Abstract
This is a description of the interface to the library of low level routines to be used on UltraSPARC machines. Second implementation adds I/O functions for all external types, some functions missing in version 1 are also added. In addition, the HalfFermion interface is simplified. The source files are unified to simplify the verification process.

1 Compilation
The library supports SunSoft version 4.0 (and presumably higher) C compiler. For efficiency the library assumes certain data alignment (e.g., all data types below must be aligned on double word boundary.) It also requires the following flags for the compiler: -dalign -xlibmil -xtarget=ultra. There will be a single header file qcd.h containing all user-visible declarations.

All arguments are presumed disjoined and nonoverlapping. Using the reference and high performance implementations may yield different results if this assumption is violated.

Almost every function exists in two flavors: the scalar variant operates on a single set of arguments and produces a single result, while the vector variant takes a vector of argument sets and produces a vector of results. Semantically the vector version is equivalent to calling the scalar version repeatedly for every set of arguments, but, believe it or not, there is substantial performance gain in moving the loop to the lower level. To accommodate various data layouts, stride arguments are added to the vector functions. All strides are measured in units of relevant data types, so that to perform an operation on all elements of an array, put all stride arguments to 1. The vector length is always the last argument.

Probably there should be some I/O functions as well, but they could wait for a moment.

2 Data Types
Following data types are supported:
ComplexF single precision complex (total of 8 bytes.)

SU2F single precision $SU(2)$ matrix (16 bytes). It is intended as an internal part of the glue heatbath update, and its internal representation is irrelevant at this level.

SU3F $SU(3)$ single precision matrix (72 bytes.) The data layout is “unnatural” for historical reasons. See appendix B for details.

ScalarF $SU(3)$ fundamental scalar, single precision (24 bytes.)

HalfFermionF two component $SU(3)$ fundamental fermion, single precision (48 bytes.) This is a hack convenient for $R = 1$ Wilson fermions.

DiracFermionF single precision Dirac fermion (96 bytes.) See appendix B for memory layout. (Though I believe it is an implementation detail and the user has no need to know about it.)

I also reserve the type names with trailing F replaced by D for a possible future implementation of double precision data types.

Following sections describe all user level functions and their calling conventions. Since the library is mainly intended as the back end to a high level code system (e.g., ZSIN), I did not try to make them easy to remember. All function names starting with QCDF, QCDD and QCDx are reserved for the library.

For simplicity all functions return their first argument as a result. They all have side effects, since all parameters of the above types are passed by an address. The interface was not designed for use with brain-dead languages\(^1\), though some functions can be probably called without any additional stubs. Other functions will require stubs to defeat stupid argument passing conventions which can be written into high performance implementation, but I really do not think it is necessary.

\[\text{3 Complex numbers.}\]

Here functions manipulating only complex numbers are listed. For functions to extract a complex number for other data types see sections below. I describe only single precision functions, to get a double precision counterpart, simply replace F by D and float by double where appropriate (and pray it is implemented.)

\[\text{3.1 Constructors, accessors and setters.}\]

```c
void QCDFmkCxS(ComplexF *res, float re, float im)
void QCDFmkCxV(ComplexF *R, int sR, const float *re, int sre,
                    const float *im, int sim, int num)
```

\(^1\)Read: FORTRAN.
Packs two reals into one complex.

```c
void QCDFgetCxReS(float *re, const ComplexF *C)
void QCDFgetCxReV(float *re, int sre, const ComplexF *C,
                        int sC, int num)
```
Get the real part of the complex number.

```c
void QCDFgetCxImS(float *im, const ComplexF *C)
void QCDFgetCxImV(float *im, int sre, const ComplexF *C,
                        int sC, int num)
```
Get the imaginary part of the complex number.

```c
void QCDFputCxReS(ComplexF *C, float re)
void QCDFputCxReV(ComplexF *C, int sC, const float *re,
                        int sre, int num)
```
Set the real part of the complex number.

```c
void QCDFputCxImS(ComplexF *C, float im)
void QCDFputCxImV(ComplexF *C, int sC, const float *im,
                        int sre, int num)
```
Set the imaginary part of the complex number.

