

Conjugate Gradient for Domain Wall Fermions

with 4-d EO preconditioning

Version 1.1.0 γ

Andrew Pochinsky

June 11, 2004

Abstract

This document presents an implementation of a conjugate gradient solver for the Domain Wall Fermion Dirac operator using Pentium 4 streaming SIMD extension (SSE). The code targets SciDAC's cluster machines implementing the QMP protocol.

Contents

1	INTRODUCTION	1
1.1	Definitions	1
1.2	Optimization Strategy	2
2	PHYSICS	3
2.1	Dirac Operator	3
2.2	Gamma matrices	3
3	CONJUGATE GRADIENT	6
3.1	Standard Algorithm	6
3.2	Overlap Opportunities	6
3.3	Non-hermitial Matrix	7
4	PRECONDITIONING	8
4.1	Q_{xx} inversion	8
5	CODE	10
5.1	Interface Functions	10
5.1.1	SSE DWF Initializer	10
5.1.2	SSE DWF Clean Up	11
5.1.3	DWF Fermion Allocator	11
5.1.4	DWF Fermion Exporter	12
5.1.5	DWF Fermion Importer	13
5.1.6	DWF Fermion Deallocator	13
5.1.7	DWF Gauge Exporter	13
5.1.8	DWF Gauge Deallocator	14
5.1.9	The Solver	14
5.2	Memory Allocation	15
5.2.1	Field allocators	16
5.3	Probing Cluster Topology	17
5.4	Moving Data	17
5.4.1	Reading the Gauge Field	17
5.4.2	Reading a Fermion	19
5.4.3	Writing a Fermion	20
5.5	Solver Initialization	21
5.5.1	Constructing the neighbor tables	21
5.5.2	Address translation routines	28

5.6	QMP Initialization	29
5.7	Parts of the Solver	32
5.7.1	Compute the RHS	32
5.8	Field Operations	32
5.8.1	Computing the even part of the result	34
5.8.2	<code>copy_o(d, s)</code> or $d \leftarrow s$	34
5.8.3	<code>compute_sum2_o(d, alpha, s)</code> , or $d \leftarrow d + \alpha s$	34
5.8.4	<code>compute_sum2x_o(d, s, alpha)</code> , or $d \leftarrow \alpha d + s$	34
5.8.5	<code>compute_sum_x(d, x, alpha, y)</code> or $q \leftarrow x + \alpha y$	35
5.8.6	Compute $d \leftarrow x + \alpha y$ and $r \leftarrow \langle d, d \rangle$	35
5.8.7	Compute $\eta \leftarrow M^\dagger M \psi$ and friends	36
5.8.8	Standalone diagonal pieces	37
5.8.9	Q_{xx}^{-1} and S_{xx}^{-1} on a single s -chain	38
5.8.10	Compute L_A^{-1} and L_B^{-1}	39
5.8.11	Compute R_A^{-1} and R_B^{-1}	42
5.8.12	Standalone off-diagonal pieces	46
5.8.13	Common off-diagonal parts	46
5.8.14	Projections to be sent	49
5.8.15	Parts of $Q_{xy}\psi$	51
5.8.16	Parts of $\eta - S_{xy}\psi$	53
5.8.17	Parts of $Q_{xx}^{-1}Q_{xy}\psi$	55
5.8.18	Parts of $S_{xx}^{-1}S_{xy}\psi$	55
5.8.19	Parts of $\eta - Q_{xx}^{-1}Q_{xy}\psi$	56
5.8.20	Miscellaneous	57
5.8.21	Combined pieces	58
5.8.22	Common Locals	58
5.8.23	Common globals	59
5.9	QMP Pieces	59
5.9.1	Global sums	60
5.10	SSE Types and Operations	60
5.10.1	SSE Types	61
5.10.2	SSE inline functions	62
5.11	Generally Useful Functions	64
5.12	Handy Constants	64
5.13	Source File	64
6	TODO LIST	65
7	CHUNKS	66

1 INTRODUCTION

The code below interfaces with a Chroma-like upper level environment to provide file access and machine initialization and configuration. In fact, this file is an implementation of a level 3 routine for solving the Dirac equation. There are some restrictions on input parameters imposed by the algorithm and a particular way the SSE is used by the implementation. There are the following restrictions on the lattice geometry:

- All four-dimensional extends of the lattice should be even. This is required for even-odd decomposition used in the preconditioner.
- The fifth-dimension extend should be a multiple of 4. It is needed for efficient use of SSE registers and simplification of vector code.
- The implementation supports up to four dimensional tori as a network topology.

Because of many issues involved in optimizing the code, it is advantageous to put together some definitions and outline here the optimization strategy used.

1.1 Definitions

Lattice extend is the total size of the lattice in a given dimension.

Network is the logical topology of the network presented by QMP to the application.

Node is a computing element in the network which runs an execution thread. For this implementation we assume that there is one compute node per network location. If an SMP is used, it is the responsibility of QMP to provide a proper abstraction to the application.

Sublattice is the part of the lattice that resides on a compute node.

Site is a point on the lattice.

1.2 Optimization Strategy

For this code we assume that scarcity of resources makes us run the inverter on a small number of nodes compared to the number of sites. This is based on the observation that physics needs grow faster than SciDAC budget and computer deployment plans. We also assume, that the current trend in computer industry persists, namely, that the processors grow faster while memory speed and latency continues to lag in relative terms. We also want a solver whose performance would degrade gracefully when one moves out of the optimization domain. In particular, we impose no limitation on the size of sublattice. There is even no requirement that all sublattices should be of the same size.

For the optimization sweetspot, we assume that the typical problem is too large to fit into the cache hierarchy and mostly resides in main memory. This is true now for existing and proposed clusters and is like to remain true for the future, since large scale computations tend to use larger lattices most of the time.

2 PHYSICS

Here we give the fermion action and γ -matrix and other conventions.

2.1 Dirac Operator

The Domain Wall Fermion Dirac operator is

$$\begin{aligned}\chi_{s,x} = D\psi &= M_0\psi_{s,x} + \sum_{\mu} \left((1 + \gamma_{\mu})U_{x,\mu}\psi_{s,x+\hat{\mu}} + (1 - \gamma_{\mu})U_{x-\hat{\mu},\mu}^{\dagger}\psi_{s,x-\hat{\mu}} \right) \\ &\quad + (1 + \gamma_5)M_s^{(+)}\psi_{s+1,x} + (1 - \gamma_5)M_s^{(-)}\psi_{s-1,x}\end{aligned}$$

where

$$M_s^{(+)} = \begin{cases} 1, & \text{if } s < N_s - 1 \\ -m_f, & \text{if } s = N_s - 1 \end{cases}$$

and

$$M_s^{(-)} = \begin{cases} 1, & \text{if } s > 0 \\ -m_f, & \text{if } s = 0 \end{cases}$$

We also assume that $\psi_{N_s,x} = \psi_{0,x}$ and $\psi_{-1,x} = \psi_{N_s-1,x}$.

2.2 Gamma matrices

We use the same γ -matrix basis as Chroma to simplify conversion between two codes. The choice below could be changed with a few modifications to the rest of the code, if γ_5 is kept diagonal, and one of other γ -matrices has all nonzero entries equal to +1.

$$\gamma_0 = -\sigma_2 \otimes \sigma_1 = \begin{pmatrix} 0 & i\sigma_1 \\ -i\sigma_1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & i \\ 0 & 0 & i & 0 \\ 0 & -i & 0 & 0 \\ -i & 0 & 0 & 0 \end{pmatrix}$$

First, the projector and the reconstructor for $(1 + \gamma_0)$:

- 4a $\langle \text{Build } (1 + \gamma_0) \text{ projection of } *f \text{ in } *g \text{ 4a} \rangle \equiv$
 $g \rightarrow f[0][c].re = f \rightarrow f[0][c].re - f \rightarrow f[3][c].im;$
 $g \rightarrow f[0][c].im = f \rightarrow f[0][c].im + f \rightarrow f[3][c].re;$
 $g \rightarrow f[1][c].re = f \rightarrow f[1][c].re - f \rightarrow f[2][c].im;$
 $g \rightarrow f[1][c].im = f \rightarrow f[1][c].im + f \rightarrow f[2][c].re;$
- 4b $\langle \text{Unproject and accumulate } (1 + \gamma_0) \text{ link 4b} \rangle \equiv$
 $qs \rightarrow f[0][c].re += gg[k].f[0][c].re; qs \rightarrow f[3][c].im -= gg[k].f[0][c].re;$
 $qs \rightarrow f[0][c].im += gg[k].f[0][c].im; qs \rightarrow f[3][c].re += gg[k].f[0][c].im;$
 $qs \rightarrow f[1][c].re += gg[k].f[1][c].re; qs \rightarrow f[2][c].im -= gg[k].f[1][c].re;$
 $qs \rightarrow f[1][c].im += gg[k].f[1][c].im; qs \rightarrow f[2][c].re += gg[k].f[1][c].im;$

Now, same for $(1 - \gamma_0)$:

- 4c $\langle \text{Build } (1 - \gamma_0) \text{ projection of } *f \text{ in } *g \text{ 4c} \rangle \equiv$
 $g \rightarrow f[0][c].re = f \rightarrow f[0][c].re + f \rightarrow f[3][c].im;$
 $g \rightarrow f[0][c].im = f \rightarrow f[0][c].im - f \rightarrow f[3][c].re;$
 $g \rightarrow f[1][c].re = f \rightarrow f[1][c].re + f \rightarrow f[2][c].im;$
 $g \rightarrow f[1][c].im = f \rightarrow f[1][c].im - f \rightarrow f[2][c].re;$
- 4d $\langle \text{Unproject and accumulate } (1 - \gamma_0) \text{ link 4d} \rangle \equiv$
 $qs \rightarrow f[0][c].re += gg[k].f[0][c].re; qs \rightarrow f[3][c].im += gg[k].f[0][c].re;$
 $qs \rightarrow f[0][c].im += gg[k].f[0][c].im; qs \rightarrow f[3][c].re -= gg[k].f[0][c].im;$
 $qs \rightarrow f[1][c].re += gg[k].f[1][c].re; qs \rightarrow f[2][c].im += gg[k].f[1][c].re;$
 $qs \rightarrow f[1][c].im += gg[k].f[1][c].im; qs \rightarrow f[2][c].re -= gg[k].f[1][c].im;$

$$\gamma_1 = \sigma_2 \otimes \sigma_2 = \begin{pmatrix} 0 & -i\sigma_2 \\ i\sigma_2 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix}$$

First, the projector and the reconstructor for $(1 + \gamma_1)$:

- 4e $\langle \text{Build } (1 + \gamma_1) \text{ projection of } *f \text{ in } *g \text{ 4e} \rangle \equiv$
 $g \rightarrow f[0][c].re = f \rightarrow f[0][c].re - f \rightarrow f[3][c].re;$
 $g \rightarrow f[0][c].im = f \rightarrow f[0][c].im - f \rightarrow f[3][c].im;$
 $g \rightarrow f[1][c].re = f \rightarrow f[1][c].re + f \rightarrow f[2][c].re;$
 $g \rightarrow f[1][c].im = f \rightarrow f[1][c].im + f \rightarrow f[2][c].im;$
- 4f $\langle \text{Unproject and accumulate } (1 + \gamma_1) \text{ link 4f} \rangle \equiv$
 $qs \rightarrow f[0][c].re += gg[k].f[0][c].re; qs \rightarrow f[3][c].re -= gg[k].f[0][c].re;$
 $qs \rightarrow f[0][c].im += gg[k].f[0][c].im; qs \rightarrow f[3][c].im -= gg[k].f[0][c].im;$
 $qs \rightarrow f[1][c].re += gg[k].f[1][c].re; qs \rightarrow f[2][c].re += gg[k].f[1][c].re;$
 $qs \rightarrow f[1][c].im += gg[k].f[1][c].im; qs \rightarrow f[2][c].im += gg[k].f[1][c].im;$

Now, same for $(1 - \gamma_1)$:

- 4g $\langle \text{Build } (1 - \gamma_1) \text{ projection of } *f \text{ in } *g \text{ 4g} \rangle \equiv$
 $g \rightarrow f[0][c].re = f \rightarrow f[0][c].re + f \rightarrow f[3][c].re;$
 $g \rightarrow f[0][c].im = f \rightarrow f[0][c].im + f \rightarrow f[3][c].im;$
 $g \rightarrow f[1][c].re = f \rightarrow f[1][c].re - f \rightarrow f[2][c].re;$
 $g \rightarrow f[1][c].im = f \rightarrow f[1][c].im - f \rightarrow f[2][c].im;$
- 4h $\langle \text{Unproject and accumulate } (1 - \gamma_1) \text{ link 4h} \rangle \equiv$
 $qs \rightarrow f[0][c].re += gg[k].f[0][c].re; qs \rightarrow f[3][c].re += gg[k].f[0][c].re;$
 $qs \rightarrow f[0][c].im += gg[k].f[0][c].im; qs \rightarrow f[3][c].im += gg[k].f[0][c].im;$
 $qs \rightarrow f[1][c].re += gg[k].f[1][c].re; qs \rightarrow f[2][c].re -= gg[k].f[1][c].re;$
 $qs \rightarrow f[1][c].im += gg[k].f[1][c].im; qs \rightarrow f[2][c].im -= gg[k].f[1][c].im;$

$$\gamma_2 = -\sigma_2 \otimes \sigma_3 = \begin{pmatrix} 0 & i\sigma_3 \\ -i\sigma_3 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & i & 0 \\ 0 & 0 & 0 & -i \\ -i & 0 & 0 & 0 \\ 0 & i & 0 & 0 \end{pmatrix}$$

First, the projector and the reconstructor for $(1 + \gamma_2)$:

- 5a $\langle \text{Build } (1 + \gamma_2) \text{ projection of } *f \text{ in } *g \text{ 5a} \rangle \equiv$
 $g \rightarrow f[0][c].re = f \rightarrow f[0][c].re - f \rightarrow f[2][c].im;$
 $g \rightarrow f[0][c].im = f \rightarrow f[0][c].im + f \rightarrow f[2][c].re;$
 $g \rightarrow f[1][c].re = f \rightarrow f[1][c].re + f \rightarrow f[3][c].im;$
 $g \rightarrow f[1][c].im = f \rightarrow f[1][c].im - f \rightarrow f[3][c].re;$
- 5b $\langle \text{Unproject and accumulate } (1 + \gamma_2) \text{ link 5b} \rangle \equiv$
 $qs \rightarrow f[0][c].re += gg[k].f[0][c].re; qs \rightarrow f[2][c].im -= gg[k].f[0][c].re;$
 $qs \rightarrow f[0][c].im += gg[k].f[0][c].im; qs \rightarrow f[2][c].re += gg[k].f[0][c].im;$
 $qs \rightarrow f[1][c].re += gg[k].f[1][c].re; qs \rightarrow f[3][c].im += gg[k].f[1][c].re;$
 $qs \rightarrow f[1][c].im += gg[k].f[1][c].im; qs \rightarrow f[3][c].re -= gg[k].f[1][c].im;$

Now, same for $(1 - \gamma_2)$:

- 5c $\langle \text{Build } (1 - \gamma_2) \text{ projection of } *f \text{ in } *g \text{ 5c} \rangle \equiv$
 $g \rightarrow f[0][c].re = f \rightarrow f[0][c].re + f \rightarrow f[2][c].im;$
 $g \rightarrow f[0][c].im = f \rightarrow f[0][c].im - f \rightarrow f[2][c].re;$
 $g \rightarrow f[1][c].re = f \rightarrow f[1][c].re - f \rightarrow f[3][c].im;$
 $g \rightarrow f[1][c].im = f \rightarrow f[1][c].im + f \rightarrow f[3][c].re;$
- 5d $\langle \text{Unproject and accumulate } (1 - \gamma_2) \text{ link 5d} \rangle \equiv$
 $qs \rightarrow f[0][c].re += gg[k].f[0][c].re; qs \rightarrow f[2][c].im += gg[k].f[0][c].re;$
 $qs \rightarrow f[0][c].im += gg[k].f[0][c].im; qs \rightarrow f[2][c].re -= gg[k].f[0][c].im;$
 $qs \rightarrow f[1][c].re += gg[k].f[1][c].re; qs \rightarrow f[3][c].im -= gg[k].f[1][c].re;$
 $qs \rightarrow f[1][c].im += gg[k].f[1][c].im; qs \rightarrow f[3][c].re += gg[k].f[1][c].im;$

$$\gamma_3 = \sigma_1 \otimes 1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

First, the projector and the reconstructor for $(1 + \gamma_3)$:

- 5e $\langle \text{Build } (1 + \gamma_3) \text{ projection of } *f \text{ in } *g \text{ 5e} \rangle \equiv$
 $g \rightarrow f[0][c].re = f \rightarrow f[0][c].re + f \rightarrow f[2][c].re;$
 $g \rightarrow f[0][c].im = f \rightarrow f[0][c].im + f \rightarrow f[2][c].im;$
 $g \rightarrow f[1][c].re = f \rightarrow f[1][c].re + f \rightarrow f[3][c].re;$
 $g \rightarrow f[1][c].im = f \rightarrow f[1][c].im + f \rightarrow f[3][c].im;$
- 5f $\langle \text{Unproject } (1 + \gamma_3) \text{ link 5f} \rangle \equiv$
 $qs \rightarrow f[0][c].re = gg[k].f[0][c].re; qs \rightarrow f[2][c].re = gg[k].f[0][c].re;$
 $qs \rightarrow f[0][c].im = gg[k].f[0][c].im; qs \rightarrow f[2][c].im = gg[k].f[0][c].im;$
 $qs \rightarrow f[1][c].re = gg[k].f[1][c].re; qs \rightarrow f[3][c].re = gg[k].f[1][c].re;$
 $qs \rightarrow f[1][c].im = gg[k].f[1][c].im; qs \rightarrow f[3][c].im = gg[k].f[1][c].im;$

Now, same for $(1 - \gamma_3)$:

- 5g $\langle \text{Build } (1 - \gamma_3) \text{ projection of } *f \text{ in } *g \text{ 5g} \rangle \equiv$
 $g \rightarrow f[0][c].re = f \rightarrow f[0][c].re - f \rightarrow f[2][c].re;$
 $g \rightarrow f[0][c].im = f \rightarrow f[0][c].im - f \rightarrow f[2][c].im;$
 $g \rightarrow f[1][c].re = f \rightarrow f[1][c].re - f \rightarrow f[3][c].re;$
 $g \rightarrow f[1][c].im = f \rightarrow f[1][c].im - f \rightarrow f[3][c].im;$
- 5h $\langle \text{Unproject and accumulate } (1 - \gamma_3) \text{ link 5h} \rangle \equiv$
 $qs \rightarrow f[0][c].re += gg[k].f[0][c].re; qs \rightarrow f[2][c].re -= gg[k].f[0][c].re;$
 $qs \rightarrow f[0][c].im += gg[k].f[0][c].im; qs \rightarrow f[2][c].im -= gg[k].f[0][c].im;$
 $qs \rightarrow f[1][c].re += gg[k].f[1][c].re; qs \rightarrow f[3][c].re -= gg[k].f[1][c].re;$
 $qs \rightarrow f[1][c].im += gg[k].f[1][c].im; qs \rightarrow f[3][c].im -= gg[k].f[1][c].im;$

3 CONJUGATE GRADIENT

Here we develop the algorithm used in the solver.

3.1 Standard Algorithm

The basic conjugate gradient algorithm 1 is simple. Its only requirement is that matrix A is hermitian. Otherwise, it appears suited for DWF better than other iterative solvers.

```
Input:  $A$ , the matrix  
Input:  $b$ , the right hand side of the linear equation  
Input:  $x_0$ , an initial guess  
Input:  $n$ , the maximum number of iterations  
Input:  $\epsilon$ , required precision  
Output:  $x$ , approximate solution  
Output:  $\rho$ , final residue  
Output:  $k$ , number of iterations used  
begin  
   $x \leftarrow x_0$   
   $p \leftarrow r \leftarrow b - Ax$   
   $\rho \leftarrow \langle r, r \rangle$   
   $k \leftarrow 0$   
  while  $\rho > \epsilon$  or  $k < n$  do  
     $q \leftarrow Ap$   
     $\alpha \leftarrow \rho / \langle p, q \rangle$   
     $r \leftarrow r - \alpha q$   
     $x \leftarrow x + \alpha p$   
     $\gamma \leftarrow \langle r, r \rangle$   
    if  $\gamma < \epsilon$  then  
       $\rho \leftarrow \gamma$   
      break  
    end  
     $\beta \leftarrow \gamma / \rho$   
     $\rho \leftarrow \gamma$   
     $p \leftarrow r + \beta p$   
     $k \leftarrow k + 1$   
  end  
  return  $x, \rho, k$ .  
end
```

Algorithm 1: Generic Conjugate Gradient Solver

3.2 Overlap Opportunities

Our approach to overlapping computations with communications is to break the sublattice into boundary and inside pieces. After that, we first compute $(1 \pm \gamma_\mu)$ projections on the boundary and start send and receive operations. While communications are in progress, everything is computed on the inside nodes of the sublattice. Once receive is complete, we compute the operator on the boundary sites. Such an approach helps to improve temporal locality (and, therefore, cache utilization) at the expense of losing the ability of overlap if one of the sublattice dimensions is 2. However, it is unlikely that we could afford a large enough cluster to be forced into this corner of the parameter space.

