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.

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.

Note: See `HPSDCholesky()` for a generalization which also works for semi-definite matrices.

5.2.2 LDL^H factorization

Though the Cholesky factorization is ideal for most HPD matrices, there exist many Hermitian matrices whose eigenvalues are not all positive. The LDL^H factorization exists as slight relaxation of the Cholesky factorization, i.e., it computes lower-triangular (with unit diagonal) L and diagonal D such that $A = LDL^H$. If A is found to be numerically singular, then a `SingularMatrixException` will be thrown.

> **Warning:** The following routines do not pivot, so please use with caution.

void **LDLH** (Matrix<F>& *A*)

void **LDLH** (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.

void **LDLH** (Matrix<F>& *A*, Matrix<F>& *d*)

void **LDLH** (DistMatrix<F>& *A*, DistMatrix<F, MC, STAR>& *d*)
Same as above, but also return the diagonal in the column vector *d*.

5.2.3 LDL^T factorization

While the LDL^H factorization targets Hermitian matrices, the LDL^T factorization targets symmetric matrices. If A is found to be numerically singular, then a `SingularMatrixException` will be thrown.

> **Warning:** The following routines do not pivot, so please use with caution.

void **LDLT** (Matrix<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.

void **LDLT** (Matrix<F>& *A*, Matrix<F>& *d*)

void **LDLT** (DistMatrix<F>& *A*, DistMatrix<F, MC, STAR>& *d*)
Same as above, but also return the diagonal in the vector *d*.

5.2.4 LU factorization

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 diaganal 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, VC, STAR>& *p*)

Overwrites the matrix A with the LU decomposition of PA, where P is represented by the pivot vector *p*.

5.2.5 LQ factorization

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 representing $\hat{Q}$ are stored within the rows of the strictly upper trapezoid.

void **LQ** (Matrix<R>& *A*)

void **LQ** (DistMatrix<R>& *A*)

Overwrite the real matrix A with L and the Householder reflectors representing $\hat{Q}$.

void **LQ** (Matrix<Complex<R>>& *A*, Matrix<Complex<R>>& *t*)

void **LQ** (DistMatrix<Complex<R>>& *A*, DistMatrix<Complex<R>, MD, STAR>& *t*)

Overwrite the complex matrix A with L and the Householder reflectors representing $\hat{Q}$; unlike the real case, phase information is needed in order to define the (generalized) Householder transformations and is stored in the column vector *t*.

5.2.6 QR factorization

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.

void **QR** (Matrix<R>& *A*)

void **QR** (DistMatrix<R>& *A*)

Overwrite the real matrix A with R and the Householder reflectors representing $\hat{Q}$.

void **QR** (Matrix<Complex<R>>& *A*, Matrix<Complex<R>>& *t*)

void **QR** (DistMatrix<Complex<R>>& *A*, DistMatrix<Complex<R>, MD, STAR>& *t*)

Overwrite the complex matrix A with R and the Householder reflectors representing $\hat{Q}$; unlike the real case, phase information is needed in order to define the (generalized) Householder transformations and is stored in the column vector *t*.

5.3 Linear solvers

5.3.1 Cholesky solve

Solves $AX = B$ for X given Hermitian positive-definite (HPD) A and right-hand side matrix B. The solution is computed by first finding the Cholesky factorization of A and then performing two successive triangular solves against B:

$$B := A^{-1}B = (LL^H)^{-1}B = L^{-H}L^{-1}B$$

void **CholeskySolve** (UpperOrLower *uplo*, Matrix<F>& *A*, Matrix<F>& *B*)

void **CholeskySolve** (UpperOrLower *uplo*, DistMatrix<F>& *A*, DistMatrix<F>& *B*)
Overwrite B with the solution to $AX = B$, where A is Hermitian positive-definite and only the triangle of A specified by *uplo* is accessed.

5.3.2 Gaussian elimination

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.

5.3.3 Householder solve

Solves $AX = B$ or $A^H X = B$ for X in a least-squares sense given a general full-rank matrix $A \in \mathbb{F}^{m \times n}$. If $m \geq n$, then the first step is to form the QR factorization of A, otherwise the LQ factorization is computed.

- If solving $AX = B$, then either $X = R^{-1}Q^H B$ or $X = Q^H L^{-1} B$.
- If solving $A^H X = B$, then either $X = QR^{-H} B$ or $X = L^{-H} QB$.

void **HouseholderSolve** (Orientation *orientation*, Matrix<F>& *A*, const Matrix<F>& *B*, Matrix<F>& *X*)

void **HouseholderSolve** (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 X is overwritten with the solution.

