# How to measure residual mass and chiral condensate for the domain wall fermion using Hadrons code

Yu Zhang

FTRT Seminar

2022.03.30

## Hadrons Code

- The modified Hadrons code which can measure chiral condensate and two point meson correlator with source-sink separation lie in x direction, y direction, z direction and t direction is located in the following directory of Fugaku:

x dir: /home/ra000001/a04428/src/modify_mres_finiteT/20220204_Add_Z2_psibarpsi_Hadrons.v20210716

y dir: /home/ra000001/a04428/src/modify_mres_finiteT/20220214_ydir_mres_Hadrons.v20210716

z dir: /home/ra000001/a04428/src/modify_mres_finiteT/20220214_zdir_mres_Hadrons.v20210716

t dir: /home/ra000001/a04428/src/20211216_Add_psibarpsi_Grid.v20210713

## How to create new module

- We need to create new modules for spatial meson correlator and chiral condensate, this can be done by executing the bash script add_module_list.sh in Hadrons code.

## Determining the residual mass

- Residual chiral symmetry breaking of domain wall fermion due to finite Ls is characterized by residual mass $m_{\mathrm{res}}$
- Measure the ratio of midpoint correlator to the pion correlator evaluated at large source-sink separations to remove the unphysical states contribution

$$m_{\mathrm{res}} = R(t) = \frac{\left\langle \sum_{\vec{x}} J^a_{5q}(\vec{x}, t)\pi^a(\vec{0}, 0) \right\rangle}{\left\langle \sum_{\vec{x}} J^a_{5}(\vec{x}, t)\pi^a(\vec{0}, 0) \right\rangle}$$

## Determining the residual mass

- $$m_{\text{res}} = R(t) = \frac{\sum_{\overrightarrow{x}} \left\langle J^a_{5q}(\overrightarrow{x}, t)\pi^a(\overrightarrow{0},0) \right\rangle}{\sum_{\overrightarrow{x}} \left\langle J^a_{5}(\overrightarrow{x}, t)\pi^a(\overrightarrow{0},0) \right\rangle}$$

**Physical four dimensional fermion:**

$$q(x) = P_L\Psi(x,0) + P_R\Psi(x, L_s - 1); \;\; \bar{q}(x) = \bar{\Psi}(x, L_s - 1)P_L + \bar{\Psi}(x,0)P_R; \;\; P_{L,R} = \frac{1 \pm \gamma_5}{2}$$

- $J^a_{5q} \approx m_{\text{res}} J^a_5$

- $J^a_5(x) = -\bar{\Psi}(x, L_s - 1)P_L t^a \Psi(x,0) + \bar{\Psi}(x,0)P_R t^a \Psi(x, L_s - 1) = \bar{q}(x)t^a\gamma_5 q(x)$

- $\pi^a(x) = i\bar{q}(x)t^a\gamma_5 q(x) = iJ^a_5$

## Motivation

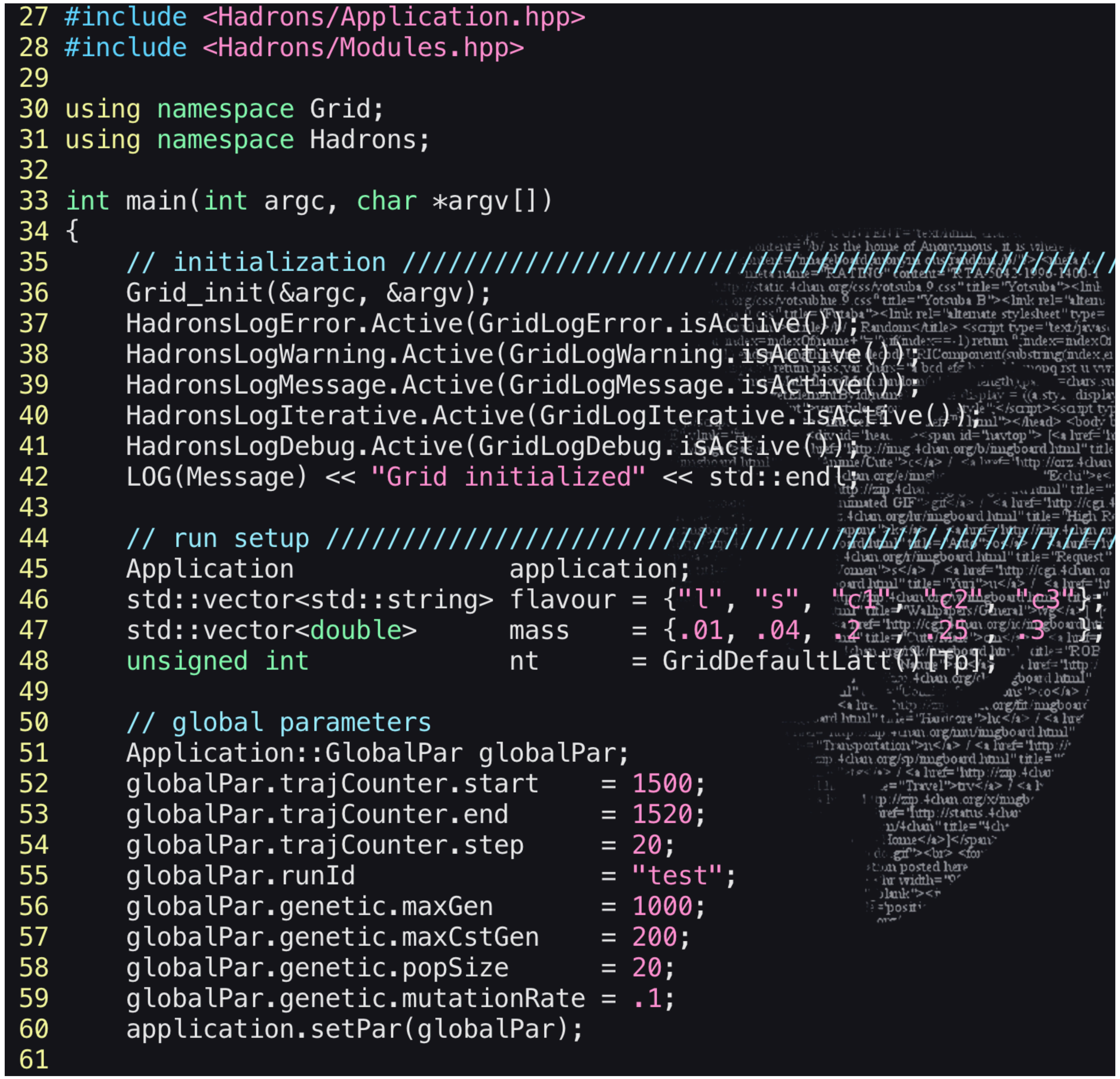
$T = \frac{1}{aN_t}, N_t$ is rather short for finite T.

In this case, one usually determine the residual mass $m_{\text{res}}$ by computing the ratio of the midpoint correlator to the pion correlator evaluated at source-sink separations which lie in a spatial rather than temporal direction.

Currently, Hadrons code has program for calculating the temporal meson correlator, but don't have program for spatial meson correlator.

How can we build spatial meson correlator program based on temporal meson correlator program ?

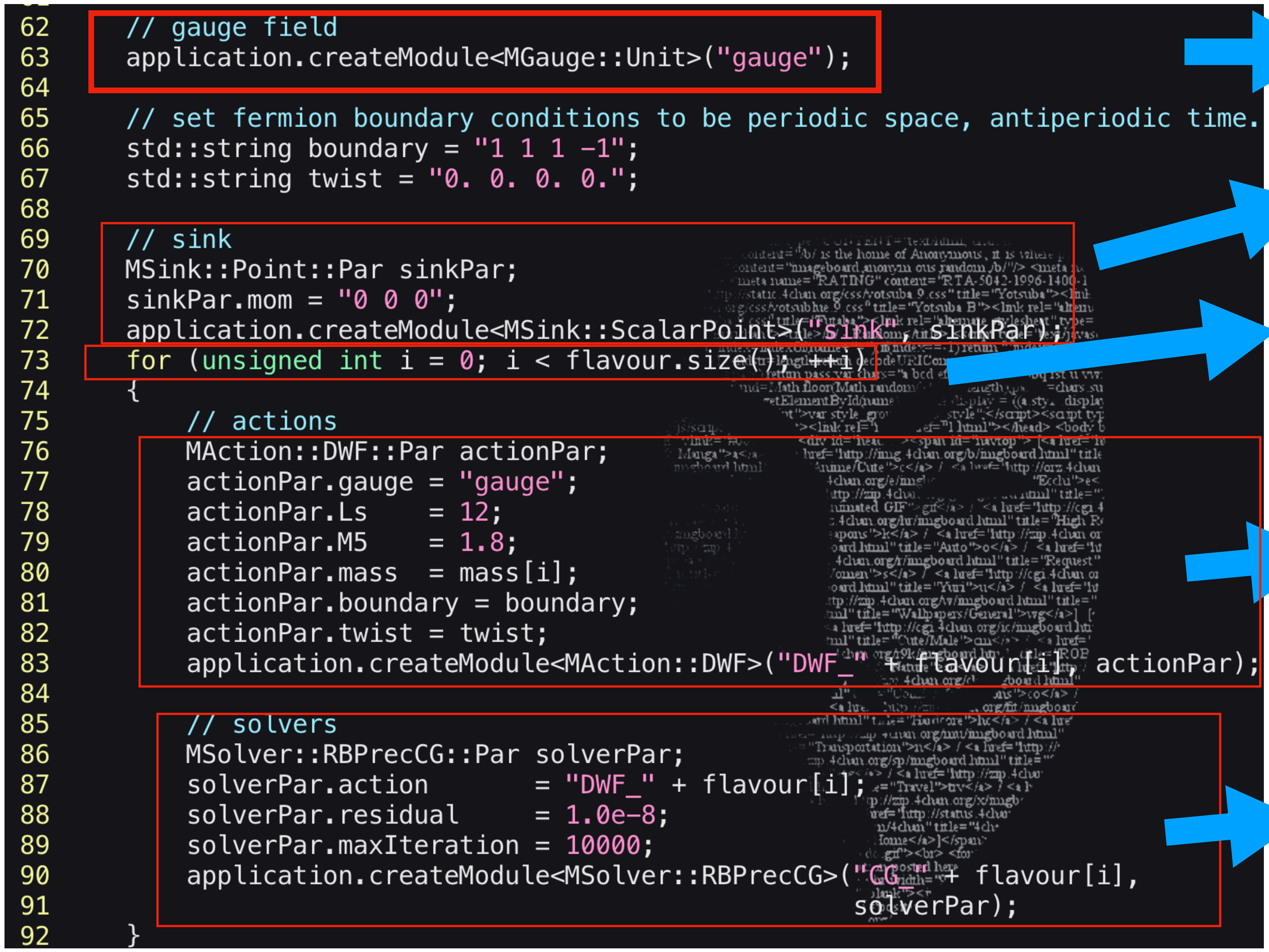
# Temporal meson correlator



```cpp
#include <Hadrons/Application.hpp>
#include <Hadrons/Modules.hpp>

using namespace Grid;
using namespace Hadrons;

int main(int argc, char *argv[])
{
    // initialization ///////////////////////////////////////////////
    Grid_init(&argc, &argv);
    HadronsLogError.Active(GridLogError.isActive());
    HadronsLogWarning.Active(GridLogWarning.isActive());
    HadronsLogMessage.Active(GridLogMessage.isActive());
    HadronsLogIterative.Active(GridLogIterative.isActive());
    HadronsLogDebug.Active(GridLogDebug.isActive());
    LOG(Message) << "Grid initialized" << std::endl;

    // run setup ////////////////////////////////////////////////////
    Application              application;
    std::vector<std::string> flavour = {"l", "s", "c1", "c2", "c3"};
    std::vector<double>      mass    = {.01, .04, .2  , .25 , .3  };
    unsigned int             nt      = GridDefaultLatt()[Tp];

    // global parameters
    Application::GlobalPar globalPar;
    globalPar.trajCounter.start    = 1500;
    globalPar.trajCounter.end      = 1520;
    globalPar.trajCounter.step     = 20;
    globalPar.runId                = "test";
    globalPar.genetic.maxGen       = 1000;
    globalPar.genetic.maxCstGen    = 200;
    globalPar.genetic.popSize      = 20;
    globalPar.genetic.mutationRate = .1;
    application.setPar(globalPar);

```

- Trajectory variables can be used to loop over configurations

# Temporal meson correlator



```
62      // gauge field
63      application.createModule<MGauge::Unit>("gauge");
64
65      // set fermion boundary conditions to be periodic space, antiperiodic time.
66      std::string boundary = "1 1 1 -1";
67      std::string twist = "0. 0. 0. 0.";
68
69      // sink
70      MSink::Point::Par sinkPar;
71      sinkPar.mom = "0 0 0";
72      application.createModule<MSink::ScalarPoint>("sink", sinkPar);
73      for (unsigned int i = 0; i < flavour.size(); ++i)
74      {
75          // actions
76          MAction::DWF::Par actionPar;
77          actionPar.gauge = "gauge";
78          actionPar.Ls    = 12;
79          actionPar.M5    = 1.8;
80          actionPar.mass  = mass[i];
81          actionPar.boundary = boundary;
82          actionPar.twist = twist;
83          application.createModule<MAction::DWF>("DWF_" + flavour[i], actionPar);
84
85          // solvers
86          MSolver::RBPrecCG::Par solverPar;
87          solverPar.action        = "DWF_" + flavour[i];
88          solverPar.residual      = 1.0e-8;
89          solverPar.maxIteration = 10000;
90          application.createModule<MSolver::RBPrecCG>("CG_" + flavour[i],
91                                                      solverPar);
92      }
```

**Set all the link variables to one, so this program calculates correlators for the free case, if one wants to load nersc format configuration, use MIO::LoadNersc**

**Create a sink module "sink" at $p_x, p_y, p_z = 0$, the value of sink corresponds to $\vec{p} = 0$**

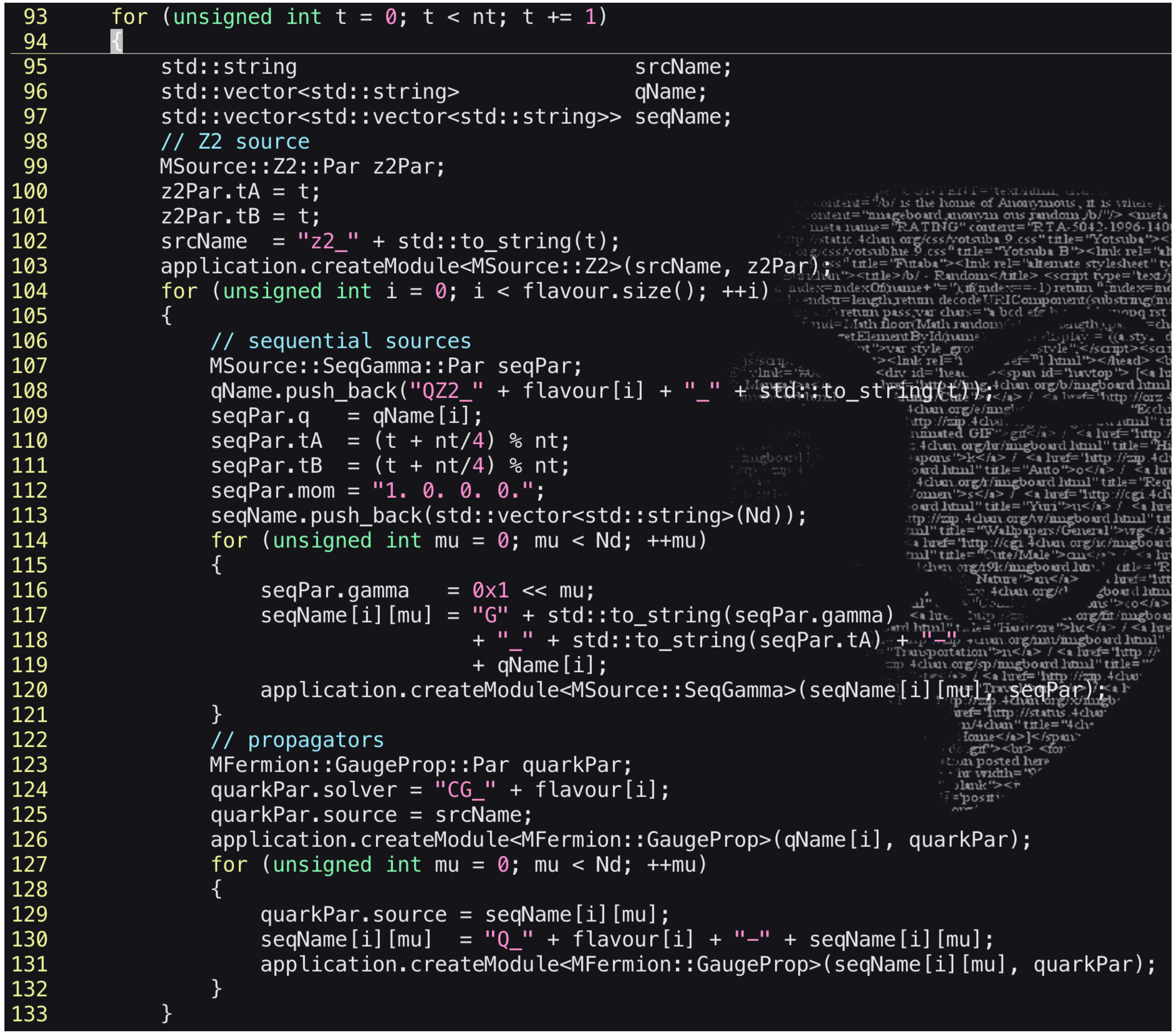
**Loop over all declared quarks to set for each flavor action, solver.**

**As action here it use domain wall fermion, where the gauge fields are from "gauge" and set Ls=12, M5=1.8, mass corresponding to the flavor, boundary and twist are set.**
**In line 83, it create a DWF action for the given flavor and name it "DWF_" + flavour[I]**

**To calculate the quark propagator, it needs to set a solver module which is used by propagator module.**
**It use the module MSolver::RBPrecCG, where the action is given by the previous declared one, the residual and maximal iteration steps are set**

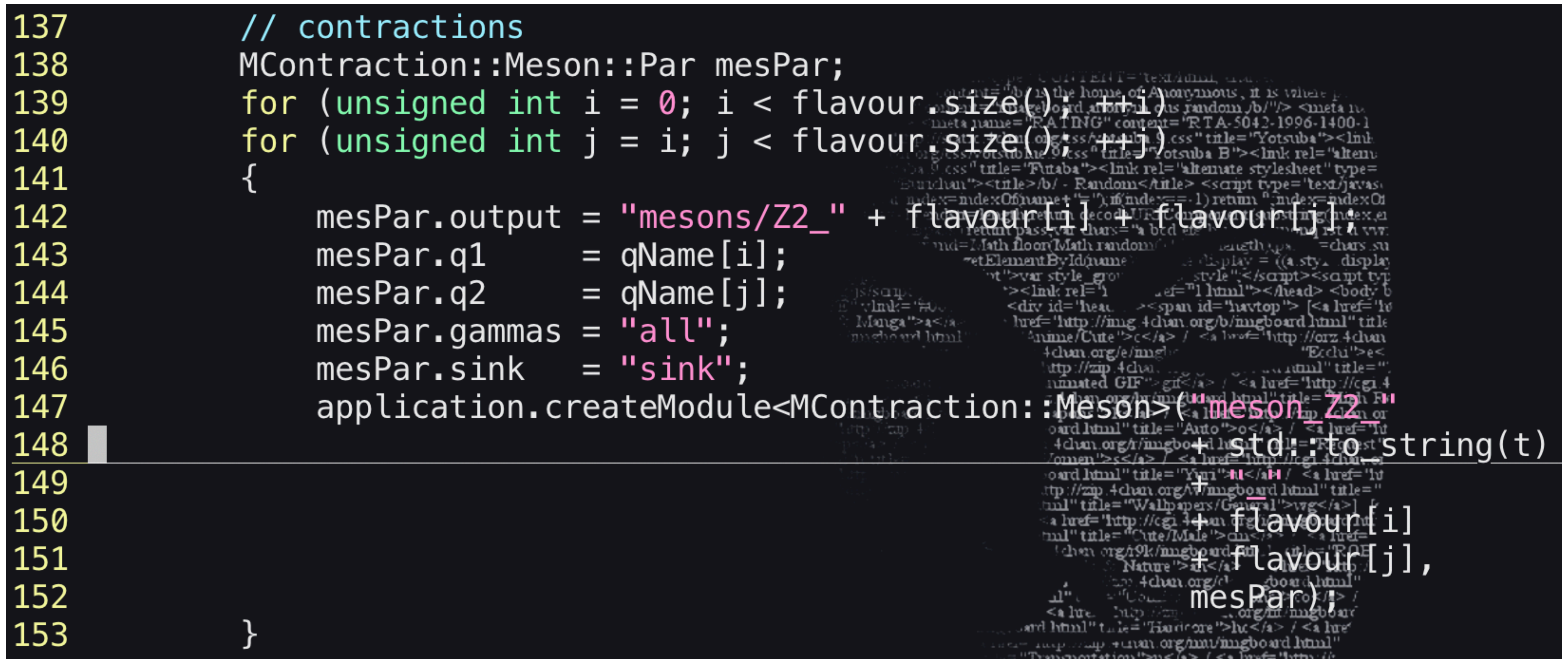
**RBPrecCG is a solver, it uses red black preconditioning and conjugate gradient method to obtain the propagator**

# Temporal meson correlator

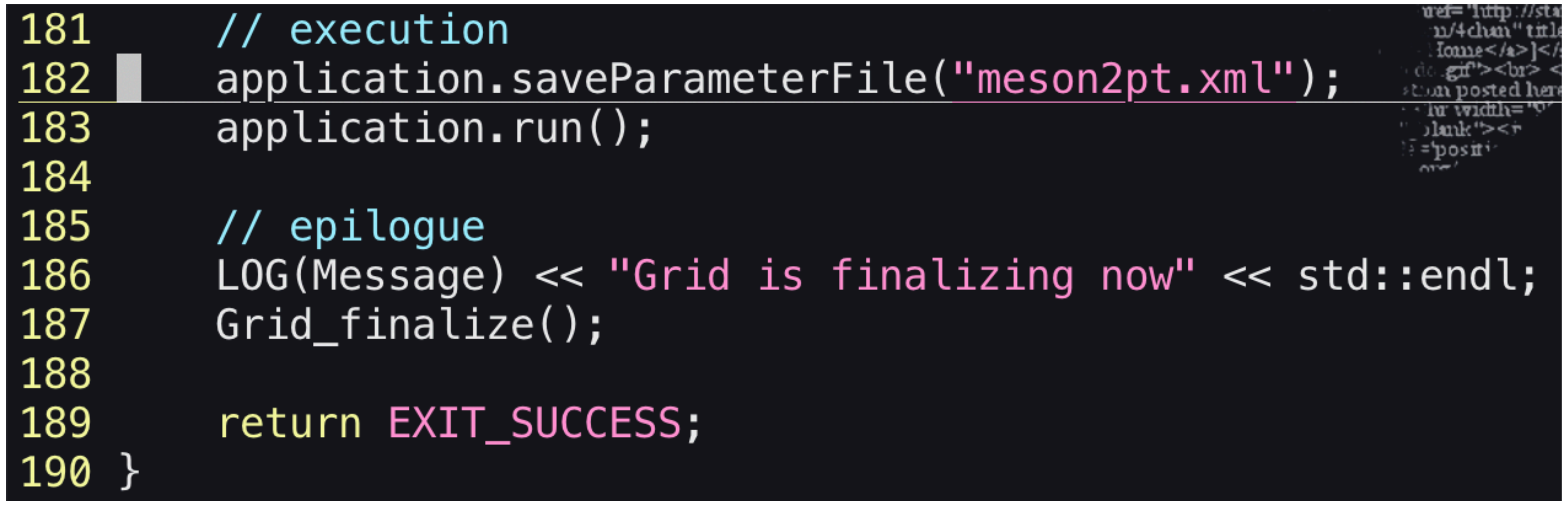


```cpp
    for (unsigned int t = 0; t < nt; t += 1)
    {
        std::string                          srcName;
        std::vector<std::string>             qName;
        std::vector<std::vector<std::string>> seqName;
        // Z2 source
        MSource::Z2::Par z2Par;
        z2Par.tA = t;
        z2Par.tB = t;
        srcName  = "z2_" + std::to_string(t);
        application.createModule<MSource::Z2>(srcName, z2Par);
        for (unsigned int i = 0; i < flavour.size(); ++i)
        {
            // sequential sources
            MSource::SeqGamma::Par seqPar;
            qName.push_back("QZ2_" + flavour[i] + "_" + std::to_string(t));
            seqPar.q   = qName[i];
            seqPar.tA  = (t + nt/4) % nt;
            seqPar.tB  = (t + nt/4) % nt;
            seqPar.mom = "1. 0. 0. 0.";
            seqName.push_back(std::vector<std::string>(Nd));
            for (unsigned int mu = 0; mu < Nd; ++mu)
            {
                seqPar.gamma   = 0x1 << mu;
                seqName[i][mu] = "G" + std::to_string(seqPar.gamma)
                                 + "_" + std::to_string(seqPar.tA) + "-"
                                 + qName[i];
                application.createModule<MSource::SeqGamma>(seqName[i][mu], seqPar);
            }
            // propagators
            MFermion::GaugeProp::Par quarkPar;
            quarkPar.solver = "CG_" + flavour[i];
            quarkPar.source = srcName;
            application.createModule<MFermion::GaugeProp>(qName[i], quarkPar);
            for (unsigned int mu = 0; mu < Nd; ++mu)
            {
                quarkPar.source = seqName[i][mu];
                seqName[i][mu]  = "Q_" + flavour[i] + "-" + seqName[i][mu];
                application.createModule<MFermion::GaugeProp>(seqName[i][mu], quarkPar);
            }
        }
```

# Temporal meson correlator



```
137        // contractions
138        MContraction::Meson::Par mesPar;
139        for (unsigned int i = 0; i < flavour.size(); ++i)
140        for (unsigned int j = i; j < flavour.size(); ++j)
141        {
142            mesPar.output = "mesons/Z2_" + flavour[i] + flavour[j];
143            mesPar.q1     = qName[i];
144            mesPar.q2     = qName[j];
145            mesPar.gammas = "all";
146            mesPar.sink   = "sink";
147            application.createModule<MContraction::Meson>("meson_Z2_"
148                                                         + std::to_string(t)
149                                                         + "_"
150                                                         + flavour[i]
151                                                         + flavour[j],
152                                                         mesPar);
153        }
```



```
181     // execution
182     application.saveParameterFile("meson2pt.xml");
183     application.run();
184
185     // epilogue
186     LOG(Message) << "Grid is finalizing now" << std::endl;
187     Grid_finalize();
188
189     return EXIT_SUCCESS;
190 }
```

$$\left\langle O_M(n)\,\bar{O}_M(l)\right\rangle_{DW} = -\,\mathrm{tr}\left[\Gamma_A\left(\tilde{D}^{-1}_{tov}(m_{f_2})\right)_{n,l}\Gamma_B\left(\tilde{D}^{-1\dagger}_{tov}(m_{f_1})\right)_{n,l}\right]$$

$\Gamma_A$ and $\Gamma_B$ combination is set by the string parameter mesPar.gammas. For example if one wants to compute the correlator for $(\Gamma_A, \Gamma_B) = (\gamma_5, \gamma_y)$ and $(\Gamma_A, \Gamma_B) = (\gamma_x, 1)$

One would write:

**mesPar.gammas = " (Gamma5 GammaY) (GammaX Identity) "**

"all" option in line145 means compute the correlator for all 256 combinations of gamma matrices

After set all modules, store all the parameters in "meson2pt.xml"
Start the execution with application.run()
Use Grid_finalize() to destroy all created objects and free memory

# Quark propagator

$$P_{L,R} = \frac{1 \pm \gamma_5}{2}$$

**Physical four dimensional fermion:**

$$q(x) = P_L \Psi(x,0) + P_R \Psi(x, L_s - 1); \quad \bar{q}(x) = \bar{\Psi}(x, L_s - 1) P_L + \bar{\Psi}(x,0) P_R$$

**Physical propagator:**

$$\langle \bar{q}(n) q(l) \rangle_{DW} = \frac{1}{Z_{DW}} \int \mathscr{D}\bar{\Psi} \mathscr{D}\Psi \mathscr{D}\bar{\Phi} \mathscr{D}\Phi \, \bar{q}(n) q(l) e^{-\bar{\Psi} D_{DW}(m) \Psi - \bar{\Phi} D_{DW}(1) \Phi}$$

$$= \frac{1}{1-m} \left( \left[ D_{tov}^{-1}(m) \right]_{l,n} - 1 \right) = \left[ \tilde{D}_{tov}^{-1}(m) \right]_{l,n}$$

$$D_{tov}(m) = \frac{1+m}{2} + \frac{1-m}{2} \gamma_5 \epsilon_{L_s}(H), \qquad \epsilon_{L_s}(H) = \frac{(1+H)^{L_s} - (1-H)^{L_s}}{(1+H)^{L_s} + (1-H)^{L_s}}, \qquad \lim_{L_s \to \infty} D_{tov} = D_{ov}$$

$$H = \gamma_5 D_{kernel} = \gamma_5 \frac{(b_5 + c_5) D_W(-M_5)}{2 + (b_5 - c_5) D_W(-M_5)}$$

$D_{kernel}$ **is Mobius domain wall kernel**

# Inversion of the Dirac operator

**The connection to the DWF operator**

$$\left[\tilde{D}_{tov}^{-1}(m)\right]_{n,l} = \left[\mathbb{P}^{\dagger} D_{DW}^{-1}(m)(-D_{-})\mathbb{P}^{\dagger}\right]_{n,l;0,L_s-1}, \qquad \mathcal{P} = \begin{pmatrix} P_L & P_R & 0 & \dots & 0 \\ 0 & P_L & P_R & \dots & 0 \\ & & \dots & & \\ 0 & 0 & \dots & P_L & P_R \\ P_R & 0 & \dots & 0 & P_L \end{pmatrix} \qquad P_{L,R} = \frac{1 \pm \gamma_5}{2}$$

**To determine the propagator, need to do the inversion of the Dirac operator, it is time consuming**

**Introduce source to just invert a part of the operator, e.g. point source** $\psi_0 = \delta_{n,n_0}\delta_{\alpha,\alpha_0}\delta_{a,a_0}$

$$\begin{aligned}\left[\tilde{D}_{tov}^{-1}(m)\right]_{n,l}\,[\psi_0]_l &= \left[\mathbb{P}^{\dagger} D_{DW}^{-1}(m)(-D_{-})\mathbb{P}^{\dagger}\right]_{n,l;0,L_s-1}\,[\psi_0]_l \\ &= \left[\mathbb{P}^{\dagger} D_{DW}^{-1}(m)\right]_{n,l;0,s}\,\left[(-D_{-})\mathbb{P}^{\dagger}\psi_0\right]_{l;s,L_s-1} \\ &= \left[\mathbb{P}^{\dagger} \boxed{D_{DW}^{-1}(m)\right]_{n,l;0,s}\,\left[\Psi_{DW,0}\right]_{l;s,L_s-1}} \end{aligned}$$

**Five dimensional point source**

**Solve** $\boxed{G = D_{DW}^{-1}(m)\Psi_{DW,0}}$ **using conjugate gradient (CG)**

**Then, project G back to four dimensions by** $\mathbb{P}^{\dagger}G$ **to get** $\left[\tilde{D}_{tov}^{-1}(m)\right]_{n,n_0}$

# Red black/even-odd preconditioning

$$\left[\tilde{D}^{-1}_{tov}(m)\right]_{n,l}[\psi_0]_l = \left[\tilde{D}^{-1}_{tov}(m)\right]_{n,n_0} = \left[\mathbb{P}^{\dagger}D^{-1}_{DW}(m)\right]_{n,l;0,s}\left[\Psi_{DW,0}\right]_{l;s,L_s-1}$$

- **Only calculate one column of the truncated overlap operator, reduced the time**

$$G = D^{-1}_{DW}(m)\Psi_{DW,0}$$

- **Need to do the above inversion for 12 sources which is all combinations of Dirac and color indices**

- **To further reduce the computational time, introduce red black preconditioning method**

**Assign for each space-time point of the four dimensional lattice a parity** $p = \{r, b\}$ **with**

$$p = (x+y+z+t) \bmod 2, \qquad 0 \to r, 1 \to b$$

$$D_{DW} = \begin{pmatrix} M_{ee} & M_{eo} \\ M_{oe} & M_{oo} \end{pmatrix}$$

## Red black/even-odd preconditioning

**Shur decomposition**

$$D_{DW} = \begin{pmatrix} M_{ee} & M_{eo} \\ M_{oe} & M_{oo} \end{pmatrix}$$

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

$$D_{DW}\psi = \eta$$

$$DU\psi = L^{-1}\eta$$

$$\begin{pmatrix} M_{ee} & M_{eo} \\ 0 & D_{oo} \end{pmatrix} \begin{pmatrix} \psi_e \\ \psi_o \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -M_{oe}M_{ee}^{-1} & 1 \end{pmatrix} \begin{pmatrix} \eta_e \\ \eta_o \end{pmatrix}$$

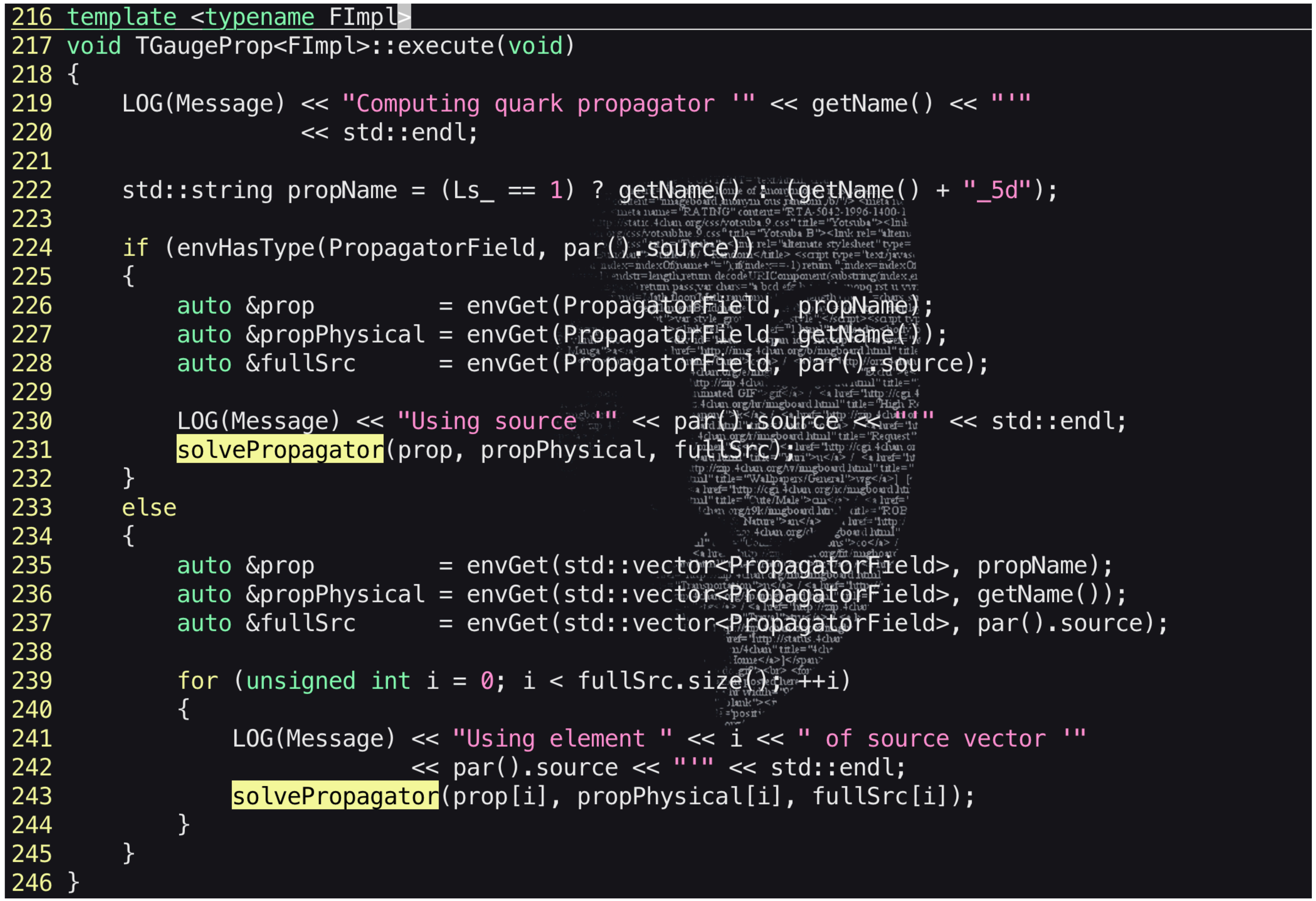
$$\begin{pmatrix} M_{ee}\psi_e + M_{eo}\psi_o \\ D_{oo}\psi_o \end{pmatrix} = \begin{pmatrix} \eta_e \\ \eta_o - M_{oe}M_{ee}^{-1}\eta_e \end{pmatrix} = \begin{pmatrix} \eta_e \\ \eta'_o \end{pmatrix} \Rightarrow D_{oo}\psi_o = \eta'_o$$

