

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

Version 1.0.3

Andrew Pochinsky

March 31, 2004

Abstract

Here is a CG solver for 4-d preconditioned domain wall fermions (DWF). The program below uses SSE instruction on Pentium 4 (and recent AMD, including AMD64 architecture and Operon processors) processors to speedup the computation. The code relies on the QMP library for communication.

There are 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.

Contents

1	INTRODUCTION	1
1.1	Definitions	1
1.2	Optimization Strategy	1
2	ALGORITHM	3
3	DWF DIRAC OPERATOR	4
4	PRECONDITIONING	4
4.1	Even-odd parititoning	4
4.2	Q_{xx} inversion	5
4.3	γ matrices	5
5	THE SOLVER	8
5.1	QMP initialization and queries	8
5.2	Node initialization routines	10
5.2.1	Communication	11
5.2.2	Table initialization	13
5.2.3	Address translation routines	15
5.2.4	Table building	17
5.2.5	<code>struct neighbor</code> and related types	19
5.3	Lattice variable initialization	20
5.3.1	Actual allocators	20
5.4	Solver little helpers	21
5.4.1	Presolvers	21
5.4.2	Aftersolvers	21
5.5	Solver proper	22
5.6	Constants for Q_{xx}^{-1}	24

6	LATTICE OPERATIONS	24
6.1	copy_o(d, s) or $d \leftarrow s$	24
6.2	compute_sum2_o(d, alpha, s), or $d \leftarrow d + \alpha s$	24
6.3	compute_sum2x_o(d, s, alpha), or $d \leftarrow \alpha d + s$	25
6.4	compute_sum_x(d, x, alpha, y) or $q \leftarrow x + \alpha y$	25
6.5	Compute $d \leftarrow x + \alpha y$ and $r \leftarrow \langle d, d \rangle$	26
6.6	Compute $\eta \leftarrow M^\dagger M \psi$ and friends	27
6.7	Standalone diagonal pieces	28
6.7.1	Compute L_A^{-1} and L_B^{-1}	29
6.7.2	Compute R_A^{-1} and R_B^{-1}	32
6.7.3	Q_{xx}^{-1} and S_{xx}^{-1} on a single s -chain	35
6.8	Standalone off-diagonal pieces	36
6.9	Combined pieces	36
6.10	Common off-diagonal parts	37
6.11	Projections to be sent	39
6.12	Parts of $Q_{xy}\psi$	41
6.13	Parts of $\eta - S_{xy}\psi$	45
6.14	Parts of $Q_{xx}^{-1}Q_{xy}\psi$	47
6.15	Parts of $S_{xx}^{-1}S_{xy}\psi$	47
6.16	Parts of $\eta - Q_{xx}^{-1}Q_{xy}\psi$	48
7	SSE	49
7.1	SSE types	49
7.2	SSE functions	50
8	QMP sods and ends	53
8.1	Print routines	53
8.2	Communication pieces	53
8.3	Global sums	54
8.4	Memory allocation	54
9	TIMING	54
10	STABS	55
10.1	Initializing the gauge field	55
10.2	Initializing the right hand side	56

1 INTRODUCTION

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

1.1 Definitions

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

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

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

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

Site is a point on the lattice.

1.2 Optimization Strategy

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

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

2 ALGORITHM

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

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

Algorithm 1: Generic Conjugate Gradient Solver

Before we embark on developing a solver for DWF, let us apply a few simple transformations to Algorithm 1. First, our DWF operator is not hermitial, so we have to use $A = M^\dagger M$ in the solver. Secondly, it would be nice to postpone the use of the dot product in the loop to let slow network collect and broadcast the result. Fortunately, these two goals can be achieved simultaneously as shown in Algorithm 2.

Note, that our algorithm is organized slightly differently from the traditional form, e.g., see MathWorld or Numerical Recipes for details.

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
 $x \leftarrow x_0$
 $p \leftarrow r \leftarrow b - M^\dagger Mx$
 $\rho \leftarrow \langle r, r \rangle$
 $k \leftarrow 0$
while $\rho > \epsilon$ *or* $k < n$ **do**
 $z \leftarrow Mp$
 $q \leftarrow M^\dagger z$
 $\alpha \leftarrow \rho / \langle z, z \rangle$
 $r \leftarrow r - \alpha q$
 $x \leftarrow x + \alpha p$
 $\gamma \leftarrow \langle r, r \rangle$
if $\gamma < \epsilon$ **then**
 $\rho \leftarrow \gamma$
break
end
 $\beta \leftarrow \gamma / \rho$
 $\rho \leftarrow \gamma$
 $p \leftarrow r + \beta p$
 $k \leftarrow k + 1$
end
return x, ρ, k .
end

Algorithm 2: DWF-ready Gradient Solver

3 DWF DIRAC OPERATOR

In this section we spell out the form of the DWF Dirac Operator we are going to use as well as the 4-d even-odd preconditioning method by Kostas Orginos. The advantage of 4-d preconditioning compared to the naïve 5-d scheme is that it allows one to decrease restrictions on the lattice size, in particular, $L_s = 4k$ is easily implementable for 4-d preconditioning.

For the sake of completeness, here is the original operator:

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

where

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

and

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

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

Since our goal is a working code, we need to be specific about γ -matrices. To avoid repetition, the γ -matrix conventions are spelled out below in section 4.3. A clever choice there allows us to make some improvements to the critical part of the code.

4 PRECONDITIONING

4.1 Even-odd parititoning

If we write color lattice sites according to the parity of the sum of their 4-d coordinates, $x_0 + x_1 + x_2 + x_3$, 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 Q_{xx} does not depend on U . That, in turn, allows us to invert Q_{xx} easily. With this in mind, one writes:

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

Now, to solve the equation

$$D\psi = \eta,$$

one needs to perform the following steps:

1. Compute

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

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

3. Compute

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

4. Solve for ψ_o the following equation using Algorithm 2

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

5. Compute

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

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

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

4.2 Q_{xx} inversion

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

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

as follows:

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

Thus,

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

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

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

and

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

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

In these terms,

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

We will also need

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

For further reference,

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

This simplifies the above computations.

4.3 γ matrices

Our last concern is basis of Dirac γ -matrices. In principle, we might have used any valid basis. However, some of the operations are simplified with a clever choice. Here is our choice.

$$\gamma_0 = 1 \otimes \sigma_2 = \begin{pmatrix} 0 & -i & 0 & 0 \\ i & 0 & 0 & 0 \\ 0 & 0 & 0 & -i \\ 0 & 0 & i & 0 \end{pmatrix}$$

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

```

⟨Build (1 + γ0) projection of *f in *g⟩≡
  g->f[0][c].re = f->f[0][c].re + f->f[1][c].im;
  g->f[0][c].im = f->f[0][c].im - f->f[1][c].re;
  g->f[1][c].re = f->f[2][c].re + f->f[3][c].im;
  g->f[1][c].im = f->f[2][c].im - f->f[3][c].re;

⟨Unproject and accumulate (1 + γ0) link⟩≡
  qs->f[0][c].re += gg[k].f[0][c].re; qs->f[1][c].im += gg[k].f[0][c].re;
  qs->f[0][c].im += gg[k].f[0][c].im; qs->f[1][c].re -= gg[k].f[0][c].im;
  qs->f[2][c].re += gg[k].f[1][c].re; qs->f[3][c].im += gg[k].f[1][c].re;
  qs->f[2][c].im += gg[k].f[1][c].im; qs->f[3][c].re -= gg[k].f[1][c].im;

```

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

```

⟨Build (1 - γ0) projection of *f in *g⟩≡
  g->f[0][c].re = f->f[0][c].re - f->f[1][c].im;
  g->f[0][c].im = f->f[0][c].im + f->f[1][c].re;
  g->f[1][c].re = f->f[2][c].re - f->f[3][c].im;
  g->f[1][c].im = f->f[2][c].im + f->f[3][c].re;

⟨Unproject and accumulate (1 - γ0) link⟩≡
  qs->f[0][c].re += gg[k].f[0][c].re; qs->f[1][c].im -= gg[k].f[0][c].re;
  qs->f[0][c].im += gg[k].f[0][c].im; qs->f[1][c].re += gg[k].f[0][c].im;
  qs->f[2][c].re += gg[k].f[1][c].re; qs->f[3][c].im -= gg[k].f[1][c].re;
  qs->f[2][c].im += gg[k].f[1][c].im; qs->f[3][c].re += gg[k].f[1][c].im;

```

$$\gamma_1 = \sigma_1 \otimes \sigma_1 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