5.3.4 Solve after Cholesky

Uses an in-place Cholesky factorization to solve against one or more right-hand sides.

void **SolveAfterCholesky** (UpperOrLower *uplo*, Orientation *orientation*, const Matrix<F>& *A*, Matrix<F>& *B*)

void **SolveAfterCholesky** (UpperOrLower *uplo*, Orientation *orientation*, const DistMatrix<F>& *A*, DistMatrix<F>& *B*)
Update $B := A^{-1}B$, $B := A^{-T}B$, or $B := A^{-H}B$, where one triangle of A has been overwritten with its Cholesky factor.

5.3.5 Solve after LU

Uses an in-place LU factorization (with or without partial pivoting) to solve against one or more right-hand sides.

void **SolveAfterLU** (Orientation *orientation*, const Matrix<F>& *A*, Matrix<F>& *B*)

void **SolveAfterLU** (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 **SolveAfterLU** (Orientation *orientation*, const Matrix<F>& *A*, const Matrix<int>& *p*, Matrix<F>& *B*)

void **SolveAfterLU** (Orientation *orientation*, const DistMatrix<F>& *A*, const DistMatrix<int, VC, STAR>& *p*, 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 partial pivoting, which satisfy $PA = LU$, where the permutation matrix P is represented by the pivot vector `p`.

5.4 Factorization-based inversion

5.4.1 General inversion

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.

5.4.2 HPD inversion

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*.

5.4.3 Triangular inversion

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.

5.5 Reduction to condensed form

5.5.1 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.

void **HermitianTridiag** (UpperOrLower *uplo*, Matrix<R>& *A*)

void **HermitianTridiag** (UpperOrLower *uplo*, DistMatrix<R>& *A*)

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.

void **HermitianTridiag** (UpperOrLower *uplo*, Matrix<Complex<R>>& *A*, Matrix<Complex<R>>& *t*)

void **HermitianTridiag** (UpperOrLower *uplo*, DistMatrix<Complex<R>>& *A*, DistMatrix<Complex<R>, STAR, STAR>& *t*)

Similar to above, but the complex Hermitian matrix is reduced to real symmetric tridiagonal form, with the added complication of needing to also store the phase information for the Householder vectors (the scaling can be inferred since the Householder vectors must be unit length); the scales with proper phases are returned in the column vector *t*.

Please see the *Tuning parameters* section for extensive information on maximizing the performance of Householder tridiagonalization.

5.5.2 General to Hessenberg

Not yet written, but it is planned and relatively straightforward after writing the reductions to tridiagonal and bidiagonal form.

5.5.3 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.

void **Bidiag** (Matrix<R>& *A*)

void **Bidiag** (DistMatrix<R>& *A*)

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.

Note: The $m < n$ case is not yet supported for the distributed version.

void **Bidiag** (Matrix<Complex<R>>& *A*, Matrix<Complex<R>>& *tP*, Matrix<Complex<R>>& *tQ*)

void **Bidiag** (DistMatrix<Complex<R>>& *A*, DistMatrix<Complex<R>, STAR, STAR>& *tP*, DistMatrix<Complex<R>, STAR, STAR>& *tQ*)

Same as above, but the complex scalings for the Householder reflectors are returned in the vectors *tP* and *tQ*.

Note: The $m < n$ case is not yet supported for the distributed version.

5.6 Eigensolvers and SVD

5.6.1 Hermitian eigensolver

Elemental provides a collection of routines for both full and partial solution of the Hermitian eigenvalue problem

$$AZ = Z\Omega,$$

where A is the given Hermitian matrix, and unitary Z and real diagonal Ω 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, $\{\omega_i\}_{i=0}^{n-1}$ or $\{(x_i, \omega_i)\}_{i=0}^{n-1}$.
2. Compute the eigenvalues or eigenpairs with a given range of indices, say $\{\omega_i\}_{i=a}^{b}$ or $\{(x_i, \omega_i)\}_{i=a}^{b}$, with $0 \leq a \leq b < n$.
3. Compute all eigenpairs (or just eigenvalues) with eigenvalues lying in a particular half-open interval, either $\{\omega_i \mid \omega_i \in (a, b]\}$ or $\{(x_i, \omega_i) \mid \omega_i \in (a, b]\}$.

As of now, all three approaches start with Householder tridiagonalization (ala `HermitianTridiag()`) and then call Matthias Petschow and Paolo Bientinesi's PMRRR for the tridiagonal eigenvalue problem.

Note: Please see the *Tuning parameters* section for information on optimizing the reduction to tridiagonal form, as it is the dominant cost in all of Elemental's Hermitian eigensolvers.