3.3 Non-hermitial Matrix

Hermiticity of M is the only obstacle in applying algorithm 1 directly to our problem $M\psi = \eta$. This issue can be easily resolved by multiplying both sides by M^\dagger . However, instead of using algorithm 1 with $A = M^\dagger M$, it is better to keep M and M^\dagger separate—this makes it possible to hide one of the global sum computations, thus improving machine size scaling. Algorithm 2 is what we use in the solver.

```
Input:  $M$ , the matrix  
Input:  $b$ , the right hand side of the linear equation  
Input:  $x_0$ , an initial guess  
Input:  $n$ , the maximum number of iterations  
Input:  $\epsilon$ , required precision  
Output:  $x$ , approximate solution  
Output:  $\rho$ , final residue  
Output:  $k$ , number of iterations used  
begin  
   $x \leftarrow x_0$   
   $p \leftarrow r \leftarrow b - M^\dagger Mx$   
   $\rho \leftarrow \langle r, r \rangle$   
   $k \leftarrow 0$   
  while  $\rho > \epsilon$  or  $k < n$  do  
     $z \leftarrow Mp$   
     $q \leftarrow M^\dagger z$   
     $\alpha \leftarrow \rho / \langle z, z \rangle$   
     $r \leftarrow r - \alpha q$   
     $x \leftarrow x + \alpha p$   
     $\gamma \leftarrow \langle r, r \rangle$   
    if  $\gamma < \epsilon$  then  
       $\rho \leftarrow \gamma$   
      break  
    end  
     $\beta \leftarrow \gamma / \rho$   
     $\rho \leftarrow \gamma$   
     $p \leftarrow r + \beta p$   
     $k \leftarrow k + 1$   
  end  
  return  $x, \rho, k$ .  
end
```

Algorithm 2: DWF-ready Gradient Solver.

4 PRECONDITIONING

We use four dimensional preconditioner to improve convergence of the CG. Following Kostas Orginos, let us color the lattice sites according to the parity of $x_0 + x_1 + x_2 + x_3$. Then we can rewrite $\chi = D\psi$ as follows:

$$\begin{pmatrix} \chi_e \\ \chi_o \end{pmatrix} = D\psi = \begin{pmatrix} Q_{ee} & Q_{eo} \\ Q_{oe} & Q_{oo} \end{pmatrix} \begin{pmatrix} \psi_e \\ \psi_o \end{pmatrix}$$

From the form of D it follows that all dependance on the gauge field is located in Q_{xy} , and that Q_{xx} does not depend on U . That, in turn, allows us to invert Q_{xx} easily. With this in mind, one writes:

$$\begin{pmatrix} Q_{ee} & Q_{eo} \\ Q_{oe} & Q_{oo} \end{pmatrix} = \begin{pmatrix} Q_{ee} & 0 \\ Q_{oe} & Q_{oo} \end{pmatrix} \begin{pmatrix} 1 & Q_{ee}^{-1}Q_{eo} \\ 0 & 1 - Q_{oo}^{-1}Q_{oe}Q_{ee}^{-1}Q_{eo} \end{pmatrix}$$

Now, to solve the equation

$$D\psi = \eta,$$

one needs to perform the following steps:

1. Compute

$$\phi_o = Q_{oo}^{-1}(\eta_o - Q_{oe}Q_{ee}^{-1}\eta_e)$$

2. Set $M = 1 - Q_{oo}^{-1}Q_{oe}Q_{ee}^{-1}Q_{eo}$ for the following.

3. Compute

$$\varphi_o = M^\dagger \phi_o$$

4. Solve for ψ_o the following equation using Algorithm 2

$$M^\dagger M \psi_o = \varphi_o$$

5. Compute

$$\psi_e = Q_{ee}^{-1}(\eta_e - Q_{eo}\psi_o)$$

Note, that $M^\dagger = 1 - (Q_{eo})^\dagger(Q_{ee}^{-1})^\dagger(Q_{oe})^\dagger(Q_{oo}^{-1})^\dagger = 1 - S_{oe}S_{ee}^{-1}S_{oe}S_{oo}^{-1}$, where

$$\begin{aligned} S_{ee} &= Q_{ee}[\gamma_5 \rightarrow -\gamma_5] \\ S_{oo} &= Q_{oo}[\gamma_5 \rightarrow -\gamma_5] \\ S_{oe} &= Q_{eo}[\gamma_\mu \rightarrow -\gamma_\mu] \\ S_{eo} &= Q_{oe}[\gamma_\mu \rightarrow -\gamma_\mu] \end{aligned}$$

4.1 Q_{xx} inversion

The previous section is based on a tacit assumption that Q_{ee} and Q_{oo} are easy to invert. Here we show that it is so. Let us rewrite

$$\chi_{s,x} = (Q_{ee}\psi)_{s,x} = M_0\psi_{s,x} + (1 + \gamma_5)M_s^{(+)}\psi_{s+1,x} + (1 - \gamma_5)M_s^{(-)}\psi_{s-1,x}$$

as follows:

$$(Q_{ee}\psi)_{s,x} = M_0 \left(\left(\frac{1 + \gamma_5}{2} \right) \left(\psi_{s,x} + \frac{2M_s^{(+)}}{M_0}\psi_{s+1,x} \right) + \left(\frac{1 - \gamma_5}{2} \right) \left(\psi_{s,x} + \frac{2M_s^{(-)}}{M_0}\psi_{s-1,x} \right) \right).$$

Thus,

$$Q_{ee} = \frac{1 + \gamma_5}{2} \begin{pmatrix} a & b & \cdots & 0 & 0 \\ 0 & a & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & a & b \\ c & 0 & \cdots & 0 & a \end{pmatrix} + \frac{1 - \gamma_5}{2} \begin{pmatrix} a & 0 & \cdots & 0 & c \\ b & a & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & a & 0 \\ 0 & 0 & \cdots & b & a \end{pmatrix} = P_+A + P_-B,$$

where $a = M_0$, $b = 2$, $c = -2m_f$. Now, since P_{\pm} commute with A and B , $Q_{ee}^{-1} = P_+ A^{-1} + P_- B^{-1}$. Computing A^{-1} and B^{-1} is done by decomposition $A = L_A R_A$, $B = L_B R_B$, where

$$R_A = \begin{pmatrix} a & b & \cdots & 0 & 0 \\ 0 & a & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & a & b \\ 0 & 0 & \cdots & 0 & a \end{pmatrix} \quad R_B = \begin{pmatrix} a & 0 & \cdots & 0 & 0 \\ b & a & & 0 & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & & a & 0 \\ 0 & 0 & \cdots & b & a \end{pmatrix},$$

and

$$L_A = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & & 0 \\ \vdots & & & & & \vdots \\ c/a & -bc/a^2 & b^2c/a^3 & -b^3c/a^4 & \cdots & 1 + (-b)^{n-1}c/a^n \end{pmatrix}$$

$$L_B = \begin{pmatrix} 1 + (-b)^{n-1}c/a^n & (-b)^{n-2}c/a^{n-1} & \cdots & b^2c/a^3 & -bc/a^2 & c/a \\ 0 & 1 & & 0 & 0 & 0 \\ \vdots & & & & & \vdots \\ 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix}.$$

In these terms,

$$Q_{ee}^{-1} = \frac{1 + \gamma_5}{2} R_A^{-1} L_A^{-1} + \frac{1 - \gamma_5}{2} R_B^{-1} L_B^{-1}.$$

We will also need

$$S_{ee}^{-1} = \frac{1 - \gamma_5}{2} R_A^{-1} L_A^{-1} + \frac{1 + \gamma_5}{2} R_B^{-1} L_B^{-1}.$$

For further reference,

$$\gamma_5 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}.$$

5 CODE

This section contains chunks of the source that go into `dwf.c` source file. We start with the interface functions and elaborate from there.

5.1 Interface Functions

We can not expect the user to call different parts of the interface in an appropriate order. Therefore, successful initialization allows the user to call other interface elements, as well as prevents repeated initializations.

```
10a  <Global variables 10a>≡
      static int init_p = 0;
```

5.1.1 SSE DWF Initializer

```
10b  <Interface functions 10b>≡
      int
      SSE_DWF_init(const int lattice[DIM+1],
                    SSE_DWF_FP_SIZE fp_size,
                    void *(*allocator)(size_t size),
                    void (*dealloc)(void *))
      {
          if (init_p)
              return 1; /* error: second init */

          <Check floating point size 11a>
          <Check lattice size 11b>
          <Get network topology 17a>
          <Setup heap management functions 10d>
          <Initialize tables 21b>
          <Allocate fields 32d>
          <Initialize QMP 29e>
          init_p = 1;
          return 0;

          <Handle init errors 10c>
      }
```

If any error occurs during initialization, we simply unroll state and fail:

```
10c  <Handle init errors 10c>≡
      error:
          SSE_DWF_fini();
          return 1;
```

Check if the user requested special allocation mechanisms:

```
10d  <Setup heap management functions 10d>≡
      if (allocator)
          tmalloc = allocator;
      else
          tmalloc = malloc;

      if (dealloc)
          tfree = dealloc;
      else
          tfree = free;
```

```
10e  <Global variables 10a>+≡
      static void *(*tmalloc)(size_t size);
      static void (*tfree)(void *ptr);
```

For now we only support single precision floating point numbers:

11a $\langle \text{Check floating point size 11a} \rangle \equiv$
`if (fp_size != SSE_DWF_FLOAT)
goto error;`

For single precision arithmentics, L_s should be a muplitple of 4.

11b $\langle \text{Check lattice size 11b} \rangle \equiv$
`if (lattice[DIM] % Vs)
goto error;
tlattice[DIM] = lattice[DIM];`

Otherwise, lattice sizes must be even to allow us to do red/black preconditioning:

11c $\langle \text{Check lattice size 11b} \rangle + \equiv$
`{
int i;
for (i = 0; i < DIM; i++) {
if (lattice[i] & 1)
goto error;
tlattice[i] = lattice[i];
}
}`

11d $\langle \text{Global variables 10a} \rangle + \equiv$
`static int tlattice[DIM+1];`

5.1.2 SSE DWF Clean Up

The cleanup routine may be called from partially initialized context, we should be able to do a partial cleanup.

11e $\langle \text{Interface functions 10b} \rangle + \equiv$
`void
SSE_DWF_fini(void)
{
 $\langle \text{Cleanup QMP 65b} \rangle$
 $\langle \text{Free fields 32e} \rangle$
 $\langle \text{Free tables 65c} \rangle$
initd_p = 0;
}`

5.1.3 DWF Fermion Allocator

When one needs an SSE DWF fermion, the allocator does the job. Remember, users are stupid enough to call this function in the uninitialized state. It is convenient to break all internal fermions into odd and even parts at this stage.

11f $\langle \text{Data types 11f} \rangle \equiv$
`struct SSE_DWF_Fermion {
vEvenFermion *even;
vOddFermion *odd;
};`

Now, the fermion allocator proper:

```

12a  <Interface functions 10b>+≡
      SSE_DWF_Fermion *
      SSE_DWF_allocate_fermion(void)
      {
          SSE_DWF_Fermion *ptr;

          if (!inited_p)
              return 0;

          ptr = tmalloc(sizeof (*ptr));
          if (ptr == 0)
              return 0;

          ptr->even = allocate_even_fermion();
          if (ptr->even == 0)
              goto error1;

          ptr->odd  = allocate_odd_fermion();
          if (ptr->odd == 0)
              goto error2;

          return ptr;
      error2:
          free16(ptr->even);
      error1:
          tfree(ptr);
          return 0;
      }

```

5.1.4 DWF Fermion Exporter

When we need to create an SSE fermion field and populate it from an outer environment, we use the following procedure

```

12b  <Interface functions 10b>+≡
      SSE_DWF_Fermion *
      SSE_DWF_load_fermion(const void *OuterFermion,
                          void *env,
                          SSE_DWF_fermion_reader reader)
      {
          SSE_DWF_Fermion *ptr = SSE_DWF_allocate_fermion();

          /* Handle both lack of memory and missing initialization */
          if (ptr == 0)
              return 0;

          <Read fermion 19c>

          return ptr;
      }

```

5.1.5 DWF Fermion Importer

For moving data back to the outer environment, the following importer is used:

```
13a  <Interface functions 10b>+≡
      void
      SSE_DWF_save_fermion(void *OuterFermion,
                           void *env,
                           SSE_DWF_fermion_writer writer,
                           SSE_DWF_Fermion *CGfermion)
      {
          if (!inited_p)
              return;

          <Write fermion 20b>
      }
```

5.1.6 DWF Fermion Deallocator

We only free pointers that we allocated. The magic is in `free16()`—it knows about all heap objects allocated by `alloc16()`.

```
13b  <Interface functions 10b>+≡
      void
      SSE_DWF_delete_fermion(SSE_DWF_Fermion *ptr)
      {
          if (!inited_p)
              return;

          free16(ptr->even);
          free16(ptr->odd);
          tfree(ptr);
      }
```

5.1.7 DWF Gauge Exporter

Unlike fermions, gauge fields are 4-d in the solver. Though they are not loaded by SSE memory operations, we still allocate 16-byte aligned memory for them (apparently for no good reason at all.)

```
13c  <Interface functions 10b>+≡
      SSE_DWF_Gauge *
      SSE_DWF_load_gauge(const void *OuterGauge_U,
                         const void *OuterGauge_V,
                         void *env,
                         SSE_DWF_gauge_reader reader)
      {
          SSE_DWF_Gauge *g;

          if (!inited_p)
              return 0;

          g = allocate_gauge_field();
          if (g == 0)
              return 0;

          <Read gauge field 17c>
          return g;
      }
```

Let us also define `SSE_DWF_Gauge` here. We do not need anything fancy for the gauge field:

```
14a  <Data types 11f>+≡
      struct SSE_DWF_Gauge {
          complex v[Nc][Nc];
      };

```

5.1.8 DWF Gauge Deallocator

Gauge deallocator is very much like fermion deallocator. We only keep them separate to help the type system cope with a error making user.

```
14b  <Interface functions 10b>+≡
      void
      SSE_DWF_delete_gauge(SSE_DWF_Gauge *ptr)
      {
          if (!initied_p)
              return;

          free16(ptr);
      }

```

5.1.9 The Solver

Finally, the solver itself. Here we check if the system has been properly initialized and dispatch on the float size (but not now yet.)

```
14c  <Interface functions 10b>+≡
      int
      SSE_DWF_cg_solver(SSE_DWF_Fermion *psi,          /* result */
                        double *out_eps,
                        int *out_iter,
                        const SSE_DWF_Gauge *gauge,
                        double m0, double M,
                        const SSE_DWF_Fermion *x0,      /* guess */
                        const SSE_DWF_Fermion *eta,     /* rhs */
                        double eps, int max_iter)
      {
          int status;

          if (!initied_p)
              return 1;

          U = (SU3 *)gauge;
          <Compute constant values for  $Q_{xx}^{-1}$  and  $S_{xx}^{-1}$  59c>
          <Compute  $\varphi_o$  32b>
          <Solve  $M^\dagger M \psi_o = \varphi_o$  32f>
          <Compute  $\psi_e$  34a>
          return status;
      }

```

Save one argument in many functions:

```
14d  <Global variables 10a>+≡
      static SU3 *U;

```

5.2 Memory Allocation

SSE does like properly aligned memory. While automatic variables are aligned by the compiler, extra care is needed when dealing with the heap. The code allocates all its own memory aligned on 16-byte boundary by calling `alloc16()`, and returns the memory through `free16()`.

```
15a  <Static functions 15a>≡
      static void *
      alloc16(int size)
      {
          int xsize = PAD16(size + sizeof (struct memblock));
          struct memblock *p = tmalloc(xsize);

          if (p == 0)
              return p;

          p->data = ALIGN16(&p[1]);
          p->size = size;
          p->next = memblock.next;
          p->prev = &memblock;
          p->next->prev = p;
          p->prev->next = p;

          return p->data;
      }
```

For readability, here are alignment operations:

```
15b  <Macro definitions 15b>≡
      #define PAD16(size) (15+(size))
      #define ALIGN16(addr) ((void *) (~15 & (15 + (size_t)(addr))))
```

For deallocation we need to find an appropriate memory block:

```
15c  <Static functions 15a>+≡
      static void
      free16(void *ptr)
      {
          struct memblock *p;

          if (ptr == 0)
              return;

          for (p = memblock.next; p != &memblock; p = p->next) {
              if (p->data != ptr)
                  continue;
              p->next->prev = p->prev;
              p->prev->next = p->next;
              tfree(p);
              return;
          }
          /* this is BAD: control should reach here! */
      }
```

The head of the memory list is stored in a static variable. Of course, such an implementation makes no threadable, but let us worry about that when the time is right.

```
15d  <Global variables 10a>+≡
      static struct memblock memblock = {
          &memblock,
          &memblock,
          NULL,
          0
      };
```


Finally, the datatype for the linked list:

```
16a  <Data types 11f>+≡
      struct memblock {
          struct memblock *next;
          struct memblock *prev;
          void *data;
          size_t size;
      };
```

5.2.1 Field allocators

First, the prototypes:

```
16b  <Static function prototypes 16b>≡
      static vEvenFermion *allocate_even_fermion(void);
      static vOddFermion *allocate_odd_fermion(void);
      static SSE_DWF_Gauge *allocate_gauge_field(void);
      /*
      vFermion *allocate_subfermion(int size);
      */
```

The only difference between even and odd fermions is (possibly) their size:

```
16c  <Static functions 15a>+≡
      vEvenFermion *
      allocate_even_fermion(void)
      {
          return alloc16(even_odd.size * S_4 * sizeof (vFermion));
      }

      vOddFermion *
      allocate_odd_fermion(void)
      {
          return alloc16(odd_even.size * S_4 * sizeof (vFermion));
      }

      SSE_DWF_Gauge *
      allocate_gauge_field(void)
      {
          return alloc16(gauge_XYZT * sizeof (SSE_DWF_Gauge));
      }
```

5.3 Probing Cluster Topology

There is no proper way to query QMP about lattice layout. We have to request the minimal meaningful information the library provides and try to repeat outer layer's partitioning of the lattice. There are good chances of success, but this is a potential danger spot.

