

Domain Wall Fermions with 4-d EO preconditioning

Version 1.3.0

Andrew Pochinsky

November 10, 2003

1 DEFINITIONS

Here is the definition of the DWF Dirac operator we are using:

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

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 $\psi_{-1,x} = \psi_{N_s-1,x}$.

As Kostas has shown, one can do 4-d even/odd preconditioning to DWF in this form. This allows us to concentrate on computing the operator

$$\phi = Q_{ee}^{-1}Q_{eo}\psi$$

Up to a scale factor, one has

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

Now, $a = 2/M_0$, and $b = -2m_f/M_0$ are numbers and color structure of Q_{ee} is trivial.

The trickiest part to do on a vector architecture is computing Q_{ee}^{-1} with high efficiency. To achieve good performance, we employ a few tricks. First, one s -slice fits comfortably into L1 cache. This allows us to (a) reuse the U field, thus reducing memory traffic, and (b) apply Q_{ee}^{-1} to the result on a s -slice by slice basis while the result of $Q_{eo}\psi$ is still in cache.

2 γ MATRICES

Choice of γ -matrices made in `sse.nw` is still convenient for two reasons:

- γ_5 is diagonal. It allows one to simplify Q_{ee}^{-1} computation.
- Signs in γ_2 make the final step in Q_{eo} a tiny bit simpler. It might be not hugely advantageous, but it cuts a bit a number of floating point operations needed, which is good.

In all cases, we project from `psi[xx.Fup[k[s]]]` to `f[2k]` for the upward direction and from `psi[xx.Fdown[k[s]]]` to `f[2k+1]` in the downward direction for color `c`.

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

```

⟨Extract 1-d addresses⟩≡
    DiracFermion1d pxp0(psi[xx.Fup[xyzt].d[0]]);

⟨Construct neighbors references⟩≡
    VDiracFermion &pp0 = pxp0[s];

⟨Project  $\gamma_1$  upward link⟩≡
    f[0*4+0+0].v[c] = amib(pp0.v[0][c], pp0.v[1][c]);
    f[0*4+0+1].v[c] = amib(pp0.v[2][c], pp0.v[3][c]);

⟨Unproject and accumulate  $\gamma_1$  upward link⟩≡
    set_ap1b(rs.v[0][c], g[0*4+0+0].v[c]);
    set_apib(rs.v[1][c], g[0*4+0+0].v[c]);
    set_ap1b(rs.v[2][c], g[0*4+0+1].v[c]);
    set_apib(rs.v[3][c], g[0*4+0+1].v[c]);

⟨Extract 1-d addresses⟩+≡
    DiracFermion1d pxm0(psi[xx.Fdown[xyzt].d[0]]);

⟨Construct neighbors references⟩+≡
    VDiracFermion &pm0 = pxm0[s];

⟨Project  $\gamma_1$  downward link⟩≡
    f[0*4+2+0].v[c] = apib(pm0.v[0][c], pm0.v[1][c]);
    f[0*4+2+1].v[c] = apib(pm0.v[2][c], pm0.v[3][c]);

⟨Unproject and accumulate  $\gamma_1$  downward link⟩≡
    set_ap1b(rs.v[0][c], g[0*4+2+0].v[c]);
    set_amib(rs.v[1][c], g[0*4+2+0].v[c]);
    set_ap1b(rs.v[2][c], g[0*4+2+1].v[c]);
    set_amib(rs.v[3][c], g[0*4+2+1].v[c]);

```

$$\gamma_2 = \sigma_1 \otimes \sigma_1 = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

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

```
DiracFermion1d pxp1(psi[xx.Fup[xyzt].d[1]]);
```

$\langle \text{Construct neighbors references} \rangle + \equiv$

```
VDiracFermion &pp1 = pxp1[s];
```

$\langle \text{Project } \gamma_2 \text{ upward link} \rangle \equiv$

```
f[1*4+0+0].v[c] = ap1b(pp1.v[0][c], pp1.v[2][c]);
f[1*4+0+1].v[c] = ap1b(pp1.v[1][c], pp1.v[3][c]);
```

This is a special case. The result of Q_{eo} is build starting from $(1 + \gamma_2)$ to save 6 negations.

$\langle \text{Unproject } \gamma_2 \text{ upward link} \rangle \equiv$

```
rs.v[0][c] = g[1*4+0+0].v[c];
rs.v[1][c] = g[1*4+0+1].v[c];
rs.v[2][c] = g[1*4+0+0].v[c];
rs.v[3][c] = g[1*4+0+1].v[c];
```

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

```
DiracFermion1d pxm1(psi[xx.Fdown[xyzt].d[1]]);
```

$\langle \text{Construct neighbors references} \rangle + \equiv$

```
VDiracFermion &pm1 = pxm1[s];
```

$\langle \text{Project } \gamma_2 \text{ downward link} \rangle \equiv$

```
f[1*4+2+0].v[c] = am1b(pm1.v[0][c], pm1.v[2][c]);
f[1*4+2+1].v[c] = am1b(pm1.v[1][c], pm1.v[3][c]);
```