**Solve** $D_{oo}^{\dagger}D_{oo}\boxed{\psi_o} = D_{oo}^{\dagger}\eta'_o$

**Put $\psi_o$ into** $\boxed{\psi_e = M_{ee}^{-1}(\eta_e - M_{eo}\psi_o)}$ **to get $\psi_e$**

# Temporal meson correlator

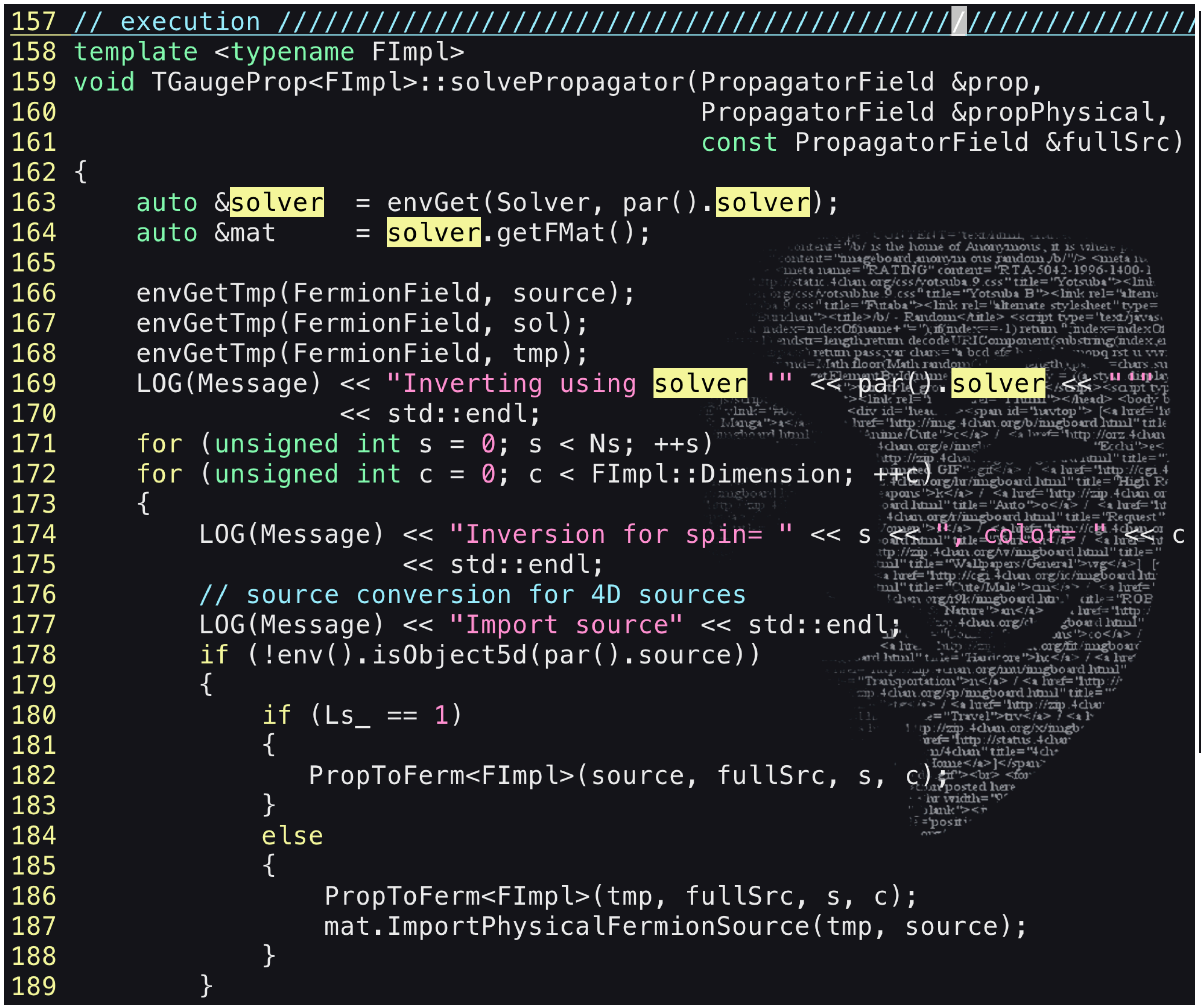
Hadrons/Modules/MFermion/GaugeProp.hpp



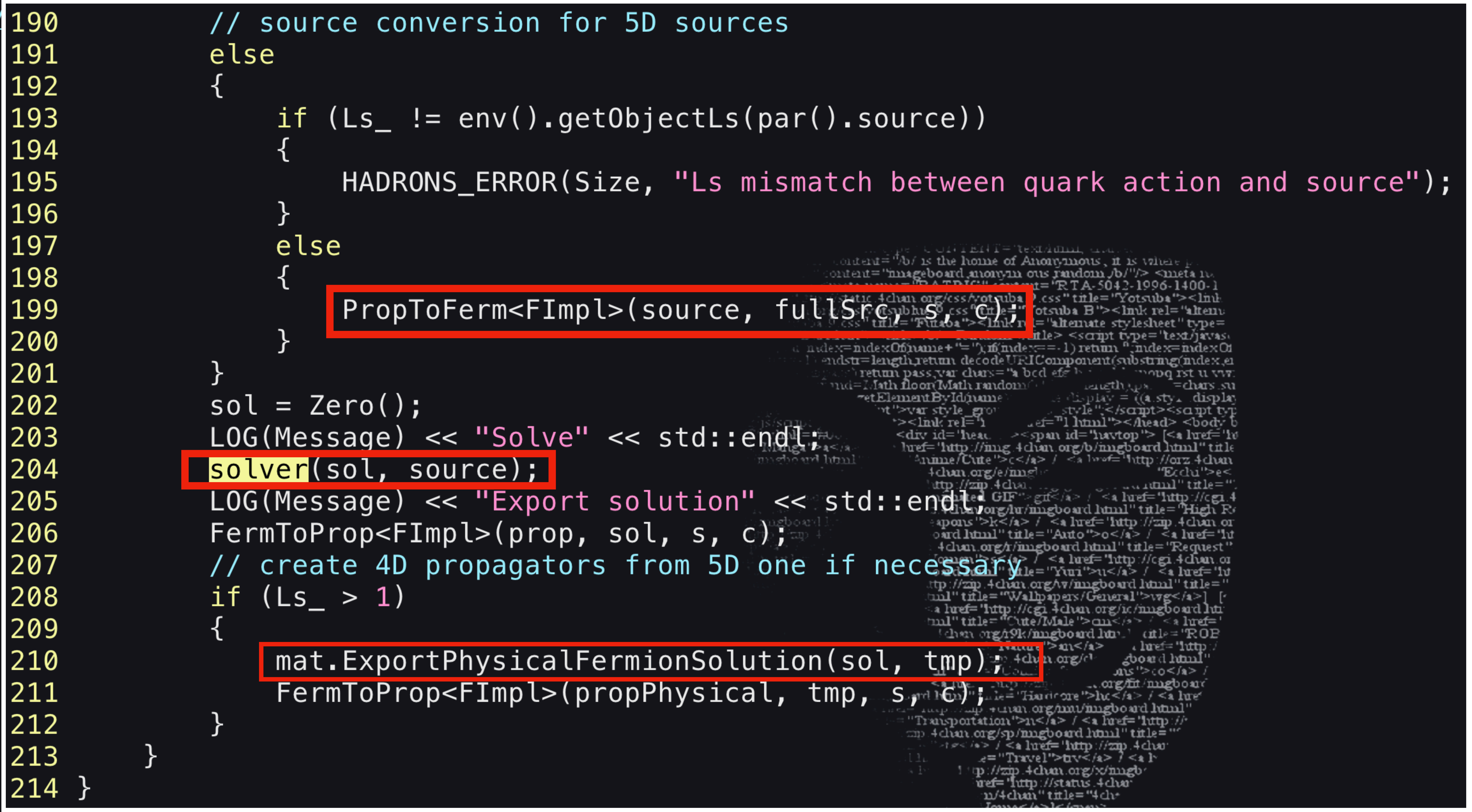
```
template <typename FImpl>
void TGaugeProp<FImpl>::execute(void)
{
    LOG(Message) << "Computing quark propagator '" << getName() << "'"
                 << std::endl;

    std::string propName = (Ls_ == 1) ? getName() : (getName() + "_5d");

    if (envHasType(PropagatorField, par().source))
    {
        auto &prop         = envGet(PropagatorField, propName);
        auto &propPhysical = envGet(PropagatorField, getName());
        auto &fullSrc      = envGet(PropagatorField, par().source);

        LOG(Message) << "Using source '" << par().source << "'" << std::endl;
        solvePropagator(prop, propPhysical, fullSrc);
    }
    else
    {
        auto &prop         = envGet(std::vector<PropagatorField>, propName);
        auto &propPhysical = envGet(std::vector<PropagatorField>, getName());
        auto &fullSrc      = envGet(std::vector<PropagatorField>, par().source);

        for (unsigned int i = 0; i < fullSrc.size(); ++i)
        {
            LOG(Message) << "Using element " << i << " of source vector '"
                         << par().source << "'" << std::endl;
            solvePropagator(prop[i], propPhysical[i], fullSrc[i]);
        }
    }
}
```

Using red black preconditioning and the conjugate gradient method to solve the propagator

# Hadrons/Modules/MFermion/GaugeProp.hpp



```
// execution ///////////////////////////////////////////////////////////////
template <typename FImpl>
void TGaugeProp<FImpl>::solvePropagator(PropagatorField &prop,
                                        PropagatorField &propPhysical,
                                        const PropagatorField &fullSrc)
{
    auto &solver  = envGet(Solver, par().solver);
    auto &mat     = solver.getFMat();

    envGetTmp(FermionField, source);
    envGetTmp(FermionField, sol);
    envGetTmp(FermionField, tmp);
    LOG(Message) << "Inverting using solver '" << par().solver << "'"
                 << std::endl;
    for (unsigned int s = 0; s < Ns; ++s)
    for (unsigned int c = 0; c < FImpl::Dimension; ++c)
    {
        LOG(Message) << "Inversion for spin= " << s << ", color= " << c
                     << std::endl;
        // source conversion for 4D sources
        LOG(Message) << "Import source" << std::endl;
        if (!env().isObject5d(par().source))
        {
            if (Ls_ == 1)
            {
               PropToFerm<FImpl>(source, fullSrc, s, c);
            }
            else
            {
                PropToFerm<FImpl>(tmp, fullSrc, s, c);
                mat.ImportPhysicalFermionSource(tmp, source);
            }
        }
```



```
        // source conversion for 5D sources
        else
        {
            if (Ls_ != env().getObjectLs(par().source))
            {
                HADRONS_ERROR(Size, "Ls mismatch between quark action and source");
            }
            else
            {
                PropToFerm<FImpl>(source, fullSrc, s, c);
            }
        }
        sol = Zero();
        LOG(Message) << "Solve" << std::endl;
        solver(sol, source);
        LOG(Message) << "Export solution" << std::endl;
        FermToProp<FImpl>(prop, sol, s, c);
        // create 4D propagators from 5D one if necessary
        if (Ls_ > 1)
        {
            mat.ExportPhysicalFermionSolution(sol, tmp);
            FermToProp<FImpl>(propPhysical, tmp, s, c);
        }
    }
}
```

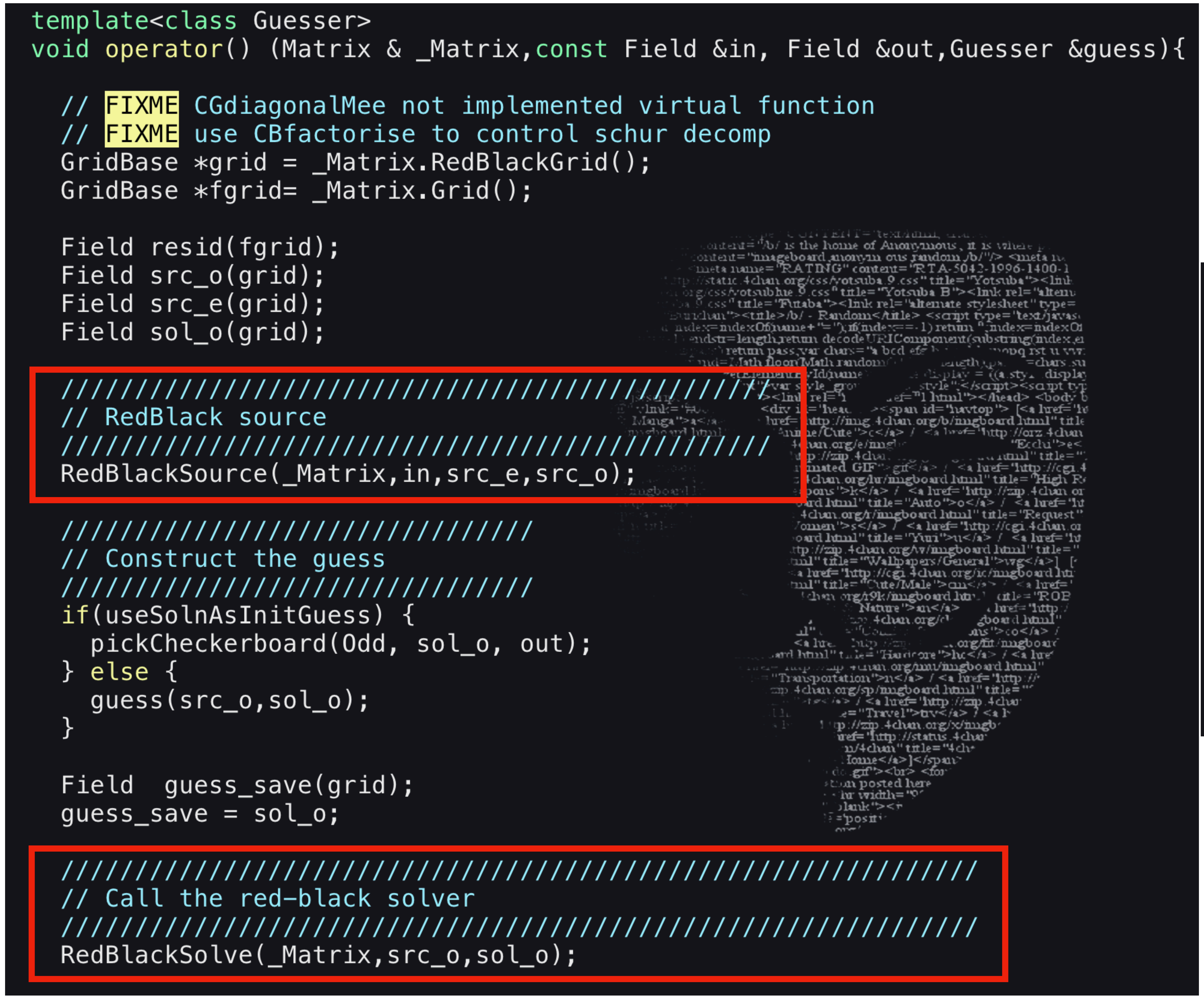
$$\langle \bar{q}(n) q(l) \rangle_{DW} = \left[\tilde{D}^{-1}_{tov}(m)\right]_{l,n}$$

$$\left[\tilde{D}^{-1}_{tov}(m)\right]_{n,l} \boxed{[\psi_0]_l} = \left[\tilde{D}^{-1}_{tov}(m)\right]_{n,n_0} = \left[\mathbb{P}^\dagger D^{-1}_{DW}(m)\right]_{n,l;0,s} \boxed{\left[\Psi_{DW,0}\right]_{l;s,L_s-1}}$$

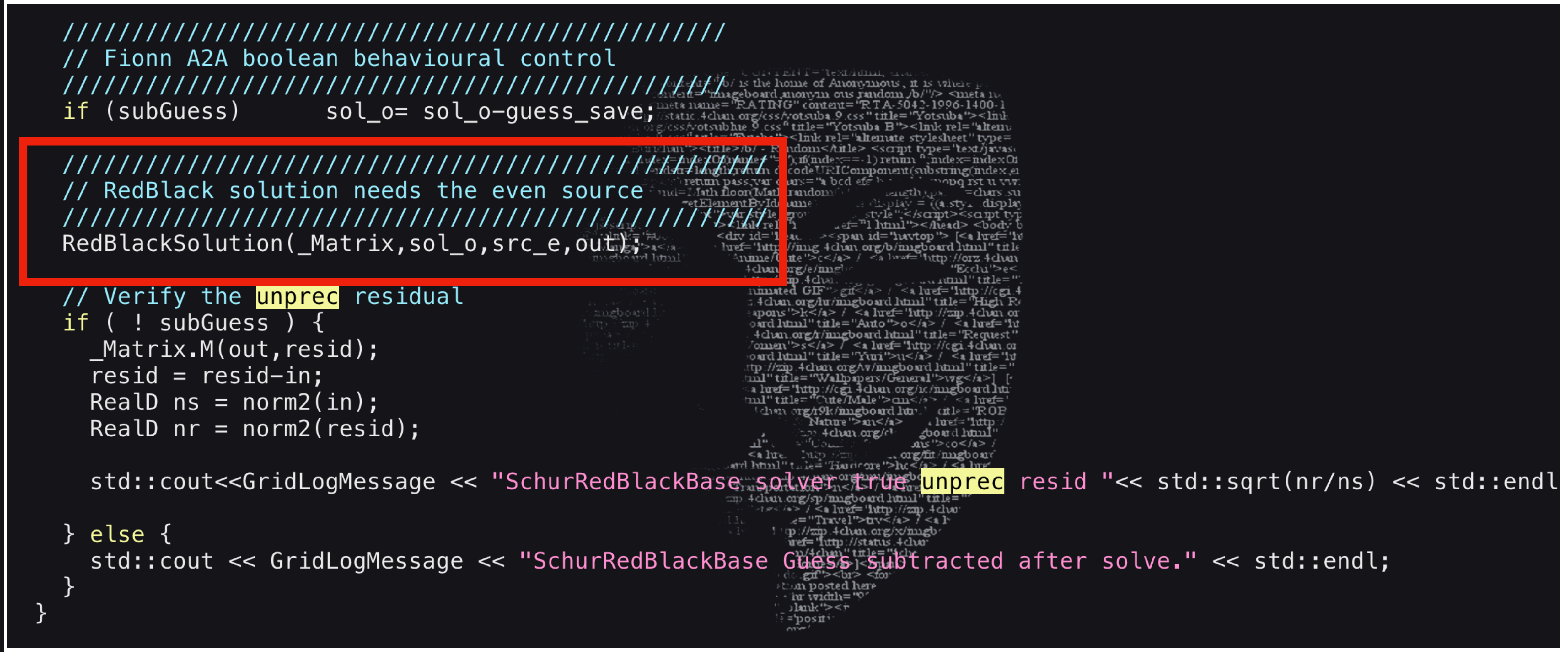
**Solve** $\boxed{G = D^{-1}_{DW}(m)\Psi_{DW,0}}$ **using CG**

**Then project G back to four dimensions by** $\mathbb{P}^\dagger G$ **to get** $\left[\tilde{D}^{-1}_{tov}(m)\right]_{n,n_0}$

# Grid/algorithms/iterative/SchurRedBlack.h

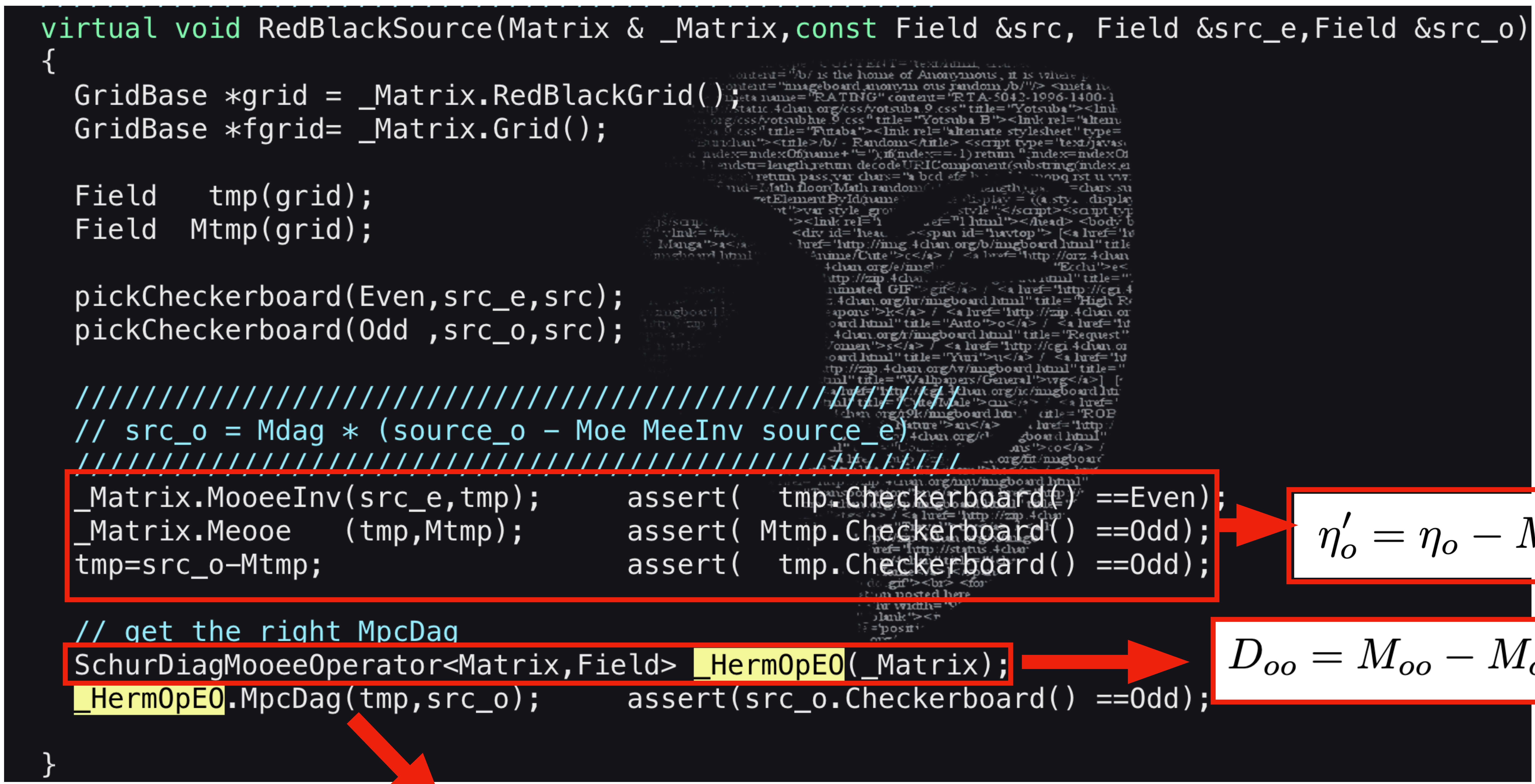


```
template<class Guesser>
void operator() (Matrix & _Matrix,const Field &in, Field &out,Guesser &guess){

  // FIXME CGdiagonalMee not implemented virtual function
  // FIXME use CBfactorise to control schur decomp
  GridBase *grid = _Matrix.RedBlackGrid();
  GridBase *fgrid= _Matrix.Grid();

  Field resid(fgrid);
  Field src_o(grid);
  Field src_e(grid);
  Field sol_o(grid);

  ////////////////////////////////////////////////
  // RedBlack source
  ////////////////////////////////////////////////
  RedBlackSource(_Matrix,in,src_e,src_o);

  ////////////////////////////////
  // Construct the guess
  ////////////////////////////////
  if(useSolnAsInitGuess) {
    pickCheckerboard(Odd, sol_o, out);
  } else {
    guess(src_o,sol_o);
  }

  Field  guess_save(grid);
  guess_save = sol_o;

  ///////////////////////////////////////////////////////////////
  // Call the red-black solver
  ///////////////////////////////////////////////////////////////
  RedBlackSolve(_Matrix,src_o,sol_o);
```



```
  ////////////////////////////////////////////////
  // Fionn A2A boolean behavioural control
  ////////////////////////////////////////////////
  if (subGuess)      sol_o= sol_o-guess_save;

  //////////////////////////////////////////////////
  // RedBlack solution needs the even source
  //////////////////////////////////////////////////
  RedBlackSolution(_Matrix,sol_o,src_e,out);

  // Verify the unprec residual
  if ( ! subGuess ) {
    _Matrix.M(out,resid);
    resid = resid-in;
    RealD ns = norm2(in);
    RealD nr = norm2(resid);

    std::cout<<GridLogMessage << "SchurRedBlackBase solver true unprec resid "<< std::sqrt(nr/ns) << std::endl

  } else {
    std::cout << GridLogMessage << "SchurRedBlackBase Guess subtracted after solve." << std::endl;
  }
}
```

# Grid/algorithms/iterative/SchurRedBlack.h

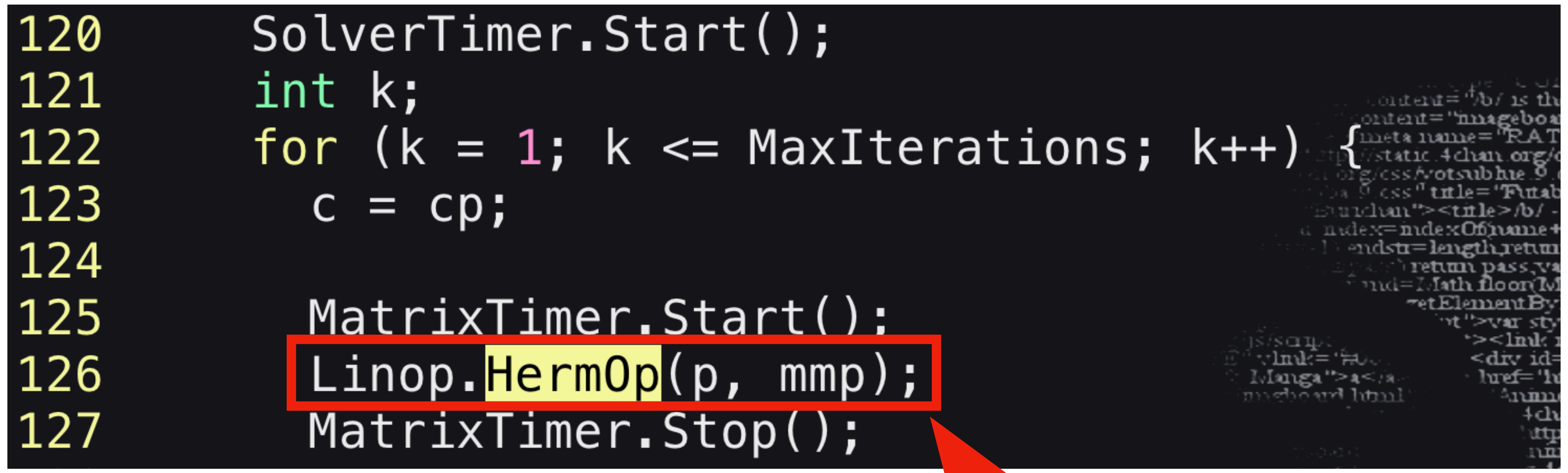


```
  virtual void RedBlackSource(Matrix & _Matrix,const Field &src, Field &src_e,Field &src_o)
  {
    GridBase *grid = _Matrix.RedBlackGrid();
    GridBase *fgrid= _Matrix.Grid();

    Field   tmp(grid);
    Field  Mtmp(grid);

    pickCheckerboard(Even,src_e,src);
    pickCheckerboard(Odd ,src_o,src);

    /////////////////////////////////////////////////////
    // src_o = Mdag * (source_o - Moe MeeInv source_e)
    /////////////////////////////////////////////////////
    _Matrix.MooeeInv(src_e,tmp);     assert(  tmp.Checkerboard() ==Even);
    _Matrix.Meooe   (tmp,Mtmp);      assert( Mtmp.Checkerboard() ==Odd);
    tmp=src_o-Mtmp;                  assert(  tmp.Checkerboard() ==Odd);

    // get the right MpcDag
    SchurDiagMooeeOperator<Matrix,Field> _HermOpEO(_Matrix);
    _HermOpEO.MpcDag(tmp,src_o);     assert(src_o.Checkerboard() ==Odd);

  }
```

$$\eta_o' = \eta_o - M_{oe} M_{ee}^{-1} \eta_e$$

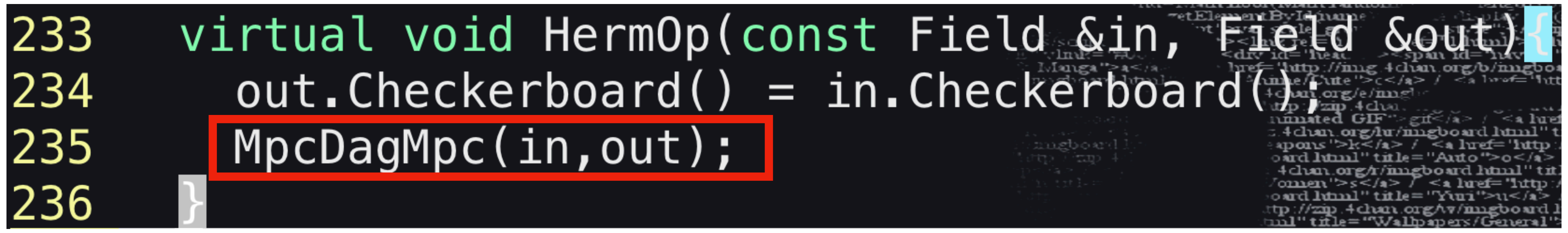
$$D_{oo} = M_{oo} - M_{oe} M_{ee}^{-1} M_{eo}$$

$$D_{oo}^{\dagger} \eta_o'$$

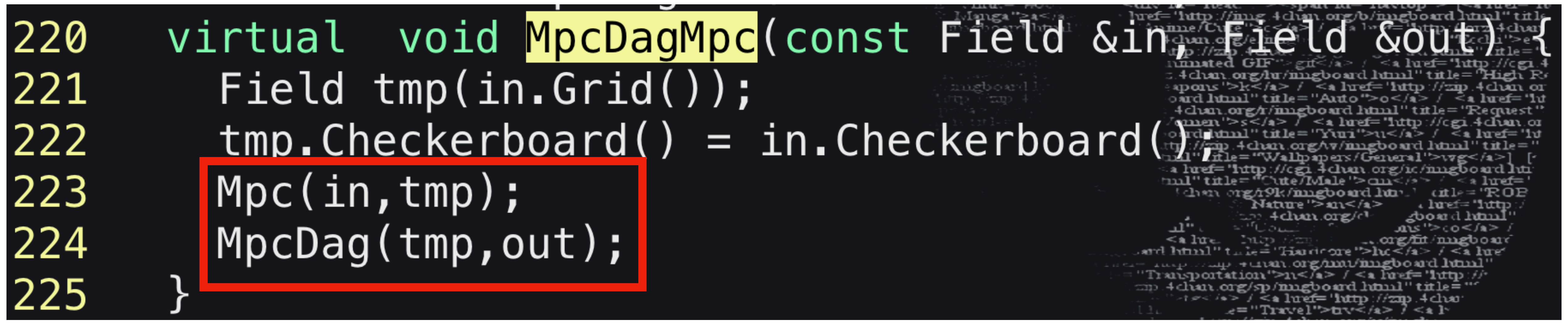


```
    SolverTimer.Start();
    int k;
    for (k = 1; k <= MaxIterations; k++) {
      c = cp;

      MatrixTimer.Start();
      Linop.HermOp(p, mmp);
      MatrixTimer.Stop();
```

$$mmp = D_{oo}^{\dagger} D_{oo} P_{o_{(i)}}$$

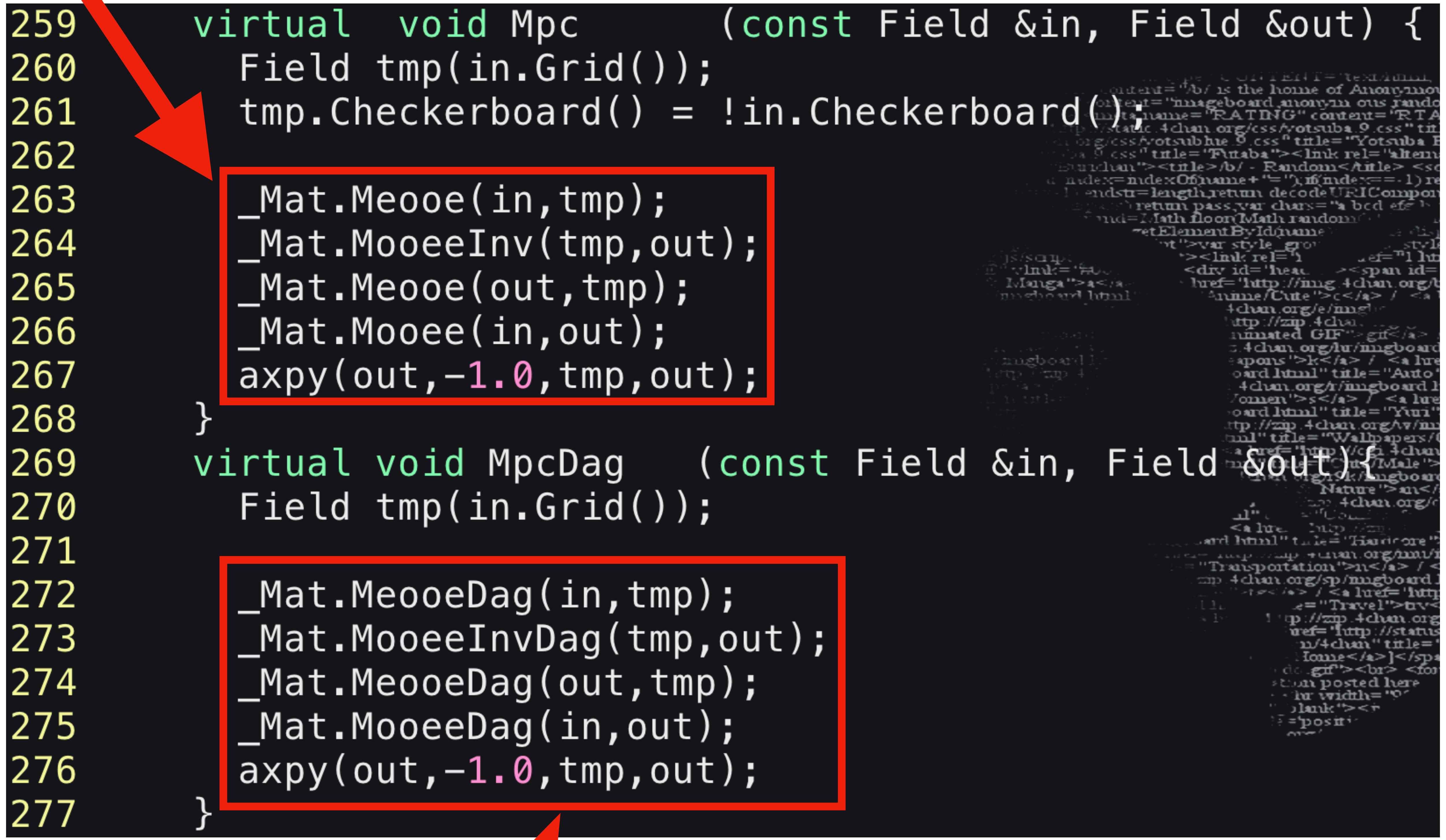


```
  virtual void HermOp(const Field &in, Field &out){
    out.Checkerboard() = in.Checkerboard();
    MpcDagMpc(in,out);
  }
```



```
  virtual  void MpcDagMpc(const Field &in, Field &out) {
    Field tmp(in.Grid());
    tmp.Checkerboard() = in.Checkerboard();
    Mpc(in,tmp);
    MpcDag(tmp,out);
  }
```

$$out = -1 \times tmp + out = -1 \times (M_{oe} M_{ee}^{-1} M_{eo} P_{o_{(i)}}) + M_{oo} P_{o_{(i)}} = D_{oo} P_{o_{(i)}}$$

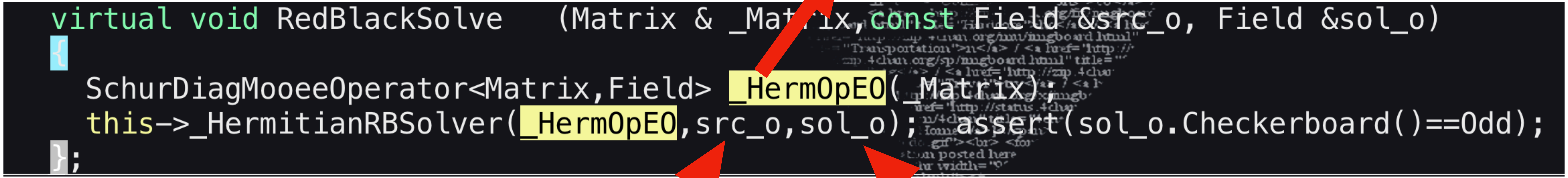


```
    virtual  void Mpc      (const Field &in, Field &out) {
      Field tmp(in.Grid());
      tmp.Checkerboard() = !in.Checkerboard();

      _Mat.Meooe(in,tmp);
      _Mat.MooeeInv(tmp,out);
      _Mat.Meooe(out,tmp);
      _Mat.Mooee(in,out);
      axpy(out,-1.0,tmp,out);
    }
    virtual void MpcDag   (const Field &in, Field &out){
      Field tmp(in.Grid());

      _Mat.MeooeDag(in,tmp);
      _Mat.MooeeInvDag(tmp,out);
      _Mat.MeooeDag(out,tmp);
      _Mat.MooeeDag(in,out);
      axpy(out,-1.0,tmp,out);
    }
```

$$out = D_{oo}^{\dagger} D_{oo} P_{o_{(i)}}$$

# Grid/algorithms/iterative/SchurRedBlack.h

$$\boxed{D_{oo} = M_{oo} - M_{oe} M_{ee}^{-1} M_{eo}}$$



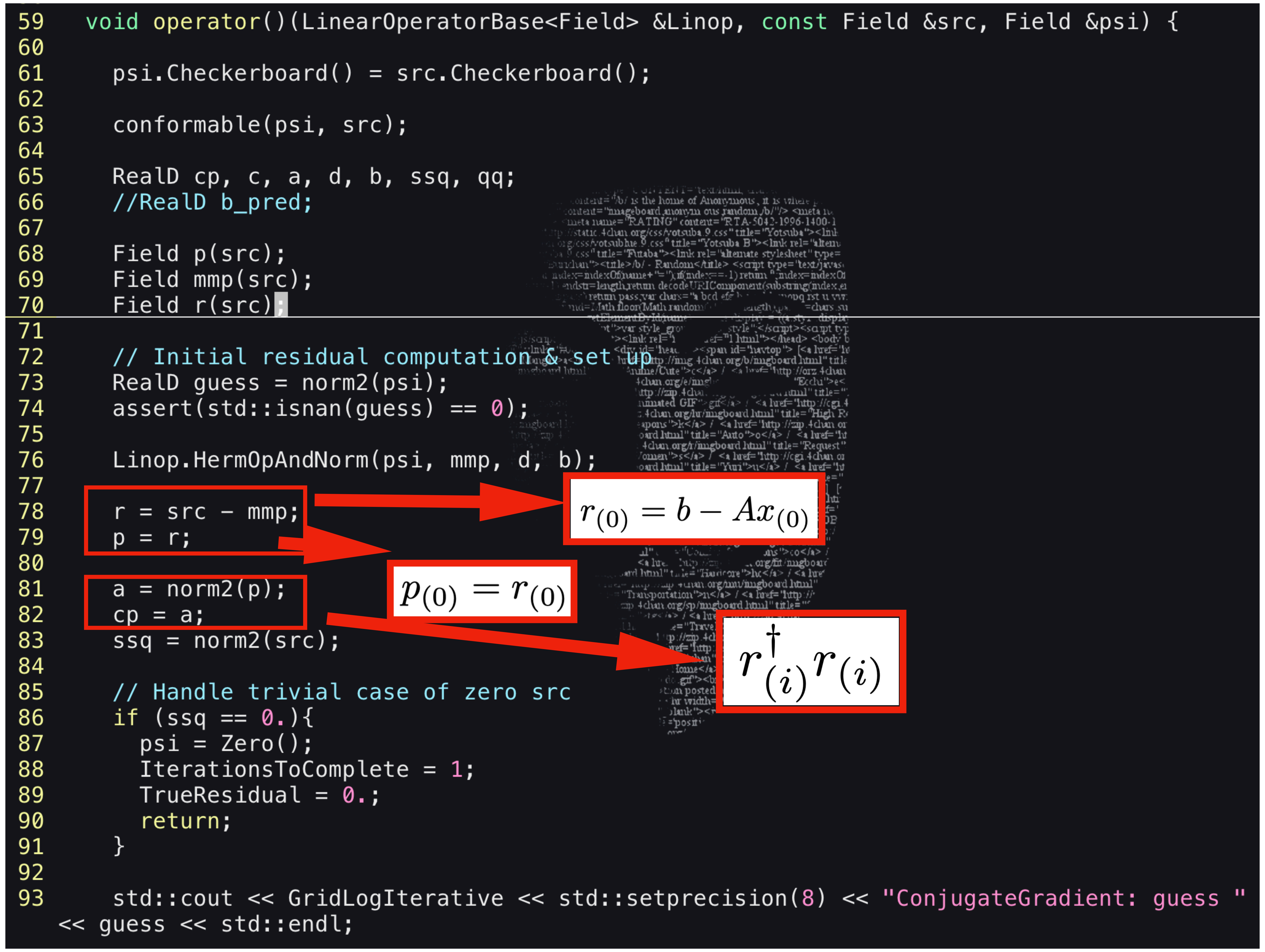
```
  virtual void RedBlackSolve   (Matrix & _Matrix,const Field &src_o, Field &sol_o)
  {
    SchurDiagMooeeOperator<Matrix,Field> _HermOpEO(_Matrix);
    this->_HermitianRBSolver(_HermOpEO,src_o,sol_o);  assert(sol_o.Checkerboard()==Odd);
  };
```

$$\boxed{\text{src\_o} = D_{oo}^{\dagger} \eta_o'} \qquad \boxed{\text{sol\_o} = \psi_o}$$

$$\boxed{\textbf{CG:} \quad D_{oo}^{\dagger} D_{oo} \psi_o = D_{oo}^{\dagger} \eta_o'}$$