Here we prepare compute where on the lattice this node is and to build up our understanding of neighbors. Maybe optimistically, we assume that once QMP is initialized, it reports logical dimensions and coordinates properly, so that we do not need to be paranoid about errors here.

```
17a  <Get network topology 17a>≡
      {
          int i, dn;
          const QMP_u32_t *xn, *xc;

          if (!QMP_logical_topology_is_declared())
              /* The user must have declared logical topology before */
              goto error;
          dn = QMP_get_logical_number_of_dimensions();
          if (dn > DIM)
              /* Too high dimension of the logical network */
              goto error;

          xn = QMP_get_logical_dimensions();
          xc = QMP_get_logical_coordinates();
          for (i = 0; i < dn; i++) {
              network[i] = xn[i];
              coord[i] = xc[i];
          }

          for (; i < dn; i++) {
              network[i] = 1;
              coord[i] = 0;
          }
      }
      Some global variables:
17b  <Global variables 10a>+≡
      static int network[DIM];
      static int coord[DIM];
```

5.4 Moving Data

5.4.1 Reading the Gauge Field

Let us start with reading of the gauge field from the outer environment first. Here we assume that there is an address translation function to help us in talking to the outer layer.

```
17c  <Read gauge field 17c>≡
      {
          int x[DIM], i, d, a, b, p1;

          <Start DIM-d sublattice scan 18b>
          <Load DIM gauge links from U at x 18a>
          <Advance DIM-d index for full sublattice scan 18d>

          for (d = 0; d < DIM; d++)
              <Load gauge boundary in direction d 19a>
      }
```

At a given site, load DIM gauge elements:

```

18a  <Load DIM gauge links from U at x 18a>≡
      p1 = gauge4to1(x);
      for (d = 0; d < DIM; d++) {
        for (a = 0; a < Nc; a++) {
          for (b = 0; b < Nc; b++) {
            g[p1 + d].v[a][b].re = reader(OuterGauge_U, env, x, d, a, b, 0);
            g[p1 + d].v[a][b].im = reader(OuterGauge_U, env, x, d, a, b, 1);
          }
        }
      }

```

To start a scan over the lattice, initialize x and start the loop:

```

18b  <Start DIM-d sublattice scan 18b>≡
      for (i = 0; i < DIM; i++)
        x[i] = bounds.lo[i];
      for (i = 0; i < DIM;) {

18c  <Start DIM-d sublattice scan locally 18c>≡
      for (i = 0; i < DIM; i++)
        x[i] = bounds->lo[i];
      for (i = 0; i < DIM;) {

```

Once all is done with the site x, we are ready to advance the index:

```

18d  <Advance DIM-d index for full sublattice scan 18d>≡
      for (i = 0; i < DIM; i++) {
        <Advance x at i 18g>
      }

18e  <Advance DIM-d index for full sublattice scan locally 18e>≡
      for (i = 0; i < DIM; i++) {
        <Advance x at i locally 18h>
      }

```

Since we are going to use a DIM-1 dimensional scan as well, let us write it down here:

```

18f  <Advance DIM-d index for DIM-1-d scan 18f>≡
      for (i = 0; i < DIM; i++) {
        if (i == d)
          continue;
        <Advance x at i 18g>
      }

```

Now we can scan DIM-dimensional indices:

```

18g  <Advance x at i 18g>≡
      if (++x[i] == bounds.hi[i])
        x[i] = bounds.lo[i];
      else
        break;

18h  <Advance x at i locally 18h>≡
      if (++x[i] == bounds->hi[i])
        x[i] = bounds->lo[i];
      else
        break;

```

DWF Dirac operator needs backward gauge links. We get them from `OuterGauge_V`. Here we only read the boundary links.

```
19a  <Load gauge boundary in direction d 19a>≡
      {
          if (network[d] == 1)
              continue;

          <Start DIM-d sublattice scan 18b>
          <Load a d gauge link from V at x 19b>
          <Advance DIM-d index for DIM-1-d scan 18f>
      }
```

Now we read a boundary element:

```
19b  <Load a d gauge link from V at x 19b>≡
      x[d] = bounds.lo[d] - 1;
      p1 = gauge4to1(x) + d;
      x[d] = bounds.lo[d];
      for (a = 0; a < Nc; a++) {
          for (b = 0; b < Nc; b++) {
              g[p1].v[a][b].re = reader(OuterGauge_V, env, x, d, a, b, 0);
              g[p1].v[a][b].im = reader(OuterGauge_V, env, x, d, a, b, 1);
          }
      }
```

5.4.2 Reading a Fermion

There are but two complications in reading the domain wall fermion. First, this is a good time to break the fermion into red and black pieces. In addition, here we construct SSE fermions.

```
19c  <Read fermion 19c>≡
      {
          int x[DIM+1], i;

          <Start DIM-d sublattice scan 18b>
          <Load an s-line of fermion at x 19d>
          <Advance DIM-d index for full sublattice scan 18d>
      }
```

Data conversion is inherently inefficient. We do not try to overoptimize it here:

```
19d  <Load an s-line of fermion at x 19d>≡
      {
          int p = parity(x);
          int p1 = p? hf_even4to1(x): hf_odd4to1(x);
          vFermion *f = p? &ptr->even[p1].f: &ptr->odd[p1].f;

          for (x[DIM] = 0; x[DIM] < tlattice[DIM]; x[DIM] += Vs, f++) {
              int d;
              for (d = 0; d < Fd; d++) {
                  int c;
                  for (c = 0; c < Nc; c++) {
                      f->f[d][c].re = import_vector(OuterFermion, env, reader,
                                                         x, c, d, 0);
                      f->f[d][c].im = import_vector(OuterFermion, env, reader,
                                                         x, c, d, 1);
                  }
              }
          }
      }
```

A simple packer of Vs elements into a vector:

```
20a  <Static function prototypes 16b>+≡
      static inline vReal
      import_vector(const void *z, void *env, SSE_DWF_fermion_reader reader,
                    int x[DIM+1], int c, int d, int re_im)
      {
          vReal f;
          REAL *v = (REAL *)&f;
          int i, xs;

          for (xs = x[DIM], i = 0; i < Vs; i++, x[DIM]++) {
              *v++ = reader(z, env, x, c, d, re_im);
          }
          x[DIM] = xs;
          return f;
      }
```

5.4.3 Writing a Fermion

Writing a fermion is not much different:

```
20b  <Write fermion 20b>≡
      {
          int x[DIM+1], i;

          <Start DIM-d sublattice scan 18b>
          <Save an s-line of fermion at x 20c>
          <Advance DIM-d index for full sublattice scan 18d>
      }

20c  <Save an s-line of fermion at x 20c>≡
      {
          int p = parity(x);
          int p1 = p? hf_odd4to1(x): hf_even4to1(x);
          vFermion *f = p? &CGfermion->odd[p1].f: &CGfermion->even[p1].f;

          for (x[DIM] = 0; x[DIM] < tlattice[DIM]; x[DIM] += Vs, f++) {
              int d;
              for (d = 0; d < Fd; d++) {
                  int c;
                  for (c = 0; c < Nc; c++) {
                      save_vector(OuterFermion, env, writer, x, c, d, 0,
                                f->f[d][c].re);
                      save_vector(OuterFermion, env, writer, x, c, d, 1,
                                f->f[d][c].im);
                  }
              }
          }
      }
```

Here's another little helper good only for writing back the fermion from SSE to the outer environment:

```
21a  <Static function prototypes 16b>+≡
      static inline void
      save_vector(void *z, void *env, SSE_DWF_fermion_writer writer,
                  int x[DIM+1], int c, int d, int re_im, vReal f)
      {
          REAL *v = (REAL *)&f;
          int i, xs;

          for (xs = x[DIM], i = 0; i < Vs; i++, x[DIM]++) {
              writer(z, env, x, c, d, re_im, *v++);
          }
          x[DIM] = xs;
      }
```

5.5 Solver Initialization

Here are all pieces for setting up the structures needed to run the solver.

5.5.1 Constructing the neighbor tables

```
21b  <Initialize tables 21b>≡
      if (init_tables()) {
          /* Something went wrong in the table construction */
          goto error;
      }
```

The table initializer creates all tables necessary for communication and computation. Memory is allocated here for index arrays.

```
21c  <Static functions 15a>+≡
      static int
      init_tables(void)
      {
          struct neighbor tmp;
          int i, v;

          init_neighbor(&bounds, &neighbor);
          <Compute init sizes 22a>
          tmp = neighbor;
          build_neighbor(&even_odd, &bounds, 0, &tmp);
          build_neighbor(&odd_even, &bounds, 1, &tmp);

          return 0;
      }
```

First, we set global data:

```
21d  <Global variables 10a>+≡
      static struct bounds bounds;
      static int gauge_XYZT;
      static int S_4, S_4_1;
```

22a *⟨Compute init sizes 22a⟩*≡

```

S_4 = tlattice[DIM] / 4;
S_4_1 = S_4 - 1;
for (v = 1, i = 0; i < DIM; i++) {
    v *= bounds.hi[i] - bounds.lo[i];
}
gauge_XYZT = DIM * v;
for (i = 0; i < DIM; i++) {
    if (network[i] < 2)
        continue;
    gauge_XYZT += v / (bounds.hi[i] - bounds.lo[i]);
}

```

The `struct bounds` helps us to navigate through the local part of the lattice. It is used by the initialization code only.

22b *⟨Data types 11f⟩*+≡

```

struct bounds {
    int lo[DIM];
    int hi[DIM];
};

```

We keep two `struct neighbor`, one for computation on the even sublattice, another—on the odd. In addition to `even_odd` and `odd_even`, we need one more `struct neighbor` to keep the allocated pointers in.

22c *⟨Global variables 10a⟩*+≡

```

static struct neighbor neighbor;
static struct neighbor odd_even;
static struct neighbor even_odd;

```

Let us start with computing the boundary of the sublattice

22d *⟨Static function prototypes 16b⟩*+≡

```

static inline int
lattice_start(int lat, int net, int coord)
{
    int q = lat / net;
    int r = lat % net;

    return coord * q + ((coord < r)? coord: r);
}

static inline void
mk_sublattice(struct bounds *bounds,
              int coord[])
{
    int i;

    for (i = 0; i < DIM; i++) {
        bounds->lo[i] = lattice_start(tlattice[i], network[i], coord[i]);
        bounds->hi[i] = lattice_start(tlattice[i], network[i], coord[i] + 1);
    }
}

```

All dynamic data are allocated in `init_neighbor` and are stored in `neighbor`.

22e *⟨Static function prototypes 16b⟩*+≡

```

static void
init_neighbor(struct bounds *bounds, struct neighbor *neighbor);

```

```

23a  <Static functions 15a>+≡
      static void
      init_neighbor(struct bounds *bounds, struct neighbor *neighbor)
      {
          int i;

          mk_sublattice(bounds, coord);
          neighbor->qmp_smask = 0;
          <Compute inside_size and boundary_size 23b>
          <Allocate inside table 23c>
          <Allocate boundary table 23d>
          <Compute send sizes and allocate index tables 23e>
      }

23b  <Compute inside_size and boundary_size 23b>≡
      for (neighbor->size = 1, neighbor->inside_size = 1, i = 0; i < DIM; i++) {
          int ext = bounds->hi[i] - bounds->lo[i];

          neighbor->size *= ext;
          if (network[i] > 1)
              neighbor->inside_size *= ext - 2;
          else
              neighbor->inside_size *= ext;
      }
      neighbor->boundary_size = neighbor->size - neighbor->inside_size;
      neighbor->site = tmalloc(neighbor->size * sizeof (struct site));

23c  <Allocate inside table 23c>≡
      if (neighbor->inside_size)
          neighbor->inside = tmalloc(neighbor->inside_size * sizeof (int));
      else
          neighbor->inside = 0;

23d  <Allocate boundary table 23d>≡
      if (neighbor->boundary_size)
          neighbor->boundary = tmalloc(neighbor->boundary_size * sizeof (struct boundary));
      else
          neighbor->boundary = 0;

23e  <Compute send sizes and allocate index tables 23e>≡
      for (i = 0; i < 2 * DIM; i++) {
          int d = i / 2;

          if (network[d] > 1) {
              neighbor->snd_size[i] = neighbor->size / (bounds->hi[d] - bounds->lo[d]);
              neighbor->snd[i] = tmalloc(neighbor->snd_size[i] * sizeof (int));
          } else {
              neighbor->snd_size[i] = 0;
              neighbor->snd[i] = 0;
          }
      }

```


Here is the definition of the neighbor table we spent soo much time initializing:

```

24a  <Data types 11f>+≡
      struct neighbor {
          int      size;           /* size of site table */
          int      inside_size;    /* number of inside sites */
          int      boundary_size;  /* number of boundary sites */
          int      snd_size[2*DIM]; /* size of send buffers in 8 dirs */
          int      rcv_size[2*DIM]; /* size of receive buffers */
          int      *snd[2*DIM];    /* i->x translation for send buffers */
          int      *inside;        /* i->x translation for inside sites */
          struct boundary *boundary; /* i->x,mask translation for boundary */
          struct site *site;       /* x->site translation for sites */
          vHalfFermion *snd_buf[2*DIM]; /* Send buffers */
          vHalfFermion *rcv_buf[2*DIM]; /* Receive buffers */

          int      qmp_size[4*DIM]; /* sizes of QMP buffers */
          vHalfFermion *qmp_buf[4*DIM]; /* send and receive buffers for QMP */
          QMP_msgmem_t qmp_mm[4*DIM]; /* msgmem's for send and receive */
          int      Nx;             /* number of msecs */

          QMP_msghandle_t qmp_sh[2*DIM]; /* handles for sends */
          QMP_msghandle_t qmp_sv[2*DIM]; /* copies of handles for finilization */
          int      qmp_smask;        /* send flags for qmp_sh[] */
          int      Ns;              /* number of send handles */

          QMP_msghandle_t qmp_rh[2*DIM]; /* handles for receives */
          int      Nr;              /* number of receive handles */
          QMP_msghandle_t qmp_cr;      /* common receive handle */
      };

```

For boundary sites we only need 8 bits for the boundary indicators. However, allocating a whole int for mask is what the compiler does anyway.

```

24b  <Data types 11f>+≡
      struct boundary {
          int      index;          /* x-index of this boundary site */
          int      mask;          /* bitmask of the borders */
      };

```

In the following structure we keep information about links and neighbors of the site. Note, that there is one address for four forward links: they are packed in memory as defined in the comment.

```

24c  <Data types 11f>+≡
      struct site {
          int      Uup;            /* up-links are Uup, Uup+1, Uup+2, Uup+3 */
          int      Udown[DIM];    /* four down-links */
          int      F[2*DIM];      /* eight neighboring fermions on the other sublattice */
      };

```

Now we can define `build_neighbor()`:

```

25a  <Static functions 15a>+≡
      static void
      build_neighbor(struct neighbor *out,
                     struct bounds  *bounds,
                     int             par,
                     struct neighbor *in)
      {
          int i,d, s, p, m;
          int x[DIM];

          <Initialize out and p 25c>
          <Walk through sublattice 25d>
          <Build outside indices 26e>
      }

25b  <Static function prototypes 16b>+≡
      static void build_neighbor(struct neighbor *out,
                                struct bounds  *bounds,
                                int             parity,
                                struct neighbor *in);

```

First part is easy: we start with copying `in` to `out`, resetting fields which will be computed shortly and setting `p` to `bounds->lo`:

```

25c  <Initialize out and p 25c>≡
      *out = *in;
      out->size = 0;
      out->inside_size = 0;
      out->boundary_size = 0;
      for (d = 0; d < DIM; d++) {
          out->rcv_size[2*d] = out->snd_size[2*d] = 0;
          out->rcv_size[2*d+1] = out->snd_size[2*d+1] = 0;
      }

```

This is a good place to reuse our lattice walking chunks.

```

25d  <Walk through sublattice 25d>≡
      <Start DIM-d sublattice scan locally 18c>
          s = parity(x);
          if (s != par)
              goto next;
          <Compute p and m 25e>
          <Setup boundary or inside 26a>
          <Build local neighbors 26d>
          out->size++;
          in->size++;
      next:
      <Advance DIM-d index for full sublattice scan locally 18e>

```

For `p` we use a function to compute it from `x`. As for `m`, its eight low bits encode if there is a boundary nearby. Note, that even bits corresponds to *step down* and odd bits correspond to *step up*.

```

25e  <Compute p and m 25e>≡
      p = to_HFlinear(x, bounds, -1, 0);
      for (m = 0, d = 0; d < DIM; d++) {
          if (network[d] > 1) {
              if (x[d] == bounds->lo[d])
                  m |= 1 << (2 * d);
              if (x[d] + 1 == bounds->hi[d])
                  m |= 1 << (2 * d + 1);
          }
      }

```

If no boundary was found near **p**, we put it into **inside**. Otherwise, **p** belongs to the boundary.

```
26a  <Setup boundary or inside 26a>≡
      if (m) {
          <Setup boundary 26c>
      } else {
          <Setup inside 26b>
      }
```

For the inside, simply add **p** to the list of sites and advance pointers and counters:

```
26b  <Setup inside 26b>≡
      *in->inside++ = p;
      out->inside_size++;
```

For the boundary, place **p** into **index** and **m** into **mask** and advance pointers. We also take the opportunity to place **p** into send buffers where bits of **m** are set

```
26c  <Setup boundary 26c>≡
      in->boundary->index = p;
      in->boundary->mask = m;
      in->boundary++;
      out->boundary_size++;
      for (d = 0; d < 2*DIM; d++) {
          if ((m & (1 << d)) == 0)
              continue;
          *in->snd[d]++ = p;
          out->snd_size[d]++;
      }
```

We are ready now to build local neighbors. All gauge fields are local, and we still have **m** to tell if the other sublattice neighbor is local or not.

```
26d  <Build local neighbors 26d>≡
      in->site->Uup = to_Ulinear(x, bounds, -1);
      for (d = 0; d < DIM; d++) {
          in->site->Udown[d] = to_Ulinear(x, bounds, d);
          if ((m & (1 << (2 * d))) == 0)
              in->site->F[2*d] = S_4 * to_HFlinear(x, bounds, d, -1);
          if ((m & (1 << (2 * d + 1))) == 0)
              in->site->F[2*d + 1] = S_4 * to_HFlinear(x, bounds, d, +1);
      }
```

The only piece left is the one dealing with outside indices. This is a tricky part, but we just happen to have almost enough machinery already to solve it:

```
26e  <Build outside indices 26e>≡
      for (d = 0; d < DIM; d++) {
          if (network[d] < 2)
              continue;
          construct_rec(out, par, bounds, d, +1);
          construct_rec(out, par, bounds, d, -1);
      }
```

We also need a function that will walk through a boundary of a neighbor building the outside part of the **site[]**.F indices.

```
26f  <Static function prototypes 16b>+≡
      static void construct_rec(struct neighbor *out,
                              int par,
                              struct bounds *bounds,
                              int dir,
                              int step);
```

```

27a  <Static functions 15a>+≡
      static void
      construct_rec(struct neighbor *out,
                    int par,
                    struct bounds *bounds,
                    int dir,
                    int step)
      {
          struct bounds xb;
          int xc[DIM], x[DIM];
          int s, d, p, k;
          int dz = dir * 2 + ((step>0)?1:0);

          <Construct the neighbor's network coordinates xc and bounds xb 27b>
          <Construct the initial point of the hypersurface 27c>
          <Walk through the hypersurface 28a>
      }

```

Constructing the neighbor's network position is straightforward:

```

27b  <Construct the neighbor's network coordinates xc and bounds xb 27b>≡
      for (d = 0; d < DIM; d++) {
          int v = coord[d] + ((d==dir)?step:0);

          if (v < 0)
              v += network[d];
          if (v >= network[d])
              v -= network[d];
          xc[d] = v;
      }
      mk_sublattice(&xb, xc);

```

The initial point should be on the surface we are walking:

```

27c  <Construct the initial point of the hypersurface 27c>≡
      for (d = 0; d < DIM; d++)
          x[d] = ((d == dir) && (step < 0)) ? (xb.hi[d] - 1) : xb.lo[d];

```

Walking through the hypersurface is very much like walking through the sublattice below. There are only two differences: (a) we are walking opposite parity sublattice surface here and, (b) while advancing the point, we should stay on the surface selected above.

```

28a  <Walk through the hypersurface 28a>≡
      /* ZZZ: This needs some cleaning */
      k = 0;
      do {
          for (d = 0, s = par; d < DIM; d++)
              s += x[d];
          if (!(s & 1))
              goto next;

          <Translate x to target p 28b>
          <Insert k into site[p].F[dx] 28c>

          next:
          for (d = 0; d < DIM; d++) {
              if (d == dir)
                  continue;
              if (++x[d] == xb.hi[d])
                  x[d] = xb.lo[d];
              else
                  break;
          }
          } while (d != DIM);
          out->rcv_size[dz^1] = k; /* XXX is it true? */
28b  <Translate x to target p 28b>≡
      p = to_HFlinear(x, bounds, dir, -step);
28c  <Insert k into site[p].F[dx] 28c>≡
      out->site[p].F[dz] = S_4 * k++;

```

5.5.2 Address translation routines

Let us define a couple of functions for translating 4-d lattice positions into 1-d offsets.