$\langle \text{Unproject and accumulate } \gamma_2 \text{ downward link} \rangle \equiv$

```
set_ap1b(rs.v[0][c], g[1*4+2+0].v[c]);
set_ap1b(rs.v[1][c], g[1*4+2+1].v[c]);
set_am1b(rs.v[2][c], g[1*4+2+0].v[c]);
set_am1b(rs.v[3][c], g[1*4+2+1].v[c]);
```

$$\gamma_3 = \sigma_2 \otimes \sigma_1 = \begin{pmatrix} 0 & 0 & 0 & -i \\ 0 & 0 & -i & 0 \\ 0 & i & 0 & 0 \\ i & 0 & 0 & 0 \end{pmatrix}$$

```

⟨Extract 1-d addresses⟩+≡
  DiracFermion1d pxp2(psi[xx.Fup[xyzt].d[2]]);

⟨Construct neighbors references⟩+≡
  VDiracFermion &pp2 = pxp2[s];

⟨Project  $\gamma_3$  upward link⟩≡
  f[2*4+0+0].v[c] = amib(pp2.v[0][c], pp2.v[3][c]);
  f[2*4+0+1].v[c] = amib(pp2.v[1][c], pp2.v[2][c]);

⟨Unproject and accumulate  $\gamma_3$  upward link⟩≡
  set_ap1b(rs.v[0][c], g[2*4+0+0].v[c]);
  set_ap1b(rs.v[1][c], g[2*4+0+1].v[c]);
  set_apib(rs.v[2][c], g[2*4+0+1].v[c]);
  set_apib(rs.v[3][c], g[2*4+0+0].v[c]);

⟨Extract 1-d addresses⟩+≡
  DiracFermion1d pxm2(psi[xx.Fdown[xyzt].d[2]]);

⟨Construct neighbors references⟩+≡
  VDiracFermion &pm2 = pxm2[s];

⟨Project  $\gamma_3$  downward link⟩≡
  f[2*4+2+0].v[c] = apib(pm2.v[0][c], pm2.v[3][c]);
  f[2*4+2+1].v[c] = apib(pm2.v[1][c], pm2.v[2][c]);

⟨Unproject and accumulate  $\gamma_3$  downward link⟩≡
  set_ap1b(rs.v[0][c], g[2*4+2+0].v[c]);
  set_ap1b(rs.v[1][c], g[2*4+2+1].v[c]);
  set_amib(rs.v[2][c], g[2*4+2+1].v[c]);
  set_amib(rs.v[3][c], g[2*4+2+0].v[c]);

```

$$\gamma_4 = \sigma_3 \otimes \sigma_1 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

```

⟨Extract 1-d addresses⟩+≡
    DiracFermion1d pxp3(psi[xx.Fup[xyzt].d[3]]);

⟨Construct neighbors references⟩+≡
    VDiracFermion &pp3 = pxp3[s];

⟨Project  $\gamma_4$  upward link⟩≡
    f[3*4+0+0].v[c] = ap1b(pp3.v[0][c], pp3.v[1][c]);
    f[3*4+0+1].v[c] = am1b(pp3.v[2][c], pp3.v[3][c]);

⟨Unproject and accumulate  $\gamma_4$  upward link⟩≡
    set_ap1b(rs.v[0][c], g[3*4+0+0].v[c]);
    set_ap1b(rs.v[1][c], g[3*4+0+0].v[c]);
    set_ap1b(rs.v[2][c], g[3*4+0+1].v[c]);
    set_am1b(rs.v[3][c], g[3*4+0+1].v[c]);

⟨Extract 1-d addresses⟩+≡
    DiracFermion1d pxm3(psi[xx.Fdown[xyzt].d[3]]);

⟨Construct neighbors references⟩+≡
    VDiracFermion &pm3 = pxm3[s];

⟨Project  $\gamma_4$  downward link⟩≡
    f[3*4+2+0].v[c] = am1b(pm3.v[0][c], pm3.v[1][c]);
    f[3*4+2+1].v[c] = ap1b(pm3.v[2][c], pm3.v[3][c]);

⟨Unproject and accumulate  $\gamma_4$  downward link⟩≡
    set_ap1b(rs.v[0][c], g[3*4+2+0].v[c]);
    set_am1b(rs.v[1][c], g[3*4+2+0].v[c]);
    set_ap1b(rs.v[2][c], g[3*4+2+1].v[c]);
    set_ap1b(rs.v[3][c], g[3*4+2+1].v[c]);

```

These γ -matrices were chosen to make γ_5 diagonal:

$$\gamma_5 = 1 \otimes \sigma_3 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

This, in turn, makes computation of $Q_{ee}^{-1}\psi$ conceptually easy, because $(1 \pm \gamma_5)$ acts only on upper/lower Dirac components. Therefore, we need a method to compute inverses of the following two matrices:

$$A = \begin{pmatrix} 1 & a & 0 & \cdots & 0 \\ 0 & 1 & a & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & & \cdots & 1 & a \\ b & 0 & \cdots & 0 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 1 & 0 & \cdots & 0 & b \\ a & 1 & \cdots & & 0 \\ 0 & a & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & 1 & 0 \\ 0 & 0 & \cdots & a & 1 \end{pmatrix}$$