# Grid/algorithms/iterative/ConjugateGradient.h

## CG:

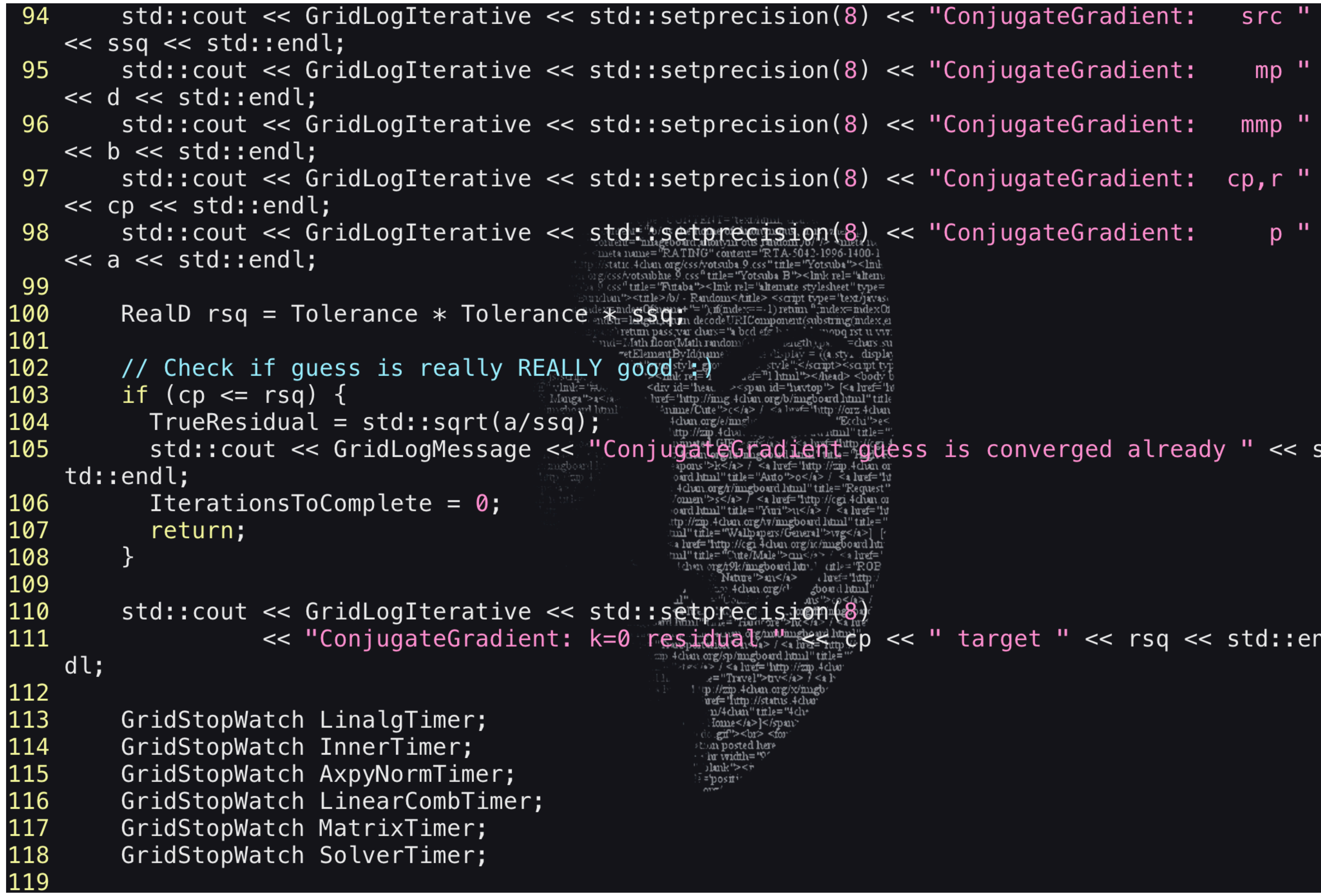


```cpp
  void operator()(LinearOperatorBase<Field> &Linop, const Field &src, Field &psi) {

    psi.Checkerboard() = src.Checkerboard();

    conformable(psi, src);

    RealD cp, c, a, d, b, ssq, qq;
    //RealD b_pred;

    Field p(src);
    Field mmp(src);
    Field r(src);

    // Initial residual computation & set up
    RealD guess = norm2(psi);
    assert(std::isnan(guess) == 0);

    Linop.HermOpAndNorm(psi, mmp, d, b);

    r = src - mmp;
    p = r;

    a = norm2(p);
    cp = a;
    ssq = norm2(src);

    // Handle trivial case of zero src
    if (ssq == 0.){
      psi = Zero();
      IterationsToComplete = 1;
      TrueResidual = 0.;
      return;
    }

    std::cout << GridLogIterative << std::setprecision(8) << "ConjugateGradient: guess " << guess << std::endl;
```

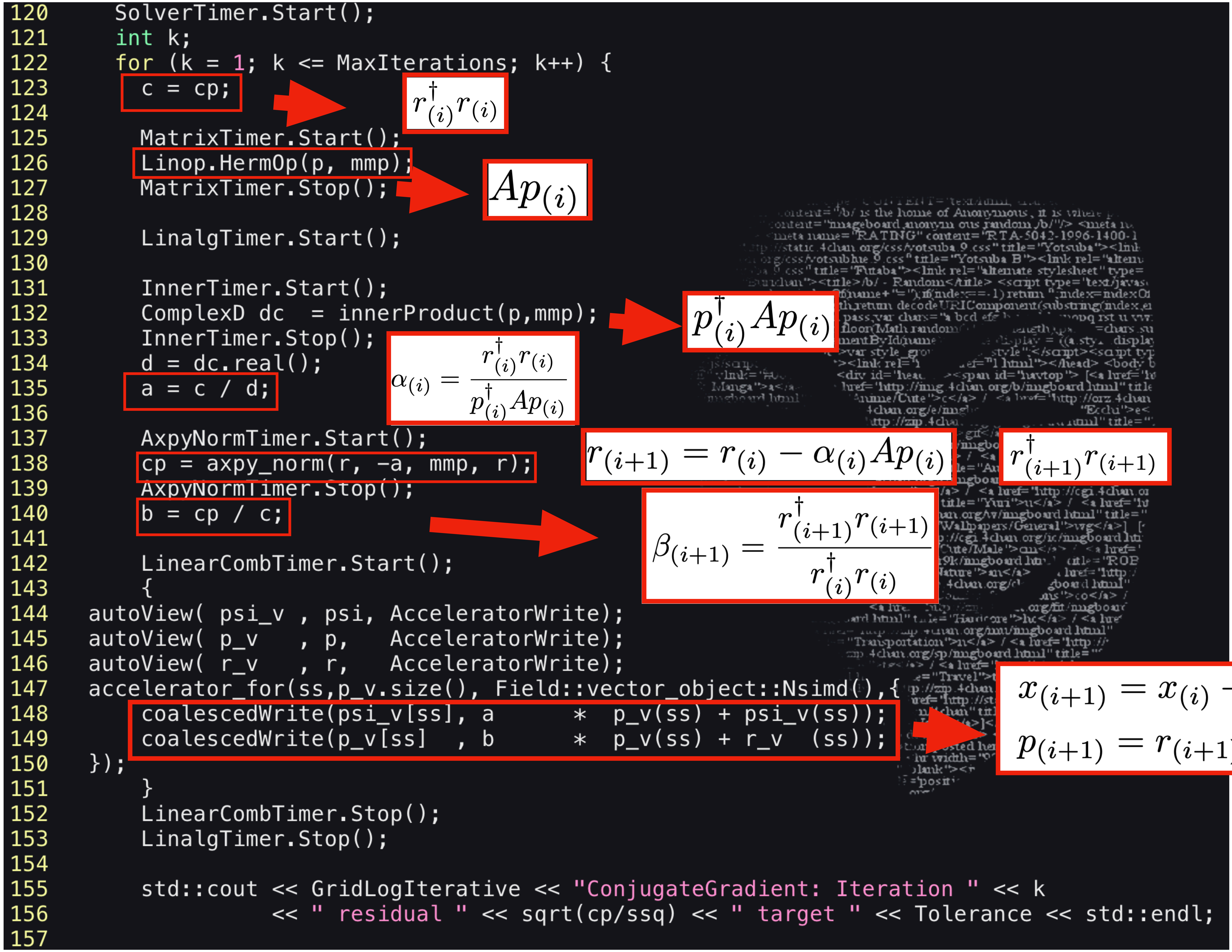
$r_{(0)} = b - Ax_{(0)}$

$p_{(0)} = r_{(0)}$

$r^{\dagger}_{(i)} r_{(i)}$



```cpp
    std::cout << GridLogIterative << std::setprecision(8) << "ConjugateGradient:   src " << ssq << std::endl;
    std::cout << GridLogIterative << std::setprecision(8) << "ConjugateGradient:    mp " << d << std::endl;
    std::cout << GridLogIterative << std::setprecision(8) << "ConjugateGradient:   mmp " << b << std::endl;
    std::cout << GridLogIterative << std::setprecision(8) << "ConjugateGradient:  cp,r " << cp << std::endl;
    std::cout << GridLogIterative << std::setprecision(8) << "ConjugateGradient:     p " << a << std::endl;

    RealD rsq = Tolerance * Tolerance * ssq;

    // Check if guess is really REALLY good :)
    if (cp <= rsq) {
      TrueResidual = std::sqrt(a/ssq);
      std::cout << GridLogMessage << "ConjugateGradient guess is converged already " << std::endl;
      IterationsToComplete = 0;
      return;
    }

    std::cout << GridLogIterative << std::setprecision(8)
              << "ConjugateGradient: k=0 residual " << cp << " target " << rsq << std::endl;

    GridStopWatch LinalgTimer;
    GridStopWatch InnerTimer;
    GridStopWatch AxpyNormTimer;
    GridStopWatch LinearCombTimer;
    GridStopWatch MatrixTimer;
    GridStopWatch SolverTimer;
```



```
    SolverTimer.Start();
    int k;
    for (k = 1; k <= MaxIterations; k++) {
      c = cp;

      MatrixTimer.Start();
      Linop.HermOp(p, mmp);
      MatrixTimer.Stop();

      LinalgTimer.Start();

      InnerTimer.Start();
      ComplexD dc  = innerProduct(p,mmp);
      InnerTimer.Stop();
      d = dc.real();
      a = c / d;

      AxpyNormTimer.Start();
      cp = axpy_norm(r, -a, mmp, r);
      AxpyNormTimer.Stop();
      b = cp / c;

      LinearCombTimer.Start();
      {
  autoView( psi_v , psi, AcceleratorWrite);
  autoView( p_v   , p,   AcceleratorWrite);
  autoView( r_v   , r,   AcceleratorWrite);
  accelerator_for(ss,p_v.size(), Field::vector_object::Nsimd(),{
      coalescedWrite(psi_v[ss], a      *  p_v(ss) + psi_v(ss));
      coalescedWrite(p_v[ss]  , b      *  p_v(ss) + r_v  (ss));
  });
      }
      LinearCombTimer.Stop();
      LinalgTimer.Stop();

      std::cout << GridLogIterative << "ConjugateGradient: Iteration " << k
                << " residual " << sqrt(cp/ssq) << " target " << Tolerance << std::endl;
```

$$r_{(i)}^\dagger r_{(i)}$$

$$Ap_{(i)}$$

$$p_{(i)}^\dagger A p_{(i)}$$

$$\alpha_{(i)} = \frac{r_{(i)}^\dagger r_{(i)}}{p_{(i)}^\dagger A p_{(i)}}$$

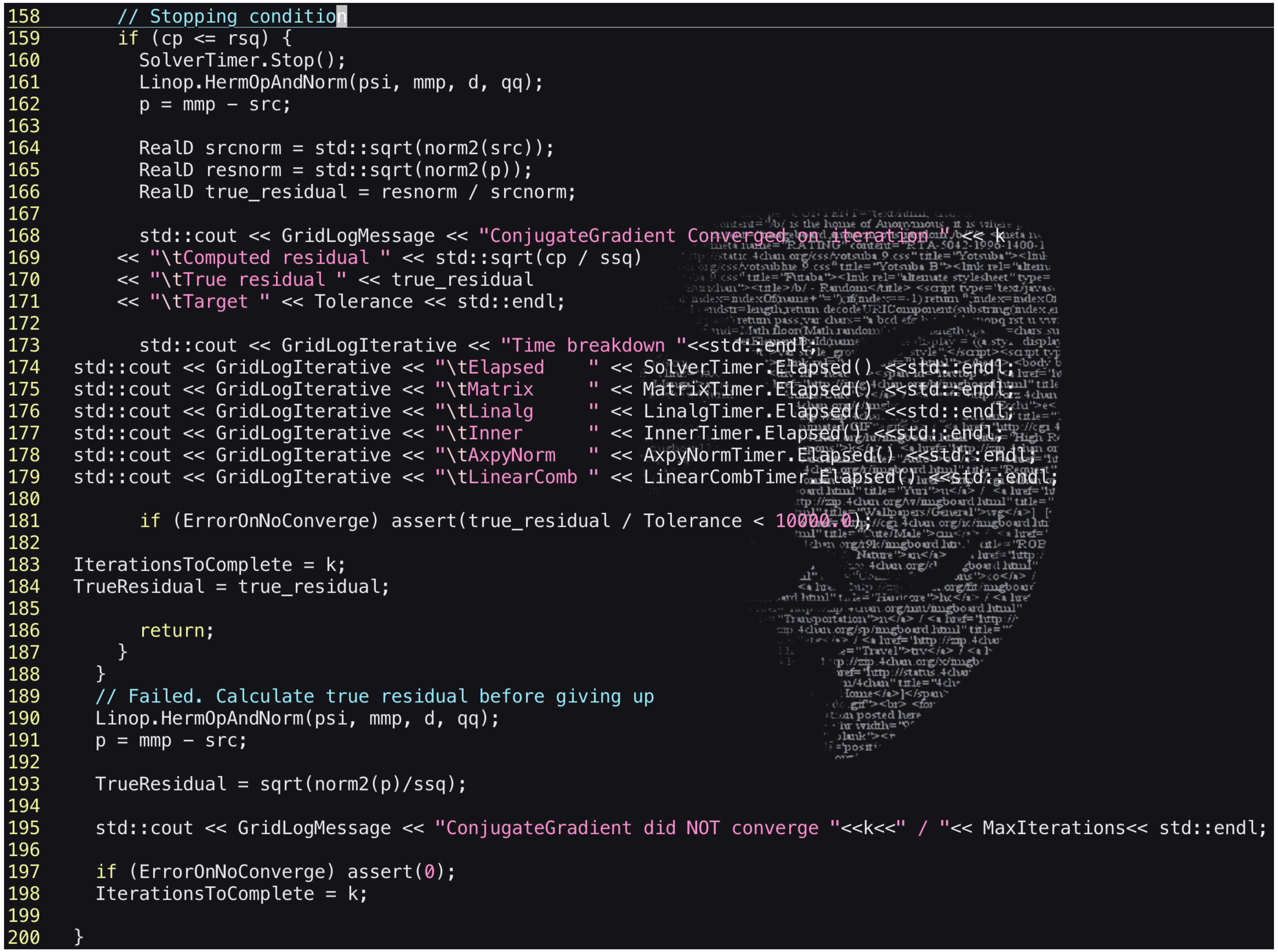
$$r_{(i+1)} = r_{(i)} - \alpha_{(i)} A p_{(i)}$$

$$r_{(i+1)}^\dagger r_{(i+1)}$$

$$\beta_{(i+1)} = \frac{r_{(i+1)}^\dagger r_{(i+1)}}{r_{(i)}^\dagger r_{(i)}}$$

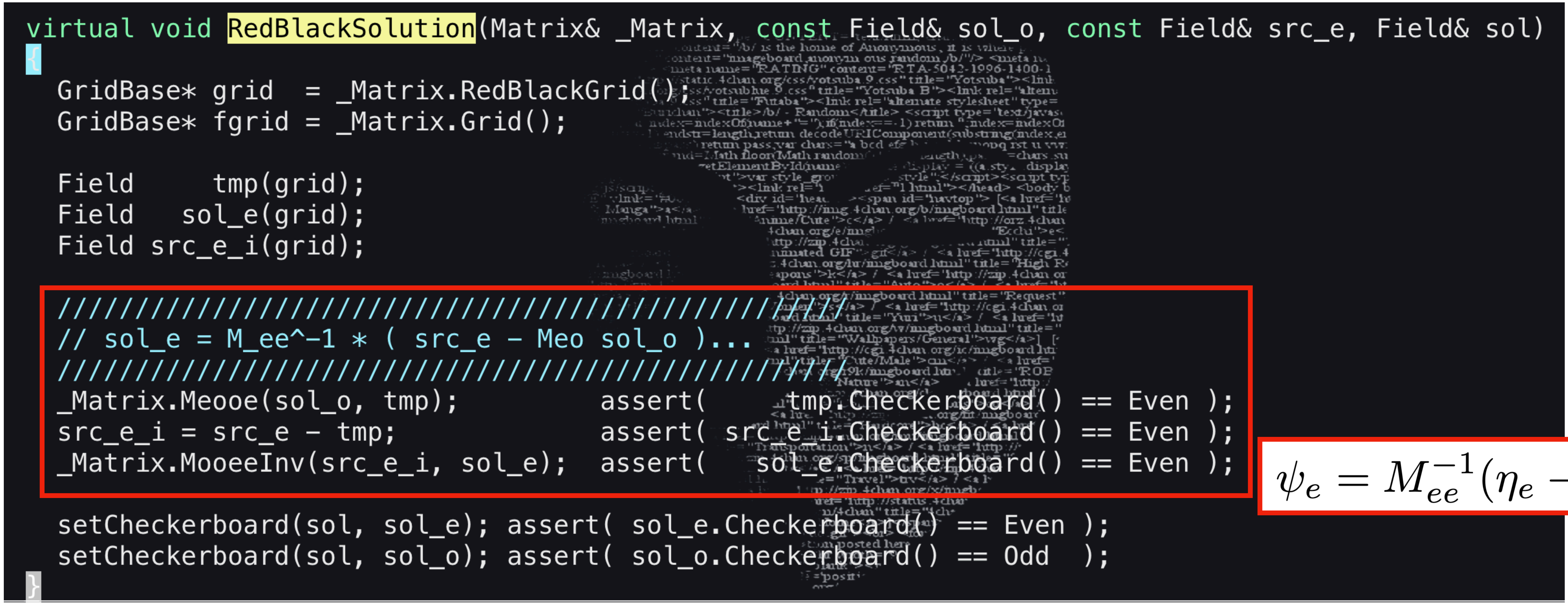
$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} p_{(i)} \,,$$
$$p_{(i+1)} = r_{(i+1)} + \beta_{(i+1)} p_{(i)} \,,$$

# Grid/algorithms/iterative/ConjugateGradient.h



```
    // Stopping condition
    if (cp <= rsq) {
      SolverTimer.Stop();
      Linop.HermOpAndNorm(psi, mmp, d, qq);
      p = mmp - src;

      RealD srcnorm = std::sqrt(norm2(src));
      RealD resnorm = std::sqrt(norm2(p));
      RealD true_residual = resnorm / srcnorm;

      std::cout << GridLogMessage << "ConjugateGradient Converged on iteration " << k
    << "\tComputed residual " << std::sqrt(cp / ssq)
    << "\tTrue residual " << true_residual
    << "\tTarget " << Tolerance << std::endl;

      std::cout << GridLogIterative << "Time breakdown "<<std::endl;
std::cout << GridLogIterative << "\tElapsed    " << SolverTimer.Elapsed() <<std::endl;
std::cout << GridLogIterative << "\tMatrix     " << MatrixTimer.Elapsed() <<std::endl;
std::cout << GridLogIterative << "\tLinalg     " << LinalgTimer.Elapsed() <<std::endl;
std::cout << GridLogIterative << "\tInner      " << InnerTimer.Elapsed() <<std::endl;
std::cout << GridLogIterative << "\tAxpyNorm   " << AxpyNormTimer.Elapsed() <<std::endl;
std::cout << GridLogIterative << "\tLinearComb " << LinearCombTimer.Elapsed() <<std::endl;

      if (ErrorOnNoConverge) assert(true_residual / Tolerance < 10000.0);

IterationsToComplete = k;
TrueResidual = true_residual;

      return;
    }
  }
  // Failed. Calculate true residual before giving up
  Linop.HermOpAndNorm(psi, mmp, d, qq);
  p = mmp - src;

  TrueResidual = sqrt(norm2(p)/ssq);

  std::cout << GridLogMessage << "ConjugateGradient did NOT converge "<<k<<" / "<< MaxIterations<< std::endl;

  if (ErrorOnNoConverge) assert(0);
  IterationsToComplete = k;

}
```

# Grid/algorithms/iterative/SchurRedBlack.h



```
virtual void RedBlackSolution(Matrix& _Matrix, const Field& sol_o, const Field& src_e, Field& sol)
{
  GridBase* grid  = _Matrix.RedBlackGrid();
  GridBase* fgrid = _Matrix.Grid();

  Field     tmp(grid);
  Field   sol_e(grid);
  Field src_e_i(grid);

  ///////////////////////////////////////////////////
  // sol_e = M_ee^-1 * ( src_e - Meo sol_o )...
  ///////////////////////////////////////////////////
  _Matrix.Meooe(sol_o, tmp);         assert(     tmp.Checkerboard() == Even );
  src_e_i = src_e - tmp;             assert( src_e_i.Checkerboard() == Even );
  _Matrix.MooeeInv(src_e_i, sol_e);  assert(   sol_e.Checkerboard() == Even );

  setCheckerboard(sol, sol_e); assert( sol_e.Checkerboard() == Even );
  setCheckerboard(sol, sol_o); assert( sol_o.Checkerboard() == Odd  );
}
```

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

**Solve** $D_{oo}^{\dagger} D_{oo} \boxed{\psi_o} = D_{oo}^{\dagger} \eta_o'$

**Put** $\psi_o$ **into** $\boxed{\psi_e = M_{ee}^{-1}(\eta_e - M_{eo}\psi_o)}$ **to get** $\psi_e$

# Temporal meson correlator

$$\left\langle O_M(n)\,\bar{O}_M(l)\right\rangle_{DW} = \left\langle \underset{a_1}{\bar{q}^{f_1}(n)_{\alpha_1}}\,\Gamma_{\alpha_1\beta_1}\,\underset{a_1}{q^{f_2}(n)_{\beta_1}}\,\underset{a_2}{\bar{q}^{f_2}(l)_{\alpha_2}}\,\Gamma_{\alpha_2\beta_2}\,\underset{a_2}{q^{f_1}(l)_{\beta_2}}\right\rangle_{DW} = -\Gamma_{\alpha_1\beta_1}\Gamma_{\alpha_2\beta_2}\left\langle \underset{a_2}{\bar{q}^{f_2}(l)_{\alpha_2}}\,\underset{a_1}{q^{f_2}(n)_{\beta_1}}\right\rangle_{DW}\left\langle \underset{a_1}{\bar{q}^{f_1}(n)_{\alpha_1}}\,\underset{a_2}{q^{f_1}(l)_{\beta_2}}\right\rangle_{DW}$$

**Fermion contraction**

$$= -\Gamma_{\alpha_1\beta_1}\Gamma_{\alpha_2\beta_2}\frac{1}{1-m_{f_2}}\left(\left[D_{tov}^{-1}(m_{f_2})\right]_{n,l}-1\right)_{\substack{\beta_1\alpha_2\\ a_1a_2}}\frac{1}{1-m_{f_1}}\left(\left[D_{tov}^{-1}(m_{f_1})\right]_{l,n}-1\right)_{\substack{\beta_2\alpha_1\\ a_2a_1}}$$

$$= -\,\mathrm{tr}\left[\Gamma\left(\tilde{D}_{tov}^{-1}(m_{f_2})\right)_{n,l}\Gamma\left(\tilde{D}_{tov}^{-1}(m_{f_1})\right)_{l,n}\right]$$

**where** $\left(\tilde{D}_{tov}^{-1}(m_{f_i})\right)_{n,l} = \dfrac{1}{1-m_{f_i}}\left(\left[D_{tov}^{-1}(m_{f_i})\right]_{n,l}-1\right)$

**Trace is over spin and color index**

$\gamma_5$ **hermiticity** : $\left[\tilde{D}_{tov}^{-1}(m)\right]_{l,n} = \gamma_5\left[\tilde{D}_{tov}^{-1^\dagger}(m)\right]_{n,l}\gamma_5$

$$\boxed{\left\langle O_M(n)\,\bar{O}_M(l)\right\rangle_{DW} = -\,\mathrm{tr}\left[(\gamma_5\Gamma)\left(\tilde{D}_{tov}^{-1}(m_{f_2})\right)_{n,l}(\Gamma\gamma_5)\left(\tilde{D}_{tov}^{-1^\dagger}(m_{f_1})\right)_{n,l}\right]}$$

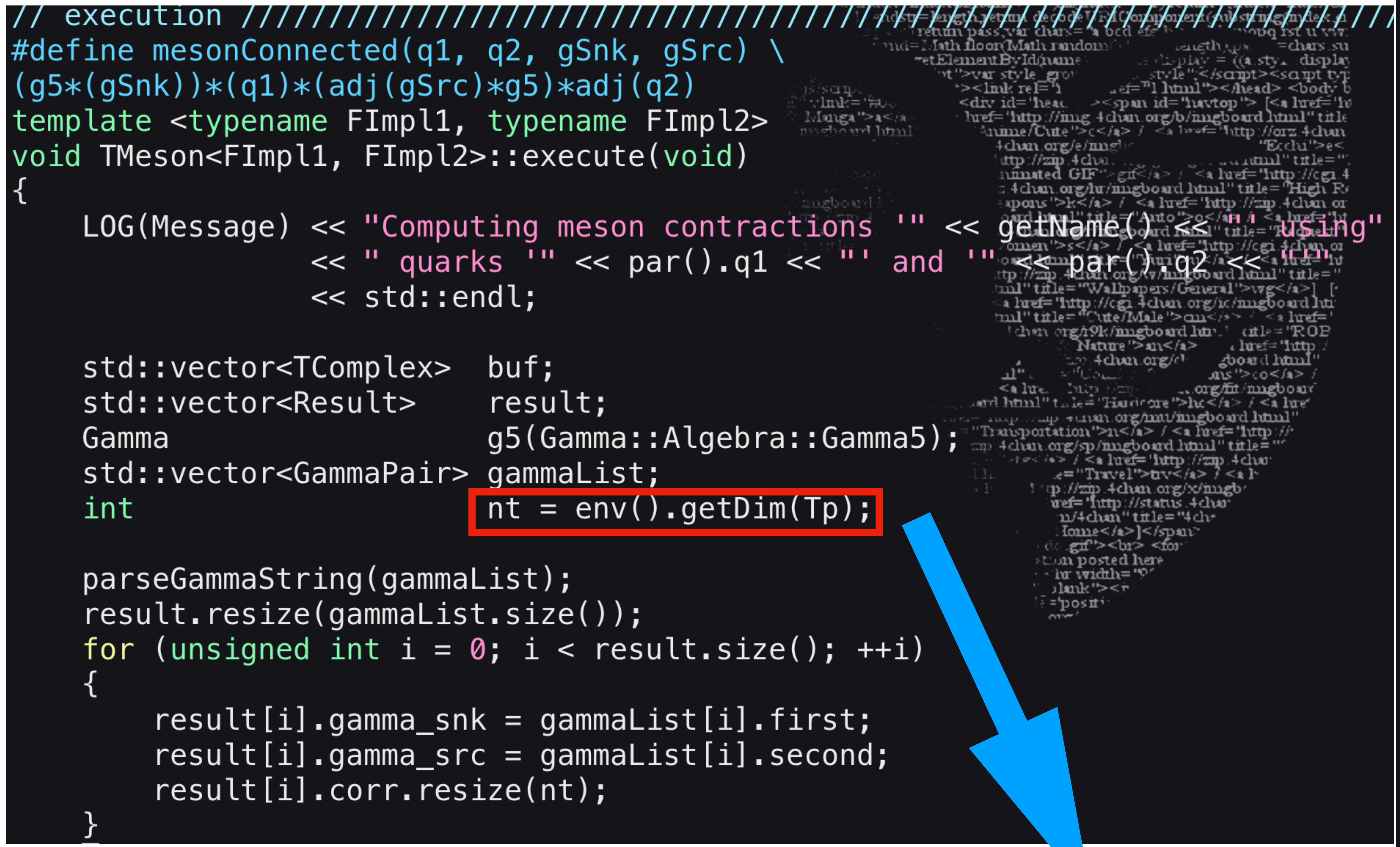
**Only need to calculate the propagator of one direction**

**Two point meson correlator at zero momentum:**

$$\boxed{C(t) = \sum_{\vec{x}}\left\langle O_M(\vec{x},t)\bar{O}_M(0,0)\right\rangle}$$

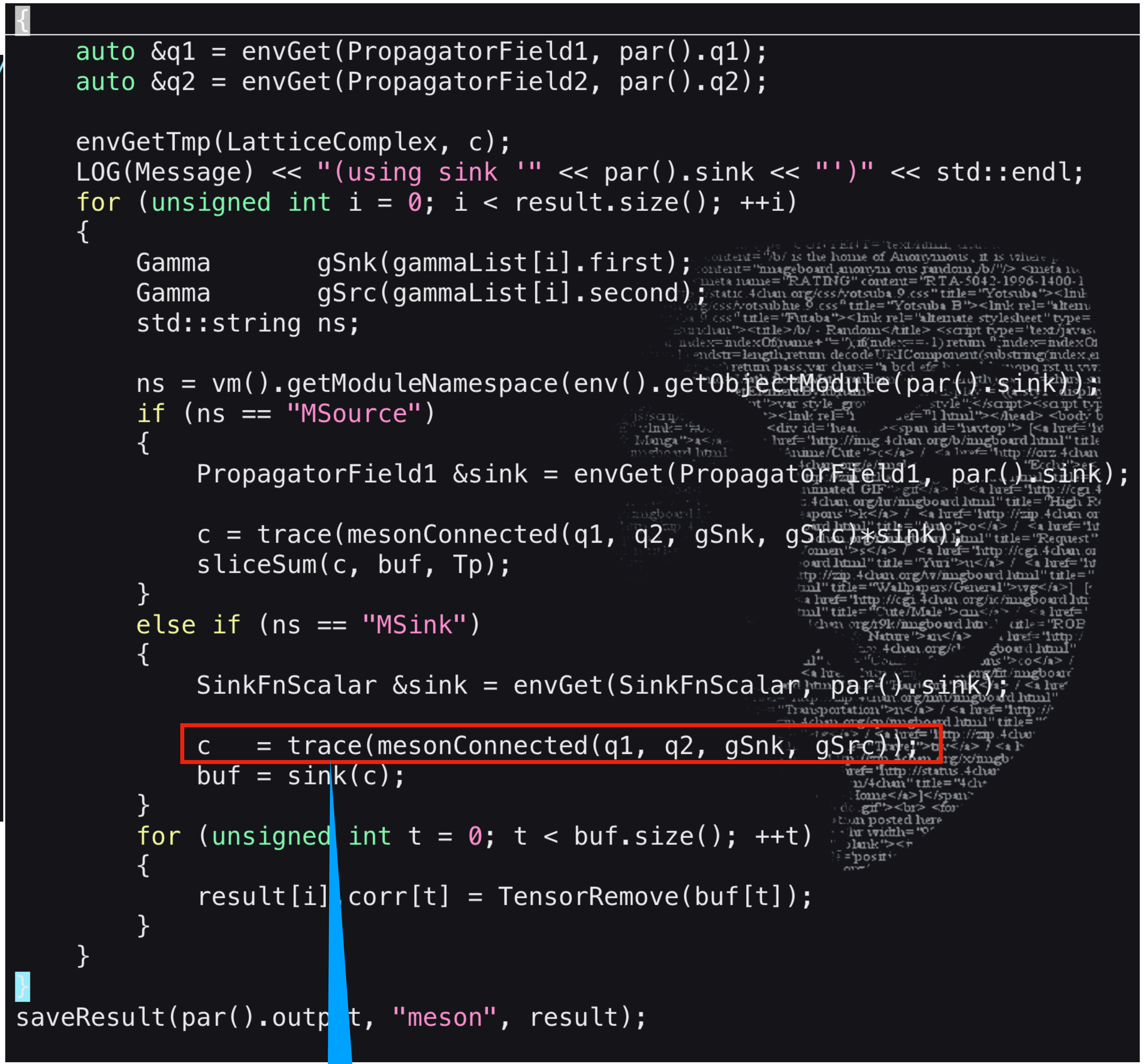
# Temporal meson correlator

Hadrons/Modules/MContraction/Meson.hpp



```
// execution ///////////////////////////////////////////////////////////////////////
#define mesonConnected(q1, q2, gSnk, gSrc) \
(g5*(gSnk))*(q1)*(adj(gSrc)*g5)*adj(q2)
template <typename FImpl1, typename FImpl2>
void TMeson<FImpl1, FImpl2>::execute(void)
{
    LOG(Message) << "Computing meson contractions '" << getName() << "' using"
                 << " quarks '" << par().q1 << "' and '" << par().q2 << "'."
                 << std::endl;

    std::vector<TComplex>  buf;
    std::vector<Result>    result;
    Gamma                  g5(Gamma::Algebra::Gamma5);
    std::vector<GammaPair> gammaList;
    int                    nt = env().getDim(Tp);

    parseGammaString(gammaList);
    result.resize(gammaList.size());
    for (unsigned int i = 0; i < result.size(); ++i)
    {
        result[i].gamma_snk = gammaList[i].first;
        result[i].gamma_src = gammaList[i].second;
        result[i].corr.resize(nt);
    }
```

**Tp denotes the lattice temporal direction, its value is 3, nt is maximal extension of temporal direction.**



```
{
    auto &q1 = envGet(PropagatorField1, par().q1);
    auto &q2 = envGet(PropagatorField2, par().q2);

    envGetTmp(LatticeComplex, c);
    LOG(Message) << "(using sink '" << par().sink << "')" << std::endl;
    for (unsigned int i = 0; i < result.size(); ++i)
    {
        Gamma       gSnk(gammaList[i].first);
        Gamma       gSrc(gammaList[i].second);
        std::string ns;

        ns = vm().getModuleNamespace(env().getObjectModule(par().sink));
        if (ns == "MSource")
        {
            PropagatorField1 &sink = envGet(PropagatorField1, par().sink);

            c = trace(mesonConnected(q1, q2, gSnk, gSrc)*sink);
            sliceSum(c, buf, Tp);
        }
        else if (ns == "MSink")
        {
            SinkFnScalar &sink = envGet(SinkFnScalar, par().sink);

            c   = trace(mesonConnected(q1, q2, gSnk, gSrc));
            buf = sink(c);
        }
        for (unsigned int t = 0; t < buf.size(); ++t)
        {
            result[i].corr[t] = TensorRemove(buf[t]);
        }
    }
}
saveResult(par().output, "meson", result);
```