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

```

⟨Build (1 + γ1) projection of *f in *g⟩≡
  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;

⟨Unproject (1 + γ1) link⟩≡
  qs->f[2][c].re = qs->f[0][c].re = gg[k].f[0][c].re;
  qs->f[2][c].im = qs->f[0][c].im = gg[k].f[0][c].im;
  qs->f[3][c].re = qs->f[1][c].re = gg[k].f[1][c].re;
  qs->f[3][c].im = qs->f[1][c].im = gg[k].f[1][c].im;

```

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

```

⟨Build (1 - γ1) projection of *f in *g⟩≡
  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;

⟨Unproject and accumulate (1 - γ1) link⟩≡
  qs->f[0][c].re += gg[k].f[0][c].re; qs->f[2][c].re -= gg[k].f[0][c].re;
  qs->f[0][c].im += gg[k].f[0][c].im; qs->f[2][c].im -= gg[k].f[0][c].im;
  qs->f[1][c].re += gg[k].f[1][c].re; qs->f[3][c].re -= gg[k].f[1][c].re;
  qs->f[1][c].im += gg[k].f[1][c].im; qs->f[3][c].im -= gg[k].f[1][c].im;

```

$$\gamma_2 = \sigma_2 \otimes \sigma_1 = \begin{pmatrix} 0 & 0 & 0 & -i \\ 0 & 0 & -i & 0 \\ 0 & i & 0 & 0 \\ i & 0 & 0 & 0 \end{pmatrix}$$

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

$\langle \text{Build } (1 + \gamma_2) \text{ projection of } *f \text{ in } *g \rangle \equiv$

```
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;
```

$\langle \text{Unproject and accumulate } (1 + \gamma_2) \text{ link} \rangle \equiv$

```
qs->f[0][c].re += gg[k].f[0][c].re; qs->f[3][c].im += gg[k].f[0][c].re;
qs->f[0][c].im += gg[k].f[0][c].im; qs->f[3][c].re -= gg[k].f[0][c].im;
qs->f[1][c].re += gg[k].f[1][c].re; qs->f[2][c].im += gg[k].f[1][c].re;
qs->f[1][c].im += gg[k].f[1][c].im; qs->f[2][c].re -= gg[k].f[1][c].im;
```

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

$\langle \text{Build } (1 - \gamma_2) \text{ projection of } *f \text{ in } *g \rangle \equiv$

```
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;
```

$\langle \text{Unproject and accumulate } (1 - \gamma_2) \text{ link} \rangle \equiv$

```
qs->f[0][c].re += gg[k].f[0][c].re; qs->f[3][c].im -= gg[k].f[0][c].re;
qs->f[0][c].im += gg[k].f[0][c].im; qs->f[3][c].re += gg[k].f[0][c].im;
qs->f[1][c].re += gg[k].f[1][c].re; qs->f[2][c].im -= gg[k].f[1][c].re;
qs->f[1][c].im += gg[k].f[1][c].im; qs->f[2][c].re += gg[k].f[1][c].im;
```

$$\gamma_3 = \sigma_3 \otimes \sigma_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

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

$\langle \text{Build } (1 + \gamma_3) \text{ projection of } *f \text{ in } *g \rangle \equiv$

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

$\langle \text{Unproject and accumulate } (1 + \gamma_3) \text{ link} \rangle \equiv$

```
qs->f[0][c].re += gg[k].f[0][c].re; qs->f[1][c].re += gg[k].f[0][c].re;
qs->f[0][c].im += gg[k].f[0][c].im; qs->f[1][c].im += gg[k].f[0][c].im;
qs->f[2][c].re += gg[k].f[1][c].re; qs->f[3][c].re -= gg[k].f[1][c].re;
qs->f[2][c].im += gg[k].f[1][c].im; qs->f[3][c].im -= gg[k].f[1][c].im;
```

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

$\langle \text{Build } (1 - \gamma_3) \text{ projection of } *f \text{ in } *g \rangle \equiv$

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

$\langle \text{Unproject and accumulate } (1 - \gamma_3) \text{ link} \rangle \equiv$

```
qs->f[0][c].re += gg[k].f[0][c].re; qs->f[1][c].re -= gg[k].f[0][c].re;
qs->f[0][c].im += gg[k].f[0][c].im; qs->f[1][c].im -= gg[k].f[0][c].im;
qs->f[2][c].re += gg[k].f[1][c].re; qs->f[3][c].re += gg[k].f[1][c].re;
qs->f[2][c].im += gg[k].f[1][c].im; qs->f[3][c].im += gg[k].f[1][c].im;
```


5 THE SOLVER

Without further ado, here is the solver:

```
<main.c>≡
<Include files>
<Type definitions>
<Global Variables>
<Function prototypes>
<Functions>
int
main(int argc, char *argv[])
{
    int status = 1;
    int maxK;
    double M0 = 10.0; /* just to have some numbers */
    double mf = 0.01; /* just to have some numbers */
    <Local variables>

    <Initialize QMP>
    if (argc != 11) {
        print("usage: cg X Y Z T S Nx Ny Nz Nt Kstart\n");
        goto end;
    }
    maxK = atoi(argv[10]);
    <Initialize solver tables>
    <Compute constant values for  $Q_{xx}^{-1}$  and  $S_{xx}^{-1}$ >
    <Allocate and initialize gauge field>
    <Allocate and initialize rhs>
    <Allocate solution>
    <Allocate auxiliary lattice variables>

    /* XXX performance measurements */
    for (;;) {
        <Compute  $\varphi_o$ >
        <Solve  $M^\dagger M \psi_o = \varphi_o$ >
        if (r1.ru_utime.tv_sec - r0.ru_utime.tv_sec > 2)
            break;
        maxK += maxK;
    }
    <Compute  $\psi_e$ >

    <Save solution>
    <Report timing>
    // Free allocated space
    <Finalize QMP>

    return status;
}
```

5.1 QMP initialization and queries

We use QMP for communication substrate. When `main()` starts, it initializes QMP and asks it about network size and position of the node in it.

```
<Include files>≡
#include "qmp.h"
```

Let us assume optimistically that QMP is good at handling SMP machines. Then we do not need to worry about that case and can rely on the message library to do the Right Thing for us.

```

<Initialize QMP>≡
    if (QMP_init_msg_passing(&argc, &argv, QMP_SMP_MULTIPLE_ADDRESS) != QMP_SUCCESS) {
        fprintf(stderr, "QMP_init failed\n");
        return 1;
    }
    this_node = QMP_get_node_number();
    <Configure logical topology>

```

It is handy to have the current node number in a variable.

```

<Global Variables>≡
    QMP_u32_t this_node;

```

Once we are under QMP, we need to shut it down properly before exiting the application. This is where all errors should `goto` after reporting the calamity on node 0.

```

<Finalize QMP>≡
    end:
        QMP_finalize_msg_passing();

```

It appears that we can not rely on the execution environment providing us with logical topology, so we have to create it ourselves. Let us assume, however, that we can freely feed 4-d vector to `QMP_declare_logical_topology()` with an extend of one where needed; e.g., passing it `[1,8,1,8]` will create 2-d torus along *Y* and *T* directions and leave *X* and *Z* local to the node. If this is not the case, there is something wrong with the way I read QMP specification.

```

<Configure logical topology>≡
    for (i = 0; i < 4; i++) {
        lattice[i] = atoi(argv[i+1]);
        network[i] = atoi(argv[i+6]);
    }
    lattice[4] = atoi(argv[5]);

    if (!QMP_declare_logical_topology(network, 4)) {
        eprint("QMP cann't create lattice of [%d,%d,%d,%d,%d]\n",
            lattice[0], lattice[1], lattice[2], lattice[3], lattice[4]);
        goto end;
    }
    d = (int)QMP_get_logical_number_of_dimensions();
    cx = QMP_get_logical_coordinates();
    for (i = 0; i < d; i++)
        coord[i] = cx[i];
    for (i = d; i < 4; i++)
        coord[i] = 0;

```

```

<Global Variables>+≡
    int lattice[5];
    QMP_u32_t network[4];
    QMP_u32_t coord[4];

```

```

<Local variables>≡
    int i, d;
    const QMP_u32_t *cx;

```

```

<Include files>+≡
    #include <stdlib.h>

```

5.2 Node initialization routines

For each site on sublattice, we compute a table of neighbor information. We need to know where to get eight gauge fields and eight neighbor fermions on the other sublattice. For the fermion, the index of the $s = 0$ neighbor element is stored for inside sites and for boundary sites with `mask & (1<<d) == 0` (e.g., inside neighbors of the boundary elements.) Otherwise, the index of $s = 0$ element in the corresponding receive buffer is stored. As a result, all stored fermion indices are multiples of `S_4`. For gauge fields, we can avoid storing three indices by the following observation. All forward links for a given site are always needed for computation, so we can store only the index of the first of them, `Uup`, and arrange the gauge fields in memory for that other three are `Uup+1`, `Uup+2`, and `Uup+3` respectively. For the backward links, some of them will come from truncated sites (where the originating site belongs to another node on the network.) In this case, we only need one gauge matrix on the node, which we store *after* all local gauge fields. Such an arrangement works for inside sites as well, because we index only one of four gauge links on the originating site.

