Conjugate Gradient for Domain Wall Fermions with 4-d EO preconditioning Version 1.1.1

Andrew Pochinsky

August 5, 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

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 apptication.

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 scarsity 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 relatice 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 hierarhy 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

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

where

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

and

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

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 ⟨*Build* $(1 + \gamma_0)$ *projection of* `*f` *in* `*g` 4a⟩≡
```
g->f[0][c].re = f->f[0][c].re - f->f[3][c].im;
g->f[0][c].im = f->f[0][c].im + f->f[3][c].re;
g->f[1][c].re = f->f[1][c].re - f->f[2][c].im;
g->f[1][c].im = f->f[1][c].im + f->f[2][c].re;
```

4b ⟨*Unproject and accumulate* $(1 + \gamma_0)$ *link* 4b⟩≡
```
qs->f[0][c].re += hh[k].f[0][c].re; qs->f[3][c].im -= hh[k].f[0][c].re;
qs->f[0][c].im += hh[k].f[0][c].im; qs->f[3][c].re += hh[k].f[0][c].im;
qs->f[1][c].re += hh[k].f[1][c].re; qs->f[2][c].im -= hh[k].f[1][c].re;
qs->f[1][c].im += hh[k].f[1][c].im; qs->f[2][c].re += hh[k].f[1][c].im;
```

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

4c ⟨*Build* $(1 - \gamma_0)$ *projection of* `*f` *in* `*g` 4c⟩≡
```
g->f[0][c].re = f->f[0][c].re + f->f[3][c].im;
g->f[0][c].im = f->f[0][c].im - f->f[3][c].re;
g->f[1][c].re = f->f[1][c].re + f->f[2][c].im;
g->f[1][c].im = f->f[1][c].im - f->f[2][c].re;
```

4d ⟨*Unproject and accumulate* $(1 - \gamma_0)$ *link* 4d⟩≡
```
qs->f[0][c].re += hh[k].f[0][c].re; qs->f[3][c].im += hh[k].f[0][c].re;
qs->f[0][c].im += hh[k].f[0][c].im; qs->f[3][c].re -= hh[k].f[0][c].im;
qs->f[1][c].re += hh[k].f[1][c].re; qs->f[2][c].im += hh[k].f[1][c].re;
qs->f[1][c].im += hh[k].f[1][c].im; qs->f[2][c].re -= hh[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 ⟨*Build* $(1 + \gamma_1)$ *projection of* `*f` *in* `*g` 4e⟩≡
```
g->f[0][c].re = f->f[0][c].re - f->f[3][c].re;
g->f[0][c].im = f->f[0][c].im - f->f[3][c].im;
g->f[1][c].re = f->f[1][c].re + f->f[2][c].re;
g->f[1][c].im = f->f[1][c].im + f->f[2][c].im;
```

4f ⟨*Unproject and accumulate* $(1 + \gamma_1)$ *link* 4f⟩≡
```
qs->f[0][c].re += hh[k].f[0][c].re; qs->f[3][c].re -= hh[k].f[0][c].re;
qs->f[0][c].im += hh[k].f[0][c].im; qs->f[3][c].im -= hh[k].f[0][c].im;
qs->f[1][c].re += hh[k].f[1][c].re; qs->f[2][c].re += hh[k].f[1][c].re;
qs->f[1][c].im += hh[k].f[1][c].im; qs->f[2][c].im += hh[k].f[1][c].im;
```

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

4g ⟨*Build* $(1 - \gamma_1)$ *projection of* `*f` *in* `*g` 4g⟩≡
```
g->f[0][c].re = f->f[0][c].re + f->f[3][c].re;
g->f[0][c].im = f->f[0][c].im + f->f[3][c].im;
g->f[1][c].re = f->f[1][c].re - f->f[2][c].re;
g->f[1][c].im = f->f[1][c].im - f->f[2][c].im;
```

4h ⟨*Unproject and accumulate* $(1 - \gamma_1)$ *link* 4h⟩≡
```
qs->f[0][c].re += hh[k].f[0][c].re; qs->f[3][c].re += hh[k].f[0][c].re;
qs->f[0][c].im += hh[k].f[0][c].im; qs->f[3][c].im += hh[k].f[0][c].im;
qs->f[1][c].re += hh[k].f[1][c].re; qs->f[2][c].re -= hh[k].f[1][c].re;
qs->f[1][c].im += hh[k].f[1][c].im; qs->f[2][c].im -= hh[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 ⟨*Build* $(1+\gamma_2)$ *projection of* `*f` *in* `*g` 5a⟩≡
```
g->f[0][c].re = f->f[0][c].re - f->f[2][c].im;
g->f[0][c].im = f->f[0][c].im + f->f[2][c].re;
g->f[1][c].re = f->f[1][c].re + f->f[3][c].im;
g->f[1][c].im = f->f[1][c].im - f->f[3][c].re;
```

5b ⟨*Unproject and accumulate* $(1+\gamma_2)$ *link* 5b⟩≡
```
qs->f[0][c].re += hh[k].f[0][c].re; qs->f[2][c].im -= hh[k].f[0][c].re;
qs->f[0][c].im += hh[k].f[0][c].im; qs->f[2][c].re += hh[k].f[0][c].im;
qs->f[1][c].re += hh[k].f[1][c].re; qs->f[3][c].im += hh[k].f[1][c].re;
qs->f[1][c].im += hh[k].f[1][c].im; qs->f[3][c].re -= hh[k].f[1][c].im;
```

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

5c ⟨*Build* $(1-\gamma_2)$ *projection of* `*f` *in* `*g` 5c⟩≡
```
g->f[0][c].re = f->f[0][c].re + f->f[2][c].im;
g->f[0][c].im = f->f[0][c].im - f->f[2][c].re;
g->f[1][c].re = f->f[1][c].re - f->f[3][c].im;
g->f[1][c].im = f->f[1][c].im + f->f[3][c].re;
```

5d ⟨*Unproject and accumulate* $(1-\gamma_2)$ *link* 5d⟩≡
```
qs->f[0][c].re += hh[k].f[0][c].re; qs->f[2][c].im += hh[k].f[0][c].re;
qs->f[0][c].im += hh[k].f[0][c].im; qs->f[2][c].re -= hh[k].f[0][c].im;
qs->f[1][c].re += hh[k].f[1][c].re; qs->f[3][c].im -= hh[k].f[1][c].re;
qs->f[1][c].im += hh[k].f[1][c].im; qs->f[3][c].re += hh[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 ⟨*Build* $(1+\gamma_3)$ *projection of* `*f` *in* `*g` 5e⟩≡
```
g->f[0][c].re = f->f[0][c].re + f->f[2][c].re;
g->f[0][c].im = f->f[0][c].im + f->f[2][c].im;
g->f[1][c].re = f->f[1][c].re + f->f[3][c].re;
g->f[1][c].im = f->f[1][c].im + f->f[3][c].im;
```

5f ⟨*Unproject* $(1+\gamma_3)$ *link* 5f⟩≡
```
qs->f[0][c].re = hh[k].f[0][c].re; qs->f[2][c].re = hh[k].f[0][c].re;
qs->f[0][c].im = hh[k].f[0][c].im; qs->f[2][c].im = hh[k].f[0][c].im;
qs->f[1][c].re = hh[k].f[1][c].re; qs->f[3][c].re = hh[k].f[1][c].re;
qs->f[1][c].im = hh[k].f[1][c].im; qs->f[3][c].im = hh[k].f[1][c].im;
```

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

5g ⟨*Build* $(1-\gamma_3)$ *projection of* `*f` *in* `*g` 5g⟩≡
```
g->f[0][c].re = f->f[0][c].re - f->f[2][c].re;
g->f[0][c].im = f->f[0][c].im - f->f[2][c].im;
g->f[1][c].re = f->f[1][c].re - f->f[3][c].re;
g->f[1][c].im = f->f[1][c].im - f->f[3][c].im;
```

5h ⟨*Unproject and accumulate* $(1-\gamma_3)$ *link* 5h⟩≡
```
qs->f[0][c].re += hh[k].f[0][c].re; qs->f[2][c].re -= hh[k].f[0][c].re;
qs->f[0][c].im += hh[k].f[0][c].im; qs->f[2][c].im -= hh[k].f[0][c].im;
qs->f[1][c].re += hh[k].f[1][c].re; qs->f[3][c].re -= hh[k].f[1][c].re;
qs->f[1][c].im += hh[k].f[1][c].im; qs->f[3][c].im -= hh[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: ϵ, required precision
Output: x, approximate solution
Output: ρ, 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, ρ, 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 expence 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: ϵ, required precision
Output: x, approximate solution
Output: ρ, final residue
Output: k, number of iterations used
begin
  $x \leftarrow x_0$
  $p \leftarrow r \leftarrow b - M^\dagger M x$
  $\rho \leftarrow \langle r, r \rangle$
  $k \leftarrow 0$
  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, ρ, 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

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

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 $P_\pm$ comute with A and B, $Q_{ee}^{-1} = P_+A^{-1} + P_-B^{-1}$. Computing A^{-1} and B^{-1} is done by decomposition $A = L_AR_A$, $B = L_BR_B$, where

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

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 inited_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 (*deallocator)(void *))
{
    if (inited_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 33g⟩
    ⟨Initialize QMP 30c⟩
    inited_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 (deallocator)
    tfree = deallocator;
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 ⟨*Check floating point size* 11a⟩≡
```
if (fp_size != SSE_DWF_FLOAT)
    goto error;
```