### 3.2 Arithmetics

```c
void QCDFbiCxop2S(ComplexF *t, const ComplexF *f)
void QCDFbiCxop2V(ComplexF *t, int st, const ComplexF *f,
                        int sf, int num)
```
Binary operations. Basically, all possible combinations of complex conjugate and multiplying by $-1, i, -i$:

- **set** $t = f$
- **neg** $t = -f$
- **add** $t = t + f$
- **sub** $t = t - f$
- **cset** $t = f^*$
- **cneg** $t = -f^*$
- **addc** $t = t + f^*$
- **subc** $t = t - f^*$
- **pI** $t = if$
- **mI** $t = -if$
- **addI** $t = t + if$
- **subI** $t = t - if$
- **cpI** $t = if^*$
- **cmI** $t = -if^*$
- **addcI** $t = t + if^*$
- **subcI** $t = t - if^*$

```c
void QCDFnorm2CxS(float *r, const ComplexF *a)
void QCDFnorm2CxV(float *r, int sr, const ComplexF *a, int sa,
                        int num)
```
Computes $r = |a|^2$.

```c
void QCDFnorm2addCxS(float *r, const ComplexF *a)
void QCDFnorm2addCxV(float *r, int sr, const ComplexF *a, int sa, int num)
```

Computes $r = r + |a|^2$.

```c
void QCDFsum2CxS(float *r, const ComplexF *a)
void QCDFsum2CxV(float *r, int sr, const ComplexF *a, int sa, int num)
```

**NB:** The scalar operation `QCDFsum2CxS` is an alias to `QCDFnorm2addCxS`.

```c
void QCDFthrCxop3S(ComplexF *r, const ComplexF *a, const ComplexF *b)
void QCDFthrCxop3V(ComplexF *r, int sr, const ComplexF *a, int sa, const ComplexF *b, int sb, int num)
```

Triadic operations. Division is excluded since it is not a frequent operation in QCD code.

```
add r = a + b         sub r = a - b
mnaa r = r + a * b    mns r = r - a * b
smnn r = a * b - r    smn r = a * b - r
mnaa r = r + a * b    mncs r = r - a * b +
smnc r = a * b - r    mncs r = r - a * b -
mcco r = r - a * b +  mccs r = a * b - r
mnn r = a * b         mcn r = a * b
mnc r = a * b         mcc r = a * b
```

4 \textbf{SU}(2)

Since \textit{SU}(2) is not intended for general use, there is a limited set of operations on it.

```c
void QCDFmkSU2rxS(SU2F *res, float a0, float a1, float a2, float a3)
void QCDFmkSU2rxV(SU2F *res, int sr, const float *a0, int s0, const float *a1, int s1, const float *a2, int s2, const float *a3, int s3, int num)
```
Construct an $SU(2)$ matrix from four floats. No normalization is performed. The constructed matrix is equivalent to

$$\begin{pmatrix} a_0 + ia_3 & a_1 + ia_2 \\ -a_1 + ia_2 & a_0 - ia_3 \end{pmatrix}.$$ 

void QCDFopSU2S(SU2F *r, const SU2F *a, const SU2F *b)
void QCDFopSU2V(SU2F *r, int sr, const SU2F *a, int sa, const SU2F *b, int sb, int num)

Multiplications:

$$
\begin{align*}
mnn &\quad r = ab \\
mnh &\quad r = a^* b^\dagger \\
mhn &\quad r = a^\dagger b \\
mhh &\quad r = a^\dagger b^\dagger
\end{align*}
$$

void QCDFretrSU2S(float *r, const SU2F *a)
void QCDFretrSU2V(float *r, int sr, const SU2F *a, int sa, int num)

Make $r = 1/2 \text{Re} \text{Tra}$. 

There are also a couple of functions for extracting $SU(2)$ subgroups from $SU(3)$ matrix and packing $SU(2)$ to $SU(3)$, see section 5.