Full spectrum computation

void **HermitianEig**(UpperOrLower *uplo*, DistMatrix<double>& *A*, DistMatrix<double, VR, STAR>& *w*)

Compute the full set of eigenvalues of the double-precision real symmetric distributed matrix *A*.

void **HermitianEig**(UpperOrLower *uplo*, DistMatrix<Complex<double>>& *A*, DistMatrix<double, VR, STAR>& *w*)

Compute the full set of eigenvalues of the double-precision complex Hermitian distributed matrix *A*.

void **HermitianEig**(UpperOrLower *uplo*, DistMatrix<double>& *A*, DistMatrix<double, VR, STAR>& *w*, DistMatrix<double>& *Z*)

Compute the full set of eigenpairs of the double-precision real symmetric distributed matrix *A*.

void **HermitianEig**(UpperOrLower *uplo*, DistMatrix<Complex<double>>& *A*, DistMatrix<double, VR, STAR>& *w*, DistMatrix<double>& *Z*)

Compute the full set of eigenpairs of the double-precision complex Hermitian distributed matrix *A*.

Index-based subset computation

void **HermitianEig**(UpperOrLower *uplo*, DistMatrix<double>& *A*, DistMatrix<double, VR, STAR>& *w*, int *a*, int *b*)

Compute the eigenvalues of a double-precision real symmetric distributed matrix *A* with indices in the range $a, a+1, ..., b$.

void **HermitianEig**(UpperOrLower *uplo*, DistMatrix<Complex<double>>& *A*, DistMatrix<double, VR, STAR>& *w*, int *a*, int *b*)

Compute the eigenvalues of a double-precision complex Hermitian distributed matrix *A* with indices in the range $a, a+1, ..., b$.

void **HermitianEig**(UpperOrLower *uplo*, DistMatrix<double>& *A*, DistMatrix<double, VR, STAR>& *w*, DistMatrix<double>& *Z*, int *a*, int *b*)

Compute the eigenpairs of a double-precision real symmetric distributed matrix *A* with indices in the range $a, a+1, ..., b$.

void **HermitianEig** (UpperOrLower *uplo*, DistMatrix<Complex<double>>& *A*, DistMatrix<double, VR, STAR>& *w*, DistMatrix<double>& *Z*)
Compute the eigenpairs of a double-precision complex Hermitian distributed matrix *A* with indices in the range $a, a+1, ..., b$.

Range-based subset computation

void **HermitianEig** (UpperOrLower *uplo*, DistMatrix<double>& *A*, DistMatrix<double, VR, STAR>& *w*, double *a*, double *b*)
Compute the eigenvalues of a double-precision real symmetric distributed matrix *A* lying in the half-open interval $(a, b]$.

void **HermitianEig** (UpperOrLower *uplo*, DistMatrix<Complex<double>>& *A*, DistMatrix<double, VR, STAR>& *w*, double *a*, double *b*)
Compute the eigenvalues of a double-precision complex Hermitian distributed matrix *A* lying in the half-open interval $(a, b]$.

void **HermitianEig** (UpperOrLower *uplo*, DistMatrix<double>& *A*, DistMatrix<double, VR, STAR>& *w*, DistMatrix<double>& *Z*, double *a*, double *b*)
Compute the eigenpairs of a double-precision real symmetric distributed matrix *A* with eigenvalues lying in the half-open interval $(a, b]$.

void **HermitianEig** (UpperOrLower *uplo*, DistMatrix<Complex<double>>& *A*, DistMatrix<double, VR, STAR>& *w*, DistMatrix<double>& *Z*)
Compute the eigenpairs of a double-precision complex Hermitian distributed matrix *A* with eigenvalues lying in the half-open interval $(a, b]$.

Sorting the eigenvalues/eigenpairs

Since extra time is required in order to sort the eigenvalues/eigenpairs, they are not sorted by default. However, this can be remedied by the appropriate routine from the following list:

void **SortEig** (Matrix<R>& *w*)

void **SortEig** (DistMatrix<R, VR, STAR>& *w*)
Sort a column-vector of real eigenvalues into non-decreasing order.

void **SortEig** (Matrix<R>& *w*, Matrix<R>& *Z*)

void **SortEig** (DistMatrix<R, VR, STAR>& *w*, DistMatrix<R>& *Z*)
Sort a set of real eigenpairs into non-decreasing order (based on the eigenvalues).

void **SortEig** (Matrix<R>& *w*, Matrix<Complex<R>>& *Z*)

void **SortEig** (DistMatrix<R, VR, STAR>& *w*, DistMatrix<Complex<R>>& *Z*)
Sort a set of real eigenvalues and complex eigenvectors into non-decreasing order (based on the eigenvalues).

5.6.2 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 $iG = Z\Omega Z^H$.

3. Recognize that $G = Z(-i\Omega)Z^H$ provides an EVD of G.

Please see the `HermitianEig()` documentation for more details.

Note: Please see the *Tuning parameters* section for information on optimizing the reduction to tridiagonal form, as it is the dominant cost in all of Elemental's Hermitian eigensolvers.