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

11b ⟨*Check lattice size* 11b⟩≡
```
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 ⟨*Check lattice size* 11b⟩+≡
```
{
    int i;
    for (i = 0; i < DIM; i++) {
        if (lattice[i] & 1)
            goto error;
        tlattice[i] = lattice[i];
    }
}
```

11d ⟨*Global variables* 10a⟩+≡
```
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 ⟨*Interface functions* 10b⟩+≡
```
void
SSE_DWF_fini(void)
{
    ⟨Cleanup QMP 32g⟩
    ⟨Free fields 33h⟩
    ⟨Free tables 28d⟩
    inited_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 ⟨*Data types* 11f⟩≡
```
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 (!inited_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 M, double m0,
                  const SSE_DWF_Fermion *x0,     /* guess */
                  const SSE_DWF_Fermion *eta,    /* rhs */
                  double eps, int max_iter)
{
    int status;

    if (!inited_p)
        return 1;

    U = (SU3 *)gauge;
    ⟨Compute constant values for $Q_{xx}^{-1}$ and $S_{xx}^{-1}$ 62b⟩
    ⟨Compute $\varphi_o$ 33e⟩
    ⟨Solve $M^\dagger M\psi_o = \varphi_o$ 34a⟩
    ⟨Compute $\psi_e$ 35b⟩
    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 paranoidal 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 = to_Ulinear(x, &bounds, -1);
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 = to_Ulinear(x, &bounds, 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 = S_4 * to_HFlinear(x, &bounds, -1, 0); /* p is taken care of! */
    vFermion *f = p? &ptr->odd[p1].f: &ptr->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++) {
                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 = S_4 * to_HFlinear(x, &bounds, -1, 0); /* p is taken care of! */
    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 environement:

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 */
    void            *qmp_xbuf[4*DIM]; /* QMP snd/rcv buffer addresses */
    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 msegs */

    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 neighboors 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->site++;
  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 opprotunity 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⟩
}
```

Constucting 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++;
```

Here we do the reverse, namely, free all memory allocated by `init_tables()`:

28d ⟨*Free tables* 28d⟩≡

```
{
    int i;

    if (neighbor.site) {
        tfree(neighbor.site);
        neighbor.site = 0;
    }

    if (neighbor.inside) {
       tfree(neighbor.inside);
       neighbor.inside = 0;
    }

    if (neighbor.boundary) {
       tfree(neighbor.boundary);
       neighbor.boundary = 0;
    }

    for (i = 2 * DIM; i--;) {
        if (neighbor.snd[i] == 0)
            continue;
        tfree(neighbor.snd[i]);
        neighbor.snd[i] = 0;
    }
}
```

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.

29a ⟨*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.

29b ⟨*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 29c⟩
    } else {
        ⟨Find index of a borrowed gauge link 30a⟩
    }
}
```

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

29c ⟨*Find index of a regular gauge link* 29c⟩≡

```
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:

30a ⟨*Find index of a borrowed gauge link* 30a⟩≡
```
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

30b ⟨*Include files* 30b⟩≡
```
#include <qmp.h>
```

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

30c ⟨*Initialize QMP* 30c⟩≡
```
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.

30d ⟨*Static function prototypes* 16b⟩+≡
```
static int build_buffers(struct neighbor *nb);
```

30e ⟨*Static functions* 15a⟩+≡
```
static int
build_buffers(struct neighbor *nb)
{
    int i, k, Nr;
    QMP_msghandle_t Rh[2*DIM];

    Nr = nb->Ns = nb->Nx = 0;
    for (i = 0; i < DIM; i++) {
        switch (network[i]) {
            case 1: break;
            case 2:
                ⟨Clump up and down directions 31a⟩
                break;
            default:
                /* Order here is important */
                ⟨Allocate down buffers 31c⟩
                ⟨Allocate up buffers 31b⟩
                break;
        }
    }
    ⟨Construct the collective handle 32d⟩
    return 0;
}
```

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

31a ⟨*Clump up and down directions* 31a⟩≡
```
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:

31b ⟨*Allocate up buffers* 31b⟩≡
```
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);
```

31c ⟨*Allocate down buffers* 31c⟩≡
```
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.

31d ⟨*Static function prototypes* 16b⟩+≡
```
static int make_buffer(struct neighbor *nb, int size);
```

31e ⟨*Static functions* 15a⟩+≡
```
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;
    sse_aligned_buffer(nb, N, 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`.

31f ⟨*Static function prototypes* 16b⟩+≡
```
static void make_send(struct neighbor *nb, int k, int i, int d);
```

32a ⟨*Static functions* 15a⟩+≡
```
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`.

32b ⟨*Static function prototypes* 16b⟩+≡
```
static int make_receive(struct neighbor *nb, int k, int i, int d,
                        QMP_msghandle_t Rh[2*DIM], int Nr);
```

32c ⟨*Static functions* 15a⟩+≡
```
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:

32d ⟨*Construct the collective handle* 32d⟩≡
```
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 permissable.

32e ⟨*Static function prototypes* 16b⟩+≡
```
static void sse_aligned_buffer(struct neighbor *nb, int k, int size);
```

32f ⟨*Static functions* 15a⟩+≡
```
static void
sse_aligned_buffer(struct neighbor *nb, int k, int size)
{
  int xcount = size + 15;
  char *ptr = QMP_allocate_aligned_memory(xcount);

  nb->qmp_buf[k] = (void *)(~15 & (15 + (unsigned long)(ptr)));
  nb->qmp_xbuf[k] = ptr;
}
```

Freeing QMP structure does the reverse of the allocator:

32g ⟨*Cleanup QMP* 32g⟩≡
```
free_buffers(&even_odd);
free_buffers(&odd_even);
```

There are some unsettling omissions in the QMP specification. What follows is based on the tribal wisdom which was not codified.

32h ⟨*Static function prototypes* 16b⟩+≡
```
static void free_buffers(struct neighbor *nb);
```

33a ⟨*Static functions* 15a⟩+≡
```
static void
free_buffers(struct neighbor *nb)
{
   int i;

   ⟨Free common receive handle 33b⟩
   ⟨Free send handles 33c⟩
   ⟨Free QMP buffers 33d⟩
}
```

Here we assume that `QMP_free_msghandle()` knows what to do with a bad handle returned from `QMP_declare_send...()` and `QMP_declare_receive...()`. The first common wisdom is that `QMP_declare_multiple()` invalidates individual handles. We only need to free one handle in `nb->qmp_cr`:

33b ⟨*Free common receive handle* 33b⟩≡
```
QMP_free_msghandle(nb->qmp_cr);
```

There is no need to walk through the dimension again: we conveniently packed all send handles into an array:

33c ⟨*Free send handles* 33c⟩≡
```
for (i = nb->Ns; i--;)
     QMP_free_msghandle(nb->qmp_sv[i]);
```

Two steps are needed to deallocate QMP memory:

33d ⟨*Free QMP buffers* 33d⟩≡
```
for (i = nb->Nx; i--;) {
    QMP_free_msgmem(nb->qmp_mm[i]);
    QMP_free_aligned_memory(nb->qmp_xbuf[i]);
}
```

5.7 Parts of the Solver

Here are three principal parts of the solver. Firts, 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.

33e ⟨*Compute* φ_o 33e⟩≡
```
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);
```

33f ⟨*Global variables* 10a⟩+≡
```
static vOddFermion *auxA_o, *auxB_o, *Phi_o;
static vEvenFermion *auxA_e;
```

33g ⟨*Allocate fields* 33g⟩≡
```
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;
```

33h ⟨*Free fields* 33h⟩≡
```
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:

34a ⟨*Solve* $M^\dagger M\psi_o = \varphi_o$ 34a⟩≡

```
status = cg(psi->odd, Phi_o, x0->odd, eps, max_iter, out_eps, out_iter);
```

34b ⟨*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);
```

34c ⟨*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 62c⟩

    for (k = 0; (rho > epsilon) && (k < N); k++) {
        compute_MxM(q_o, &norm_z, p_o);
        ⟨Finalize ⟨r, r⟩ computation 62c⟩
        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 62c⟩
        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 62g⟩
    *out_N = k;
    *out_eps = rho;

    return status;
}
```

Temporaries used by the CG

34d ⟨*Global variables* 10a⟩+≡

```
static vOddFermion *r_o, *p_o, *q_o;
```

34e ⟨*Allocate fields* 33g⟩+≡

```
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;
```

35a ⟨*Free fields* 33h⟩+≡
```
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:

35b ⟨*Compute* ψ_e 35b⟩≡
```
compute_Qeo(auxA_e, psi->odd);
compute_sum_e(auxB_e, eta->even, -1, auxA_e);
compute_Qee1(psi->even, auxB_e);
```

35c ⟨*Global variables* 10a⟩+≡
```
static vEvenFermion *auxB_e;
```

35d ⟨*Allocate fields* 33g⟩+≡
```
auxB_e = allocate_even_fermion();  if (auxB_e == 0) goto error;
```

35e ⟨*Free fields* 33h⟩+≡
```
if (auxB_e) free16(auxB_e); auxB_e = 0;
```

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.

35f ⟨*Static function prototypes* 16b⟩+≡
```
static void copy_o(vOddFermion *dst, const vOddFermion *src);
```

35g ⟨*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.

35h ⟨*Static function prototypes* 16b⟩+≡
```
static void compute_sum2_o(vOddFermion *dst, double alpha, const vOddFermion *src);
```

35i ⟨*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.

36a ⟨*Static function prototypes* 16b⟩+≡
```
static void compute_sum2x_o(vOddFermion *dst, const vOddFermion *src, double alpha);
```

36b ⟨*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 distict from the sources.
Do we really need separate functions for these?

36c ⟨*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);
```

36d ⟨*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++;
}
```

36e ⟨*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.