5 $SU(3)$

void QCDFzeroSU3S(SU3F *a)
void QCDFzeroSU3V(SU3F *a, int sa, int num)

Put the zero matrix into $a$. (Yes, it is not an element of $SU(3)$, just a handy operation to have around.)

void QCDFoneSU3S(SU3F *a)
void QCDFoneSU3V(SU3F *a, int sa, int num)

Put the unity matrix into $a$.

void QCDFgetSU3S(ComplexF *r, const SU3F *a, int i, int j)
void QCDFgetSU3V(ComplexF *r, int sr, const SU3F *a, int sa, int i, int j, int num)

Extract $r = a_{ij}, i, j = 0...2$.

void QCDFputSU3S(SU3F *a, int i, int j, const ComplexF *r)
void QCDFputSU3V(SU3F *a, int sa, int i, int j, const ComplexF *r, int sr, int num)
Put $a_{ij} = r$, $i, j = 0 \ldots 2$.

```c
void QCDFget2SU3S(SU2F *r, const SU3F *a, int i, int j)
void QCDFget2SU3V(SU2F *r, int sr, const SU3F *a, int sa, int i, int j, int num)
```

Extract $(i,j)$ SU(2) “canonical” subgroup used in Cabbibo-Marinari updates. $i$ and $j$ must be distinct and $i, j = 0 \ldots 2$.

```c
void QCDFmkSU3S(SU3F *r, const SU2F *a, int i, int j)
void QCDFmkSU3V(SU3F *r, int sr, const SU2F *a, int sa, int i, int j, int num)
```

Build $SU(3)$ matrix corresponding to the “canonical” $(i,j)$ embedding of $SU(2)$. $i$ and $j$ must be distinct and $i, j = 0 \ldots 2$.

```c
void QCDFretrSU3S(float *r, const SU3F *a)
void QCDFretrSU3V(float *r, int sr, const SU3F *a, int sa, int num)
```

Put $r = \text{Re}Tr a$. 

*NB: there is no $1/3$ in front of the trace.*

```c
void QCDFconjSU3S(SU3F *r, SU3F *a)
void QCDFconjSU3V(SU3F *r, int sr, SU3F *a, int sa, int num)
```

Hermitian conjugation. $r = a^\dagger$.

### 5.1 Multiplications

```c
void QCDFxSU3opS(SU3F *r, const SU3F *a, const SU3F *b)
void QCDFxSU3opV(SU3F *r, int sr, const SU3F *a, int sa, const SU3F *b, int sb, int num)
```

Multiply and add:

$$
\begin{align*}
  mn & = ab  & man & = r + ab \\
  mhn & = a^\dagger b & mahn & = r + a^\dagger b \\
  mhh & = a^\dagger b^\dagger & mahh & = r + a^\dagger b^\dagger
\end{align*}
$$

```c
void QCDFstSU3opS(SU3F *r, const SU3F *a, const SU3F *b, const SU3F *c)
void QCDFstSU3opV(SU3F *r, int sr, const SU3F *a, int sa, const SU3F *b, int sb, const SU3F *c, int sc, int num)
```

The left and right staples for the Wilson action:

\[
\text{left } r = r + a^\dagger bc \quad \text{right } r = r + abc^\dagger
\]

void QCDFcdotSU3S(ComplexF *r, const SU3F *a, const SU3F *b)
void QCDFcdotSU3V(ComplexF *r, int sr, const SU3F *a, int sa,
const SU3F *b, int sb, int num)
The \(SU(3)\) scalar product: \(r = \text{Tr}(a^\dagger b)\).

void QCDFrdotSU3S(ComplexF *r, const SU3F *a, const SU3F *b)
void QCDFrdotSU3V(ComplexF *r, int sr, const SU3F *a, int sa,
const SU3F *b, int sb, int num)
The real part of the \(SU(3)\) scalar product: \(r = \text{ReTr}(a^\dagger b)\).