Computing linear position on the sublattice is used often enough to be placed in a function. To avoid writing two very similar functions, we pass two arguments q , and z to specify that q -component of p should adjusted by z . If $q < 0$, q and z are ignored.

```

28d  <Static function prototypes 16b>+≡
      static int
      to_HFlinear(int p[],
                  struct bounds *b,
                  int q,
                  int z)
      {
          int x, d;
          for (x = 0, d = 4; d--;) {
              int v = p[d] + ((d == q)?z:0);
              int s = b->hi[d] - b->lo[d];
              if (v < 0)
                  v += tlattice[d];
              if (v >= tlattice[d])
                  v -= tlattice[d];
              x = x * s + v - b->lo[d];
          }
          return x / 2;
      }

```

Computing the index of the gauge link is similar to `to_HFlinear`, except that the extra parameter `q` tells us which of `p` should be stepped down by one. If $q < 0$, we are computing forward link position.

```
29a  <Static function prototypes 16b>+≡
      static int
      to_Ulinear(int p[],
                  struct bounds *b,
                  int q)
      {
          int x, d;

          if ((q < 0) || (p[q] > b->lo[q]) || (network[q] < 2)) {
              <Find index of a regular gauge link 29b>
          } else {
              <Find index of a borrowed gauge link 29c>
          }
      }
```

Regular gauge links sits four per site and their indices are easy to compute:

```
29b  <Find index of a regular gauge link 29b>≡
      for (x = 0, d = 4; d--;) {
          int s = b->hi[d] - b->lo[d];
          int v = p[d] - ((q == d)?1:0);
          if (v < 0)
              v += tlattice[d];
          x = x * s + v - b->lo[d];
      }
      return 4 * x + ((q < 0)?0:q);
```

For borrowed links we need first to skip all regulars and previous faces and then count position on the borrowed 3-face:

```
29c  <Find index of a borrowed gauge link 29c>≡
      int s0, v0;
      for (d = 0, v0 = 1; d < 4; d++)
          v0 *= b->hi[d] - b->lo[d];
      for (d = 0, s0 = 4 * v0; d < q; d++)
          s0 += v0 / (b->hi[d] - b->lo[d]);
      for (d = 4, x = 0; d--;) {
          int s = b->hi[d] - b->lo[d];
          int v = p[d];

          if (d == q)
              continue;
          x = x * s + v - b->lo[d];
      }
      return s0 + x;
```

5.6 QMP Initialization

```
29d  <Include files 29d>≡
      #include <qmp.h>
```

Once the tables and sizes are known, allocate all send and receive buffers and register them with QMP.

```
29e  <Initialize QMP 29e>≡
      if (build_buffers(&even_odd)) goto error;
      if (build_buffers(&odd_even)) goto error;
```

There are three cases we need to consider when preparing the communication handles. Note: return 1 if there was trouble.

```
29f  <Static function prototypes 16b>+≡
      static int build_buffers(struct neighbor *nb);
```

```

30a  <Static functions 15a>+≡
      static int
      build_buffers(struct neighbor *nb)
      {
          int i, k, Nr;
          QMP_msghandle_t Rh[8];

          Nr = nb->Ns = nb->Nx = 0;
          for (i = 0; i < 4; i++) {
              switch (network[i]) {
                  case 1: break;
                  case 2:
                      <Clump up and down directions 30b>
                      break;
                  default:
                      /* Order here is important */
                      <Allocate down buffers 30d>
                      <Allocate up buffers 30c>
                      break;
              }
          }
          <Construct the collective handle 31f>
          return 0;
      }

```

If there is only two nodes in a direction, we use only up link to communicate (because there is only one wire between the nodes.)

```

30b  <Clump up and down directions 30b>≡
      k = make_buffer(nb, nb->snd_size[2*i] + nb->snd_size[2*i+1]);
      nb->snd_buf[2*i] = nb->qmp_buf[k];
      nb->snd_buf[2*i+1] = nb->snd_buf[2*i] + S_4 * nb->snd_size[2*i];
      make_send(nb, k, i, +1);

      k = make_buffer(nb, nb->rcv_size[2*i] + nb->rcv_size[2*i+1]);
      nb->rcv_buf[2*i] = nb->qmp_buf[k];
      nb->rcv_buf[2*i+1] = nb->snd_buf[2*i] + S_4 * nb->snd_size[2*i];
      Nr = make_receive(nb, k, i, -1, Rh, Nr); /* -1 here helps with a bug in GigE QMP */

```

On a large machine, up and down buffers are separate:

```

30c  <Allocate up buffers 30c>≡
      k = make_buffer(nb, nb->snd_size[2*i+1]);
      nb->snd_buf[2*i+1] = nb->qmp_buf[k];
      make_send(nb, k, i, +1);

      k = make_buffer(nb, nb->rcv_size[2*i+1]);
      nb->rcv_buf[2*i+1] = nb->qmp_buf[k];
      Nr = make_receive(nb, k, i, +1, Rh, Nr);

30d  <Allocate down buffers 30d>≡
      k = make_buffer(nb, nb->snd_size[2*i]);
      nb->snd_buf[2*i] = nb->qmp_buf[k];
      make_send(nb, k, i, -1);

      k = make_buffer(nb, nb->rcv_size[2*i]);
      nb->rcv_buf[2*i] = nb->qmp_buf[k];
      Nr = make_receive(nb, k, i, -1, Rh, Nr);

```

Allocate a buffer of size vHalfFermion's fit for send and/or receive.

```

30e  <Static function prototypes 16b>+≡
      static int make_buffer(struct neighbor *nb, int size);

```

31a $\langle \text{Static functions 15a} \rangle + \equiv$

```
static int
make_buffer(struct neighbor *nb, int size)
{
    int bcount = size * S_4 * sizeof (vHalfFermion);
    int N = nb->Nx;

    nb->qmp_size[N] = size;
    nb->qmp_buf[N] = sse_aligned_buffer(bcount);
    nb->qmp_mm[N] = QMP_declare_msgmem(nb->qmp_buf[N], bcount);
    nb->Nx = N + 1;

    return N;
}
```

Construct a send handle. This function also places a copy of the send handle into a proper place and sets a bit in `qmp_smask`.

31b $\langle \text{Static function prototypes 16b} \rangle + \equiv$

```
static void make_send(struct neighbor *nb, int k, int i, int d);
```

31c $\langle \text{Static functions 15a} \rangle + \equiv$

```
static void
make_send(struct neighbor *nb, int k, int i, int d)
{
    QMP_msghandle_t h = QMP_declare_send_relative(nb->qmp_mm[k], i, d, 1);
    int j = 2 * i + ((d < 0)? 0: 1);

    nb->qmp_sh[j] = h;
    nb->qmp_sv[nb->Ns++] = h;
    nb->qmp_smask |= (1 << j);
}
```

Constructing a receive handle is similar. We increment `Nr` to keep the count of filled positions in `Rh`.

31d $\langle \text{Static function prototypes 16b} \rangle + \equiv$

```
static int make_receive(struct neighbor *nb, int k, int i, int d,
                      QMP_msghandle_t Rh[2*DIM], int Nr);
```

31e $\langle \text{Static functions 15a} \rangle + \equiv$

```
static int
make_receive(struct neighbor *nb, int k, int i, int d,
            QMP_msghandle_t Rh[2*DIM], int Nr)
{
    Rh[Nr] = QMP_declare_receive_relative(nb->qmp_mm[k], i, d, 1);
    return Nr+1;
}
```

Finally, aggregate all receive handles:

31f $\langle \text{Construct the collective handle 31f} \rangle \equiv$

```
nb->qmp_cr = QMP_declare_multiple(Rh, Nr);
```

SSE likes its memory aligned at 16 bytes. We need to keep that in mind when asking for QMP memory. Note, that this function may be in violation of a strict interpretation of the QMP Specification, but on many SciDAC calls numerous assurances were given that such usage is permissible.

31g $\langle \text{Static function prototypes 16b} \rangle + \equiv$

```
static void *sse_aligned_buffer(int size);
```



```

32a  <Static functions 15a>+≡
      static void *
      sse_aligned_buffer(int size)
      {
        int xcount = size + 15;
        char *ptr = QMP_allocate_aligned_memory(xcount);

        return (void *) (~15 & (15 + (unsigned long)(ptr)));
      }

```

5.7 Parts of the Solver

Here are three principal parts of the solver. First, we compute the right hand side of the equation to be solved by the CG. Next, there is a solver of a hermitian matrix. Finally, the second half of the solution is computed.

5.7.1 Compute the RHS

Here we perform steps 1–3 of the outline above.

```

32b  <Compute  $\varphi_o$  32b>≡
      compute_Qee1(auxA_e, eta->even);
      compute_Qoe(auxB_o, auxA_e);
      compute_sum_o(auxA_o, eta->odd, -1, auxB_o);
      compute_Qoo1(auxB_o, auxA_o);
      compute_Mx(Phi_o, auxB_o);

32c  <Global variables 10a>+≡
      static vOddFermion *auxA_o, *auxB_o, *Phi_o;
      static vEvenFermion *auxA_e;

32d  <Allocate fields 32d>≡
      Phi_o = allocate_odd_fermion(); if (Phi_o == 0) goto error;
      auxA_o = allocate_odd_fermion(); if (auxA_o == 0) goto error;
      auxB_o = allocate_odd_fermion(); if (auxB_o == 0) goto error;
      auxA_e = allocate_even_fermion(); if (auxA_e == 0) goto error;

32e  <Free fields 32e>≡
      if (auxA_e) free16(auxA_e); auxA_e = 0;
      if (auxB_o) free16(auxB_o); auxB_o = 0;
      if (auxA_o) free16(auxA_o); auxA_o = 0;
      if (Phi_o) free16(Phi_o); Phi_o = 0;

```

5.8 Field Operations

Hermitian solver follows:

```

32f  <Solve  $M^\dagger M \psi_o = \varphi_o$  32f>≡
      status = cg(psi->odd, Phi_o, x0->odd, eps, max_iter, out_eps, out_iter);

32g  <Static function prototypes 16b>+≡
      static int cg(vOddFermion *psi,
                    const vOddFermion *b,
                    const vOddFermion *x0,
                    double epsilon, int max_iter,
                    double *out_eps, int *out_iter);

```

```

33a  <Static functions 15a>+≡
      static int
      cg(vOddFermion *x_o,
         const vOddFermion *b,
         const vOddFermion *x0,
         double epsilon, int N,
         double *out_eps, int *out_N)
      {
        double rho, alpha, beta, gamma, norm_z;
        int status = 1;
        int k;

        copy_o(x_o, x0);
        compute_MxM(p_o, &norm_z, x_o);
        compute_sum_oN(r_o, &rho, b, -1, p_o);
        copy_o(p_o, r_o);
        <Finalize <r,r> computation 59e>

        for (k = 0; (rho > epsilon) && (k < N); k++) {
          compute_MxM(q_o, &norm_z, p_o);
          <Finalize <r,r> computation 59e>
          alpha = rho / norm_z;
          compute_sum2_oN(r_o, &gamma, -alpha, q_o);
          compute_sum2_o(x_o, alpha, p_o);
          <Finalize <r,r> computation 59e>
          if (gamma < epsilon) {
            rho = gamma;
            status = 0;
            break;
          }
          beta = gamma / rho;
          rho = gamma;
          compute_sum2x_o(p_o, r_o, beta);
        }
        <Finish old off-diagonal sends 60b>
        *out_N = k;
        *out_eps = rho;

        return status;
      }

Temporaries used by the CG
33b  <Global variables 10a>+≡
      static vOddFermion *r_o, *p_o, *q_o;

33c  <Allocate fields 32d>+≡
      r_o = allocate_odd_fermion(); if (r_o == 0) goto error;
      p_o = allocate_odd_fermion(); if (p_o == 0) goto error;
      q_o = allocate_odd_fermion(); if (q_o == 0) goto error;

33d  <Free fields 32e>+≡
      if (r_o) free16(r_o); r_o = 0;
      if (p_o) free16(p_o); p_o = 0;
      if (q_o) free16(q_o); q_o = 0;

```

5.8.1 Computing the even part of the result

Again, this is simpling performing step 5 of the outline above:

```

34a  <Compute  $\psi_e$  34a>≡
      compute_Qeo(auxA_e, psi->odd);
      compute_sum_e(auxB_e, eta->even, -1, auxA_e);
      compute_Qee1(psi->even, auxB_e);

34b  <Global variables 10a>+≡
      vEvenFermion *auxB_e;

/* XXX allocate me! */

```

5.8.2 copy_o(d, s) or $d \leftarrow s$

This is a copies $d \leftarrow s$. Since it is used outside of the cg loop, we do not worry too much about efficiency here. Hence, cache pollution.

```

34c  <Static function prototypes 16b>+≡
      static void copy_o(vOddFermion *dst, const vOddFermion *src);

34d  <Static functions 15a>+≡
      static void
      copy_o(vOddFermion *dst, const vOddFermion *src)
      {
          int i = odd_even.size * S_4 * sizeof (vOddFermion) / sizeof (vReal);
          vReal *d = (vReal *)dst;
          const vReal *s = (const vReal *)src;

          for ( ;i--;)
              *d++ = *s++;
      }

```

5.8.3 compute_sum2_o(d,alpha,s), or $d \leftarrow d + \alpha s$

This is a function we can not speedup much: too many bytes are needed per operation. In principle, one can play with uncached loads and stores, but let us leave that for later.

```

34e  <Static function prototypes 16b>+≡
      static void compute_sum2_o(vOddFermion *dst, double alpha, const vOddFermion *src);

34f  <Static functions 15a>+≡
      static void
      compute_sum2_o(vOddFermion *dst, double alpha, const vOddFermion *src)
      {
          int i = odd_even.size * S_4 * sizeof (vOddFermion) / sizeof (vReal);
          vReal a = vmk1(alpha);
          vReal *d = (vReal *)dst;
          const vReal *s = (const vReal *)src;

          for ( ;i--;)
              *d++ += a * *s++;
      }

```

5.8.4 compute_sum2x_o(d,s,alpha), or $d \leftarrow \alpha d + s$

Almost the same as the previous one, but scaling is applied to another summand.

```

34g  <Static function prototypes 16b>+≡
      static void compute_sum2x_o(vOddFermion *dst, const vOddFermion *src, double alpha);

```

```

35a  <Static functions 15a>+≡
      static void
      compute_sum2x_o(vOddFermion *dst, const vOddFermion *src, double alpha)
      {
          int i = odd_even.size * S_4 * sizeof (vOddFermion) / sizeof (vReal);
          vReal a = vmk1(alpha);
          vReal *d = (vReal *)dst;
          const vReal *s = (const vReal *)src;

          for (;i--; d++)
              *d += a * *d + *s++;
      }

```

5.8.5 compute_sum_x(d,x,alpha,y) or $q \leftarrow x + \alpha y$

Next are a pair of general sums with the destination distinct from the sources.
Do we really need separate functions for these?

```

35b  <Static function prototypes 16b>+≡
      static void compute_sum_e(vEvenFermion *d,
                                const vEvenFermion *x, double alpha, const vEvenFermion *y);
      static void compute_sum_o(vOddFermion *d,
                                const vOddFermion *x, double alpha, const vOddFermion *y);

```

```

35c  <Static functions 15a>+≡
      static void
      compute_sum_e(vEvenFermion *d,
                    const vEvenFermion *x, double alpha, const vEvenFermion *y)
      {
          const vReal *X = (const vReal *)x;
          const vReal *Y = (const vReal *)y;
          vReal *D = (vReal *)d;
          vReal a = vmk1(alpha);
          int i = even_odd.size * S_4 * sizeof (vEvenFermion) / sizeof (vReal);

          for (;i--;)
              *D++ = *X++ + a * *Y++;
      }

```

```

35d  <Static functions 15a>+≡
      static void
      compute_sum_o(vOddFermion *d,
                    const vOddFermion *x, double alpha, const vOddFermion *y)
      {
          const vReal *X = (const vReal *)x;
          const vReal *Y = (const vReal *)y;
          vReal *D = (vReal *)d;
          vReal a = vmk1(alpha);
          int i = odd_even.size * S_4 * sizeof (vOddFermion) / sizeof (vReal);

          for (;i--;)
              *D++ = *X++ + a * *Y++;
      }

```

5.8.6 Compute $d \leftarrow x + \alpha y$ and $r \leftarrow \langle d, d \rangle$

There are two remaining sums which compute a sum of two fermions and the norm of the result at the same time.

```

35e  <Static function prototypes 16b>+≡
      static void compute_sum_oN(vOddFermion *d, double *norm,
                                const vOddFermion *x, double alpha, const vOddFermion *y);

```

```

36a  <Static functions 15a>+≡
      static void
      compute_sum_oN(vOddFermion *d, double *norm,
                     const vOddFermion *x, double alpha, const vOddFermion *y)
      {
        const vReal *X = (const vReal *)x;
        const vReal *Y = (const vReal *)y;
        vReal *D = (vReal *)d;
        vReal a = vmk1(alpha);
        vReal s = vmk1(0.0);
        vReal v;
        int i = odd_even.size * S_4 * sizeof (vOddFermion) / sizeof (vReal);

        for (;i--;) {
          v = *X++ + a * *Y++;
          s += v * v;
          *D++ = v;
        }
        *norm = vsum(s);
        <Start <r,r> computation 60d>
      }

36b  <Static function prototypes 16b>+≡
      static void compute_sum2_oN(vOddFermion *d, double *norm,
                                double alpha, const vOddFermion *y);

36c  <Static functions 15a>+≡
      static void
      compute_sum2_oN(vOddFermion *d, double *norm,
                     double alpha, const vOddFermion *y)
      {
        const vReal *Y = (const vReal *)y;
        vReal *D = (vReal *)d;
        vReal a = vmk1(alpha);
        vReal s = vmk1(0.0);
        vReal v;
        int i = odd_even.size * S_4 * sizeof (vOddFermion) / sizeof (vReal);

        for (;i--;) {
          v = *D + a * *Y++;
          s += v * v;
          *D++ = v;
        }
        *norm = vsum(s);
        <Start <r,r> computation 60d>
      }

```

5.8.7 Compute $\eta \leftarrow M^\dagger M \psi$ and friends

Last three easy pieces.

```

36d  <Static function prototypes 16b>+≡
      static void compute_MxM(vOddFermion *eta, double *norm,
                             const vOddFermion *psi);
      static void compute_M(vOddFermion *eta, double *norm,
                             const vOddFermion *psi);
      static void compute_Mx(vOddFermion *eta,
                             const vOddFermion *psi);

```

```

37a  <Static functions 15a>+≡
      static void
      compute_MxM(vOddFermion *eta, double *norm,
                  const vOddFermion *psi)
      {
          compute_M(auxB_o, norm, psi);
          compute_Mx(eta, auxB_o);
      }

```

Computation of M starts the global sum which will be completed separately.

```

37b  <Static functions 15a>+≡
      static void compute_M(vOddFermion *eta, double *norm,
                          const vOddFermion *psi)
      {
          compute_Qee1Qeo(auxA_e, psi);
          compute_1Qoo1Qoe(eta, norm, psi, auxA_e);
      }

```

For M^\dagger the order of factors differs from optimal. For now we have to live with the inefficiency here.

```

37c  <Static functions 15a>+≡
      static void compute_Mx(vOddFermion *eta,
                          const vOddFermion *psi)
      {
          compute_Soo1(auxA_o, psi);
          compute_See1Seo(auxA_e, auxA_o);
          compute_1Soe(eta, psi, auxA_e);
      }

```

5.8.8 Standalone diagonal pieces

Some code savings are still possible, since `compute_Qee1()` may differ from `compute_Qoo1()` by the number of sites only.

```

37d  <Static function prototypes 16b>+≡
      static void compute_Qxx1(vFermion *eta, const vFermion *psi, int xyz);
      static void inline compute_Qee1(vEvenFermion *eta, const vEvenFermion *psi)
      {
          compute_Qxx1(&eta->f, &psi->f, even_odd.size);
      }
      static void inline compute_Qoo1(vOddFermion *eta, const vOddFermion *psi)
      {
          compute_Qxx1(&eta->f, &psi->f, odd_even.size);
      }
      static void compute_Soo1(vOddFermion *eta, const vOddFermion *psi);

```

$$\chi = Q_{xx}^{-1}\psi$$

```

37e  <Static functions 15a>+≡
      static void
      compute_Qxx1(vFermion *chi, const vFermion *psi, int size)
      {
          const vFermion *qs, *qx5;
          <Q common locals 58b>
          <Qxx locals 59a>

          for (i = 0; i < size; i++) {
              xyz5 = i * S_4;
              <Compute rx5 58f>
              <Compute qx5 58g>
              <Compute Qxx-1 part on the s-chain 38b>
          }
      }

```

$$\chi = S_{oo}^{-1}\psi$$

38a $\langle \text{Static functions 15a} \rangle + \equiv$
 static void
 compute_Soo1(vOddFermion *Chi, const vOddFermion *Psi)
 {
 vFermion *chi = &Chi->f;
 const vFermion *psi = &Psi->f;
 int size = odd_even.size;
 const vFermion *qs, *qx5;
 $\langle Q \text{ common locals 58b} \rangle$
 $\langle Qxx \text{ locals 59a} \rangle$

 for (i = 0; i < size; i++) {
 xyzt5 = i * S_4;
 $\langle \text{Compute rx5 58f} \rangle$
 $\langle \text{Compute qx5 58g} \rangle$
 $\langle \text{Compute } S_{xx}^{-1} \text{ part on the } s\text{-chain 38c} \rangle$
 }
 }

5.8.9 Q_{xx}^{-1} and S_{xx}^{-1} on a single s -chain

Therefore,

38b $\langle \text{Compute } Q_{xx}^{-1} \text{ part on the } s\text{-chain 38b} \rangle \equiv$
 $\langle \text{Compute } A^{-1}\psi \text{ on the upper two components 38d} \rangle$
 $\langle \text{Compute } B^{-1}\psi \text{ on the lower two components 38g} \rangle$

38c $\langle \text{Compute } S_{xx}^{-1} \text{ part on the } s\text{-chain 38c} \rangle \equiv$
 $\langle \text{Compute } A^{-1}\psi \text{ on the lower two components 38e} \rangle$
 $\langle \text{Compute } B^{-1}\psi \text{ on the upper two components 38f} \rangle$