37a ⟨*Static function prototypes* 16b⟩+≡

```
static void compute_sum_oN(vOddFermion *d, double *norm,
                           const vOddFermion *x, double alpha, const vOddFermion *y);
```

37b ⟨*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 63a⟩
}
```

37c ⟨*Static function prototypes* 16b⟩+≡

```
static void compute_sum2_oN(vOddFermion *d, double *norm,
                            double alpha, const vOddFermion *y);
```

37d ⟨*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 63a⟩
}
```

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

Last three easy pieces.

38a ⟨*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);
```

38b ⟨*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.

38c ⟨*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.

38d ⟨*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.

38e ⟨*Static function prototypes* 16b⟩+≡
```
static void compute_Qxx1(vFermion *eta, const vFermion *psi, int xyzt);
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);
```

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

39a ⟨*Static functions* 15a⟩+≡

```
static void
compute_Qxx1(vFermion *chi, const vFermion *psi, int size)
{
    const vFermion *qs, *qx5;
    ⟨Q common locals 61b⟩
    ⟨Qxx locals 42c⟩

    for (i = 0; i < size; i++) {
        xyzt5 = i * S_4;
        ⟨Compute rx5 61f⟩
        ⟨Compute qx5 62a⟩
        ⟨Compute Q_xx^-1 part on the s-chain 40a⟩
    }
}
```

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

39b ⟨*Static functions* 15a⟩+≡

```
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;
    ⟨Q common locals 61b⟩
    ⟨Qxx locals 42c⟩

    for (i = 0; i < size; i++) {
        xyzt5 = i * S_4;
        ⟨Compute rx5 61f⟩
        ⟨Compute qx5 62a⟩
        ⟨Compute S_xx^-1 part on the s-chain 40b⟩
    }
}
```

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

Therefore,

40a ⟨*Compute* Q_{xx}^{-1} *part on the s-chain* 40a⟩≡
⟨*Compute* $A^{-1}\psi$ *on the upper two components* 40c⟩
⟨*Compute* $B^{-1}\psi$ *on the lower two components* 40f⟩

40b ⟨*Compute* S_{xx}^{-1} *part on the s-chain* 40b⟩≡
⟨*Compute* $A^{-1}\psi$ *on the lower two components* 40d⟩
⟨*Compute* $B^{-1}\psi$ *on the upper two components* 40e⟩

And

40c ⟨*Compute* $A^{-1}\psi$ *on the upper two components* 40c⟩≡
⟨*Compute* L_A^{-1} *on the upper components* 42d⟩
⟨*Compute* R_A^{-1} *on the upper components* 45a⟩

40d ⟨*Compute* $A^{-1}\psi$ *on the lower two components* 40d⟩≡
⟨*Compute* L_A^{-1} *on the lower components* 43b⟩
⟨*Compute* R_A^{-1} *on the lower components* 45b⟩

40e ⟨*Compute* $B^{-1}\psi$ *on the upper two components* 40e⟩≡
⟨*Compute* L_B^{-1} *on the upper components* 44a⟩
⟨*Compute* R_B^{-1} *on the upper components* 45c⟩

40f ⟨*Compute* $B^{-1}\psi$ *on the lower two components* 40f⟩≡
⟨*Compute* L_B^{-1} *on the lower components* 44b⟩
⟨*Compute* R_B^{-1} *on the lower components* 45d⟩

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

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

$$y_k^{(B)} = \begin{cases} \frac{1}{a} z_0, & \text{if } k = 0 \\ \frac{1}{a} z_k - \frac{b}{a} y_{k-1}^{(B)}, & \text{otherwise} \end{cases}$$

It turns out, that these computations are faster on the regular FP instructions than on SSE. For this reason corresponding parts for L_X^{-1} depend on the memory layout of `vReal`.
Let us compute constant pieces first. Division is slow, so we compute $1/a$ and $-b/a$ once and for all:

40g ⟨*Global variables* 10a⟩+≡
```
static REAL inv_a;
static REAL b_over_a;
```

40h ⟨*Compute values from* a, b *and* c 40h⟩≡
```
inv_a = 1.0 / a;
b_over_a = -b * inv_a;
```

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

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

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

We need to rewrite these expressions in a form suitable for SSE. Let us write

$$\begin{aligned} z_{n-1}^{(A)} &= z_a^{(A)} + z_b^{(A)} + z_c^{(A)} + z_d^{(A)} \\ z_0^{(B)} &= z_a^{(B)} + z_b^{(B)} + z_c^{(B)} + z_d^{(B)} \end{aligned}$$

where

$$\begin{aligned} z_a^{(A)} &= \sum_{j=0}^{n/4-1} \frac{-b^{4j} c/a^{4j+1}}{1+(-b)^{n-1}c/a^n} x_{4j} \\ z_b^{(A)} &= \sum_{j=0}^{n/4-1} \frac{b^{4j+1} c/a^{4j+2}}{1+(-b)^{n-1}c/a^n} x_{4j+1} \\ z_c^{(A)} &= \sum_{j=0}^{n/4-1} \frac{-b^{4j+2} c/a^{4j+3}}{1+(-b)^{n-1}c/a^n} x_{4j+2} \\ z_d^{(A)} &= \sum_{j=0}^{n/4-2} \frac{b^{4j+3} c/a^{4j+4}}{1+(-b)^{n-1}c/a^n} x_{4j+3} + \frac{1}{1+(-b)^{n-1}c/a^n} x_{n-1} \\ z_a^{(B)} &= \frac{1}{1+(-b)^{n-1}c/a^n} x_0 + \sum_{j=1}^{n/4-1} \frac{b^{n-1-4j} c/a^{n-4j}}{1+(-b)^{n-1}c/a^n} x_{4j} \\ z_b^{(B)} &= \sum_{j=0}^{n/4-1} \frac{-b^{n-2-4j} c/a^{n-4j-1}}{1+(-b)^{n-1}c/a^n} x_{4j+1} \\ z_b^{(B)} &= \sum_{j=0}^{n/4-1} \frac{b^{n-3-4j} c/a^{n-4j-2}}{1+(-b)^{n-1}c/a^n} x_{4j+2} \\ z_b^{(B)} &= \sum_{j=0}^{n/4-1} \frac{-b^{n-4-4j} c/a^{n-4j-3}}{1+(-b)^{n-1}c/a^n} x_{4j+3} \end{aligned}$$

These sums could be effectively computed with SSE, because their structures match DWF memory layout.
Here are constants needed to compute $z^{(A)}$ and $z^{(B)}$:

41a ⟨*Global variables* 10a⟩+≡
```
static vReal vfx_A;
static vReal vfx_B;
static vReal vab;
static REAL c0;
```

41b ⟨*Compute values from a, b and c* 40h⟩+≡
```
c0 = 1./(1.-c/b*pow(b/a, S_4*4.));
vab = vmk1(pow(b/a, 4.));
vfx_A = vmk4(-c0*c/a, c0*c*b/(a*a), -c0*c*b*b/(a*a*a), c0*c*b*b*b/(a*a*a*a));
vfx_B = vmk4(c0*c*b*b*b/(a*a*a*a), -c0*c*b*b/(a*a*a), c0*c*b/(a*a), -c0*c/a);
```

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

41c ⟨*Include files* 30b⟩+≡
```
#include <math.h>
```

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.

Before spelling out the details, let us define a few handy macros:

42a ⟨*Check xx-aliasing of q* 42a⟩≡
```
#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
```

42b ⟨*End xx-aliasing of q* 42b⟩≡
```
#undef QSETUP
#undef Q2R
```

For completeness, here are definitions of variables used in the pieces bellow:

42c ⟨*Qxx locals* 42c⟩≡
```
vReal fx;
vHalfFermion zV;
vcomplex zn;
complex zX[2][3];
```

42d ⟨*Compute L_A^{-1} on the upper components* 42d⟩≡
```
vhfzero(&zV);
fx = vfx_A;
⟨Check xx-aliasing of q 42a⟩
for (s = 0; s < S_4_1; s++, fx = fx * vab) {
    rs = &rx5[s];
    QSETUP(s)
    ⟨Compute zV ← zV + fx ∗ qs^up 43a⟩
}
rs = &rx5[S_4_1];
QSETUP(S_4_1)
vput_3(&fx, c0);
⟨Compute zV ← zV + fx ∗ qs^up 43a⟩
for (c = 0; c < 3; c++) {
  ⟨Compute wall value in zX[c] 44c⟩

  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 42b⟩
```

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

43a ⟨*Compute* $zV \leftarrow zV + fx * qs^{up}$ 43a⟩≡

```
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.

43b ⟨*Compute* L_A^{-1} *on the lower components* 43b⟩≡

```
vhfzero(&zV);
fx = vfx_A;
⟨Check xx-aliasing of q 42a⟩
for (s = 0; s < S_4_1; s++, fx = fx * vab) {
    rs = &rx5[s];
    QSETUP(s)
    ⟨Compute zV ← zV + fx * qs^down 43c⟩
}
rs = &rx5[S_4_1];
QSETUP(S_4_1)
vput_3(&fx, c0);
⟨Compute zV ← zV + fx * qs^down 43c⟩
for (c = 0; c < 3; c++) {
  ⟨Compute wall value in zX[c] 44c⟩

  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 42b⟩
```

43c ⟨*Compute* $zV \leftarrow zV + fx * qs^{down}$ 43c⟩≡

```
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:

44a ⟨*Compute* L_B^{-1} *on the upper components* 44a⟩≡

```
vhfzero(&zV);
fx = vfx_B;
⟨Check xx-aliasing of q 42a⟩
for (s = S_4; --s; fx = fx * vab) {
    rs = &rx5[s];
    QSETUP(s)
    ⟨Compute zV ← zV + fx ∗ qs^up 43a⟩
}
rs = &rx5[0];
QSETUP(0)
vput_0(&fx, c0);
⟨Compute zV ← zV + fx ∗ qs^up 43a⟩
for (c = 0; c < 3; c++) {
  ⟨Compute wall value in zX[c] 44c⟩

  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;
}
⟨End xx-aliasing of q 42b⟩
```

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

44b ⟨*Compute* L_B^{-1} *on the lower components* 44b⟩≡

```
vhfzero(&zV);
fx = vfx_B;
⟨Check xx-aliasing of q 42a⟩
for (s = S_4; --s; fx = fx * vab) {
  rs = &rx5[s];
  QSETUP(s)
  ⟨Compute zV ← zV + fx ∗ qs^down 43c⟩
}
rs = &rx5[0];
QSETUP(0)
vput_0(&fx, c0);
⟨Compute zV ← zV + fx ∗ qs^down 43c⟩
for (c = 0; c < 3; c++) {
  ⟨Compute wall value in zX[c] 44c⟩

  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;
}
⟨End xx-aliasing of q 42b⟩
```

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

44c ⟨*Compute wall value in* zX[c] 44c⟩≡

```
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.