<Type definitions>≡

```
struct site {
    int Uup;          /* up-links are Uup, Uup+1, Uup+2, Uup+3 */
    int Udown[4];     /* four down-links */
    int F[8];         /* eight neighboring fermions on the other sublattice */
};
```

To speedup the main cg loop, we precompute offsets needed for computation of $M\psi$.

<Initialize solver tables>≡

```
if (cg_init()) {
    eprint("Can not initialize neighbor tables\n");
    goto end;
}
```

The DWF initializer creates all things necessary for communication and computation. Since we do not know the problem size until the runtime, memory is allocated here for index arrays.

<Functions>≡

```
int
cg_init(void)
{
    struct neighbor tmp;
    struct bounds bounds;
    int i, v;

    init_neighbor(&bounds, &neighbor);
    <Compute init sizes>
    tmp = neighbor;
    build_neighbor(&even_odd, &bounds, 0, &tmp);
    build_neighbor(&odd_even, &bounds, 1, &tmp);
    <Allocate and setup all send and receive buffers>

    return 0;
}
```

First, we set global data:

<Global Variables>+≡

```
static int gauge_XYZT;
static int S_4, S_4_1;
```

<Compute init sizes>≡

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

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

```

⟨Allocate and setup all send and receive buffers⟩≡
    if (build_buffers(&even_odd)) return 1;
    if (build_buffers(&odd_even)) return 1;

```

5.2.1 Communication

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

```

⟨Function prototypes⟩≡
    static int build_buffers(struct neighbor *nb);

```

```

⟨Functions⟩+≡
    static int
    build_buffers(struct neighbor *nb)
    {
        int i, k, Nr;
        QMP_msghandle_t Rh[8];

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

```

⟨Clump up and down directions⟩≡
    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:

```

⟨Allocate up buffers⟩≡
    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);

```

```

⟨Allocate down buffers⟩≡
    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 VecHalfFermion's fit for send and/or receive.

```

⟨Function prototypes⟩+≡
    static int make_buffer(struct neighbor *nb, int size);

⟨Functions⟩+≡
    static int
    make_buffer(struct neighbor *nb, int size)
    {
        int bcount = size * S_4 * sizeof (VecHalfFermion);
        int N = nb->Nx;

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

        return N;
    }

```

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

```

⟨Function prototypes⟩+≡
    static void make_send(struct neighbor *nb, int k, int i, int d);

⟨Functions⟩+≡
    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.

```

⟨Function prototypes⟩+≡
    static int make_receive(struct neighbor *nb, int k, int i, int d,
                           QMP_msghandle_t Rh[8], int Nr);

⟨Functions⟩+≡
    static int
    make_receive(struct neighbor *nb, int k, int i, int d,
                 QMP_msghandle_t Rh[8], int Nr)
    {
        Rh[Nr] = QMP_declare_receive_relative(nb->qmp_mm[k], i, d, 1);
        return Nr+1;
    }

```

Finally, aggregate all receive handles:

```

⟨Construct the collective handle⟩≡
    nb->qmp_cr = QMP_declare_multiple(Rh, Nr);

```

5.2.2 Table initialization

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

<Type definitions>+≡

```
struct bounds {
    int lo[4];
    int hi[4];
};
```

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.

<Global Variables>+≡

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

Let us start with computing the boundary of the sublattice

<Function prototypes>+≡

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

    for (i = 0; i < 4; i++) {
        bounds->lo[i] = lattice_start(lattice[i], network[i], coord[i]);
        bounds->hi[i] = lattice_start(lattice[i], network[i], coord[i] + 1);
    }
}
```

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

<Function prototypes>+≡

```
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>
    <Allocate inside table>
    <Allocate boundary table>
    <Compute send sizes and allocate index tables>
}
```

```

⟨Compute inside_size and boundary_size⟩≡
for (neighbor->size = 1, neighbor->inside_size = 1, i = 0; i < 4; 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 = calloc(neighbor->size, sizeof (struct site));

⟨Allocate inside table⟩≡
if (neighbor->inside_size)
    neighbor->inside = calloc(neighbor->inside_size, sizeof (int));
else
    neighbor->inside = 0;

⟨Allocate boundary table⟩≡
if (neighbor->boundary_size)
    neighbor->boundary = calloc(neighbor->boundary_size, sizeof (struct boundary));
else
    neighbor->boundary = 0;

⟨Compute send sizes and allocate index tables⟩≡
for (i = 0; i < 8; i++) {
    int d = i / 2;

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

```

5.2.3 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.

(Function prototypes)+≡

```
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 += lattice[d];
        if (v >= lattice[d])
            v -= lattice[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.

(Function prototypes)+≡

```
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
    } else {
        Find index of a borrowed gauge link
    }
}
```

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

(Find index of a regular gauge link)≡

```
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 += lattice[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:

⟨Find index of a borrowed gauge link⟩≡

```
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;
```

In addition to the above translations, we need a function that will walk through a boundary of a neighbor building the outside part of the `site[]`.F indices.

⟨Function prototypes⟩+≡

```
static void
construct_rec(struct neighbor *out,
              int parity,
              struct bounds *bounds,
              int dir,
              int step)
{
    struct bounds xb;
    int xc[4], p[4];
    int s, d, x, k;
    int dz = dir * 2 + ((step>0)?1:0);

    ⟨Construct the neighbor's network coordinates xc and bounds xb⟩
    ⟨Construct the initial point of the hypersurface⟩
    ⟨Walk through the hypersurface⟩
}
```

Constructing the neighbor's network position is straightforward:

⟨Construct the neighbor's network coordinates xc and bounds xb⟩≡

```
for (d = 0; d < 4; 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:

⟨Construct the initial point of the hypersurface⟩≡

```
for (d = 0; d < 4; d++)
    p[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.

```

⟨Walk through the hypersurface⟩≡
    k = 0;
    do {
        for (d = 0, s = parity; d < 4; d++)
            s += p[d];
        if (!(s & 1))
            goto next;

        ⟨Translate p to target x⟩
        ⟨Insert k into site[x].F[dx]⟩

    next:
        for (d = 0; d < 4; d++) {
            if (d == dir)
                continue;
            if (++p[d] == xb.hi[d])
                p[d] = xb.lo[d];
            else
                break;
        }
    } while (d != 4);
    out->rcv_size[dz^1] = k; /* XXX is it true? */

⟨Translate p to target x⟩≡
    x = to_HFlinear(p, bounds, dir, -step);

⟨Insert k into site[x].F[dx]⟩≡
    out->site[x].F[dz] = S_4 * k++;

```

5.2.4 Table building

Now we can define `build_neighbor()`:

```

⟨Function prototypes⟩+≡
    static void
    build_neighbor(struct neighbor *out,
                  struct bounds *bounds,
                  int parity,
                  struct neighbor *in)
    {
        int d, s, x, m;
        int p[4];

        ⟨Initialize out and p⟩
        ⟨Walk through sublattice⟩
        ⟨Build outside indices⟩
    }

```

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

```

⟨Initialize out and p⟩≡
    *out = *in;
    out->size = 0;
    out->inside_size = 0;
    out->boundary_size = 0;
    for (d = 0; d < 4; 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;
        p[d] = bounds->lo[d];
    }

```

Walking through the sublattice is done without nested loops. See Knuth if it looks suspicious to you.

⟨Walk through sublattice⟩≡

```
do {
    for (d = 0, s = parity; d < 4; d++)
        s += p[d];
    if ((s & 1) != 0)
        goto next;
    ⟨Compute x and m⟩
    ⟨Setup boundary or inside⟩
    ⟨Build local neighbors⟩
    out->size++;
    in->size++;
next:
    for (d = 0; d < 4; d++) {
        if (++p[d] == bounds->hi[d])
            p[d] = bounds->lo[d];
        else
            break;
    }
} while (d != 4);
```

For *x* we use a function to compute it from *p*. 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*.

⟨Compute x and m⟩≡

```
x = to_HFlinear(p, bounds, -1, 0);
for (m = 0, d = 0; d < 4; d++) {
    if (network[d] > 1) {
        if (p[d] == bounds->lo[d])
            m |= 1 << (2 * d);
        if (p[d] + 1 == bounds->hi[d])
            m |= 1 << (2 * d + 1);
    }
}
```

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

⟨Setup boundary or inside⟩≡

```
if (m) {
    ⟨Setup boundary⟩
} else {
    ⟨Setup inside⟩
}
```

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

⟨Setup inside⟩≡

```
*in->inside++ = x;
out->inside_size++;
```

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

⟨Setup boundary⟩≡