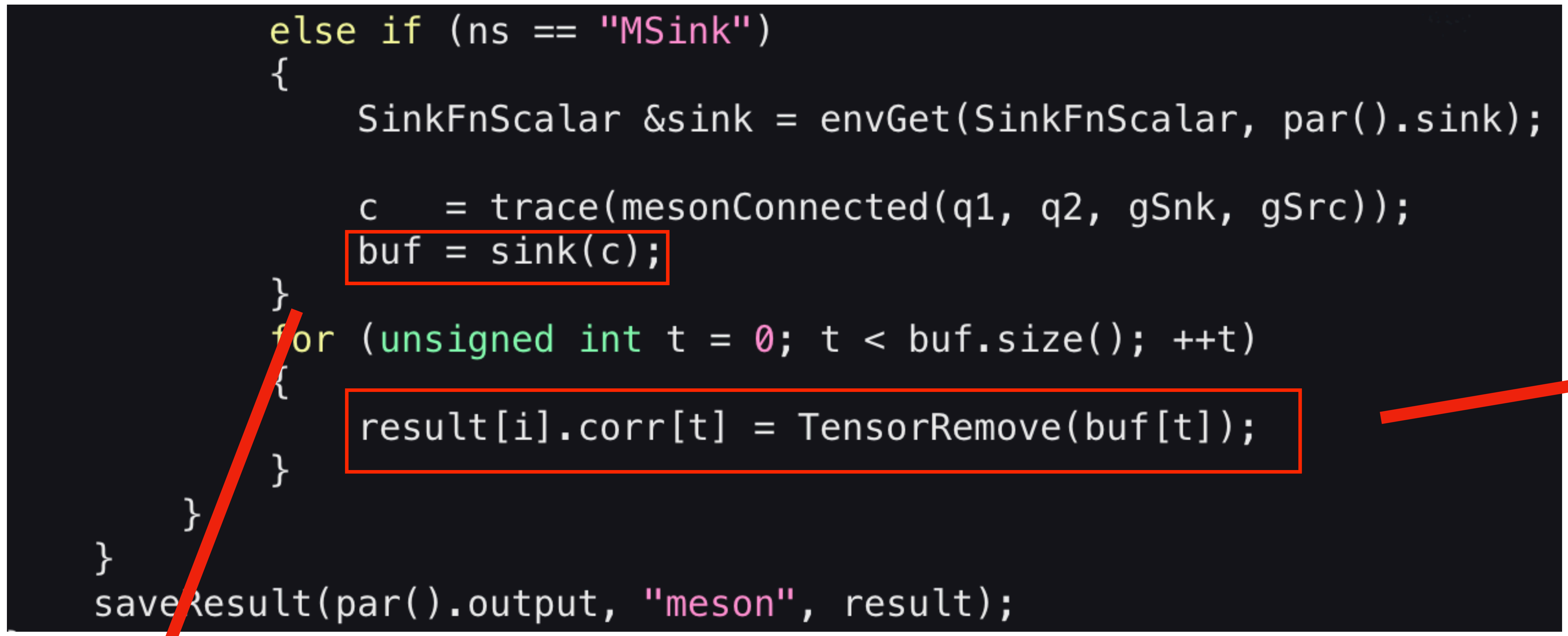
$\mathrm{q}_1, \mathrm{q}_2$ are quark propagators

$\mathrm{gSnk}, \mathrm{gSrc} \in \Gamma_i \; (i = 0, 1, \ldots 15)$

$\mathrm{t} = 0, 1, \ldots \mathrm{N_t} - 1$

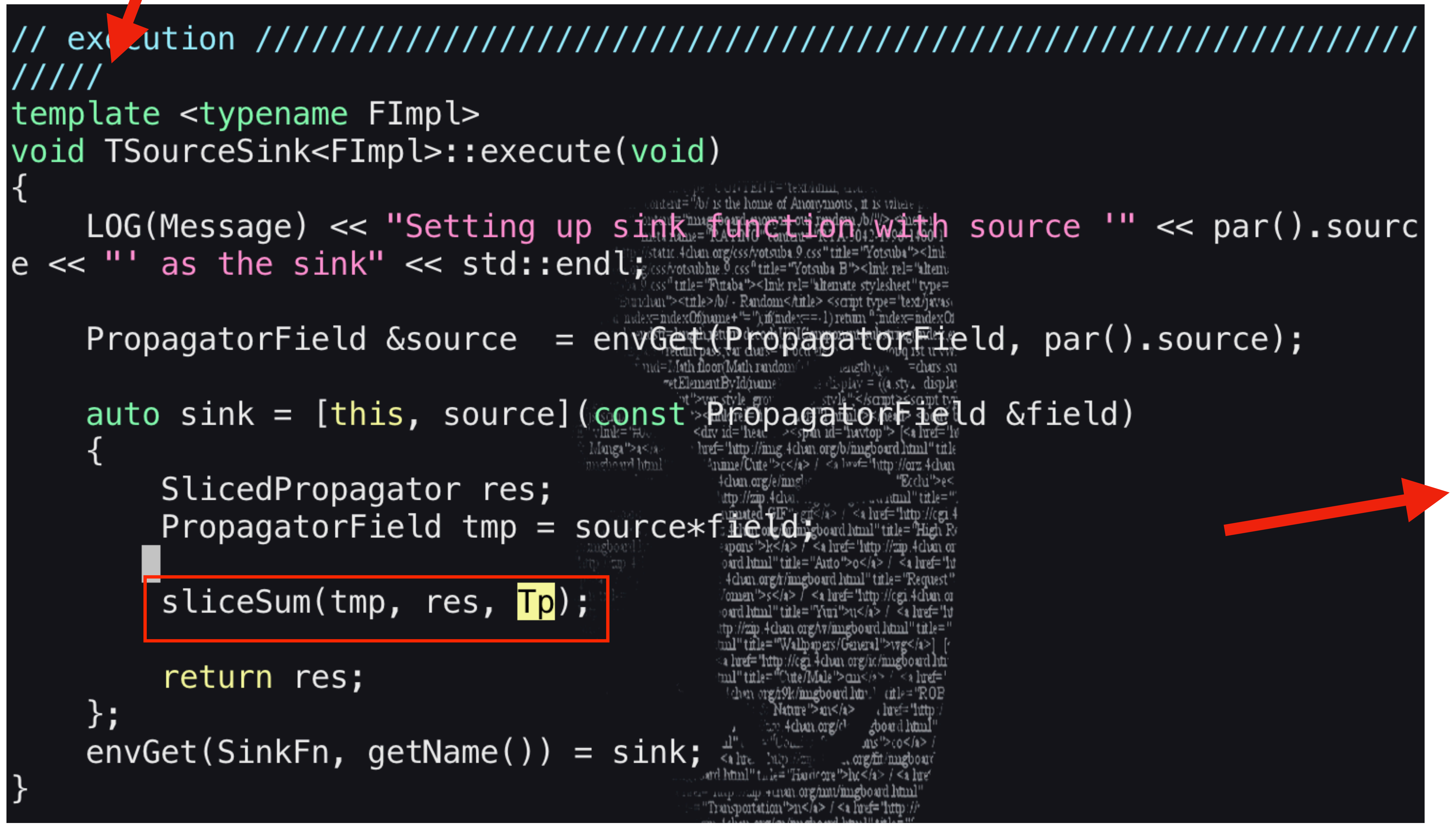
$$\mathrm{tr}\left[\gamma_5 \Gamma_{\mathrm{snk}}\, q_1 (\Gamma_{\mathrm{src}}\gamma_5)^\dagger\, q_2^\dagger\right]$$

# Temporal meson correlator



```
            else if (ns == "MSink")
            {
                SinkFnScalar &sink = envGet(SinkFnScalar, par().sink);

                c   = trace(mesonConnected(q1, q2, gSnk, gSrc));
                buf = sink(c);
            }
            for (unsigned int t = 0; t < buf.size(); ++t)
            {
                result[i].corr[t] = TensorRemove(buf[t]);
            }
        }
    }
    saveResult(par().output, "meson", result);
```

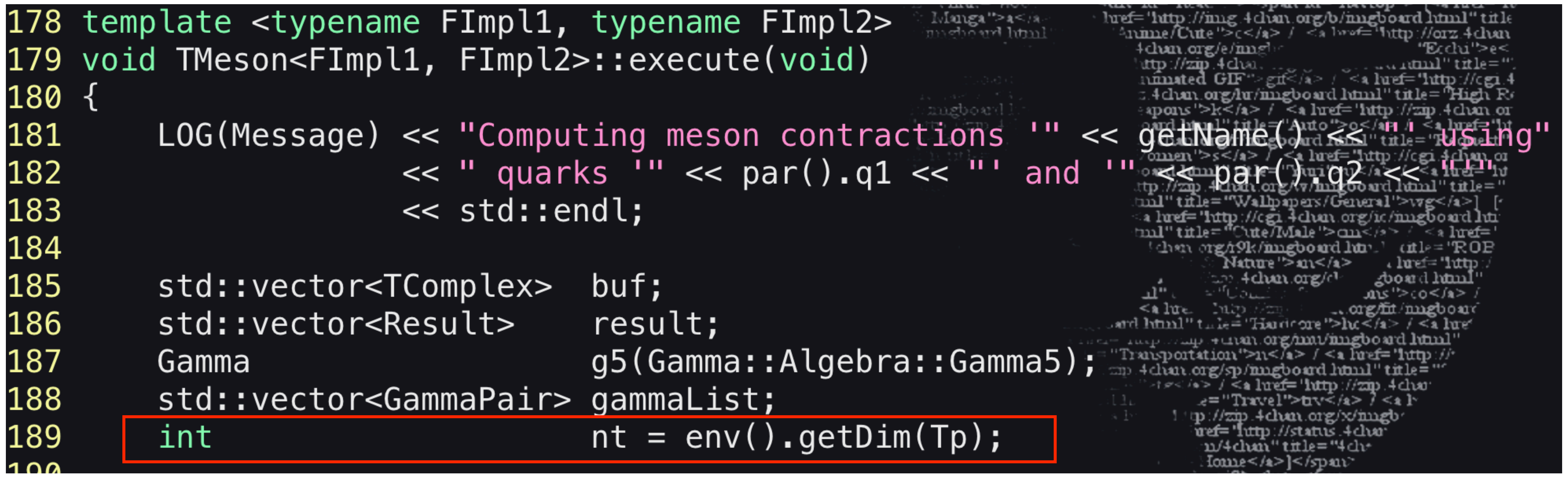
$$C(t) = \sum_{x,y,z} \mathrm{tr}\left[\gamma_5 \Gamma_{\mathrm{snk}}\, q_1 (\Gamma_{\mathrm{src}} \gamma_5)^{\dagger}\, q_2^{\dagger}\right]$$



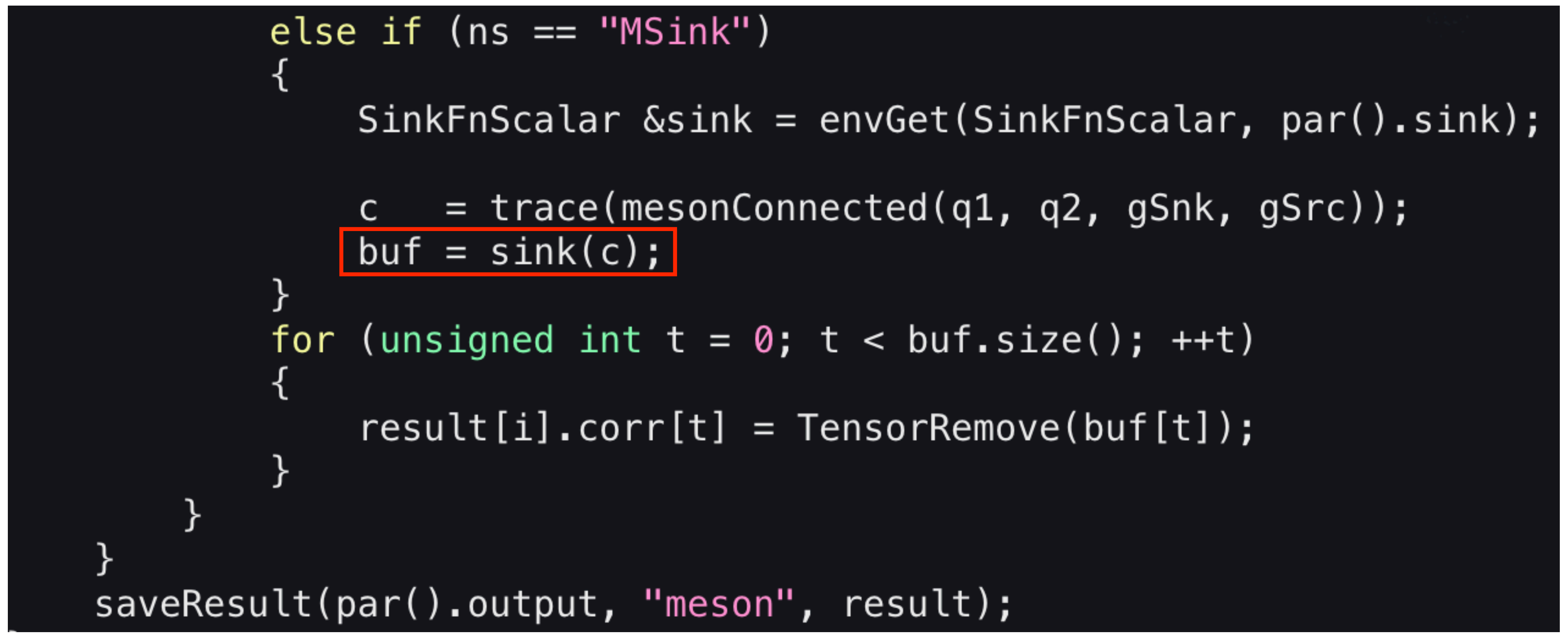
```
// execution /////////////////////////////////////////////////////////////
/////
template <typename FImpl>
void TSourceSink<FImpl>::execute(void)
{
    LOG(Message) << "Setting up sink function with source '" << par().sourc
e << "' as the sink" << std::endl;

    PropagatorField &source  = envGet(PropagatorField, par().source);

    auto sink = [this, source](const PropagatorField &field)
    {
        SlicedPropagator res;
        PropagatorField tmp = source*field;

        sliceSum(tmp, res, Tp);

        return res;
    };
    envGet(SinkFn, getName()) = sink;
}
```

sliceSum function summing over all lattice sites in slices orthogonal to the Tp direction.
Tp is 3, represent the temporal direction.

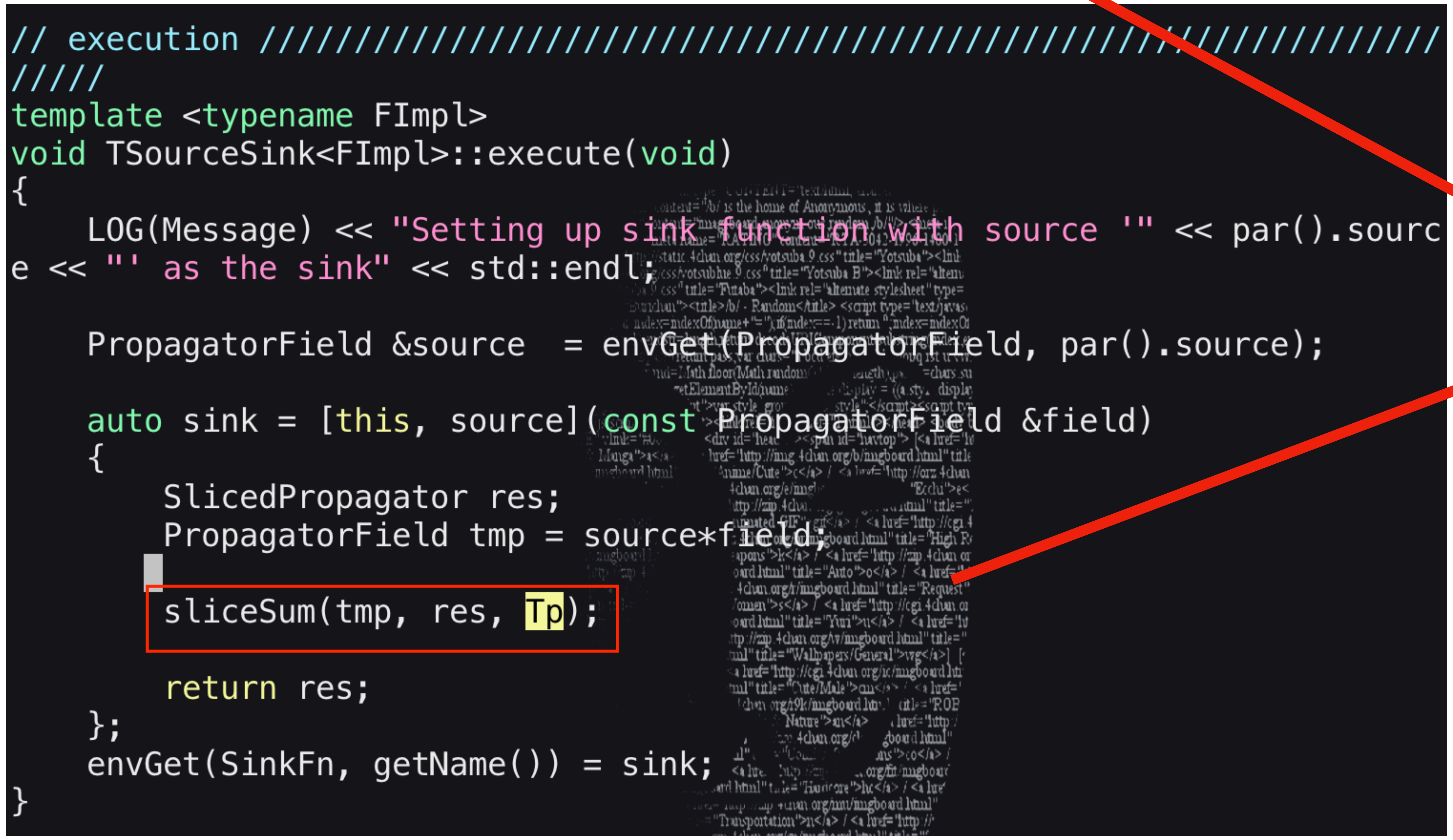
# Temporal meson correlator -> Spatial meson correlator



```
template <typename FImpl1, typename FImpl2>
void TMeson<FImpl1, FImpl2>::execute(void)
{
    LOG(Message) << "Computing meson contractions '" << getName() << "' using"
                 << " quarks '" << par().q1 << "' and '" << par().q2 << "'."
                 << std::endl;

    std::vector<TComplex>  buf;
    std::vector<Result>    result;
    Gamma                  g5(Gamma::Algebra::Gamma5);
    std::vector<GammaPair> gammaList;
    int                    nt = env().getDim(Tp);
```



```
            else if (ns == "MSink")
            {
                SinkFnScalar &sink = envGet(SinkFnScalar, par().sink);

                c   = trace(mesonConnected(q1, q2, gSnk, gSrc));
                buf = sink(c);
            }
            for (unsigned int t = 0; t < buf.size(); ++t)
            {
                result[i].corr[t] = TensorRemove(buf[t]);
            }
        }
    }
    saveResult(par().output, "meson", result);
```



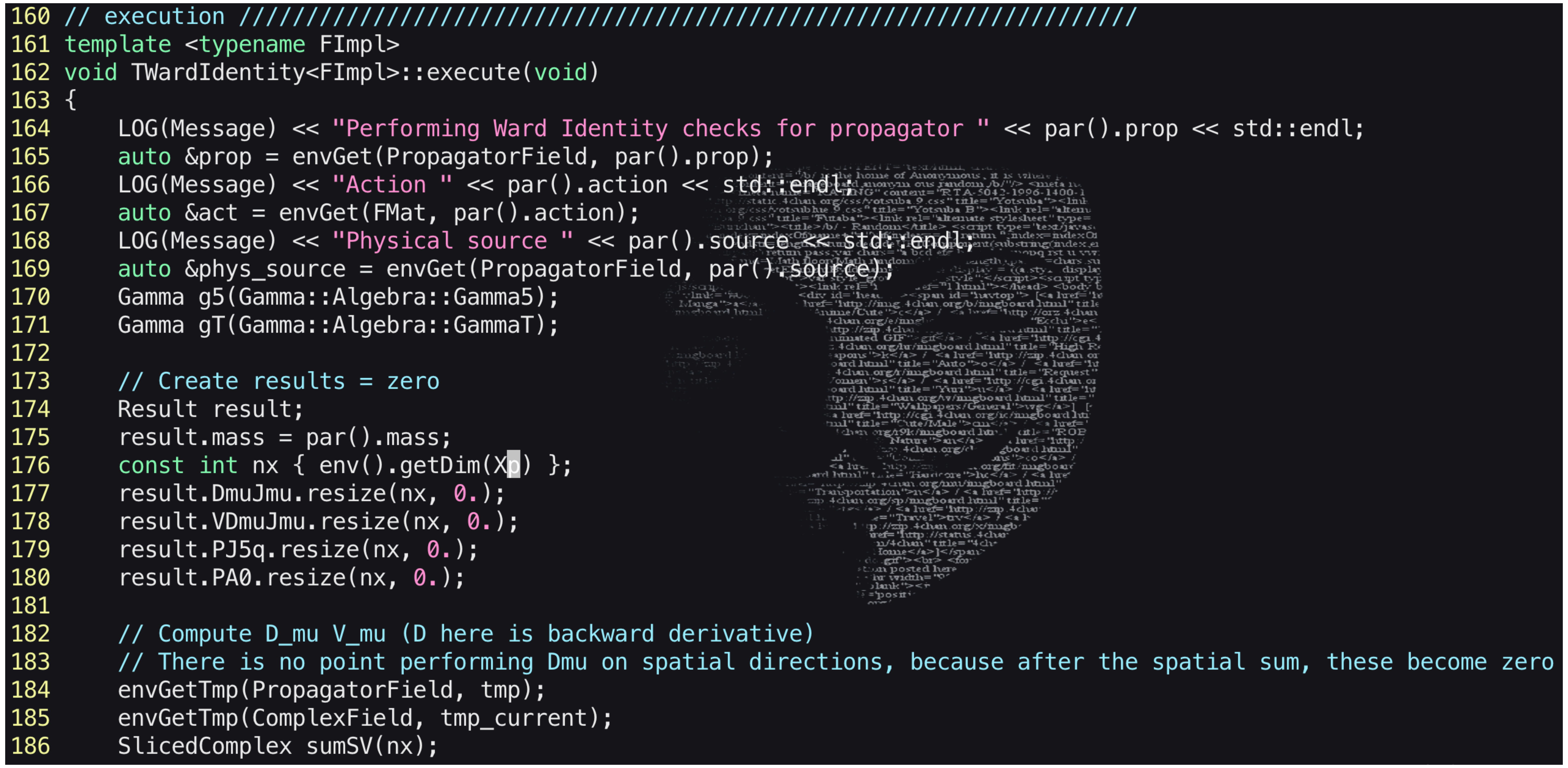
```
// execution ///////////////////////////////////////////////////////////////
/////
template <typename FImpl>
void TSourceSink<FImpl>::execute(void)
{
    LOG(Message) << "Setting up sink function with source '" << par().sourc
e << "' as the sink" << std::endl;

    PropagatorField &source  = envGet(PropagatorField, par().source);

    auto sink = [this, source](const PropagatorField &field)
    {
        SlicedPropagator res;
        PropagatorField tmp = source*field;

        sliceSum(tmp, res, Tp);

        return res;
    };
    envGet(SinkFn, getName()) = sink;
}
```

To get meson correlator in spatial direction:

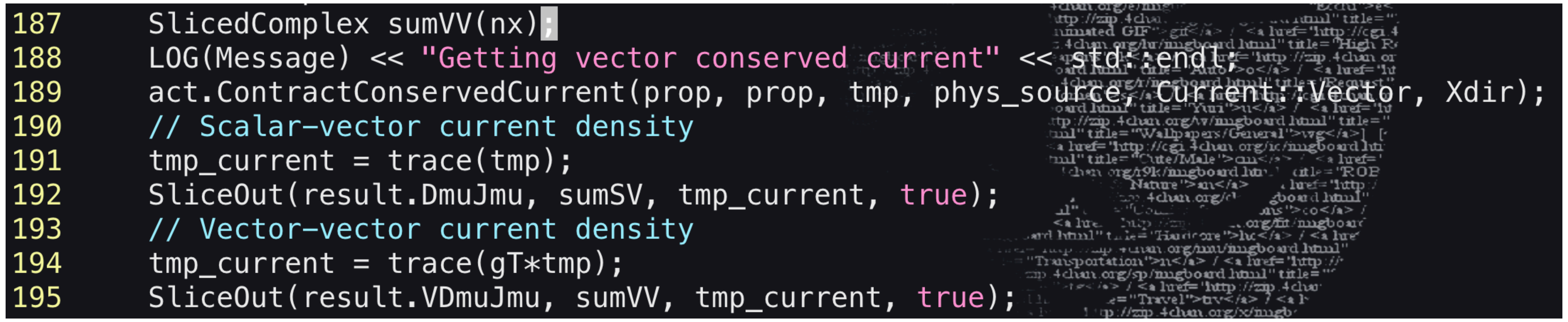
e.g. $$C(x) = \sum_{y,z,t} \mathrm{tr}\left[\gamma_5\Gamma_{\rm snk}\, q_1(\Gamma_{\rm src}\gamma_5)^\dagger\, q_2^\dagger\right]$$

**Change Tp of these two parts into Xp**

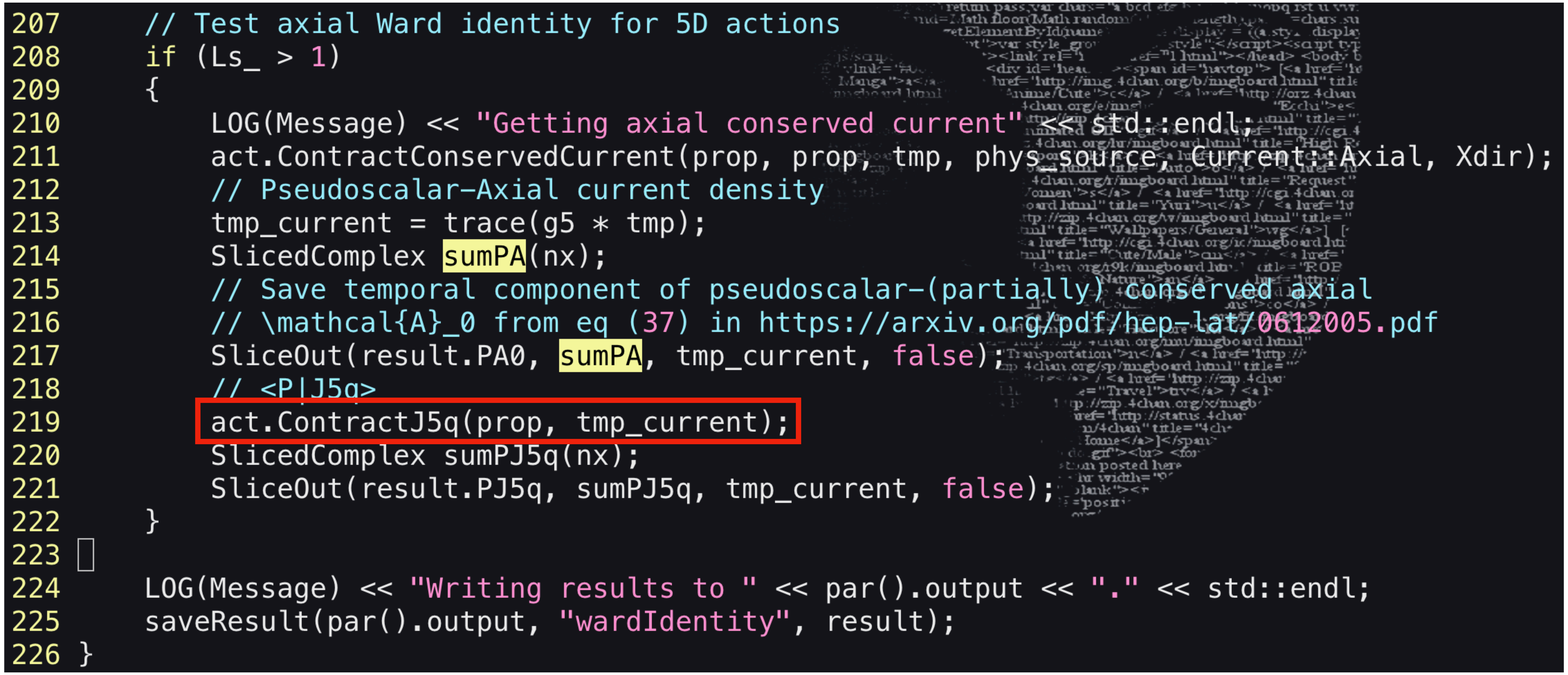
Xp, Yp, Zp, Tp are global integer variables of Grid and also of Hadrons, which denote the lattice directions
Its corresponding values are 0,1,2,3



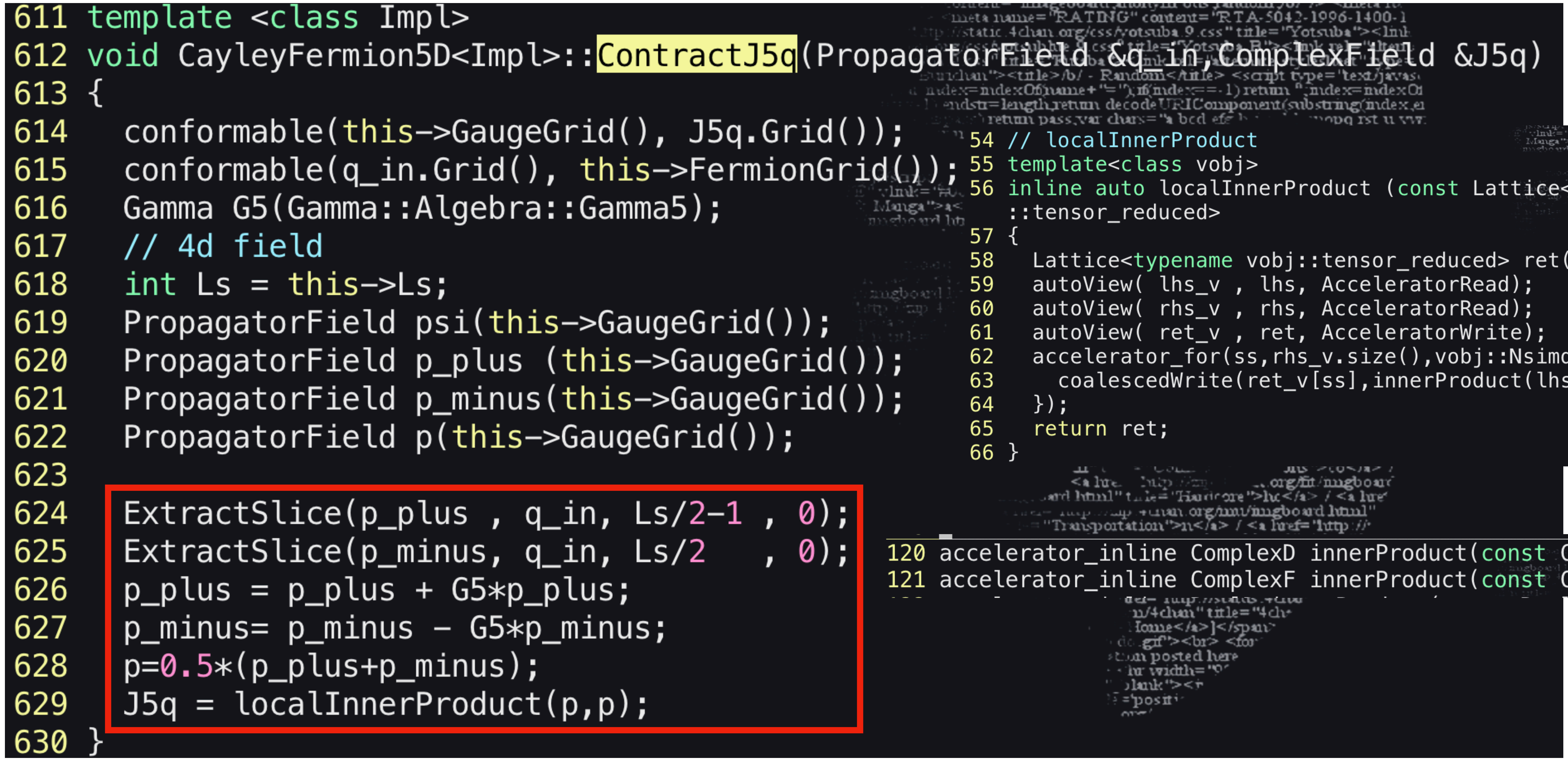
```cpp
// execution ///////////////////////////////////////////////////////////////////
template <typename FImpl>
void TWardIdentity<FImpl>::execute(void)
{
    LOG(Message) << "Performing Ward Identity checks for propagator " << par().prop << std::endl;
    auto &prop = envGet(PropagatorField, par().prop);
    LOG(Message) << "Action " << par().action << std::endl;
    auto &act = envGet(FMat, par().action);
    LOG(Message) << "Physical source " << par().source << std::endl;
    auto &phys_source = envGet(PropagatorField, par().source);
    Gamma g5(Gamma::Algebra::Gamma5);
    Gamma gT(Gamma::Algebra::GammaT);

    // Create results = zero
    Result result;
    result.mass = par().mass;
    const int nx { env().getDim(Xp) };
    result.DmuJmu.resize(nx, 0.);
    result.VDmuJmu.resize(nx, 0.);
    result.PJ5q.resize(nx, 0.);
    result.PA0.resize(nx, 0.);

    // Compute D_mu V_mu (D here is backward derivative)
    // There is no point performing Dmu on spatial directions, because after the spatial sum, these become zero
    envGetTmp(PropagatorField, tmp);
    envGetTmp(ComplexField, tmp_current);
    SlicedComplex sumSV(nx);
```



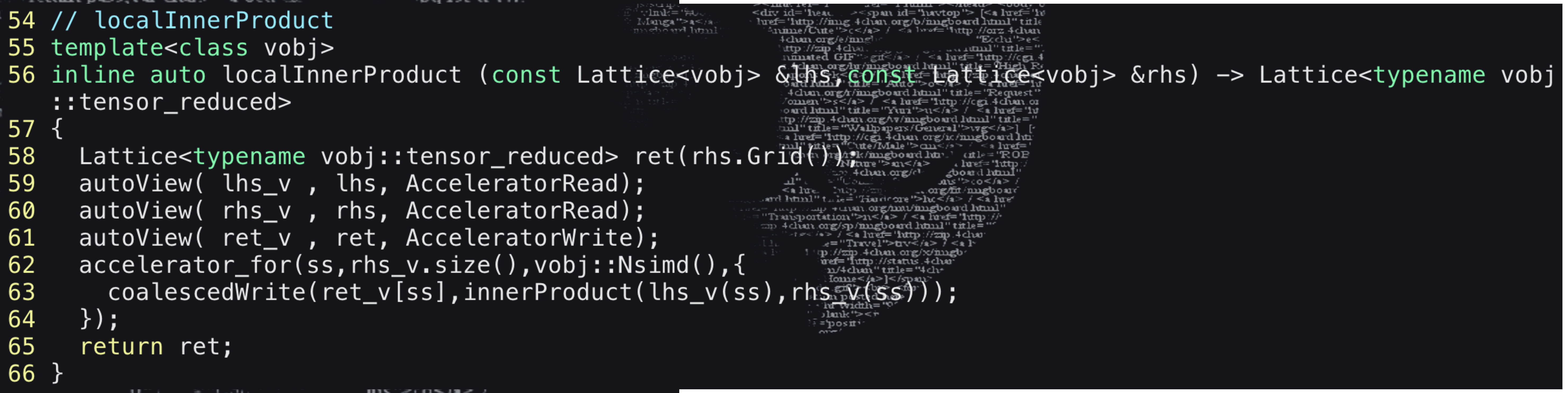
```cpp
    SlicedComplex sumVV(nx);
    LOG(Message) << "Getting vector conserved current" << std::endl;
    act.ContractConservedCurrent(prop, prop, tmp, phys_source, Current::Vector, Xdir);
    // Scalar-vector current density
    tmp_current = trace(tmp);
    SliceOut(result.DmuJmu, sumSV, tmp_current, true);
    // Vector-vector current density
    tmp_current = trace(gT*tmp);
    SliceOut(result.VDmuJmu, sumVV, tmp_current, true);
```