If we know how to compute A^{-1} and B^{-1} , computing Q_{ee}^{-1} is easy:

$$Q_{ee}^{-1} = \frac{1 + \gamma_5}{2} A^{-1} + \frac{1 - \gamma_5}{2} B^{-1}.$$

$\langle \text{Compute } Q_{ee}^{-1} \text{ part on the } s\text{-slice} \rangle \equiv$
 $\langle \text{Compute } A^{-1}\psi \text{ on upper two components} \rangle$
 $\langle \text{Compute } B^{-1}\psi \text{ on lower two components} \rangle$

If $A = L_A R_A$, $B = L_B R_B$, where R_A and R_B are bidiagonal:

$$R_A = \begin{pmatrix} 1 & a & 0 & \cdots & 0 \\ 0 & 1 & a & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & & \cdots & 1 & a \\ 0 & 0 & \cdots & 0 & 1 \end{pmatrix} \quad R_B = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 \\ a & 1 & \cdots & & 0 \\ 0 & a & \ddots & \vdots & \vdots \\ \vdots & \vdots & \ddots & 1 & 0 \\ 0 & 0 & \cdots & a & 1 \end{pmatrix},$$

one can easily find

$$L_A = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ \vdots & & \ddots & & & \vdots \\ b & -ab & a^2b & -a^3b & \cdots & 1 + (-a)^{n-1}b \end{pmatrix} \quad L_B = \begin{pmatrix} 1 + (-a)^{n-1}b & (-a)^{n-2}b & \cdots & a^2b & -ab & b \\ 0 & 1 & 0 & \cdots & & 0 \\ \vdots & & \ddots & & & \vdots \\ 0 & \cdots & & & 0 & 1 \end{pmatrix}$$

In these terms,

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

$\langle \text{Compute } A^{-1}\psi \text{ on upper two components} \rangle \equiv$
 $\langle \text{Compute } L_A^{-1} \text{ part} \rangle$
 $\langle \text{Compute } R_A^{-1} \text{ part} \rangle$

$\langle \text{Compute } B^{-1}\psi \text{ on lower two components} \rangle \equiv$
 $\langle \text{Compute } L_B^{-1} \text{ part} \rangle$
 $\langle \text{Compute } R_B^{-1} \text{ part} \rangle$

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

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

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

We can compute z *in situ*. Care should be taken, however, to use SSE in the sums.

```

⟨Compute  $L_A^{-1}$  part⟩≡
    zV = vhfzero; fx = ab;
    for (int s = 0; s < S4_1; s++, fx = fx * va4) {
        VDiracFermion &rs = rx[s];
        ⟨Compute  $zV \leftarrow zV + fx * rx_{[s]}^{up}$ ⟩
    }
    {
        VDiracFermion &rs = rx[S4_1];
        fx.vput_3(c0);
        ⟨Compute  $zV \leftarrow zV + fx * rx_{[s]}^{up}$ ⟩
        for (int c = 0; c < 3; c++) {
            ⟨Compute wall value in zX[c]⟩

            zn = rs.v[0][c];
            zn.vput_3(zX[0][c]);
            rs.v[0][c] = zn;

            zn = rs.v[1][c];
            zn.vput_3(zX[1][c]);
            rs.v[1][c] = zn;
        }
    }

```

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

```

⟨QQxx locals⟩≡
    vreal fx;
    VDiracFermion zV;
    vcomplex zn, z1, z2, z3;
    complex zX[2][3];

```

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

```

⟨Compute  $zV \leftarrow zV + fx * rx_{[s]}^{up}$ ⟩≡
    for (int c = 0; c < 3; c++) {
        zV.v[0][c] = zV.v[0][c] + fx * rs.v[0][c];
        zV.v[1][c] = zV.v[1][c] + fx * rs.v[1][c];
    }

```

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

```

⟨Compute wall value in zX[c]⟩≡
    zX[0][c] = zV.v[0][c].sum();
    zX[1][c] = zV.v[1][c].sum();

```

Computing L_B^{-1} differs fro L_A^{-1} only in the direction of the loop over the fifth dimension and change from the upper to the lower half of fermion indices.

```

⟨Compute  $L_B^{-1}$  part⟩≡
    zV = vhfzero; fx = ab;
    for (int s = S4; --s; fx = fx * va4) {
        VDiracFermion &rs = rx[s];
        ⟨Compute  $zV \leftarrow zV + fx * rx_{[s]}^{down}$ ⟩
    }
    {
        VDiracFermion &rs = rx[0];
        fx.vput_0(c0);
        ⟨Compute  $zV \leftarrow zV + fx * rx_{[s]}^{down}$ ⟩
        for (int c = 0; c < 3; c++) {
            ⟨Compute wall value in zX[c]⟩

            zn = rs.v[2][c];
            zn.vput_0(zX[0][c]);
            rs.v[2][c] = zn;

            zn = rs.v[3][c];
            zn.vput_0(zX[1][c]);
            rs.v[3][c] = zn;
        }
    }
}