```
in->boundary->index = x;
in->boundary->mask = m;
in->boundary++;
out->boundary_size++;
for (d = 0; d < 8; d++) {
    if ((m & (1 << d)) == 0)
        continue;
    *in->snd[d]++ = x;
    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.

```

<Build local neighbors>≡
in->site->Uup = to_Ulinear(p, bounds, -1);
for (d = 0; d < 4; d++) {
    in->site->Udown[d] = to_Ulinear(p, bounds, d);
    if ((m & (1 << (2 * d))) == 0)
        in->site->F[2*d] = S_4 * to_HFlinear(p, bounds, d, -1);
    if ((m & (1 << (2 * d + 1))) == 0)
        in->site->F[2*d + 1] = S_4 * to_HFlinear(p, 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:

```

<Build outside indices>≡
for (d = 0; d < 4; d++) {
    if (network[d] < 2)
        continue;
    construct_rec(out, parity, bounds, d, +1);
    construct_rec(out, parity, bounds, d, -1);
}

```

5.2.5 struct neighbor and related types

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

```

<Type definitions>+≡
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[8];   /* size of send buffers in 8 dirs */
    int          rcv_size[8];   /* size of receive buffers */
    int          *snd[8];       /* 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 */
    VecHalfFermion *snd_buf[8]; /* Send buffers */
    VecHalfFermion *rcv_buf[8]; /* Receive buffers */

    int          qmp_size[16]; /* sizes of QMP buffers */
    VecHalfFermion *qmp_buf[16]; /* send and receive buffers for QMP */
    QMP_msgmem_t  qmp_mm[16]; /* msgmem's for send and receive */
    int          Nx;           /* number of msecs */

    QMP_msghandle_t qmp_sh[8]; /* handles for sends */
    QMP_msghandle_t qmp_sv[8]; /* copies of handles for finilization */
    int          qmp_smask; /* send flags for qmp_sh[] */
    int          Ns;        /* number of send handles */

    QMP_msghandle_t qmp_rh[8]; /* 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.

```

<Type definitions>+≡
struct boundary {
    int    index; /* x-index of this boundary site */
    int    mask;  /* bitmask of the borders */
};

```

All SSE-related stuff is in `sse.h`. We need to include it before using vector datatypes.

```
<Include files>+≡  
#include "sse.h"
```

5.3 Lattice variable initialization

Most lattice variables are used by SSE and need to be allocated at 16 bytes boundaries. We should use `posix_memalign()` instead of `malloc()` here.

Let us start with the solution vector.

```
<Allocate solution>≡  
psi_e = allocate_even_fermion();  
psi_o = allocate_odd_fermion();
```

```
<Local variables>+≡  
VecEvenFermion *psi_e;  
VecOddFermion *psi_o;
```

Allocating the right hand side is not different:

```
<Allocate and initialize rhs>≡  
eta_e = allocate_even_fermion();  
eta_o = allocate_odd_fermion();  
initialize_fermion(eta_e, eta_o);
```

```
<Local variables>+≡  
VecEvenFermion *eta_e;  
VecOddFermion *eta_o;
```

Gauge field is next. Unlike fermions, there is no point in dividing it into even and odd subfield.

```
<Allocate and initialize gauge field>≡  
U = allocate_gauge_field();  
initialize_gauge_field(U);
```

```
<Global Variables>+≡  
SU3 *U;
```

5.3.1 Actual allocators

First, the prototypes:

```
<Function prototypes>+≡  
VecEvenFermion *allocate_even_fermion(void);  
VecOddFermion *allocate_odd_fermion(void);  
SU3 *allocate_gauge_field(void);  
VecFermion *allocate_subfermion(int size);
```

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

```
<Functions>+≡  
VecEvenFermion *  
allocate_even_fermion(void)  
{  
    return allocate(even_odd.size * S_4 * sizeof (VecFermion));  
}  
  
VecOddFermion *  
allocate_odd_fermion(void)  
{  
    return allocate(odd_even.size * S_4 * sizeof (VecFermion));  
}  
  
SU3 *  
allocate_gauge_field(void)  
{  
    return allocate(gauge_XYZT * sizeof (SU3));  
}
```

Actual allocation is done here. It appears that linux lack `memalign()` and `posix_memalign()`, so we have to improvise. As a quick hack, let us ignore the problem of deallocating for the present. Then there is no need to keep the original heap pointer around.

```

<Function prototypes>+=
    void *allocate(int size);

<Functions>+=
    void *
    allocate(int size)
    {
        char *ptr = malloc(size + 15);
        intptr_t p = (15 + (intptr_t)ptr) & ~15;

        return (void *)p;
    }

<Include files>+=
    #include <stdint.h>

```

5.4 Solver little helpers

Preconditioned CG requires that the input is converted to proper form and, once the solution is obtained, the result is constructed from it according to preconditioning scheme. For even-odd preconditioning, the steps are well known.

5.4.1 Presolvers

```

<Compute  $\varphi_o$ >=
    Phi_o = allocate_odd_fermion();
    compute_Phi_o(Phi_o, eta_e, eta_o);

```

```

<Local variables>+=
    VecOddFermion *Phi_o;

```

Here is the actual computation (these are steps 1–3 in the outline above)

```

<Functions>+=
    void
    compute_Phi_o(VecOddFermion *Phi_o,
                  const VecEvenFermion *eta_e,
                  const VecOddFermion *eta_o)
    {
        compute_Qee1(auxA_e, eta_e);
        compute_Qoe(auxB_o, auxA_e);
        compute_sum_o(auxA_o, eta_o, -1, auxB_o);
        compute_Qoo1(auxB_o, auxA_o);
        compute_Mx(Phi_o, auxB_o);
    }

```

```

<Global Variables>+=
    VecOddFermion *auxA_o, *auxB_o;
    VecEvenFermion *auxA_e;

```

```

<Allocate auxiliary lattice variables>=
    auxA_e = allocate_even_fermion();
    auxA_o = allocate_odd_fermion();
    auxB_o = allocate_odd_fermion();

```

5.4.2 Aftersolvers

```

<Compute  $\psi_e$ >=
    compute_psi_e(psi_e, eta_e, psi_o);

```

```

⟨Functions⟩+≡
void
compute_psi_e(VecEvenFermion *psi_e,
               const VecEvenFermion *eta_e,
               const VecOddFermion *psi_o)
{
    compute_Qeo(auxA_e, psi_o);
    compute_sum_e(auxB_e, eta_e, -1, auxA_e);
    compute_Qee1(psi_e, auxB_e);
}

⟨Global Variables⟩+≡
VecEvenFermion *auxB_e;

⟨Allocate auxiliary lattice variables⟩+≡
auxB_e = allocate_even_fermion();

```

5.5 Solver proper

Once again, let us abstract the solver into a procedure.

```

⟨Solve  $M^\dagger M \psi_o = \varphi_o$ ⟩≡
cg(psi_o, Phi_o, Phi_o, 1e-10, maxK);

```

The solver itself mimics Algorithm 2 closely.

```

⟨Functions⟩+=
void
cg(VecOddFermion *x_o,
  const VecOddFermion *b,
  const VecOddFermion *x0,
  double epsilon,
  int N)
{
  double rho, alpha, beta, gamma, norm_z;
  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⟩
  ⟨Start timer⟩
  for (k = 0; (rho > epsilon) && (k < N); k++) {
    compute_MxM(q_o, &norm_z, p_o);
    ⟨Finalize ⟨r,r⟩ computation⟩
    ⟨Fake cg scalars for performance I⟩
    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⟩
    ⟨Fake cg scalars for performance II⟩
    if (gamma < epsilon) {
      rho = gamma;
      break;
    }
    beta = gamma / rho;
    rho = gamma;
    compute_sum2x_o(p_o, r_o, beta);
  }
  ⟨Finish old off-diagonal sends⟩
  ⟨Stop timer⟩
  print("cg: final rho=%g, k=%d\n", rho, k);
}

```

To simplify initialization code, all lattice variable logically belonging to `cg()` are declared globally and allocated in `main()`:

```

⟨Global Variables⟩+=
/* VecEvenFermion *z_e; */
VecOddFermion *r_o, *p_o, *q_o;

⟨Allocate auxiliary lattice variables⟩+=
/* z_e = allocate_even_fermion(); */
r_o = allocate_odd_fermion();
p_o = allocate_odd_fermion();
q_o = allocate_odd_fermion();

```


5.6 Constants for Q_{xx}^{-1}

Given M and mf, compute SSE constants needed for the diagonal piece.

```

<Compute constant values for  $Q_{xx}^{-1}$  and  $S_{xx}^{-1}$ >≡
{
    double a = M0;
    double b = 2.;
    double c = -2*mf;
    vreal ab = vmk1(-b/a);

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

```

For pow() we will need the math library:

```

<Include files>+≡
#include <math.h>

```

6 LATTICE OPERATIONS

Here all operations acting on (half) lattice are implemented.