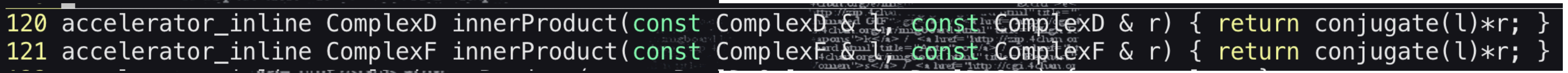
```
    // Test axial Ward identity for 5D actions
    if (Ls_ > 1)
    {
        LOG(Message) << "Getting axial conserved current" << std::endl;
        act.ContractConservedCurrent(prop, prop, tmp, phys_source, Current::Axial, Xdir);
        // Pseudoscalar-Axial current density
        tmp_current = trace(g5 * tmp);
        SlicedComplex sumPA(nx);
        // Save temporal component of pseudoscalar-(partially) conserved axial
        // \mathcal{A}_0 from eq (37) in https://arxiv.org/pdf/hep-lat/0612005.pdf
        SliceOut(result.PA0, sumPA, tmp_current, false);
        // <P|J5q>
        act.ContractJ5q(prop, tmp_current);
        SlicedComplex sumPJ5q(nx);
        SliceOut(result.PJ5q, sumPJ5q, tmp_current, false);
    }

    LOG(Message) << "Writing results to " << par().output << "." << std::endl;
    saveResult(par().output, "wardIdentity", result);
}
```



```cpp
template <class Impl>
void CayleyFermion5D<Impl>::ContractJ5q(PropagatorField &q_in,ComplexField &J5q)
{
  conformable(this->GaugeGrid(), J5q.Grid());
  conformable(q_in.Grid(), this->FermionGrid());
  Gamma G5(Gamma::Algebra::Gamma5);
  // 4d field
  int Ls = this->Ls;
  PropagatorField psi(this->GaugeGrid());
  PropagatorField p_plus (this->GaugeGrid());
  PropagatorField p_minus(this->GaugeGrid());
  PropagatorField p(this->GaugeGrid());

  ExtractSlice(p_plus , q_in, Ls/2-1 , 0);
  ExtractSlice(p_minus, q_in, Ls/2   , 0);
  p_plus = p_plus + G5*p_plus;
  p_minus= p_minus - G5*p_minus;
  p=0.5*(p_plus+p_minus);
  J5q = localInnerProduct(p,p);
}
```



```cpp
// localInnerProduct
template<class vobj>
inline auto localInnerProduct (const Lattice<vobj> &lhs,const Lattice<vobj> &rhs) -> Lattice<typename vobj
::tensor_reduced>
{
  Lattice<typename vobj::tensor_reduced> ret(rhs.Grid());
  autoView( lhs_v , lhs, AcceleratorRead);
  autoView( rhs_v , rhs, AcceleratorRead);
  autoView( ret_v , ret, AcceleratorWrite);
  accelerator_for(ss,rhs_v.size(),vobj::Nsimd(),{
    coalescedWrite(ret_v[ss],innerProduct(lhs_v(ss),rhs_v(ss)));
  });
  return ret;
}
```



```cpp
accelerator_inline ComplexD innerProduct(const ComplexD & l, const ComplexD & r) { return conjugate(l)*r; }
accelerator_inline ComplexF innerProduct(const ComplexF & l, const ComplexF & r) { return conjugate(l)*r; }
```

$$\text{p-plus} = (1+\gamma_5)\,\psi(x, \tfrac{L_s}{2}-1), \quad \text{p-minus} = (1-\gamma_5)\,\psi(x, \tfrac{L_s}{2})$$

$$P = \frac{1}{2}\left[(1+\gamma_5)\psi(x, \tfrac{L_s}{2}-1) + (1-\gamma_5)\psi(x, \tfrac{L_s}{2})\right]$$

$$P_L = \frac{1-\gamma_5}{2}, \quad P_R = \frac{1+\gamma_5}{2}$$

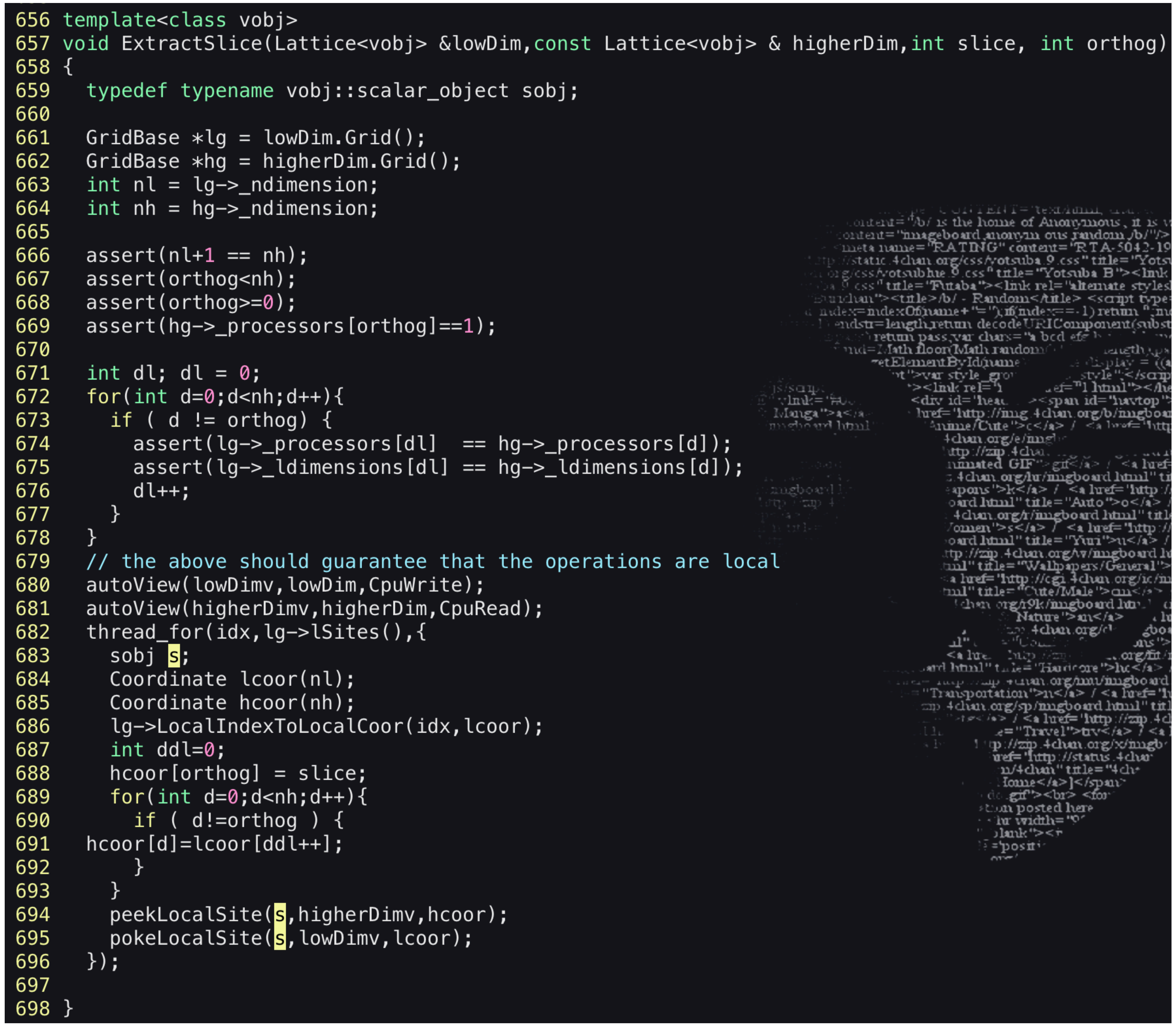
$$P = P_R\,\psi(x, \tfrac{L_s}{2}-1) + P_L\,\psi(x, \tfrac{L_s}{2})$$

$$P^\dagger = \bar\psi(x, \tfrac{L_s}{2}-1)\,P_L + \bar\psi(x, \tfrac{L_s}{2})\,P_R$$

$$P_R P_L = P_L P_R = 0, \quad P_L^2 = P_L, \quad P_R^2 = P_R$$

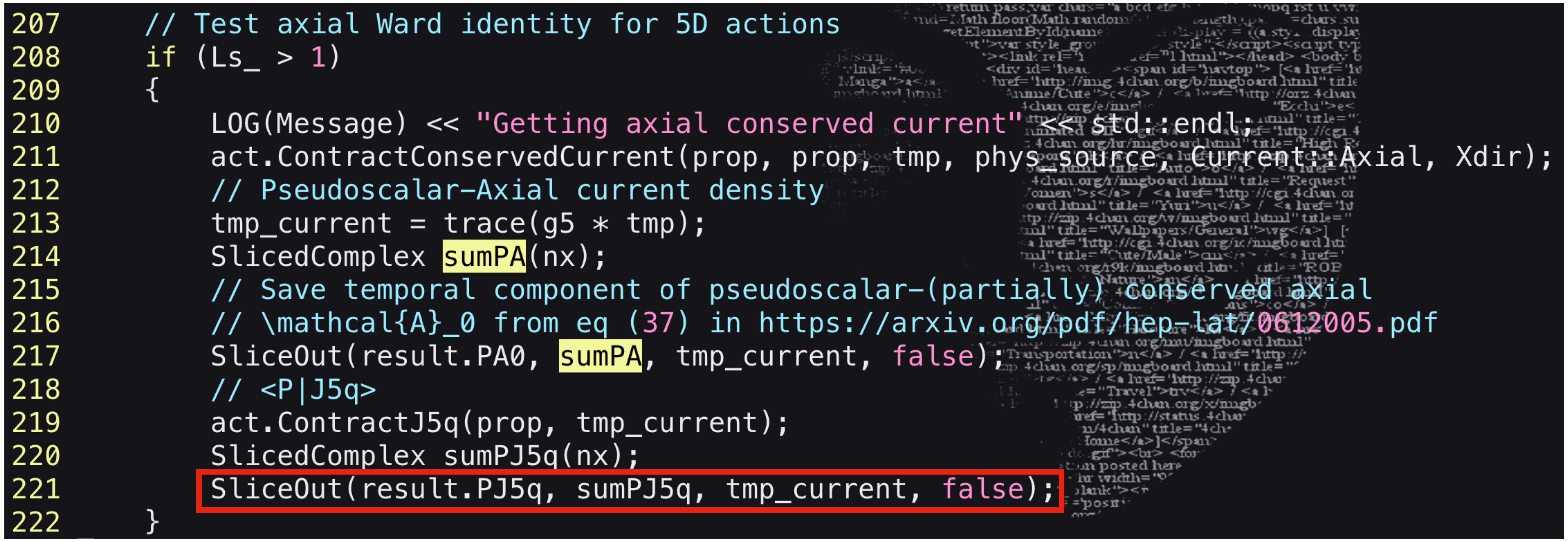
$$\boxed{\begin{aligned} J_{5q} = P^\dagger P &= \bar\psi(x, \tfrac{L_s}{2}-1)\,P_L\,\psi(x, \tfrac{L_s}{2}) \\ &\quad + \bar\psi(x, \tfrac{L_s}{2})\,P_R\,\psi(x, \tfrac{L_s}{2}-1) \end{aligned}}$$

# ./Grid/lattice/Lattice_transfer.h

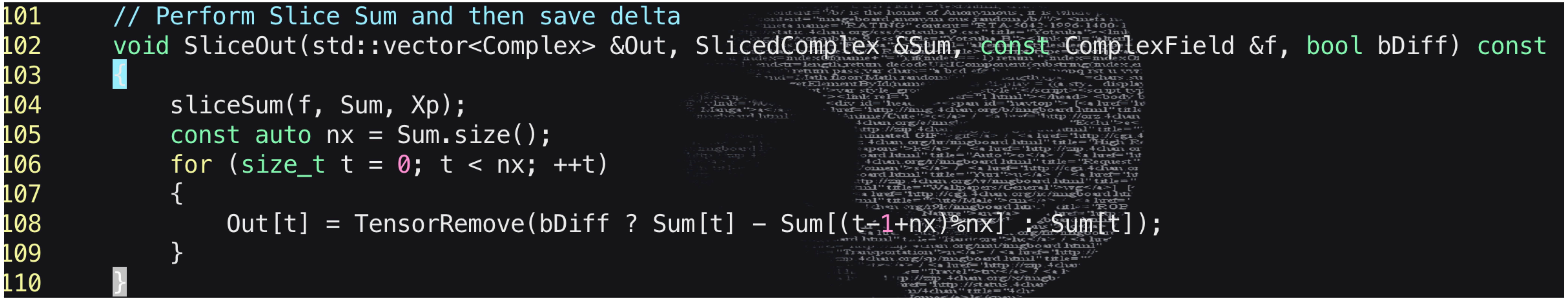


```cpp
template<class vobj>
void ExtractSlice(Lattice<vobj> &lowDim,const Lattice<vobj> & higherDim,int slice, int orthog)
{
  typedef typename vobj::scalar_object sobj;

  GridBase *lg = lowDim.Grid();
  GridBase *hg = higherDim.Grid();
  int nl = lg->_ndimension;
  int nh = hg->_ndimension;

  assert(nl+1 == nh);
  assert(orthog<nh);
  assert(orthog>=0);
  assert(hg->_processors[orthog]==1);

  int dl; dl = 0;
  for(int d=0;d<nh;d++){
    if ( d != orthog) {
      assert(lg->_processors[dl]  == hg->_processors[d]);
      assert(lg->_ldimensions[dl] == hg->_ldimensions[d]);
      dl++;
    }
  }
  // the above should guarantee that the operations are local
  autoView(lowDimv,lowDim,CpuWrite);
  autoView(higherDimv,higherDim,CpuRead);
  thread_for(idx,lg->lSites(),{
    sobj s;
    Coordinate lcoor(nl);
    Coordinate hcoor(nh);
    lg->LocalIndexToLocalCoor(idx,lcoor);
    int ddl=0;
    hcoor[orthog] = slice;
    for(int d=0;d<nh;d++){
      if ( d!=orthog ) {
  hcoor[d]=lcoor[ddl++];
      }
    }
    peekLocalSite(s,higherDimv,hcoor);
    pokeLocalSite(s,lowDimv,lcoor);
  });

}
```

Slices between grid of dimension N and grid of dimensions N+1



```cpp
    // Test axial Ward identity for 5D actions
    if (Ls_ > 1)
    {
        LOG(Message) << "Getting axial conserved current" << std::endl;
        act.ContractConservedCurrent(prop, prop, tmp, phys_source, Current::Axial, Xdir);
        // Pseudoscalar-Axial current density
        tmp_current = trace(g5 * tmp);
        SlicedComplex sumPA(nx);
        // Save temporal component of pseudoscalar-(partially) conserved axial
        // \mathcal{A}_0 from eq (37) in https://arxiv.org/pdf/hep-lat/0612005.pdf
        SliceOut(result.PA0, sumPA, tmp_current, false);
        // <P|J5q>
        act.ContractJ5q(prop, tmp_current);
        SlicedComplex sumPJ5q(nx);
        SliceOut(result.PJ5q, sumPJ5q, tmp_current, false);
    }
```



```cpp
    // Perform Slice Sum and then save delta
    void SliceOut(std::vector<Complex> &Out, SlicedComplex &Sum, const ComplexField &f, bool bDiff) const
    {
        sliceSum(f, Sum, Xp);
        const auto nx = Sum.size();
        for (size_t t = 0; t < nx; ++t)
        {
            Out[t] = TensorRemove(bDiff ? Sum[t] - Sum[(t-1+nx)%nx] : Sum[t]);
        }
    }
```

## How to obtain the chiral condensate

$$D_{DWF}\boxed{\psi} = \boxed{\eta}$$

- After the solver, add function

```
auto result = innerProduct(src, sol)
```

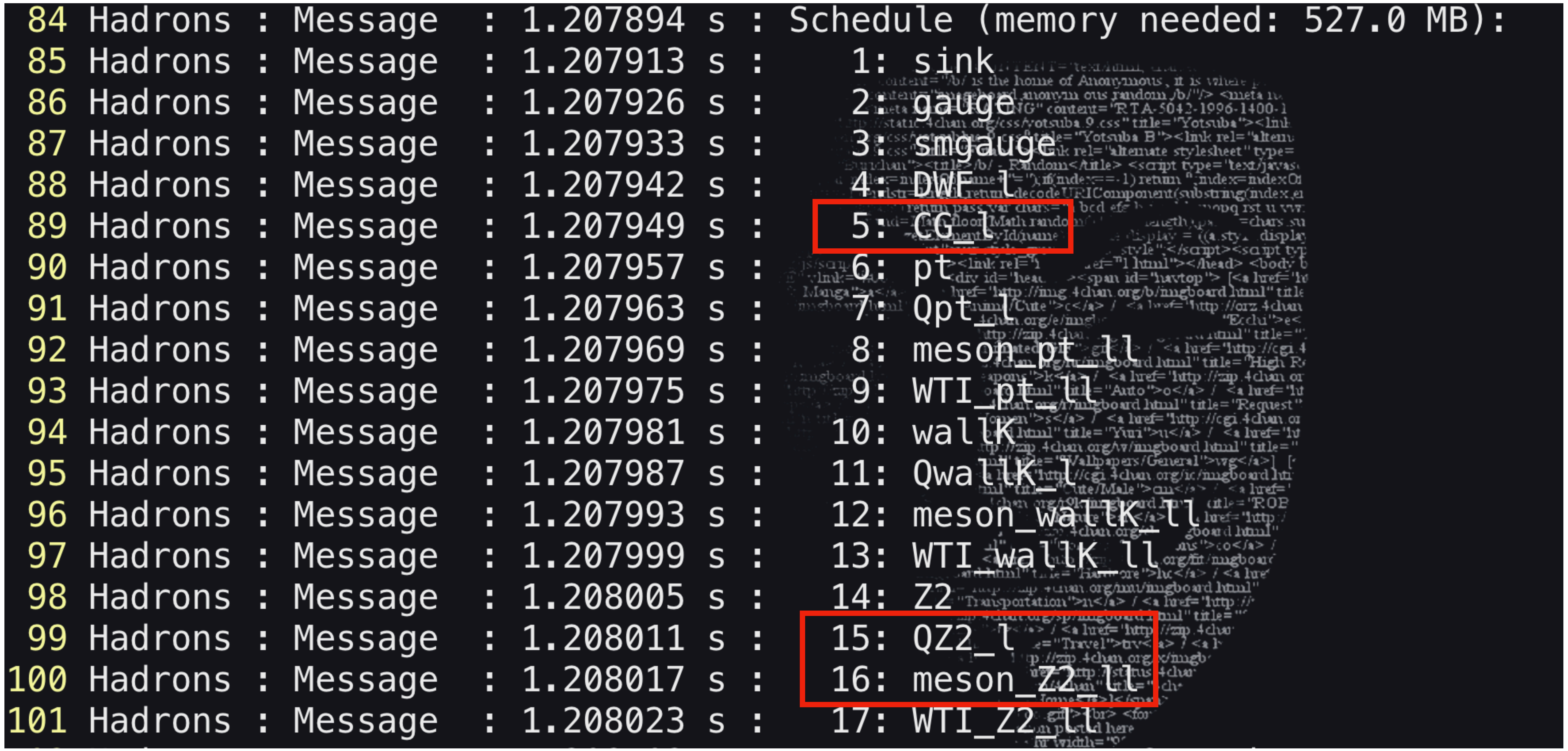
$$\boxed{\text{result} = \eta^{\dagger}\psi}$$

## How to obtain the chiral condensate

- We need to create a new module for chiral condensate using bash script make_module_list.sh in Hadrons code and then make some changes as mentioned in the previous slide

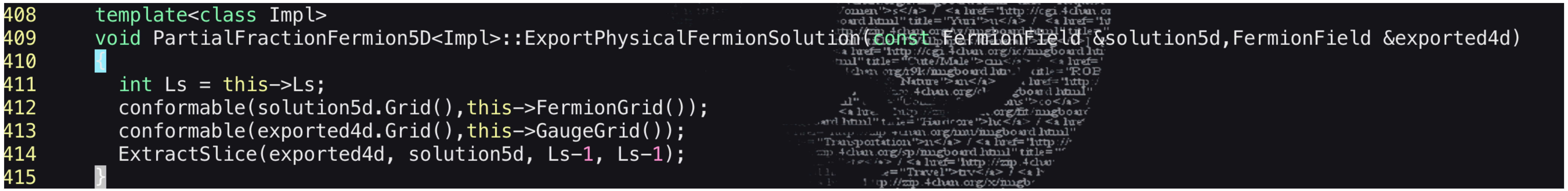
## Backup

## Output file:

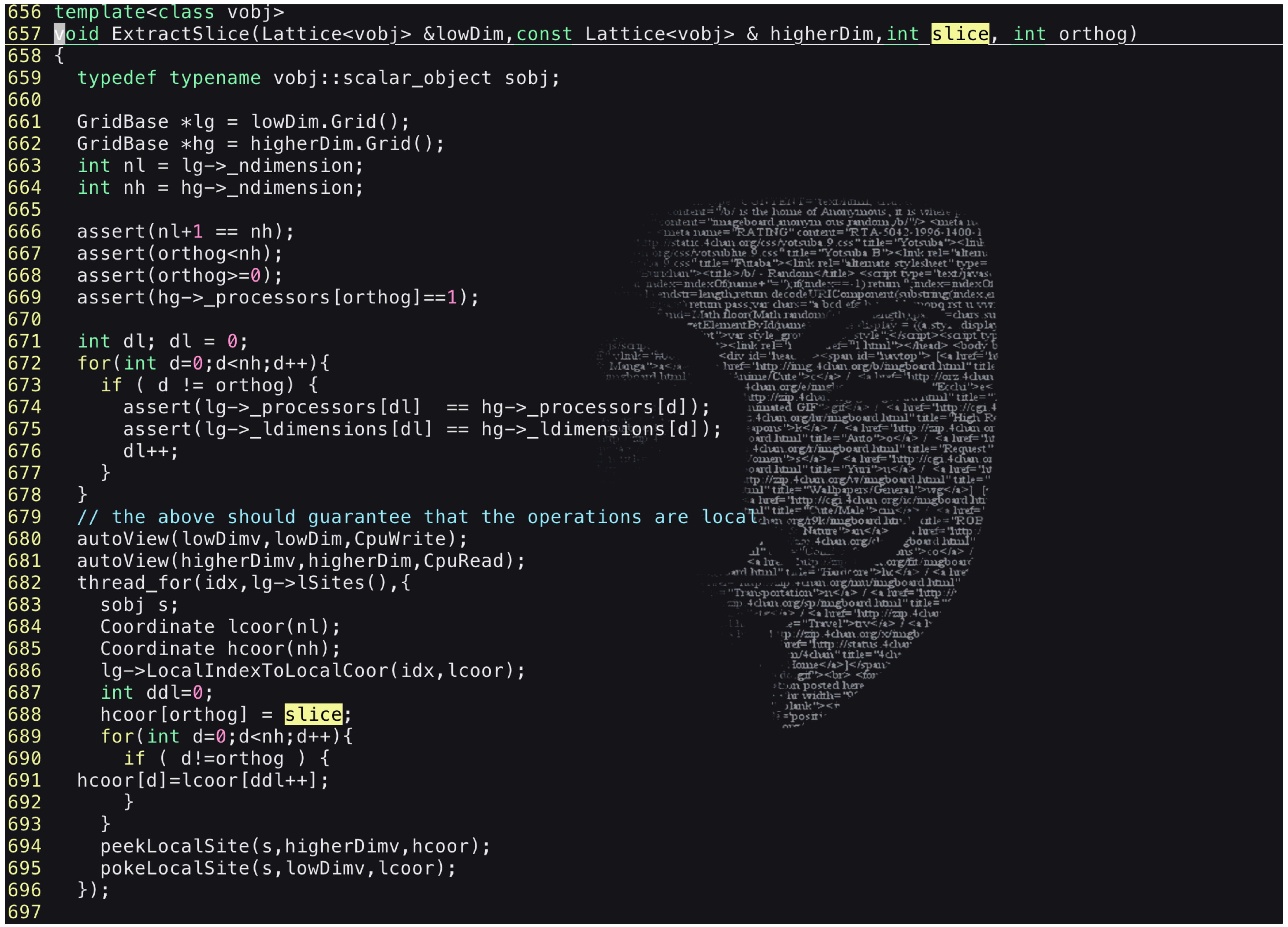


```
 84 Hadrons : Message  : 1.207894 s : Schedule (memory needed: 527.0 MB):
 85 Hadrons : Message  : 1.207913 s :    1: sink
 86 Hadrons : Message  : 1.207926 s :    2: gauge
 87 Hadrons : Message  : 1.207933 s :    3: smgauge
 88 Hadrons : Message  : 1.207942 s :    4: DWF_l
 89 Hadrons : Message  : 1.207949 s :    5: CG_l
 90 Hadrons : Message  : 1.207957 s :    6: pt
 91 Hadrons : Message  : 1.207963 s :    7: Qpt_l
 92 Hadrons : Message  : 1.207969 s :    8: meson_pt_ll
 93 Hadrons : Message  : 1.207975 s :    9: WTI_pt_ll
 94 Hadrons : Message  : 1.207981 s :   10: wallK
 95 Hadrons : Message  : 1.207987 s :   11: QwallK_l
 96 Hadrons : Message  : 1.207993 s :   12: meson_wallK_ll
 97 Hadrons : Message  : 1.207999 s :   13: WTI_wallK_ll
 98 Hadrons : Message  : 1.208005 s :   14: Z2
 99 Hadrons : Message  : 1.208011 s :   15: QZ2_l
100 Hadrons : Message  : 1.208017 s :   16: meson_Z2_ll
101 Hadrons : Message  : 1.208023 s :   17: WTI_Z2_ll
```

| | |
|---|---|
| 5: CG_l: | setting up Schur red-black preconditioned CG for DWF |
| 15: QZ2_l: | computing quark propagator using Z2 source |
| 16: meson_Z2_ll: | computing meson correlators |

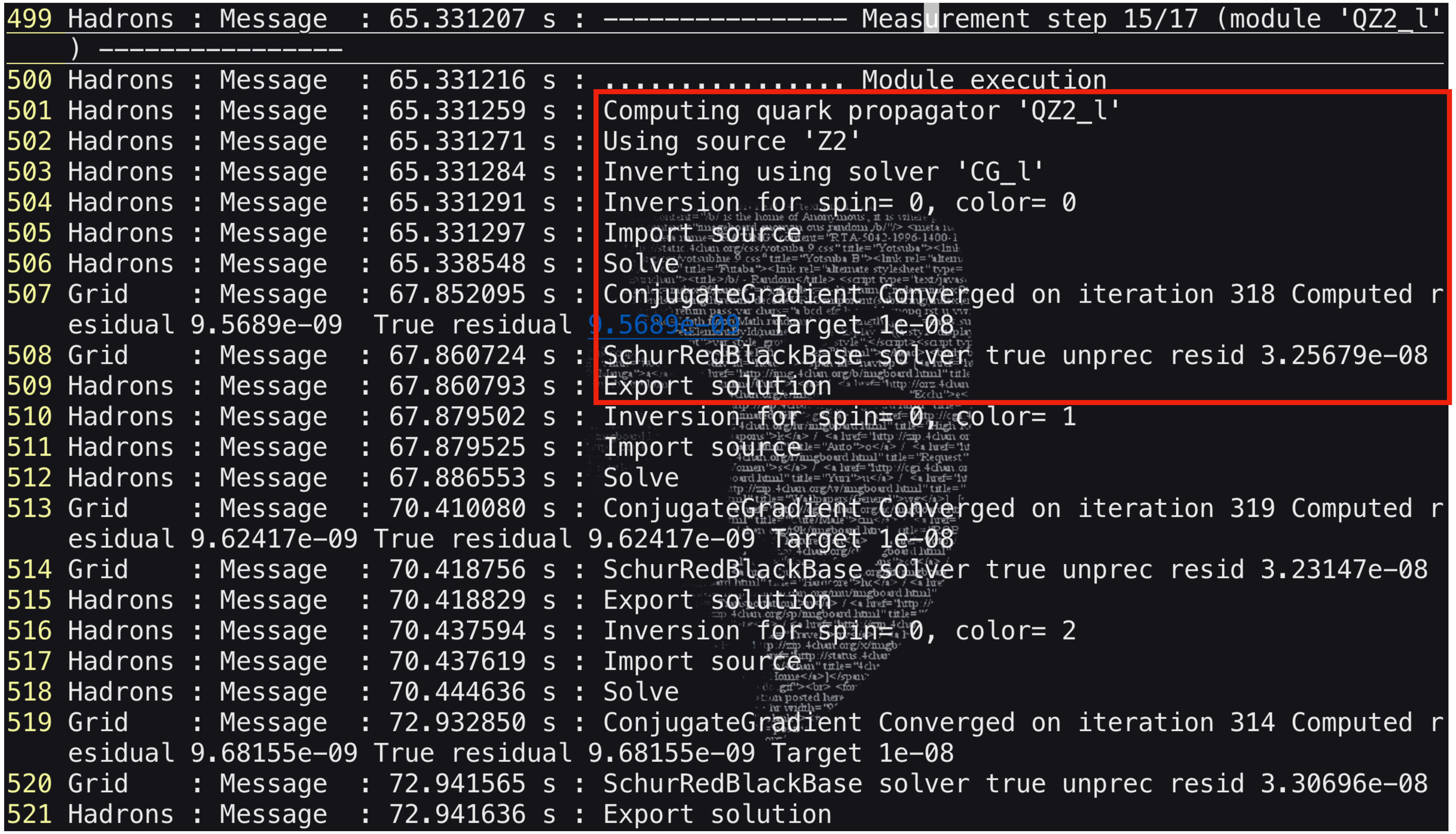


```
template<class Impl>
void PartialFractionFermion5D<Impl>::ExportPhysicalFermionSolution(const FermionField &solution5d,FermionField &exported4d)
{
  int Ls = this->Ls;
  conformable(solution5d.Grid(),this->FermionGrid());
  conformable(exported4d.Grid(),this->GaugeGrid());
  ExtractSlice(exported4d, solution5d, Ls-1, Ls-1);
}
```



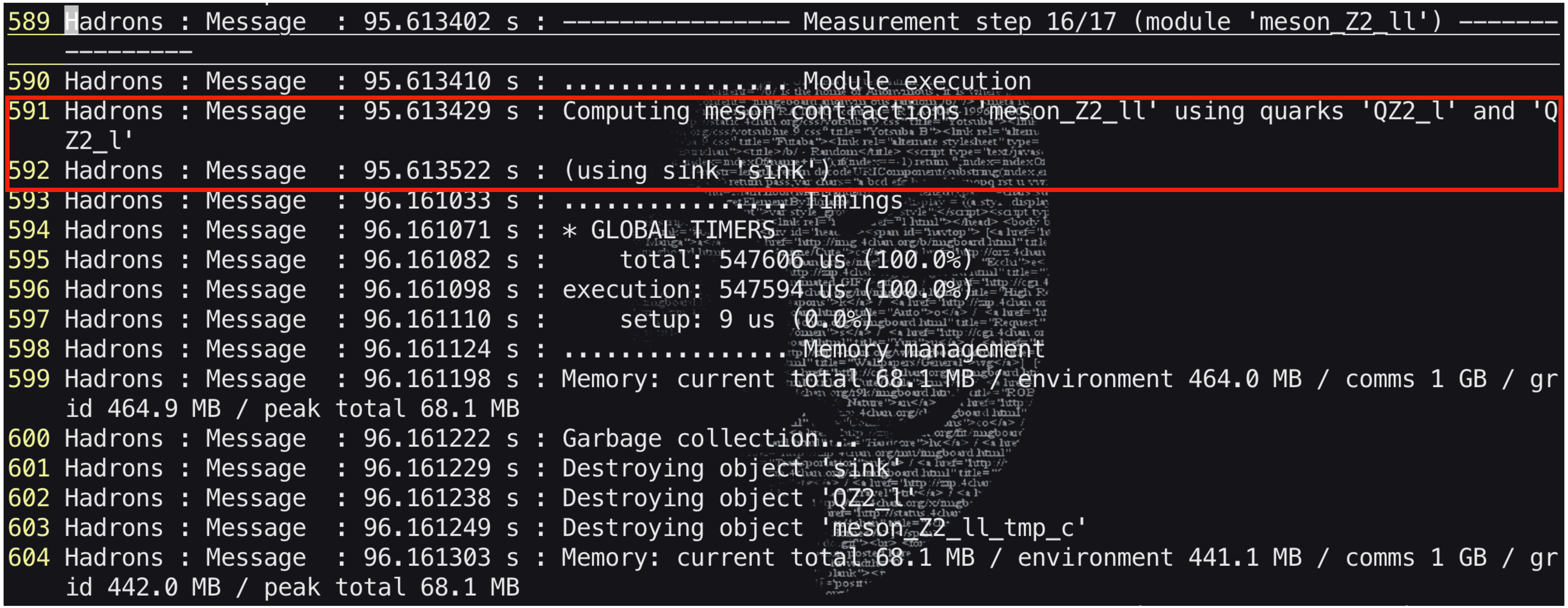
```
template<class vobj>
void ExtractSlice(Lattice<vobj> &lowDim,const Lattice<vobj> & higherDim,int slice, int orthog)
{
  typedef typename vobj::scalar_object sobj;

  GridBase *lg = lowDim.Grid();
  GridBase *hg = higherDim.Grid();
  int nl = lg->_ndimension;
  int nh = hg->_ndimension;

  assert(nl+1 == nh);
  assert(orthog<nh);
  assert(orthog>=0);
  assert(hg->_processors[orthog]==1);

  int dl; dl = 0;
  for(int d=0;d<nh;d++){
    if ( d != orthog) {
      assert(lg->_processors[dl]  == hg->_processors[d]);
      assert(lg->_ldimensions[dl] == hg->_ldimensions[d]);
      dl++;
    }
  }
  // the above should guarantee that the operations are local
  autoView(lowDimv,lowDim,CpuWrite);
  autoView(higherDimv,higherDim,CpuRead);
  thread_for(idx,lg->lSites(),{
    sobj s;
    Coordinate lcoor(nl);
    Coordinate hcoor(nh);
    lg->LocalIndexToLocalCoor(idx,lcoor);
    int ddl=0;
    hcoor[orthog] = slice;
    for(int d=0;d<nh;d++){
      if ( d!=orthog ) {
  hcoor[d]=lcoor[ddl++];
      }
    }
    peekLocalSite(s,higherDimv,hcoor);
    pokeLocalSite(s,lowDimv,lcoor);
  });

```

## Output file:



```
Hadrons : Message   : 65.331207 s : ---------------- Measurement step 15/17 (module 'QZ2_l') ----------------
Hadrons : Message   : 65.331216 s : ................ Module execution
Hadrons : Message   : 65.331259 s : Computing quark propagator 'QZ2_l'
Hadrons : Message   : 65.331271 s : Using source 'Z2'
Hadrons : Message   : 65.331284 s : Inverting using solver 'CG_l'
Hadrons : Message   : 65.331291 s : Inversion for spin= 0, color= 0
Hadrons : Message   : 65.331297 s : Import source
Hadrons : Message   : 65.338548 s : Solve
Grid    : Message   : 67.852095 s : ConjugateGradient Converged on iteration 318 Computed residual 9.5689e-09  True residual 9.5689e-09  Target 1e-08
Grid    : Message   : 67.860724 s : SchurRedBlackBase solver true unprec resid 3.25679e-08
Hadrons : Message   : 67.860793 s : Export solution
Hadrons : Message   : 67.879502 s : Inversion for spin= 0, color= 1
Hadrons : Message   : 67.879525 s : Import source
Hadrons : Message   : 67.886553 s : Solve
Grid    : Message   : 70.410080 s : ConjugateGradient Converged on iteration 319 Computed residual 9.62417e-09 True residual 9.62417e-09 Target 1e-08
Grid    : Message   : 70.418756 s : SchurRedBlackBase solver true unprec resid 3.23147e-08
Hadrons : Message   : 70.418829 s : Export solution
Hadrons : Message   : 70.437594 s : Inversion for spin= 0, color= 2
Hadrons : Message   : 70.437619 s : Import source
Hadrons : Message   : 70.444636 s : Solve
Grid    : Message   : 72.932850 s : ConjugateGradient Converged on iteration 314 Computed residual 9.68155e-09 True residual 9.68155e-09 Target 1e-08
Grid    : Message   : 72.941565 s : SchurRedBlackBase solver true unprec resid 3.30696e-08
Hadrons : Message   : 72.941636 s : Export solution
```

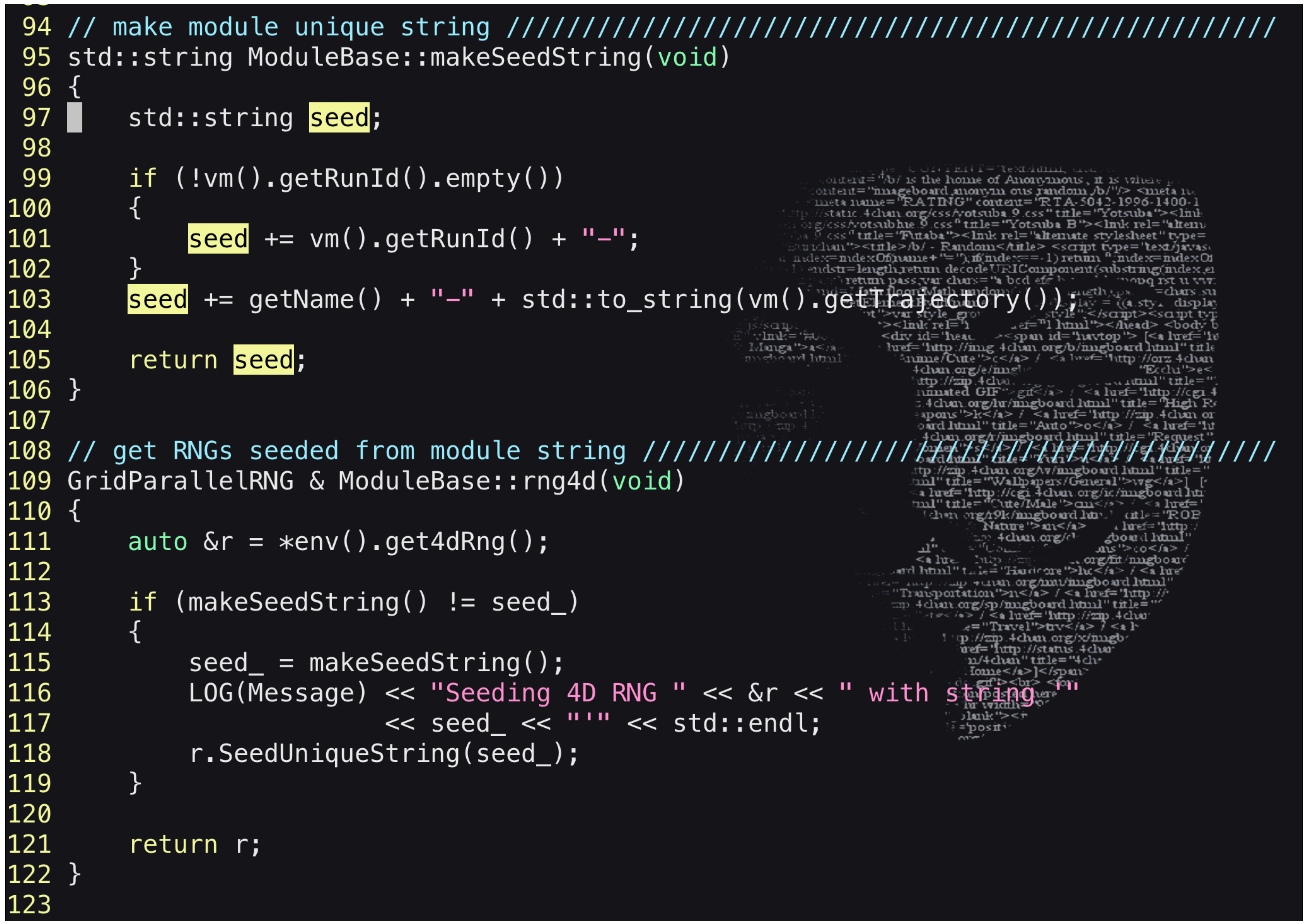
# Output file:



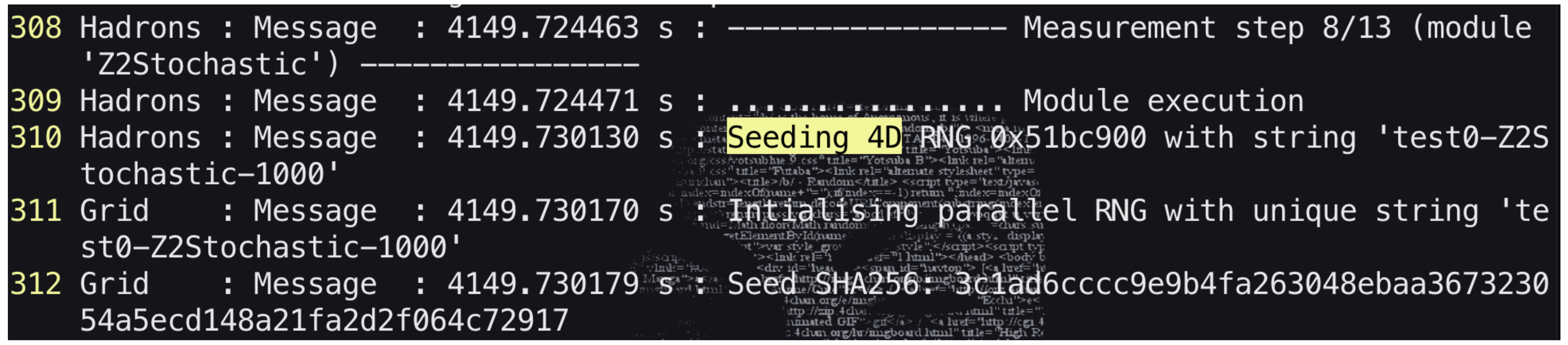
```
Hadrons : Message   : 95.613402 s : ---------------- Measurement step 16/17 (module 'meson_Z2_ll') ----------------
Hadrons : Message   : 95.613410 s : ................ Module execution
Hadrons : Message   : 95.613429 s : Computing meson contractions 'meson_Z2_ll' using quarks 'QZ2_l' and 'QZ2_l'
Hadrons : Message   : 95.613522 s : (using sink 'sink')
Hadrons : Message   : 96.161033 s : ................ Timings
Hadrons : Message   : 96.161071 s : * GLOBAL TIMERS
Hadrons : Message   : 96.161082 s :     total: 547606 us (100.0%)
Hadrons : Message   : 96.161098 s : execution: 547594 us (100.0%)
Hadrons : Message   : 96.161110 s :     setup: 9 us (0.0%)
Hadrons : Message   : 96.161124 s : ................ Memory management
Hadrons : Message   : 96.161198 s : Memory: current total 68.1 MB / environment 464.0 MB / comms 1 GB / grid 464.9 MB / peak total 68.1 MB
Hadrons : Message   : 96.161222 s : Garbage collection...
Hadrons : Message   : 96.161229 s : Destroying object 'sink'
Hadrons : Message   : 96.161238 s : Destroying object 'QZ2_l'
Hadrons : Message   : 96.161249 s : Destroying object 'meson_Z2_ll_tmp_c'
Hadrons : Message   : 96.161303 s : Memory: current total 68.1 MB / environment 441.1 MB / comms 1 GB / grid 442.0 MB / peak total 68.1 MB
```

- ./application-template/par-example.xml:37:
  <!-- run id (is part of seed for random numbers!) —>

# ./Hadrons/Module.cpp

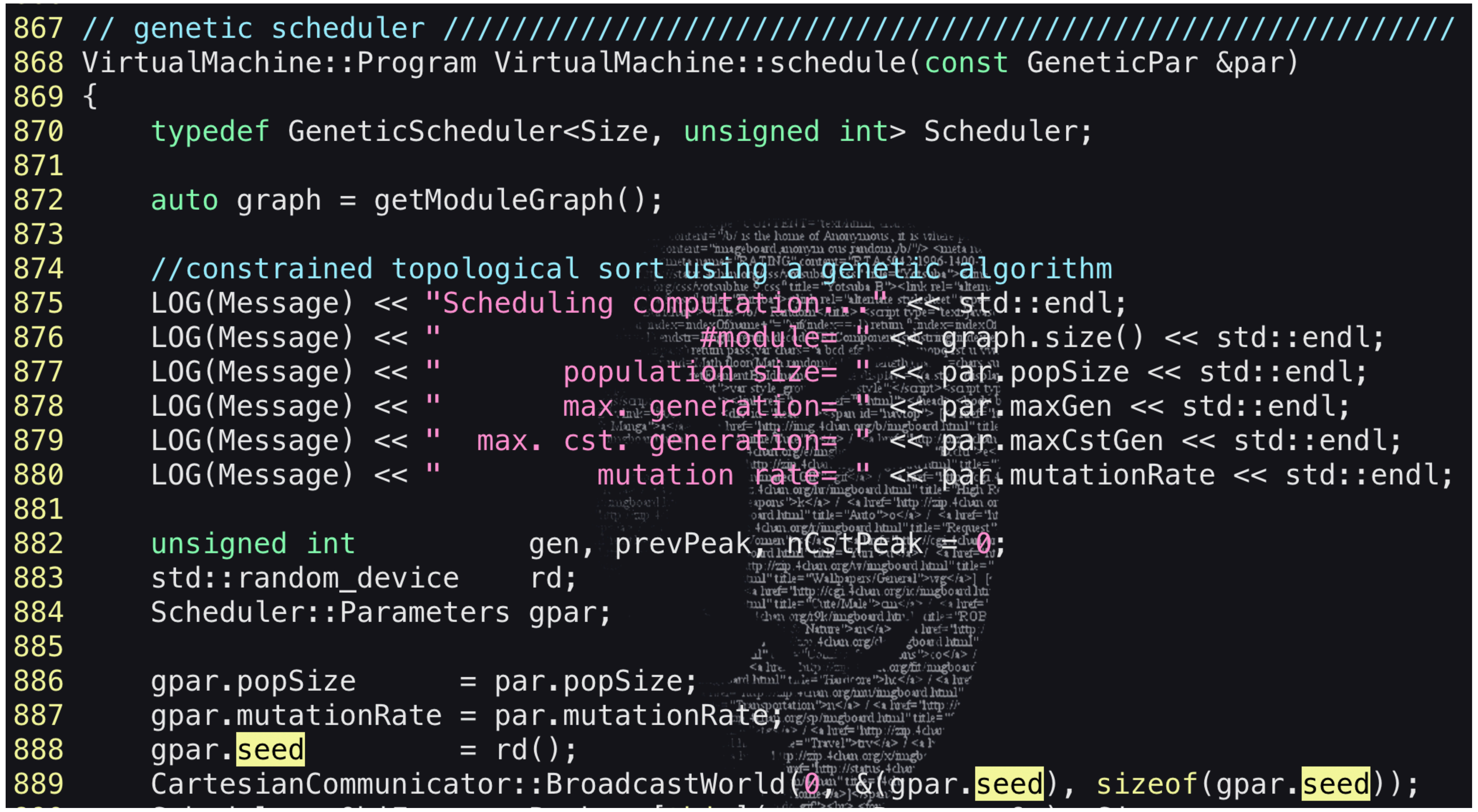


```cpp
// make module unique string ///////////////////////////////////////////////////
std::string ModuleBase::makeSeedString(void)
{
    std::string seed;

    if (!vm().getRunId().empty())
    {
        seed += vm().getRunId() + "-";
    }
    seed += getName() + "-" + std::to_string(vm().getTrajectory());

    return seed;
}

// get RNGs seeded from module string //////////////////////////////////////////
GridParallelRNG & ModuleBase::rng4d(void)
{
    auto &r = *env().get4dRng();

    if (makeSeedString() != seed_)
    {
        seed_ = makeSeedString();
        LOG(Message) << "Seeding 4D RNG " << &r << " with string '"
                     << seed_ << "'" << std::endl;
        r.SeedUniqueString(seed_);
    }

    return r;
}
```

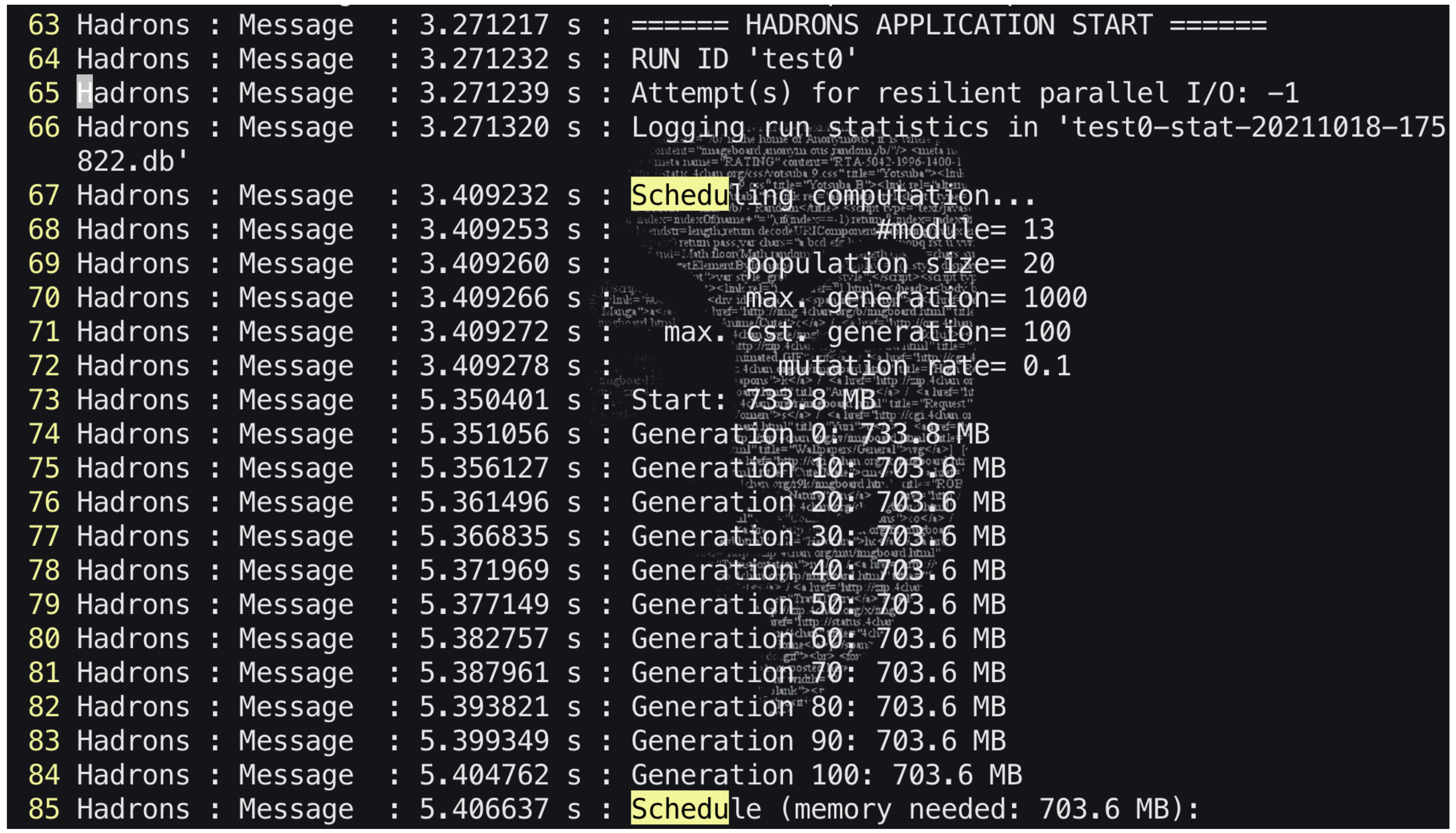


```
Hadrons : Message  : 4149.724463 s : ---------------- Measurement step 8/13 (module 'Z2Stochastic') ----------------
Hadrons : Message  : 4149.724471 s : ................ Module execution
Hadrons : Message  : 4149.730130 s : Seeding 4D RNG 0x51bc900 with string 'test0-Z2Stochastic-1000'
Grid    : Message  : 4149.730170 s : Intialising parallel RNG with unique string 'test0-Z2Stochastic-1000'
Grid    : Message  : 4149.730179 s : Seed SHA256: 3c1ad6cccc9e9b4fa263048ebaa367323054a5ecd148a21fa2d2f064c72917
```

# ./Hadrons/VirtualMachine.cpp



```cpp
// genetic scheduler /////////////////////////////////////////////////////////////
VirtualMachine::Program VirtualMachine::schedule(const GeneticPar &par)
{
    typedef GeneticScheduler<Size, unsigned int> Scheduler;

    auto graph = getModuleGraph();

    //constrained topological sort using a genetic algorithm
    LOG(Message) << "Scheduling computation..." << std::endl;
    LOG(Message) << "               #module= " << graph.size() << std::endl;
    LOG(Message) << "       population size= " << par.popSize << std::endl;
    LOG(Message) << "        max. generation= " << par.maxGen << std::endl;
    LOG(Message) << "  max. cst. generation= " << par.maxCstGen << std::endl;
    LOG(Message) << "         mutation rate= " << par.mutationRate << std::endl;

    unsigned int          gen, prevPeak, nCstPeak = 0;
    std::random_device    rd;
    Scheduler::Parameters gpar;

    gpar.popSize      = par.popSize;
    gpar.mutationRate = par.mutationRate;
    gpar.seed         = rd();
    CartesianCommunicator::BroadcastWorld(0, &(gpar.seed), sizeof(gpar.seed));
```



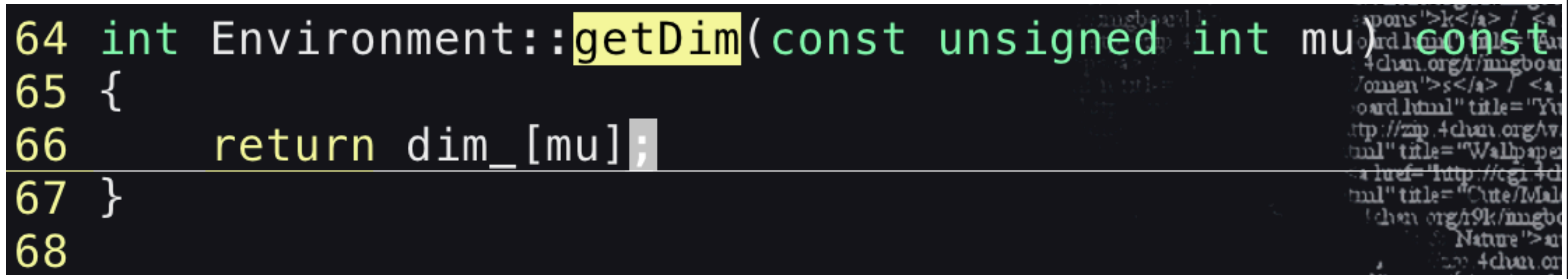
```
Hadrons : Message  : 3.271217 s : ====== HADRONS APPLICATION START ======
Hadrons : Message  : 3.271232 s : RUN ID 'test0'
Hadrons : Message  : 3.271239 s : Attempt(s) for resilient parallel I/O: -1
Hadrons : Message  : 3.271320 s : Logging run statistics in 'test0-stat-20211018-175
822.db'
Hadrons : Message  : 3.409232 s : Scheduling computation...
Hadrons : Message  : 3.409253 s :                #module= 13
Hadrons : Message  : 3.409260 s :         population size= 20
Hadrons : Message  : 3.409266 s :         max. generation= 1000
Hadrons : Message  : 3.409272 s :   max. cst. generation= 100
Hadrons : Message  : 3.409278 s :          mutation rate= 0.1
Hadrons : Message  : 5.350401 s : Start: 733.8 MB
Hadrons : Message  : 5.351056 s : Generation 0: 733.8 MB
Hadrons : Message  : 5.356127 s : Generation 10: 703.6 MB
Hadrons : Message  : 5.361496 s : Generation 20: 703.6 MB
Hadrons : Message  : 5.366835 s : Generation 30: 703.6 MB
Hadrons : Message  : 5.371969 s : Generation 40: 703.6 MB
Hadrons : Message  : 5.377149 s : Generation 50: 703.6 MB
Hadrons : Message  : 5.382757 s : Generation 60: 703.6 MB
Hadrons : Message  : 5.387961 s : Generation 70: 703.6 MB
Hadrons : Message  : 5.393821 s : Generation 80: 703.6 MB
Hadrons : Message  : 5.399349 s : Generation 90: 703.6 MB
Hadrons : Message  : 5.404762 s : Generation 100: 703.6 MB
Hadrons : Message  : 5.406637 s : Schedule (memory needed: 703.6 MB):
```

```
//template <class Prop, class Ferm>
template <class Fimpl>
void PropToFerm(typename Fimpl::FermionField &f, const typename Fimpl::PropagatorField &p, const int s, const int c)
{
  for(int j = 0; j < Ns; ++j)
    {
      auto pjs = peekSpin(p, j, s);
      auto fj  = peekSpin(f, j);

      for(int i = 0; i < Fimpl::Dimension; ++i)
  {
    pokeColour(fj, peekColour(pjs, i, c), i);
  }
      pokeSpin(f, fj, j);
    }
}
```

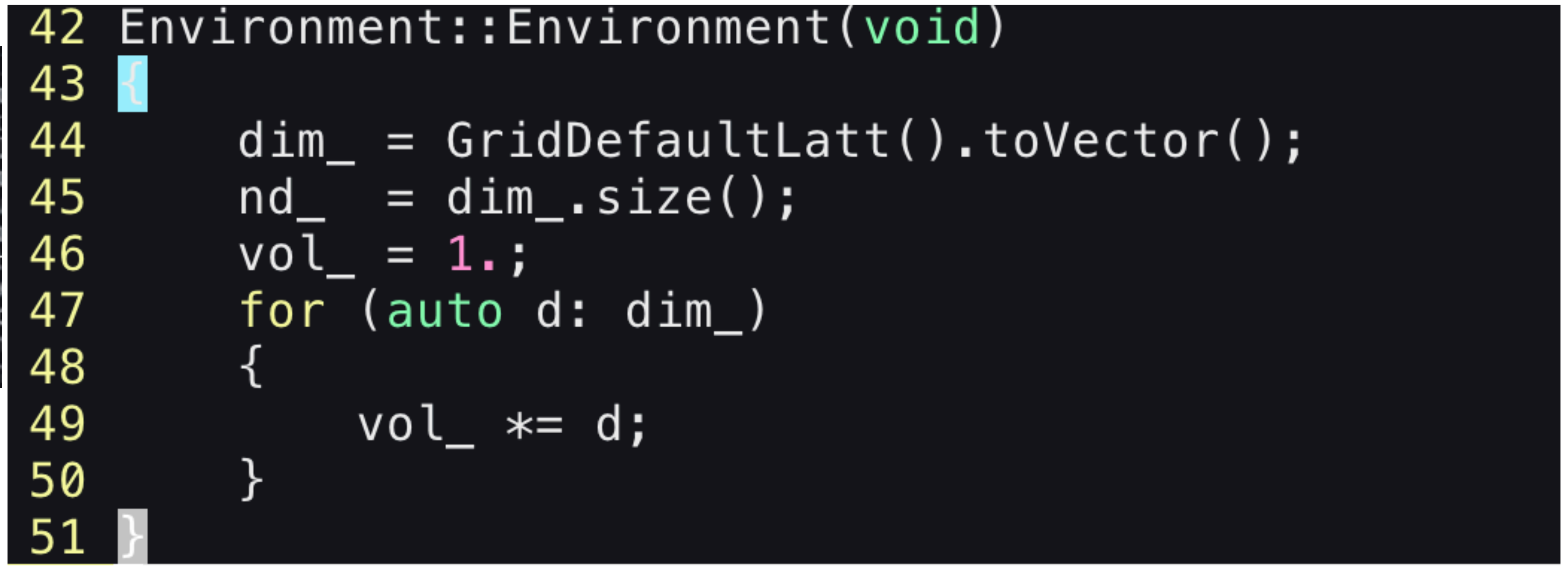
$$\mathrm{pjs} = \mathrm{peekSpin}(\mathrm{p}, \mathrm{j}, \mathrm{s}) : \mathrm{pjs} = \mathrm{p}_{\mathrm{js}}\,; \qquad \mathrm{fj} = \mathrm{peekSpin}(\mathrm{f}, \mathrm{j}) : \mathrm{fj} = \mathrm{f}_{\mathrm{j}}$$

$$\mathrm{peekColour}(\mathrm{pjs}, \mathrm{i}, \mathrm{c}) : \underset{\mathrm{ic}}{\mathrm{p}_{\mathrm{js}}}; \qquad \mathrm{pokeColour}(\mathrm{fj}, \mathrm{peekColour}(\mathrm{pjs}, \mathrm{i}, \mathrm{c}), \mathrm{i}) : \mathrm{f}_{\mathrm{j}_{\mathrm{i}}} = \underset{\mathrm{ic}}{\mathrm{p}_{\mathrm{js}}}$$

$$\mathrm{pokeSpin}(\mathrm{f}, \mathrm{fj}, \mathrm{j}) : \mathrm{f}_{\mathrm{j}} = \mathrm{f}_{\mathrm{j}}$$



```
int Environment::getDim(const unsigned int mu) const
{
    return dim_[mu];
}

```



```
Environment::Environment(void)
{
    dim_ = GridDefaultLatt().toVector();
    nd_  = dim_.size();
    vol_ = 1.;
    for (auto d: dim_)
    {
        vol_ *= d;
    }
}
```

## 15.2 Grid Initialization

Grid itself is initialized with a call:

```
Grid_init(&argc, &argv);
```

Command line options include:

```
--mpi n.n.n.n   : default MPI decomposition
--threads n     : default number of OMP threads
--grid n.n.n.n  : default Grid size
```

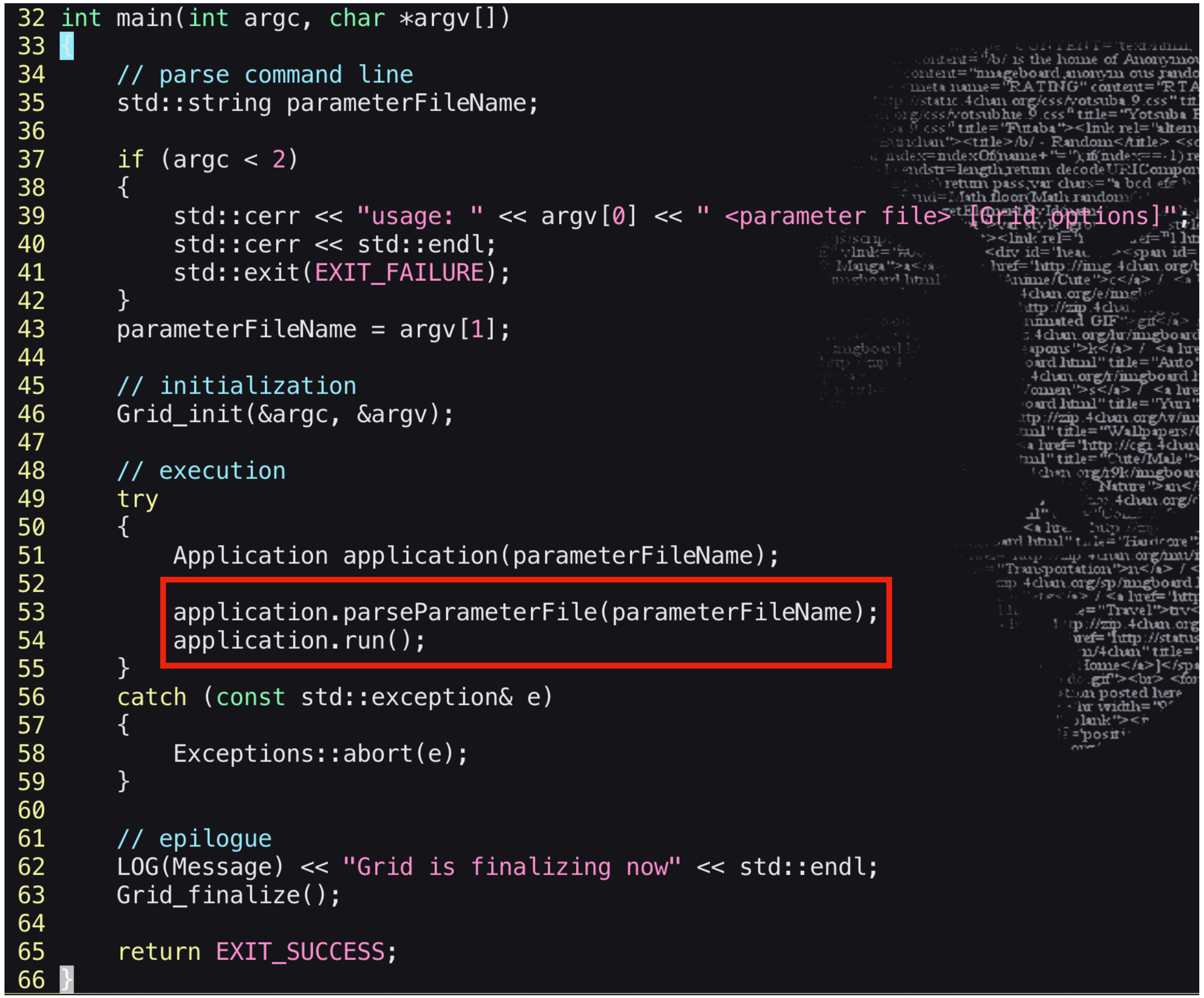
where *argc* and *argv* are constructed to simulate the command-line options described above. At a minimum one usually provides the *–grid* and *–mpi* parameters. The former specifies the lattice dimensions and the latter specifies the grid of processors (MPI ranks). If these parameters are not specified with the *Grid_init* call, they need to be supplied later when creating Grid fields.

The following Grid procedures are useful for verifying that Grid "default" values are properly initialized.

| Grid procedure | returns |
|---|---|
| std::vector<int> GridDefaultLatt(); | lattice size |
| std::vector<int> GridDefaultSimd(int Nd,vComplex::Nsimd()); | SIMD layout |
| std::vector<int> GridDefaultMpi(); | MPI layout |
| int Grid::GridThread::GetThreads(); | number of threads |

## Code running procedure

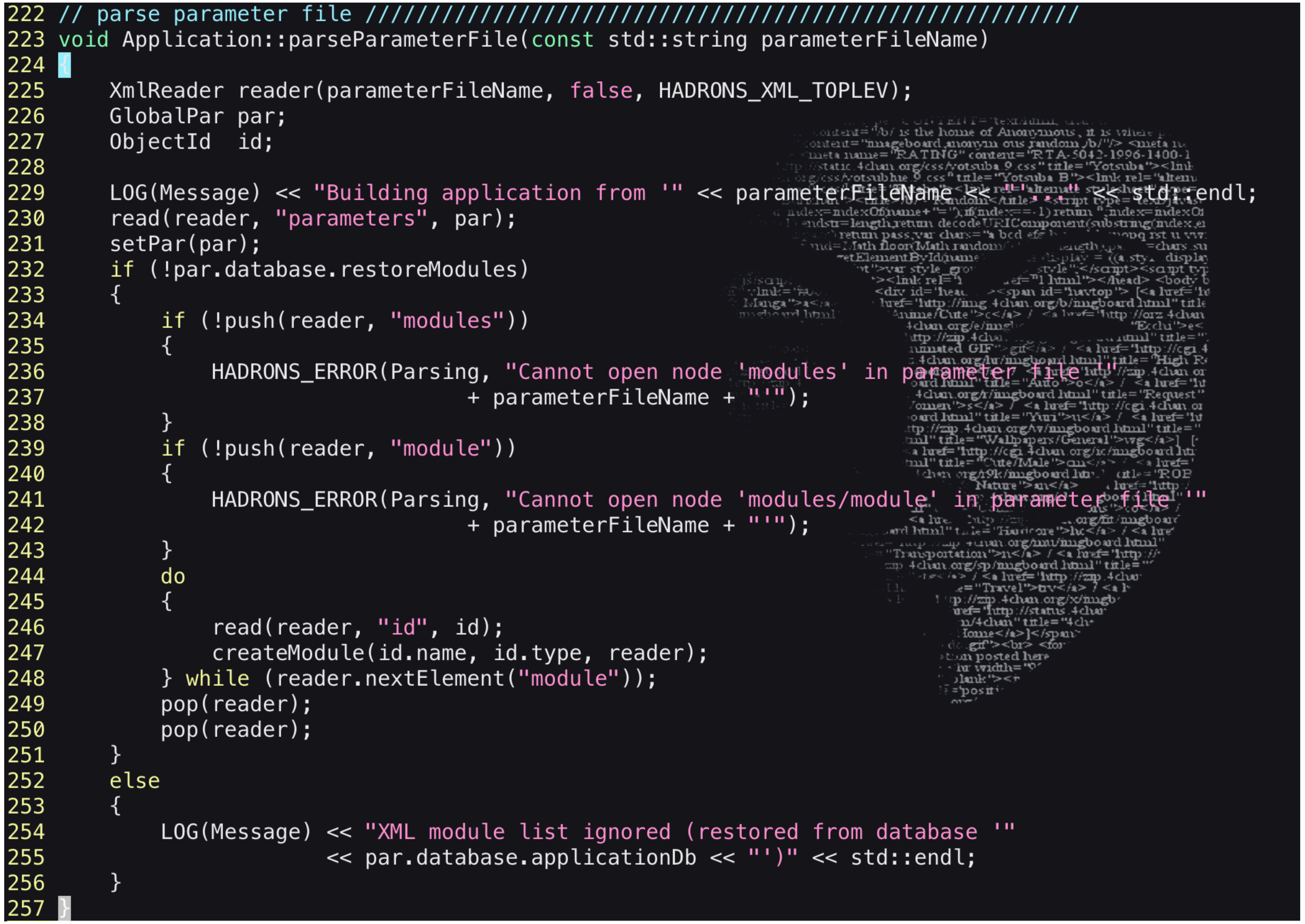
# utilities/HadronsXmlRun.cpp



```
int main(int argc, char *argv[])
{
    // parse command line
    std::string parameterFileName;

    if (argc < 2)
    {
        std::cerr << "usage: " << argv[0] << " <parameter file> [Grid options]";
        std::cerr << std::endl;
        std::exit(EXIT_FAILURE);
    }
    parameterFileName = argv[1];

    // initialization
    Grid_init(&argc, &argv);

    // execution
    try
    {
        Application application(parameterFileName);

        application.parseParameterFile(parameterFileName);
        application.run();
    }
    catch (const std::exception& e)
    {
        Exceptions::abort(e);
    }

    // epilogue
    LOG(Message) << "Grid is finalizing now" << std::endl;
    Grid_finalize();

    return EXIT_SUCCESS;
}
```

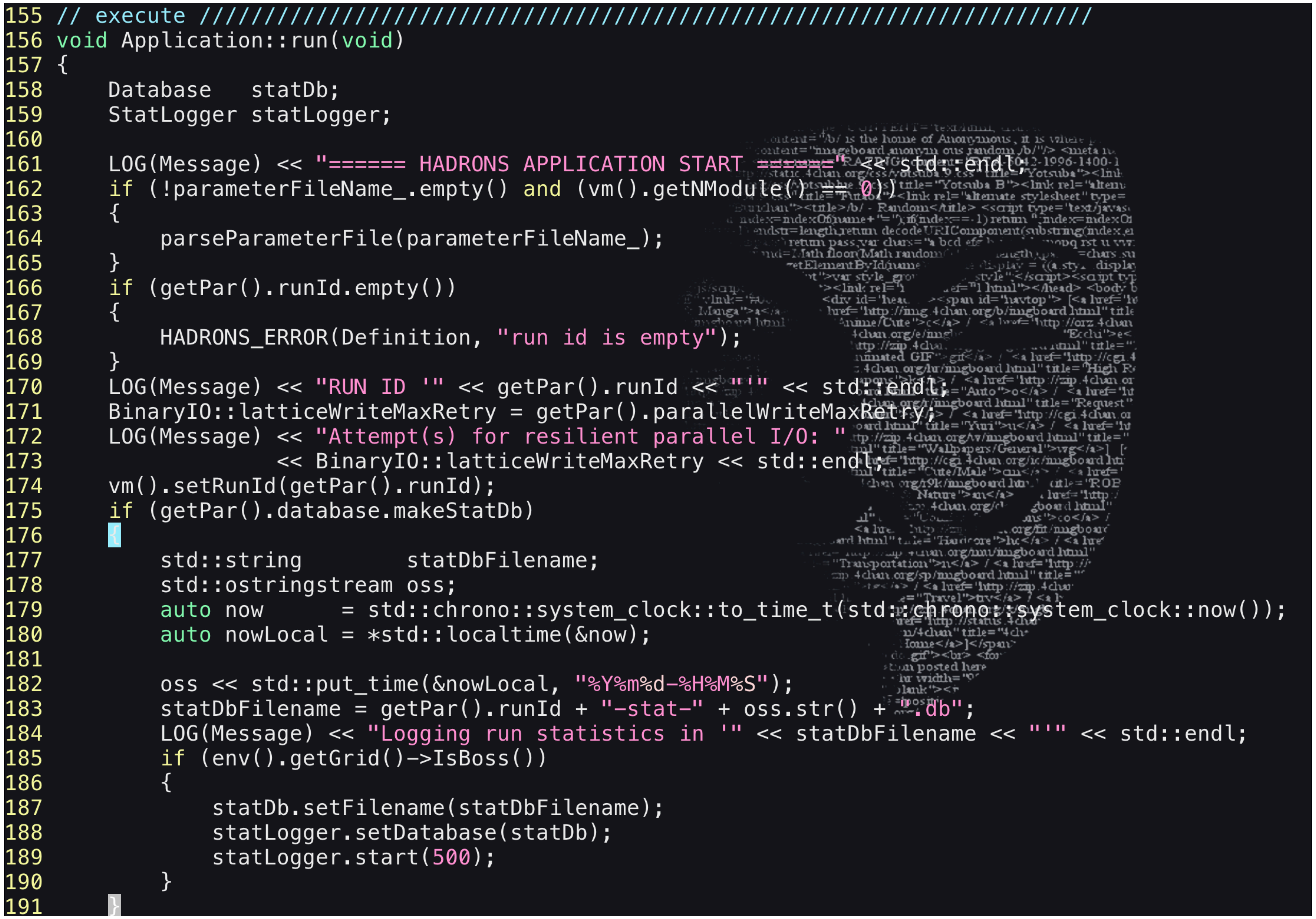
The main function of every GRID program starts with an initialization Grid_init() and ends with the corresponding Grid_finalize() function call

# Application.cpp



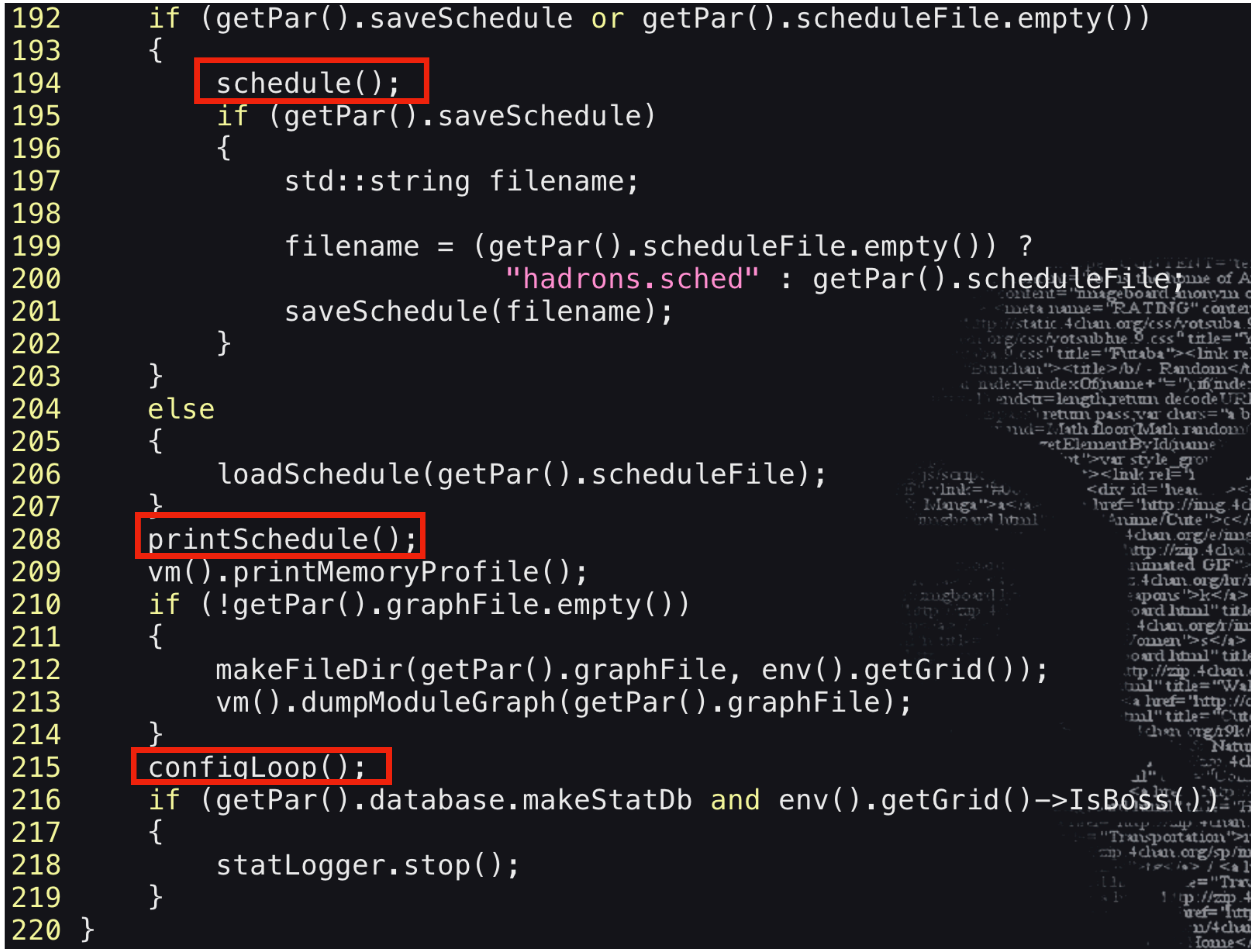
```cpp
// parse parameter file ///////////////////////////////////////////////////////
void Application::parseParameterFile(const std::string parameterFileName)
{
    XmlReader reader(parameterFileName, false, HADRONS_XML_TOPLEV);
    GlobalPar par;
    ObjectId  id;

    LOG(Message) << "Building application from '" << parameterFileName << "'..." << std::endl;
    read(reader, "parameters", par);
    setPar(par);
    if (!par.database.restoreModules)
    {
        if (!push(reader, "modules"))
        {
            HADRONS_ERROR(Parsing, "Cannot open node 'modules' in parameter file '"
                                   + parameterFileName + "'");
        }
        if (!push(reader, "module"))
        {
            HADRONS_ERROR(Parsing, "Cannot open node 'modules/module' in parameter file '"
                                   + parameterFileName + "'");
        }
        do
        {
            read(reader, "id", id);
            createModule(id.name, id.type, reader);
        } while (reader.nextElement("module"));
        pop(reader);
        pop(reader);
    }
    else
    {
        LOG(Message) << "XML module list ignored (restored from database '"
                     << par.database.applicationDb << "')" << std::endl;
    }
}
```