Full spectrum computation

void **SkewHermitianEig**(UpperOrLower *uplo*, DistMatrix<double>& *G*, DistMatrix<double, VR, STAR>& *wImag*)
: Compute the full set of eigenvalues of the double-precision real skew-symmetric distributed matrix *G*.

void **SkewHermitianEig**(UpperOrLower *uplo*, DistMatrix<Complex<double>>& *G*, DistMatrix<double, VR, STAR>& *wImag*)
: Compute the full set of eigenvalues of the double-precision complex skew-Hermitian distributed matrix *G*.

void **SkewHermitianEig**(UpperOrLower *uplo*, DistMatrix<double>& *G*, DistMatrix<double, VR, STAR>& *wImag*, DistMatrix<Complex<double>>& *Z*)
: Compute the full set of eigenpairs of the double-precision real skew-symmetric distributed matrix *G*.

void **SkewHermitianEig**(UpperOrLower *uplo*, DistMatrix<Complex<double>>& *G*, DistMatrix<double, VR, STAR>& *wImag*, DistMatrix<Complex<double>>& *Z*)
: Compute the full set of eigenpairs of the double-precision complex skew-Hermitian distributed matrix *G*.

Index-based subset computation

void **SkewHermitianEig**(UpperOrLower *uplo*, DistMatrix<double>& *G*, DistMatrix<double, VR, STAR>& *wImag*, int *a*, int *b*)
: Compute the eigenvalues of a double-precision real skew-symmetric distributed matrix *G* with indices in the range $a, a+1, ..., b$.

void **SkewHermitianEig**(UpperOrLower *uplo*, DistMatrix<Complex<double>>& *G*, DistMatrix<double, VR, STAR>& *wImag*, int *a*, int *b*)
: Compute the eigenvalues of a double-precision complex skew-Hermitian distributed matrix *G* with indices in the range $a, a+1, ..., b$.

void **SkewHermitianEig**(UpperOrLower *uplo*, DistMatrix<double>& *G*, DistMatrix<double, VR, STAR>& *wImag*, DistMatrix<Complex<double>>& *Z*, int *a*, int *b*)
: Compute the eigenpairs of a double-precision real skew-symmetric distributed matrix *G* with indices in the range $a, a+1, ..., b$.

void **SkewHermitianEig**(UpperOrLower *uplo*, DistMatrix<Complex<double>>& *G*, DistMatrix<double, VR, STAR>& *wImag*, DistMatrix<Complex<double>>& *Z*)
: Compute the eigenpairs of a double-precision complex skew-Hermitian distributed matrix *G* with indices in the range $a, a+1, ..., b$.

Range-based subset computation

void **SkewHermitianEig**(UpperOrLower *uplo*, DistMatrix<double>& *G*, DistMatrix<double, VR, STAR>& *wImag*, double *a*, double *b*)
: Compute the eigenvalues of a double-precision real skew-symmetric distributed matrix *G* lying in the half-open interval $(a, b]i$.

void **SkewHermitianEig** (UpperOrLower *uplo*, DistMatrix<Complex<double>>& *G*, DistMatrix<double, VR, STAR>& *wImag*, double *a*, double *b*)
Compute the eigenvalues of a double-precision complex skew-Hermitian distributed matrix *G* lying in the half-open interval $(a, b]i$.

void **SkewHermitianEig** (UpperOrLower *uplo*, DistMatrix<double>& *G*, DistMatrix<double, VR, STAR>& *wImag*, DistMatrix<Complex<double>>& *Z*, double *a*, double *b*)
Compute the eigenpairs of a double-precision real skew-symmetric distributed matrix *G* with eigenvalues lying in the half-open interval $(a, b]i$.

void **SkewHermitianEig** (UpperOrLower *uplo*, DistMatrix<Complex<double>>& *G*, DistMatrix<double, VR, STAR>& *wImag*, DistMatrix<Complex<double>>& *Z*)
Compute the eigenpairs of a double-precision complex skew-Hermitian distributed matrix *G* with eigenvalues lying in the half-open interval $(a, b]i$.

5.6.3 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:

$$ABx = \omega x,$$

which is denoted with the value `ABX` via the `HermitianGenDefiniteEigType` enum,

$$BAx = \omega x,$$

which uses the `BAX` value, and finally

$$Ax = \omega Bx,$$

which uses the `AXBX` enum value.

type HermitianGenDefiniteEigType
An enum for specifying either the `ABX`, `BAX`, or `AXBX` generalized eigenvalue problems (described above).