6.1 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.

```

<Function prototypes>+≡
static void copy_o(VecOddFermion *dst, const VecOddFermion *src);

<Functions>+≡
static void
copy_o(VecOddFermion *dst,
      const VecOddFermion *src)
{
    int i = odd_even.size * S_4 * sizeof (VecOddFermion) / sizeof (vreal);
    vreal *d = (vreal *)dst;
    const vreal *s = (const vreal *)src;

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

```

6.2 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.

```

<Function prototypes>+≡
static void compute_sum2_o(VecOddFermion *dst, double alpha, const VecOddFermion *src);

```

```

<Functions>+=
static void
compute_sum2_o(VecOddFermion *dst,
               double alpha, const VecOddFermion *src)
{
    int i = odd_even.size * S_4 * sizeof (VecOddFermion) / sizeof (vreal);
    vreal a = vmk1(alpha);
    vreal *d = (vreal *)dst;
    const vreal *s = (const vreal *)src;

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

```

6.3 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.

```

<Function prototypes>+=
static void compute_sum2x_o(VecOddFermion *dst, const VecOddFermion *src, double alpha);

<Functions>+=
static void
compute_sum2x_o(VecOddFermion *dst,
               const VecOddFermion *src, double alpha)
{
    int i = odd_even.size * S_4 * sizeof (VecOddFermion) / sizeof (vreal);
    vreal a = vmk1(alpha);
    vreal *d = (vreal *)dst;
    const vreal *s = (const vreal *)src;

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

```

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

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

```

<Function prototypes>+=
static void compute_sum_e(VecEvenFermion *d,
                         const VecEvenFermion *x, double alpha, const VecEvenFermion *y);
static void compute_sum_o(VecOddFermion *d,
                         const VecOddFermion *x, double alpha, const VecOddFermion *y);

<Functions>+=
static void
compute_sum_e(VecEvenFermion *d,
             const VecEvenFermion *x, double alpha, const VecEvenFermion *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 (VecEvenFermion) / sizeof (vreal);

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

```

```

⟨Functions⟩+=
static void
compute_sum_o(VecOddFermion *d,
               const VecOddFermion *x, double alpha, const VecOddFermion *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 (VecOddFermion) / sizeof (vreal);

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

```

6.5 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.

```

⟨Function prototypes⟩+=
static void compute_sum_oN(VecOddFermion *d, double *norm,
                           const VecOddFermion *x, double alpha, const VecOddFermion *y);

⟨Functions⟩+=
static void
compute_sum_oN(VecOddFermion *d, double *norm,
               const VecOddFermion *x, double alpha, const VecOddFermion *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 (VecOddFermion) / sizeof (vreal);

    for (;i--;) {
        v = *X++ + a * *Y++;
        s += v * v;
        *D++ = v;
    }
    *norm = vsum(s);
    ⟨Start ⟨r,r⟩ computation⟩
}

```

```

⟨Function prototypes⟩+=
static void compute_sum2_oN(VecOddFermion *d, double *norm,
                           double alpha, const VecOddFermion *y);

```

```

⟨Functions⟩+=
static void
compute_sum2_oN(VecOddFermion *d, double *norm,
                double alpha, const VecOddFermion *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 (VecOddFermion) / sizeof (vreal);

    for (;i--;) {
        v = *D + a * *Y++;
        s += v * v;
        *D++ = v;
    }
    *norm = vsum(s);
    ⟨Start ⟨r,r⟩ computation⟩
}

```

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

Last three easy pieces.

```

⟨Function prototypes⟩+=
static void compute_MxM(VecOddFermion *eta, double *norm,
                        const VecOddFermion *psi);
static void compute_M(VecOddFermion *eta, double *norm,
                        const VecOddFermion *psi);
static void compute_Mx(VecOddFermion *eta,
                        const VecOddFermion *psi);

```

```

⟨Functions⟩+=
static void
compute_MxM(VecOddFermion *eta, double *norm,
             const VecOddFermion *psi)
{
    compute_M(auxB_o, norm, psi);
    compute_Mx(eta, auxB_o);
}

```

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

```

⟨Functions⟩+=
static void compute_M(VecOddFermion *eta, double *norm,
                      const VecOddFermion *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.

```

⟨Functions⟩+=
static void compute_Mx(VecOddFermion *eta,
                       const VecOddFermion *psi)
{
    compute_Soo1(auxA_o, psi);
    compute_See1Seo(auxA_e, auxA_o);
    compute_1Soe(eta, psi, auxA_e);
}

```

6.7 Standalone diagonal pieces

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

<Function prototypes>+≡

```
static void compute_Qxx1(VecFermion *eta, const VecFermion *psi, int xyzt);
static void inline compute_Qee1(VecEvenFermion *eta, const VecEvenFermion *psi)
{
    compute_Qxx1(&eta->f, &psi->f, even_odd.size);
}
static void inline compute_Qoo1(VecOddFermion *eta, const VecOddFermion *psi)
{
    compute_Qxx1(&eta->f, &psi->f, odd_even.size);
}
static void compute_Soo1(VecOddFermion *eta, const VecOddFermion *psi);
```

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

<Functions>+≡

```
static void
compute_Qxx1(VecFermion *chi, const VecFermion *psi, int size)
{
    const VecFermion *qs, *qx5;
    <Q common locals>
    <Qxx locals>

    for (i = 0; i < size; i++) {
        xyzt5 = i * S_4;
        <Compute rx5>
        <Compute qx5>
        <Compute Qxx-1 part on the s-chain>
    }
}
```

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

<Functions>+≡

```
static void
compute_Soo1(VecOddFermion *Chi, const VecOddFermion *Psi)
{
    VecFermion *chi = &Chi->f;
    const VecFermion *psi = &Psi->f;
    int size = odd_even.size;
    const VecFermion *qs, *qx5;
    <Q common locals>
    <Qxx locals>

    for (i = 0; i < size; i++) {
        xyzt5 = i * S_4;
        <Compute rx5>
        <Compute qx5>
        <Compute Sxx-1 part on the s-chain>
    }
}
```

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

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

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

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

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

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

[[NB: This is likely wrong with respect to values, by the data flow is correct.]]

6.7.1 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.

```

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

    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  $q$ ⟩

```

To avoid strange things gcc does when SSE data is declared local to a block, we place all such variables on the function level: This piece is used twice: once in the loop over L_s , and the second time after correcting s_3 :

```

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

```

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

```

⟨Compute wall value in  $zX[c]$ ⟩≡
  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);

```

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.

```

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

    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⟩

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

```

⟨Compute  $L_B^{-1}$  on the upper components⟩≡
  vhfzero(&zV);
  fx = ab_LB;
  ⟨Check xx-aliasing of q⟩
  for (s = S_4; --s; fx = fx * va4) {
    rs = &rx5[s];
    QSETUP(s)
    ⟨Compute  $zV \leftarrow zV + fx * qs^{up}$ ⟩
  }
  rs = &rx5[0];
  QSETUP(0)
  vput_0(&fx, c0);
  ⟨Compute  $zV \leftarrow zV + fx * qs^{up}$ ⟩
  for (c = 0; c < 3; c++) {
    ⟨Compute wall value in  $zX[c]$ ⟩

    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⟩

```

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

```

⟨Compute  $L_B^{-1}$  on the lower components⟩≡
  vhfzero(&zV);
  fx = ab_LB;
  ⟨Check xx-aliasing of q⟩
  for (s = S_4; --s; fx = fx * va4) {
    rs = &rx5[s];
    QSETUP(s)
    ⟨Compute  $zV \leftarrow zV + fx * qs^{down}$ ⟩
  }
  rs = &rx5[0];
  QSETUP(0)
  vput_0(&fx, c0);
  ⟨Compute  $zV \leftarrow zV + fx * qs^{down}$ ⟩
  for (c = 0; c < 3; c++) {
    ⟨Compute wall value in  $zX[c]$ ⟩

    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⟩

```


6.7.2 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.