# Application.cpp



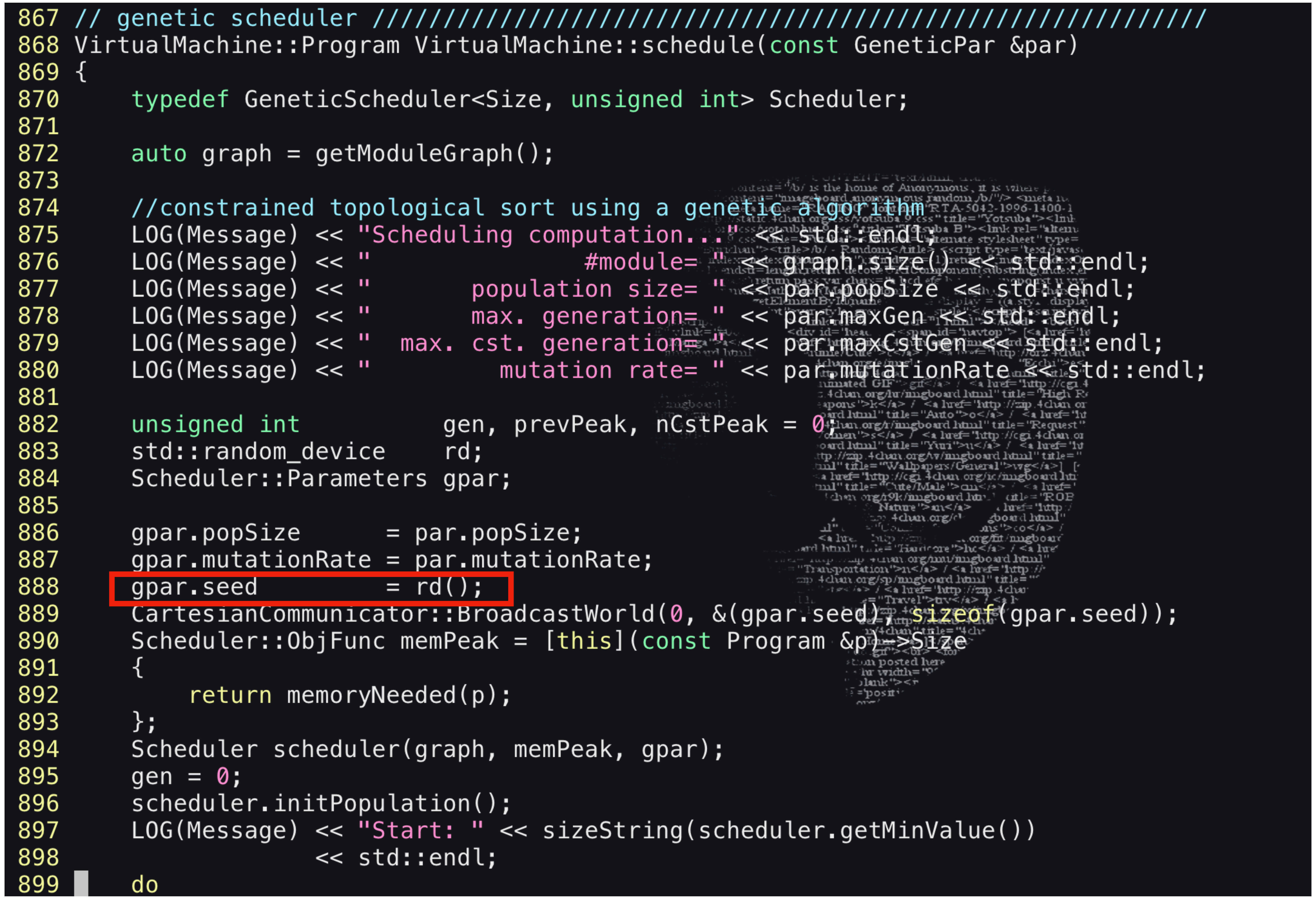
```cpp
// execute /////////////////////////////////////////////////////////////////////
void Application::run(void)
{
    Database   statDb;
    StatLogger statLogger;

    LOG(Message) << "====== HADRONS APPLICATION START ======" << std::endl;
    if (!parameterFileName_.empty() and (vm().getNModule() == 0))
    {
        parseParameterFile(parameterFileName_);
    }
    if (getPar().runId.empty())
    {
        HADRONS_ERROR(Definition, "run id is empty");
    }
    LOG(Message) << "RUN ID '" << getPar().runId << "'" << std::endl;
    BinaryIO::latticeWriteMaxRetry = getPar().parallelWriteMaxRetry;
    LOG(Message) << "Attempt(s) for resilient parallel I/O: "
                 << BinaryIO::latticeWriteMaxRetry << std::endl;
    vm().setRunId(getPar().runId);
    if (getPar().database.makeStatDb)
    {
        std::string        statDbFilename;
        std::ostringstream oss;
        auto now      = std::chrono::system_clock::to_time_t(std::chrono::system_clock::now());
        auto nowLocal = *std::localtime(&now);

        oss << std::put_time(&nowLocal, "%Y%m%d-%H%M%S");
        statDbFilename = getPar().runId + "-stat-" + oss.str() + ".db";
        LOG(Message) << "Logging run statistics in '" << statDbFilename << "'" << std::endl;
        if (env().getGrid()->IsBoss())
        {
            statDb.setFilename(statDbFilename);
            statLogger.setDatabase(statDb);
            statLogger.start(500);
        }
    }
```

# Application.cpp



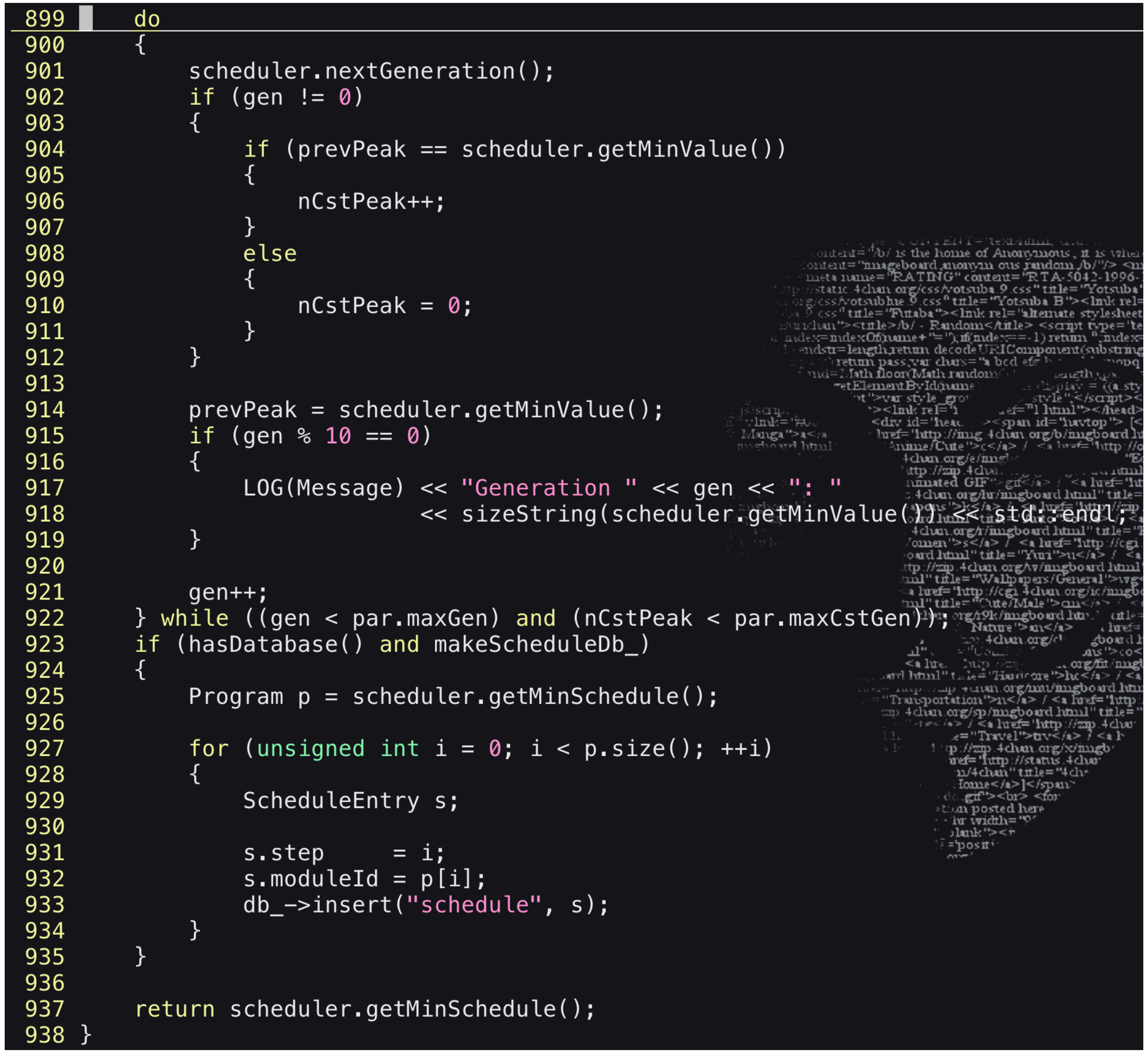
```
    if (getPar().saveSchedule or getPar().scheduleFile.empty())
    {
        schedule();
        if (getPar().saveSchedule)
        {
            std::string filename;

            filename = (getPar().scheduleFile.empty()) ?
                         "hadrons.sched" : getPar().scheduleFile;
            saveSchedule(filename);
        }
    }
    else
    {
        loadSchedule(getPar().scheduleFile);
    }
    printSchedule();
    vm().printMemoryProfile();
    if (!getPar().graphFile.empty())
    {
        makeFileDir(getPar().graphFile, env().getGrid());
        vm().dumpModuleGraph(getPar().graphFile);
    }
    configLoop();
    if (getPar().database.makeStatDb and env().getGrid()->IsBoss())
    {
        statLogger.stop();
    }
}
```

# VirtualMachine.cpp



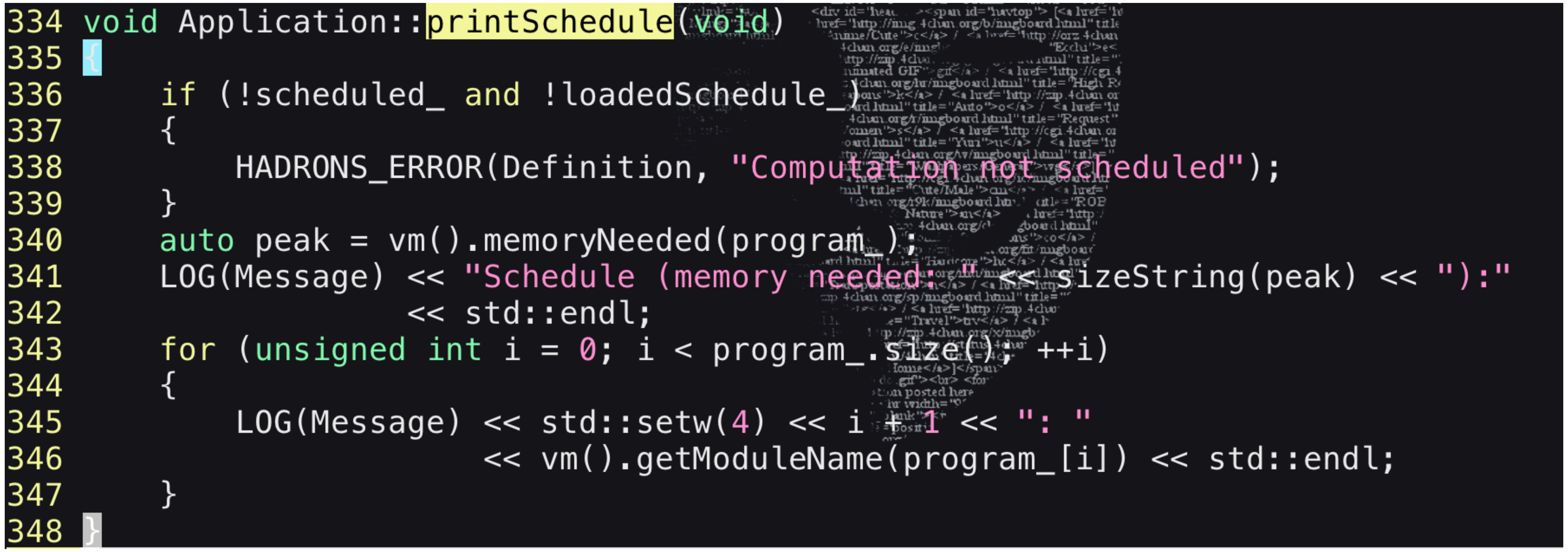
```
// genetic scheduler ////////////////////////////////////////////////////////////
VirtualMachine::Program VirtualMachine::schedule(const GeneticPar &par)
{
    typedef GeneticScheduler<Size, unsigned int> Scheduler;

    auto graph = getModuleGraph();

    //constrained topological sort using a genetic algorithm
    LOG(Message) << "Scheduling computation..." << std::endl;
    LOG(Message) << "               #module= " << graph.size() << std::endl;
    LOG(Message) << "       population size= " << par.popSize << std::endl;
    LOG(Message) << "       max. generation= " << par.maxGen << std::endl;
    LOG(Message) << "  max. cst. generation= " << par.maxCstGen << std::endl;
    LOG(Message) << "         mutation rate= " << par.mutationRate << std::endl;

    unsigned int          gen, prevPeak, nCstPeak = 0;
    std::random_device    rd;
    Scheduler::Parameters gpar;

    gpar.popSize      = par.popSize;
    gpar.mutationRate = par.mutationRate;
    gpar.seed         = rd();
    CartesianCommunicator::BroadcastWorld(0, &(gpar.seed), sizeof(gpar.seed));
    Scheduler::ObjFunc memPeak = [this](const Program &p)->Size
    {
        return memoryNeeded(p);
    };
    Scheduler scheduler(graph, memPeak, gpar);
    gen = 0;
    scheduler.initPopulation();
    LOG(Message) << "Start: " << sizeString(scheduler.getMinValue())
                 << std::endl;
    do
```

# VirtualMachine.cpp



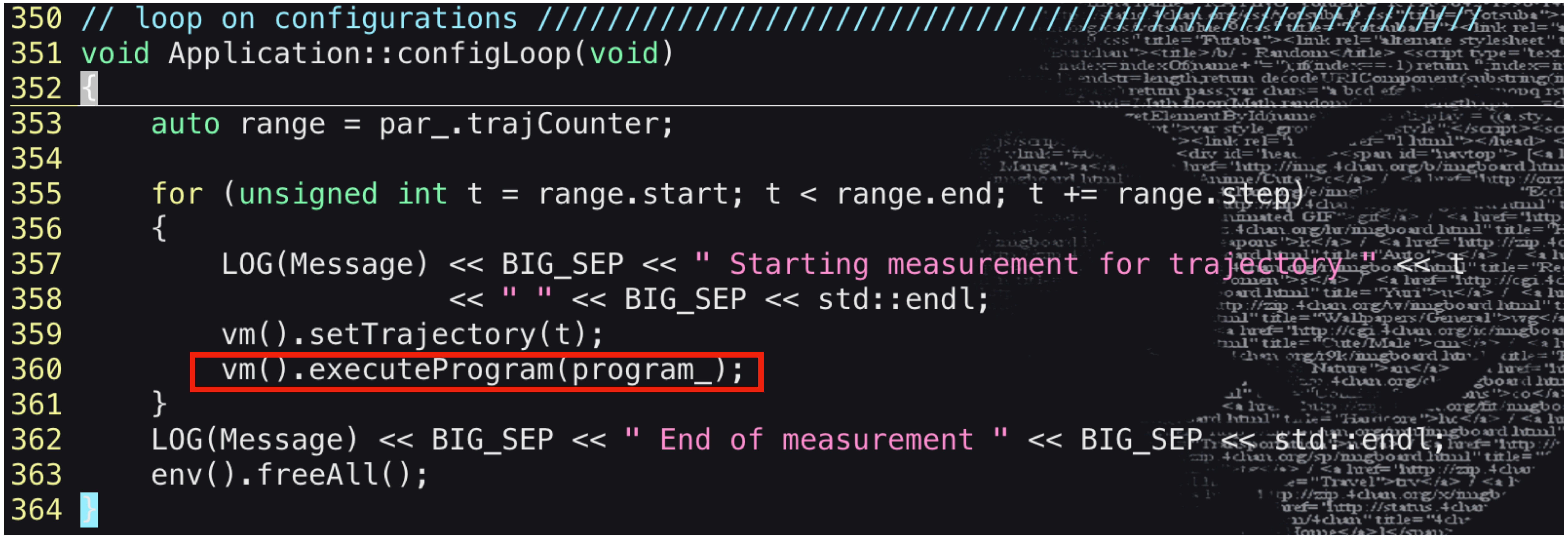
```cpp
    do
    {
        scheduler.nextGeneration();
        if (gen != 0)
        {
            if (prevPeak == scheduler.getMinValue())
            {
                nCstPeak++;
            }
            else
            {
                nCstPeak = 0;
            }
        }

        prevPeak = scheduler.getMinValue();
        if (gen % 10 == 0)
        {
            LOG(Message) << "Generation " << gen << ": "
                         << sizeString(scheduler.getMinValue()) << std::endl;
        }

        gen++;
    } while ((gen < par.maxGen) and (nCstPeak < par.maxCstGen));
    if (hasDatabase() and makeScheduleDb_)
    {
        Program p = scheduler.getMinSchedule();

        for (unsigned int i = 0; i < p.size(); ++i)
        {
            ScheduleEntry s;

            s.step     = i;
            s.moduleId = p[i];
            db_->insert("schedule", s);
        }
    }

    return scheduler.getMinSchedule();
}
```

# Application.cpp



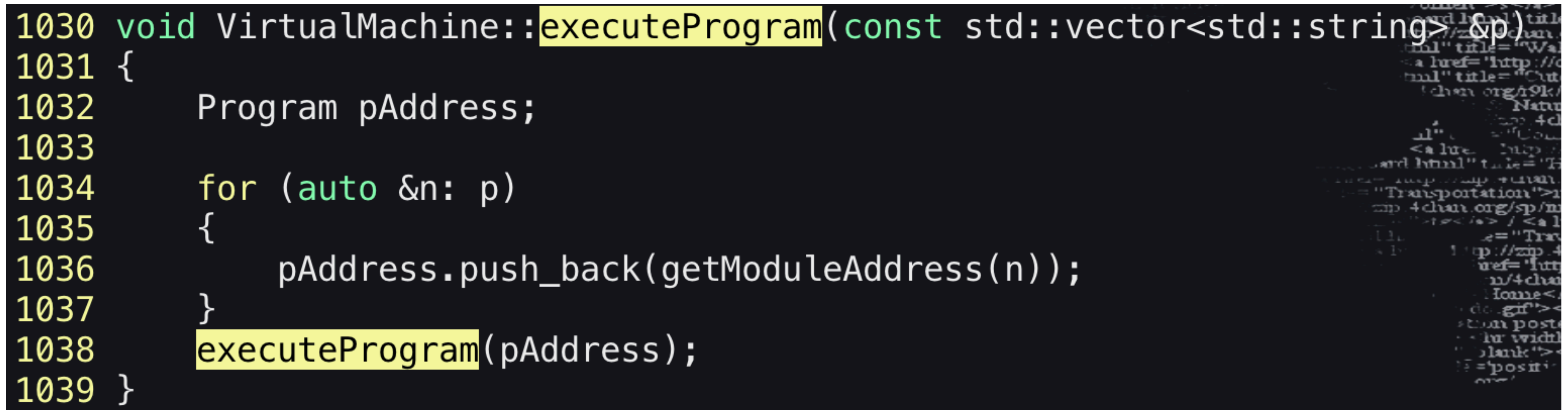
```cpp
void Application::printSchedule(void)
{
    if (!scheduled_ and !loadedSchedule_)
    {
        HADRONS_ERROR(Definition, "Computation not scheduled");
    }
    auto peak = vm().memoryNeeded(program_);
    LOG(Message) << "Schedule (memory needed: " << sizeString(peak) << "):"
                 << std::endl;
    for (unsigned int i = 0; i < program_.size(); ++i)
    {
        LOG(Message) << std::setw(4) << i + 1 << ": "
                     << vm().getModuleName(program_[i]) << std::endl;
    }
}
```

# Application.cpp

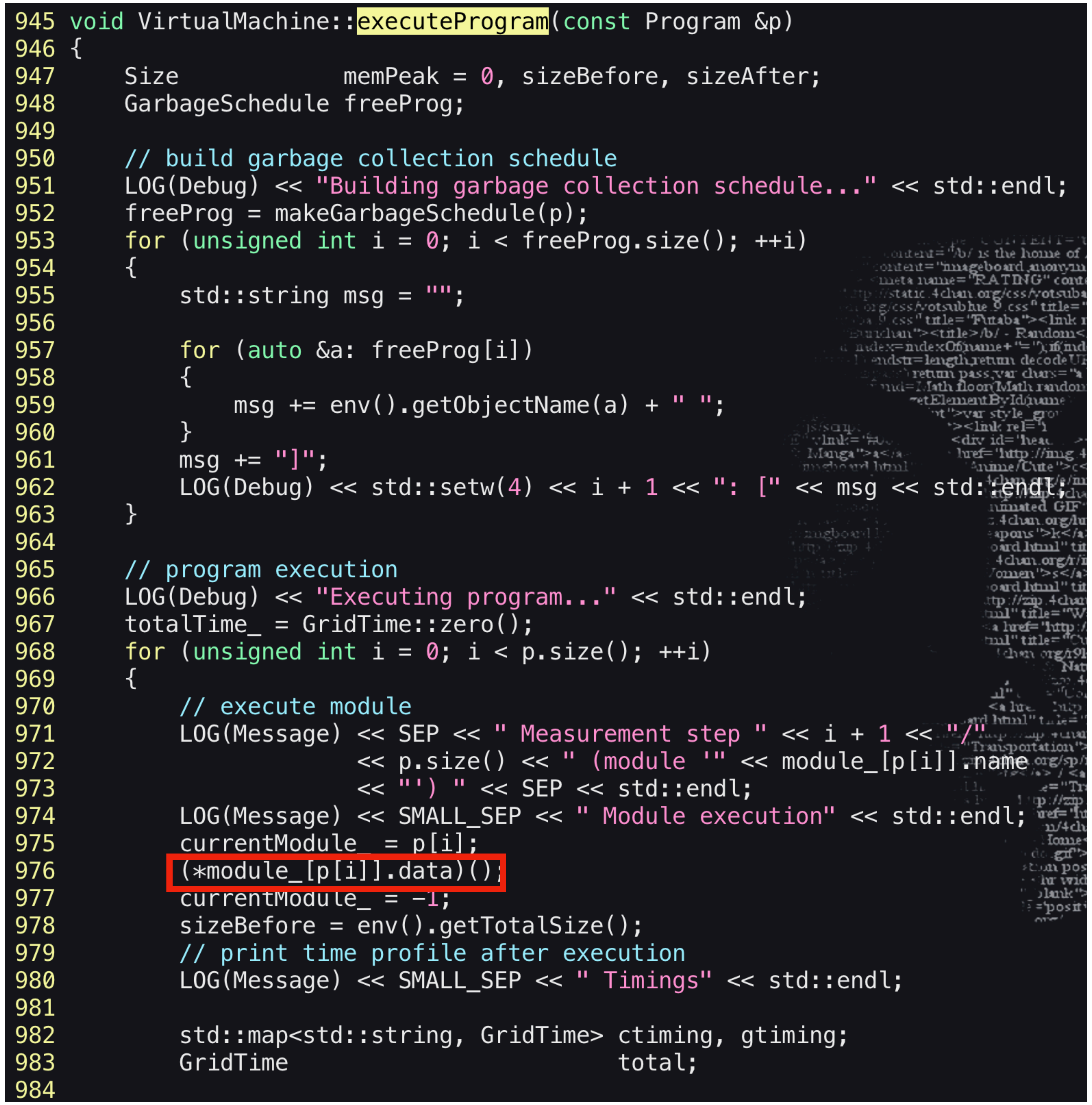


```
// loop on configurations ///////////////////////////////////////////////////////////////
void Application::configLoop(void)
{
    auto range = par_.trajCounter;

    for (unsigned int t = range.start; t < range.end; t += range.step)
    {
        LOG(Message) << BIG_SEP << " Starting measurement for trajectory " << t
                     << " " << BIG_SEP << std::endl;
        vm().setTrajectory(t);
        vm().executeProgram(program_);
    }
    LOG(Message) << BIG_SEP << " End of measurement " << BIG_SEP << std::endl;
    env().freeAll();
}
```

# VirtualMachine.cpp

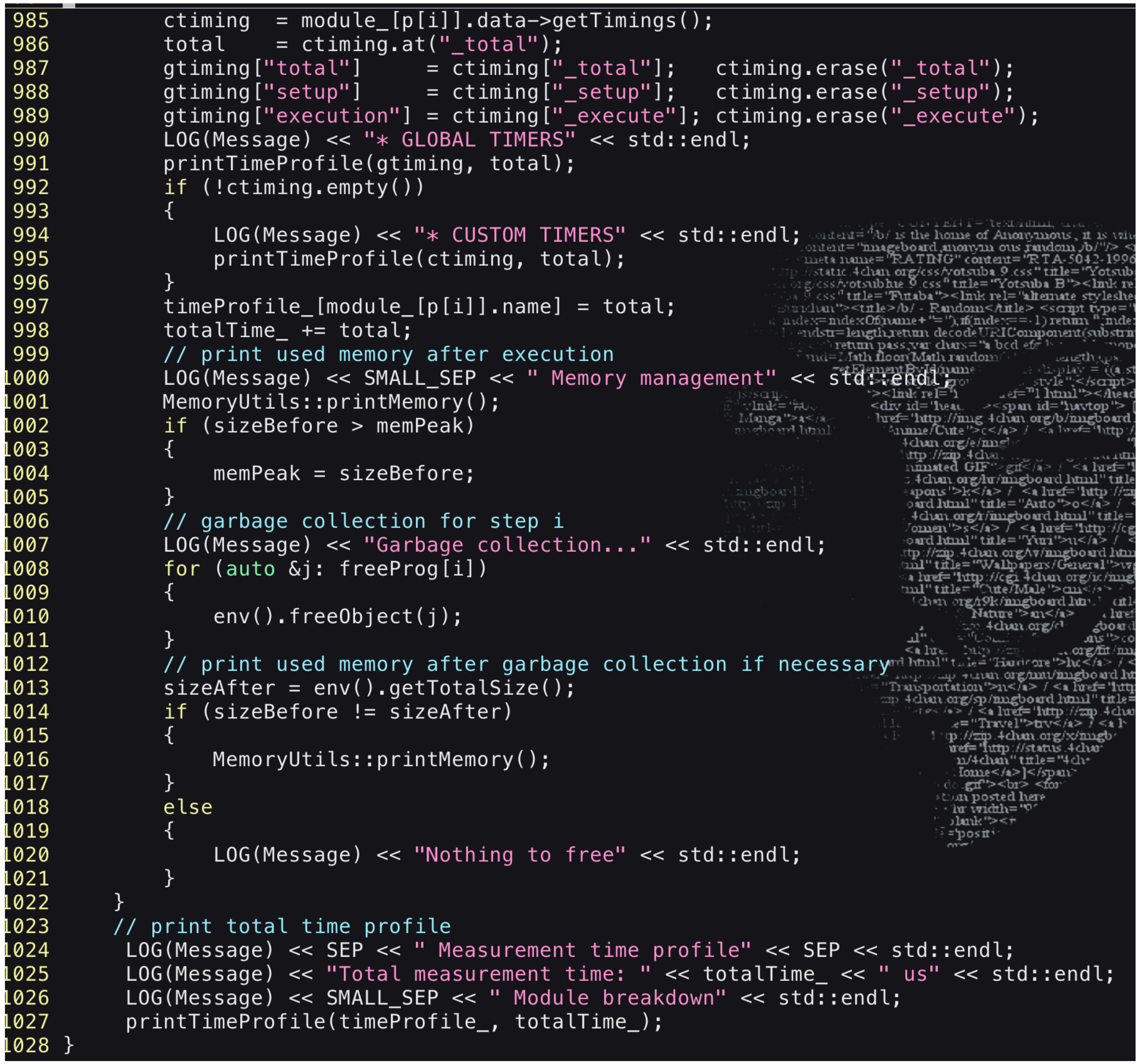


```cpp
void VirtualMachine::executeProgram(const std::vector<std::string> &p)
{
    Program pAddress;

    for (auto &n: p)
    {
        pAddress.push_back(getModuleAddress(n));
    }
    executeProgram(pAddress);
}
```



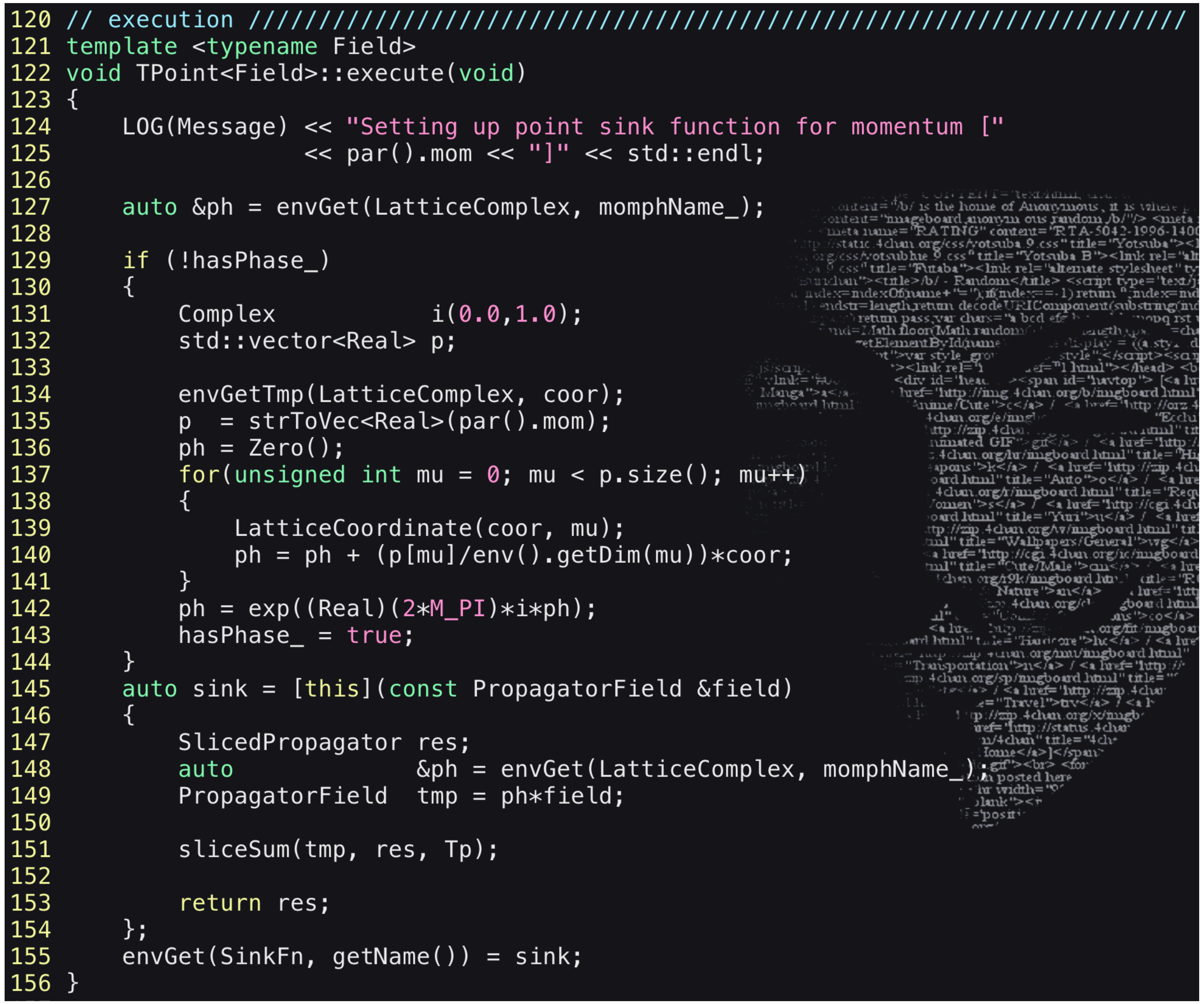
```cpp
void VirtualMachine::executeProgram(const Program &p)
{
    Size            memPeak = 0, sizeBefore, sizeAfter;
    GarbageSchedule freeProg;

    // build garbage collection schedule
    LOG(Debug) << "Building garbage collection schedule..." << std::endl;
    freeProg = makeGarbageSchedule(p);
    for (unsigned int i = 0; i < freeProg.size(); ++i)
    {
        std::string msg = "";

        for (auto &a: freeProg[i])
        {
            msg += env().getObjectName(a) + " ";
        }
        msg += "]";
        LOG(Debug) << std::setw(4) << i + 1 << ": [" << msg << std::endl;
    }

    // program execution
    LOG(Debug) << "Executing program..." << std::endl;
    totalTime_ = GridTime::zero();
    for (unsigned int i = 0; i < p.size(); ++i)
    {
        // execute module
        LOG(Message) << SEP << " Measurement step " << i + 1 << "/"
                     << p.size() << " (module '" << module_[p[i]].name
                     << "') " << SEP << std::endl;
        LOG(Message) << SMALL_SEP << " Module execution" << std::endl;
        currentModule_ = p[i];
        (*module_[p[i]].data)();
        currentModule_ = -1;
        sizeBefore = env().getTotalSize();
        // print time profile after execution
        LOG(Message) << SMALL_SEP << " Timings" << std::endl;

        std::map<std::string, GridTime> ctiming, gtiming;
        GridTime                        total;

```

# VirtualMachine.cpp



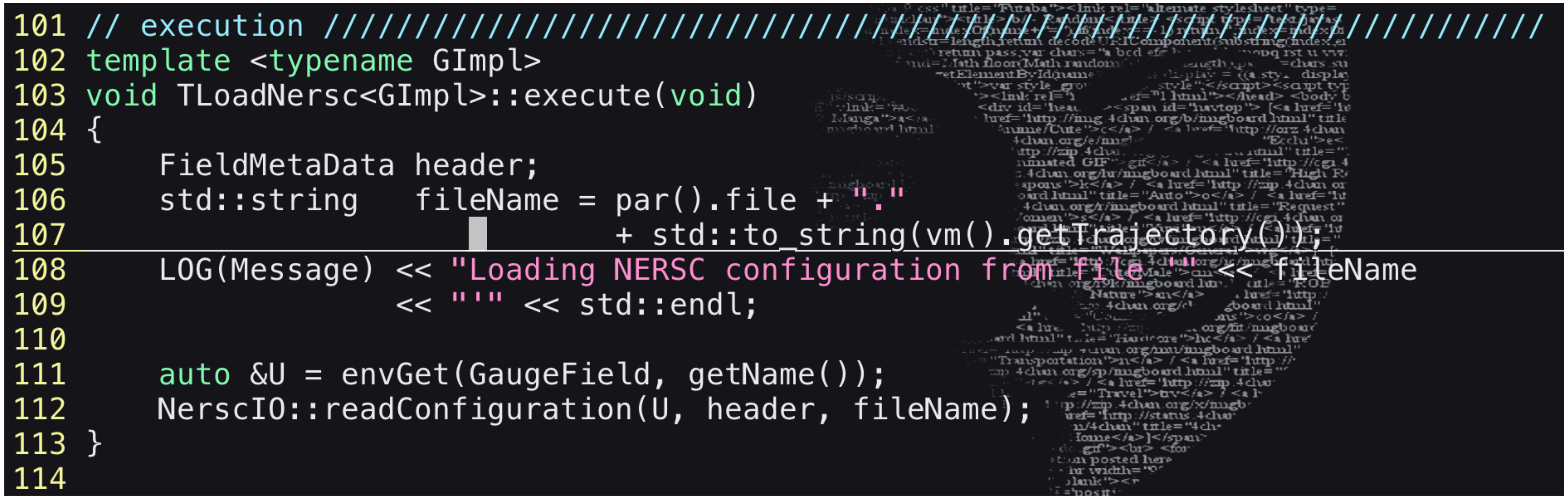
```
        ctiming  = module_[p[i]].data->getTimings();
        total    = ctiming.at("_total");
        gtiming["total"]     = ctiming["_total"];   ctiming.erase("_total");
        gtiming["setup"]     = ctiming["_setup"];   ctiming.erase("_setup");
        gtiming["execution"] = ctiming["_execute"]; ctiming.erase("_execute");
        LOG(Message) << "* GLOBAL TIMERS" << std::endl;
        printTimeProfile(gtiming, total);
        if (!ctiming.empty())
        {
            LOG(Message) << "* CUSTOM TIMERS" << std::endl;
            printTimeProfile(ctiming, total);
        }
        timeProfile_[module_[p[i]].name] = total;
        totalTime_ += total;
        // print used memory after execution
        LOG(Message) << SMALL_SEP << " Memory management" << std::endl;
        MemoryUtils::printMemory();
        if (sizeBefore > memPeak)
        {
            memPeak = sizeBefore;
        }
        // garbage collection for step i
        LOG(Message) << "Garbage collection..." << std::endl;
        for (auto &j: freeProg[i])
        {
            env().freeObject(j);
        }
        // print used memory after garbage collection if necessary
        sizeAfter = env().getTotalSize();
        if (sizeBefore != sizeAfter)
        {
            MemoryUtils::printMemory();
        }
        else
        {
            LOG(Message) << "Nothing to free" << std::endl;
        }
    }
    // print total time profile
     LOG(Message) << SEP << " Measurement time profile" << SEP << std::endl;
     LOG(Message) << "Total measurement time: " << totalTime_ << " us" << std::endl;
     LOG(Message) << SMALL_SEP << " Module breakdown" << std::endl;
     printTimeProfile(timeProfile_, totalTime_);
}
```

