QLA Linear Algebra Interface for QCD

Version 1.2

SciDAC Software Coordinating Committee

June 25, 2006

1 Introduction

This is a user's guide for the C binding for the QCD Linear Algebra Applications Programmer Interface developed under the auspices of the U.S. Department of Energy Scientific Discovery through Advanced Computing (SciDAC) program.

The QLA interface operates on single node data. Data objects are primitives, such as SU(N) matrices or SU(N) vectors, or arrays of such objects. Linear algebra operations, such as products of matrices and vectors, are carried out either on single objects or arrays of objects. For arrays of objects a variety of indirect addressing schemes is supported, including indirect indexing (gathers and scatters) and pointers. Various reduction operations (e.g. global sums) and fill operations (e.g. setting values to zero) are also included. It is intended that a subset of operations in this interface be optimized for specific architectures, with attention paid to efficient cache loading and fast instructions.

2 Datatypes

2.1 Generic Names

There are two levels of generic naming: fully generic in which both color and precision may be controlled globally through two macros and color-generic in which precision is explicit but not color. Generic naming applies to datatypes, module names, and accessor macros and follows similar rules.

Names for fully generic datatypes are listed in the table below.

name	abbreviation	description
`QLA_Real`	`R`	real
`QLA_Complex`	`C`	complex
`QLA_Int`	`I`	integer
`QLA_ColorMatrix`	`M`	$N_c \times N_c$ complex matrix
`QLA_HalfFermion`	`H`	two-spin, N_c color spinor
`QLA_DiracFermion`	`D`	four-spin, N_c color spinor
`QLA_ColorVector`	`V`	one-spin, N_c color spinor
`QLA_DiracPropagator`	`P`	$4N_c \times 4N_c$ complex matrix
`QLA_RandomState`	`S`	implementation dependent

Names for color-generic datatypes are obtained by inserting an `_F` for single precision, `_D` for double precision, or `_Q` for extended precision after `QLA` where appropriate. Thus `QLA_D_`ColorMatrix specifies a double precision color matrix with color to be set through a global macro.

An extended precision type with abbreviation `Q` is also defined, but it is currently intended only for extending precision in the global reduction of double precision data, and then only where the architecture and compiler supports it. Only a handful of functions use it, namely, reduction, type conversion and some copying and incrementing.

2.2 Specific Types for Color and Precision

According to the chosen color and precision, names for specific floating point types are constructed from names for generic types. Thus `QLA_ColorMatrix` becomes `QLA_`PC`_ColorMatrix` where the precision P is `F`, `D`, or `Q` according to the table below

abbreviation	description
`F`	single precision
`D`	double precision
`Q`	extended precision

and C is `2`, `3`, or `N`, if color is a consideration, as listed below.

abbreviation	description
`2`	SU(2)
`3`	SU(3)
`N`	SU(N)

(For the moment there is no provision for distinguishing among integer sizes.)

For example, the type

```
QLA_F3_DiracFermion
```

describes a four-spin, three-color spinor.

The general color choice `N` can also be used for specialized $SU(2)$ or $SU(3)$ at the cost of degrading performance.

2.3 Color and Precision Uniformity

In standard coding practice it is assumed that a user keeps one of the precision and color options in force throughout the compilation. So as a rule all functions in the interface take operands of the same precision and color. As with data type names, function names come in generic and color- and precision-specific forms, as described in the next section. Exceptions to this rule are functions that explicitly convert from double to single precision and vice versa. These and functions that do not depend on color or precision are divided among seventeen separate libraries. If the user chooses to adopt color and precision uniformity, then all variables can be defined with generic types and all functions accessed through generic names. The prevailing color and precision is then defined through macros. The interface automatically translates data type names and function names to the appropriate specific type names through typedefs and macros. With such a scheme and careful coding, changing only two macros and the QLA library converts code from one color and precision choice to another.

2.4 Breaking Color and Precision Uniformity

It is permissible for a user to mix precision and color choices. This is done by declaring variables with specific type names, using functions with specific names, and making appropriate precision conversions when needed. In this case it may be necessary to link against a larger set of libraries.

3 Function Naming Conventions

3.1 Scalar Example

Function names are constructed with a pattern that suggests their functionality. Thus the function

```
QLA_V_eq_M_times_V(c,u,b)
```

carries out the product

$$c = ub$$

where `c` and `b` are pointers to staggered fermion vectors and `u` is a pointer to a color matrix. The elements of the function name are separated by an underscore "_" for readability. All function names in this interface begin with "`QLA`". The specific name continues with a precision and color label as in

```
QLA_F3_V_eq_M_times_V
```

for single precision $SU(3)$. Then comes a string of elements that mimics the algebraic expression. The next character "`V`" abbreviates the type for the destination operand, in this case the argument "`c`". The abbreviations are listed in Sec. 2. The next string "`eq`" specifies the assignment operator. In this case it is a straight replacement, but increments of various types are also supported, as described below. Then comes the first rhs operand type "M" followed by a string "times" specifying the operation and a character "V" specifying the second rhs operand type.

Supported variants of the assignment operator are tabulated below.

abbreviation	meaning
eq	$=$
peq	$+=$
meq	$-=$
eqm	$=-$

Some functions allow all of these and some take only a simple replacement (=).

3.2 Vector Example

To denote operation on vectors we attach the prefix "v" to the assignment operator and add an argument for the number of elements. So

```
QLA_V_veq_M_times_V(c,u,b,n)
```

does

```
c[i] = u[i]*b[i]
```

for $i = 1 \ldots n$.

3.3 Indirect Addressing for Binary Operations

Various indexing schemes are supported. The table below elaborates the choices for the matrix-vector product example above, including the two illustrated above. In all cases it is assumed $i = 0 \ldots n-1$.

	function name	meaning
1	QLA_V_eq_M_times_V(c,u,b)	c = u*b
2	QLA_V_veq_M_times_V(c,u,b,n)	c[i] = u[i]*b[i]
3	QLA_V_xeq_M_times_V(c,u,b,j,n)	c[j[i]] = u[j[i]]*b[j[i]]
4	QLA_V_eq_xM_times_V(c,u,ju,b,n)	c[i] = u[ju[i]]*b[i]
5	QLA_V_eq_M_times_xV(c,u,b,jb,n)	c[i] = u[i]*b[jb[i]]
6	QLA_xV_eq_M_times_V(c,jc,u,b,n)	c[jc[i]] = u[i]*b[i]
7	QLA_V_eq_xM_times_xV(c,u,ju,b,jb,n)	c[i] = u[ju[i]]*b[jb[i]]
8	QLA_xV_eq_M_times_xV(c,jc,u,b,jb,n)	c[jc[i]] = u[i]*b[jb[i]]
9	QLA_xV_eq_xM_times_V(c,jc,u,b,jb,n)	c[jc[i]] = u[ju[i]]*b[i]
10	QLA_xV_eq_xM_times_xV(c,jc,u,ju,b,jb,n)	c[jc[i]] = u[ju[i]]*b[jb[i]]
11	QLA_V_veq_pM_times_V(c,u,b,n)	c[i] = (*u[i])*b[i]
12	QLA_V_veq_M_times_pV(c,u,b,n)	c[i] = u[i]*(*b[i])
13	QLA_V_veq_pM_times_pV(c,u,b,n)	c[i] = (*u[i])*(*b[i])
14	QLA_V_xeq_pM_times_V(c,u,b,j,n)	c[j[i]] = (*u[j[i]])*b[j[i]]
15	QLA_V_xeq_M_times_pV(c,u,b,j,n)	c[j[i]] = (*u[i])*(*b[j[i]])
16	QLA_V_xeq_pM_times_pV(c,u,b,j,n)	c[j[i]] = (*u[j[i]])*(*b[j[i]])

Gang-indexing variant 3 applies the same indexing scheme to all three arguments simultaneously. The convention is to place an "`x`" before the assignment operator. The indexing variants $4 - 10$ apply an index to each of the operands separately in various combinations. In this case the convention places an "`x`" before the type abbreviation for the variable that is indexed. Variants $11 - 16$ replace the array argument with an array of pointers. The symbol "`p`" before the type abbreviation identifies the affected variable.