Full spectrum computation

void **HermitianGenDefiniteEig** (HermitianGenDefiniteEigType *type*, UpperOrLower *uplo*, DistMatrix<double>& *A*, DistMatrix<double>& *B*, DistMatrix<double, VR, STAR>& *w*)
Compute the full set of eigenvalues of a generalized EVP involving the double-precision real symmetric distributed matrices *A* and *B*, where *B* is also positive-definite.

void **HermitianGenDefiniteEig** (HermitianGenDefiniteEigType *type*, UpperOrLower *uplo*, DistMatrix<Complex<double>>& *A*, DistMatrix<Complex<double>>& *B*, DistMatrix<double, VR, STAR>& *w*)
Compute the full set of eigenvalues of a generalized EVP involving the double-precision complex Hermitian distributed matrices *A* and *B*, where *B* is also positive-definite.

void **HermitianGenDefiniteEig** (HermitianGenDefiniteEigType *type*, UpperOrLower *uplo*, DistMatrix<double>& *A*, DistMatrix<double>& *B*, DistMatrix<double, VR, STAR>& *w*, DistMatrix<double>& *Z*)
Compute the full set of eigenpairs of a generalized EVP involving the double-precision real symmetric distributed matrices *A* and *B*, where *B* is also positive-definite.

void **HermitianGenDefiniteEig** (HermitianGenDefiniteEigType *type*, UpperOrLower *uplo*, DistMatrix<Complex<double>>& *A*, DistMatrix<Complex<double>>& *B*, DistMatrix<double, VR, STAR>& *w*, DistMatrix<double>& *Z*)
Compute the full set of eigenpairs of a generalized EVP involving the double-precision complex Hermitian distributed matrices *A* and *B*, where *B* is also positive-definite.

Index-based subset computation

void **HermitianGenDefiniteEig**(HermitianGenDefiniteEigType *type*, UpperOrLower *uplo*, DistMatrix<double>& *A*, DistMatrix<double>& *B*, DistMatrix<double, VR, STAR>& *w*, int *a*, int *b*)

Compute the eigenvalues with indices in the range $a, a+1, ..., b$ of a generalized EVP involving the double-precision real symmetric distributed matrices *A* and *B*, where *B* is also positive-definite.

void **HermitianGenDefiniteEig**(HermitianGenDefiniteEigType *type*, UpperOrLower *uplo*, DistMatrix<Complex<double>>& *A*, DistMatrix<Complex<double>>& *B*, DistMatrix<double, VR, STAR>& *w*, int *a*, int *b*)

Compute the eigenvalues with indices in the range $a, a+1, ..., b$ of a generalized EVP involving the double-precision complex Hermitian distributed matrices *A* and *B*, where *B* is also positive-definite.

void **HermitianGenDefiniteEig**(HermitianGenDefiniteEigType *type*, UpperOrLower *uplo*, DistMatrix<double>& *A*, DistMatrix<double>& *B*, DistMatrix<double, VR, STAR>& *w*, DistMatrix<double>& *Z*, int *a*, int *b*)

Compute the eigenpairs with indices in the range $a, a+1, ..., b$ of a generalized EVP involving the double-precision real symmetric distributed matrices *A* and *B*, where *B* is also positive-definite.

void **HermitianGenDefiniteEig**(HermitianGenDefiniteEigType *type*, UpperOrLower *uplo*, DistMatrix<Complex<double>>& *A*, DistMatrix<Complex<double>>& *B*, DistMatrix<double, VR, STAR>& *w*, DistMatrix<double>& *Z*)

Compute the eigenpairs with indices in the range $a, a+1, ..., b$ of a generalized EVP involving the double-precision complex Hermitian distributed matrices *A* and *B*, where *B* is also positive-definite.

Range-based subset computation

void **HermitianGenDefiniteEig**(HermitianGenDefiniteEigType *type*, UpperOrLower *uplo*, DistMatrix<double>& *A*, DistMatrix<double>& *B*, DistMatrix<double, VR, STAR>& *w*, double *a*, double *b*)

Compute the eigenvalues lying in the half-open interval $(a, b]$ of a generalized EVP involving the double-precision real symmetric distributed matrices *A* and *B*, where *B* is also positive-definite.

void **HermitianGenDefiniteEig**(HermitianGenDefiniteEigType *type*, UpperOrLower *uplo*, DistMatrix<Complex<double>>& *A*, DistMatrix<Complex<double>>& *B*, DistMatrix<double, VR, STAR>& *w*, double *a*, double *b*)

Compute the eigenvalues lying in the half-open interval $(a, b]$ of a generalized EVP involving the double-precision complex Hermitian distributed matrices *A* and *B*, where *B* is also positive-definite.

void **HermitianGenDefiniteEig**(HermitianGenDefiniteEigType *type*, UpperOrLower *uplo*, DistMatrix<double>& *A*, DistMatrix<double>& *B*, DistMatrix<double, VR, STAR>& *w*, DistMatrix<double>& *Z*, double *a*, double *b*)

Compute the eigenpairs whose eigenvalues lie in the half-open interval $(a, b]$ of a generalized EVP involving the double-precision real symmetric distributed matrices *A* and *B*, where *B* is also positive-definite.

void **HermitianGenDefiniteEig**(HermitianGenDefiniteEigType *type*, UpperOrLower *uplo*, DistMatrix<Complex<double>>& *A*, DistMatrix<Complex<double>>& *B*, DistMatrix<double, VR, STAR>& *w*, DistMatrix<double>& *Z*)

Compute the eigenpairs whose eigenvalues lie in the half-open interval $(a, b]$ of a generalized EVP involving the double-precision complex Hermitian distributed matrices *A* and *B*, where *B* is also positive-definite.