6 Matter of Color

Two types of matter fields, ScalarF and DiracFermionF share the same set of operations. They differ only in the access/setter section. Here common operations are listed. \(MT\) is \(SC\) and \(DF\) for ScalarF and DiracFermionF respectively. \(Type\) is ScalarF or DiracFermionF. (Since this interface is C oriented, there is no name overloading at this level.)

void QCDFzeroMTS(Type *a)
void QCDFzeroMTV(Type *a, int sa, int num)
Put zero into \(a\).

void QCDFnorm2MTS(float *r, const Type *a)
void QCDFnorm2MTV(float *r, int sr, const Type *a, int sa,
int num)
Returns \(\sum_{ix} |a_{ix}|^2\) over all indices.

void QCDFdotMTS(ComplexF *r, const Type *a, const Type *b)
void QCDFdotMTV(ComplexF *r, const Type *a, int sa,
const Type *b, int sb, int num)
Calculate the dot product of \(a\) and \(b\): \(r = r + \sum_{nx} a_n^\dagger(x)b_n(x)\).

void QCDFcaddMTS(Type *r, const Type *a,
const ComplexF *gamma, const Type *b)
void QCDFraddMTS(Type *r, const Type *a, float gamma,
const Type *b)
void QCDFcaddMTV(Type *r, int sr, const Type *a, int sa, const ComplexF *gamma, const Type *b, int sb, num)
void QCDFraddMTV(Type *r, int sr, const Type *a, int sa, float gamma, const Type *b, int sb, num)

Add two matter fields: \( r = a + \gamma b \).

void QCDFthrMTSU3opS(Type *r, const SU3F *U, const Type *a)
void QCDFthrMTSU3opV(Type *r, int sr, const SU3F *U, int sU, const Type *a, int sa, int num)
void QCDFthrMTSU3opC1(Type *r, const SU3F *U, const int *sU, const Type *a, int num)
void QCDFthrMTSU3opC2(Type *r, const SU3F *U, const Type *a, const int *sa, int num)
void QCDFthrMTSU3opCX(Type *r, const SU3F *U, const int *sU, const Type *a, const int *sa, int num)

Multiplies a by \( SU(3) \) matrix:

\[
\begin{align*}
    mn & \quad r = Ua & \quad ma & \quad r = r + Ua \\
    mh & \quad r = U^\dagger a & \quad mah & \quad r = r + U^\dagger a
\end{align*}
\]

For semantics of \( \text{Cx} \)-versions, see appendix C.

### 6.1 Scalars

void QCDFgetSCS(ComplexF *r, const ScalarF *S, int c)
void QCDFgetSCV(ComplexF *r, int sr, const ScalarF *S, int sS, int c, int num)

Get \( c \)th component of the scalar, \( c = 0 \ldots 2 \).

void QCDFputSCS(ScalarF *s, const ComplexF *v, int c)
void QCDFputSCV(ScalarF *s, int ss, const ComplexF *v, int sv, int c, int num)

Set \( c \)th component of the scalar, \( c = 0 \ldots 2 \).

### 6.2 Dirac Fermions

void QCDFgetDFS(ComplexF *r, const DiracFermionF *S, int c, int d)
void QCDFgetDFV(ComplexF *r, int sr, const DiracFermionF *S, int ss, int c, int d, int num)
Get $c$th (color), $d$th (spinor) component of the DiracFermion, $c = 0 \ldots 2$, $d = 0 \ldots 3$.

```c
void QCDFputDFS(DiracFermionF *s, const ComplexF *v, int c)
void QCDFputDFV(DiracFermionF *s, int ss, const ComplexF *v, int sv, int c, int num)
```

Set $c$th (color), $d$th (spinor) component of the Dirac fermion, $c = 0 \ldots 2$, $d = 0 \ldots 3$.

```c
void QCDFgDFopS(DiracFermionF *r, int idx, const DiracFermionF *s)
void QCDFgDFopV(DiracFermionF *r, int sr, int idx, const DiracFermionF *s, int ss, int num)
```

Multiply Dirac fermion by a product of $\gamma$-matrices and store the result. $\text{idx}$ is interpreted according to appendix A. $\text{op}$ is one of the following:

- **set** $r = \gamma^{\text{idx}} s$
- **neg** $r = -\gamma^{\text{idx}} s$
- **add** $r = r + \gamma^{\text{idx}} s$
- **sub** $r = r - \gamma^{\text{idx}} s$
- **iset** $r = i\gamma^{\text{idx}} s$
- **ineg** $r = -i\gamma^{\text{idx}} s$
- **iadd** $r = r + i\gamma^{\text{idx}} s$
- **isub** $r = r - i\gamma^{\text{idx}} s$