3.4 Constant Arguments

As illustrated above, functions generally have vector and scalar variants. In this example, the vector variants require all operands to be indexed. However, in some cases it is desirable to keep some arguments constant - i.e. never subscripted. For example the function

```
QLA_V_veq_c_times_V(c,z,b,n)
```

multiplies an array of color vectors by a complex constant as in

```
c[i] = z*b[i]
```

for $i = 1 \ldots n$. In this case we specify that the argument is constant (nonsubscripted) by converting the type abbreviation to lower case: "`c`".

3.5 Indirect Addressing for Binary Operations with a Constant

When an argument in a binary operation is a constant it is never subscripted by definition. The list of variants is shorter. For example, for multiplication of an SU(N) vector by a constant we have the following.

	function name	meaning
1	`QLA_V_eq_c_times_V(c,z,b)`	`c = z*b`
2	`QLA_V_veq_c_times_V(c,z,b,n)`	`c[i] = z*b[i]`
3	`QLA_V_xeq_c_times_V(c,z,b,j,n)`	`c[j[i]] = z*b[j[i]]`
5	`QLA_V_eq_c_times_xV(c,z,b,jb,n)`	`c[i] = z*b[jb[i]]`
6	`QLA_xV_eq_c_times_V(c,jc,z,b,n)`	`c[jc[i]] = z*b[i]`
8	`QLA_xV_eq_c_times_xV(c,jc,z,b,jb,n)`	`c[jc[i]] = z*b[jb[i]]`
12	`QLA_V_veq_c_times_pV(c,z,b,n)`	`c[i] = z*(*b[i])`
15	`QLA_V_xeq_c_times_pV(c,z,b,j,n)`	`c[j[i]] = z*(*b[j[i]])`

The scalar variant `QLA_V_eq_C_times_V(c,z,b)` is also in the library and is equivalent to variant 1 above.

3.6 Indirect Addressing for Unary Operations

For unary operations the list is also shorter. For example, for copying an SU(N) staggered fermion vector the variants are as follows.

	function name	meaning
1	`QLA_V_eq_V(c,b)`	`c = b`
2	`QLA_V_veq_V(c,b,n)`	`c[i] = b[i]`
3	`QLA_V_xeq_V(c,b,j,n)`	`c[j[i]] = b[j[i]]`
5	`QLA_V_eq_xV(c,b,jb,n)`	`c[i] = b[jb[i]]`
6	`QLA_xV_eq_V(c,jc,b,n)`	`c[jc[i]] = b[i]`
8	`QLA_xV_eq_xV(c,jc,b,jb,n)`	`c[jc[i]] = b[jb[i]]`
12	`QLA_V_veq_pV(c,b,n)`	`c[i] = (*b[i])`
15	`QLA_V_xeq_pV(c,b,j,n)`	`c[j[i]] = (*b[j[i]])`

3.7 Indirect Addressing for Unary Operations with a Constant

For unary operations with a constant rhs the list is even shorter. For example, for setting a staggered fermion vector to zero the variants are as follows.

	function name	meaning
1	`QLA_V_eq_zero(c)`	`c = 0`
2	`QLA_V_veq_V(c,n)`	`c[i] = 0`
3	`QLA_V_xeq_V(c,j,n)`	`c[j[i]] = 0`

3.8 Color argument for SU(N)

For the general color case $SU(N)$ the specific function call requires an extra argument giving the number of colors. It always comes first. Thus in the above example we would write

```
QLA_FN_V_veq_c_times_V(nc,c,z,b,n)
```

where `nc` specifies the number of colors. In normal practice, the value of the argument `nc` should be derived from the required user-defined macro `QLA_Nc` specifying the prevailing number of colors. The generic function is actually a macro and is automatically converted to this usage with `QLA_Nc` for the first argument. However, if the specific name is used, the user must supply the argument.

3.9 Adjoint

The adjoint of an operand is specified by a suffix `a` after the type abbreviation. Thus

```
QLA_F3_V_eq_Ma_times_V(c,u,b)
```

carries out the product

```
c = adjoint(u)*b
```

4 Compilation with QLA

4.1 Generic header and macros

As described above, normally the user selects a prevailing color and precision for the entire calculation. In that case it is permissible to use the generic function names and datatypes, making it possible to change colors and precision with a simple recompilation, if desired. For this purpose the generic header file is `qla.h`. The following macros must be defined by the user prior to including this header file:

required macro	choices
`QLA_Precision`	`'F','D'`
`QLA_Colors`	`2, 3, 'N'`
`QLA_Nc`	n_c if `QLA_Colors= 'N'`

Single quotes are required around nonnumeric values.

A sample code preamble for double precision $SU(3)$ reads

```
#define QLA_Precision 'D'
#define QLA_Colors 3
#include <qla.h>
```

with the include search path set to `$QLA_HOME/include` and `$QLA_HOME` set to the home directory for QLA. With such a preamble the generic function names and datatypes are automatically mapped to the requested specific types. Of course the precision and color macros can also be defined through a compiler flag, as in

```
gcc  -DQLA_Precision=\'D\' -DQLA_Colors=3 ...
```

The single quotes are required and they must each be preceded by a backslash to keep them from being eaten by the shell.

For $SU(4)$ one might do

```
#define QLA_Precision 'F'
#define QLA_Colors 'N'
#define QLA_Nc 4
#include <qla.h>
```

In the current implementation the maximum number of colors is 5, specified by the macro

```
  QLA_MAX_Nc
```

defined in the auxiliary header `qla_types.h`. Changing it is trivial, but requires recompiling the libraries (but not rebuilding the source code).

4.2 Libraries

Normally, it is necessary to link six libraries for a given choice of color and precision. Routines involving only integers, the scalar complex math and random number libraries, and the standard math library are common to all choices. Routines involving only real or complex numbers are common to all colors. Thus for single precision $SU(3)$ the required libraries are linked through

```
  -lqla_int -lqla_f -lqla_f3 -lm -lqla_cmath -lqla_random
```

with the library search path set to `$QLA_HOME/lib`. A complete list of the QLA libraries is given below. Each library module is standalone, except that a few require scalar functions in `qla_cmath`, `qla_random`, and `m`, the standard C math library. If the `round` function is needed and the compiler is not C99-compliant, the library `-lqla_c99` is also required.

name	purpose
`libqla_cmath.a`	double precision `cexp, clog, csqrt, cexpi`
`libqla_random.a`	random number generator
`libqla_c99.a`	round
`libqla_int.a`	integers, boolean
`libqla_f.a`	real, complex, single precision
`libqla_d.a`	real, complex, double precision
`libqla_q.a`	real, complex, extended precision
`libqla_df.a`	real, complex, precision conversion
`libqla_dq.a`	real, complex, precision conversion
`libqla_f3.a`	SU(3), single precision
`libqla_d3.a`	SU(3), double precision
`libqla_q3.a`	SU(3), extended precision
`libqla_df3.a`	SU(3), precision conversion
`libqla_dq3.a`	SU(3), precision conversion
`libqla_f2.a`	SU(2), single precision
`libqla_d2.a`	SU(2), double precision
`libqla_q2.a`	SU(2), extended precision
`libqla_df2.a`	SU(2), precision conversion
`libqla_dq2.a`	SU(2), precision conversion
`libqla_fn.a`	SU(N), single precision
`libqla_dn.a`	SU(N), double precision
`libqla_qn.a`	SU(N), extended precision
`libqla_dfn.a`	SU(N), precision conversion
`libqla_dqn.a`	SU(N), precision conversion