5.6.4 Unitary eigensolver

Not yet written, will likely be based on Ming Gu's unitary Divide and Conquer algorithm for unitary Hessenberg matrices.

5.6.5 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.

5.6.6 Schur decomposition

Not yet written, will likely eventually include Greg Henry et al.'s and Robert Granat et al.'s approaches.

5.6.7 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.

void **HermitianSVD** (UpperOrLower *uplo*, DistMatrix<F>& *A*, DistMatrix<typename Base<F>::type, 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 **HermitianSingularValues** (UpperOrLower *uplo*, DistMatrix<F>& *A*, DistMatrix<typename Base<F>::type, VR, STAR>& *s*)

Return the singular values of A in s. Note that the appropriate triangle of *A* is overwritten during computation.

5.6.8 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 Halley iteration.

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 **Halley** (Matrix<F>& *A*, typename Base<F>::type *upperBound*)

void **Halley** (DistMatrix<F>& *A*, typename Base<F>::type *upperBound*)

Overwrites A with the Q from the polar decomposition using a simple QR-based Halley iteration. **TODO: better explanation**

void **QDWH** (Matrix<F>& *A*, typename Base<F>::type *lowerBound*, typename Base<F>::type *upperBound*)

void **QDWH** (DistMatrix<F>& *A*, typename Base<F>::type *lowerBound*, typename Base<F>::type *upperBound*)

Overwrites A with the Q from the polar decomposition using a QR-based dynamically weighted Halley iteration. **TODO: better explanation**

5.6.9 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.

void **SVD** (Matrix<F>& *A*, Matrix<typename Base<F>::type>& *s*, Matrix<F>& *V*)

void **SVD** (DistMatrix<F>& *A*, DistMatrix<typename Base<F>::type, VR, STAR>& *s*, DistMatrix<F>& *V*)
Overwrites A with U, s with the diagonal entries of Σ, and V with V.

void **SingularValues** (Matrix<F>& *A*, Matrix<typename Base<F>::type>& *s*)

void **SingularValues** (DistMatrix<F>& *A*, DistMatrix<typename Base<F>::type, VR, STAR>& *s*)
Forms the singular values of A in s. Note that A is overwritten in order to compute the singular values.

5.7 Matrix functions

5.7.1 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 $A = Z\Omega Z^H$, then

$$f(A) = f(Z\Omega Z^H) = Zf(\Omega)Z^H.$$

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*, DistMatrix<F>& *A*, const RealFunctor& *f*)
Modifies the eigenvalues of the passed-in Hermitian matrix by replacing each eigenvalue ω_i with $f(\omega_i) \in \mathbb{R}$. `RealFunctor` is any class which has the member function `R operator()( R omega ) const`. See examples/lapack-like/RealSymmetricFunction.cpp for an example usage.

void **ComplexHermitianFunction** (UpperOrLower *uplo*, DistMatrix<Complex<R>>& *A*, const ComplexFunctor& *f*)
Modifies the eigenvalues of the passed-in complex Hermitian matrix by replacing each eigenvalue ω_i with $f(\omega_i) \in \mathbb{C}$. `ComplexFunctor` can be any class which has the member function `Complex<R> operator()( R omega ) const`. See examples/lapack-like/ComplexHermitianFunction.cpp for an example usage.

TODO: A version of ComplexHermitianFunction which begins with a real matrix

5.7.2 Pseudoinverse

Pseudoinverse (Matrix<F>& *A*)

Pseudoinverse (DistMatrix<F>& A)

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.

HermitianPseudoinverse (UpperOrLower *uplo*, DistMatrix<F>& A)

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.

5.7.3 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 **HPSDSquareRoot** (UpperOrLower *uplo*, DistMatrix<F>& A)

Hermitian matrices with non-negative eigenvalues have a natural matrix square root which remains Hermitian. This routine attempts to overwrite a matrix with its square root and throws a `NonHPSDMatrixException` if any sufficiently negative eigenvalues are computed.

TODO: HermitianSquareRoot

5.7.4 Semi-definite Cholesky

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 $-n\epsilon\|A\|_2$, then a `NonHPSDMatrixException` will be thrown.

void **HPSDCholesky** (UpperOrLower *uplo*, DistMatrix<F>& A)

Overwrite the *uplo* triangle of the potentially singular matrix A with its Cholesky factor.

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).

5.8.2 Applying packed Householder transforms

TODO: Describe `ApplyPackedReflectors` here.

