1 Simple data types

The following data types are defined as C structures containing a single array of elements.

Name	array of	size	members name
complex	double	2	re, im
suNg_vector	complex	Nc	c[]
suNg	complex	Nc×Nc	c[]
suNg_spinor	suNg_vector	4	c[]
suNg_algebra_vector	double	Nc×Nc-1	c[]
suNf_vector	complex	dr	c[]
suNf	complex or double	dr×dr	c[]
suNfc	complex	dr×dr	c[]
suNf_spinor	suNf_vector	4	c[]

Legend: Nc=number of colors ; dr=dimension of the fermion representation

The data type "suNf" can be real or complex depending on the representation being real or complex.

Every data type has a corresponding single precision data type the name of which is obtained adding the suffix "_flt".

2 Operations on simple data types

The following linear algebra operations are defined as C preprocessor macros and can act on both double and single precision data types.

2.1 Vector operations

Vector operations act on one or more suNg_vectors or suNf_vectors. The suffix "_g" or "_f" indicates the appropriate data type.

_vector_zero_[gf](r) : r=0
suN[gf]_vector r

_vector_minus_[gf](r,s) : r-=s
suN[gf]_vector r,s

_vector_mul_[gf](r,k,s) : r=k∗s
suN[gf]_vector r,s ;
double k

_vector_mulc_[gf](r,z,s) : r=z∗s
suN[gf]_vector r,s;
complex z

_vector_add_[gf](r,s1,s2) : r=s1+s2
suN[gf]_vector r,s1,s2

_vector_sub_[gf](r,s1,s2) : r=s1-s2
suN[gf]_vector r,s1,s2

_vector_i_add_[gf](r,s1,s2) : r=s1+i∗s2
suN[gf]_vector r,s1,s2

_vector_i_sub_[gf](r,s1,s2) : r=s1-i∗s2
suN[gf]_vector r,s1,s2

_vector_add_assign_[gf](r,s) : r+=s
suN[gf]_vector r,s

_vector_sub_assign_[gf](r,s) : r-=s
suN[gf]_vector r,s

_vector_i_add_assign_[gf](r,s) : r+=i∗s
suN[gf]_vector r,s

_vector_i_sub_assign_[gf](r,s) : r-=i∗s
suN[gf]_vector r,s

_vector_prod_re_[gf](k,r,s) : k=Re($r^\dagger$∗s)
suN[gf]_vector r,s;
double k

_vector_prod_im_[gf](k,r,s) : k=Im($r^\dagger$∗s)
suN[gf]_vector r,s;
double k

_vector_mulc_add_assign_[gf](r,z,s) : r+=z∗s (z complex)
suN[gf]_vector r,s;
complex z

_vector_mul_add_assign_[gf](r,k,s) : r+=k∗s (k real)
suN[gf]_vector r,s;
double k

_vector_lc_[gf](r,k1,s1,k2,s2) : r=k1∗s1+k2∗s2
suN[gf]_vector r,s1,s2;
double k1,k2

_vector_lc_add_assign_[gf](r,k1,s1,k2,s2) : r+=k1∗s1+k2∗s2
suN[gf]_vector r,s1,s2;

double k1,k2

_vector_clc_[gf](r,z1,s1,z2,s2) : r=z1∗s1+z2∗s2
suN[gf]_vector r,s1,s2;
complex z1,z2

_vector_clc_add_assign_[gf](r,z1,s1,z2,s2) : r=z1∗s1+z2∗s2
suN[gf]_vector r,s1,s2;
complex z1,z2

_vector_prod_assign_[gf](z,r,s) : z+=r†∗s
suN[gf]_vector z,r,s

_vector_project_[gf](r,z,s) : r-=z∗s
suN[gf]_vector r,z,s

2.2 Algebra vectors operations

_algebra_vector_mul_add_assign_g(r,k,s) : r+=k∗s
suNg_algebra_vector r,s
double k