And

38d $\langle \text{Compute } A^{-1}\psi \text{ on the upper two components 38d} \rangle \equiv$
 $\langle \text{Compute } L_A^{-1} \text{ on the upper components 39} \rangle$
 $\langle \text{Compute } R_A^{-1} \text{ on the upper components 42a} \rangle$

38e $\langle \text{Compute } A^{-1}\psi \text{ on the lower two components 38e} \rangle \equiv$
 $\langle \text{Compute } L_A^{-1} \text{ on the lower components 40b} \rangle$
 $\langle \text{Compute } R_A^{-1} \text{ on the lower components 42b} \rangle$

38f $\langle \text{Compute } B^{-1}\psi \text{ on the upper two components 38f} \rangle \equiv$
 $\langle \text{Compute } L_B^{-1} \text{ on the upper components 41a} \rangle$
 $\langle \text{Compute } R_B^{-1} \text{ on the upper components 42c} \rangle$

38g $\langle \text{Compute } B^{-1}\psi \text{ on the lower two components 38g} \rangle \equiv$
 $\langle \text{Compute } L_B^{-1} \text{ on the lower components 41b} \rangle$
 $\langle \text{Compute } R_B^{-1} \text{ on the lower components 42d} \rangle$

For both Q_{xx}^{-1} and S_{xx}^{-1} we need to compute R_A and R_B . This can be done iteratively:

$$\begin{aligned} y_k^{(A)} &= \begin{cases} \frac{1}{a}x_k, & \text{if } k = n-1 \\ \frac{1}{a}x_k - \frac{b}{a}y_{k+1}^{(A)}, & \text{otherwise} \end{cases} \\ y_k^{(B)} &= \begin{cases} \frac{1}{a}x_0, & \text{if } k = 0 \\ \frac{1}{a}x_k - \frac{b}{a}y_{k-1}^{(B)}, & \text{otherwise} \end{cases} \end{aligned}$$

However, the memory layout and our quest for high SSE utilization demand that we use SSE instructions for these computations. Therefore, we sacrifice some efficiency here by unrolling the loop four times and extending $x_k = 0$ if $k < 0$ or $k \geq n$ and set $y_k^{(A)} = 0$ for $k \geq n$ and $y_k^{(B)} = 0$ for $k < 0$. Then we get the following

$$\begin{aligned} y_k^{(A)} &= \frac{1}{a}x_k - \frac{b}{a^2}x_{k+1} + \frac{b^2}{a^3}x_{k+2} - \frac{b^3}{a^4}x_{k+3} + \frac{b^4}{a^5}y_{k+4}^{(A)}, \\ y_k^{(B)} &= \frac{1}{a}x_k - \frac{b}{a^2}x_{k-1} + \frac{b^2}{a^3}x_{k-2} - \frac{b^3}{a^4}x_{k-3} + \frac{b^4}{a^5}y_{k-4}^{(B)}. \end{aligned}$$

Computing $z^{(A)} = L_A^{-1}x$ and $z^{(B)} = L_B^{-1}x$ is easy:

$$\begin{aligned} z_k^{(A)} &= \begin{cases} \sum_{j=0}^{n-2} \frac{(-b)^j c / a^{j+1}}{1 + (-b)^{n-1} c / a^n} x_j + \frac{1}{1 + (-b)^{n-1} c / a^n} x_{n-1}, & \text{if } k = n-1 \\ x_k, & \text{otherwise} \end{cases} \\ z_k^{(B)} &= \begin{cases} \frac{1}{1 + (-b)^{n-1} c / a^n} x_0 + \sum_{j=1}^{n-1} \frac{(-b)^{n-1-j} c / a^{n-j}}{1 + (-b)^{n-1} c / a^n} x_j, & \text{if } k = 0 \\ x_k, & \text{otherwise} \end{cases} \end{aligned}$$

5.8.10 Compute L_A^{-1} and L_B^{-1}

There are two cases:

1. L_X^{-1} is computed as part of standalone diagonal piece. In this case, the computation is done from q to r and L_X^{-1} copies elements as needed.
2. Q_{xx}^{-1} is part of combined operator. In this case q is aliased to r , and no copy is performed.

```

39  <Compute  $L_A^{-1}$  on the upper components 39>≡
    vhfzero(&zV);
    fx = ab_LA;
    <Check xx-aliasing of q 45c>
    for (s = 0; s < S_4_1; s++, fx = fx * va4) {
        rs = &rx5[s];
        QSETUP(s)
        <Compute  $zV \leftarrow zV + fx * qs^{up}$  40a>
    }
    rs = &rx5[S_4_1];
    QSETUP(S_4_1)
    vput_3(&fx, c0);
    <Compute  $zV \leftarrow zV + fx * qs^{up}$  40a>
    for (c = 0; c < 3; c++) {
        <Compute wall value in zX[c] 41c>

        zn.re = qs->f[0][c].re;      zn.im = qs->f[0][c].im;
        vput_3(&zn.re, zX[0][c].re); vput_3(&zn.im, zX[0][c].im);
        rs->f[0][c].re = zn.re;      rs->f[0][c].im = zn.im;

        zn.re = qs->f[1][c].re;      zn.im = qs->f[1][c].im;
        vput_3(&zn.re, zX[1][c].re); vput_3(&zn.im, zX[1][c].im);

```



```

    rs->f[1][c].re = zn.re;      rs->f[1][c].im = zn.im;
}
⟨End xx-aliasing of q 45d⟩

```

To avoid strange things gcc does when SSE data is declared local to a block, we place all such variables on the function level:

This piece is used twice: once in the loop over L_s , and the second time after correcting s_3 :

```

40a  ⟨Compute  $zV \leftarrow zV + fx * qs^{up}$  40a⟩≡
    for (c = 0; c < 3; c++) {
        zV.f[0][c].re += fx * qs->f[0][c].re; Q2R(0,re)
        zV.f[0][c].im += fx * qs->f[0][c].im; Q2R(0,im)
        zV.f[1][c].re += fx * qs->f[1][c].re; Q2R(1,re)
        zV.f[1][c].im += fx * qs->f[1][c].im; Q2R(1,im)
    }

```

The only difference between L_A^{-1} on lower components is the source of the data and the destination of the result. We have to repeat most of the above pieces though.

```

40b  ⟨Compute  $L_A^{-1}$  on the lower components 40b⟩≡
    vhfzero(&zV);
    fx = ab_LA;
    ⟨Check xx-aliasing of q 45c⟩
    for (s = 0; s < S_4_1; s++, fx = fx * va4) {
        rs = &rx5[s];
        QSETUP(s)
        ⟨Compute  $zV \leftarrow zV + fx * qs^{down}$  40c⟩
    }
    rs = &rx5[S_4_1];
    QSETUP(S_4_1)
    vput_3(&fx, c0);
    ⟨Compute  $zV \leftarrow zV + fx * qs^{down}$  40c⟩
    for (c = 0; c < 3; c++) {
        ⟨Compute wall value in zX[c] 41c⟩

        zn.re = qs->f[2][c].re;      zn.im = qs->f[2][c].im;
        vput_3(&zn.re, zX[0][c].re); vput_3(&zn.im, zX[0][c].im);
        rs->f[2][c].re = zn.re;      rs->f[2][c].im = zn.im;

        zn.re = qs->f[3][c].re;      zn.im = qs->f[3][c].im;
        vput_3(&zn.re, zX[1][c].re); vput_3(&zn.im, zX[1][c].im);
        rs->f[3][c].re = zn.re;      rs->f[3][c].im = zn.im;
    }
    ⟨End xx-aliasing of q 45d⟩

```

```

40c  ⟨Compute  $zV \leftarrow zV + fx * qs^{down}$  40c⟩≡
    for (c = 0; c < 3; c++) {
        zV.f[0][c].re += fx * qs->f[2][c].re; Q2R(2,re)
        zV.f[0][c].im += fx * qs->f[2][c].im; Q2R(2,im)
        zV.f[1][c].re += fx * qs->f[3][c].re; Q2R(3,re)
        zV.f[1][c].im += fx * qs->f[3][c].im; Q2R(3,im)
    }

```

For L_B^{-1} the difference is in the direction of the sweep along the s -chain:

41a $\langle \text{Compute } L_B^{-1} \text{ on the upper components 41a} \rangle \equiv$

```

    vhfzero(&zV);
    fx = ab_LB;
     $\langle \text{Check } xx\text{-aliasing of } q \text{ 45c} \rangle$ 
    for (s = S_4; --s; fx = fx * va4) {
        rs = &rx5[s];
        QSETUP(s)
         $\langle \text{Compute } zV \leftarrow zV + fx * qs^{up} \text{ 40a} \rangle$ 
    }
    rs = &rx5[0];
    QSETUP(0)
    vput_0(&fx, c0);
     $\langle \text{Compute } zV \leftarrow zV + fx * qs^{up} \text{ 40a} \rangle$ 
    for (c = 0; c < 3; c++) {
         $\langle \text{Compute wall value in } zX[c] \text{ 41c} \rangle$ 

        zn.re = qs->f[0][c].re;    zn.im = qs->f[0][c].im;
        vput_0(&zn.re, zX[0][c].re); vput_0(&zn.im, zX[0][c].im);
        rs->f[0][c].re = zn.re;    rs->f[0][c].im = zn.im;

        zn.re = qs->f[1][c].re;    zn.im = qs->f[1][c].im;
        vput_0(&zn.re, zX[1][c].re); vput_0(&zn.im, zX[1][c].im);
        rs->f[1][c].re = zn.re;    rs->f[1][c].im = zn.im;
    }
     $\langle \text{End } xx\text{-aliasing of } q \text{ 45d} \rangle$ 

```

Again, some repetition is needed for the lower component case:

41b $\langle \text{Compute } L_B^{-1} \text{ on the lower components 41b} \rangle \equiv$

```

    vhfzero(&zV);
    fx = ab_LB;
     $\langle \text{Check } xx\text{-aliasing of } q \text{ 45c} \rangle$ 
    for (s = S_4; --s; fx = fx * va4) {
        rs = &rx5[s];
        QSETUP(s)
         $\langle \text{Compute } zV \leftarrow zV + fx * qs^{down} \text{ 40c} \rangle$ 
    }
    rs = &rx5[0];
    QSETUP(0)
    vput_0(&fx, c0);
     $\langle \text{Compute } zV \leftarrow zV + fx * qs^{down} \text{ 40c} \rangle$ 
    for (c = 0; c < 3; c++) {
         $\langle \text{Compute wall value in } zX[c] \text{ 41c} \rangle$ 

        zn.re = qs->f[2][c].re;    zn.im = qs->f[2][c].im;
        vput_0(&zn.re, zX[0][c].re); vput_0(&zn.im, zX[0][c].im);
        rs->f[2][c].re = zn.re;    rs->f[2][c].im = zn.im;

        zn.re = qs->f[3][c].re;    zn.im = qs->f[3][c].im;
        vput_0(&zn.re, zX[1][c].re); vput_0(&zn.im, zX[1][c].im);
        rs->f[3][c].re = zn.re;    rs->f[3][c].im = zn.im;
    }
     $\langle \text{End } xx\text{-aliasing of } q \text{ 45d} \rangle$ 

```

By now, we have four partial sums which must be combined into z_{n-1} :

41c $\langle \text{Compute wall value in } zX[c] \text{ 41c} \rangle \equiv$

```

    zX[0][c].re = vsum(zV.f[0][c].re);
    zX[0][c].im = vsum(zV.f[0][c].im);
    zX[1][c].re = vsum(zV.f[1][c].re);
    zX[1][c].im = vsum(zV.f[1][c].im);

```

5.8.11 Compute R_A^{-1} and R_B^{-1}

Since R_X^{-1} is always computed after L_X^{-1} , it takes its source from **rs** and places the result back into **rs**.

For R_X^{-1} , again combinations of A and B and upper and lower parts require some cut, paste and edit.

42a $\langle \text{Compute } R_A^{-1} \text{ on the upper components 42a} \rangle \equiv$
 $\langle \text{Init out of bound } x \text{ and } y \text{ 42e} \rangle$
for (s = S_4; s--;) {
rs = &rx5[s];
for (c = 0; c < 3; c++) {
 $\langle \text{Compute } y_{k,[0]}^{(A)} \text{ 43a} \rangle$
 $\langle \text{Compute } y_{k,[1]}^{(A)} \text{ 43b} \rangle$
}
}

42b $\langle \text{Compute } R_A^{-1} \text{ on the lower components 42b} \rangle \equiv$
 $\langle \text{Init out of bound } x \text{ and } y \text{ 42e} \rangle$
for (s = S_4; s--;) {
rs = &rx5[s];
for (c = 0; c < 3; c++) {
 $\langle \text{Compute } y_{k,[2]}^{(A)} \text{ 43c} \rangle$
 $\langle \text{Compute } y_{k,[3]}^{(A)} \text{ 44a} \rangle$
}
}

42c $\langle \text{Compute } R_B^{-1} \text{ on the upper components 42c} \rangle \equiv$
 $\langle \text{Init out of bound } x \text{ and } y \text{ 42e} \rangle$
for (s = 0; s < S_4; s++) {
rs = &rx5[s];
for (c = 0; c < 3; c++) {
 $\langle \text{Compute } y_{k,[0]}^{(B)} \text{ 44b} \rangle$
 $\langle \text{Compute } y_{k,[1]}^{(B)} \text{ 44c} \rangle$
}
}

42d $\langle \text{Compute } R_B^{-1} \text{ on the lower components 42d} \rangle \equiv$
 $\langle \text{Init out of bound } x \text{ and } y \text{ 42e} \rangle$
for (s = 0; s < S_4; s++) {
rs = &rx5[s];
for (c = 0; c < 3; c++) {
 $\langle \text{Compute } y_{k,[2]}^{(B)} \text{ 45a} \rangle$
 $\langle \text{Compute } y_{k,[3]}^{(B)} \text{ 45b} \rangle$
}
}

The only piece we can share here is

42e $\langle \text{Init out of bound } x \text{ and } y \text{ 42e} \rangle \equiv$
vhfzero(&xOut);
vhfzero(&yOut);

Now, the magic of copy paste:

```

43a  <Compute  $y_{k,[0]}^{(A)}$  43a>≡
      zn.re = rs->f[0][c].re;
      zn.im = rs->f[0][c].im;
      z1.re = shift_down1(zn.re, xOut.f[0][c].re);
      z1.im = shift_down1(zn.im, xOut.f[0][c].im);
      z2.re = shift_down2(zn.re, xOut.f[0][c].re);
      z2.im = shift_down2(zn.im, xOut.f[0][c].im);
      z3.re = shift_down3(zn.re, xOut.f[0][c].re);
      z3.im = shift_down3(zn.im, xOut.f[0][c].im);
      rs->f[0][c].re = va0*zn.re + va1*z1.re + va2*z2.re + va3*z3.re + va4*yOut.f[0][c].re;
      rs->f[0][c].im = va0*zn.im + va1*z1.im + va2*z2.im + va3*z3.im + va4*yOut.f[0][c].im;
      yOut.f[0][c].re = rs->f[0][c].re;
      yOut.f[0][c].im = rs->f[0][c].im;
      xOut.f[0][c].re = zn.re;
      xOut.f[0][c].im = zn.im;

43b  <Compute  $y_{k,[1]}^{(A)}$  43b>≡
      zn.re = rs->f[1][c].re;
      zn.im = rs->f[1][c].im;
      z1.re = shift_down1(zn.re, xOut.f[1][c].re);
      z1.im = shift_down1(zn.im, xOut.f[1][c].im);
      z2.re = shift_down2(zn.re, xOut.f[1][c].re);
      z2.im = shift_down2(zn.im, xOut.f[1][c].im);
      z3.re = shift_down3(zn.re, xOut.f[1][c].re);
      z3.im = shift_down3(zn.im, xOut.f[1][c].im);
      rs->f[1][c].re = va0*zn.re + va1*z1.re + va2*z2.re + va3*z3.re + va4*yOut.f[1][c].re;
      rs->f[1][c].im = va0*zn.im + va1*z1.im + va2*z2.im + va3*z3.im + va4*yOut.f[1][c].im;
      yOut.f[1][c].re = rs->f[1][c].re;
      yOut.f[1][c].im = rs->f[1][c].im;
      xOut.f[1][c].re = zn.re;
      xOut.f[1][c].im = zn.im;

43c  <Compute  $y_{k,[2]}^{(A)}$  43c>≡
      zn.re = rs->f[2][c].re;
      zn.im = rs->f[2][c].im;
      z1.re = shift_down1(zn.re, xOut.f[0][c].re);
      z1.im = shift_down1(zn.im, xOut.f[0][c].im);
      z2.re = shift_down2(zn.re, xOut.f[0][c].re);
      z2.im = shift_down2(zn.im, xOut.f[0][c].im);
      z3.re = shift_down3(zn.re, xOut.f[0][c].re);
      z3.im = shift_down3(zn.im, xOut.f[0][c].im);
      rs->f[2][c].re = va0*zn.re + va1*z1.re + va2*z2.re + va3*z3.re + va4*yOut.f[0][c].re;
      rs->f[2][c].im = va0*zn.im + va1*z1.im + va2*z2.im + va3*z3.im + va4*yOut.f[0][c].im;
      yOut.f[0][c].re = rs->f[2][c].re;
      yOut.f[0][c].im = rs->f[2][c].im;
      xOut.f[0][c].re = zn.re;
      xOut.f[0][c].im = zn.im;

```

44a $\langle \text{Compute } y_{k,[3]}^{(A)} \text{ 44a} \rangle \equiv$

```

zn.re = rs->f[3][c].re;
zn.im = rs->f[3][c].im;
z1.re = shift_down1(zn.re, xOut.f[1][c].re);
z1.im = shift_down1(zn.im, xOut.f[1][c].im);
z2.re = shift_down2(zn.re, xOut.f[1][c].re);
z2.im = shift_down2(zn.im, xOut.f[1][c].im);
z3.re = shift_down3(zn.re, xOut.f[1][c].re);
z3.im = shift_down3(zn.im, xOut.f[1][c].im);
rs->f[3][c].re = va0*zn.re + va1*z1.re + va2*z2.re + va3*z3.re + va4*yOut.f[1][c].re;
rs->f[3][c].im = va0*zn.im + va1*z1.im + va2*z2.im + va3*z3.im + va4*yOut.f[1][c].im;
yOut.f[1][c].re = rs->f[3][c].re;
yOut.f[1][c].im = rs->f[3][c].im;
xOut.f[1][c].re = zn.re;
xOut.f[1][c].im = zn.im;

```

44b $\langle \text{Compute } y_{k,[0]}^{(B)} \text{ 44b} \rangle \equiv$

```

zn.re = rs->f[0][c].re;
zn.im = rs->f[0][c].im;
z1.re = shift_up1(xOut.f[0][c].re, zn.re);
z1.im = shift_up1(xOut.f[0][c].im, zn.im);
z2.re = shift_up2(xOut.f[0][c].re, zn.re);
z2.im = shift_up2(xOut.f[0][c].im, zn.im);
z3.re = shift_up3(xOut.f[0][c].re, zn.re);
z3.im = shift_up3(xOut.f[0][c].im, zn.im);
rs->f[0][c].re = va0*zn.re + va1*z1.re + va2*z2.re + va3*z3.re + va4*yOut.f[0][c].re;
rs->f[0][c].im = va0*zn.im + va1*z1.im + va2*z2.im + va3*z3.im + va4*yOut.f[0][c].im;
yOut.f[0][c].re = rs->f[0][c].re;
yOut.f[0][c].im = rs->f[0][c].im;
xOut.f[0][c].re = zn.re;
xOut.f[0][c].im = zn.im;

```

44c $\langle \text{Compute } y_{k,[1]}^{(B)} \text{ 44c} \rangle \equiv$

```

zn.re = rs->f[1][c].re;
zn.im = rs->f[1][c].im;
z1.re = shift_up1(xOut.f[1][c].re, zn.re);
z1.im = shift_up1(xOut.f[1][c].im, zn.im);
z2.re = shift_up2(xOut.f[1][c].re, zn.re);
z2.im = shift_up2(xOut.f[1][c].im, zn.im);
z3.re = shift_up3(xOut.f[1][c].re, zn.re);
z3.im = shift_up3(xOut.f[1][c].im, zn.im);
rs->f[1][c].re = va0*zn.re + va1*z1.re + va2*z2.re + va3*z3.re + va4*yOut.f[1][c].re;
rs->f[1][c].im = va0*zn.im + va1*z1.im + va2*z2.im + va3*z3.im + va4*yOut.f[1][c].im;
yOut.f[1][c].re = rs->f[1][c].re;
yOut.f[1][c].im = rs->f[1][c].im;
xOut.f[1][c].re = zn.re;
xOut.f[1][c].im = zn.im;

```

```

45a  <Compute  $y_{k,[2]}^{(B)}$  45a>≡
      zn.re = rs->f[2][c].re;
      zn.im = rs->f[2][c].im;
      z1.re = shift_up1(xOut.f[0][c].re, zn.re);
      z1.im = shift_up1(xOut.f[0][c].im, zn.im);
      z2.re = shift_up2(xOut.f[0][c].re, zn.re);
      z2.im = shift_up2(xOut.f[0][c].im, zn.im);
      z3.re = shift_up3(xOut.f[0][c].re, zn.re);
      z3.im = shift_up3(xOut.f[0][c].im, zn.im);
      rs->f[2][c].re = va0*zn.re + va1*z1.re + va2*z2.re + va3*z3.re + va4*yOut.f[0][c].re;
      rs->f[2][c].im = va0*zn.im + va1*z1.im + va2*z2.im + va3*z3.im + va4*yOut.f[0][c].im;
      yOut.f[0][c].re = rs->f[2][c].re;
      yOut.f[0][c].im = rs->f[2][c].im;
      xOut.f[0][c].re = zn.re;
      xOut.f[0][c].im = zn.im;

45b  <Compute  $y_{k,[3]}^{(B)}$  45b>≡
      zn.re = rs->f[3][c].re;
      zn.im = rs->f[3][c].im;
      z1.re = shift_up1(xOut.f[1][c].re, zn.re);
      z1.im = shift_up1(xOut.f[1][c].im, zn.im);
      z2.re = shift_up2(xOut.f[1][c].re, zn.re);
      z2.im = shift_up2(xOut.f[1][c].im, zn.im);
      z3.re = shift_up3(xOut.f[1][c].re, zn.re);
      z3.im = shift_up3(xOut.f[1][c].im, zn.im);
      rs->f[3][c].re = va0*zn.re + va1*z1.re + va2*z2.re + va3*z3.re + va4*yOut.f[1][c].re;
      rs->f[3][c].im = va0*zn.im + va1*z1.im + va2*z2.im + va3*z3.im + va4*yOut.f[1][c].im;
      yOut.f[1][c].re = rs->f[3][c].re;
      yOut.f[1][c].im = rs->f[3][c].im;
      xOut.f[1][c].re = zn.re;
      xOut.f[1][c].im = zn.im;

We use #if defined(qs) to tell if q is separate from r.

45c  <Check xx-aliasing of q 45c>≡
      #if defined(qs)
      #define QSETUP(s)
      #define Q2R(d,pt)
      #else
      #define QSETUP(s) qs = &qx5[s];
      #define Q2R(d,pt) rs->f[d][c].pt = qs->f[d][c].pt;
      #endif

45d  <End xx-aliasing of q 45d>≡
      #undef QSETUP
      #undef Q2R

```

5.8.12 Standalone off-diagonal pieces

First, simple off-diagonal parts.