45a ⟨*Compute* R_A^{-1} *on the upper components* 45a⟩≡
```
⟨Init out of bound y 45f⟩
for (s = S_4; s--;) {
    rs = &rx5[s];
    for (c = 0; c < 3; c++) {
        ⟨Compute y^(A)_{k,[0]} 46a⟩
        ⟨Compute y^(A)_{k,[1]} 46b⟩
    }
}
```

45b ⟨*Compute* R_A^{-1} *on the lower components* 45b⟩≡
```
⟨Init out of bound y 45f⟩
for (s = S_4; s--;) {
    rs = &rx5[s];
    for (c = 0; c < 3; c++) {
        ⟨Compute y^(A)_{k,[2]} 46c⟩
        ⟨Compute y^(A)_{k,[3]} 47a⟩
    }
}
```

45c ⟨*Compute* R_B^{-1} *on the upper components* 45c⟩≡
```
⟨Init out of bound y 45f⟩
for (s = 0; s < S_4; s++) {
    rs = &rx5[s];
    for (c = 0; c < 3; c++) {
        ⟨Compute y^(B)_{k,[0]} 47b⟩
        ⟨Compute y^(B)_{k,[1]} 47c⟩
    }
}
```

45d ⟨*Compute* R_B^{-1} *on the lower components* 45d⟩≡
```
⟨Init out of bound y 45f⟩
for (s = 0; s < S_4; s++) {
    rs = &rx5[s];
    for (c = 0; c < 3; c++) {
        ⟨Compute y^(B)_{k,[2]} 48a⟩
        ⟨Compute y^(B)_{k,[3]} 48b⟩
    }
}
```

We do not handle boundary cases in a special way. Instead, the previos value of y is stored in `yOut`:

45e ⟨*Qxx locals* 42c⟩+≡
```
complex yOut[2][3];
```

45f ⟨*Init out of bound y* 45f⟩≡
```
yOut[0][0].re = yOut[0][0].im = 0;
yOut[0][1].re = yOut[0][1].im = 0;
yOut[0][2].re = yOut[0][2].im = 0;
yOut[1][0].re = yOut[1][0].im = 0;
yOut[1][1].re = yOut[1][1].im = 0;
yOut[1][2].re = yOut[1][2].im = 0;
```

Now, the magic of copy paste:

46a ⟨*Compute* $y^{(A)}_{k,[0]}$ 46a⟩≡

```
{
    REAL *rs0re = (REAL *)&rs->f[0][c].re;
    REAL *rs0im = (REAL *)&rs->f[0][c].im;

    rs0re[3] = inv_a * rs0re[3] + b_over_a * yOut[0][c].re;
    rs0re[2] = inv_a * rs0re[2] + b_over_a * rs0re[3];
    rs0re[1] = inv_a * rs0re[1] + b_over_a * rs0re[2];
    yOut[0][c].re = rs0re[0] = inv_a * rs0re[0] + b_over_a * rs0re[1];

    rs0im[3] = inv_a * rs0im[3] + b_over_a * yOut[0][c].im;
    rs0im[2] = inv_a * rs0im[2] + b_over_a * rs0im[3];
    rs0im[1] = inv_a * rs0im[1] + b_over_a * rs0im[2];
    yOut[0][c].im = rs0im[0] = inv_a * rs0im[0] + b_over_a * rs0im[1];
}
```

46b ⟨*Compute* $y^{(A)}_{k,[1]}$ 46b⟩≡

```
{
    REAL *rs1re = (REAL *)&rs->f[1][c].re;
    REAL *rs1im = (REAL *)&rs->f[1][c].im;

    rs1re[3] = inv_a * rs1re[3] + b_over_a * yOut[1][c].re;
    rs1re[2] = inv_a * rs1re[2] + b_over_a * rs1re[3];
    rs1re[1] = inv_a * rs1re[1] + b_over_a * rs1re[2];
    yOut[1][c].re = rs1re[0] = inv_a * rs1re[0] + b_over_a * rs1re[1];

    rs1im[3] = inv_a * rs1im[3] + b_over_a * yOut[1][c].im;
    rs1im[2] = inv_a * rs1im[2] + b_over_a * rs1im[3];
    rs1im[1] = inv_a * rs1im[1] + b_over_a * rs1im[2];
    yOut[1][c].im = rs1im[0] = inv_a * rs1im[0] + b_over_a * rs1im[1];
}
```

46c ⟨*Compute* $y^{(A)}_{k,[2]}$ 46c⟩≡

```
{
    REAL *rs2re = (REAL *)&rs->f[2][c].re;
    REAL *rs2im = (REAL *)&rs->f[2][c].im;

    rs2re[3] = inv_a * rs2re[3] + b_over_a * yOut[0][c].re;
    rs2re[2] = inv_a * rs2re[2] + b_over_a * rs2re[3];
    rs2re[1] = inv_a * rs2re[1] + b_over_a * rs2re[2];
    yOut[0][c].re = rs2re[0] = inv_a * rs2re[0] + b_over_a * rs2re[1];

    rs2im[3] = inv_a * rs2im[3] + b_over_a * yOut[0][c].im;
    rs2im[2] = inv_a * rs2im[2] + b_over_a * rs2im[3];
    rs2im[1] = inv_a * rs2im[1] + b_over_a * rs2im[2];
    yOut[0][c].im = rs2im[0] = inv_a * rs2im[0] + b_over_a * rs2im[1];
}
```

47a ⟨*Compute* $y^{(A)}_{k,[3]}$ 47a⟩≡

```
{
    REAL *rs3re = (REAL *)&rs->f[3][c].re;
    REAL *rs3im = (REAL *)&rs->f[3][c].im;

    rs3re[3] = inv_a * rs3re[3] + b_over_a * yOut[1][c].re;
    rs3re[2] = inv_a * rs3re[2] + b_over_a * rs3re[3];
    rs3re[1] = inv_a * rs3re[1] + b_over_a * rs3re[2];
    yOut[1][c].re = rs3re[0] = inv_a * rs3re[0] + b_over_a * rs3re[1];

    rs3im[3] = inv_a * rs3im[3] + b_over_a * yOut[1][c].im;
    rs3im[2] = inv_a * rs3im[2] + b_over_a * rs3im[3];
    rs3im[1] = inv_a * rs3im[1] + b_over_a * rs3im[2];
    yOut[1][c].im = rs3im[0] = inv_a * rs3im[0] + b_over_a * rs3im[1];
}
```

47b ⟨*Compute* $y^{(B)}_{k,[0]}$ 47b⟩≡

```
{
    REAL *rs0re = (REAL *)&rs->f[0][c].re;
    REAL *rs0im = (REAL *)&rs->f[0][c].im;

    rs0re[0] = inv_a * rs0re[0] + b_over_a * yOut[0][c].re;
    rs0re[1] = inv_a * rs0re[1] + b_over_a * rs0re[0];
    rs0re[2] = inv_a * rs0re[2] + b_over_a * rs0re[1];
    yOut[0][c].re = rs0re[3] = inv_a * rs0re[3] + b_over_a * rs0re[2];

    rs0im[0] = inv_a * rs0im[0] + b_over_a * yOut[0][c].im;
    rs0im[1] = inv_a * rs0im[1] + b_over_a * rs0im[0];
    rs0im[2] = inv_a * rs0im[2] + b_over_a * rs0im[1];
    yOut[0][c].im = rs0im[3] = inv_a * rs0im[3] + b_over_a * rs0im[2];
}
```

47c ⟨*Compute* $y^{(B)}_{k,[1]}$ 47c⟩≡

```
{
    REAL *rs1re = (REAL *)&rs->f[1][c].re;
    REAL *rs1im = (REAL *)&rs->f[1][c].im;

    rs1re[0] = inv_a * rs1re[0] + b_over_a * yOut[1][c].re;
    rs1re[1] = inv_a * rs1re[1] + b_over_a * rs1re[0];
    rs1re[2] = inv_a * rs1re[2] + b_over_a * rs1re[1];
    yOut[1][c].re = rs1re[3] = inv_a * rs1re[3] + b_over_a * rs1re[2];

    rs1im[0] = inv_a * rs1im[0] + b_over_a * yOut[1][c].im;
    rs1im[1] = inv_a * rs1im[1] + b_over_a * rs1im[0];
    rs1im[2] = inv_a * rs1im[2] + b_over_a * rs1im[1];
    yOut[1][c].im = rs1im[3] = inv_a * rs1im[3] + b_over_a * rs1im[2];
}
```