4.3 Nonuniform color and precision

Users wishing to vary color and precision within a single calculation must use specific type names and function names whenever these types and names differ from the prevailing precision and color. For example, if an $SU(3)$ calculation is done primarily in single precision, but has double precision components, the following preamble is appropriate:

```
#define QLA_Precision 'F'
#define QLA_Colors 3
#include <qla.h>
#include <qla_d.h>
#include <qla_d3.h>
```

and the following linkage to get the corresponding libraries:

```
  -lqla_int -lqla_f -lqla_f3 -lqla_d -lqla_df
      -lqla_d3 -lqla_df3 -lqla_cmath -lqla_random -lm
```

As in the previous example, the single precision and type conversion components for $SU(3)$ are automatically included through `qla.h`. Then we need the corresponding double precision components. And we also need the `DF` libraries to do conversions between single and double precision. They, too have colored and noncolored members.

The following table lists all the QLA headers.

name	purpose
`qla.h`	Master header
`qla_cmath.h`	double precision `cexp, clog, csqrt, cexpi`
`qla_random.h`	random number seed and generation
`qla_int.h`	integers, boolean
`qla_f.h`	real, complex, single precision
`qla_d.h`	real, complex, double precision
`qla_q.h`	real, complex, extended precision
`qla_df.h`	real, complex, precision conversion
`qla_dq.h`	real, complex, precision conversion
`qla_f3.h`	SU(3), single precision
`qla_d3.h`	SU(3), double precision
`qla_q3.h`	SU(3), extended precision
`qla_df3.h`	SU(3), precision conversion
`qla_dq3.h`	SU(3), precision conversion
`qla_f2.h`	SU(2), single precision
`qla_d2.h`	SU(2), double precision
`qla_q2.h`	SU(2), extended precision
`qla_df2.h`	SU(2), precision conversion
`qla_dq2.h`	SU(2), precision conversion
`qla_fn.h`	SU(N), single precision
`qla_dn.h`	SU(N), double precision
`qla_qn.h`	SU(N), extended precision
`qla_dfn.h`	SU(N), precision conversion
`qla_dqn.h`	SU(N), precision conversion

5 Auxiliary Features

In addition to the library of linear algebra routines, the QLA interface provides implementation-independent macros for accessing derived data types and for setting color and precision values.

5.1 Accessor macros

The data layout for the composite types is left to the implementation. However, it is then necessary to provide tools for accessing the components of these types. The following table lists macros that an implementer must supply for building the QLA library.

macro
`QLA_`PC`_elem_M(a,ic,jc)`
`QLA_`PC`_elem_H(a,ic,is)`
`QLA_`PC`_elem_D(a,ic,is)`
`QLA_`PC`_elem_V(a,ic)`
`QLA_`PC`_elem_D(a,ic,is,jc,js)`

Both generic and precision-color-specific versions of these macros are provided. Generic names omit the precision-color suffix PC, the symbol P standing for the precision label `Q`, `D` or `F`, and C, the color label `2`, `3`, or `N`.

Here are the definitions of the dummy macro arguments above:

argument	meaning
`a`	name of structure (value, not pointer)
`ic`	color row index
`is`	spin row index
`jc`	color column index
`js`	spin column index

The number of Dirac spins is fixed at four for a full spinor and two for a half-spinor. The number of colors is 2, 3 or `QLA_Nc` for the $SU(N)$ case.

5.2 Complex functions

The following single-variable functions involving the complex type are also provided.

name	meaning
`ComplexD QLA_cexp(ComplexD *a)`	$\exp(a)$
`ComplexD QLA_csqrt(ComplexD *a)`	$\sqrt{a} \quad \arg(a) \in (-\pi, \pi]$
`ComplexD QLA_clog(ComplexD *a)`	$\log(a) \quad \arg(a) \in [0, 2\pi)$
`ComplexD QLA_cexpi(RealD a)`	$\exp(ia)$

5.3 Complex variable macros

The following macros are provided for access and complex arithmetic.

Complex macros with one argument

name	meaning
`QLA_real(c)`	$\mathrm{Re}\, c$
`QLA_imag(c)`	$\mathrm{Im}\, c$
`QLA_arg(c)`	$\arg(c)$
`QLA_norm_c(c)`	$\|c\|$
`QLA_norm2_c(c)`	$\|c\|^2$

Complex macros with two arguments The naming conventions follow closely those for the functions, except that the arguments are all values, rather than pointers.

name	meaning
`QLA_c_eq_r(c,a)`	$c = a$ (real)
`QLA_c_eq_c(c,a)`	$c = a$
`QLA_c_eqm_c(c,a)`	$c = -a$
`QLA_c_eqm_c(c,a)`	$c = -a$
`QLA_c_eqm_r(c,a)`	$c = -a$ (real)
`QLA_c_peq_r(c,a)`	$c = c + a$ (real)
`QLA_c_peq_c(c,a)`	$c = c + a$
`QLA_c_meq_r(c,a)`	$c = c - a$ (real)
`QLA_c_meq_c(c,a)`	$c = c - a$
`QLA_c_meq_c(c,a)`	$c = c - a$
`QLA_c_eq_ca(c,a)`	$c = a^*$
`QLA_c_peq_ca(c,a)`	$c = c + a^*$
`QLA_c_meq_ca(c,a)`	$c = c - a^*$
`QLA_c_eqm_ca(c,a)`	$c = -a^*$
`QLA_r_eq_Re_c(c,a)`	$c = \mathrm{Re}(a)$
`QLA_r_eq_Im_c(c,a)`	$c = \mathrm{Im}(a)$
`QLA_r_peq_Re_c(c,a)`	$c = c + \mathrm{Re}(a)$
`QLA_r_peq_Im_c(c,a)`	$c = c + \mathrm{Im}(a)$
`QLA_r_meq_Re_c(c,a)`	$c = c - \mathrm{Re}(a)$
`QLA_r_meq_Im_c(c,a)`	$c = c - \mathrm{Im}(a)$
`QLA_r_eqm_Re_c(c,a)`	$c = -\mathrm{Re}(a)$
`QLA_r_eqm_Im_c(c,a)`	$c = -\mathrm{Im}(a)$
`QLA_c_eq_ic(c,a)`	$c = ia$
`QLA_c_eqm_ic(c,a)`	$c = -ia$
`QLA_c_peq_ic(c,a)`	$c = c + ia$
`QLA_c_meq_ic(c,a)`	$c = c - ia$
`QLA_c_meq_ic(c,a)`	$c = c - ia$
`QLA_c_eqm_ic(c,a)`	$c = -ia$

Complex macros with three arguments Again, the naming conventions follow closely those for the functions, except that the arguments are all values, rather than pointers.