46a $\langle \text{Static function prototypes 16b} \rangle + \equiv$

```

static void compute_Qxy(vFermion *d, const vFermion *s, struct neighbor *nb);
static void inline compute_Qoe(vOddFermion *d, const vEvenFermion *s)
{
    compute_Qxy(&d->f, &s->f, &odd_even);
}

static void inline compute_Qeo(vEvenFermion *d, const vOddFermion *s)
{
    compute_Qxy(&d->f, &s->f, &even_odd);
}

static void compute_1Sxy(vFermion *d,
                        const vFermion *q,
                        const vFermion *s,
                        struct neighbor *nb);
static void inline compute_1Soe(vOddFermion *d,
                               const vOddFermion *q,
                               const vEvenFermion *s)
{
    compute_1Sxy(&d->f, &q->f, &s->f, &odd_even);
}

```

5.8.13 Common off-diagonal parts

We start from the top level:

$$\chi = Q_{xy}\psi$$

46b $\langle \text{Static functions 15a} \rangle + \equiv$

```

static void
compute_Qxy(vFermion *chi,
            const vFermion *psi,
            struct neighbor *nb)
{
     $\langle Q \text{ common locals 58b} \rangle$ 
     $\langle Q_{xy} \text{ locals 58c} \rangle$ 

     $\langle \text{Setup } xy\text{-aliasing of } q \text{ 48c} \rangle$ 
     $\langle \text{Start off-diagonal receives 59f} \rangle$ 
     $\langle \text{Finish old off-diagonal sends 60b} \rangle$ 
     $\langle \text{Compute projections for } Q \text{ send 49a} \rangle$ 
     $\langle \text{Compute inside part for } Q_{xy} \text{ 51a} \rangle$ 
     $\langle \text{Finish off-diagonal receives 59g} \rangle$ 
     $\langle \text{Compute boundary part for } Q_{xy} \text{ 51b} \rangle$ 
     $\langle \text{Finish } xy\text{-aliasing of } q \text{ 48d} \rangle$ 
}

```

For other functions, little need to be changes at this granularity. E.g., the final part of M^\dagger is

$$\chi = \eta - S_{xy}\psi$$

47a *⟨Static functions 15a⟩+≡*
`static void`
`compute_1Sxy(vFermion *chi,`
`const vFermion *eta,`
`const vFermion *psi,`
`struct neighbor *nb)`
`{`
⟨Q common locals 58b⟩
⟨Qxy locals 58c⟩

⟨Setup xy-aliasing of q 48c⟩
⟨Start off-diagonal receives 59f⟩
⟨Finish old off-diagonal sends 60b⟩
⟨Compute projections for S send 49b⟩
⟨Compute inside part for $1 - S_{xy}$ 53b⟩
⟨Finish off-diagonal receives 59g⟩
⟨Compute boundary part for $1 - S_{xy}$ 53c⟩
⟨Finish xy-aliasing of q 48d⟩
`}`

Likewise,

$$\chi = Q_{xx}^{-1}Q_{xy}\psi$$

47b *⟨Static functions 15a⟩+≡*
`static void`
`compute_Qxx1Qxy(vFermion *chi,`
`const vFermion *psi,`
`struct neighbor *nb)`
`{`
⟨Q common locals 58b⟩
⟨Qxy locals 58c⟩
⟨Qxx locals 59a⟩

⟨Setup xy-aliasing of q 48c⟩
⟨Start off-diagonal receives 59f⟩
⟨Finish old off-diagonal sends 60b⟩
⟨Compute projections for Q send 49a⟩
⟨Compute inside part for $Q_{xx}^{-1}Q_{xy}$ 55b⟩
⟨Finish off-diagonal receives 59g⟩
⟨Compute boundary part for $Q_{xx}^{-1}Q_{xy}$ 55c⟩
⟨Finish xy-aliasing of q 48d⟩
`}`

and

$$\chi = S_{xx}^{-1} S_{xy} \psi$$

```
48a  <Static functions 15a>+=
      static void
      compute_Sxx1Sxy(vFermion *chi,
                      const vFermion *psi,
                      struct neighbor *nb)
      {
        <Q common locals 58b>
        <Qxy locals 58c>
        <Qxx locals 59a>

        <Setup xy-aliasing of q 48c>
        <Start off-diagonal receives 59f>
        <Finish old off-diagonal sends 60b>
        <Compute projections for S send 49b>
        <Compute inside part for  $S_{xx}^{-1} S_{xy}$  55d>
        <Finish off-diagonal receives 59g>
        <Compute boundary part for  $S_{xx}^{-1} S_{xy}$  55e>
        <Finish xy-aliasing of q 48d>
      }
```

Finally,

$$\chi = \eta - Q_{xx}^{-1} Q_{xy} \psi$$

```
48b  <Static functions 15a>+=
      static void
      compute_1Qxx1Qxy(vFermion *chi,
                      double *norm,
                      const vFermion *eta,
                      const vFermion *psi,
                      struct neighbor *nb)
      {
        <Q common locals 58b>
        <Qxy locals 58c>
        <Qxx locals 59a>
        vReal vv;
        vReal nv = vmk1(0.0);

        <Setup xy-aliasing of q 48c>
        <Start off-diagonal receives 59f>
        <Finish old off-diagonal sends 60b>
        <Compute projections for Q send 49a>
        <Compute inside part for  $1 - Q_{xx}^{-1} Q_{xy}$  56d>
        <Finish off-diagonal receives 59g>
        <Compute boundary part for  $1 - Q_{xx}^{-1} Q_{xy}$  56e>
        *norm = vsum(nv);
        <Start <r,r> computation 60d>
        <Finish xy-aliasing of q 48d>
      }
```

Remember, that Z_{xy} always puts result into q. For standalone diagonal pieces a couple of `define`'s help to manage `__restrict__` pointers properly.

```
48c  <Setup xy-aliasing of q 48c>=
      #define qx5 rx5
      #define qs rs

48d  <Finish xy-aliasing of q 48d>=
      #undef qs
      #undef qx5
```

5.8.14 Projections to be sent

Next we compute $(1 \pm \gamma_\mu)$ projections to be sent to our neighbors. There are two cases here, one of Q_{xy} and another for S_{xy} . In principle, we might have handled both of them with some jungling of the `struct neighbor` tables, but let us go a simple if extensive way for now.

```

49a  <Compute projections for Q send 49a>≡
      {
          int k, i, s, c, *src;
          const vFermion *f;
          vHalfFermion *g;

          k = 0; <Construct (1 +  $\gamma_0$ ) send k-buffer 49c>
          k = 1; <Construct (1 -  $\gamma_0$ ) send k-buffer 49d>
          k = 2; <Construct (1 +  $\gamma_1$ ) send k-buffer 50a>
          k = 3; <Construct (1 -  $\gamma_1$ ) send k-buffer 50b>
          k = 4; <Construct (1 +  $\gamma_2$ ) send k-buffer 50c>
          k = 5; <Construct (1 -  $\gamma_2$ ) send k-buffer 50d>
          k = 6; <Construct (1 +  $\gamma_3$ ) send k-buffer 50e>
          k = 7; <Construct (1 -  $\gamma_3$ ) send k-buffer 50f>
      }

49b  <Compute projections for S send 49b>≡
      {
          int k, i, s, c, *src;
          const vFermion *f;
          vHalfFermion *g;

          k = 0; <Construct (1 -  $\gamma_0$ ) send k-buffer 49d>
          k = 1; <Construct (1 +  $\gamma_0$ ) send k-buffer 49c>
          k = 2; <Construct (1 -  $\gamma_1$ ) send k-buffer 50b>
          k = 3; <Construct (1 +  $\gamma_1$ ) send k-buffer 50a>
          k = 4; <Construct (1 -  $\gamma_2$ ) send k-buffer 50d>
          k = 5; <Construct (1 +  $\gamma_2$ ) send k-buffer 50c>
          k = 6; <Construct (1 -  $\gamma_3$ ) send k-buffer 50f>
          k = 7; <Construct (1 +  $\gamma_3$ ) send k-buffer 50e>
      }

49c  <Construct (1 +  $\gamma_0$ ) send k-buffer 49c>≡
      for (i = nb->snd_size[k], g = nb->snd_buf[k], src = nb->snd[k]; i--; src++) {
          for (s = S_4, f = &psi[*src]; s--; g++, f++) {
              for (c = 0; c < 3; c++) {
                  <Build (1 +  $\gamma_0$ ) projection of *f in *g 4a>
              }
          }
      }
      <Start k-send 60a>

49d  <Construct (1 -  $\gamma_0$ ) send k-buffer 49d>≡
      for (i = nb->snd_size[k], g = nb->snd_buf[k], src = nb->snd[k]; i--; src++) {
          for (s = S_4, f = &psi[*src]; s--; g++, f++) {
              for (c = 0; c < 3; c++) {
                  <Build (1 -  $\gamma_0$ ) projection of *f in *g 4c>
              }
          }
      }
      <Start k-send 60a>

```

```

50a  ⟨Construct (1 +  $\gamma_1$ ) send k-buffer 50a⟩≡
      for (i = nb->snd_size[k], g = nb->snd_buf[k], src = nb->snd[k]; i--; src++) {
        for (s = S_4, f = &psi[*src]; s--; g++, f++) {
          for (c = 0; c < 3; c++) {
            ⟨Build (1 +  $\gamma_1$ ) projection of *f in *g 4e⟩
          }
        }
      }
      ⟨Start k-send 60a⟩

50b  ⟨Construct (1 -  $\gamma_1$ ) send k-buffer 50b⟩≡
      for (i = nb->snd_size[k], g = nb->snd_buf[k], src = nb->snd[k]; i--; src++) {
        for (s = S_4, f = &psi[*src]; s--; g++, f++) {
          for (c = 0; c < 3; c++) {
            ⟨Build (1 -  $\gamma_1$ ) projection of *f in *g 4g⟩
          }
        }
      }
      ⟨Start k-send 60a⟩

50c  ⟨Construct (1 +  $\gamma_2$ ) send k-buffer 50c⟩≡
      for (i = nb->snd_size[k], g = nb->snd_buf[k], src = nb->snd[k]; i--; src++) {
        for (s = S_4, f = &psi[*src]; s--; g++, f++) {
          for (c = 0; c < 3; c++) {
            ⟨Build (1 +  $\gamma_2$ ) projection of *f in *g 5a⟩
          }
        }
      }
      ⟨Start k-send 60a⟩

50d  ⟨Construct (1 -  $\gamma_2$ ) send k-buffer 50d⟩≡
      for (i = nb->snd_size[k], g = nb->snd_buf[k], src = nb->snd[k]; i--; src++) {
        for (s = S_4, f = &psi[*src]; s--; g++, f++) {
          for (c = 0; c < 3; c++) {
            ⟨Build (1 -  $\gamma_2$ ) projection of *f in *g 5c⟩
          }
        }
      }
      ⟨Start k-send 60a⟩

50e  ⟨Construct (1 +  $\gamma_3$ ) send k-buffer 50e⟩≡
      for (i = nb->snd_size[k], g = nb->snd_buf[k], src = nb->snd[k]; i--; src++) {
        for (s = S_4, f = &psi[*src]; s--; g++, f++) {
          for (c = 0; c < 3; c++) {
            ⟨Build (1 +  $\gamma_3$ ) projection of *f in *g 5e⟩
          }
        }
      }
      ⟨Start k-send 60a⟩

50f  ⟨Construct (1 -  $\gamma_3$ ) send k-buffer 50f⟩≡
      for (i = nb->snd_size[k], g = nb->snd_buf[k], src = nb->snd[k]; i--; src++) {
        for (s = S_4, f = &psi[*src]; s--; g++, f++) {
          for (c = 0; c < 3; c++) {
            ⟨Build (1 -  $\gamma_3$ ) projection of *f in *g 5g⟩
          }
        }
      }
      ⟨Start k-send 60a⟩

```

5.8.15 Parts of $Q_{xy}\psi$

Let us start with the simplest of the five operators we need.

```

51a  <Compute inside part for  $Q_{xy}$  51a>≡
      for (i = 0; i < nb->inside_size; i++) {
          xyzt = nb->inside[i];
          xyzt5 = xyzt * S_4;
          <Extract 1-d addresses 58e>
          <Build SSE SU(3) objects 57c>
          <Compute  $Q_{xy}$  part on the inside s-chain 51c>
      }

51b  <Compute boundary part for  $Q_{xy}$  51b>≡
      for (i = 0; i < nb->boundary_size; i++) {
          int m = nb->boundary[i].mask;

          xyzt = nb->boundary[i].index;
          xyzt5 = xyzt * S_4;
          <Extract 1-d addresses 58e>
          <Build SSE SU(3) objects 57c>
          <Compute  $Q_{xy}$  part on the boundary s-chain 51d>
      }

51c  <Compute  $Q_{xy}$  part on the inside s-chain 51c>≡
      for (s = 0; s < S_4; s++) {
          <Compute  $Q$  inside  $\gamma$ -projections 51e>
          <Inside multiply by  $V$ s 52c>
          <Compute  $Q$   $\gamma$ -unprojections and sum the results 52b>
      }

51d  <Compute  $Q_{xy}$  part on the boundary s-chain 51d>≡
      for (s = 0; s < S_4; s++) {
          <Compute  $Q$  boundary  $\gamma$ -projections 52a>
          <Boundary multiply by  $V$ s 52d>
          <Compute  $Q$   $\gamma$ -unprojections and sum the results 52b>
      }

51e  <Compute  $Q$  inside  $\gamma$ -projections 51e>≡
      <Construct neighbor pointers 57d>
      for (c = 0; c < 3; c++) {
          k=0; f=&psi[ps[0]]; g=&gg[0]; <Build  $(1 + \gamma_0)$  projection of *f in *g 4a>
          k=1; f=&psi[ps[1]]; g=&gg[1]; <Build  $(1 - \gamma_0)$  projection of *f in *g 4c>
          k=2; f=&psi[ps[2]]; g=&gg[2]; <Build  $(1 + \gamma_1)$  projection of *f in *g 4e>
          k=3; f=&psi[ps[3]]; g=&gg[3]; <Build  $(1 - \gamma_1)$  projection of *f in *g 4g>
          k=4; f=&psi[ps[4]]; g=&gg[4]; <Build  $(1 + \gamma_2)$  projection of *f in *g 5a>
          k=5; f=&psi[ps[5]]; g=&gg[5]; <Build  $(1 - \gamma_2)$  projection of *f in *g 5c>
          k=6; f=&psi[ps[6]]; g=&gg[6]; <Build  $(1 + \gamma_3)$  projection of *f in *g 5e>
          k=7; f=&psi[ps[7]]; g=&gg[7]; <Build  $(1 - \gamma_3)$  projection of *f in *g 5g>
      }

```

```

52a  <Compute Q boundary  $\gamma$ -projections 52a>≡
      <Construct neighbor pointers 57d>
      for (c = 0; c < 3; c++) {
        if ((m & 0x01) == 0) {
          k=0; f=&psi[ps[0]]; g=&gg[0]; <Build  $(1 + \gamma_0)$  projection of *f in *g 4a>
        }
        if ((m & 0x02) == 0) {
          k=1; f=&psi[ps[1]]; g=&gg[1]; <Build  $(1 - \gamma_0)$  projection of *f in *g 4c>
        }
        if ((m & 0x04) == 0) {
          k=2; f=&psi[ps[2]]; g=&gg[2]; <Build  $(1 + \gamma_1)$  projection of *f in *g 4e>
        }
        if ((m & 0x08) == 0) {
          k=3; f=&psi[ps[3]]; g=&gg[3]; <Build  $(1 - \gamma_1)$  projection of *f in *g 4g>
        }
        if ((m & 0x10) == 0) {
          k=4; f=&psi[ps[4]]; g=&gg[4]; <Build  $(1 + \gamma_2)$  projection of *f in *g 5a>
        }
        if ((m & 0x20) == 0) {
          k=5; f=&psi[ps[5]]; g=&gg[5]; <Build  $(1 - \gamma_2)$  projection of *f in *g 5c>
        }
        if ((m & 0x40) == 0) {
          k=6; f=&psi[ps[6]]; g=&gg[6]; <Build  $(1 + \gamma_3)$  projection of *f in *g 5e>
        }
        if ((m & 0x80) == 0) {
          k=7; f=&psi[ps[7]]; g=&gg[7]; <Build  $(1 - \gamma_3)$  projection of *f in *g 5g>
        }
      }

```

```

52b  <Compute Q  $\gamma$ -unprojections and sum the results 52b>≡
      rs = &rx5[s];
      for (c = 0; c < 3; c++) {
        k = 6; <Unproject  $(1 + \gamma_3)$  link 5f>
        k = 7; <Unproject and accumulate  $(1 - \gamma_3)$  link 5h>
        k = 2; <Unproject and accumulate  $(1 + \gamma_1)$  link 4f>
        k = 3; <Unproject and accumulate  $(1 - \gamma_1)$  link 4h>
        k = 0; <Unproject and accumulate  $(1 + \gamma_0)$  link 4b>
        k = 1; <Unproject and accumulate  $(1 - \gamma_0)$  link 4d>
        k = 4; <Unproject and accumulate  $(1 + \gamma_2)$  link 5b>
        k = 5; <Unproject and accumulate  $(1 - \gamma_2)$  link 5d>
      }

```

Now we have everything we need to compute $U(1 \pm \gamma_\mu)\psi$ pieces:

```

52c  <Inside multiply by Vs 52c>≡
      for (d = 0; d < 8; d++) {
        vHalfFermion * __restrict__ h = &hh[d];
        vSU3 *u = &V[d];
        g = &gg[d];
        <Multiply *u by *g and store the result in *h 53a>
      }

```

If the neighbor is on another node, it is in the receive buffer by now.

```

52d  <Boundary multiply by Vs 52d>≡
      for (d = 0; d < 8; d++) {
        vHalfFermion * __restrict__ h = &hh[d];
        vSU3 *u = &V[d];
        g = (m & (1 << d)) ? &nb->rcv_buf[d][ps[d]] : &gg[d];
        <Multiply *u by *g and store the result in *h 53a>
      }