48a ⟨*Compute* $y^{(B)}_{k,[2]}$ 48a⟩≡

```
{
    REAL *rs2re = (REAL *)&rs->f[2][c].re;
    REAL *rs2im = (REAL *)&rs->f[2][c].im;

    rs2re[0] = inv_a * rs2re[0] + b_over_a * yOut[0][c].re;
    rs2re[1] = inv_a * rs2re[1] + b_over_a * rs2re[0];
    rs2re[2] = inv_a * rs2re[2] + b_over_a * rs2re[1];
    yOut[0][c].re = rs2re[3] = inv_a * rs2re[3] + b_over_a * rs2re[2];

    rs2im[0] = inv_a * rs2im[0] + b_over_a * yOut[0][c].im;
    rs2im[1] = inv_a * rs2im[1] + b_over_a * rs2im[0];
    rs2im[2] = inv_a * rs2im[2] + b_over_a * rs2im[1];
    yOut[0][c].im = rs2im[3] = inv_a * rs2im[3] + b_over_a * rs2im[2];
}
```

48b ⟨*Compute* $y^{(B)}_{k,[3]}$ 48b⟩≡

```
{
    REAL *rs3re = (REAL *)&rs->f[3][c].re;
    REAL *rs3im = (REAL *)&rs->f[3][c].im;

    rs3re[0] = inv_a * rs3re[0] + b_over_a * yOut[1][c].re;
    rs3re[1] = inv_a * rs3re[1] + b_over_a * rs3re[0];
    rs3re[2] = inv_a * rs3re[2] + b_over_a * rs3re[1];
    yOut[1][c].re = rs3re[3] = inv_a * rs3re[3] + b_over_a * rs3re[2];

    rs3im[0] = inv_a * rs3im[0] + b_over_a * yOut[1][c].im;
    rs3im[1] = inv_a * rs3im[1] + b_over_a * rs3im[0];
    rs3im[2] = inv_a * rs3im[2] + b_over_a * rs3im[1];
    yOut[1][c].im = rs3im[3] = inv_a * rs3im[3] + b_over_a * rs3im[2];
}
```

5.8.12 Standalone off-diagonal pieces

First, simple off-diagonal parts.

48c ⟨*Static function prototypes* 16b⟩+≡

```
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:

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

49a ⟨*Static functions* 15a⟩+≡

```
static void
compute_Qxy(vFermion *chi,
            const vFermion *psi,
            struct neighbor *nb)
{
    ⟨Q common locals 61b⟩
    ⟨Qxy locals 61c⟩

    ⟨Setup xy-aliasing of q 51b⟩
    ⟨Start off-diagonal receives 62d⟩
    ⟨Finish old off-diagonal sends 62g⟩
    ⟨Compute projections for Q send 51d⟩
    ⟨Compute inside part for Q_xy 53e⟩
    ⟨Finish off-diagonal receives 62e⟩
    ⟨Compute boundary part for Q_xy 53f⟩
    ⟨Finish xy-aliasing of q 51c⟩
}
```

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

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

49b ⟨*Static functions* 15a⟩+≡

```
static void
compute_1Sxy(vFermion *chi,
             const vFermion *eta,
             const vFermion *psi,
             struct neighbor *nb)
{
    ⟨Q common locals 61b⟩
    ⟨Qxy locals 61c⟩

    ⟨Setup xy-aliasing of q 51b⟩
    ⟨Start off-diagonal receives 62d⟩
    ⟨Finish old off-diagonal sends 62g⟩
    ⟨Compute projections for S send 52a⟩
    ⟨Compute inside part for 1 − S_xy 55e⟩
    ⟨Finish off-diagonal receives 62e⟩
    ⟨Compute boundary part for 1 − S_xy 56a⟩
    ⟨Finish xy-aliasing of q 51c⟩
}
```

Likewise,

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

50a ⟨*Static functions* 15a⟩+≡

```
static void
compute_Qxx1Qxy(vFermion *chi,
                const vFermion *psi,
                struct neighbor *nb)
{
```

⟨*Q common locals* 61b⟩
⟨*Qxy locals* 61c⟩
⟨*Qxx locals* 42c⟩

⟨*Setup xy-aliasing of q* 51b⟩
⟨*Start off-diagonal receives* 62d⟩
⟨*Finish old off-diagonal sends* 62g⟩
⟨*Compute projections for Q send* 51d⟩
⟨*Compute inside part for* $Q_{xx}^{-1} Q_{xy}$ 58a⟩
⟨*Finish off-diagonal receives* 62e⟩
⟨*Compute boundary part for* $Q_{xx}^{-1} Q_{xy}$ 58b⟩
⟨*Finish xy-aliasing of q* 51c⟩

```
}
```

and

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

50b ⟨*Static functions* 15a⟩+≡

```
static void
compute_Sxx1Sxy(vFermion *chi,
                const vFermion *psi,
                struct neighbor *nb)
{
```

⟨*Q common locals* 61b⟩
⟨*Qxy locals* 61c⟩
⟨*Qxx locals* 42c⟩

⟨*Setup xy-aliasing of q* 51b⟩
⟨*Start off-diagonal receives* 62d⟩
⟨*Finish old off-diagonal sends* 62g⟩
⟨*Compute projections for S send* 52a⟩
⟨*Compute inside part for* $S_{xx}^{-1} S_{xy}$ 58c⟩
⟨*Finish off-diagonal receives* 62e⟩
⟨*Compute boundary part for* $S_{xx}^{-1} S_{xy}$ 58d⟩
⟨*Finish xy-aliasing of q* 51c⟩

```
}
```

Finally,

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

51a ⟨*Static functions* 15a⟩+≡

```
static void
compute_1Qxx1Qxy(vFermion *chi,
                 double *norm,
                 const vFermion *eta,
                 const vFermion *psi,
                 struct neighbor *nb)
{
    ⟨Q common locals 61b⟩
    ⟨Qxy locals 61c⟩
    ⟨Qxx locals 42c⟩
    vReal vv;
    vReal nv;
    *norm = 0;

    ⟨Setup xy-aliasing of q 51b⟩
    ⟨Start off-diagonal receives 62d⟩
    ⟨Finish old off-diagonal sends 62g⟩
    ⟨Compute projections for Q send 51d⟩
    ⟨Compute inside part for 1 − Q_xx^{-1} Q_xy 59b⟩
    ⟨Finish off-diagonal receives 62e⟩
    ⟨Compute boundary part for 1 − Q_xx^{-1} Q_xy 59c⟩
    ⟨Start ⟨r, r⟩ computation 63a⟩
    ⟨Finish xy-aliasing of q 51c⟩
}
```

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

51b ⟨*Setup xy-aliasing of q* 51b⟩≡

```
#define qx5 rx5
#define qs rs
```

51c ⟨*Finish xy-aliasing of q* 51c⟩≡

```
#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.

51d ⟨*Compute projections for Q send* 51d⟩≡

```
{
   int k, i, s, c, *src;
   const vFermion *f;
   vHalfFermion *g;

   k = 0; ⟨Construct (1 + γ_0) send k-buffer 52b⟩
   k = 1; ⟨Construct (1 − γ_0) send k-buffer 52c⟩
   k = 2; ⟨Construct (1 + γ_1) send k-buffer 52d⟩
   k = 3; ⟨Construct (1 − γ_1) send k-buffer 52e⟩
   k = 4; ⟨Construct (1 + γ_2) send k-buffer 53a⟩
   k = 5; ⟨Construct (1 − γ_2) send k-buffer 53b⟩
   k = 6; ⟨Construct (1 + γ_3) send k-buffer 53c⟩
   k = 7; ⟨Construct (1 − γ_3) send k-buffer 53d⟩
}
```

52a ⟨*Compute projections for S send* 52a⟩≡

```
{
   int k, i, s, c, *src;
   const vFermion *f;
   vHalfFermion *g;

   k = 0; ⟨Construct (1 − γ0) send k-buffer 52c⟩
   k = 1; ⟨Construct (1 + γ0) send k-buffer 52b⟩
   k = 2; ⟨Construct (1 − γ1) send k-buffer 52e⟩
   k = 3; ⟨Construct (1 + γ1) send k-buffer 52d⟩
   k = 4; ⟨Construct (1 − γ2) send k-buffer 53b⟩
   k = 5; ⟨Construct (1 + γ2) send k-buffer 53a⟩
   k = 6; ⟨Construct (1 − γ3) send k-buffer 53d⟩
   k = 7; ⟨Construct (1 + γ3) send k-buffer 53c⟩
}
```

52b ⟨*Construct* $(1+\gamma_0)$ *send k-buffer* 52b⟩≡

```
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 + γ0) projection of *f in *g 4a⟩
        }
    }
}
⟨Start k-send 62f⟩
```

52c ⟨*Construct* $(1-\gamma_0)$ *send k-buffer* 52c⟩≡

```
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 − γ0) projection of *f in *g 4c⟩
        }
    }
}
⟨Start k-send 62f⟩
```

52d ⟨*Construct* $(1+\gamma_1)$ *send k-buffer* 52d⟩≡

```
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 + γ1) projection of *f in *g 4e⟩
        }
    }
}
⟨Start k-send 62f⟩
```

52e ⟨*Construct* $(1-\gamma_1)$ *send k-buffer* 52e⟩≡

```
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 − γ1) projection of *f in *g 4g⟩
        }
    }
}
⟨Start k-send 62f⟩
```

53a ⟨*Construct* $(1+\gamma_2)$ *send k-buffer* 53a⟩≡
```
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 + γ2) projection of *f in *g 5a⟩
        }
    }
}
⟨Start k-send 62f⟩
```