5.8.3 Applying pivots

TODO

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 tridiaogal 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 signficant. 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()` and `Hemv()` (for real and complex data, respectively), so users interested in maximizing the performance of the first algorithm will likely want to investigate different values for the local blocksizes through the routines

`SetLocalSymvBlocksize<T>( int blocksize )` and `SetLocalHemvBlocksize<T>( int blocksize )`; the default values are both 64.

void **SetHermitianTridiagApproach**(HermitianTridiagApproach *approach*)
Sets the algorithm used by subsequent calls to `HermitianTridiag()`.

HermitianTridiagApproach **GetHermitianTridiagApproach**()
Queries the currently set approach for the reduction of a Hermitian matrix to tridiagonal form.

void **SetHermitianTridiagGridOrder**(GridOrder *order*)
Sets the ordering to use for the first $\hat{p}^2$ processes in the construction of the $\hat{p} \times \hat{p}$ subgrid. This is only relevant to the `HERMITIAN_TRIDIAG_SQUARE` approach.

GridOrder **GetHermitianTridiagGridOrder**()
Queries the currently set approach for the ordering of the square subgrid needed by the `HERMITIAN_TRIDIAG_SQUARE` approach to the tridiagonalization of a Hermitian matrix.

CHAPTER

SIX

CONVEX OPTIMIZATION

6.1 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, $-\log(\det(A))$.

typename Base<F>::type **LogBarrier** (UpperOrLower *uplo*, const Matrix<F>& *A*)

typename Base<F>::type **LogBarrier** (UpperOrLower *uplo*, const DistMatrix<F>& *A*)

typename Base<F>::type **LogBarrier** (UpperOrLower *uplo*, Matrix<F>& *A*, bool *canOverwrite=false*)

typename Base<F>::type **LogBarrier** (UpperOrLower *uplo*, DistMatrix<F>& *A*, bool *canOverwrite=false*)

6.2 LogDetDivergence

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

$$D_{ld}(A,B) = \|Z\|_F^2 - 2\log(\det(Z)) - n.$$

typename Base<F>::type **LogDetDivergence** (UpperOrLower *uplo*, const Matrix<F>& *A*, const Matrix<F>& *B*)

typename Base<F>::type **LogDetDivergence** (UpperOrLower *uplo*, const DistMatrix<F>& *A*, const DistMatrix<F>& *B*)

CHAPTER

SEVEN

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.

7.1 Deterministic

7.1.1 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** (const std::vector<F>& *x*, const std::vector<F>& *y*, Matrix<F>& *A*)

void **Cauchy** (const std::vector<F>& *x*, const std::vector<F>& *y*, DistMatrix<F, U, V>& *A*)
    Generate a Cauchy matrix using the defining vectors, x and y.

7.1.2 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** (const std::vector<F>& *r*, const std::vector<F>& *s*, const std::vector<F>& *x*, const std::vector<F>& *y*, Matrix<F>& *A*)

void **CauchyLike** (const std::vector<F>& *r*, const std::vector<F>& *s*, const std::vector<F>& *x*, const std::vector<F>& *y*, DistMatrix<F, U, V>& *A*)
Generate a Cauchy-like matrix using the defining vectors: r, s, x, and y.

7.1.3 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** (const std::vector<T>& *a*, Matrix<T>& *A*)

void **Circulant** (const std::vector<T>& *a*, DistMatrix<T, U, V>& *A*)
Generate a circulant matrix using the vector `a`.

7.1.4 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** (const std::vector<T>& *d*, Matrix<T>& *D*)

void **Diagonal** (const std::vector<T>& *d*, DistMatrix<T, U, V>& *D*)
Construct a diagonal matrix from the vector of diagonal values, d.

7.1.5 DiscreteFourier

The $n \times n$ *Discrete Fourier Transform* (DFT) matrix, say A, is given by

$$\alpha_{i,j} = \frac{e^{-2\pi ij/n}}{\sqrt{n}}.$$

void **DiscreteFourier** (int *n*, Matrix<Complex<R>>& *A*)

void **DiscreteFourier** (int *n*, DistMatrix<Complex<R>, U, V>& *A*)
Set the matrix `A` equal to the $n \times n$ DFT matrix.

void **MakeDiscreteFourier** (Matrix<Complex<R>>& *A*)

void **MakeDiscreteFourier** (DistMatrix<Complex<R>, U, V>& *A*)
Turn the existing $n \times n$ matrix `A` into a discrete Fourier matrix.

7.1.6 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** (int *m*, int *n*, const std::vector<T>& *b*, Matrix<T>& *A*)

void **Hankel** (int *m*, int *n*, const std::vector<T>& *b*, DistMatrix<T, U, V>& *A*)
Create an $m \times n$ Hankel matrix from the generate vector, b.

7.1.7 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** (int *n*, Matrix<F>& *A*)

void **Hilbert** (int *n*, DistMatrix<F, U, V>& *A*)
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.

7.1.8 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** (int *m*, int *n*, Matrix<T>& *A*)

void **Identity** (int *m*, int *n*, DistMatrix<T, U, V>& *A*)
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.