name	meaning
`QLA_c_eq_c_plus_c(c,a,b)`	$c = a + b$
`QLA_c_eq_c_plus_ic(c,a,b)`	$c = a + ib$
`QLA_c_eq_r_plus_ir(c,a,b)`	$c = a + ib$
`QLA_c_peq_r_plus_ir(c,a,b)`	$c = c + a + ib$
`QLA_c_eqm_r_plus_ir(c,a,b)`	$c = -a + ib$
`QLA_c_meq_r_plus_ir(c,a,b)`	$c = c - a + ib$
`QLA_c_eq_c_minus_c(c,a,b)`	$c = a - b$
`QLA_c_eq_c_minus_ca(c,a,b)`	$c = a - b^*$
`QLA_c_eq_c_minus_c(c,a,b)`	$c = a - ib$
`QLA_c_eq_c_times_c(c,a,b)`	$c = ab$
`QLA_c_peq_c_times_c(c,a,b)`	$c = c + ab$
`QLA_c_eqm_c_times_c(c,a,b)`	$c = -ab$
`QLA_c_meq_c_times_c(c,a,b)`	$c = c - ab$
`QLA_r_eq_Re_c_times_c(c,a,b)`	$c = \mathrm{Re}(ab)$
`QLA_r_peq_Re_c_times_c(c,a,b)`	$c = c + \mathrm{Re}(ab)$
`QLA_r_eqm_Re_c_times_c(c,a,b)`	$c = -\mathrm{Re}(ab)$
`QLA_r_meq_Re_c_times_c(c,a,b)`	$c = c - \mathrm{Re}(ab)$
`QLA_r_eq_Im_c_times_c(c,a,b)`	$c = \mathrm{Im}(ab)$
`QLA_r_peq_Im_c_times_c(c,a,b)`	$c = c + \mathrm{Im}(ab)$
`QLA_r_eqm_Im_c_times_c(c,a,b)`	$c = -\mathrm{Im}(ab)$
`QLA_r_meq_Im_c_times_c(c,a,b)`	$c = c - \mathrm{Im}(ab)$
`QLA_c_eq_c_div_c(c,a,b)`	$c = a/b$
`QLA_c_eq_c_times_ca(c,a,b)`	$c = ab^*$
`QLA_c_peq_c_times_ca(c,a,b)`	$c = c + ab^*$
`QLA_c_eqm_c_times_ca(c,a,b)`	$c = -ab^*$
`QLA_c_meq_c_times_ca(c,a,b)`	$c = c - ab^*$
`QLA_r_eq_Re_c_times_ca(c,a,b)`	$c = \mathrm{Re}(ab^*)$
`QLA_r_peq_Re_c_times_ca(c,a,b)`	$c = c + \mathrm{Re}(ab^*)$
`QLA_r_eqm_Re_c_times_ca(c,a,b)`	$c = -\mathrm{Re}(ab^*)$
`QLA_r_meq_Re_c_times_ca(c,a,b)`	$c = c - \mathrm{Re}(ab^*)$
`QLA_r_eq_Im_c_times_ca(c,a,b)`	$c = \mathrm{Im}(ab^*)$
`QLA_r_peq_Im_c_times_ca(c,a,b)`	$c = c + \mathrm{Im}(ab^*)$
`QLA_r_eqm_Im_c_times_ca(c,a,b)`	$c = -\mathrm{Im}(ab^*)$
`QLA_r_meq_Im_c_times_ca(c,a,b)`	$c = c - \mathrm{Im}(ab^*)$
`QLA_c_eq_ca_times_c(c,a,b)`	$c = a^*b$
`QLA_c_peq_ca_times_c(c,a,b)`	$c = c + a^*b$
`QLA_c_eqm_ca_times_c(c,a,b)`	$c = -a^*b$
`QLA_c_meq_ca_times_c(c,a,b)`	$c = c - a^*b$

name	meaning
`QLA_r_eq_Re_ca_times_c(c,a,b)`	$c = \mathrm{Re}(a^*b)$
`QLA_r_peq_Re_ca_times_c(c,a,b)`	$c = c + \mathrm{Re}(a^*b)$
`QLA_r_eqm_Re_ca_times_c(c,a,b)`	$c = -\mathrm{Re}(a^*b)$
`QLA_r_meq_Re_ca_times_c(c,a,b)`	$c = c - \mathrm{Re}(a^*b)$
`QLA_r_eq_Im_ca_times_c(c,a,b)`	$c = \mathrm{Im}(a^*b)$
`QLA_r_peq_Im_ca_times_c(c,a,b)`	$c = c + \mathrm{Im}(a^*b)$
`QLA_r_eqm_Im_ca_times_c(c,a,b)`	$c = -\mathrm{Im}(a^*b)$
`QLA_r_meq_Im_ca_times_c(c,a,b)`	$c = c - \mathrm{Im}(a^*b)$
`QLA_c_eq_ca_times_ca(c,a,b)`	$c = a^*b^*$
`QLA_c_peq_ca_times_ca(c,a,b)`	$c = c + a^*b^*$
`QLA_c_eqm_ca_times_ca(c,a,b)`	$c = -a^*b^*$
`QLA_c_meq_ca_times_ca(c,a,b)`	$c = c - a^*b^*$
`QLA_r_eq_Re_ca_times_ca(c,a,b)`	$c = \mathrm{Re}(a^*b^*)$
`QLA_r_peq_Re_ca_times_ca(c,a,b)`	$c = c + \mathrm{Re}(a^*b^*)$
`QLA_r_eqm_Re_ca_times_ca(c,a,b)`	$c = -\mathrm{Re}(a^*b^*)$
`QLA_r_meq_Re_ca_times_ca(c,a,b)`	$c = c - \mathrm{Re}(a^*b^*)$
`QLA_r_eq_Im_ca_times_ca(c,a,b)`	$c = \mathrm{Im}(a^*b^*)$
`QLA_r_peq_Im_ca_times_ca(c,a,b)`	$c = c + \mathrm{Im}(a^*b^*)$
`QLA_r_eqm_Im_ca_times_ca(c,a,b)`	$c = -\mathrm{Im}(a^*b^*)$
`QLA_r_meq_Im_ca_times_ca(c,a,b)`	$c = c - \mathrm{Im}(a^*b^*)$
`QLA_c_eq_c_times_r(c,a,b)`	$c = ab$ (b real)
`QLA_c_peq_c_times_r(c,a,b)`	$c = c + ab$ (b real)
`QLA_c_eqm_c_times_r(c,a,b)`	$c = -ab$ (b real)
`QLA_c_meq_c_times_r(c,a,b)`	$c = c - ab$ (b real)
`QLA_c_peq_c_times_r(c,a,b)`	$c = c + ab$ (b real)
`QLA_c_eq_r_times_c(c,a,b)`	$c = ab$ (a real)
`QLA_c_peq_r_times_c(c,a,b)`	$c = c + ab$ (a real)
`QLA_c_eqm_r_times_c(c,a,b)`	$c = -ab$ (a real)
`QLA_c_meq_r_times_c(c,a,b)`	$c = c - ab$ (a real)
`QLA_c_peq_r_times_c(c,a,b)`	$c = c + ab$ (a real)
`QLA_c_eq_c_div_r(c,a,b)`	$c = a/b$ (b real)

Complex macros with four arguments

name	meaning
`QLA_c_eq_c_times_c_plus_c(c,a,x,b)`	$c = ax + b$
`QLA_c_eq_c_times_c_minus_c(c,a,x,b)`	$c = ax - b$
`QLA_c_eq_c_times_r_plus_r(c,a,x,b)`	$c = ax + b$ (x real)
`QLA_c_eq_c_times_r_minus_r(c,a,x,b)`	$c = ax - b$ (x real)
`QLA_c_eq_r_times_c_plus_c(c,a,x,b)`	$c = ax + b$ (a real)
`QLA_c_eq_r_times_c_minus_c(c,a,x,b)`	$c = ax - b$ (a real)

