This document exists to inform you about how to use the code and the conventions I use. As of writing, the code supports NERSC, SCIDAC, MILC, `ILDG_SCIDAC`, `ILDG_BQCD`, HiRep and generic LIME configuration files. You have already (even if you didn't realise) compiled-in the gauge group to SU(NC) and the number of dimensions ND, and the floating point precision (default is double).

First things first, you have a compiled binary called "GLU". The code expects input of the form,

```
./GLU -i {input_file} -c {configuration} -o {outputfile}
```

The code works on a per-configuration basis.

The input file works by dumping the strings in memory and simply fscanf-ing for tags of the form,

```
{TAG} = {VALUE}
```

hence the spaces before and after the equals are important the terms `{TAG} = {VALUE}` have to appear in the input file as they are checked for in the code even if they are not used.

The code must have at least the input file and the configuration as arguments. The output file is optional, and only corresponds to writing out a configuration.

1 Modes

The first line of the input file is quite an important one, it specifies the mode of operations for the code with options (options are always bracketed with curly braces { })

```
MODE = {CUTTING,GAUGE_FIXING,SMEARING,SUNxU1,...}
```

anything else (denoted by the ...) switches to the default behaviour which measures some gauge invariant loops and the gauge fixing accuracy for both Landau and Coulomb gauges.

The specific modes and what they can do for you are discussed in the other documentation files. If you are really impatient though, you can access the in-built help via the command,

```
./GLU --help
```

Which will tell you about the possible options available for use in the input file and can be used to automatically generate input file examples.

2 Field conventions

We store our gauge fields in full $NC \times NC$ format as a 1-dimensional complex array (using complex arithmetic from <complex.h>). There are ND polarisations of this matrices per site. I store these fields in a lattice-wide 1-D array, whose geometry is encoded lexicographically.

Within the lattice field struct, I include information on the nearest neighbours of the site. The links flowing away from the site to the next in the direction mu are found in the neighbor[mu] array. And the ones flowing to the site are in the back[mu] array.

As my lattice conventions, I have the x direction running fastest, then the y ... And so on. With the final index being the temporal. For a 4D field with lattice lengths L_x, L_y, L_z, L_t, the site index (i) of the position (x,y,z,t) can be taken from,

$$i = x + L_x(y + L_y(z + L_z * t)). \tag{1}$$

I have defined in macros (most macro definitions are in ``GLU_definitions.h''), using the example for 4D,

$$\begin{aligned} \mathrm{LSQ} &= L_x L_y, \\ \mathrm{LCU} &= L_x L_y L_z, \\ \mathrm{LVOLUME} &= L_x L_y L_z L_t. \end{aligned} \tag{2}$$

For $ND \neq 4$ these definitions are still valid, with `LVOLUME` being the total number of lattice sites and `LCU` being the spatial hypercube size.

3 Random numbers

The code uses as a default the Well (P)RNG of Matsumoto, L'ecuyer et al. This RNG, like the Mersenne Twister uses a table of 600 numbers. We ask for a single SEED value, and use a Lagged Fibonacci generator to fill the table. If everyone is OK with GSL doing this, why the hell can't I? There are also versions of the `KISS, MWC_1038, MWC_4096` RNGs available. The RNG also burns the first 2000 or so numbers in a warm up phase.

If the seed is 0, the code looks in `/dev/urandom/` for a seed out of the Linux entropy pool. If urandom doesn't exist, the code will complain and exit.

If you want to randomly gauge transform your initial gauge fields (as a check for gauge invariance of measures, or for looking for a more convergent local minimum for the gauge fixing) the option in the input file,

```
RANDOM_TRANSFORM = {YES}
```

ssuffices, all other options will be considered as not wanting to randomly transform the configuration.

4 Support

he geometry of the lattice is taken from the header of these files, or if not reading a configuration it is specified in the input file.

To specify what type of file you are attempting to read the option,

```
HEADER = {NERSC,HIREP,MILC,SCIDAC,ILDG,INSTANTON,RANDOM,UNIT}
```

is your friend.

If you are reading a HIREP, MILC, SCIDAC, ILDG configuration and writing out to a NERSC header, or something weird like that. These formats (worryingly) do not put the configuration number in the header, and so you should specify it in the input file,

```
CONFNO = {}
```

does the trick.

The final three options require some explanation. `INSTANTON` generates a BPST instanton configuration with a single instanton. `RANDOM` creates a random configuration. And unit creates a gauge configuration which is made entirely of identity matrices.

If using `{INSTANTON,RANDOM,UNIT}` you must specify the dimensions of the problem. These are set in the input file as

```
DIM_0 = {}
DIM_1 = {}
....
```

The code will attempt to read ND of these. Any more dimensions specified will be ignored.

5 Output

The output file's type can be specified with,

```
OUTPUT = {NERSC_SMALL,NERSC_GAUGE,NERSC_NCxNC,HIREP,MILC,ILDG_SCIDAC,ILDG_
```

With the default being `NERSC_GAUGE`. The first three are NERSC-like files. `NERSC_GAUGE,NERSC_NCxNC` satisfy the NERSC conventions (and obvious NC and ND generic extensions), whereas `NERSC_SMALL` is technically not a NERSC compliant configuration. It shares the header but is only available for SU(2) and SU(3). It uses relations of the matrix to store the configurations in only 3 real constants for SU(2) and 8 real constants for SU(3)[1], it is *perhaps* a little unstable (never really seen this myself) but I was told it was.

[1]The minimal number of parameters needed to represent a matrix is the number of generators of the group, this could allow for an exact exponentiation routine for larger NC matrices, which would be a large saving

NERSC_GAUGE stores the top $NC-1$ rows and fills in the final one using the signed minors. NERSC_NCxNC writes the whole matrix. HIREP is the same as NERSC_NCxNC but with a different, less detailed header and with a different geometry (theirs is (t,x,y,z) for some reason, is this a european convention? It just seems wrong). All of the others write out the field as binary data in Big-Endian format the same as NERSC_NCxNC.

The input file option,

```
INFO = {}
```

Is the information about the configuration that appears in the NERSC header.

The default behaviour of the writers is to write out configurations in Big Endian format in the working precision of the code.