7.1.9 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

$$\beta_j = \frac{1}{2}\left(1 - (2(j+1))^{-2}\right)^{-1/2},$$

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** (int *n*, Matrix<F>& *A*)

void **Legendre** (int *n*, DistMatrix<F, U, V>& *A*)
Sets the matrix `A` equal to the $n \times n$ Jacobi matrix.

7.1.10 Ones

Create an $m \times n$ matrix of all ones.

void **Ones** (int *m*, int *n*, Matrix<T>& *A*)

void **Ones** (int *m*, int *n*, DistMatrix<T, U, V>& *A*)
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.

7.1.11 OneTwoOne

A "1-2-1" matrix is tridiagonal with a diagonal of all twos and sub- and super-diagonals of all ones.

void **OneTwoOne** (int *n*, Matrix<T>& *A*)

void **OneTwoOne** (int *n*, DistMatrix<T, U, V>& *A*)
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".

7.1.12 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** (int *m*, int *n*, const std::vector<T>& *b*, Matrix<T>& *A*)

void **Toeplitz** (int *m*, int *n*, const std::vector<T>& *b*, DistMatrix<T, U, V>& *A*)
Build the matrix `A` using the generating vector b.

void **MakeToeplitz** (const std::vector<T>& *b*, Matrix<T>& *A*)

void **MakeToeplitz** (const std::vector<T>& *b*, DistMatrix<T, U, V>& *A*)
Turn the matrix `A` into a Toeplitz matrix defined from the generating vector b.

7.1.13 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** (int *k*, Matrix<T>& *W*, bool *binary=false*)

void **Walsh** (int *k*, DistMatrix<T, U, V>& *W*, bool *binary=false*)
Set the matrix W equal to the k'th (possibly binary) Walsh matrix.

7.1.14 Wilkinson

A *Wilkinson matrix* of order k is a tridiagonal matrix with diagonal

$$[k, k-1, k-2, ..., 1, 0, 1, ..., k-2, k-1, k],$$

and sub- and super-diagonals of all ones.

void **Wilkinson** (int *k*, Matrix<T>& *W*)

void **Wilkinson** (int *k*, DistMatrix<T, U, V>& *W*)
Set the matrix W equal to the k'th Wilkinson matrix.

7.1.15 Zeros

Create an $m \times n$ matrix of all zeros.

void **Zeros** (int *m*, int *n*, Matrix<T>& *A*)

void **Zeros** (int *m*, int *n*, DistMatrix<T, U, V>& *A*)
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.

7.2 Random

7.2.1 Uniform

We call an $m \times n$ matrix 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** (int *m*, int *n*, Matrix<T>& *A*, T *center=0*, typename Base<T>::type *radius=1*)

void **Uniform** (int *m*, int *n*, DistMatrix<T, U, V>& *A*, T *center=0*, typename Base<T>::type *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*, typename Base<T>::type *radius=1*)

void **MakeUniform** (DistMatrix<T, U, V>& *A*, T *center=0*, typename Base<T>::type *radius=1*)
Sample each entry of `A` from $U(B_r(x))$, where r is given by `radius` and x is given by `center`.

7.2.2 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** (int *n*, Matrix<F>& *A*, typename Base<F>::type *lower=0*, typename Base<F>::type *upper=1*)

void **HermitianUniformSpectrum** (int *n*, DistMatrix<F, U, V>& *A*, typename Base<F>::type *lower=0*, typename Base<F>::type *upper=1*)
Build the $n \times n$ matrix `A` with a spectrum sampled uniformly from the interval $(lower, upper]$.

void **MakeHermitianUniformSpectrum**(Matrix<F>& *A*, typename Base<F>::type *lower=0*, typename Base<F>::type *upper=1*)

void **MakeHermitianUniformSpectrum**(DistMatrix<F, U, V>& *A*, typename Base<F>::type *lower=0*, typename Base<F>::type *upper=1*)

Sample the entries of the square matrix `A` from the interval $(lower, upper]$.

7.2.3 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**(int *n*, Matrix<Complex<R>>& *A*, Complex<R> *center=0*, R *radius=1*)

void **NormalUniformSpectrum**(int *n*, DistMatrix<Complex<R>, U, V>& *A*, Complex<R> *center=0*, R *radius=1*)

Build the $n \times n$ matrix `A` with a spectrum sampled uniformly from the ball $B_{\mathrm{radius}}(\mathrm{center})$.

void **MakeNormalUniformSpectrum**(Matrix<Complex<R>>& *A*, Complex<R> *center=0*, R *radius=1*)

void **MakeNormalUniformSpectrum**(DistMatrix<Complex<R>, U, V>& *A*, Complex<R> *center=0*, R *radius=1*)

Sample the entries of the square matrix `A` from the ball in the complex plane centered at `center` with radius `radius`.

CHAPTER EIGHT

INDICES

- *genindex*
- *search*

INDEX

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z