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

```

     $\langle \text{Init out of bound } x \text{ and } y \rangle$ 
    for (s = S_4; s--;) {
        rs = &rx5[s];
        for (c = 0; c < 3; c++) {
             $\langle \text{Compute } y_{k,[0]}^{(A)} \rangle$ 
             $\langle \text{Compute } y_{k,[1]}^{(A)} \rangle$ 
        }
    }

```

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

```

     $\langle \text{Init out of bound } x \text{ and } y \rangle$ 
    for (s = S_4; s--;) {
        rs = &rx5[s];
        for (c = 0; c < 3; c++) {
             $\langle \text{Compute } y_{k,[2]}^{(A)} \rangle$ 
             $\langle \text{Compute } y_{k,[3]}^{(A)} \rangle$ 
        }
    }

```

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

```

     $\langle \text{Init out of bound } x \text{ and } y \rangle$ 
    for (s = 0; s < S_4; s++) {
        rs = &rx5[s];
        for (c = 0; c < 3; c++) {
             $\langle \text{Compute } y_{k,[0]}^{(B)} \rangle$ 
             $\langle \text{Compute } y_{k,[1]}^{(B)} \rangle$ 
        }
    }

```

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

```

     $\langle \text{Init out of bound } x \text{ and } y \rangle$ 
    for (s = 0; s < S_4; s++) {
        rs = &rx5[s];
        for (c = 0; c < 3; c++) {
             $\langle \text{Compute } y_{k,[2]}^{(B)} \rangle$ 
             $\langle \text{Compute } y_{k,[3]}^{(B)} \rangle$ 
        }
    }

```

The only piece we can share here is

$\langle \text{Init out of bound } x \text{ and } y \rangle \equiv$

```

    vhfzero(&xOut);
    vhfzero(&yOut);

```

Now, the magic of copy paste:

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

```
zn.re = rs->f[0][c].re;
zn.im = rs->f[0][c].im;
z1.re = shift_down1(zn.re, xOut.f[0][c].re);
z1.im = shift_down1(zn.im, xOut.f[0][c].im);
z2.re = shift_down2(zn.re, xOut.f[0][c].re);
z2.im = shift_down2(zn.im, xOut.f[0][c].im);
z3.re = shift_down3(zn.re, xOut.f[0][c].re);
z3.im = shift_down3(zn.im, xOut.f[0][c].im);
rs->f[0][c].re = va0*zn.re + va1*z1.re + va2*z2.re + va3*z3.re + va4*yOut.f[0][c].re;
rs->f[0][c].im = va0*zn.im + va1*z1.im + va2*z2.im + va3*z3.im + va4*yOut.f[0][c].im;
yOut.f[0][c].re = rs->f[0][c].re;
yOut.f[0][c].im = rs->f[0][c].im;
xOut.f[0][c].re = zn.re;
xOut.f[0][c].im = zn.im;
```

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

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

$\langle \text{Compute } y_{k,[2]}^{(A)} \rangle \equiv$

```
zn.re = rs->f[2][c].re;
zn.im = rs->f[2][c].im;
z1.re = shift_down1(zn.re, xOut.f[0][c].re);
z1.im = shift_down1(zn.im, xOut.f[0][c].im);
z2.re = shift_down2(zn.re, xOut.f[0][c].re);
z2.im = shift_down2(zn.im, xOut.f[0][c].im);
z3.re = shift_down3(zn.re, xOut.f[0][c].re);
z3.im = shift_down3(zn.im, xOut.f[0][c].im);
rs->f[2][c].re = va0*zn.re + va1*z1.re + va2*z2.re + va3*z3.re + va4*yOut.f[0][c].re;
rs->f[2][c].im = va0*zn.im + va1*z1.im + va2*z2.im + va3*z3.im + va4*yOut.f[0][c].im;
yOut.f[0][c].re = rs->f[2][c].re;
yOut.f[0][c].im = rs->f[2][c].im;
xOut.f[0][c].re = zn.re;
xOut.f[0][c].im = zn.im;
```

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

```

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

```

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

```

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

```

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

```

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

```

```

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

```

```

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

```

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

Therefore,

```

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

```

```

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

```

And

```

⟨Compute  $A^{-1}\psi$  on the upper two components⟩ ≡
  ⟨Compute  $L_A^{-1}$  on the upper components⟩
  ⟨Compute  $R_A^{-1}$  on the upper components⟩

```

```

⟨Compute  $A^{-1}\psi$  on the lower two components⟩ ≡
  ⟨Compute  $L_A^{-1}$  on the lower components⟩
  ⟨Compute  $R_A^{-1}$  on the lower components⟩

```

```

⟨Compute  $B^{-1}\psi$  on the upper two components⟩ ≡
  ⟨Compute  $L_B^{-1}$  on the upper components⟩
  ⟨Compute  $R_B^{-1}$  on the upper components⟩

```

```

⟨Compute  $B^{-1}\psi$  on the lower two components⟩ ≡
  ⟨Compute  $L_B^{-1}$  on the lower components⟩
  ⟨Compute  $R_B^{-1}$  on the lower components⟩

```

6.8 Standalone off-diagonal pieces

First, simple off-diagonal parts.

(Function prototypes)+≡

```
static void compute_Qxy(VecFermion *d, const VecFermion *s, struct neighbor *nb);
static void inline compute_Qoe(VecOddFermion *d, const VecEvenFermion *s)
{
    compute_Qxy(&d->f, &s->f, &odd_even);
}

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

static void compute_1Sxy(VecFermion *d,
                        const VecFermion *q,
                        const VecFermion *s,
                        struct neighbor *nb);
static void inline compute_1Soe(VecOddFermion *d,
                        const VecOddFermion *q,
                        const VecEvenFermion *s)
{
    compute_1Sxy(&d->f, &q->f, &s->f, &odd_even);
}
```

6.9 Combined pieces

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

(Function prototypes)+≡

```
static void compute_Qxx1Qxy(VecFermion *d,
                        const VecFermion *s,
                        struct neighbor *nb);
static void inline compute_Qee1Qeo(VecEvenFermion *d, const VecOddFermion *s)
{
    compute_Qxx1Qxy(&d->f, &s->f, &even_odd);
}

static void compute_Sxx1Sxy(VecFermion *d,
                        const VecFermion *s,
                        struct neighbor *nb);
static void inline compute_See1Seo(VecEvenFermion *d, const VecOddFermion *s)
{
    compute_Sxx1Sxy(&d->f, &s->f, &even_odd);
}

static void compute_1Qxx1Qxy(VecFermion *d,
                        double *norm,
                        const VecFermion *q,
                        const VecFermion *s,
                        struct neighbor *nb);
static void inline compute_1Qoo1Qoe(VecOddFermion *d,
                        double *norm,
                        const VecOddFermion *q,
                        const VecEvenFermion *s)
{
    compute_1Qxx1Qxy(&d->f, norm, &q->f, &s->f, &odd_even);
}
```

6.10 Common off-diagonal parts

We start from the top level:

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

```

<Functions>+=
static void
compute_Qxy(VecFermion *chi,
             const VecFermion *psi,
             struct neighbor *nb)
{
    <Q common locals>
    <Qxy locals>

    <Setup xy-aliasing of q>
    <Start off-diagonal receives>
    <Finish old off-diagonal sends>
    <Compute projections for Q send>
    <Compute inside part for Qxy>
    <Finish off-diagonal receives>
    <Compute boundary part for Qxy>
    <Finish xy-aliasing of q>
}

```

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

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

```

<Functions>+=
static void
compute_1Sxy(VecFermion *chi,
             const VecFermion *eta,
             const VecFermion *psi,
             struct neighbor *nb)
{
    <Q common locals>
    <Qxy locals>

    <Setup xy-aliasing of q>
    <Start off-diagonal receives>
    <Finish old off-diagonal sends>
    <Compute projections for S send>
    <Compute inside part for 1 - Sxy>
    <Finish off-diagonal receives>
    <Compute boundary part for 1 - Sxy>
    <Finish xy-aliasing of q>
}

```

Likewise,

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

```

<Functions>+=
static void
compute_Qxx1Qxy(VecFermion *chi,
                 const VecFermion *psi,
                 struct neighbor *nb)
{
    <Q common locals>
    <Qxy locals>
    <Qxx locals>

    <Setup xy-aliasing of q>
    <Start off-diagonal receives>
    <Finish old off-diagonal sends>
    <Compute projections for Q send>
    <Compute inside part for Qxx-1Qxy>
    <Finish off-diagonal receives>
    <Compute boundary part for Qxx-1Qxy>
    <Finish xy-aliasing of q>
}

```

and

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

```

<Functions>+=
static void
compute_Sxx1Sxy(VecFermion *chi,
                 const VecFermion *psi,
                 struct neighbor *nb)
{
    <Q common locals>
    <Qxy locals>
    <Qxx locals>

    <Setup xy-aliasing of q>
    <Start off-diagonal receives>
    <Finish old off-diagonal sends>
    <Compute projections for S send>
    <Compute inside part for Sxx-1Sxy>
    <Finish off-diagonal receives>
    <Compute boundary part for Sxx-1Sxy>
    <Finish xy-aliasing of q>
}

```

Finally,

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

```

⟨Functions⟩ +=
static void
compute_1Qxx1Qxy(VecFermion *chi,
                  double *norm,
                  const VecFermion *eta,
                  const VecFermion *psi,
                  struct neighbor *nb)
{
    ⟨Q common locals⟩
    ⟨Qxy locals⟩
    ⟨Qxx locals⟩
    vreal vv;
    vreal nv = vmk1(0.0);

    ⟨Setup xy-aliasing of q⟩
    ⟨Start off-diagonal receives⟩
    ⟨Finish old off-diagonal sends⟩
    ⟨Compute projections for Q send⟩
    ⟨Compute inside part for 1 - Qxx-1Qxy⟩
    ⟨Finish off-diagonal receives⟩
    ⟨Compute boundary part for 1 - Qxx-1Qxy⟩
    *norm = vsum(nv);
    ⟨Start ⟨r,r⟩ computation⟩
    ⟨Finish xy-aliasing of q⟩
}

```

Note, that we do not start all sends together for two reasons: (a) sent data construction is performed sequentially, and, (b) we assume that it is permissible to start several handles separately and wait for them together. The last point is being clarified with SciDAC software group.