```

This piece is used twice: once in the loop over L_s , and the second time after correcting s_0 :

```

⟨Compute  $zV \leftarrow zV + fx * rx_{[s]}^{down}$ ⟩≡
    for (int c = 0; c < 3; c++) {
        zV.v[0][c] = zV.v[0][c] + fx * rs.v[2][c];
        zV.v[1][c] = zV.v[1][c] + fx * rs.v[3][c];
    }
}

```

One can compute $y^{(A)} = R_A^{-1}x$ and $y^{(B)} = R_B^{-1}x$ iteratively:

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

We need one last step to make computation of R^{-1} vector-friendly. Unrolling iterative definitions four times, one gets:

$$\begin{aligned}
 y_k^{(A)} &= \begin{cases} x_k, & \text{if } k = n-1 \\ x_k - ay_{k+1}^{(A)}, & \text{if } n-4 \leq k \leq n-2 \\ x_k - ax_{k+1} + a^2x_{k+2} - a^3x_{k+3} + a^4y_{k+4}^{(A)}, & \text{otherwise} \end{cases} \\
 y_k^{(B)} &= \begin{cases} x_0, & \text{if } k = 0 \\ x_k - ay_{k-1}^{(B)}, & \text{if } 1 \leq k \leq 3 \\ x_k - ax_{k-1} + a^2x_{k-2} - a^3x_{k-3} + a^4y_{k-4}^{(B)}, & \text{otherwise} \end{cases}
 \end{aligned}$$

If we extend x by setting $x_k = 0$ iff $k < 0$ or $k \geq n$ and also set $y_k^{(A)} = 0$ for $k \geq n$ and $y_k^{(B)} = 0$ for $k < 0$, we do not need special cases on the boundaries.

```

⟨Init out of bound x and y⟩≡
    xOut = vhfzero;
    yOut = vhfzero;

```

Again, it is better to place xOut and yOut on the function level:

```

⟨QQxx locals⟩+≡
    VDiracFermion xOut;
    VDiracFermion yOut;

```


With such an extended x and y we can use the following formulae:

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

```

⟨Compute  $R_A^{-1}$  part⟩≡
  ⟨Init out of bound  $x$  and  $y$ ⟩
  for (int s = S4; s--;) {
    VDiracFermion &rs = rx[s];
    for (int c = 0; c < 3; c++) {
      ⟨Compute  $y_{k,[0]}^{(A)}⟩$ 
      ⟨Compute  $y_{k,[1]}^{(A)}⟩$ 
    }
  }

⟨Compute  $y_{k,[0]}^{(A)}⟩$ ≡
  zn = rs.v[0][c];
  z1 = zn.shift_down1(xOut.v[0][c]);
  z2 = zn.shift_down2(xOut.v[0][c]);
  z3 = zn.shift_down3(xOut.v[0][c]);
  rs.v[0][c] = zn - va1 * z1 + va2 * z2 - va3 * z3 + va4 * yOut.v[0][c];
  yOut.v[0][c] = rs.v[0][c];
  xOut.v[0][c] = zn;

⟨Compute  $y_{k,[1]}^{(A)}⟩$ ≡
  zn = rs.v[1][c];
  z1 = zn.shift_down1(xOut.v[1][c]);
  z2 = zn.shift_down2(xOut.v[1][c]);
  z3 = zn.shift_down3(xOut.v[1][c]);
  rs.v[1][c] = zn - va1 * z1 + va2 * z2 - va3 * z3 + va4 * yOut.v[1][c];
  xOut.v[1][c] = zn;
  yOut.v[1][c] = rs.v[1][c];

⟨Compute  $R_B^{-1}$  part⟩≡
  ⟨Init out of bound  $x$  and  $y$ ⟩
  for (int s = 0; s < S4; s++) {
    VDiracFermion &rs = rx[s];
    for (int c = 0; c < 3; c++) {
      ⟨Compute  $y_{k,[2]}^{(B)}⟩$ 
      ⟨Compute  $y_{k,[3]}^{(B)}⟩$ 
    }
  }

⟨Compute  $y_{k,[2]}^{(B)}⟩$ ≡
  zn = rs.v[2][c];
  z1 = xOut.v[2][c].shift_up1(zn);
  z2 = xOut.v[2][c].shift_up2(zn);
  z3 = xOut.v[2][c].shift_up3(zn);
  rs.v[2][c] = zn - va1 * z1 + va2 * z2 - va3 * z3 + va4 * yOut.v[2][c];
  yOut.v[2][c] = rs.v[2][c];
  xOut.v[2][c] = zn;

⟨Compute  $y_{k,[3]}^{(B)}⟩$ ≡
  zn = rs.v[3][c];
  z1 = xOut.v[3][c].shift_up1(zn);
  z2 = xOut.v[3][c].shift_up2(zn);
  z3 = xOut.v[3][c].shift_up3(zn);
  rs.v[3][c] = zn - va1 * z1 + va2 * z2 - va3 * z3 + va4 * yOut.v[3][c];
  yOut.v[3][c] = rs.v[3][c];
  xOut.v[3][c] = zn;

```

The only part left is computing lattice-independant values for the fifth dimension.

```

<Compute constant values for  $Q_{ee}^{-1}$ >≡
    // XXXXX ----- Do it once all logic is cheched.
    real c0 = 1;
    vreal va1;
    vreal va2;
    vreal va3;
    vreal va4;
    vreal ab;
    VDiracFermion vhfzero;

```

This completes computation of $Q_{ee}^{-1}x$.