There is a couple of macros for the charge conjugation operations, they should be thought of as the function calls with the prototypes:

```c
void QCDFCCS(DiracFermionF *r, const DiracFermionF *s)
void QCDFCCaddS(DiracFermionF *r, const DiracFermionF *s)
void QCDFCCV(DiracFermionF *r, int sr, const DiracFermionF *s, int ss, int num)
void QCDFCCaddV(DiracFermionF *r, int sr, const DiracFermionF *s, int ss, int num)
```

Multiply Dirac fermion by a product of $\gamma$-matrices and store the result. The product of gamma matrices is determined by values of $n_k$: $\Gamma = \gamma^{n_1_2_3_4}$, all $n_k$ are 0 or 1. The gamma matrices are defined in Appendix A, $\text{op}$ is one of the following:

- **set** $r = \Gamma s$
- **iset** $r = i\Gamma s$
- **neg** $r = -\Gamma s$
- **ineg** $r = -i\Gamma s$
- **add** $r = r + \Gamma s$
- **iadd** $r = r + i\Gamma s$
- **sub** $r = r - \Gamma s$
- **isub** $r = r - i\Gamma s$

Frequent operations used in Wilson fermion action are provided.
Depending on \( \text{op} \) the following is calculated:

\[
\begin{align*}
plSU3man \; r &= r + (1 + \gamma_d)Ua \\
plSU3mah \; r &= r + (1 + \gamma_d)U^\dagger a
\end{align*}
\]

### 6.3 Half Fermions

This type is used internally to implement optimized calculation of \((1 \pm \gamma_\mu)U \psi\). I feel that it is advantageous to expose the matrix operations and conversion routines to the user.

Builds HalfFermion corresponding to \((1 + \gamma_d)s(\ldots pxF.)\) or \((1 - \gamma_d)s(\ldots mxF.)\).

Restore 4 components from HalfFermion corresponding to \((1 + \gamma_d)s(\ldots pxF.)\) or \((1 - \gamma_d)s(\ldots mxF.)\).
void QCDFthrHFSU3opC1(HalfFermion *r, const SU3F *U, 
    const int *sU, const HalfFermion *a, 
    int num)
void QCDFthrHFSU3opC2(HalfFermion *r, const SU3F *U, 
    const HalfFermion *a, const int *sa, 
    int num)
void QCDFthrHFSU3opCX(HalfFermion *r, const SU3F *U, 
    const int *sU, const HalfFermion *a, 
    const int *sa, int num)

Multiplies a by SU(3) matrix:
\[
    mn \quad r = U a \quad mh \quad r = U^\dagger a
\]

See appendix C for semantics of Cx-calls.

7 I/O

For ComplexF, SU3F, ScalarF and DiracFermionF there is a simple file I/O interface. read() and write() operations are provided for each data type. The external representation follows appendix B.

void QCDFreadDTS(Type *r, int fd)
void QCDFreadDTV(Type *r, int sr, int fd, int num)
void QCDFwriteDTS(int fd, const Type *r)
void QCDFwriteDTV(int fd, const Type *r, int sr, int num)

The Type and DT are as follows:

\[
\begin{array}{|c|}
\hline
DT & Type \\
\hline
Cx & ComplexF \\
SU3 & SU3F \\
SC & ScalarF \\
DF & DiracFermionF \\
\hline
\end{array}
\]

A Gamma Matrices

user is strongly encouraged to use the interface as described in the preceding sections instead of poking into implementation details below.

The indexing scheme for the gamma matrices is as follows. There are four matrices, \( \gamma_1, \gamma_2, \gamma_3 \) and \( \gamma_4 \), and the basis of 16 matrices \( 1, \gamma_1, \ldots, \gamma_5 \) is indexed by an integer \( d = 0 \ldots 15 \) as:

\[
\gamma^{(d)} = \gamma_1^{b_0} \gamma_2^{b_1} \gamma_3^{b_2} \gamma_4^{b_3}, \quad d = 8b_3 + 4b_2 + 2b_1 + b_0.
\]
Though there is no legal need to know the basis in Clifford algebra, here it is for an enquiring mind.