5.4 Gamma matrices

The gamma matrix basis used is:

$$
\begin{aligned}
\gamma_1 &= \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ -i & 0 & 0 & 0 \end{pmatrix} &= \begin{pmatrix} 0 & i\sigma_1 \\ -i\sigma_1 & 0 \end{pmatrix} &= -\sigma_2 \otimes \sigma_1 \\
\gamma_2 &= \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix} &= \begin{pmatrix} 0 & -i\sigma_2 \\ i\sigma_2 & 0 \end{pmatrix} &= \sigma_2 \otimes \sigma_2 \\
\gamma_3 &= \begin{pmatrix} 0 & 0 & i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{pmatrix} &= \begin{pmatrix} 0 & i\sigma_3 \\ -i\sigma_3 & 0 \end{pmatrix} &= -\sigma_2 \otimes \sigma_3 \\
\gamma_4 &= \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} &= \begin{pmatrix} 0 & \mathbf{1} \\ \mathbf{1} & 0 \end{pmatrix} &= \sigma_1 \otimes 1
\end{aligned}
$$

This is a chiral basis and is the same basis currently used in QDP++.

For functions that multiply by a gamma matrix the gamma matrix is specified by an integer between 0 and 15. The mapping from the integer to a general gamma matrix is:

$$\Gamma(n) = \gamma_1^{n_0} \gamma_2^{n_1} \gamma_3^{n_2} \gamma_4^{n_3}$$

where the binary representation of n is $n_3 n_2 n_1 n_0$.

5.5 Random numbers

The generation of random numbers is based on the state of the generator, stored in an implementation-specific structure of type `RandomState`. The following functions are provided for seeding and accessing random numbers.

name

`RealF QLA_random(RandomState *s);`
`RealF QLA_gaussian(RandomState *s);`
`void QLA_seed_random(RandomState *s, Int seed, Int i);`

The first of these returns a random real value uniformly distributed on $[0, 1]$. The second returns a real Gaussian normal deviate $N(0, 1)$. The third seeds a single random number generator from a pair of integers `seed, i`. In typical usage, many generators are maintained. Thus the integer `i` distinguishes them and the integer `seed` seeds it, resulting in the initial state `s`. It is assumed that storage for the state `s` has been allocated prior to the call.

6 Function Details

This section describes in some detail the names and functionality for all functions in the interface. Because of the variety of indexing schemes, datatypes, and assignment operations, there are several thousand names altogether. However, there are only a couple dozen categories. It is hoped that the construction of the names is sufficiently natural that with only a little practice, the user can guess the name of any function and determine its functionality without consulting a list.

6.1 Unary Operations

Elementary unary functions on reals

Syntax	`void QLA_R_`*eqop_func*`_R ( restrict QLA_Real *r, QLA_Real *a)`
Meaning	$r = \text{func}(a)$
func	cos, sin, tan, acos, asin, atan, sqrt, fabs, exp, log, sign, ceil, floor, cosh, sinh, tanh, log10
eqop	`eq`

Elementary unary functions real to complex

Syntax	`void QLA_C_`*eqop_func*`_R ( restrict QLA_Complex *r, QLA_Real *a)`
Meaning	$r = \exp(ia)$
func	cexpi
eqop	`eq`

Elementary unary functions complex to real

Syntax	`void QLA_R_`*eqop_func*`_C ( restrict QLA_Real *r, QLA_Complex *a)`
Meaning	$r = \text{func}(a)$
func	norm, arg
eqop	`eq`

Elementary unary functions on complex values

Syntax	`void QLA_C_`*eqop_func*`_C ( restrict QLA_Complex *r, QLA_Complex *a)`
Meaning	$r = \text{func}(a)$
func	cexp, csqrt, clog
eqop	`eq`

Elementary binary functions on reals

Syntax	`void QLA_R_`*eqop*`_R_`*func*`_R( restrict QLA_Real *r, QLA_Real *a, QLA_Real *b)`
exception	`void QLA_R_`*eqop*`_R_ldexp_I( restrict QLA_Real *r, QLA_Real *a, QLA_Int *b)`
Meaning	$r = \text{func}(a, b)$
func	`mod, max, min, ldexp, pow, atan2`
eqop	`eq`

Elementary binary functions on integers

Syntax	`void QLA_I_`*eqop*`_I_`*op*`_I( restrict` QLA_Int *r, QLA_Int *a, `QLA_Int *b)`
Meaning	$r = \text{func}(a, b)$
op	`lshift, rshift, mod, max, min`
eqop	`eq`

Copying and incrementing

Syntax	`void QLA_`*T*`_`*eqop*`_`*T*`( restrict` *Type* `*r,` *Type* `*a)`
Meaning	$r = a$, etc.
T	`I, R, C, V, H, D, M, P`
eqop	`eq, peq, meq, eqm`

Hermitian conjugate

Syntax	`void QLA_`*T*`_`*eqop*`_`*T*`a( restrict` *Type* `*r,` *Type* `*a)`
Meaning	$r = a^\dagger$, etc.
T	`C, M, P`
eqop	`eq, peq, meq, eqm`

Transpose

Syntax	`void QLA_`*T*`_`*eqop*`_transpose_`*T*`( restrict` *Type* `*r,` *Type* `*a)`
Meaning	$r = \text{transpose}(a)$, etc.
T	`M, P`
eqop	`eq, peq, meq, eqm`

Complex conjugate

Syntax	`void QLA_`*T*`_`*eqop*`_conj_`*T*`( restrict` *Type* `*r,` *Type* `*a)`
Meaning	$r = a^*$, etc.
T	`C, V, H, D, M, P`
eqop	`eq, peq, meq, eqm`

Local squared norm: uniform precision

Syntax	`void QLA_R_`*eqop*`_norm2_`*T*`( restrict QLA_Real  *r,` *Type* `*a)`
Meaning	$r = \|a\|^2$
eqop	`eq`
T	`C, M, H, D, V, P`

6.2 Type conversion and component extraction and insertion

Convert float to double

Syntax	`void QLA_DF_`*T*`_`*eqop*`_`*T*`( restrict`*Type*`_D *r,`*Type*`_F *a)`
Meaning	$r = a$
T	`R, C, V, H, D, M, P`
eqop	`eq`

Convert double to float

Syntax	`void QLA_FD_`*T*`_`*eqop*`_`*T*`( restrict`*Type*`_F *r,`*Type*`_D *a)`
Meaning	$r = a$
T	`R, C, V, H, D, M, P`
eqop	`eq`

Convert long double to double

Syntax	`void QLA_DQ_`*T*`_`*eqop*`_`*T*`( restrict`*Type*`_D *r,`*Type*`_Q *a)`
Meaning	$r = a$
T	`R, C, H, P, V, D, M`
eqop	`eq`

Convert real to complex (zero imaginary part)

Syntax	`void QLA_C_`*eqop*`_R( restrict QLA_Complex *r, QLA_Real *a)`
Meaning	$\mathrm{Re}\, r = a$, $\mathrm{Im}\, r = 0$
eqop	`eq`

Convert real and imaginary to complex

Syntax	`void QLA_C_`*eqop*`_R_plus_i_R( restrict QLA_Complex *r, QLA_Real *a, QLA_Real *b)`
Meaning	$\mathrm{Re}\, r = a$, $\mathrm{Im}\, r = b$
eqop	`eq`

Real part of complex

Syntax	`void QLA_R_`*eqop*`_re_C( restrict QLA_Real  *r, QLA_Complex  *a)`
Meaning	$r = \mathrm{Re}\, a$
eqop	`eq`