3 INTERFACE

It is convenient to choose such a basis of Clifford algebra that γ_5 is diagonal—we can use γ -matrices from the previous note.

```

<kostas.hh>≡
    #ifndef KOSTAS_HH
    #define KOSTAS_HH
    #include "qcd.hh"
    namespace QCDSSE {
    <Gauge Fields>
    <Dirac Fermions>

    <QCD classes>
    };
    #endif

```

We can borrow SU3 and VSU3 from `sse.nw`:

```

<Gauge Fields>≡
    struct SU3 {
        complex v[3][3];
    };

```

When applying the parallel transport to fermions, we need to fill SSE 4-vectors with the same values. Because gcc does register allocation for us, we introduce VSU3 structure:

```

<Gauge Fields>+≡
    struct VSU3 {
        vcomplex v[3][3];
    <Vector Gauge Field constructors>
    };

```

We need an empty constructor for VSU3 because gcc is not very smart.

```

<Vector Gauge Field constructors>≡
    inline VSU3() {}

```

Five dimensional fermions are packaged along the s -direction

```

<Dirac Fermions>≡
    struct VDiracFermion {
        vcomplex v[4][3];
    };

```

We will also need results of $(1 \pm \gamma_\mu)$ projections. It is convenient to store them into colored scalars

```

<Dirac Fermions>+≡
    struct VScalar {
        vcomplex v[3];
    };

```

QCD interface is quite similar to the original SSE version

```

<QCD classes>≡
class GaugeField {
    SU3 *v;
public:
    GaugeField(SU3 *r): v(r) {}
    SU3 &operator[](int i) { return v[i];}
};

class GaugeField4d {
    SU3 *v;
public:
    GaugeField4d(SU3 *r): v(r) {}
    GaugeField operator[](int i) const { return GaugeField(v + 4 * i); }
};

```

For Dirac fermions, on the other hand, it is convenient to have 1- and 5-dimensional classes:

```

<QCD classes>+≡
class DiracFermion1d {
    VDiracFermion *f;
public:
    DiracFermion1d(): f(0) {}
    DiracFermion1d(VDiracFermion *x): f(x) {}
    VDiracFermion &operator[](int i) { return f[i]; }
};

class DiracFermion5d {
    VDiracFermion *f;
    int S4;
public:
    DiracFermion5d(VDiracFermion *x, int s4): f(x), S4(s4) {}
    DiracFermion1d operator[](int i) const { return DiracFermion1d(f + S4 * i);}
};

```

For the sake of exposition, let us collect all pieces describing lattice geometry into one class. It is likely, that an application will have only one variable of class `Lattice5d`, but what the heck.

```

⟨QCD classes⟩+=
class Lattice5d {
    int X, Y, Z, T, S4, S4_1;
    int XYZT2;

    struct step {
        int d[4];
    };
    struct rb_lattice {
        step *Fdown;
        step *Fup;
        step *Udown;
        int *Uup;
    };
    rb_lattice eo;
    void QQxx(DiracFermion5d &result,
              const GaugeField4d &U,
              const DiracFermion5d &psi,
              double a, double b,
              const rb_lattice &xx);
    int full_lattice(int x, int y, int z, int t);
    int even_lattice(int x, int y, int z, int t);
    int odd_lattice(int x, int y, int z, int t);
public:
    Lattice5d(int x, int y, int z, int t, int s);
    ~Lattice5d();
    DiracFermion5d create_fermion(void);
    GaugeField4d create_gauge(void);
    void Qee1Qeo(DiracFermion5d &result,
                 const GaugeField4d &U,
                 const DiracFermion5d &psi,
                 double a, double b) { QQxx(result, U, psi, a, b, eo); }
    double QQ0ps(void);
};

```

3.1 Field Allocators

In its present form, gcc does not perform `new` properly on `VREAL` objects. For this reason, we implement two methods in `[Lattice5d]` responsible for allocation of gauge fields and five-dimensional fermions respectively.

```

⟨QCD methods⟩=

GaugeField4d
Lattice5d::create_gauge(void)
{
    SU3 *v = new SU3[XYZT2 * 2 * 4];

    ⟨Initialize Gauge Field⟩
    return v;
}

```

For now, we fill gauge field with random numbers:

```

⟨Initialize Gauge Field⟩=
    real *f = (real *)v;
    for (int i = sizeof(SU3) * 4 * 2 * XYZT2 / sizeof(real); i--;) {
        f[i] = rand() / (double)RAND_MAX;
    }