```

53a $\langle \text{Multiply } *u \text{ by } *g \text{ and store the result in } *h \text{ 53a} \rangle \equiv$

```

for (c = 0; c < 3; c++) {
    h->f[0][c].re=u->v[c][0].re*g->f[0][0].re-u->v[c][0].im*g->f[0][0].im
    +u->v[c][1].re*g->f[0][1].re-u->v[c][1].im*g->f[0][1].im
    +u->v[c][2].re*g->f[0][2].re-u->v[c][2].im*g->f[0][2].im;
    h->f[0][c].im=u->v[c][0].im*g->f[0][0].re+u->v[c][0].re*g->f[0][0].im
    +u->v[c][1].im*g->f[0][1].re+u->v[c][1].re*g->f[0][1].im
    +u->v[c][2].im*g->f[0][2].re+u->v[c][2].re*g->f[0][2].im;
    h->f[1][c].re=u->v[c][0].re*g->f[1][0].re-u->v[c][0].im*g->f[1][0].im
    +u->v[c][1].re*g->f[1][1].re-u->v[c][1].im*g->f[1][1].im
    +u->v[c][2].re*g->f[1][2].re-u->v[c][2].im*g->f[1][2].im;
    h->f[1][c].im=u->v[c][0].im*g->f[1][0].re+u->v[c][0].re*g->f[1][0].im
    +u->v[c][1].im*g->f[1][1].re+u->v[c][1].re*g->f[1][1].im
    +u->v[c][2].im*g->f[1][2].re+u->v[c][2].re*g->f[1][2].im;
}

```

5.8.16 Parts of $\eta - S_{xy}\psi$

53b $\langle \text{Compute inside part for } 1 - S_{xy} \text{ 53b} \rangle \equiv$

```

for (i = 0; i < nb->inside_size; i++) {
    const vFermion *ex5, *es;

    xyzt = nb->inside[i];
    xyzt5 = xyzt * S_4;
     $\langle \text{Extract 1-d addresses 58e} \rangle$ 
    ex5 = &eta[xyzt5];
     $\langle \text{Build SSE } SU(3) \text{ objects 57c} \rangle$ 
     $\langle \text{Compute } 1 - S_{xy} \text{ part on the inside s-chain 53d} \rangle$ 
}

```

53c $\langle \text{Compute boundary part for } 1 - S_{xy} \text{ 53c} \rangle \equiv$

```

for (i = 0; i < nb->boundary_size; i++) {
    const vFermion *ex5, *es;
    int m = nb->boundary[i].mask;

    xyzt = nb->boundary[i].index;
    xyzt5 = xyzt * S_4;
     $\langle \text{Extract 1-d addresses 58e} \rangle$ 
    ex5 = &eta[xyzt5];
     $\langle \text{Build SSE } SU(3) \text{ objects 57c} \rangle$ 
     $\langle \text{Compute } 1 - S_{xy} \text{ part on the boundary s-chain 53e} \rangle$ 
}

```

53d $\langle \text{Compute } 1 - S_{xy} \text{ part on the inside s-chain 53d} \rangle \equiv$

```

for (s = 0; s < S_4; s++) {
     $\langle \text{Compute } S \text{ inside } \gamma\text{-projections 54a} \rangle$ 
     $\langle \text{Inside multiply by } Vs \text{ 52c} \rangle$ 
     $\langle \text{Compute } 1 - S \gamma\text{-unprojections and sum the results 54c} \rangle$ 
}

```

53e $\langle \text{Compute } 1 - S_{xy} \text{ part on the boundary s-chain 53e} \rangle \equiv$

```

for (s = 0; s < S_4; s++) {
     $\langle \text{Compute } S \text{ boundary } \gamma\text{-projections 54b} \rangle$ 
     $\langle \text{Boundary multiply by } Vs \text{ 52d} \rangle$ 
     $\langle \text{Compute } 1 - S \gamma\text{-unprojections and sum the results 54c} \rangle$ 
}

```

```

54a  <Compute  $S$  inside  $\gamma$ -projections 54a>≡
      <Construct neighbor pointers 57d>
      for (c = 0; c < 3; c++) {
          k=0; f=&psi[ps[0]]; g=&gg[0]; <Build  $(1 - \gamma_0)$  projection of  $*f$  in  $*g$  4c>
          k=1; f=&psi[ps[1]]; g=&gg[1]; <Build  $(1 + \gamma_0)$  projection of  $*f$  in  $*g$  4a>
          k=2; f=&psi[ps[2]]; g=&gg[2]; <Build  $(1 - \gamma_1)$  projection of  $*f$  in  $*g$  4g>
          k=3; f=&psi[ps[3]]; g=&gg[3]; <Build  $(1 + \gamma_1)$  projection of  $*f$  in  $*g$  4e>
          k=4; f=&psi[ps[4]]; g=&gg[4]; <Build  $(1 - \gamma_2)$  projection of  $*f$  in  $*g$  5c>
          k=5; f=&psi[ps[5]]; g=&gg[5]; <Build  $(1 + \gamma_2)$  projection of  $*f$  in  $*g$  5a>
          k=6; f=&psi[ps[6]]; g=&gg[6]; <Build  $(1 - \gamma_3)$  projection of  $*f$  in  $*g$  5g>
          k=7; f=&psi[ps[7]]; g=&gg[7]; <Build  $(1 + \gamma_3)$  projection of  $*f$  in  $*g$  5e>
      }

54b  <Compute  $S$  boundary  $\gamma$ -projections 54b>≡
      <Construct neighbor pointers 57d>
      for (c = 0; c < 3; c++) {
          if ((m & 0x01) == 0) {
              k=0; f=&psi[ps[0]]; g=&gg[0]; <Build  $(1 - \gamma_0)$  projection of  $*f$  in  $*g$  4c>
          }
          if ((m & 0x02) == 0) {
              k=1; f=&psi[ps[1]]; g=&gg[1]; <Build  $(1 + \gamma_0)$  projection of  $*f$  in  $*g$  4a>
          }
          if ((m & 0x04) == 0) {
              k=2; f=&psi[ps[2]]; g=&gg[2]; <Build  $(1 - \gamma_1)$  projection of  $*f$  in  $*g$  4g>
          }
          if ((m & 0x08) == 0) {
              k=3; f=&psi[ps[3]]; g=&gg[3]; <Build  $(1 + \gamma_1)$  projection of  $*f$  in  $*g$  4e>
          }
          if ((m & 0x10) == 0) {
              k=4; f=&psi[ps[4]]; g=&gg[4]; <Build  $(1 - \gamma_2)$  projection of  $*f$  in  $*g$  5c>
          }
          if ((m & 0x20) == 0) {
              k=5; f=&psi[ps[5]]; g=&gg[5]; <Build  $(1 + \gamma_2)$  projection of  $*f$  in  $*g$  5a>
          }
          if ((m & 0x40) == 0) {
              k=6; f=&psi[ps[6]]; g=&gg[6]; <Build  $(1 - \gamma_3)$  projection of  $*f$  in  $*g$  5g>
          }
          if ((m & 0x80) == 0) {
              k=7; f=&psi[ps[7]]; g=&gg[7]; <Build  $(1 + \gamma_3)$  projection of  $*f$  in  $*g$  5e>
          }
      }

54c  <Compute  $1 - S$   $\gamma$ -unprojections and sum the results 54c>≡
      rs = &rx5[s];
      es = &ex5[s];
      for (c = 0; c < 3; c++) {
          k = 7; <Unproject  $(1 + \gamma_3)$  link 5f>
          k = 6; <Unproject and accumulate  $(1 - \gamma_3)$  link 5h>
          k = 3; <Unproject and accumulate  $(1 + \gamma_1)$  link 4f>
          k = 2; <Unproject and accumulate  $(1 - \gamma_1)$  link 4h>
          k = 0; <Unproject and accumulate  $(1 - \gamma_0)$  link 4d>
          k = 1; <Unproject and accumulate  $(1 + \gamma_0)$  link 4b>
          k = 4; <Unproject and accumulate  $(1 - \gamma_2)$  link 5d>
          k = 5; <Unproject and accumulate  $(1 + \gamma_2)$  link 5b>
          <Compute  $(*rs) \leftarrow \eta - (*rs)$  for color  $c$  55a>
      }

```

55a $\langle \text{Compute } (*rs) \leftarrow \eta - (*rs) \text{ for color } c \text{ 55a} \rangle \equiv$
 $rs \rightarrow f[0][c].re = es \rightarrow f[0][c].re - rs \rightarrow f[0][c].re;$
 $rs \rightarrow f[0][c].im = es \rightarrow f[0][c].im - rs \rightarrow f[0][c].im;$
 $rs \rightarrow f[1][c].re = es \rightarrow f[1][c].re - rs \rightarrow f[1][c].re;$
 $rs \rightarrow f[1][c].im = es \rightarrow f[1][c].im - rs \rightarrow f[1][c].im;$
 $rs \rightarrow f[2][c].re = es \rightarrow f[2][c].re - rs \rightarrow f[2][c].re;$
 $rs \rightarrow f[2][c].im = es \rightarrow f[2][c].im - rs \rightarrow f[2][c].im;$
 $rs \rightarrow f[3][c].re = es \rightarrow f[3][c].re - rs \rightarrow f[3][c].re;$
 $rs \rightarrow f[3][c].im = es \rightarrow f[3][c].im - rs \rightarrow f[3][c].im;$

5.8.17 Parts of $Q_{xx}^{-1}Q_{xy}\psi$

55b $\langle \text{Compute inside part for } Q_{xx}^{-1}Q_{xy} \text{ 55b} \rangle \equiv$
for (i = 0; i < nb->inside_size; i++) {
 xyzt = nb->inside[i];
 xyzt5 = xyzt * S_4;
 $\langle \text{Extract 1-d addresses 58e} \rangle$
 $\langle \text{Build SSE } SU(3) \text{ objects 57c} \rangle$
 $\langle \text{Compute } Q_{xy} \text{ part on the inside s-chain 51c} \rangle$
 $\langle \text{Compute } Q_{xx}^{-1} \text{ part on the s-chain 38b} \rangle$
}

55c $\langle \text{Compute boundary part for } Q_{xx}^{-1}Q_{xy} \text{ 55c} \rangle \equiv$
for (i = 0; i < nb->boundary_size; i++) {
 int m = nb->boundary[i].mask;

 xyzt = nb->boundary[i].index;
 xyzt5 = xyzt * S_4;
 $\langle \text{Extract 1-d addresses 58e} \rangle$
 $\langle \text{Build SSE } SU(3) \text{ objects 57c} \rangle$
 $\langle \text{Compute } Q_{xy} \text{ part on the boundary s-chain 51d} \rangle$
 $\langle \text{Compute } Q_{xx}^{-1} \text{ part on the s-chain 38b} \rangle$
}

5.8.18 Parts of $S_{xx}^{-1}S_{xy}\psi$

55d $\langle \text{Compute inside part for } S_{xx}^{-1}S_{xy} \text{ 55d} \rangle \equiv$
for (i = 0; i < nb->inside_size; i++) {
 xyzt = nb->inside[i];
 xyzt5 = xyzt * S_4;
 $\langle \text{Extract 1-d addresses 58e} \rangle$
 $\langle \text{Build SSE } SU(3) \text{ objects 57c} \rangle$
 $\langle \text{Compute } S_{xy} \text{ part on the inside s-chain 56a} \rangle$
 $\langle \text{Compute } S_{xx}^{-1} \text{ part on the s-chain 38c} \rangle$
}

55e $\langle \text{Compute boundary part for } S_{xx}^{-1}S_{xy} \text{ 55e} \rangle \equiv$
for (i = 0; i < nb->boundary_size; i++) {
 int m = nb->boundary[i].mask;

 xyzt = nb->boundary[i].index;
 xyzt5 = xyzt * S_4;
 $\langle \text{Extract 1-d addresses 58e} \rangle$
 $\langle \text{Build SSE } SU(3) \text{ objects 57c} \rangle$
 $\langle \text{Compute } S_{xy} \text{ part on the boundary s-chain 56b} \rangle$
 $\langle \text{Compute } S_{xx}^{-1} \text{ part on the s-chain 38c} \rangle$
}


```

56a  <Compute  $S_{xy}$  part on the inside  $s$ -chain 56a>≡
      for (s = 0; s < S_4; s++) {
        <Compute  $S$  inside  $\gamma$ -projections 54a>
        <Inside multiply by  $Vs$  52c>
        <Compute  $S$   $\gamma$ -unprojections and sum the results 56c>
      }

56b  <Compute  $S_{xy}$  part on the boundary  $s$ -chain 56b>≡
      for (s = 0; s < S_4; s++) {
        <Compute  $S$  boundary  $\gamma$ -projections 54b>
        <Boundary multiply by  $Vs$  52d>
        <Compute  $S$   $\gamma$ -unprojections and sum the results 56c>
      }

56c  <Compute  $S$   $\gamma$ -unprojections and sum the results 56c>≡
      rs = &rx5[s];
      for (c = 0; c < 3; c++) {
        k = 7; <Unproject  $(1 + \gamma_3)$  link 5f>
        k = 6; <Unproject and accumulate  $(1 - \gamma_3)$  link 5h>
        k = 3; <Unproject and accumulate  $(1 + \gamma_1)$  link 4f>
        k = 2; <Unproject and accumulate  $(1 - \gamma_1)$  link 4h>
        k = 0; <Unproject and accumulate  $(1 - \gamma_0)$  link 4d>
        k = 1; <Unproject and accumulate  $(1 + \gamma_0)$  link 4b>
        k = 4; <Unproject and accumulate  $(1 - \gamma_2)$  link 5d>
        k = 5; <Unproject and accumulate  $(1 + \gamma_2)$  link 5b>
      }

```

5.8.19 Parts of $\eta - Q_{xx}^{-1}Q_{xy}\psi$

```

56d  <Compute inside part for  $1 - Q_{xx}^{-1}Q_{xy}$  56d>≡
      for (i = 0; i < nb->inside_size; i++) {
        const vFermion *ex5, *es;

        xyz5 = nb->inside[i];
        xyz5 = xyz5 * S_4;
        <Extract 1-d addresses 58e>
        ex5 = &eta[xyz5];
        <Build SSE  $SU(3)$  objects 57c>
        <Compute  $Q_{xy}$  part on the inside  $s$ -chain 51c>
        <Compute  $1 - Q_{xx}^{-1}$  part on the  $s$ -chain 57a>
      }

56e  <Compute boundary part for  $1 - Q_{xx}^{-1}Q_{xy}$  56e>≡
      for (i = 0; i < nb->boundary_size; i++) {
        const vFermion *ex5, *es;
        int m = nb->boundary[i].mask;

        xyz5 = nb->boundary[i].index;
        xyz5 = xyz5 * S_4;
        <Extract 1-d addresses 58e>
        ex5 = &eta[xyz5];
        <Build SSE  $SU(3)$  objects 57c>
        <Compute  $Q_{xy}$  part on the boundary  $s$ -chain 51d>
        <Compute  $1 - Q_{xx}^{-1}$  part on the  $s$ -chain 57a>
      }

```

57a $\langle \text{Compute } 1 - Q_{xx}^{-1} \text{ part on the } s\text{-chain } 57a \rangle \equiv$
 $\langle \text{Compute } Q_{xx}^{-1} \text{ part on the } s\text{-chain } 38b \rangle$
for (s = 0; s < S_4; s++) {
rs = &rx5[s];
es = &ex5[s];
for (c = 0; c < 3; c++) {
 $\langle \text{Compute } (*rs) \leftarrow \eta - (*rs) \text{ and collect } \langle r, r \rangle \text{ } 57b \rangle$
}
}
}

57b $\langle \text{Compute } (*rs) \leftarrow \eta - (*rs) \text{ and collect } \langle r, r \rangle \text{ } 57b \rangle \equiv$
vv = es->f[0][c].re - rs->f[0][c].re; rs->f[0][c].re = vv; nv += vv * vv;
vv = es->f[0][c].im - rs->f[0][c].im; rs->f[0][c].im = vv; nv += vv * vv;
vv = es->f[1][c].re - rs->f[1][c].re; rs->f[1][c].re = vv; nv += vv * vv;
vv = es->f[1][c].im - rs->f[1][c].im; rs->f[1][c].im = vv; nv += vv * vv;
vv = es->f[2][c].re - rs->f[2][c].re; rs->f[2][c].re = vv; nv += vv * vv;
vv = es->f[2][c].im - rs->f[2][c].im; rs->f[2][c].im = vv; nv += vv * vv;
vv = es->f[3][c].re - rs->f[3][c].re; rs->f[3][c].re = vv; nv += vv * vv;
vv = es->f[3][c].im - rs->f[3][c].im; rs->f[3][c].im = vv; nv += vv * vv;

5.8.20 Miscallienious

We also need to uplift the gauge fields

57c $\langle \text{Build SSE } SU(3) \text{ objects } 57c \rangle \equiv$
Uup = &U[nb->site[xyzt].Uup];
for (d = 0; d < 4; d++, Uup++) {
Udown = &U[nb->site[xyzt].Udown[d]];
for (c1 = 0; c1 < 3; c1++) {
for (c2 = 0; c2 < 3; c2++) {
V[d*2+0].v[c1][c2].re = vmk1(Uup->v[c1][c2].re);
V[d*2+0].v[c1][c2].im = vmk1(Uup->v[c1][c2].im);
/* conjugate down-link */
V[d*2+1].v[c1][c2].re = vmk1(Udown->v[c2][c1].re);
V[d*2+1].v[c1][c2].im = vmk1(-Udown->v[c2][c1].im);
}
}
}
}

We want to keep code small, so computing the neighbors is done in a loop:

57d $\langle \text{Construct neighbor pointers } 57d \rangle \equiv$
for (d = 0; d < 8; d++) {
ps[d] = p5[d] + s;
}

5.8.21 Combined pieces

In these cases, Q_{xx}^{-1} is applied to the result of Q_{xy}

```

58a  ⟨Static function prototypes 16b⟩+≡
      static void compute_Qxx1Qxy(vFermion *d,
                                const vFermion *s,
                                struct neighbor *nb);
      static void inline compute_Qee1Qeo(vEvenFermion *d, const vOddFermion *s)
      {
        compute_Qxx1Qxy(&d->f, &s->f, &even_odd);
      }

      static void compute_Sxx1Sxy(vFermion *d,
                                const vFermion *s,
                                struct neighbor *nb);
      static void inline compute_See1Seo(vEvenFermion *d, const vOddFermion *s)
      {
        compute_Sxx1Sxy(&d->f, &s->f, &even_odd);
      }

      static void compute_1Qxx1Qxy(vFermion *d,
                                double *norm,
                                const vFermion *q,
                                const vFermion *s,
                                struct neighbor *nb);
      static void inline compute_1Qoo1Qoe(vOddFermion *d,
                                double *norm,
                                const vOddFermion *q,
                                const vEvenFermion *s)
      {
        compute_1Qxx1Qxy(&d->f, norm, &q->f, &s->f, &odd_even);
      }

```

5.8.22 Common Locals

Some local bindings are used by all parts above. Let us collect them together.

```

58b  ⟨Q common locals 58b⟩≡
      int i, xyzt5, s, c;
      vFermion * __restrict__ rx5, * __restrict__ rs;

```

Others are used only in Z_{xy} parts:

```

58c  ⟨Qxy locals 58c⟩≡
      int xyzt, k, d;
      const vFermion *f;
      vHalfFermion *g;
      vHalfFermion gg[8], hh[8];
      vSU3 V[8];
      int ps[8], p5[8];

```

```

58d  ⟨Qxy locals 58c⟩+≡
      const SU3 *Up, *Udown;
      int c1, c2;

```

```

58e  ⟨Extract 1-d addresses 58e⟩≡
      ⟨Compute rx5 58f⟩

```

```

58f  ⟨Compute rx5 58f⟩≡
      rx5 = &chi[xyzt5];

```

```

58g  ⟨Compute qx5 58g⟩≡
      qx5 = &psi[xyzt5];

```

Local variables needed to compute diagonal parts

```
59a  <Qxx locals 59a>≡
      vReal fx;
      vHalfFermion zV;
      vcomplex zn, z1, z2, z3;
      complex zX[2][3];
      vHalfFermion xOut;
      vHalfFermion yOut;
```

5.8.23 Common globals

Globals that do not change from run to run:

```
59b  <Global variables 10a>+≡
      static REAL c0;
      static vReal va0;
      static vReal va1;
      static vReal va2;
      static vReal va3;
      static vReal va4;
      static vReal ab_LA;
      static vReal ab_LB;
```