Imaginary part of complex

Syntax	`void QLA_R_`*eqop*`_im_C( restrict QLA_Real  *r, QLA_Complex  *a)`
Meaning	$r = \mathrm{Im}\, a$
eqop	`eq`

Integer to real

Syntax	`void QLA_R_`*eqop*`_I( restrict QLA_Real *r, QLA_Int *a)`
Meaning	$r = a$
eqop	`eq`

Real to integer (truncate)

Syntax	`void QLA_I_`*eqop*`_trunc_R ( restrict QLA_Int *r, QLA_Real *a)`
Meaning	$r = a$
eqop	`eq`

Real to integer (round)

Syntax	`void QLA_I_`*eqop*`_round_R ( restrict QLA_Int *r, QLA_Real *a)`
Meaning	`r = a>=0 ?  a+0.5 :  a-0.5`
eqop	`eq`

Accessing a color matrix element

Syntax	`void QLA_C_`*eqop*`_elem_`T`( restrict QLA_Complex *r, Type *a, int i, int j)`
Meaning	$r = a_{i,j}$
T	`M`
eqop	`eq`

Inserting a color matrix element

Syntax	`void QLA_`*T*`_`*eqop*`_elem_C( restrict Type *r, QLA_Complex *a, int i, int j)`
Meaning	$r_{i,j} = a$
T	`M`
eqop	`eq`

Accessing a half fermion or Dirac fermion spinor element

Syntax	`void QLA_C_`*eqop*`_elem_`*T*`( restrict QLA_Complex *r, Type *a, int i_c, int i_s)`
Meaning	$r = a_{i_c,i_s}$
T	`H, D`
eqop	`eq`

Inserting a half fermion or Dirac fermion spinor element

Syntax	`void QLA_`*T*`_`*eqop*`_elem_C( restrict Type *r, QLA_Complex *a, int i_c, int i_s)`
Meaning	$r_{i_c,i_s} = a$
T	`H, D`
eqop	`eq`

Accessing a staggered fermion spinor element

Syntax	`void QLA_C_`*eqop*`_elem_V( restrict QLA_Complex *r, QLA_ColorVector *a, int i)`
Meaning	$r = a_i$
eqop	`eq`

Inserting a staggered fermion spinor element

Syntax	`void QLA_V_`*eqop*`_elem_C( restrict QLA_ColorVector *r, QLA_Complex *a, int i)`
Meaning	$r_i = a$
eqop	`eq`

Accessing a Dirac propagator matrix element

Syntax	`void QLA_C_`*eqop*`_elem_P( restrict QLA_Complex *r, QLA_DiracPropagator *a, int i_c, int i_s, int j_c, int j_s)`
Meaning	$r = a_{i_c,i_s,j_c,j_s}$
eqop	`eq`

Inserting a Dirac propagator matrix element

Syntax	`void QLA_P_`*eqop*`_elem_C( restrict QLA_DiracPropagator *r, QLA_Complex *a, int i_c, int i_s, int j_c, int j_s)`
Meaning	$r_{i_c,i_s,j_c,j_s} = a$
eqop	`eq`

Extracting a color column vector from a color matrix

Syntax	`void QLA_V_`*eqop*`_colorvec_`*T*`( restrict QLA_ColorVector *r, Type *a, int j)`
Meaning	$r_i = a_{i,j} \quad \text{for } i = 0 \ldots n_c - 1$
T	`M`
eqop	`eq`

Inserting a color vector into a color matrix

Syntax	`void QLA_`*T*`_`*eqop*`_colorvec_V( restrict Type *r, QLA_ColorVector *a, int j)`
Meaning	$r_{i,j} = a_i \quad \text{for } i = 0 \ldots n_c - 1$
T	`M`
eqop	`eq`

Extracting a color column vector from a half fermion or Dirac fermion spinor

Syntax	`void QLA_V_`*eqop*`_colorvec_`*T*`( restrict QLA_ColorVector *r, Type *a, int i_s)`
Meaning	$r_{i_c} = a_{i_c,i_s} \quad \text{for } i_c = 0 \ldots n_c - 1$
T	`H, D`
eqop	`eq`

Inserting a color column vector into a half fermion or Dirac fermion spinor

Syntax	`void QLA_`*T*`_`*eqop*`_colorvec_V( restrict Type *r, QLA_ColorVector *a, int i_s)`
Meaning	$r_{i_c,i_s} = a_{i_c} \quad \text{for } i_c = 0 \ldots n_c - 1$
T	`H, D`
eqop	`eq`

Extracting a Dirac column vector from a Dirac propagator matrix

Syntax	`void QLA_D_`*eqop*`_diracvec_P( restrict QLA_DiracFermion *r, QLA_DiracPropagator *a, int jc, int js)`
Meaning	$r_{i_c,i_s} = a_{i_c,i_s,j_c,j_s}$ for $i_c = 0 \dots n_c - 1, i_s = 0 \dots 3$
eqop	`eq`

Inserting a Dirac column vector into a Dirac propagator matrix

Syntax	`void QLA_P_`*eqop*`_diracvec_D( restrict QLA_DiracPropagator *r, QLA_DiracFermion *a, int jc, int js)`
Meaning	$r_{i_c,i_s,j_c,j_s} = a_{i_c,i_s}$ for $i_c = 0 \dots n_c - 1, i_s = 0 \dots 3$
eqop	`eq`

Trace of color matrix

Syntax	`void QLA_C_`*eqop*`_trace_M( restrict QLA_Complex  *r, QLA_ColorMatrix  *a)`
Meaning	$r = \mathrm{Tr}\, a$
eqop	`eq`

Real trace of color matrix

Syntax	`void QLA_R_`*eqop*`_re_trace_M( restrict QLA_Real  *r, QLA_ColorMatrix  *a)`
Meaning	$r = \mathrm{Re}\,\mathrm{Tr}\, a$
eqop	`eq`

Imaginary trace of color matrix

Syntax	`void QLA_R_`*eqop*`_im_trace_M( restrict QLA_Real  *r, QLA_ColorMatrix  *a)`
Meaning	$r = \mathrm{Im}\,\mathrm{Tr}\, a$
eqop	`eq`

Traceless antihermitian part of color matrix

Syntax	`void QLA_M_`*eqop*`_antiherm_M( restrict QLA_ColorMatrix  *r, QLA_ColorMatrix  *a)`
Meaning	$r = (a - a^\dagger)/2 - i\, \mathrm{Im}\,\mathrm{Tr}\, a/n_c$
eqop	`eq`

Spin trace of Dirac propagator

Syntax	`void QLA_M_`*eqop*`_spintrace_P( restrict QLA_ColorMatrix  *r, QLA_DiracPropagator  *a)`
Meaning	$r_{i_c,j_c} = \sum_{i_s} a_{i_c,i_s,j_c,i_s}$
eqop	`eq`

Dirac spin projection

Syntax	`void QLA_H_`*eqop*`_spproj_D( restrict QLA_HalfFermion *r, QLA_DiracFermion *a, int d, int p)`
Meaning	$r = (1 + p\gamma_d)a$
eqop	`eq`

Dirac spin reconstruction

Syntax	`void QLA_D_`*eqop*`_sprecon_H( restrict QLA_DiracFermion *r, QLA_HalfFermion *a, int d, int p)`
Meaning	$r = \mathrm{recon}\,(p, d, a)$
eqop	`eq`