```

Equally simple is allocating a fresh red/black sublattice of VDiracFermion. Unfortunately, gcc's `new` is broken, we need to align SSE data manually.

```

<QCD methods>+=
  DiracFermion5d
  Lattice5d::create_fermion(void)
  {
    char *ptr = new char[sizeof (VDiracFermion) * (XYZT2 * 2 * S4 + 1)];
    unsigned long v = (unsigned long)ptr;
    unsigned long v_aligned = (v + 15) & ~15;

    <Initialize 5d Fermion>
    return DiracFermion5d((VDiracFermion *)v_aligned, S4);
  }

```

Fermion field is also initialized with random numbers:

```

<Initialize 5d Fermion>=
  real *f = (real *)ptr;
  for (int i = sizeof(VDiracFermion) / sizeof(f) * XYZT2 * 2 * S4; i--;) {
    f[i] = rand() / (double)RAND_MAX;
  }

```

In the Lattice5d object we store all information needed to navigate through the lattice and create lattice objects (e.g., gauge and fermion fields)

```

<QCD methods>+=
  Lattice5d::Lattice5d(int x, int y, int z, int t, int s)
  {
    <Compute Lattice5d sizes>
    <Create eo neighbor tables>
  }

```

Remember that four-vectors are aligned along the *s*-direction and even/odd is done in 4-d slices:

```

<Compute Lattice5d sizes>=
  X = x; Y = y; Z = z; T = t; S4 = s/4; S4_1 = S4 - 1;
  XYZT2 = X * Y * Z * T / 2;

```

We only compute `eo` tables here.

```

<Create eo neighbor tables>=
  eo.Fdown = new step[XYZT2];
  eo.Fup = new step[XYZT2];
  eo.Udown = new step[XYZT2];
  eo.Uup = new int[XYZT2];
  <Fill neighbor tables values>

```

To fill the values in the neighbor tables, start with walking over four dimensions

```

<Fill neighbor tables values>=
  for (int x = 0; x < X; x++) {
    for (int y = 0; y < Y; y++) {
      for (int z = 0; z < Z; z++) {
        int p = (x + y + z) & 1;
        for (int t = p; t < T; t +=2) {
          int i = even_lattice(x,y,z,t);
          <Compute eo[i] elements for (x,y,z,t)>
        }
      }
    }
  }
}

```

Since we have both linear and four-dimensional addresses of the current point, computing the neighbors is easy:

```

⟨Compute eo[i] elements for (x,y,z,t)⟩≡
    eo.Uup[i] = full_lattice(x,y,z,t);
    eo.Udown[i].d[0] = full_lattice(x-1,y,z,t);
    eo.Udown[i].d[1] = full_lattice(x,y-1,z,t);
    eo.Udown[i].d[2] = full_lattice(x,y,z-1,t);
    eo.Udown[i].d[3] = full_lattice(x,y,z,t-1);
    eo.Fup[i].d[0] = odd_lattice(x+1,y,z,t);
    eo.Fup[i].d[1] = odd_lattice(x,y+1,z,t);
    eo.Fup[i].d[2] = odd_lattice(x,y,z+1,t);
    eo.Fup[i].d[3] = odd_lattice(x,y,z,t+1);
    eo.Fdown[i].d[0] = odd_lattice(x-1,y,z,t);
    eo.Fdown[i].d[1] = odd_lattice(x,y-1,z,t);
    eo.Fdown[i].d[2] = odd_lattice(x,y,z-1,t);
    eo.Fdown[i].d[3] = odd_lattice(x,y,z,t-1);

```

Three converters from 4-d lattice address to 1-d memory address:

```

⟨QCD methods⟩+≡
    int
    Lattice5d::full_lattice(int x, int y, int z, int t)
    {
        if (x < 0) x += X; if (x >= X) x -= X;
        if (y < 0) y += Y; if (y >= Y) y -= Y;
        if (z < 0) z += Z; if (z >= Z) z -= Z;
        if (t < 0) t += T; if (t >= T) t -= T;
        return t + T * (z + Z * (y + Y * x));
    }

```

```

⟨QCD methods⟩+≡
    int
    Lattice5d::even_lattice(int x, int y, int z, int t)
    {
        return full_lattice(x,y,z,t)/2;
    }

```

```

⟨QCD methods⟩+≡
    int
    Lattice5d::odd_lattice(int x, int y, int z, int t)
    {
        return full_lattice(x,y,z,t)/2;
    }

```

Finally, we need to cleanup space used by the tables when Lattice5d is freed:

```

⟨QCD methods⟩+≡
    Lattice5d::~~Lattice5d()
    {
        delete eo.Fdown;
        delete eo.Fup;
        delete eo.Udown;
        delete eo.Uup;
    }

```

4 THE BEEF

Armed with all the experience of SSE magic, we boldly go ahead:

```

⟨kostas.cc⟩≡
    #include <stdlib.h>
    #include "kostas.hh"
    using namespace QCDSSSE;
    ⟨QCD methods⟩

