This short note summarizes the requirements for a level 1 linear algebra API that provides operations on single site and arrays (lists) of sites. It is intended to be the low level routines for a higher level structure, for example the level 2 interface.

The API is intended to be sufficiently flexible for all Lattice QCD applications, and execute efficiently on all existing and anticipated platforms.

**Capability Requirements**

1. Lattice objects have a unique (internally known) data-layout.
2. There should be single site only calls. Could just be CPP macro calls around the `vector` calls, namely those that run over an array of sites.
3. There should be vector calls that run over an array of sites.
4. There should be no reference or knowledge of the underlying lattice.
5. To support communications at higher levels, must allow gather and scatter variants of vector calls that have an auxiliary index array passed to the routines.
6. Should support with some type names both Wilson-like and Staggered-like fermions, namely 4 spin component and 1-spin component fermions, respectively.
7. Support SU(3) and SU(2) (????)

**API Design: Performance Requirements**

1. Operations do not (by default) make conversions of objects from one layout into another layout so as to perform the operation. An error is generated if objects are in incompatible layouts.

**API Open Issues**

1. How to deal with potential choices of data ordering, like row-major or column-major. Could generate code in both representations.
2. What about Domain wall fermions?
3. TO DO: fix up for staggered fermions
4. Should memory management routines be exposed at this level?

**Hardware Issues**

Want to have high cache locality
1. For vector calls, data should be contiguous in memory.
API

Data Types
Mathematically, the primitive types we want at a site are either simple scalar values, or live within the product space of a vector space over color components with a vector space over spin components. So, we want generically

- Gauge fields: Product(Matrix(Nc),Scalar)
- Fermions: Product(Vector(Nc),Vector(Ns))
- Scalars: Scalar
- Propagators: Product(Matrix(Nc),Matrix(Ns))

where Nc indicates the dimension of the color vector space and Ns indicates the dimension of the spin vector space. For example, Scalars can multiply gauge fields or fermions. A gamma matrix can multiply a fermion, but not a gauge field. A gauge field can multiply a fermion, but a fermion cannot multiply another fermion unless there is a conjugation involved. For example, a fermion can be laid out in memory as a Nc x Ns matrix, but mathematically a fermion is not equivalent to a matrix.

Given these motivations, the following data-types are supported:

- **ComplexF** single precision complex real (total of 8 bytes.)
- **SU2F** single precision SU(2) matrix (16 bytes)
- **GaugeF** SU(N) single precision matrix (72 bytes for N=3.)
- **RealF** scalar real, single precision (4 bytes)
- **IntF** scalar integer (4 bytes)
- **BooleanF** scalar Boolean (unfortunately, still 4 bytes???)
- **HalfFermionF** two spin component SU(N) fundamental fermion, single precision (48 bytes N=3.)
- **DiracFermionF** four spin component SU(N) Dirac fermion (96 bytes N=3)
- **StaggeredFermionF** one spin component SU(N) staggered fermion (24 bytes N=3)
- **PropagatorF** SU(N) 12 source color-spin and 12 sink color-spin components, single precision (576 bytes N=3).

Single precision versions have a trailing F and double precision versions have a trailing D.

Naming conventions
All library routines begin with QCDF for single precision and QCDD for double precision. If relevant for the particular operation, the next letter is the N in SU(N). The type names above are shortened for compactness:

- **R** scalar RealF
- **I** scalar IntF
- **G** GaugeF
- **H** HalfFermionF
- **D** DiracFermionF
- **S** StaggeredFermionF
- **P** PropagatorF

(should DiracFermionF be a WilsonFermion and then can have DomainWallFermionF ???)

The destination is the first argument and the sources come next.

The trailing character:

- **S** a single site variant
- **V** vector variant, namely runs over a list of elements up to some maximum
- **C1** Index (gather) with suitable integer index array for the first source argument
- **C2** Index (gather) with suitable integer index array for the second source argument
- **CX** Gather on both first and second argument
- **PC1** Gather on first argument, and a scatter on the destination
- **PC2** Gather on second argument, and a scatter on the destination
- **PCX** Gather on both first and second argument, and a scatter on the destination

Only a few routines come in the full blown C and PC variants.

Routines are grouped by the operation rather than the type to emphasize commonality and possible overloading.