Using Pauli matrices:

\[ \sigma^1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \sigma^2 = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \sigma^3 = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \]

The gamma matrices \( \gamma^\mu \) are defined as:

\[ \gamma_1 = -\sigma^2 \otimes \sigma^1, \quad \gamma_2 = \sigma^2 \otimes \sigma^2, \quad \gamma_3 = -\sigma^2 \otimes \sigma^3, \quad \gamma_4 = \sigma^1 \otimes 1, \]

where

\[ 1 \otimes X = \begin{pmatrix} X & 0 \\ 0 & X \end{pmatrix}, \quad \sigma^1 \otimes X = \begin{pmatrix} 0 & X \\ X & 0 \end{pmatrix}, \quad \sigma^2 \otimes X = \begin{pmatrix} 0 & -iX \\ iX & 0 \end{pmatrix}, \quad \sigma^3 \otimes X = \begin{pmatrix} X & 0 \\ 0 & -X \end{pmatrix} \]

In this representation the charge conjugation is:

\[ C = -i \otimes \sigma^2 = \gamma^{\{0101\}} = \gamma^{(5)} \]

B Data Layout

Guess what?—It is weird!

Complex numbers are stored as real, imag.

The \( SU(3) \) matrix \( a \) is stored as

\[
\begin{array}{cccccccc}
\text{Re}a_{00}, & \text{Im}a_{00}, & \text{Re}a_{10}, & \text{Im}a_{10}, & \text{Re}a_{20}, & \text{Im}a_{20}, \\
\text{Re}a_{01}, & \text{Im}a_{01}, & \text{Re}a_{11}, & \text{Im}a_{11}, & \text{Re}a_{21}, & \text{Im}a_{21}, \\
\text{Re}a_{02}, & \text{Im}a_{02}, & \text{Re}a_{12}, & \text{Im}a_{12}, & \text{Re}a_{22}, & \text{Im}a_{22};
\end{array}
\]

in increasing offsets.

I do not disclose \( SU(2) \) storage.

The scalar \( a \) is stored (in increasing addresses) as

\[
\begin{array}{cccc}
\text{Re}a_0, & \text{Im}a_0, & \text{Re}a_1, & \text{Im}a_1, & \text{Re}a_2, & \text{Im}a_2.
\end{array}
\]

Half fermions are in some beautiful internal format which is not to be discussed here.

Assuming that the first index is color, the Dirac fermions are stored as

\[
\begin{array}{cccccccccccc}
\text{Re}a_{00}, & \text{Im}a_{00}, & \text{Re}a_{10}, & \text{Im}a_{10}, & \text{Re}a_{20}, & \text{Im}a_{20}, \\
\text{Re}a_{01}, & \text{Im}a_{01}, & \text{Re}a_{11}, & \text{Im}a_{11}, & \text{Re}a_{21}, & \text{Im}a_{21}, \\
\text{Re}a_{02}, & \text{Im}a_{02}, & \text{Re}a_{12}, & \text{Im}a_{12}, & \text{Re}a_{22}, & \text{Im}a_{22}, \\
\text{Re}a_{03}, & \text{Im}a_{03}, & \text{Re}a_{13}, & \text{Im}a_{13}, & \text{Re}a_{23}, & \text{Im}a_{23}.
\end{array}
\]
C  How *C? calls work

Due to popular demand, some functions now have C1, C2 and CX variants in addition to C and V. Semantics of these functions is illustrated below using a fictitious operation foo:

```c
void fooS(T1 *r, const TA *a, const TB *b) { ... }

void fooC1(T1 *r,
           const TA *a, const int *sa,
           const TB *b,
           int num)
{
    int i;
    for (i = 0; i < num; i++, r++, b++, sa++)
        fooS(r, a + *sa, b);
}

void fooC2(T1 *r,
           const TA *a,
           const TB *b, const int *sb,
           int num)
{
    int i;
    for (i = 0; i < num; i++, r++, a++, sb++)
        fooS(r, a, b + *sb);
}

void fooCX(T1 *r,
           const TA *a, const int *sa,
           const TB *b, const int *sb,
           int num)
{
    int i;
    for (i = 0; i < num; i++, r++, sa++, sb++)
        fooS(r, a + *sa, b + *sb);
}
```

However, the implementation may differ in the order of operations (as any function in the library) and may run noticeably faster than the code above.