53b ⟨*Construct* $(1-\gamma_2)$ *send k-buffer* 53b⟩≡
```
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 − γ2) projection of *f in *g 5c⟩
        }
    }
}
⟨Start k-send 62f⟩
```

53c ⟨*Construct* $(1+\gamma_3)$ *send k-buffer* 53c⟩≡
```
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 + γ3) projection of *f in *g 5e⟩
        }
    }
}
⟨Start k-send 62f⟩
```

53d ⟨*Construct* $(1-\gamma_3)$ *send k-buffer* 53d⟩≡
```
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 − γ3) projection of *f in *g 5g⟩
        }
    }
}
⟨Start k-send 62f⟩
```

5.8.15 Parts of $Q_{xy}\psi$

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

53e ⟨*Compute inside part for* Q_{xy} 53e⟩≡
```
for (i = 0; i < nb->inside_size; i++) {
    xyzt = nb->inside[i];
    xyzt5 = xyzt * S_4;
    ⟨Extract 1-d addresses 61e⟩
    ⟨Build SSE SU(3) objects 60b⟩
    ⟨Compute Qxy part on the inside s-chain 54a⟩
}
```

53f ⟨*Compute boundary part for* Q_{xy} 53f⟩≡
```
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 61e⟩
    ⟨Build SSE SU(3) objects 60b⟩
    ⟨Compute Qxy part on the boundary s-chain 54b⟩
}
```

54a ⟨*Compute Q_{xy} part on the inside s-chain* 54a⟩≡

```
for (s = 0; s < S_4; s++) {
    ⟨Compute Q inside γ-projections 54c⟩
    ⟨Inside multiply by Vs 55b⟩
    ⟨Compute Q γ-unprojections and sum the results 55a⟩
}
```

54b ⟨*Compute Q_{xy} part on the boundary s-chain* 54b⟩≡

```
for (s = 0; s < S_4; s++) {
  ⟨Compute Q boundary γ-projections 54d⟩
  ⟨Boundary multiply by Vs 55c⟩
  ⟨Compute Q γ-unprojections and sum the results 55a⟩
}
```

54c ⟨*Compute Q inside γ-projections* 54c⟩≡

```
⟨Construct neighbor pointers 60c⟩
for (c = 0; c < 3; c++) {
    k=0; f=&psi[ps[0]]; g=&gg[0]; ⟨Build (1+γ0) projection of *f in *g 4a⟩
    k=1; f=&psi[ps[1]]; g=&gg[1]; ⟨Build (1−γ0) projection of *f in *g 4c⟩
    k=2; f=&psi[ps[2]]; g=&gg[2]; ⟨Build (1+γ1) projection of *f in *g 4e⟩
    k=3; f=&psi[ps[3]]; g=&gg[3]; ⟨Build (1−γ1) projection of *f in *g 4g⟩
    k=4; f=&psi[ps[4]]; g=&gg[4]; ⟨Build (1+γ2) projection of *f in *g 5a⟩
    k=5; f=&psi[ps[5]]; g=&gg[5]; ⟨Build (1−γ2) projection of *f in *g 5c⟩
    k=6; f=&psi[ps[6]]; g=&gg[6]; ⟨Build (1+γ3) projection of *f in *g 5e⟩
    k=7; f=&psi[ps[7]]; g=&gg[7]; ⟨Build (1−γ3) projection of *f in *g 5g⟩
}
```

54d ⟨*Compute Q boundary γ-projections* 54d⟩≡

```
⟨Construct neighbor pointers 60c⟩
for (c = 0; c < 3; c++) {
    if ((m & 0x01) == 0) {
        k=0; f=&psi[ps[0]]; g=&gg[0]; ⟨Build (1+γ0) projection of *f in *g 4a⟩
    }
    if ((m & 0x02) == 0) {
        k=1; f=&psi[ps[1]]; g=&gg[1]; ⟨Build (1−γ0) projection of *f in *g 4c⟩
    }
    if ((m & 0x04) == 0) {
        k=2; f=&psi[ps[2]]; g=&gg[2]; ⟨Build (1+γ1) projection of *f in *g 4e⟩
    }
    if ((m & 0x08) == 0) {
        k=3; f=&psi[ps[3]]; g=&gg[3]; ⟨Build (1−γ1) projection of *f in *g 4g⟩
    }
    if ((m & 0x10) == 0) {
        k=4; f=&psi[ps[4]]; g=&gg[4]; ⟨Build (1+γ2) projection of *f in *g 5a⟩
    }
    if ((m & 0x20) == 0) {
        k=5; f=&psi[ps[5]]; g=&gg[5]; ⟨Build (1−γ2) projection of *f in *g 5c⟩
    }
    if ((m & 0x40) == 0) {
        k=6; f=&psi[ps[6]]; g=&gg[6]; ⟨Build (1+γ3) projection of *f in *g 5e⟩
    }
    if ((m & 0x80) == 0) {
        k=7; f=&psi[ps[7]]; g=&gg[7]; ⟨Build (1−γ3) projection of *f in *g 5g⟩
    }
}
```

55a ⟨*Compute Q γ-unprojections and sum the results* 55a⟩≡
```
rs = &rx5[s];
for (c = 0; c < 3; c++) {
    k = 6; ⟨Unproject (1 + γ3) link 5f⟩
    k = 7; ⟨Unproject and accumulate (1 − γ3) link 5h⟩
    k = 2; ⟨Unproject and accumulate (1 + γ1) link 4f⟩
    k = 3; ⟨Unproject and accumulate (1 − γ1) link 4h⟩
    k = 0; ⟨Unproject and accumulate (1 + γ0) link 4b⟩
    k = 1; ⟨Unproject and accumulate (1 − γ0) link 4d⟩
    k = 4; ⟨Unproject and accumulate (1 + γ2) link 5b⟩
    k = 5; ⟨Unproject and accumulate (1 − γ2) link 5d⟩
}
```

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

55b ⟨*Inside multiply by V s* 55b⟩≡
```
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 55d⟩
}
```

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

55c ⟨*Boundary multiply by V s* 55c⟩≡
```
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 55d⟩
}
```

55d ⟨*Multiply* `*u` *by* `*g` *and store the result in* `*h` 55d⟩≡
```
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 $\eta - S_{xy}\psi$

55e ⟨*Compute inside part for* $1 - S_{xy}$ 55e⟩≡
```
for (i = 0; i < nb->inside_size; i++) {
    const vFermion *ex5, *es;

    xyzt = nb->inside[i];
    xyzt5 = xyzt * S_4;
    ⟨Extract 1-d addresses 61e⟩
    ex5 = &eta[xyzt5];
    ⟨Build SSE SU(3) objects 60b⟩
    ⟨Compute 1 − S_xy part on the inside s-chain 56b⟩
}
```

56a ⟨*Compute boundary part for* $1 - S_{xy}$ 56a⟩≡

```
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;
    ⟨Extract 1-d addresses 61e⟩
    ex5 = &eta[xyzt5];
    ⟨Build SSE SU(3) objects 60b⟩
    ⟨Compute 1 − S_xy part on the boundary s-chain 56c⟩
}
```

56b ⟨*Compute* $1 - S_{xy}$ *part on the inside s-chain* 56b⟩≡

```
for (s = 0; s < S_4; s++) {
    ⟨Compute S inside γ-projections 56d⟩
    ⟨Inside multiply by Vs 55b⟩
    ⟨Compute 1 − S γ-unprojections and sum the results 57b⟩
}
```

56c ⟨*Compute* $1 - S_{xy}$ *part on the boundary s-chain* 56c⟩≡

```
for (s = 0; s < S_4; s++) {
  ⟨Compute S boundary γ-projections 57a⟩
  ⟨Boundary multiply by Vs 55c⟩
  ⟨Compute 1 − S γ-unprojections and sum the results 57b⟩
}
```

56d ⟨*Compute S inside* γ*-projections* 56d⟩≡

```
⟨Construct neighbor pointers 60c⟩
for (c = 0; c < 3; c++) {
    k=0; f=&psi[ps[0]]; g=&gg[0]; ⟨Build (1 − γ_0) projection of *f in *g 4c⟩
    k=1; f=&psi[ps[1]]; g=&gg[1]; ⟨Build (1 + γ_0) projection of *f in *g 4a⟩
    k=2; f=&psi[ps[2]]; g=&gg[2]; ⟨Build (1 − γ_1) projection of *f in *g 4g⟩
    k=3; f=&psi[ps[3]]; g=&gg[3]; ⟨Build (1 + γ_1) projection of *f in *g 4e⟩
    k=4; f=&psi[ps[4]]; g=&gg[4]; ⟨Build (1 − γ_2) projection of *f in *g 5c⟩
    k=5; f=&psi[ps[5]]; g=&gg[5]; ⟨Build (1 + γ_2) projection of *f in *g 5a⟩
    k=6; f=&psi[ps[6]]; g=&gg[6]; ⟨Build (1 − γ_3) projection of *f in *g 5g⟩
    k=7; f=&psi[ps[7]]; g=&gg[7]; ⟨Build (1 + γ_3) projection of *f in *g 5e⟩
}
```

57a ⟨*Compute S boundary γ-projections* 57a⟩≡