6.3 Binary Operations with Constants

Multiplication by real constant

Syntax	`void QLA_`*T*`_`*eqop*`_r_times_`*T*`( restrict`*Type*` *r, QLA_Real *a,` *Type* `*b)`
exception	`void QLA_I_`*eqop*`_i_times_I( restrict QLA_Int *r, QLA_Int *a, QLA_Int *b)`
Meaning	$r = a * b$, etc. (a real, constant)
T	`I, R, C, V, H, D, M, P`
eqop	`eq, peq, meq, eqm`

Multiplication by complex constant

Syntax	`void QLA_`*T*`_`*eqop*`_c_times_`*T*`( restrict`*Type*` *r, QLA_Complex *a,` *Type* `*b)`
Meaning	$r = a * b$, etc. (a complex, constant)
T	`C, V, H, D, M, P`
eqop	`eq, peq, meq, eqm`

Multiplication by i

Syntax	`void QLA_`*T*`_`*eqop*`_i_`*T*`( restrict`*Type* `*r, QLA_Complex *a,` *Type* `*b)`
Meaning	$r = i * a$, etc.
T	`C, V, H, D, M, P`
eqop	`eq, peq, meq, eqm`

Left multiplication by gamma matrix

Syntax	`void QLA_`*T*`_`*eqop*`_gamma_times_`*T*`( restrict`*Type* `*r,` *Type* `*a, int d)`
Meaning	$r = \gamma_d a$
T	`D, P`
eqop	`eq`

Right multiplication by gamma matrix

Syntax	`void QLA_P_`*eqop*`_P_times_gamma( restrict`*Type* `*r,` *Type* `*a, int d)`
Meaning	$r = a\gamma_d$
eqop	`eq`

6.4 Binary Operations with Fields

Division of real, complex, and integer fields

Syntax	`void QLA_R_`*eqop*`_R_divide_R( restrict QLA_Real *r, QLA_Real *a, QLA_Real *b)`
Meaning	$r = a/b$
eqop	`eq`

Syntax	`void QLA_C_`*eqop*`_C_divide_C( restrict QLA_Complex *r, QLA_Complex *a, QLA_Complex *b)`
Meaning	$r = a/b$
eqop	`eq`

Syntax	`void QLA_I_`*eqop*`_I_divide_I( restrict QLA_Int *r, QLA_Int *a, QLA_Int *b)`
Meaning	$r = a/b$
eqop	`eq`

Addition

Syntax	`void QLA_`*T*`_`*eqop*`_`*T*`_plus_`*T*`( restrict`*Type* `*r,` *Type* `*a,` *Type* `*b)`
Meaning	$r = a + b$
T	`I, R, C, V, H, D, M, P`
eqop	`eq`

Subtraction

Syntax	`void QLA_`*T*`_`*eqop*`_`*T*`_minus_`*T*`( restrict`*Type* `*r,` *Type* `*a,` *Type* `*b)`
Meaning	$r = a - b$
T	`I, R, C, V, H, D, M, P`
eqop	`eq`

Multiplication: uniform types

Syntax	`void QLA_`*T*`_`*eqop*`_`*T*`_times_`*T*`( restrict`*Type* `*r,` *Type* `*a,` *Type* `*b)`
Meaning	$r = a * b$, etc.
T	`I, R, C, M, P`
eqop	`eq, peq, meq, eqm`

Local inner product

Syntax	`void QLA_C_`*eqop*`_`*T*`_dot_`*T*`( restrict QLA_Complex *r,` *Type *a, Type* `*b)`
exception	`void QLA_R_`*eqop*`_R_dot_R( restrict QLA_Real *r, QLA_Real *a, QLA_Real *b)` `void QLA_R_`*eqop*`_I_dot_I( restrict QLA_Real *r, QLA_Int *a, QLA_Int *b)`
Meaning	$r_i = \mathrm{Tr}\, a_i^\dagger \cdot b_i$
T	`C, M, H, D, V, P`
eqop	`eq`

Syntax	`void QLA_R_`*eqop*`_re_`*T*`_dot_`*T*`( restrict QLA_Complex *r,` *Type*`*a,` *Type* `*b)`
Meaning	$r_i = \mathrm{Re}\,\mathrm{Tr}\, a_i^\dagger \cdot b_i$
T	`C, V, H, D, M, P`
eqop	`eq`

Color matrix from outer product

Syntax	`void QLA_`*M*`_`*eqop*`_V_times_Va( restrictQLA_ColorMatrix *r, QLA_ColorVector *a, QLA_ColorVector *b)`
Meaning	$r_{i,j} = a_i * b_j^*$, etc.
eqop	`eq, peq, meq, eqm`

Left multiplication by color matrix

Syntax	`void QLA_`*T*`_`*eqop*`_M_times_`*T*`( restrict`*Type* `*r, QLA_ColorMatrix *a,` *Type* `*b)`
Meaning	$r = a * b$, etc.
T	`V, H, D, M, P`
eqop	`eq, peq, meq, eqm`

Adjoint of color matrix times adjoint of color matrix

Syntax	`void QLA_M_`*eqop*`_Ma_times_Ma ( restrict QLA_ColorMatrix *r, QLA_ColorMatrix *a, QLA_ColorMatrix *b)`
Meaning	$r = a^\dagger * b^\dagger$
eqop	`eq, peq, meq, eqm`

Left multiplication by adjoint of color matrix

Syntax	`void QLA_`*T*`_`*eqop*`_Ma_times_`*T*`( restrict`*Type* `*r, QLA_ColorMatrix *a,` *Type* `*b)`
Meaning	$r = a * b$, etc.
T	`V, H, D, M, P`
eqop	`eq, peq, meq, eqm`

Right multiplication by color matrix

Syntax	`void QLA_`*T*`_`*eqop*`_`*T*`_times_M ( restrict`*Type* `*r, QLA_ColorMatrix *a,` *Type* `*b)`
Meaning	$r = a * b$, etc.
T	`M, P`
eqop	`eq, peq, meq, eqm`

Right multiplication by adjoint of color matrix

Syntax	`void QLA_`*T*`_`*eqop*`_`*T*`_times_Ma ( restrict`*Type* `*r, QLA_ColorMatrix *a,` *Type* `*b)`
Meaning	$r = a * b^\dagger$, etc.
T	`M, P`
eqop	`eq, peq, meq, eqm`

6.5 Ternary Operations with Fields

Addition with real scalar multiplication

Syntax	`void QLA_`*T*`_`*eqop*`_r_times_`*T*`_plus`*T*`( restrict`*Type* `*r, QLA_Real *a,` *Type* `*b,` *Type* `*c)`
exception	`void QLA_I_`*eqop*`_i_times_I_plusI( restrict QLA_Int *r, QLA_Int *a, QLA_Int *b, QLA_Int *c)`
Meaning	$r = a * b + c$, etc.
T	`I, R, C, V, H, D, M, P`
eqop	`eq`

Subtraction with real scalar multiplication

Syntax	`void QLA_`*T*`_`*eqop*`_r_times_`*T*`_minus`*T*`( restrict`*Type* `*r, QLA_Real *a,` *Type* `*b,` *Type* `*c)`
exception	`void QLA_I_`*eqop*`_i_times_I_minus_I( restrict QLA_Int *r, QLA_Int *a, QLA_Int *b, QLA_Int *c)`
Meaning	$r = a * b - c$, etc.
T	`I, R, C, V, H, D, M, P`
eqop	`eq`

Addition with complex scalar multiplication

Syntax	`void QLA_`*T*`_`*eqop*`_c_times_`*T*`_plus`*T*`( restrict`*Type* `*r, QLA_Complex *a,` *Type* `*b,` *Type* `*c)`
Meaning	$r = a * b + c$, etc.
T	`C, V, H, D, M, P`
eqop	`eq`