Some of these values depend of `m0` and `M`. Here we compute their values:

```
59c  <Compute constant values for  $Q_{xx}^{-1}$  and  $S_{xx}^{-1}$  59c>≡
      {
          double a = M;
          double b = 2.;
          double c = -2*m0;
          vReal ab = vmk1(-b/a);

          c0 = 1./(1+c/b*pow(b/a, S_4*4));
          ab_LA = vmk4(c*c0/a, -b*c*c0/(a*a), b*b*c*c0/(a*a*a), -b*b*b*c*c0/(a*a*a*a));
          ab_LB = vmk4(-b*b*b*c*c0/(a*a*a*a), b*b*c*c0/(a*a*a), -b*c*c0/(a*a), c*c0/a);
          va0 = vmk1(1/a);
          va1 = va0 * ab;
          va2 = va1 * ab;
          va3 = va2 * ab;
          va4 = va3 * ab;
      }
```

We need `math.h` for the prototype of `pow()`:

```
59d  <Include files 29d>+≡
      #include <math.h>
```

5.9 QMP Pieces

Here are miscellaneous piece of QMP:

We are ready to use the result of the global sum. Check that it has been computed.

```
59e  <Finalize <r,r> computation 59e>≡
      /* relax, QMP does not support split reductions yet. */
```

All receive operations are bundled together, so that starting and stopping them is easy:

```
59f  <Start off-diagonal receives 59f>≡
      QMP_start(nb->qmp_cr);
```

```
59g  <Finish off-diagonal receives 59g>≡
      QMP_wait(nb->qmp_cr);
```

For the send operations, we can easily start them separately, because of the way the send buffers are filled. Since the case of `network[d]==2` is handled specially, we use `qmp_smask` to decide when to start send.

```
60a  <Start k-send 60a>≡
      if (nb->qmp_smask & (1 << k)) {
          QMP_start(nb->qmp_sh[k]);
          sending = nb;
      }
```

It is convenient to invert waiting for the send to complete—we only wait for it just before the send buffer is filled again and at the end of CG to provide a clean completion to the routine.

```
60b  <Finish old off-diagonal sends 60b>≡
      if (sending) {
          int i; /* This is QMP_wait_vector(nb->qmp_sv, nb->Ns); */
          for (i = sending->Ns; i--;)
              QMP_wait(sending->qmp_sv[i]);
          sending = 0;
      }
```

A global variable `sending` is set to a corresponding `struct neighbor` when a send operation is started, so that we do not wait on never started send.

```
60c  <Global variables 10a>+≡
      static struct neighbor *sending = 0;
```

5.9.1 Global sums

Until the split global sums are implemented in QMP, everything is done at the beginning, when `*norm` contains the local part of the sum. Start the global operation which will distribute the pieces, compute the sum, and provide the result to each node.

```
60d  <Start <r, r> computation 60d>≡
      QMP_sum_double(norm);
```

5.10 SSE Types and Operations

It is convenient to place all SSE specific matter into a separate file which we include into the solver source:

```
60e  <Include files 29d>+≡
      #define Vs 4 /* Length of SSE vector */
      #define REAL float /* floating point type compatible with vReal */
      #include <sse.h>
```

Here we define the top level structure of `sse.h`:

```
60f  <sse.h 60f>≡
      #ifndef _SSE_H
      #define _SSE_H
      <SSE types 61a>
      <SSE inline functions 62a>
      #endif
```

5.10.1 SSE Types

Let us start with the floating point scalar type. The C standard does not provide proper type encapsulation in `typedef`, but we do not need a fool-proof solution here.

First is our floating point vector type. The fact that its length is four is used heavily in the code above and below. The attribute `aligned(16)` helps gcc to keep variables properly aligned.

```
61a  <SSE types 61a>≡
      typedef REAL vReal __attribute__((mode(V4SF),aligned(16)));
```

Now, let us declare complex types. They come in two kinds: scalar and vector, as usual.

```
61b  <SSE types 61a>+≡
      typedef struct {
          REAL re, im;
      } complex;

      typedef struct {
          vReal re, im;
      } vcomplex;
```

We handled enough general cases to express lattice specific data types. The gauge field needs two kind of types (yes, they are scalar and vector, what else?):

```
61c  <SSE types 61a>+≡
      typedef struct SU3 {
          complex v[3][3];
      } SU3;

      typedef struct {
          vcomplex v[3][3];
      } vSU3;
```

But we only use vector fermions. However, two component spinors come handy (note that we have committed to the color index varying faster than the spinor index. Is it a good choice?—That is unclear. Changes throughout the code are needed, however, to flip the order of indices.)

```
61d  <SSE types 61a>+≡
      typedef struct {
          vcomplex f[4][3];
      } vFermion;

      typedef struct {
          vcomplex f[2][3];
      } vHalfFermion;
```

Strictly speaking, there is no need to have separate types for even/odd sublattices. But, while writing the CG, the compiler caught quite a few logic errors because of these two tiny structures.

```
61e  <SSE types 61a>+≡
      typedef struct {
          vFermion f;
      } vEvenFermion;

      typedef struct {
          vFermion f;
      } vOddFermion;
```

5.10.2 SSE inline functions

For efficiency, all functions dealing with SSE data are inlined. The code below requires `gcc 3.3.x`.

By the good grace of `gcc` we already have arithmetic operations and assignments on SSE vectors. A few more functions will complete the needed set. All functions below are defined `inline`, so that `gcc` can eliminate the standard function call dance. [NB: In fact, `gcc` does a reasonably good job in dissolving inline function calls in C, but sometimes a residue is left. If you want to use `inline`, frequent consultations with `gcc -S` are advantageous].

First, propagate a scalar value to all four components of the SSE vector.

```
62a  <SSE inline functions 62a>≡
      static inline vReal vmk1(REAL a) {
          vReal v = __builtin_ia32_loadss((float *)&a);
          asm("shufps\t$0,%0,%0" : "+x" (v));
          return v;
      }
```

Packaging four values into an SSE vector is next. This defines the numbering conventions: which element is zeroth etc.

```
62b  <SSE inline functions 62a>+≡
      static inline vReal vmk4(REAL a0, REAL a1, REAL a2, REAL a3) {
          vReal v;
          REAL *r = (REAL *)&v;
          r[0] = a0;
          r[1] = a1;
          r[2] = a2;
          r[3] = a3;
          return v;
      }
```

Next, sum all four components of the SSE vector. Maybe there is a craftier way to do it, but let us leave it as an exercise for now:

```
62c  <SSE inline functions 62a>+≡
      static inline REAL vsum(vReal v)
      {
          REAL *vv = (REAL *)&v;
          return vv[0] + vv[1] + vv[2] + vv[3];
      }
```

Mutators for vector numbers. We only need access to the 0th and the 3rd elements of the vector:

```
62d  <SSE inline functions 62a>+≡
      static inline void vput_3(vReal *v, REAL a3)
      {
          ((REAL *)v)[3] = a3;
      }

      static inline void vput_0(vReal *v, REAL a0)
      {
          ((REAL *)v)[0] = a0;
      }
```

Given

$$\begin{aligned} a &= (a_0, a_1, a_2, a_3), \\ b &= (b_0, b_1, b_2, b_3), \end{aligned}$$

compute various shifts as follows:

$$\text{shift_up1} \leftarrow (a_1, a_2, a_3, b_0)$$

63a $\langle \text{SSE inline functions 62a} \rangle + \equiv$

```
static inline vReal shift_up1(vReal a, vReal b)
{
    vReal x = a;
    vReal y = b;
    asm("shufps\t$0x30,%0,%1\n\t"
        "shufps\t$0x29,%1,%0"
        : "+x" (x), "+x" (y));
    return x;
}
```

$$\text{shift_up2} \leftarrow (a_2, a_3, b_0, b_1)$$

63b $\langle \text{SSE inline functions 62a} \rangle + \equiv$

```
static inline vReal shift_up2(vReal a, vReal b)
{
    vReal x = a;
    asm("shufps\t$0x4e,%1,%0"
        : "+x" (x): "x" (b));
    return x;
}
```

$$\text{shift_up3} \leftarrow (a_3, b_0, b_1, b_2)$$

63c $\langle \text{SSE inline functions 62a} \rangle + \equiv$

```
static inline vReal shift_up3(vReal a, vReal b)
{
    vReal x = a;
    asm("shufps\t$0x03,%1,%0\n\t"
        "shufps\t$0x9c,%1,%0"
        : "+x" (x): "x" (b));
    return x;
}
```

$$\text{shift_down1} \leftarrow (a_3, b_0, b_1, b_2)$$

63d $\langle \text{SSE inline functions 62a} \rangle + \equiv$

```
static inline vReal shift_down1(vReal a, vReal b)
{
    return shift_up3(a, b);
}
```

$$\text{shift_down2} \leftarrow (a_2, a_3, b_0, b_1)$$

63e $\langle \text{SSE inline functions 62a} \rangle + \equiv$

```
static inline vReal shift_down2(vReal a, vReal b)
{
    return shift_up2(a, b);
}
```


$\text{shift_down3} \leftarrow (a_1, a_2, a_3, b_0)$

```
64a  <SSE inline functions 62a>+≡
      static inline vReal shift_down3(vReal a, vReal b)
      {
          return shift_up1(a, b);
      }
```

The very last of the SSE functions: clear a half fermion:

```
64b  <SSE inline functions 62a>+≡
      static inline void vhfzero(vHalfFermion *v)
      {
          vReal z = vmk1(0.0);

          v->f[0][0].re = v->f[0][0].im =
          v->f[0][1].re = v->f[0][1].im =
          v->f[0][2].re = v->f[0][2].im =
          v->f[1][0].re = v->f[1][0].im =
          v->f[1][1].re = v->f[1][1].im =
          v->f[1][2].re = v->f[1][2].im = z;
      }
```

5.11 Generally Useful Functions

Here is a collection of simple functions that are useful throughout the code:

```
64c  <Static function prototypes 16b>+≡
      static inline int
      parity(const int x[DIM])
      {
          int i, v;
          for (i = v = 0; i < DIM; i++)
              v += x[i];
          return v & 1;
      }
```

5.12 Handy Constants

For some constants it is better to have symbolic names even if one can not easily change their values.

```
64d  <Macro definitions 15b>+≡
      #define Nc  3      /* Number of colors */
      #define DIM 4      /* number of dimensions */
      #define Fd  4      /* Fermion representation dimension */
```

5.13 Source File

Finally, let us put together all the pieces:

```
64e  <dwf.c 64e>≡
      #include <stdlib.h>
      #include "sse-dwf-cg.h"
      <Include files 29d>
      <Macro definitions 15b>

      <Data types 11f>
      <Global variables 10a>
      <Static function prototypes 16b>
      <Static functions 15a>
      <Interface functions 10b>
```

6 TODO LIST

- 65a \langle *Static function prototypes* 16b $\rangle + \equiv$
 int hf_even4to1(const int []);
 int hf_odd4to1(const int []);
 int gauge4to1(const int []);
- 65b \langle *Cleanup QMP* 65b $\rangle \equiv$
 /* XXX undo Init */
- 65c \langle *Free tables* 65c $\rangle \equiv$
 /* XXX --- go through init tables and get all memory allocations */

7 CHUNKS

<Advance DIM- d index for DIM-1- d scan 18f>
 <Advance DIM- d index for full sublattice scan 18d>
 <Advance DIM- d index for full sublattice scan locally 18e>
 <Advance x at i 18g>
 <Advance x at i locally 18h>
 <Allocate boundary table 23d>
 <Allocate down buffers 30d>
 <Allocate fields 32d>
 <Allocate inside table 23c>
 <Allocate up buffers 30c>
 <Boundary multiply by V s 52d>
 <Build SSE $SU(3)$ objects 57c>
 <Build $(1 + \gamma_0)$ projection of $*f$ in $*g$ 4a>
 <Build $(1 + \gamma_1)$ projection of $*f$ in $*g$ 4e>
 <Build $(1 + \gamma_2)$ projection of $*f$ in $*g$ 5a>
 <Build $(1 + \gamma_3)$ projection of $*f$ in $*g$ 5e>
 <Build $(1 - \gamma_0)$ projection of $*f$ in $*g$ 4c>
 <Build $(1 - \gamma_1)$ projection of $*f$ in $*g$ 4g>
 <Build $(1 - \gamma_2)$ projection of $*f$ in $*g$ 5c>
 <Build $(1 - \gamma_3)$ projection of $*f$ in $*g$ 5g>
 <Build local neighbors 26d>
 <Build outside indices 26e>
 <Check floating point size 11a>
 <Check lattice size 11b>
 <Check xx -aliasing of q 45c>
 <Cleanup QMP 65b>
 <Clump up and down directions 30b>
 <Compute $A^{-1}\psi$ on the lower two components 38e>
 <Compute $A^{-1}\psi$ on the upper two components 38d>
 <Compute $B^{-1}\psi$ on the lower two components 38g>
 <Compute $B^{-1}\psi$ on the upper two components 38f>
 <Compute L_A^{-1} on the lower components 40b>
 <Compute L_A^{-1} on the upper components 39>
 <Compute L_B^{-1} on the lower components 41b>
 <Compute L_B^{-1} on the upper components 41a>
 <Compute Q boundary γ -projections 52a>
 <Compute Q γ -unprojections and sum the results 52b>
 <Compute Q inside γ -projections 51e>
 <Compute $1 - Q_{xx}^{-1}$ part on the s -chain 57a>
 <Compute Q_{xx}^{-1} part on the s -chain 38b>
 <Compute Q_{xy} part on the boundary s -chain 51d>
 <Compute Q_{xy} part on the inside s -chain 51c>
 <Compute R_A^{-1} on the lower components 42b>
 <Compute R_A^{-1} on the upper components 42a>
 <Compute R_B^{-1} on the lower components 42d>
 <Compute R_B^{-1} on the upper components 42c>
 <Compute S boundary γ -projections 54b>
 <Compute $1 - S$ γ -unprojections and sum the results 54c>
 <Compute S γ -unprojections and sum the results 56c>
 <Compute S inside γ -projections 54a>
 <Compute S_{xx}^{-1} part on the s -chain 38c>
 <Compute $1 - S_{xy}$ part on the boundary s -chain 53e>
 <Compute S_{xy} part on the boundary s -chain 56b>
 <Compute $1 - S_{xy}$ part on the inside s -chain 53d>
 <Compute S_{xy} part on the inside s -chain 56a>
 <Compute boundary part for $1 - Q_{xx}^{-1}Q_{xy}$ 56e>
 <Compute boundary part for $Q_{xx}^{-1}Q_{xy}$ 55c>
 <Compute boundary part for Q_{xy} 51b>

⟨Compute boundary part for $S_{xx}^{-1}S_{xy}$ 55e⟩
 ⟨Compute boundary part for $1 - S_{xy}$ 53c⟩
 ⟨Compute constant values for Q_{xx}^{-1} and S_{xx}^{-1} 59c⟩
 ⟨Compute init sizes 22a⟩
 ⟨Compute inside part for $1 - Q_{xx}^{-1}Q_{xy}$ 56d⟩
 ⟨Compute inside part for $Q_{xx}^{-1}Q_{xy}$ 55b⟩
 ⟨Compute inside part for Q_{xy} 51a⟩
 ⟨Compute inside part for $S_{xx}^{-1}S_{xy}$ 55d⟩
 ⟨Compute inside part for $1 - S_{xy}$ 53b⟩
 ⟨Compute **inside_size** and **boundary_size** 23b⟩
 ⟨Compute **p** and **m** 25e⟩
 ⟨Compute projections for Q send 49a⟩
 ⟨Compute projections for S send 49b⟩
 ⟨Compute ψ_e 34a⟩
 ⟨Compute **qx5** 58g⟩
 ⟨Compute $\langle *rs \rangle \leftarrow \eta - \langle *rs \rangle$ and collect $\langle r, r \rangle$ 57b⟩
 ⟨Compute $\langle *rs \rangle \leftarrow \eta - \langle *rs \rangle$ for color c 55a⟩
 ⟨Compute **rx5** 58f⟩
 ⟨Compute send sizes and allocate index tables 23e⟩
 ⟨Compute φ_o 32b⟩
 ⟨Compute wall value in **zX[c]** 41c⟩
 ⟨Compute $y_{k,[0]}^{(A)}$ 43a⟩
 ⟨Compute $y_{k,[1]}^{(A)}$ 43b⟩
 ⟨Compute $y_{k,[2]}^{(A)}$ 43c⟩
 ⟨Compute $y_{k,[3]}^{(A)}$ 44a⟩
 ⟨Compute $y_{k,[0]}^{(B)}$ 44b⟩
 ⟨Compute $y_{k,[1]}^{(B)}$ 44c⟩
 ⟨Compute $y_{k,[2]}^{(B)}$ 45a⟩
 ⟨Compute $y_{k,[3]}^{(B)}$ 45b⟩
 ⟨Compute $zV \leftarrow zV + fx * qs^{down}$ 40c⟩
 ⟨Compute $zV \leftarrow zV + fx * qs^{up}$ 40a⟩
 ⟨Construct $(1 + \gamma_0)$ send k -buffer 49c⟩
 ⟨Construct $(1 + \gamma_1)$ send k -buffer 50a⟩
 ⟨Construct $(1 + \gamma_2)$ send k -buffer 50c⟩
 ⟨Construct $(1 + \gamma_3)$ send k -buffer 50e⟩
 ⟨Construct $(1 - \gamma_0)$ send k -buffer 49d⟩
 ⟨Construct $(1 - \gamma_1)$ send k -buffer 50b⟩
 ⟨Construct $(1 - \gamma_2)$ send k -buffer 50d⟩
 ⟨Construct $(1 - \gamma_3)$ send k -buffer 50f⟩
 ⟨Construct neighbor pointers 57d⟩
 ⟨Construct the collective handle 31f⟩
 ⟨Construct the initial point of the hypersurface 27c⟩
 ⟨Construct the neighbor's network coordinates **xc** and bounds **xb** 27b⟩
 ⟨Data types 11f⟩
 ⟨End xx -aliasing of q 45d⟩
 ⟨Extract 1-d addresses 58e⟩
 ⟨Finalize $\langle r, r \rangle$ computation 59e⟩
 ⟨Find index of a borrowed gauge link 29c⟩
 ⟨Find index of a regular gauge link 29b⟩
 ⟨Finish off-diagonal receives 59g⟩
 ⟨Finish old off-diagonal sends 60b⟩
 ⟨Finish xy -aliasing of q 48d⟩
 ⟨Free fields 32e⟩
 ⟨Free tables 65c⟩
 ⟨Get network topology 17a⟩
 ⟨Global variables 10a⟩
 ⟨Handle init errors 10c⟩

<Include files 29d>
 <Init out of bound x and y 42e>
 <Initialize QMP 29e>
 <Initialize **out** and **p** 25c>
 <Initialize tables 21b>
 <Insert **k** into **site[p].F[dx]** 28c>
 <Inside multiply by Vs 52c>
 <Interface functions 10b>
 <Load DIM gauge links from U at \mathbf{x} 18a>
 <Load a **d** gauge link from V at \mathbf{x} 19b>
 <Load an s -line of fermion at \mathbf{x} 19d>
 <Load gauge boundary in direction **d** 19a>
 <Macro definitions 15b>
 <Multiply ***u** by ***g** and store the result in ***h** 53a>
 < Q common locals 58b>
 < Q_{xx} locals 59a>
 < Q_{xy} locals 58c>
 <Read fermion 19c>
 <Read gauge field 17c>
 <SSE inline functions 62a>
 <SSE types 61a>
 <Save an s -line of fermion at \mathbf{x} 20c>
 <Setup **boundary** 26c>
 <Setup **boundary** or **inside** 26a>
 <Setup heap management functions 10d>
 <Setup **inside** 26b>
 <Setup xy -aliasing of q 48c>
 <Solve $M^\dagger M \psi_o = \varphi_o$ 32f>
 <Start DIM- d sublattice scan 18b>
 <Start DIM- d sublattice scan locally 18c>
 <Start k -send 60a>
 <Start $\langle r, r \rangle$ computation 60d>
 <Start off-diagonal receives 59f>
 <Static function prototypes 16b>
 <Static functions 15a>
 <Translate \mathbf{x} to target **p** 28b>
 <Unproject and accumulate $(1 + \gamma_0)$ link 4b>
 <Unproject and accumulate $(1 + \gamma_1)$ link 4f>
 <Unproject and accumulate $(1 + \gamma_2)$ link 5b>
 <Unproject and accumulate $(1 - \gamma_0)$ link 4d>
 <Unproject and accumulate $(1 - \gamma_1)$ link 4h>
 <Unproject and accumulate $(1 - \gamma_2)$ link 5d>
 <Unproject and accumulate $(1 - \gamma_3)$ link 5h>
 <Unproject $(1 + \gamma_3)$ link 5f>
 <Walk through sublattice 25d>
 <Walk through the hypersurface 28a>
 <Write fermion 20b>
 <dwf.c 64e>
 <sse.h 60f>