```
⟨Construct neighbor pointers 60c⟩
for (c = 0; c < 3; c++) {
    if ((m & 0x01) == 0) {
        k=0; f=&psi[ps[0]]; g=&gg[0]; ⟨Build (1 − γ0) projection of *f in *g 4c⟩
    }
    if ((m & 0x02) == 0) {
        k=1; f=&psi[ps[1]]; g=&gg[1]; ⟨Build (1 + γ0) projection of *f in *g 4a⟩
    }
    if ((m & 0x04) == 0) {
        k=2; f=&psi[ps[2]]; g=&gg[2]; ⟨Build (1 − γ1) projection of *f in *g 4g⟩
    }
    if ((m & 0x08) == 0) {
        k=3; f=&psi[ps[3]]; g=&gg[3]; ⟨Build (1 + γ1) projection of *f in *g 4e⟩
    }
    if ((m & 0x10) == 0) {
        k=4; f=&psi[ps[4]]; g=&gg[4]; ⟨Build (1 − γ2) projection of *f in *g 5c⟩
    }
    if ((m & 0x20) == 0) {
        k=5; f=&psi[ps[5]]; g=&gg[5]; ⟨Build (1 + γ2) projection of *f in *g 5a⟩
    }
    if ((m & 0x40) == 0) {
        k=6; f=&psi[ps[6]]; g=&gg[6]; ⟨Build (1 − γ3) projection of *f in *g 5g⟩
    }
    if ((m & 0x80) == 0) {
        k=7; f=&psi[ps[7]]; g=&gg[7]; ⟨Build (1 + γ3) projection of *f in *g 5e⟩
    }
}
```

57b ⟨*Compute $1 - S$ γ-unprojections and sum the results* 57b⟩≡

```
rs = &rx5[s];
es = &ex5[s];
for (c = 0; c < 3; c++) {
    k = 7; ⟨Unproject (1 + γ3) link 5f⟩
    k = 6; ⟨Unproject and accumulate (1 − γ3) link 5h⟩
    k = 3; ⟨Unproject and accumulate (1 + γ1) link 4f⟩
    k = 2; ⟨Unproject and accumulate (1 − γ1) link 4h⟩
    k = 0; ⟨Unproject and accumulate (1 − γ0) link 4d⟩
    k = 1; ⟨Unproject and accumulate (1 + γ0) link 4b⟩
    k = 4; ⟨Unproject and accumulate (1 − γ2) link 5d⟩
    k = 5; ⟨Unproject and accumulate (1 + γ2) link 5b⟩
    ⟨Compute (∗rs) ← η − (∗rs) for color c 57c⟩
}
```

57c ⟨*Compute* $(*rs) \leftarrow \eta - (*rs)$ *for color c* 57c⟩≡

```
rs->f[0][c].re = es->f[0][c].re - rs->f[0][c].re;
rs->f[0][c].im = es->f[0][c].im - rs->f[0][c].im;
rs->f[1][c].re = es->f[1][c].re - rs->f[1][c].re;
rs->f[1][c].im = es->f[1][c].im - rs->f[1][c].im;
rs->f[2][c].re = es->f[2][c].re - rs->f[2][c].re;
rs->f[2][c].im = es->f[2][c].im - rs->f[2][c].im;
rs->f[3][c].re = es->f[3][c].re - rs->f[3][c].re;
rs->f[3][c].im = es->f[3][c].im - rs->f[3][c].im;
```

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

58a ⟨*Compute inside part for* $Q_{xx}^{-1}Q_{xy}$ 58a⟩≡
```
for (i = 0; i < nb->inside_size; i++) {
    xyzt = nb->inside[i];
    xyzt5 = xyzt * S_4;
    ⟨Extract 1-d addresses 61e⟩
    ⟨Build SSE SU(3) objects 60b⟩
    ⟨Compute Q_xy part on the inside s-chain 54a⟩
    ⟨Compute Q_xx^-1 part on the s-chain 40a⟩
}
```

58b ⟨*Compute boundary part for* $Q_{xx}^{-1}Q_{xy}$ 58b⟩≡
```
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 61e⟩
    ⟨Build SSE SU(3) objects 60b⟩
    ⟨Compute Q_xy part on the boundary s-chain 54b⟩
    ⟨Compute Q_xx^-1 part on the s-chain 40a⟩
}
```

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

58c ⟨*Compute inside part for* $S_{xx}^{-1}S_{xy}$ 58c⟩≡
```
for (i = 0; i < nb->inside_size; i++) {
    xyzt = nb->inside[i];
    xyzt5 = xyzt * S_4;
    ⟨Extract 1-d addresses 61e⟩
    ⟨Build SSE SU(3) objects 60b⟩
    ⟨Compute S_xy part on the inside s-chain 58e⟩
    ⟨Compute S_xx^-1 part on the s-chain 40b⟩
}
```

58d ⟨*Compute boundary part for* $S_{xx}^{-1}S_{xy}$ 58d⟩≡
```
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 61e⟩
    ⟨Build SSE SU(3) objects 60b⟩
    ⟨Compute S_xy part on the boundary s-chain 58f⟩
    ⟨Compute S_xx^-1 part on the s-chain 40b⟩
}
```

58e ⟨*Compute* S_{xy} *part on the inside s-chain* 58e⟩≡
```
for (s = 0; s < S_4; s++) {
    ⟨Compute S inside γ-projections 56d⟩
    ⟨Inside multiply by Vs 55b⟩
    ⟨Compute S γ-unprojections and sum the results 59a⟩
}
```

58f ⟨*Compute* S_{xy} *part on the boundary s-chain* 58f⟩≡
```
for (s = 0; s < S_4; s++) {
  ⟨Compute S boundary γ-projections 57a⟩
  ⟨Boundary multiply by Vs 55c⟩
  ⟨Compute S γ-unprojections and sum the results 59a⟩
}
```

59a ⟨*Compute S γ-unprojections and sum the results* 59a⟩≡

```
rs = &rx5[s];
for (c = 0; c < 3; c++) {
    k = 7; ⟨Unproject (1 + γ3) link 5f⟩
    k = 6; ⟨Unproject and accumulate (1 − γ3) link 5h⟩
    k = 3; ⟨Unproject and accumulate (1 + γ1) link 4f⟩
    k = 2; ⟨Unproject and accumulate (1 − γ1) link 4h⟩
    k = 0; ⟨Unproject and accumulate (1 − γ0) link 4d⟩
    k = 1; ⟨Unproject and accumulate (1 + γ0) link 4b⟩
    k = 4; ⟨Unproject and accumulate (1 − γ2) link 5d⟩
    k = 5; ⟨Unproject and accumulate (1 + γ2) link 5b⟩
}
```

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

59b ⟨*Compute inside part for* $1 - Q_{xx}^{-1}Q_{xy}$ 59b⟩≡

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

    xyzt = nb->inside[i];
    xyzt5 = xyzt * S_4;
    ⟨Extract 1-d addresses 61e⟩
    ex5 = &eta[xyzt5];
    ⟨Build SSE SU(3) objects 60b⟩
    ⟨Compute Q_xy part on the inside s-chain 54a⟩
    ⟨Compute 1 − Q_xx^-1 part on the s-chain 59d⟩
}
```

59c ⟨*Compute boundary part for* $1 - Q_{xx}^{-1}Q_{xy}$ 59c⟩≡

```
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;
    ⟨Extract 1-d addresses 61e⟩
    ex5 = &eta[xyzt5];
    ⟨Build SSE SU(3) objects 60b⟩
    ⟨Compute Q_xy part on the boundary s-chain 54b⟩
    ⟨Compute 1 − Q_xx^-1 part on the s-chain 59d⟩
}
```

59d ⟨*Compute* $1 - Q_{xx}^{-1}$ *part on the s-chain* 59d⟩≡

```
⟨Compute Q_xx^-1 part on the s-chain 40a⟩
for (s = 0; s < S_4; s++) {
    rs = &rx5[s];
    es = &ex5[s];
    nv = vmk1(0.0);
    for (c = 0; c < 3; c++) {
        ⟨Compute (*rs) ← η − (*rs) and collect ⟨r, r⟩ 60a⟩
    }
    *norm += vsum(nv);
}
```

60a ⟨*Compute* $(*rs) \leftarrow \eta - (*rs)$ *and collect* $\langle r, r \rangle$ 60a⟩≡

```
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 Miscallenious

We also need to uplift the gauge fields

60b ⟨*Build SSE* $SU(3)$ *objects* 60b⟩≡

```
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:

60c ⟨*Construct neighbor pointers* 60c⟩≡

```
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}

61a ⟨*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.

61b ⟨*Q common locals* 61b⟩≡

```
int i, xyzt5, s, c;
vFermion * __restrict__ rx5, * __restrict__ rs;
```

Others are used only in Z_{xy} parts:

61c ⟨*Qxy locals* 61c⟩≡

```
int xyzt, k, d;
const vFermion *f;
vHalfFermion *g;
vHalfFermion gg[8], hh[8];
vSU3 V[8];
int ps[8], p5[8];
```

61d ⟨*Qxy locals* 61c⟩+≡

```
const SU3 *Uup, *Udown;
int c1, c2;
```

For the inside sites, compute the s-chain address of the neighbor. For the boundary sites, the address of the s-chain in the receive buffer is used instead:

61e ⟨*Extract 1-d addresses* 61e⟩≡

```
for (d = 0; d < 8; d++)
    p5[d] = nb->site[xyzt].F[d];
```

⟨*Compute* `rx5` 61f⟩

61f ⟨*Compute* `rx5` 61f⟩≡

```
rx5 = &chi[xyzt5];
```

62a ⟨*Compute* qx5 62a⟩≡
```
qx5 = &psi[xyzt5];
```

5.8.23 Common globals

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

62b ⟨*Compute constant values for* Q_{xx}^{-1} *and* S_{xx}^{-1} 62b⟩≡
```
{
    double a = M;
    double b = 2.;
    double c = -2*m0;

    ⟨Compute values from a, b and c 40h⟩
}
```

5.9 QMP Pieces

Here are miscalenious piece of QMP:
We are ready to use the result of the global sum. Check that it has been computed.

62c ⟨*Finalize* $\langle r, r \rangle$ *computation* 62c⟩≡
```
/* relax, QMP does not support split reductions yet. */
```

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

62d ⟨*Start off-diagonal receives* 62d⟩≡
```
QMP_start(nb->qmp_cr);
```

62e ⟨*Finish off-diagonal receives* 62e⟩≡
```
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.