Subtraction with complex scalar multiplication

Syntax	`void QLA_`*T*`_`*eqop*`_c_times_`*T*`_minus`*T*`( restrict`*Type* `*r, QLA_Complex *a,` *Type* `*b,` *Type* `*c)`
Meaning	$r = a * b - c$, etc.
T	`C, V, H, D, M, P`
eqop	`eq`

6.6 Boolean and Bit Operations

Comparisons of integers and reals

Syntax	`void QLA_I_`*eqop*`_`*T*`_`*op*`_`*T*`( restrict` *Type* `*r,` *Type*`*a,` *Type* `*b)`
Meaning	$r = \mathrm{op}(a, b)$
T	`I, R`
op	`eq, ne, gt, lt, ge, le`
eqop	`eq`

Boolean Operations

Syntax	`void QLA_I_`*eqop*`_I_`*op*`_I( restrict` QLA_Int *r, QLA_Int *a, QLA_Int *b)
Meaning	$r = a\,\mathrm{op}\,b$
op	`or, and, xor`
eqop	`eq`

Syntax	`void QLA_I_`*eqop*`_not_I( restrict` QLA_Int *r, QLA_Int *a)
Meaning	$r = \mathrm{not}\,a$
eqop	`eq`

Copymask

Syntax	`void QLA_`*T*`_`*eqop*`_`*T*`_mask_I( restrict` *Type* `*r,` *Type* `*a, QLA_Int *b)`
Meaning	$r = b$ if a is true
T	`I, R, C, V, H, D, M, P`
eqop	`eq`

6.7 Reductions

Global squared norm: uniform precision

Syntax	`void QLA_r_`*eqop*`_norm2_`*T*`( restrict QLA_Real *r,` *Type* `*a)`
Meaning	$r = \sum \lvert a\rvert^2$
eqop	`eq`
T	`I, R, C, V, H, D, M, P`

Global squared norm: float to double

Syntax	`void QLA_DF_r_`*eqop*`_norm2_`*T*`( restrict QLA_Real _D *r,` *Type*`_F *a)`
Meaning	$r = \sum \lvert a\rvert^2$
eqop	`eq`
T	`I, R, C, V, H, D, M, P`

Global squared norm: double to long double

Syntax	`void QLA_QD_r_`*eqop*`_norm2_`*T*`( restrict QLA_Real _Q *r,` *Type*`_D *a)`
Meaning	$r = \sum \lvert a \rvert^2$
eqop	`eq`
T	`I, R, C, V, H, D, M, P`

Global inner product

Syntax	`void QLA_c_`*eqop*`_`*T*`_dot_`*T*`( restrict QLA_Complex *r,` *Type *a, Type* `*b)`

exception	`void QLA_r_`*eqop*`_R_dot_R( restrict QLA_Real *r, QLA_Real *a, QLA_Real *b)`
	 `void QLA_r_`*eqop*`_R_dot_R( restrict QLA_Real *r, QLA_Int *a, QLA_Int *b)`

Meaning	$r = \sum \mathrm{Tr}\, a^\dagger \cdot b$
T	`I, R, C, V, H, D, M, P`
eqop	`eq`

Syntax	`void QLA_r_`*eqop*`_re_`*T*`_dot_`*T*`( restrict QLA_Complex *r,` *Type*`*a,` *Type* `*b)`
Meaning	$r = \mathrm{Re} \sum \mathrm{Tr}\, a^\dagger \cdot b$
T	`C, H, D, V, P, M`
eqop	`eq`

Global inner product: float to double

Syntax	`void QLA_DF_c_`*eqop*`_`*T*`_dot_`*T*`( restrict QLA_Complex _D *r,` *Type*`_F *a,` *Type*`_F *b)`
exception	`void QLA_DF_r_`*eqop*`_R_dot_R( restrict QLA_Real _D *r, QLA_Real _F *a, QLA_Real _F *b)`
Meaning	$r = \sum \mathrm{Tr}\, a^\dagger \cdot b$
T	`R, C, M, H, D, V, P`
eqop	`eq`

Syntax	`void QLA_DF_r_`*eqop*`_re_`*T*`_dot_`*T*`( restrict QLA_Complex _D *r,` *Type*`_F *a,` *Type*`_F *b)`
Meaning	$r = \mathrm{Re} \sum \mathrm{Tr}\, a^\dagger \cdot b$
T	`C, M, H, D, V, P`
eqop	`eq`

Global inner product: double to long double

Syntax	void QLA_QD_c_eqop_T_dot_T(restrict QLA_Complex _Q *r, Type_D *a, Type_D *b)
exception	void QLA_QD_r_eqop_R_dot_R(restrict QLA_Real _Q *r, QLA_Real _D *a, QLA_Real _D *b)
Meaning	$r = \sum \mathrm{Tr}\, a^\dagger \cdot b$
T	R, C, M, H, D, V, P
eqop	eq

Syntax	void QLA_QD_r_eqop_re_T_dot_T(restrict QLA_Complex _Q *r, Type_D *a, Type_D *b)
Meaning	$r = \mathrm{Re} \sum \mathrm{Tr}\, a^\dagger \cdot b$
T	C, M, H, D, V, P
eqop	eq

Global sums

Syntax	void QLA_t_eqop_sum_T(restrict Type *r, Type *a)
exception	void QLA_r_eqop_sum_I(restrict QLA_Real *r, QLA_Int *a)
Meaning	$r = \sum a$
T	I, R, C, V, H, D, M, P
eqop	eq

Global sums: float to double

Syntax	void QLA_DF_t_eqop_sum_T(restrict QLA_Type_D *r, Type_F *a)
Meaning	$r = \sum a$
T	I, R, C, V, H, D, M, P
eqop	eq

Global sums: double to long double

Syntax	void QLA_QD_t_eqop_sum_T(restrict QLA_Type_Q *r, Type_D *a)
Meaning	$r = \sum a$
T	I, R, C, V, H, D, M, P
eqop	eq

The scalar variants of the reduction functions are equivalent to an assignment.

6.8 Fills

Zero fills

Syntax	`void QLA_`*T*`_`*eqop*`_zero(`*Type* `*r)`
Meaning	$r = 0$
T	`I, R, C, V, H, D, M, P`
eqop	`eq`

Constant fills and random numbers

Syntax	`void QLA_`*T*`_`*eqop*`_`*t*`( restrict` *Type* `*r,`*Type* `*a)`
Meaning	$r = a$ (constant)
T	`I, R, C, V, H, D, M, P`
eqop	`eq`

Syntax	`void QLA_M_`*eqop*`_c ( restrict QLA_ColorMatrix *r,`*QLA_Type* `*a)`
Meaning	$r = \text{diag}(a, a, \ldots)$ (constant a)
eqop	`eq`

Uniform random number fills

Syntax	`void QLA_R_`*eqop*`_random_S ( restrict QLA_Real *r, QLA_RandomState *a)`
Meaning	r random, uniform on $[0, 1]$
eqop	`eq`

Gaussian random number fills

Syntax	`void QLA_`*T*`_`*eqop*`_gaussian_S ( restrict` *Type* `*r, QLA_RandomState *a)`
Meaning	r normal Gaussian
T	`R, C, V, H, D, M, P`
eqop	`eq`

Seeding the random number generator field from an integer field

Syntax	`void QLA_S_`*eqop*`_seed_i_I (`*Type* QLA_RandomState *r, int seed, `QLA_Int *a)`
Meaning	seed r from field a and constant `seed`
eqop	`eq`

For details see the discussion of the corresponding scalar function `qla_random.h`.