6.11 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.

```

⟨Compute projections for Q send⟩ =
{
    int k, i, s, c, *src;
    const VecFermion *f;
    VecHalfFermion *g;

    k = 0; ⟨Construct (1 + γ0) send k-buffer⟩
    k = 1; ⟨Construct (1 - γ0) send k-buffer⟩
    k = 2; ⟨Construct (1 + γ1) send k-buffer⟩
    k = 3; ⟨Construct (1 - γ1) send k-buffer⟩
    k = 4; ⟨Construct (1 + γ2) send k-buffer⟩
    k = 5; ⟨Construct (1 - γ2) send k-buffer⟩
    k = 6; ⟨Construct (1 + γ3) send k-buffer⟩
    k = 7; ⟨Construct (1 - γ3) send k-buffer⟩
}

```



```

⟨Compute projections for S send⟩≡
{
    int k, i, s, c, *src;
    const VecFermion *f;
    VecHalfFermion *g;

    k = 0; ⟨Construct (1 -  $\gamma_0$ ) send k-buffer⟩
    k = 1; ⟨Construct (1 +  $\gamma_0$ ) send k-buffer⟩
    k = 2; ⟨Construct (1 -  $\gamma_1$ ) send k-buffer⟩
    k = 3; ⟨Construct (1 +  $\gamma_1$ ) send k-buffer⟩
    k = 4; ⟨Construct (1 -  $\gamma_2$ ) send k-buffer⟩
    k = 5; ⟨Construct (1 +  $\gamma_2$ ) send k-buffer⟩
    k = 6; ⟨Construct (1 -  $\gamma_3$ ) send k-buffer⟩
    k = 7; ⟨Construct (1 +  $\gamma_3$ ) send k-buffer⟩
}

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

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

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

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

```

```

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

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

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

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

```

6.12 Parts of $Q_{xy}\psi$

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

```

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

⟨Compute boundary part for  $Q_{xy}$ ⟩≡
  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⟩
    ⟨Build SSE SU(3) objects⟩
    ⟨Compute  $Q_{xy}$  part on the boundary s-chain⟩
  }

```

```

⟨Compute  $Q_{xy}$  part on the inside  $s$ -chain⟩≡
  for (s = 0; s < S_4; s++) {
    ⟨Compute  $Q$  inside  $\gamma$ -projections⟩
    ⟨Inside multiply by  $Vs$ ⟩
    ⟨Compute  $Q$   $\gamma$ -unprojections and sum the results⟩
  }

⟨Compute  $Q_{xy}$  part on the boundary  $s$ -chain⟩≡
  for (s = 0; s < S_4; s++) {
    ⟨Compute  $Q$  boundary  $\gamma$ -projections⟩
    ⟨Boundary multiply by  $Vs$ ⟩
    ⟨Compute  $Q$   $\gamma$ -unprojections and sum the results⟩
  }

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

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

```

$\langle \text{Compute } Q \text{ } \gamma\text{-unprojections and sum the results} \rangle \equiv$

```
rs = &rx5[s];
for (c = 0; c < 3; c++) {
    k = 2;  $\langle \text{Unproject } (1 + \gamma_1) \text{ link} \rangle$ 
    k = 3;  $\langle \text{Unproject and accumulate } (1 - \gamma_1) \text{ link} \rangle$ 
    k = 0;  $\langle \text{Unproject and accumulate } (1 + \gamma_0) \text{ link} \rangle$ 
    k = 1;  $\langle \text{Unproject and accumulate } (1 - \gamma_0) \text{ link} \rangle$ 
    k = 4;  $\langle \text{Unproject and accumulate } (1 + \gamma_2) \text{ link} \rangle$ 
    k = 5;  $\langle \text{Unproject and accumulate } (1 - \gamma_2) \text{ link} \rangle$ 
    k = 6;  $\langle \text{Unproject and accumulate } (1 + \gamma_3) \text{ link} \rangle$ 
    k = 7;  $\langle \text{Unproject and accumulate } (1 - \gamma_3) \text{ link} \rangle$ 
}
```

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

$\langle \text{Inside multiply by } Vs \rangle \equiv$

```
for (d = 0; d < 8; d++) {
    VecHalfFermion * __restrict__ h = &hh[d];
    VecSU3 *u = &V[d];
    g = &gg[d];
     $\langle \text{Multiply } *u \text{ by } *g \text{ and store the result in } *h \rangle$ 
}
```

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

$\langle \text{Boundary multiply by } Vs \rangle \equiv$

```
for (d = 0; d < 8; d++) {
    VecHalfFermion * __restrict__ h = &hh[d];
    VecSU3 *u = &V[d];
    g = (m & (1 << d)) ? &nb->rcv_buf[d][ps[d]] : &gg[d];
     $\langle \text{Multiply } *u \text{ by } *g \text{ and store the result in } *h \rangle$ 
}
```

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

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

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

$\langle Q \text{ common locals} \rangle \equiv$

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

Others are used only in Z_{xy} parts:

$\langle Q_{xy} \text{ locals} \rangle \equiv$

```
int xyzt, k, d;
const VecFermion *f;
VecHalfFermion *g;
VecHalfFermion gg[8], hh[8];
VecSU3 V[8];
int ps[8], p5[8];
```

$\langle \text{Extract 1-d addresses} \rangle \equiv$

$\langle \text{Compute } rx5 \rangle$

```

⟨Compute rx5⟩≡
    rx5 = &chi[xyzt5];

```

```

⟨Compute qx5⟩≡
    qx5 = &psi[xyzt5];

```

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

```

⟨Setup xy-aliasing of q⟩≡
    #define qx5 rx5
    #define qs rs

```

```

⟨Finish xy-aliasing of q⟩≡
    #undef qs
    #undef qx5

```

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

```

⟨Check xx-aliasing of q⟩≡
    #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

```

```

⟨End xx-aliasing of q⟩≡
    #undef QSETUP
    #undef Q2R

```

We also need to uplift the gauge fields

```

⟨Build SSE SU(3) objects⟩≡
    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);
            }
        }
    }
}

```

```

⟨Qxy locals⟩+≡
    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:

```

⟨Extract 1-d addresses⟩+≡
    for (d = 0; d < 8; d++)
        p5[d] = nb->site[d].F[xyzt];

```

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

```

⟨Construct neighbor pointers⟩≡
    for (d = 0; d < 8; d++) {
        ps[d] = p5[d] + s;
    }

```

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

```

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

    xyzt = nb->inside[i];
    xyzt5 = xyzt * S_4;
    <Extract 1-d addresses>
    ex5 = &eta[xyzt5];
    <Build SSE SU(3) objects>
    <Compute  $1 - S_{xy}$  part on the inside s-chain>
  }

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

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

<Compute  $1 - S_{xy}$  part on the inside s-chain>≡
  for (s = 0; s < S_4; s++) {
    <Compute S inside  $\gamma$ -projections>
    <Inside multiply by Vs>
    <Compute  $1 - S$   $\gamma$ -unprojections and sum the results>
  }

<Compute  $1 - S_{xy}$  part on the boundary s-chain>≡
  for (s = 0; s < S_4; s++) {
    <Compute S boundary  $\gamma$ -projections>
    <Boundary multiply by Vs>
    <Compute  $1 - S$   $\gamma$ -unprojections and sum the results>
  }

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

```

```

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

```

```

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

```

```

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

```

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

```

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

<Compute boundary part for  $Q_{xx}^{-1}Q_{xy}$ >≡
  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>
    <Build SSE SU(3) objects>
    <Compute  $Q_{xy}$  part on the boundary s-chain>
    <Compute  $Q_{xx}^{-1}$  part on the s-chain>
  }

```

Local variables needed to compute diagonal parts

```

<Qxx locals>≡
  vreal fx;
  VecHalfFermion zV;
  vcomplex zn, z1, z2, z3;
  complex zX[2][3];
  VecHalfFermion xOut;
  VecHalfFermion yOut;

```

Globals that do not change from run to run:

```

<Global Variables>+≡
  static real c0;
  static vreal va0;
  static vreal va1;
  static vreal va2;
  static vreal va3;
  static vreal va4;
  static vreal ab_LA;
  static vreal ab_LB;

```

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

```

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

```



```

<Compute boundary part for  $S_{xx}^{-1}S_{xy}$ >≡
  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>
    <Build SSE SU(3) objects>
    <Compute  $S_{xy}$  part on the boundary s-chain>
    <Compute  $S_{xx}^{-1}$  part on the s-chain>
  }

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

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

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

```

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

```

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

    xyzt = nb->inside[i];
    xyzt5 = xyzt * S_4;
    <Extract 1-d addresses>
    ex5 = &eta[xyzt5];
    <Build SSE SU(3) objects>
    <Compute  $Q_{xy}$  part on the inside s-chain>
    <Compute  $1 - Q_{xx}^{-1}$  part on the s-chain>
  }

```

```

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

    xyzt = nb->boundary[i].index;
    xyzt5 = xyzt * S_4;
    ⟨Extract 1-d addresses⟩
    ex5 = &eta[xyzt5];
    ⟨Build SSE SU(3) objects⟩
    ⟨Compute  $Q_{xy}$  part on the boundary s-chain⟩
    ⟨Compute  $1 - Q_{xx}^{-1}$  part on the s-chain⟩
}

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

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

```

7 SSE

Most of SSE related details are in `sse.h`. Unfortunately, the rest of the code contains implicit knowledge of SSE, so we do not have a clean abstraction barrier here. As usual, the top level is simple: there are types and there are functions.

```

⟨sse.h⟩≡
#ifdef SSE_H
#define SSE_H
⟨SSE types⟩
⟨SSE inline functions⟩
#endif

```

7.1 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.

```

⟨SSE types⟩≡
typedef float real;

```

Next 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.

```

⟨SSE types⟩+=
typedef real vreal __attribute__((mode(V4SF),aligned(16)));

```

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

```
<SSE types>+=
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?):

```
<SSE types>+=
typedef struct SU3 {
    complex v[3][3];
} SU3;

typedef struct {
    vcomplex v[3][3];
} VecSU3;
```

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.)

```
<SSE types>+=
typedef struct {
    vcomplex f[4][3];
} VecFermion;

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

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.

```
<SSE types>+=
typedef struct {
    VecFermion f;
} VecEvenFermion;

typedef struct {
    VecFermion f;
} VecOddFermion;
```

7.2 SSE functions

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.

```
<SSE inline functions>≡
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.

```

⟨SSE inline functions⟩+=
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:

```

⟨SSE inline functions⟩+=
static inline real vsum(vreal v)
{
    real *vv = (real *)&v;
    return vv[0] + vv[1] + vv[2] + vv[3];
}

```

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

```

⟨SSE inline functions⟩+=
static inline void vput_3(vreal *v, real a3)
{
    ((real *)v)[3] = a3;
}

static inline void vput_0(vreal *v, real a0)
{
    ((real *)v)[0] = a0;
}

```

Given

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

compute various shifts as follows:

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

```

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

```

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

```

⟨SSE inline functions⟩+=
static inline vreal shift_up2(vreal a, vreal b)
{
    vreal x = a;
    asm("shufps\t$0x4e,%1,%0"
        : "+x" (x): "x" (b));
    return x;
}

```

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

```

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

```

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

```

<SSE inline functions>+=
static inline vreal shift_down1(vreal a, vreal b)
{
    return shift_up3(a, b);
}

```

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

```

<SSE inline functions>+=
static inline vreal shift_down2(vreal a, vreal b)
{
    return shift_up2(a, b);
}

```

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

```

<SSE inline functions>+=
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:

```

<SSE inline functions>+=
static inline void vhfzero(VecHalfFermion *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;
}

```

8 QMP sods and ends

The time has come to put together QMP related things which were unclear in the QMP Specification. Let us start with printing routines. [NB: There is a discussion underway about including them into QMP spec, but if the present rate of resolving issues will persist, we can expect them implemented sometime the next millenium.]

8.1 Print routines

Two functions printing things on node 0 and doing nothing otherwise are needed. One prints normal messages:

```
<Function prototypes>+≡
void print(const char *fmt, ...);

<Functions>+≡
void
print(const char *fmt, ...)
{
    va_list va;

    if (this_node != 0)
        return;

    va_start(va, fmt);
    vprintf(fmt, va);
    va_end(va);
}
```

Another outputs into the standard error stream:

```
<Function prototypes>+≡
void eprint(const char *fmt, ...);
void
eprint(const char *fmt, ...)
{
    va_list va;

    if (this_node != 0)
        return;

    va_start(va, fmt);
    vfprintf(stderr, fmt, va);
    va_end(va);
}
```

They need input files:

```
<Include files>+≡
#include <stdio.h>
#include <stdarg.h>
```

8.2 Communication pieces

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

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

<Finish off-diagonal receives>≡
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.

```
<Start k-send>≡
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.

```

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

```

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

```

⟨Global Variables⟩+≡
    static struct neighbor *sending = 0;

```

8.3 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.

```

⟨Start ⟨r,r⟩ computation⟩≡
    QMP_sum_double(norm);

```

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

```

⟨Finalize ⟨r,r⟩ computation⟩≡
    /* relax */

```

8.4 Memory allocation

SSE likes its memory aligned at 16 bytes. We need to keep that in mind when asking for QMP memory.

```

⟨Function prototypes⟩+≡
    static void *sse_aligned_buffer(int size);

⟨Functions⟩+≡
    static void *
    sse_aligned_buffer(int size)
    {
        int xcount = size + 15;
        char *ptr = QMP_allocate_aligned_memory(xcount);

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

```

9 TIMING

We only print time it took on node 0 to run the algorithm.

```

⟨Start timer⟩≡
    getrusage(RUSAGE_SELF, &r0);

⟨Stop timer⟩≡
    getrusage(RUSAGE_SELF, &r1);
    iter = k;

```

When reporting timing, we compute the number of floating point operations and print MFlops as well.

⟨Report timing⟩≡

```
{
    double dt = (r1.ru_utime.tv_sec - r0.ru_utime.tv_sec)
                + 10e-6 * (r1.ru_utime.tv_usec - r0.ru_utime.tv_usec);
    print("DWF CG s %d k %d odd %d even %d time %g\n",
          4 * S_4, iter, odd_even.size, even_odd.size, dt);
    print("Performance: lattice %d %d %d %d %d network %d %d %d %d : %g MFlops\n",
          lattice[0], lattice[1], lattice[2], lattice[3], lattice[4],
          network[0], network[1], network[2], network[3],
          performance(S_4 * 4, odd_even.size, even_odd.size, iter, dt));
}
```

Here's a magic formula for performance

⟨Functions⟩+≡

```
static double
performance(int S, int odd, int even, int k, double dt)
{
    double ops = k * ( even * (S * 3168. + 48.) + odd * (S * 3456. + 48.));
    return ops / (dt * 1e6);
}
```

⟨Global Variables⟩+≡

```
static struct rusage r0, r1;
static int iter;
```

⟨Include files⟩+≡

```
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
```

10 STABS

Things that should be implemented in the production version, but are not needed for timing runs.

10.1 Initializing the gauge field

We do not worry about consistency here—but the real thing should send the boundary values to the neighbors instead. In fact, we do not even try to keep the gauge field in $SU(3)$, random numbers are used instead.

Here is the “initializer”

⟨Functions⟩+≡

```
void
random_real(real *v, int count)
{
    int i;

    for (i = count; i--; )
        *v++ = (real)random()/RAND_MAX;
}
```

⟨Function prototypes⟩+≡

```
void initialize_gauge_field(SU3 *u);
void random_real(real *r, int count);
```



```

⟨Functions⟩+=
void
initialize_gauge_field(SU3 *u)
{
    random_real((real *)u, gauge_XYZT * sizeof (SU3) / sizeof (real));
}

```

10.2 Initializing the right hand side

Likewise, the fermions are initialized to random numbers.

```

⟨Function prototypes⟩+=
void initialize_fermion(VecEvenFermion *e, VecOddFermion *o);

⟨Functions⟩+=
void
initialize_fermion(VecEvenFermion *e, VecOddFermion *o)
{
    random_real((real *)e, even_odd.size * S_4 * sizeof (VecFermion) / sizeof (real));
    random_real((real *)o, odd_even.size * S_4 * sizeof (VecFermion) / sizeof (real));
}

```

```

⟨Save solution⟩≡
/* XXX Save solution */

```

Here are some changes to `cg()` to stop it from converging.

```

⟨Fake cg scalars for performance I⟩≡
rho = 1;
norm_z = 1e5;

⟨Fake cg scalars for performance II⟩≡
gamma=1;
rho = 1e5;

```