62f ⟨*Start k-send* 62f⟩≡
```
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.

62g ⟨*Finish old off-diagonal sends* 62g⟩≡
```
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 corrsponding struct neighbor when a send operation is started, so that we do not wait on never started send.

62h ⟨*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.

63a ⟨*Start* $\langle r, r \rangle$ *computation* 63a⟩≡
```
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:

63b ⟨*Include files* 30b⟩+≡
```
#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`:

63c ⟨*sse.h* 63c⟩≡
```
#ifndef _SSE_H
#define _SSE_H
⟨SSE types 63d⟩
⟨SSE inline functions 64c⟩
#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.

63d ⟨*SSE types* 63d⟩≡
```
typedef REAL vReal __attribute__((mode(V4SF),aligned(16)));
```

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

63e ⟨*SSE types* 63d⟩+≡
```
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?):

63f ⟨*SSE types* 63d⟩+≡
```
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.)

64a ⟨*SSE types* 63d⟩+≡
```
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.

64b ⟨*SSE types* 63d⟩+≡
```
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.

64c ⟨*SSE inline functions* 64c⟩≡
```
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.

64d ⟨*SSE inline functions* 64c⟩+≡
```
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:

65a ⟨*SSE inline functions* 64c⟩+≡

```
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 0^{th} and the 3^{rd} elements of the vector:

65b ⟨*SSE inline functions* 64c⟩+≡

```
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:

$$\texttt{shift\_up1} \leftarrow (a_1, a_2, a_3, b_0)$$

65c ⟨*SSE inline functions* 64c⟩+≡

```
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;
}
```

$$\texttt{shift\_up2} \leftarrow (a_2, a_3, b_0, b_1)$$

65d ⟨*SSE inline functions* 64c⟩+≡

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

$$\texttt{shift\_up3} \leftarrow (a_3, b_0, b_1, b_2)$$

65e ⟨*SSE inline functions* 64c⟩+≡

```
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;
}
```

$$\texttt{shift\_down1} \leftarrow (a_3, b_0, b_1, b_2)$$

66a ⟨*SSE inline functions* 64c⟩+≡
```
static inline vReal shift_down1(vReal a, vReal b)
{
   return shift_up3(a, b);
}
```

$$\texttt{shift\_down2} \leftarrow (a_2, a_3, b_0, b_1)$$

66b ⟨*SSE inline functions* 64c⟩+≡
```
static inline vReal shift_down2(vReal a, vReal b)
{
   return shift_up2(a, b);
}
```

$$\texttt{shift\_down3} \leftarrow (a_1, a_2, a_3, b_0)$$

66c ⟨*SSE inline functions* 64c⟩+≡
```
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:

66d ⟨*SSE inline functions* 64c⟩+≡
```
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:

66e ⟨*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.

66f ⟨*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:

67 ⟨*dwf.c* 67⟩≡

```
#include <stdlib.h>
#include "sse-dwf-cg.h"
⟨Include files 30b⟩
⟨Macro definitions 15b⟩

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

6 CHUNKS

⟨ *Compute boundary part for* $S_{xx}^{-1} S_{xy}$ 58d ⟩
⟨ *Compute boundary part for* $1 - S_{xy}$ 56a ⟩
⟨ *Compute constant values for* Q_{xx}^{-1} *and* S_{xx}^{-1} 62b ⟩
⟨ *Compute init sizes* 22a ⟩
⟨ *Compute inside part for* $1 - Q_{xx}^{-1} Q_{xy}$ 59b ⟩
⟨ *Compute inside part for* $Q_{xx}^{-1} Q_{xy}$ 58a ⟩
⟨ *Compute inside part for* Q_{xy} 53e ⟩
⟨ *Compute inside part for* $S_{xx}^{-1} S_{xy}$ 58c ⟩
⟨ *Compute inside part for* $1 - S_{xy}$ 55e ⟩
⟨ *Compute* `inside_size` *and* `boundary_size` 23b ⟩
⟨ *Compute* `p` *and* `m` 25e ⟩
⟨ *Compute projections for* Q *send* 51d ⟩
⟨ *Compute projections for* S *send* 52a ⟩
⟨ *Compute* ψ_e 35b ⟩
⟨ *Compute* `qx5` 62a ⟩
⟨ *Compute* $(*rs) \leftarrow \eta - (*rs)$ *and collect* $\langle r, r \rangle$ 60a ⟩
⟨ *Compute* $(*rs) \leftarrow \eta - (*rs)$ *for color* c 57c ⟩
⟨ *Compute* `rx5` 61f ⟩
⟨ *Compute send sizes and allocate index tables* 23e ⟩
⟨ *Compute values from* a, b *and* c 40h ⟩
⟨ *Compute* φ_o 33e ⟩
⟨ *Compute wall value in* `zX[c]` 44c ⟩
⟨ *Compute* $y^{(A)}_{k,[0]}$ 46a ⟩
⟨ *Compute* $y^{(A)}_{k,[1]}$ 46b ⟩
⟨ *Compute* $y^{(A)}_{k,[2]}$ 46c ⟩
⟨ *Compute* $y^{(A)}_{k,[3]}$ 47a ⟩
⟨ *Compute* $y^{(B)}_{k,[0]}$ 47b ⟩
⟨ *Compute* $y^{(B)}_{k,[1]}$ 47c ⟩
⟨ *Compute* $y^{(B)}_{k,[2]}$ 48a ⟩
⟨ *Compute* $y^{(B)}_{k,[3]}$ 48b ⟩
⟨ *Compute* $zV \leftarrow zV + fx * qs^{down}$ 43c ⟩
⟨ *Compute* $zV \leftarrow zV + fx * qs^{up}$ 43a ⟩
⟨ *Construct* $(1 + \gamma_0)$ *send k-buffer* 52b ⟩
⟨ *Construct* $(1 + \gamma_1)$ *send k-buffer* 52d ⟩
⟨ *Construct* $(1 + \gamma_2)$ *send k-buffer* 53a ⟩
⟨ *Construct* $(1 + \gamma_3)$ *send k-buffer* 53c ⟩
⟨ *Construct* $(1 - \gamma_0)$ *send k-buffer* 52c ⟩
⟨ *Construct* $(1 - \gamma_1)$ *send k-buffer* 52e ⟩
⟨ *Construct* $(1 - \gamma_2)$ *send k-buffer* 53b ⟩
⟨ *Construct* $(1 - \gamma_3)$ *send k-buffer* 53d ⟩
⟨ *Construct neighbor pointers* 60c ⟩
⟨ *Construct the collective handle* 32d ⟩
⟨ *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* 42b ⟩
⟨ *Extract 1-d addresses* 61e ⟩
⟨ *Finalize* $\langle r, r \rangle$ *computation* 62c ⟩
⟨ *Find index of a borrowed gauge link* 30a ⟩
⟨ *Find index of a regular gauge link* 29c ⟩
⟨ *Finish off-diagonal receives* 62e ⟩
⟨ *Finish old off-diagonal sends* 62g ⟩
⟨ *Finish xy-aliasing of q* 51c ⟩
⟨ *Free QMP buffers* 33d ⟩
⟨ *Free common receive handle* 33b ⟩
⟨ *Free fields* 33h ⟩
⟨ *Free send handles* 33c ⟩

⟨*Free tables* 28d⟩
⟨*Get network topology* 17a⟩
⟨*Global variables* 10a⟩
⟨*Handle init errors* 10c⟩
⟨*Include files* 30b⟩
⟨*Init out of bound y* 45f⟩
⟨*Initialize QMP* 30c⟩
⟨*Initialize* `out` *and* `p` 25c⟩
⟨*Initialize tables* 21b⟩
⟨*Insert* `k` *into* `site[p].F[dx]` 28c⟩
⟨*Inside multiply by V s* 55b⟩
⟨*Interface functions* 10b⟩
⟨*Load* `DIM` *gauge links from U at* `x` 18a⟩
⟨*Load a* `d` *gauge link from V at* `x` 19b⟩
⟨*Load an s-line of fermion at* `x` 19d⟩
⟨*Load gauge boundary in direction* `d` 19a⟩
⟨*Macro definitions* 15b⟩
⟨*Multiply* `*u` *by* `*g` *and store the result in* `*h` 55d⟩
⟨*Q common locals* 61b⟩
⟨*Qxx locals* 42c⟩
⟨*Qxy locals* 61c⟩
⟨*Read fermion* 19c⟩
⟨*Read gauge field* 17c⟩
⟨*SSE inline functions* 64c⟩
⟨*SSE types* 63d⟩
⟨*Save an s-line of fermion at* `x` 20c⟩
⟨*Setup* `boundary` 26c⟩
⟨*Setup* `boundary` *or* `inside` 26a⟩
⟨*Setup heap management functions* 10d⟩
⟨*Setup* `inside` 26b⟩
⟨*Setup xy-aliasing of q* 51b⟩
⟨*Solve* $M^\dagger M\psi_o = \varphi_o$ 34a⟩
⟨*Start* `DIM`-*d sublattice scan* 18b⟩
⟨*Start* `DIM`-*d sublattice scan locally* 18c⟩
⟨*Start k-send* 62f⟩
⟨*Start* $\langle r, r\rangle$ *computation* 63a⟩
⟨*Start off-diagonal receives* 62d⟩
⟨*Static function prototypes* 16b⟩
⟨*Static functions* 15a⟩
⟨*Translate* `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* 67⟩
⟨*sse.h* 63c⟩