**Complex and Scalar numbers**

**Accessors**

Here are accessor routines along with simple algebra

```c
void QCDFmkCS(ComplexF *res, RealF re, RealF im)
void QCDFmkCV(ComplexF *res, const RealF *re, const RealF *im, int num)
```

Packs two reals into one complex.
void QCDFgetCReS(RealF *re, const ComplexF *c)
void QCDFgetCReV(RealF *re, const ComplexF *c, int num)
    Get the real part of the complex number.

void QCDFgetClmS(RealF *im, const ComplexF *c)
void QCDFgetClmV(RealF *im, const ComplexF *c, int num)
    Get the imaginary part of the complex number.

Norms and Inner Products

Norms
void QCDFnorm2TS(RealF *r, const Type *a)
void QCDFnorm2TV(RealF *r, const Type *a, int num)
    Computes \( r = |a|^2 \) where \( T \) is a shortened type name for the type Type
    RealF, ComplexF, GaugeF, DiracF

void QCDFnorm2addTS(RealF *r, const Type *a)
void QCDFnorm2addTV(RealF *r, const Type *a, int num)
    Computes \( r = r + |a|^2 \) where \( T \) is a shortened type name for the type Type
    RealF, ComplexF, GaugeF, DiracF

Norms of integers are also useful.
void QCDFnorm2IS(IntF *r, const IntF *a)
void QCDFnorm2IV(IntF *r, const IntF *a, int num)
    Computes \( r = |a|^2 \)

Inner Products
void QCDFdotTS(ComplexF *r, const Type *a)
void QCDFdotTV(ComplexF *r, const Type *a, int num)
    Computes \( r = \sum_{n,x} \text{conj}(a_n(x))*b_n(x) \) where \( T \) is a shortened type name for the type Type
    ComplexF, GaugeF, DiracF

Division
Division is only supported on very basic fields. This is not something to encourage using
since the inverse of a field can be used for multiplication which is typically more efficient
void QCDFdivRS(RealF *r, const RealF *a, const RealF *b)
void QCDFdivRV(RealF *r, const RealF *a, const RealF *b, int num)
    \( r = a / b. \)
Addition and Subtraction
Basic addition and subtraction of two fields into a third field

```c
void QCDFmult\text{Top2S}(\text{Type}\ r, \text{const Type}\ a, \text{const Type}\ b) 
void QCDFmult\text{Top2V}(\text{Type}\ r, \text{const Type}\ a, \text{const Type}\ b, \text{int num})
```

Binary operations where $T$ is a shortened type name for the type $\text{Type}$
- $\text{RealF}$, $\text{IntF}$, $\text{ComplexF}$, $\text{GaugeF}$, $\text{DiracFermionF}$, $\text{HalfFermion}$, $\text{PropagatorF}$

and $op2$ is chosen from below where the types must be conforming, e.g. a gauge field adds to a gauge field to produce a gauge field

- $\text{add } r = a + b$
- $\text{sub } r = a - b$

Copying
Copying onto a target with optional replacement or addition is a special case of Addition above, but is more optimal. Note, with the C1 variant a communication is possible

```c
void QCDFcp\text{Top2S}(\text{Type}\ r, \text{const Type}\ a) 
void QCDFcp\text{Top2V}(\text{Type}\ r, \text{const Type}\ a, \text{int num})
void QCDFcp\text{Top2C1}(\text{Type}\ r, \text{const Type}\ a, \text{int *ind1, int num}) 
void QCDFcp\text{Top2PC1}(\text{Type}\ r, \text{int *indr, const Type}\ a, \text{int *ind1, int num})
```

Binary operations where $T$ is a shortened type name for the type $\text{Type}$
- $\text{RealF}$, $\text{IntF}$, $\text{ComplexF}$, $\text{GaugeF}$, $\text{DiracFermionF}$, $\text{HalfFermion}$,
  $\text{StaggeredFermionF}$, $\text{PropagatorF}$

and $op2$ is chosen from

- $\text{rep } r = a$
- $\text{neg } r = -a$
- $\text{add } r = r + a$
- $\text{sub } r = r - a$

Conjugation
Stand-alone conjugation

```c
void QCDFconj\text{Top2S}(\text{Type}\ r, \text{const Type}\ a) 
void QCDFconj\text{Top2V}(\text{Type}\ r, \text{const Type}\ a, \text{int num})
```

Binary operations where $T$ is a shortened type name for the type $\text{Type}$
- $\text{ComplexF}$, $\text{GaugeF}$, $\text{PropagatorF}$

and $op2$ is chosen from

- $\text{rep } r = \text{conj}(a)$
- $\text{neg } r = -\text{conj}(a)$
- $\text{add } r = r + \text{conj}(a)$
- $\text{sub } r = r - \text{conj}(a)$
Multiplication