_algebra_vector_mul_g(r,k,s) : r=k∗s
suNg_algebra_vector r,s
double k

_algebra_vector_zero_g(r) : r=0
suNg_algebra_vector r

_algebra_vector_sqnorm_g(k,r) : k=|v|^2
suNg_algebra_vector r
double k

2.3 Matrix-Vector operations

_suN[gf]_multiply(r,u,s) : r=u∗s
suN[gf]_vector r,s
suN[gf] u

_suN[gf]_inverse_multiply(r,u,s) : r=u†∗s
suN[gf]_vector r,s
suN[gf] u

2.4 Matrix operations

_suN[gf]_dagger(u,v) : u=$v^\dagger$
suN[gf] u,v

_suN[gf]_times_suN[gf](u,v,w) : u=v∗w
suN[gf] u,v,w

_suN[gf]_times_suN[gf]_dagger(u,v,w) : u=v∗$w^\dagger$
suN[gf] u,v,w

_suN[gf]_dagger_times_suN[gf](u,v,w) : u=$v^\dagger$∗w
suN[gf] u,v,w

_suN[gf]_zero(u) : u=0
suN[gf] u

_suN[gf]_unit(u) : u=1
suN[gf] u

_suN[gf]_minus(u,v) : u=-v
suN[gf] u,v

_suN[gf]_mul(u,r,v) : u=r∗v
suN[gf] u,v double r

_suN[gf]_add_assign(u,v) : u+=v
suN[gf] u,v

_suN[gf]_sub_assign(u,v) : u-=v
suN[gf] u,v

_suN[gf]_sqnorm(k,u) : k=$|u|^2$
double k suN[gf] u

_suN[gf]_sqnorm_m1(k,u) : k=$|\, 1 - u \,|^2$
double k suN[gf] u

_suN[gf]_trace_re(k,u) : k=Re Tr (u)
suN[gf] k,u

_suN[gf]_trace_im(k,u) : k=Im Tr (u)
suN[gf] k,u

_suN[gf]_2TA(u,v) : u=v - $v^\dagger$ -1/N Tr(v - $v^\dagger$)∗I suN[gf] u,v

_suN[gf]_TA(u,v) : u=(v - $v^\dagger$ -1/N Tr(v - $v^\dagger$)*I)/2
suN[gf] u,v

2.5 Spinor operations

Vector operations defined in Sec. **??** naturally define spinor operations componentwise, since a spinor a just 4 copies of a vector. The name convention for spinor operations is to replace the prefix "_vector" with "_spinor". For example:
_spinor_mul_[gf](r,k,s) : r=k*s
suN[gf]_spinor r,s
double k

In addition to the above spinor operations which act trivially on the Dirac structure, the following are also defined:
_spinor_g5_[gf](s,r) : s=γ_5*r
suN[gf]_spinor s,r

_spinor_g5_assign_[gf](r) : r=γ_5*r
suN[gf]_spinor r

_spinor_g5_prod_re_[gf](k,r,s) : k=Re [$(\gamma_5$*r$)^\dagger$*s]
double k
suN[gf]_spinor r,s

_spinor_g5_prod_im_[gf](k,r,s) : k=Im [$(\gamma_5$*r$)^\dagger$*s]
double k
suN[gf]_spinor r,s

3 Fields data types

A field data type is defined as a C structure containing the following members:

1. a pointer to an array of elements named "ptr";
2. a pointer to a geometry_descriptor defining the geometry on which the field lives named "type";
3. IF compiled for MPI: a pointer to an array of MPI_request for communication requests named "comm_req".

The following field types are defined:

Name	field of
suNg_field	suNg
suNf_field	suNf
spinor_field	suNf_spinor
suNg_av_field	suNg_algebra_vector
scalar_field	double

4 Geometry

Fields living on the 4-dimesional lattice are defined to be C arrays of elements (see Sec.**??**). The geometry of the lattice is defined by assigning an index n of the array to each site (t, x, y, z). The mapping between the cartesian coordinates of the local lattice and the array index is given by the macro "ipt(t,x,y,z)=n". Neighboring relations between the sites are given by the macros "iup(n,dir)" and "idn(n,dir)" which, given the index n of the current site, return the index of the site whose cartesian coordinate in direction dir is increased or decreased by one respectively.

In the MPI version of the code, the set of indeces which correspond to the local lattice are NOT all contiguous, instead the lattice is divided into different blocks. Each of these blocks then corresponds to a contiguous set of indeces. In the MPI version, each field also contain the send/receive buffers necessary for communications. Before a communication to another MPI process can be made, the send buffer must be filled. The division of the local lattice into blocks, the location of the send/receive buffers, and the copies to fill the send buffers are described by the following C structure:

```
typedef struct _geometry_descriptor {
  unsigned int local_master_pieces, total_master_pieces;
  unsigned int *master_start, *master_end;
  unsigned int ncopies;
  unsigned int *copy_from, *copy_to, *copy_len;
  unsigned int nbuffers;
  unsigned int *rbuf_len, *sbuf_len;
  unsigned int *rbuf_from_proc, *rbuf_start;
  unsigned int *sbuf_to_proc, *sbuf_start;
  unsigned int gsize;
} geometry_descriptor;
```

local_master_pieces: number of pieces in which the local lattice is divided. We call "master" a piece of the array which does not contain copies of other sites.

total_master_pieces: local_master_pieces + number of receive buffers. The latter can be though of as an extension of the local lattice in those directions which are parallelized (i.e. the global lattice is split in that direction).

master_start, master_end: these two array contain the start and end index for each master piece in the lattice.

ncopies: this is the number of copies necessary to fill the send buffers.

copy_from, copy_to, copy_len: these three array contain the origin and destination index for each copy and the length in sites.