```

We traditionally start with an implementation of the preconditioned Dirak operator.

```

<QCD methods>+=
void
Lattice5d::QQxx(DiracFermion5d &result,
                const GaugeField4d &U,
                const DiracFermion5d &psi,
                double a, double b,
                const rb_lattice &xx)
{
    <QQxx locals>

    <Compute constant values for  $Q_{ee}^{-1}$ >
    for (int xyz = XYZT2; xyz--;) {
        <Extract 1-d addresses>
        <Build SSE  $SU(3)$  objects>
        <Compute  $Q_{eo}$  part on the s-slice>
        <Compute  $Q_{ee}^{-1}$  part on the s-slice>
    }
}

```

The Q-code has grown too large to fit into the trace cache. We need to compact the inner loop. (Such a change seems not to effect performance of P4, by the way). We pack all VSU3 into an array, so that all sixteen $SU(3)$ multiplications can be performed in a loop.

```

<Build SSE  $SU(3)$  objects>=
GaugeField Uup = U[xx.Uup[xyz]];
for (int d = 0; d < 4; d++) {
    SU3 &Udown = U[xx.Udown[xyz].d[d]][d];
    for (int a = 0; a < 3; a++) {
        for (int b = 0; b < 3; b++) {
            V[d*2+0].v[a][b] = vcomplex(Uup[d].v[a][b].re, Uup[d].v[a][b].im);
            V[d*2+1].v[a][b] = vcomplex(Udown.v[b][a].re, -Udown.v[b][a].im);
        }
    }
}

```

Unfortunately, gcc does something quite unexplicable with some block-level declarations. To avoid run-time overhead, all SSE variables are declared at the function level in QQxx:

```

<QQxx locals>+=
VSU3 V[8];

```

With all gauge fields brought into L1 cache and converted into SSE vectors, we are ready to compute an s-slice of Q_{eo} .

```

<Compute  $Q_{eo}$  part on the s-slice>=
for (int s = 0; s < S4; s++) {
    <Compute  $\gamma$ -projections>
    <Multiply by V>
    <Compute  $\gamma$ -unprojections and sum the results>
}

```

To prepare for matrix multiplication, γ -projections are combined with collecting the fermion pieces into an L1 array.

```

<Compute  $\gamma$ -projections>=
<Construct neighbors references>
for (int c = 0; c < 3; c++) {
    <Project  $\gamma_1$  upward link>
    <Project  $\gamma_1$  downward link>
    <Project  $\gamma_2$  upward link>
    <Project  $\gamma_2$  downward link>
    <Project  $\gamma_3$  upward link>
    <Project  $\gamma_3$  downward link>
    <Project  $\gamma_4$  upward link>
    <Project  $\gamma_4$  downward link>
}

```

```

⟨QQxx locals⟩+≡
    VScalar f[16], g[16];

```

Here is a major space saver. By the clever tricks above, all pieces are in L1 and we can do the multiplications in a neat loop.

```

⟨Multiply by V⟩≡
    for (int n = 0; n < 8; n++) {
        VScalar &f0 = f[2*n+0];
        VScalar &f1 = f[2*n+1];
        VScalar &g0 = g[2*n+0];
        VScalar &g1 = g[2*n+1];
        VSU3 &Vx = V[n];

        for (int c = 0; c < 3; c++) {
            g0.v[c] = Vx.v[c][0]*f0.v[0] + Vx.v[c][1]*f0.v[1] + Vx.v[c][2]*f0.v[2];
            g1.v[c] = Vx.v[c][0]*f1.v[0] + Vx.v[c][1]*f1.v[1] + Vx.v[c][2]*f1.v[2];
        }
    }

```

The final step in Q_{eo} could also be compacted. One peculiarity here is that we start collecting results from $(1 + \gamma_2)$ because it saves us a few additions.

```

⟨Compute  $\gamma$ -unprojections and sum the results⟩≡
    VDiracFermion &rs = rx[s];
    for (int c = 0; c < 3; c++) {
        ⟨Unproject  $\gamma_2$  upward link⟩
        ⟨Unproject and accumulate  $\gamma_2$  downward link⟩
        ⟨Unproject and accumulate  $\gamma_1$  upward link⟩
        ⟨Unproject and accumulate  $\gamma_1$  downward link⟩
        ⟨Unproject and accumulate  $\gamma_3$  upward link⟩
        ⟨Unproject and accumulate  $\gamma_3$  downward link⟩
        ⟨Unproject and accumulate  $\gamma_4$  upward link⟩
        ⟨Unproject and accumulate  $\gamma_4$  downward link⟩
    }

```

This simply helps gcc in its quest for optimization

```

⟨Extract 1-d addresses⟩+≡
    DiracFermion1d rx(result[xyzt]);

```

Finally, let us count the floating point operations.