Multiplication is a fundamental operation and quite important for speed. It should support the many variants

```c
void QCDFmultT1T2T3op3S(Type3 *r, const Type1 *a, const Type2 *b)
void QCDFmultT1T2T3op3V(Type3 *r, const Type1 *a, const Type2 *b, int num)
void QCDFmultT1T2T3op3C1 (Type3 *r, const Type1 *a, int *ind1, const Type2 *b, int num)
void QCDFmultT1T2T3op3C2 (Type3 *r, const Type1 *a, const Type2 *b, int *ind2, int num)
void QCDFmultT1T2T3op3CX (Type3 *r, const Type1 *a, int *ind1, const Type2 *b, int *ind2, int num)
void QCDFmultT1T2T3op3PC1 (Type3 *r, const int *indr, const Type1 *a, const Type2 *b, int num)
void QCDFmultT1T2T3op3PC2 (Type3 *r, const int *indr, const Type1 *a, const Type2 *b, const int *ind2, int num)
void QCDFmultT1T2T3op3PCX (Type3 *r, const int *indr, const Type1 *a, const int *ind1, const Type2 *b, int *ind2, int num)
```

Triadic operations where $T_1$, $T_2$, $T_3$ are shortened type names for the type $Type1$, $Type2$ and $Type3$

- GaugeF, DiracFermionF, HalfFermionF, PropagatorF
- and $op3$ is chosen from below where the types must be conforming, e.g. a gauge field multiplies a dirac field, or a $conj(GaugeF)*DiracFermionF$ is required. Note, only $conj(GaugeF)$ is supported

```
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>nnn</td>
<td>r = -a*b</td>
<td>ncr</td>
<td>r = a*conj(b)</td>
</tr>
<tr>
<td>ncr</td>
<td>r = a*conj(b)</td>
<td>nnn</td>
<td>r = -a*b</td>
</tr>
<tr>
<td>cnr</td>
<td>r = conj(a)*b</td>
<td>ncn</td>
<td>r = -a*conj(b)</td>
</tr>
<tr>
<td>ccr</td>
<td>r = conj(a)*conj(b)</td>
<td>cnn</td>
<td>r = -conj(a)*b</td>
</tr>
<tr>
<td>nna</td>
<td>r = r + a*b</td>
<td>nns</td>
<td>r = r – a*b</td>
</tr>
<tr>
<td>nca</td>
<td>r = r + a*conj(b)</td>
<td>ncs</td>
<td>r = r – a*conj(b)</td>
</tr>
<tr>
<td>cna</td>
<td>r = r + conj(a)*b</td>
<td>cna</td>
<td>r = r – conj(a)*b</td>
</tr>
<tr>
<td>cca</td>
<td>r = r + conj(a)*conj(b)</td>
<td>ccs</td>
<td>r = r - conj(a)*conj(b)</td>
</tr>
</tbody>
</table>
```

For the types RealF and IntF, there is not a huge need for communication variants and certainly conjugation variants are not needed, so

```c
void QCDFmultTop3S(Type *r, const Type *a, const Type *b)
```
void QCDFmult\(\text{Top3V}(\text{Type } *r, \text{const Type } *a, \text{const Type } *b, \text{int num})\)

Triadic operations where \(T\) is a shortened type name for the type \(\text{Type}\)
RealF and IntF

and \(\text{op3}\) is

\[
\begin{align*}
nnr & \quad r = a\times b & \quad nnn & \quad r = -a\times b \\
nna & \quad r = r + a\times b & \quad nns & \quad r = r - a\times b
\end{align*}
\]

There is also need for cross type multiplication of RealF and ComplexF against other types
void QCDFmult\(\text{T1T2T2op3S}(\text{Type}2 *r, \text{const Type1 } *a, \text{const Type2 } *b)\)
void QCDFmult\(\text{T1T2T2op3V}(\text{Type}2 *r, \text{const Type1 } *a, \text{const Type2 } *b, \text{int num})\)

Triadic operations where \(T1\) is a shortened type name for
RealF, ComplexF

and \(T2\) is a shortened type name for the type \(\text{Type}2\)
GaugeF, DiracFermionF, HalfFermionF, PropagatorF

and \(\text{op3}\) is chosen from below where now the second source and the target types must
be the same. Note, the case of \(\text{conj}(\text{RealF})\) is not supported

\[
\begin{align*}
mnnr & \quad r = a\times b & \quad mnnn & \quad r = -a\times b \\
mcnr & \quad r = \text{conj}(a)\times b & \quad mcnn & \quad r = -\text{conj}(a)\times b \\
mnna & \quad r = r + a\times b & \quad mnns & \quad r = r - a\times b \\
mcna & \quad r = r + \text{conj}(a)\times b & \quad mcna & \quad r = r - \text{conj}(a)\times b
\end{align*}
\]

**Filling**

void QCDFzero\(\text{TS}(\text{Type } *r)\)
void QCDFzero\(\text{TV}(\text{Type } *r, \text{int num})\)

Fill \(r\) with the 0 field (e.g., a matrix) where \(T\) is a shortened type name for the type \(\text{Type}\)
RealF, IntF, ComplexF, GaugeF, DiracF, PropagatorF

void QCDFone\(\text{TS}(\text{Type } *r)\)
void QCDFone\(\text{TV}(\text{Type } *r, \text{int num})\)

Fill \(r\) with the identity field (e.g., a matrix) where \(T\) is a shortened type name for the type \(\text{Type}\)
RealF, IntF, ComplexF, GaugeF

void QCDFi\(\text{TS}(\text{Type } *r)\)
void QCDFi\(\text{TV}(\text{Type } *r, \text{int num})\)
Fill r with the imaginary I times the identity field (maybe a matrix) [useful in HMD] where \( T \) is a shortened type name for the type **Type**

- ComplexF, GaugeF

```c
void QCDFrandomTS(Type *r)
void QCDFrandomTV(Type *r, int num)
```

- Fill all elements with uniform \([0,1]\) random numbers where \( T \) is a shortened type name for the type **Type**
  - RealF, ComplexF, GaugeF, DiracFermionF

```c
void QCDFgaussianTS(Type *r)
void QCDFgaussianTV(Type *r, int num)
```

- Fill all elements with normal \( N(0,1) \) gaussian random numbers where \( T \) is a shortened type name for the type **Type**
  - RealF, ComplexF, GaugeF, DiracFermionF

**Note,** need ways to set and read random number seeds, specify random numbers, etc.

**Trace**

```c
void QCDFretrGS(RealF *r, const GaugeF *a)
void QCDFretrGV(RealF *r, const GaugeF *a, int num)
```

- Compute \( r = \text{ReTr}(a) \)

```c
void QCDFimtrGS(RealF *r, const GaugeF *a)
void QCDFimtrGV(RealF *r, const GaugeF *a, int num)
```

- Compute \( r = \text{ImTr}(a) \)

```c
void QCDFtrGS(ComplexF *r, const GaugeF *a)
void QCDFtrGV(ComplexF *r, const GaugeF *a, int num)
```

- Compute complex part of trace \( r = \text{Tr}(a) \)

**Spin Product**

Multiply Dirac fermion by a product of gamma-matrices and store the result. \( \{idx\} \) is interpreted according to the following scheme:

The indexing scheme for the gamma matrices is as follows. There are four matrices, \( \text{gamma}_1, \text{gamma}_2, \text{gamma}_3 \) and \( \text{gamma}_4 \), and the basis of 16 matrices

\( 1, \text{gamma}_1, \ldots, \text{gamma}_5 \) is indexed by an integer \( d \)
\[ \gamma^d = \gamma_1^{b_0} \gamma_2^{b_1} \gamma_3^{b_2} \gamma_4^{b_3} \]
\[ d = 8b_3 + 4b_2 + 2b_1 + b_0. \]

Note, this construction is gamma matrix basis independent.

\[
\text{void QCDFspprodTopS(Type *r, const Type *a, int d)}
\]
\[
\text{void QCDFspprodTopV(Type *r, const Type *a, int d, int num)}
\]
\[
\text{r = Gamma^d * a where Type is DiracFermionF, PropagatorF}
\]
\[
\text{where op is}
\]
\[
\begin{array}{ll}
\text{rep} & \text{r = Gamma^d * a} \\
\text{neg} & \text{r = -Gamma^d * a} \\
\text{add} & \text{r = r + Gamma^d * a} \\
\text{sub} & \text{r = r - Gamma^d * a}
\end{array}
\]

**Spin Projection and Reconstruction**

The spin projection trick is useful for Wilson-like fermions, namely 4 spin component fermions