| | | | |
|------------------------------------|---|---|------|
| γ -projections | $2(\text{complex}) \times 3(\text{colors}) \times 2(\text{half fermions}) \times 8(\text{directions}) \times 1(\text{add})$ | = | 96 |
| γ -unprojections | $2(\text{complex}) \times 3(\text{colors}) \times 4(\text{full fermions}) \times 7(\text{directions}) \times 1(\text{add})$ | = | 168 |
| Gauge multiplications | $2(\text{complex}) \times 3(\text{colors}) \times 2(\text{half fermions}) \times 8(\text{directions}) \times (6(\text{multiply}) + 5(\text{add}))$ | = | 1056 |
| Applying L_A^{-1} and L_B^{-1} | $2(L_A^{-1} \text{ and } L_B^{-1}) \times 2(\text{complex}) \times 3(\text{colors}) \times 2(\text{half fermions}) \times (1(\text{multiply}) + 1(\text{add}))$ | = | 48 |
| Applying R_A^{-1} and R_B^{-1} | $2(R_A^{-1} \text{ and } R_B^{-1}) \times 2(\text{complex}) \times 3(\text{colors}) \times 2(\text{half fermions}) \times (4(\text{multiply}) + 4(\text{add}))$ | = | 192 |
| Total per site | | = | 1560 |

thus,

```

⟨QCD methods⟩+≡
    double
    Lattice5d::QQOps(void)
    {
        return (double)XYZT2 * (double)S4 * 4. * 1560.;
    }

```


5 TEST CODE

The test is to loop K_{eo} enough time to get an estimate of performance.

```
<k-main.cc>≡
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <unistd.h>
#include "kostas.hh"

int
main(int argc, char *argv[]) {
    using namespace QCDSSSE;
    <check arguments>
    <create dynamic objects>
    struct rusage us0, us1;
    struct timeval t0, t1;
    getrusage(RUSAGE_SELF, &us0);
    <perform SSE operations>
    getrusage(RUSAGE_SELF, &us1);
    t0 = us0.ru_utime;
    t1 = us1.ru_utime;
    <show time elapsed and performance>

    return 0;
}
```

Lattice size and number of iterations are arguments of `main()`:

```
<check arguments>≡
if (argc != 7) {
    fprintf(stderr, "usage: sse X Y Z T S iter\n");
    return 1;
}
```

From X, Y, Z and T we only require to be even and positive:

```
<check arguments>+≡
int X = atoi(argv[1]);
if ((X <= 0) || (X & 1 != 0)) {
    fprintf(stderr, "X must be even positive. (it is %d)\n", X);
    return 1;
}
```

```
<check arguments>+≡
int Y = atoi(argv[2]);
if ((Y <= 0) || (Y & 1 != 0)) {
    fprintf(stderr, "Y must be even positive. (it is %d)\n", Y);
    return 1;
}
```

```
<check arguments>+≡
int Z = atoi(argv[3]);
if ((Z <= 0) || (Z & 1 != 0)) {
    fprintf(stderr, "Z must be even positive. (it is %d)\n", Z);
    return 1;
}
```

```

<check arguments>+=
    int T = atoi(argv[4]);
    if ((T <= 0) || (T & 1 != 0)) {
        fprintf(stderr, "T must be even positive. (it is %d)\n", T);
        return 1;
    }

```

However, S must be a multiple of 8:

```

<check arguments>+=
    int S = atoi(argv[5]);
    if ((S <= 4) || (S & 3 != 0)) {
        fprintf(stderr,
            "S must be positive multiple of 4 and at least 8 (it is %d)\n",
            S);
        return 1;
    }

```

Number of iteration is simply taken from the sixth argument:

```

<check arguments>+=
    int count = atoi(argv[6]);

```

Next, we create Lattice5d object that will handle all other memory allocations.

```

<create dynamic objects>=
    Lattice5d Lattice(X, Y, Z, T, S);

```

We need one gauge field on Lattice:

```

<create dynamic objects>+=
    GaugeField4d U = Lattice.create_gauge();

```

and two Dirac fermions (even and odd sublattices)

```

<create dynamic objects>+=
    DiracFermion5d phi = Lattice.create_fermion();
    DiracFermion5d psi = Lattice.create_fermion();

```

Now we have all the pieces to compute $\phi = K_{eo}\psi$. Values of $a = 2/M_0$ and $b = -2m_f/M_0$ are not important for performance measurements:

```

<perform SSE operations>=
    for (int i = count; i--;) {
        Lattice.Qee1Qeo(phi, U, psi, 0.1, -0.01);
    }

```

Computing elapsed time is easy

```

<show time elapsed and performance>=
    double dt = t1.tv_sec - t0.tv_sec + 10e-6*(t1.tv_usec - t0.tv_usec);

```

However, the elapsed time may be too short to compute performance. Check of overflow here to avoid embarrassment

```

<show time elapsed and performance>+=
    if (dt > 0) {
        printf("Performance %g Mflops/sec. Lattice %d %d %d %d %d ."
            " %d Iterations. %g total seconds (%.0f ops/QQ)\n",
            (double)count * Lattice.QQOps() / dt / 1e6,
            X, Y, Z, T, S, count, dt, Lattice.QQOps());
    } else {
        printf("*** Qee1Qeo ran too fast. Try increasing count\n");
        return 1;
    }

```