*NOTE: WANT MANY MU VERSION AS WELL??*

\[
\text{void QCDFspprojHDS(HalfFermionF *r, const DiracFermionF *a, int mu)}
\]
\[
\text{void QCDFspprojHDV(HalfFermionF *r, const DiracFermionF *a, int mu, int num)}
\]
\[
\text{r = spin_project(a, mu) -> (1+gamma_mu)*a}
\]

\[
\text{void QCDFspreconHDS(DiracFermionF *r, const HalfFermionF *a, int mu)}
\]
\[
\text{void QCDFspreconHDV(DiracFermionF *r, const HalfFermionF *a, int mu, int num)}
\]
\[
\text{r = spin_reconstruct(a, mu)}
\]

\[
\text{void QCDFspprojHUopDS(HalfFermionF *r, const GaugeF *u, const DiracFermionF *a, int mu)}
\]
\[
\text{void QCDFspprojHUopDV(HalfFermionF *r, const GaugeF *u, const DiracFermionF *a, int mu, int num)}
\]
\[
\text{void QCDFspprojHUopDC1(HalfFermionF *r, const GaugeF *u, const int *ind1, const DiracFermionF *a, int mu, int num)}
\]
\[
\text{void QCDFspprojHUopDC2(HalfFermionF *r, const GaugeF *u, const DiracFermionF *a, const int *ind2, int mu, int num)}
\]
\[
\text{void QCDFspprojHUopDCX(HalfFermionF *r, const GaugeF *u, const int *ind1, const DiracFermionF *a, const int *ind2, int mu, int num)}
\]
\[
\text{r = U*spin_project(a, mu)}
\]
\[
\text{where op is}
\]
\[
\begin{array}{ll}
\text{mnnr} & \text{r = U*spin_project(a, mu)} \\
\text{mcnr} & \text{r = conj(U)*spin_project(a, mu)}
\end{array}
\]

\[
\text{void QCDFspreconDUopHS(DiracFermionF *r, const HalfFermionF *a, int mu)}
\]
\[
\text{void QCDFspreconDUopHV(DiracFermionF *r, const GaugeF *u, const HalfFermionF *a, int mu, int num)}
\]
void QCDFspreconDUopHC1(DiracFermionF *r, const GaugeF *u, const int *ind1, const HalfFermionF *a, int mu, int num)
void QCDFspreconDUopHC2(DiracFermionF *r, const GaugeF *u, const HalfFermionF *a, const int *ind2, int mu, int num)
void QCDFspreconDUopHCX(DiracFermionF *r, const GaugeF *u, const int *ind1, const HalfFermionF *a, const int *ind2, int mu, int num)

r = U*spin_reconstruct(a, mu)

where op is


Comparisons
Use comparisons to generate Boolean mask fields later used in copy under a mask

void QCDFcompToppS(BooleanF *r, const Type *a, const Type *b)
void QCDFcompToppV(BooleanF *r, const Type *a, const Type *b, int num)
void QCDFcompToppconstV(BooleanF *r, const Type *a, const Type b, int num)

r = a op b where T is a shortened name for Type

RealF, IntF

where op is


Boolean Operations
Boolean operations on Boolean values

void QCDFboolBoppS(BooleanF *r, const BooleanF *a, const BooleanF *b)
void QCDFboolBoppV(BooleanF *r, const BooleanF *a, const BooleanF*b, int num)

r = a op b

where op is


void QCDFboolBnotpS(BooleanF *r, const BooleanF *a)
void QCDFboolBnotpV(BooleanF *r, const BooleanF *a, int num)

r = not(a)
Copymask
Rather than replicate all possible operations with versions for a bit mask field, just make a
stand-alone copy under a mask. This is not optimal, but dramatically lower the explosion of
possible variants

```c
void QCDFmaskTopS(TypeDef *r, const BooleanF *a, const Type *b)
void QCDFmaskTopV(TypeDef *r, const BooleanF *a, const Type *b, int num)
```

where T is a shortened name for Type and op is

- rep \( r = (a)?b \)
- neg \( r = -(a)?b \)
- add \( r += (a)?b \)
- sub \( r -= (a)?b \)

Data-Layout
There seems to be a fixation on knowing exactly what data-layouts are in use.

**SZIN**
In fortran style ordering for indices (note, row major ordering):

- site(checkerboarded x,y,z,t)
- checkerboard x = int(x/2)
- checkerboard index = \( (x+y+z+t)\%2 \)

Lattice_GaugeField(real/imag, color row, color column, site, checkerboard, direction)
Lattice_DiracFermion(real/imag, color row, spin column, site, checkerboard)
Lattice_HalfFermion(real/imag, color row, spin column[2 components], site)

This is the generic ordering. For optimized Dirac operators, a common internal usage is
Lattice_GaugeField(real/imag, color row, color column, direction, site, checkerboard)

**Spin conventions**
Same as MILC and CPS. A chiral basis with a negative sign convention on gamma_2.