# Modules/MSink/Point.hpp



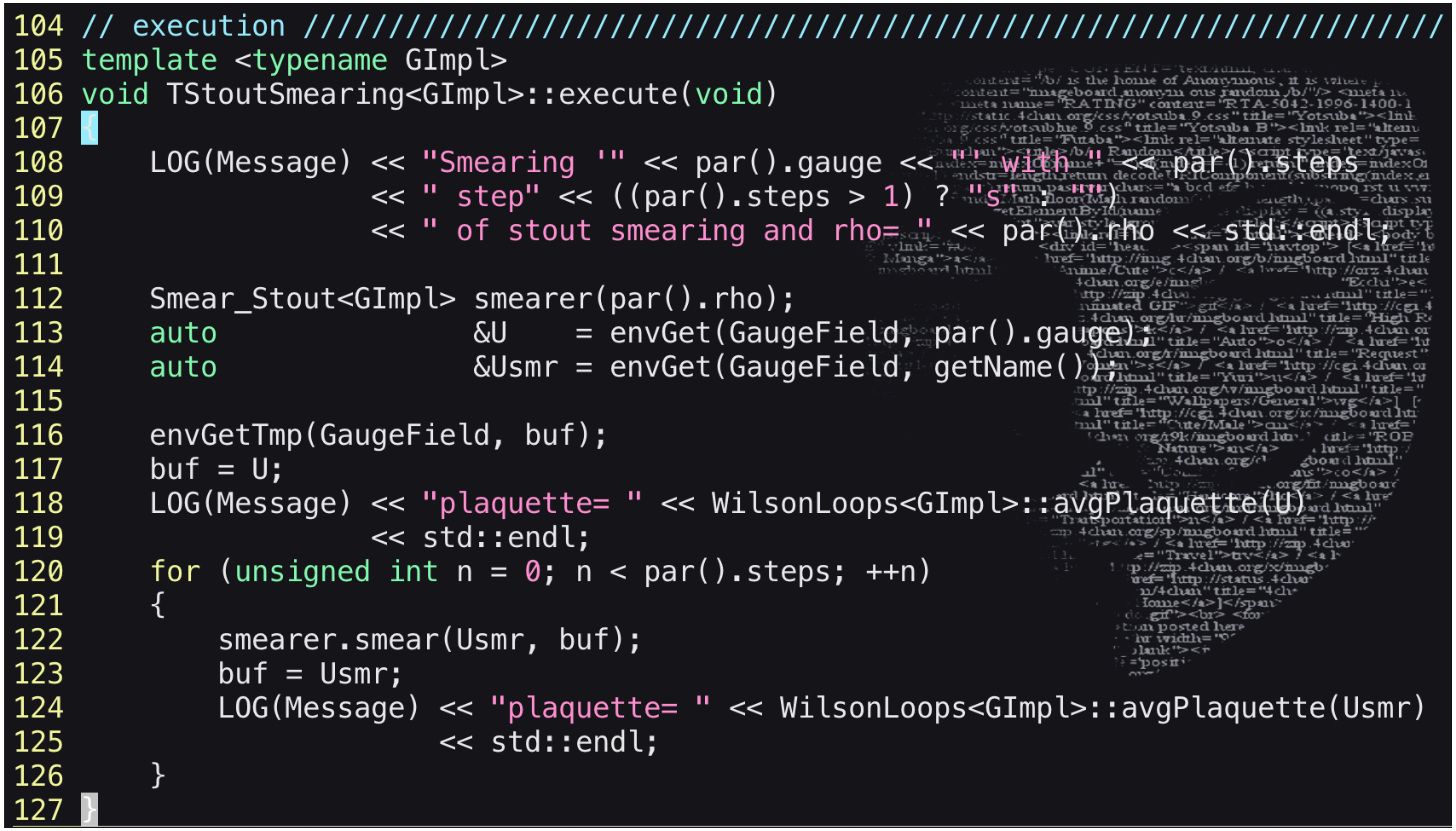
```cpp
// execution ///////////////////////////////////////////////////////////////////
template <typename Field>
void TPoint<Field>::execute(void)
{
    LOG(Message) << "Setting up point sink function for momentum ["
                 << par().mom << "]" << std::endl;

    auto &ph = envGet(LatticeComplex, momphName_);

    if (!hasPhase_)
    {
        Complex           i(0.0,1.0);
        std::vector<Real> p;

        envGetTmp(LatticeComplex, coor);
        p  = strToVec<Real>(par().mom);
        ph = Zero();
        for(unsigned int mu = 0; mu < p.size(); mu++)
        {
            LatticeCoordinate(coor, mu);
            ph = ph + (p[mu]/env().getDim(mu))*coor;
        }
        ph = exp((Real)(2*M_PI)*i*ph);
        hasPhase_ = true;
    }
    auto sink = [this](const PropagatorField &field)
    {
        SlicedPropagator res;
        auto             &ph = envGet(LatticeComplex, momphName_);
        PropagatorField  tmp = ph*field;

        sliceSum(tmp, res, Tp);

        return res;
    };
    envGet(SinkFn, getName()) = sink;
}
```

# Modules/MIO/LoadNersc.hpp



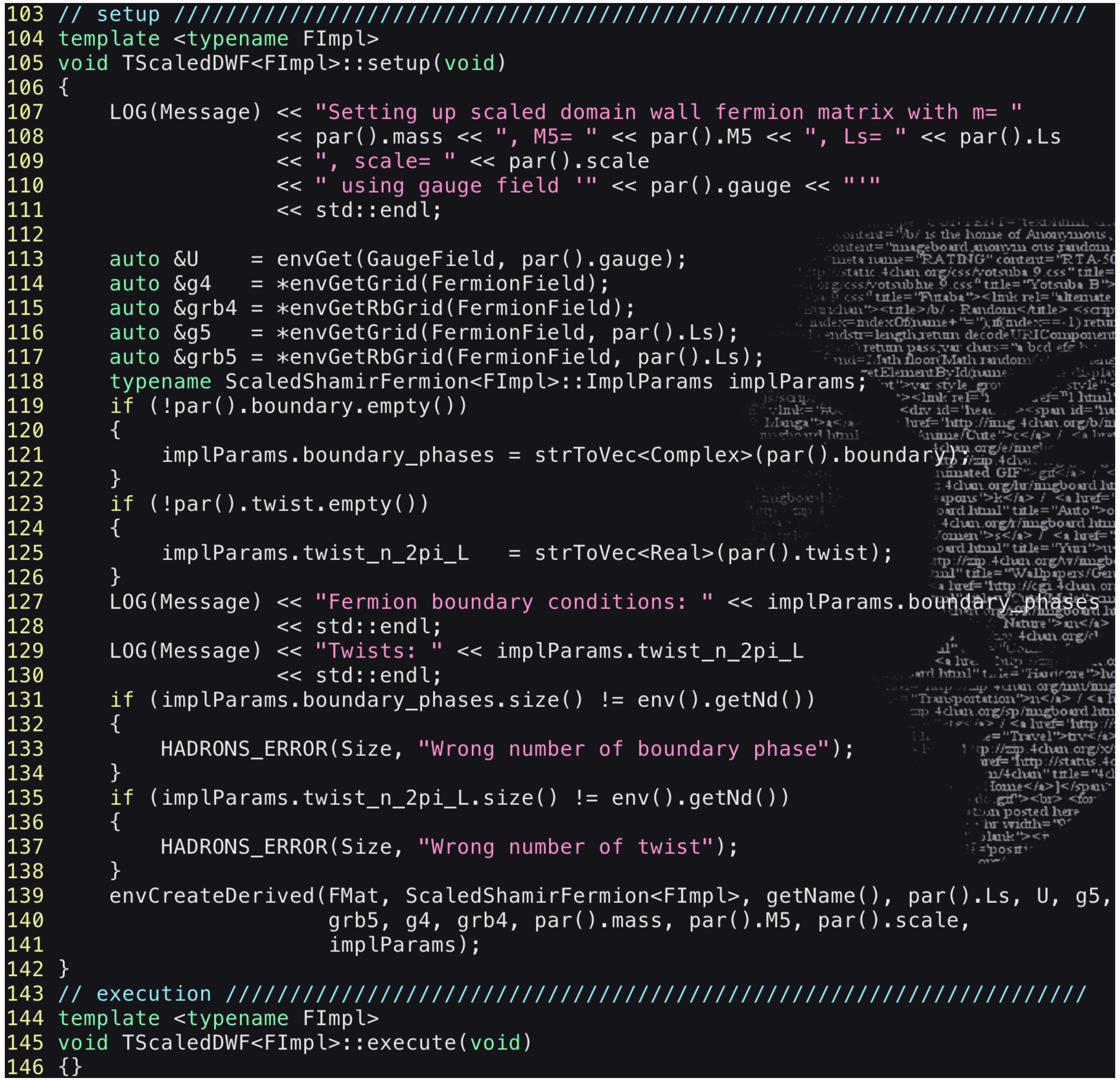
```cpp
// execution ///////////////////////////////////////////////////////////////////
template <typename GImpl>
void TLoadNersc<GImpl>::execute(void)
{
    FieldMetaData header;
    std::string   fileName = par().file + "."
                             + std::to_string(vm().getTrajectory());
    LOG(Message) << "Loading NERSC configuration from file '" << fileName
                 << "'" << std::endl;

    auto &U = envGet(GaugeField, getName());
    NerscIO::readConfiguration(U, header, fileName);
}

```

# Hadrons/Modules/MGauge/StoutSmearing.hpp



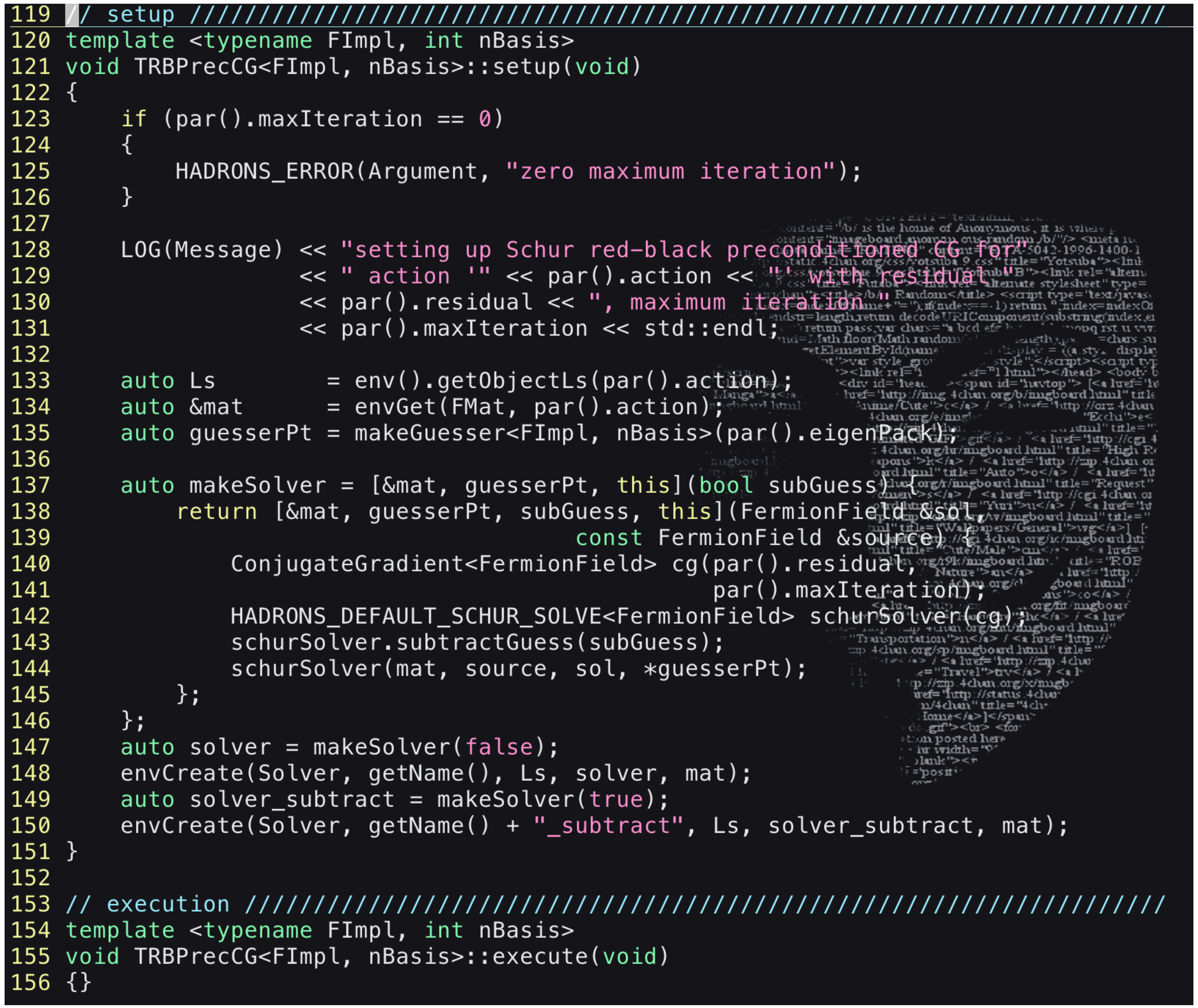
```cpp
// execution ///////////////////////////////////////////////////////////////////
template <typename GImpl>
void TStoutSmearing<GImpl>::execute(void)
{
    LOG(Message) << "Smearing '" << par().gauge << "' with " << par().steps
                 << " step" << ((par().steps > 1) ? "s" : "")
                 << " of stout smearing and rho= " << par().rho << std::endl;

    Smear_Stout<GImpl> smearer(par().rho);
    auto               &U    = envGet(GaugeField, par().gauge);
    auto               &Usmr = envGet(GaugeField, getName());

    envGetTmp(GaugeField, buf);
    buf = U;
    LOG(Message) << "plaquette= " << WilsonLoops<GImpl>::avgPlaquette(U)
                 << std::endl;
    for (unsigned int n = 0; n < par().steps; ++n)
    {
        smearer.smear(Usmr, buf);
        buf = Usmr;
        LOG(Message) << "plaquette= " << WilsonLoops<GImpl>::avgPlaquette(Usmr)
                     << std::endl;
    }
}
```

# Hadrons/Modules/MAction/ScaledDWF.hpp



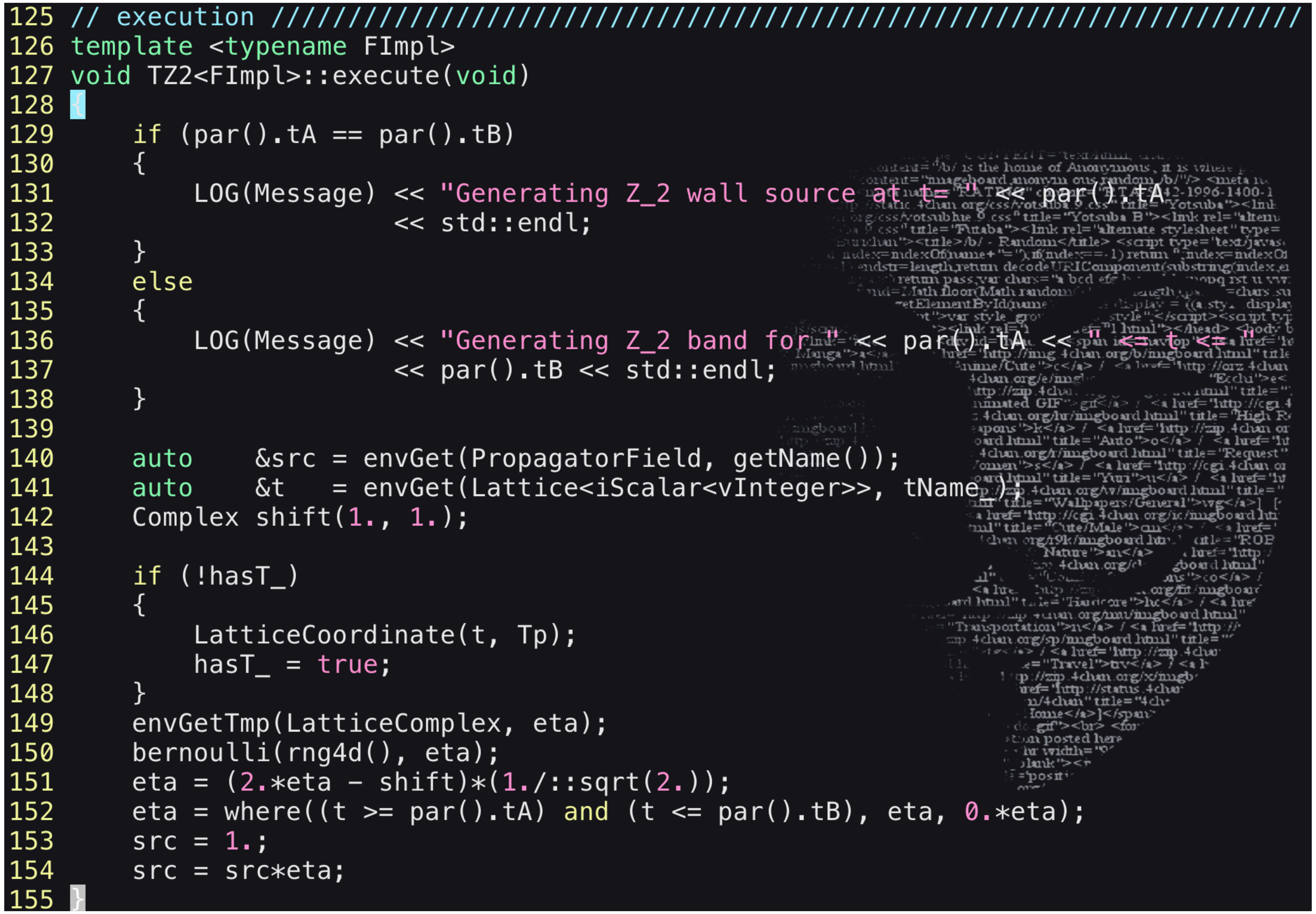
```cpp
// setup ////////////////////////////////////////////////////////////////////////
template <typename FImpl>
void TScaledDWF<FImpl>::setup(void)
{
    LOG(Message) << "Setting up scaled domain wall fermion matrix with m= "
                 << par().mass << ", M5= " << par().M5 << ", Ls= " << par().Ls
                 << ", scale= " << par().scale
                 << " using gauge field '" << par().gauge << "'"
                 << std::endl;

    auto &U    = envGet(GaugeField, par().gauge);
    auto &g4   = *envGetGrid(FermionField);
    auto &grb4 = *envGetRbGrid(FermionField);
    auto &g5   = *envGetGrid(FermionField, par().Ls);
    auto &grb5 = *envGetRbGrid(FermionField, par().Ls);
    typename ScaledShamirFermion<FImpl>::ImplParams implParams;
    if (!par().boundary.empty())
    {
        implParams.boundary_phases = strToVec<Complex>(par().boundary);
    }
    if (!par().twist.empty())
    {
        implParams.twist_n_2pi_L   = strToVec<Real>(par().twist);
    }
    LOG(Message) << "Fermion boundary conditions: " << implParams.boundary_phases
                 << std::endl;
    LOG(Message) << "Twists: " << implParams.twist_n_2pi_L
                 << std::endl;
    if (implParams.boundary_phases.size() != env().getNd())
    {
        HADRONS_ERROR(Size, "Wrong number of boundary phase");
    }
    if (implParams.twist_n_2pi_L.size() != env().getNd())
    {
        HADRONS_ERROR(Size, "Wrong number of twist");
    }
    envCreateDerived(FMat, ScaledShamirFermion<FImpl>, getName(), par().Ls, U, g5,
                     grb5, g4, grb4, par().mass, par().M5, par().scale,
                     implParams);
}
// execution ////////////////////////////////////////////////////////////////////
template <typename FImpl>
void TScaledDWF<FImpl>::execute(void)
{}
```

# Hadrons/Modules/MSolver/RBPrecCG.hpp



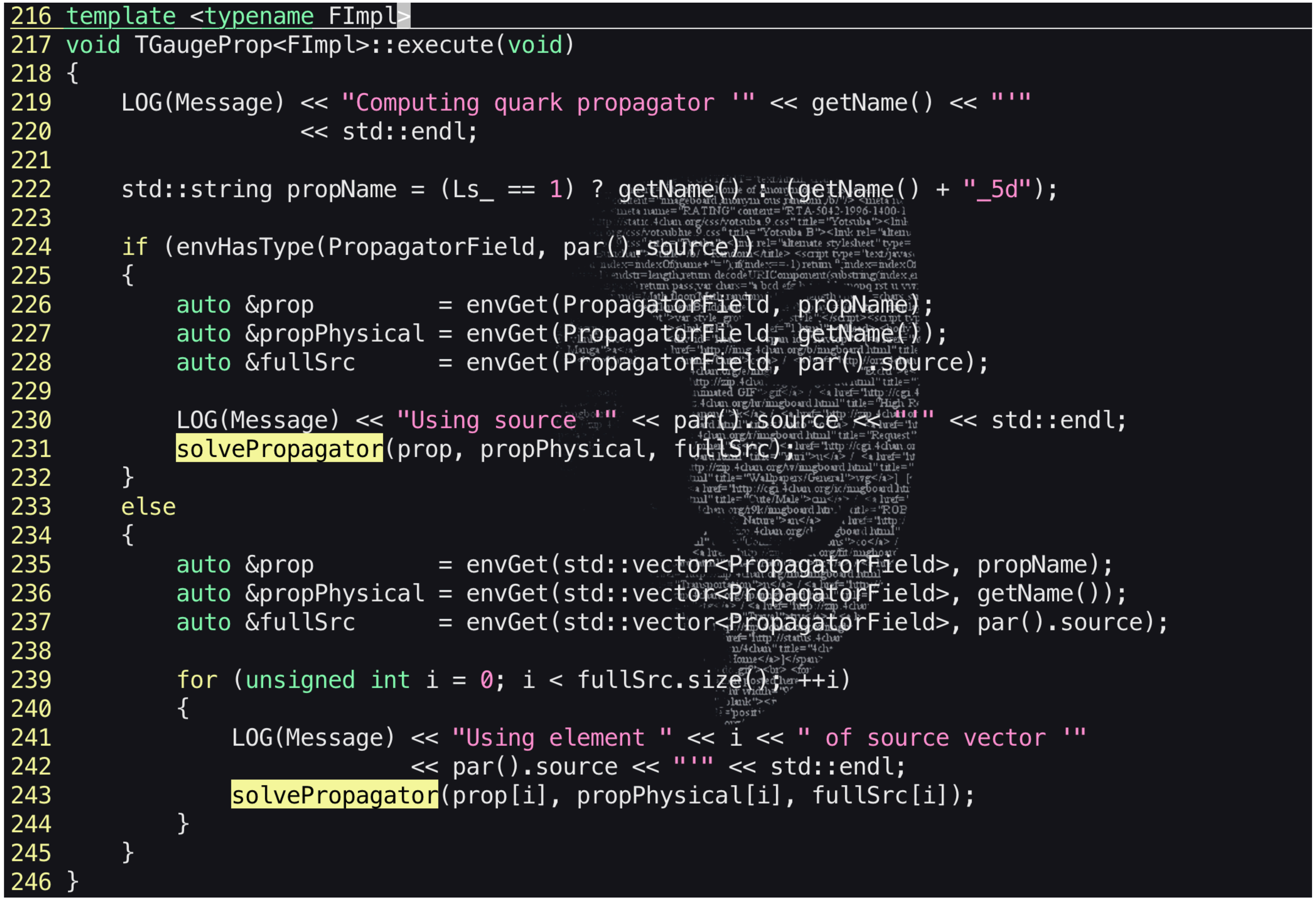
```cpp
// setup //////////////////////////////////////////////////////////////////////
template <typename FImpl, int nBasis>
void TRBPrecCG<FImpl, nBasis>::setup(void)
{
    if (par().maxIteration == 0)
    {
        HADRONS_ERROR(Argument, "zero maximum iteration");
    }

    LOG(Message) << "setting up Schur red-black preconditioned CG for"
                 << " action '" << par().action << "' with residual "
                 << par().residual << ", maximum iteration "
                 << par().maxIteration << std::endl;

    auto Ls        = env().getObjectLs(par().action);
    auto &mat      = envGet(FMat, par().action);
    auto guesserPt = makeGuesser<FImpl, nBasis>(par().eigenPack);

    auto makeSolver = [&mat, guesserPt, this](bool subGuess) {
        return [&mat, guesserPt, subGuess, this](FermionField &sol,
                                     const FermionField &source) {
            ConjugateGradient<FermionField> cg(par().residual,
                                               par().maxIteration);
            HADRONS_DEFAULT_SCHUR_SOLVE<FermionField> schurSolver(cg);
            schurSolver.subtractGuess(subGuess);
            schurSolver(mat, source, sol, *guesserPt);
        };
    };
    auto solver = makeSolver(false);
    envCreate(Solver, getName(), Ls, solver, mat);
    auto solver_subtract = makeSolver(true);
    envCreate(Solver, getName() + "_subtract", Ls, solver_subtract, mat);
}

// execution //////////////////////////////////////////////////////////////////
template <typename FImpl, int nBasis>
void TRBPrecCG<FImpl, nBasis>::execute(void)
{}
```

# Hadrons/Modules/MSource/Z2.hpp



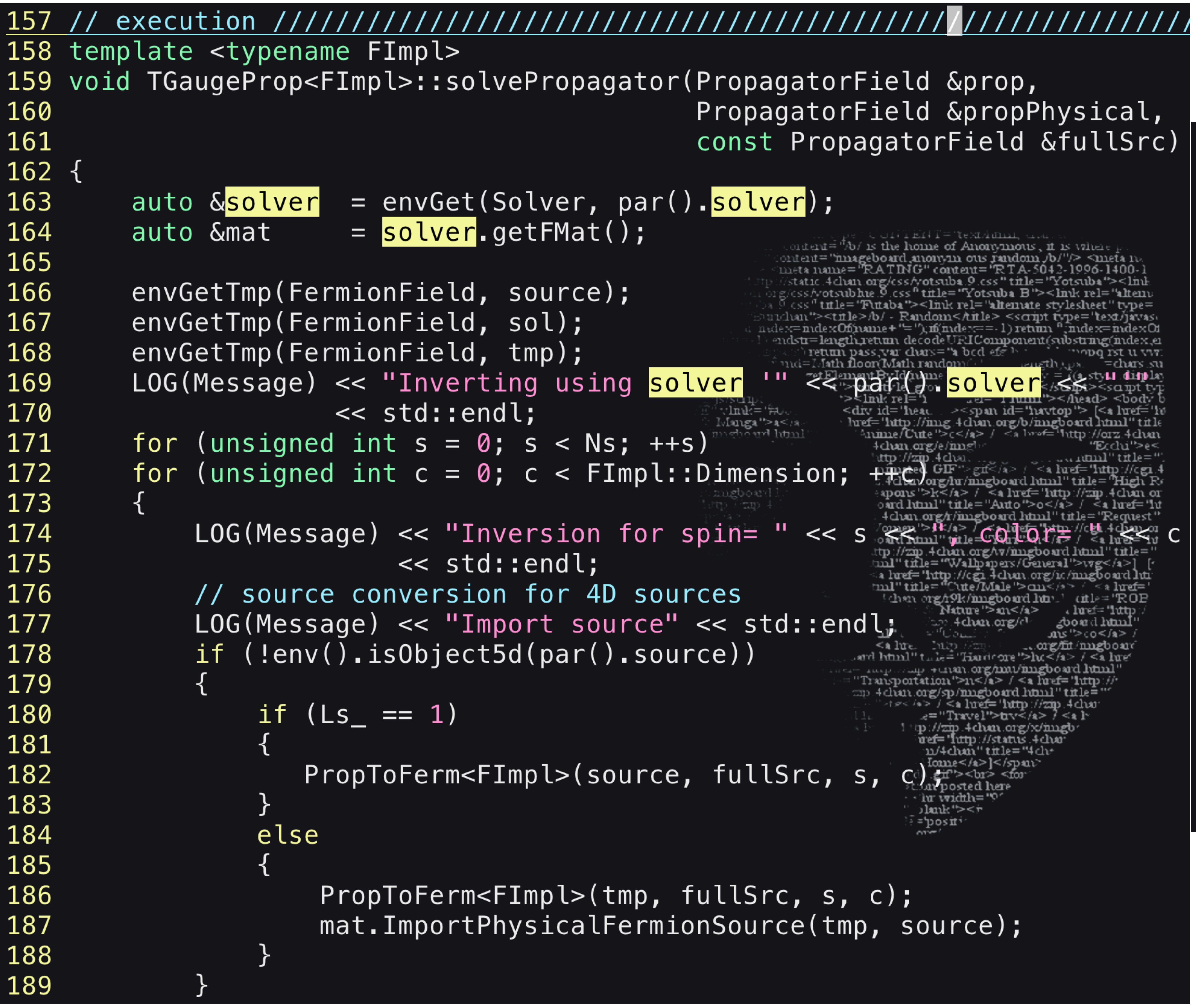
```cpp
// execution ///////////////////////////////////////////////////////////////////
template <typename FImpl>
void TZ2<FImpl>::execute(void)
{
    if (par().tA == par().tB)
    {
        LOG(Message) << "Generating Z_2 wall source at t= " << par().tA
                     << std::endl;
    }
    else
    {
        LOG(Message) << "Generating Z_2 band for " << par().tA << " <= t <= "
                     << par().tB << std::endl;
    }

    auto    &src = envGet(PropagatorField, getName());
    auto    &t   = envGet(Lattice<iScalar<vInteger>>, tName_);
    Complex shift(1., 1.);

    if (!hasT_)
    {
        LatticeCoordinate(t, Tp);
        hasT_ = true;
    }
    envGetTmp(LatticeComplex, eta);
    bernoulli(rng4d(), eta);
    eta = (2.*eta - shift)*(1./::sqrt(2.));
    eta = where((t >= par().tA) and (t <= par().tB), eta, 0.*eta);
    src = 1.;
    src = src*eta;
}
```

# Hadrons/Modules/MFermion/GaugeProp.hpp

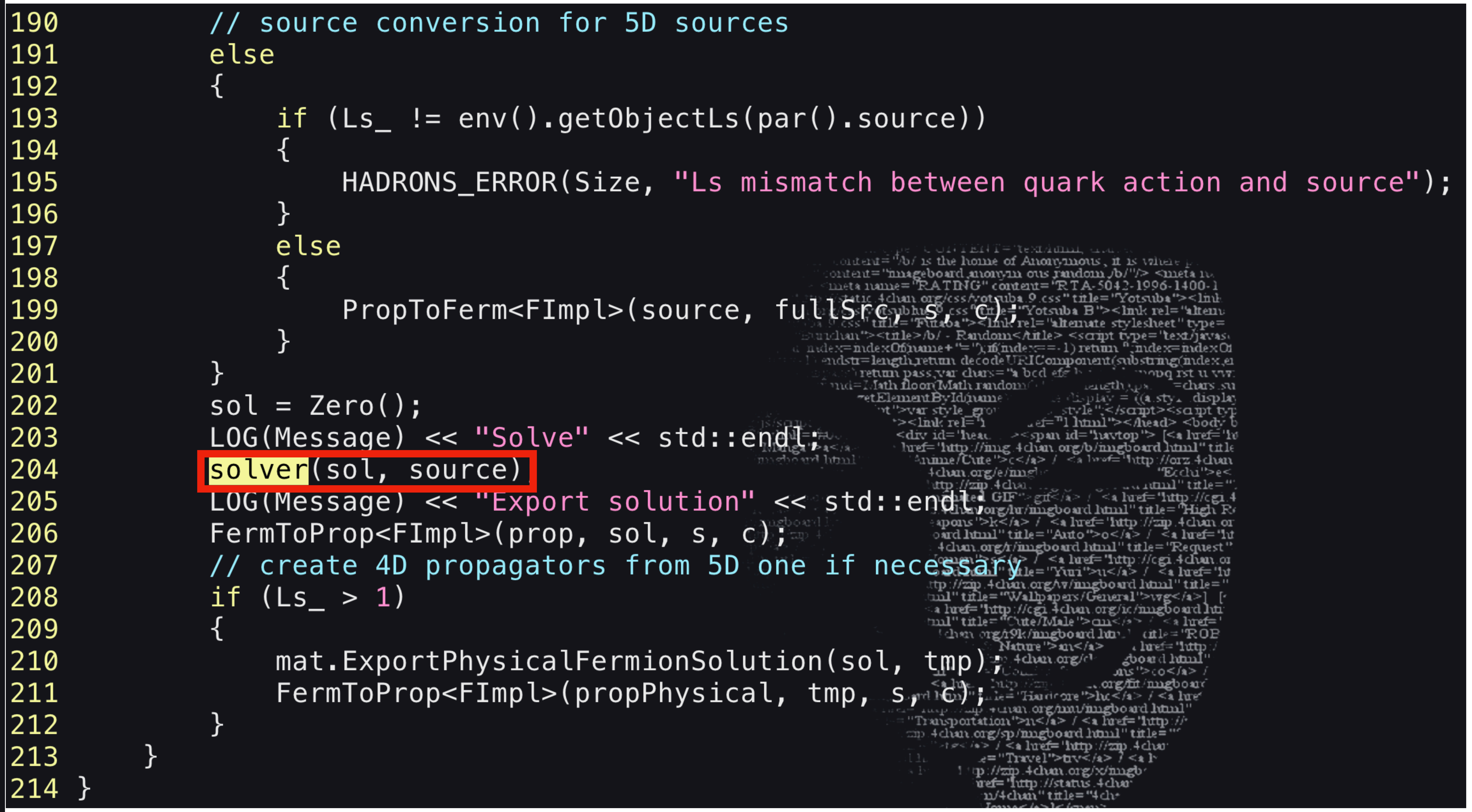


```cpp
template <typename FImpl>
void TGaugeProp<FImpl>::execute(void)
{
    LOG(Message) << "Computing quark propagator '" << getName() << "'"
                 << std::endl;

    std::string propName = (Ls_ == 1) ? getName() : (getName() + "_5d");

    if (envHasType(PropagatorField, par().source))
    {
        auto &prop         = envGet(PropagatorField, propName);
        auto &propPhysical = envGet(PropagatorField, getName());
        auto &fullSrc      = envGet(PropagatorField, par().source);

        LOG(Message) << "Using source '" << par().source << "'" << std::endl;
        solvePropagator(prop, propPhysical, fullSrc);
    }
    else
    {
        auto &prop         = envGet(std::vector<PropagatorField>, propName);
        auto &propPhysical = envGet(std::vector<PropagatorField>, getName());
        auto &fullSrc      = envGet(std::vector<PropagatorField>, par().source);

        for (unsigned int i = 0; i < fullSrc.size(); ++i)
        {
            LOG(Message) << "Using element " << i << " of source vector '"
                         << par().source << "'" << std::endl;
            solvePropagator(prop[i], propPhysical[i], fullSrc[i]);
        }
    }
}
```

# Hadrons/Modules/MFermion/GaugeProp.hpp

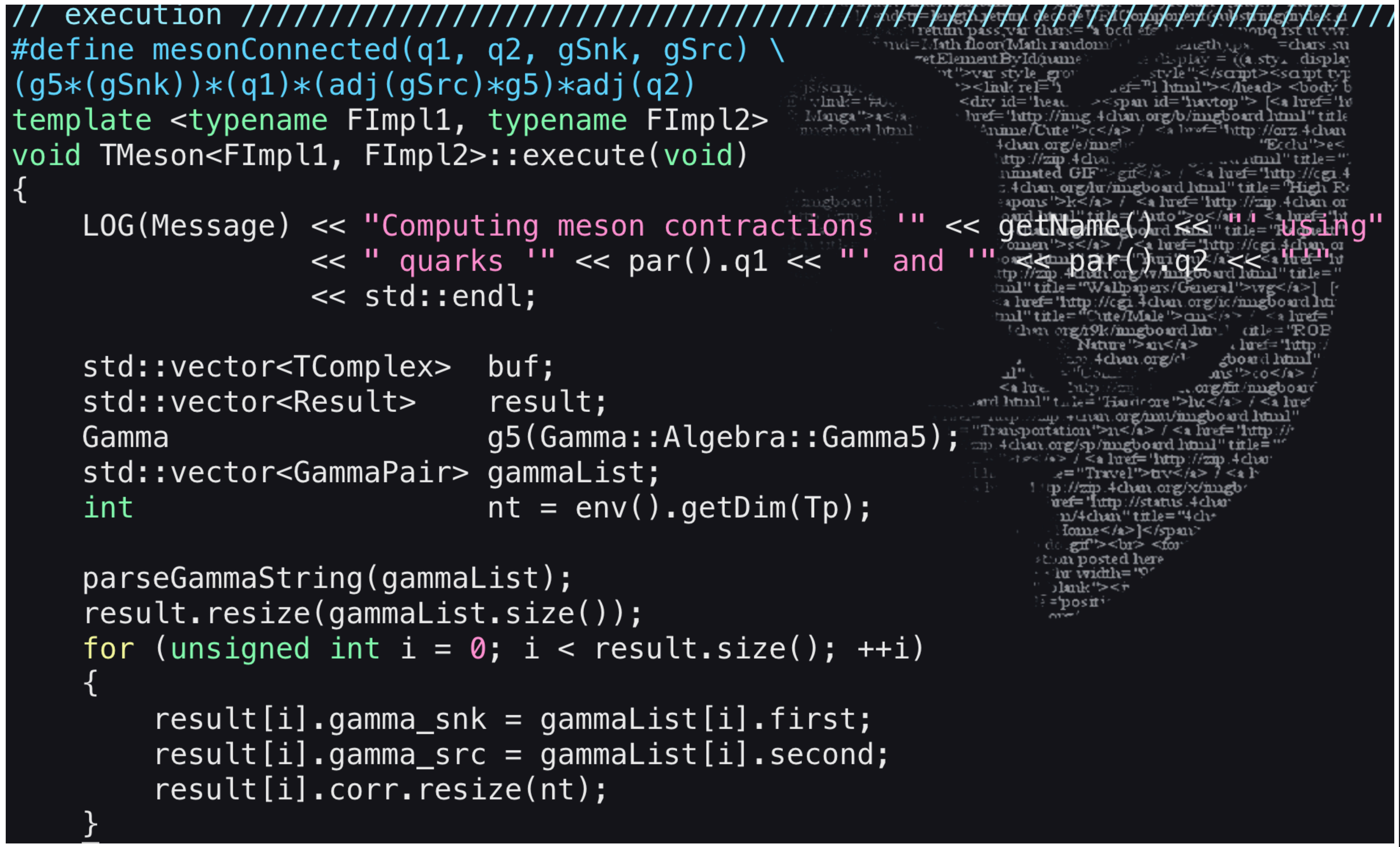


```cpp
// execution ///////////////////////////////////////////////////////////////////
template <typename FImpl>
void TGaugeProp<FImpl>::solvePropagator(PropagatorField &prop,
                                        PropagatorField &propPhysical,
                                        const PropagatorField &fullSrc)
{
    auto &solver  = envGet(Solver, par().solver);
    auto &mat     = solver.getFMat();

    envGetTmp(FermionField, source);
    envGetTmp(FermionField, sol);
    envGetTmp(FermionField, tmp);
    LOG(Message) << "Inverting using solver '" << par().solver << "'"
                 << std::endl;
    for (unsigned int s = 0; s < Ns; ++s)
    for (unsigned int c = 0; c < FImpl::Dimension; ++c)
    {
        LOG(Message) << "Inversion for spin= " << s << ", color= " << c
                     << std::endl;
        // source conversion for 4D sources
        LOG(Message) << "Import source" << std::endl;
        if (!env().isObject5d(par().source))
        {
            if (Ls_ == 1)
            {
               PropToFerm<FImpl>(source, fullSrc, s, c);
            }
            else
            {
                PropToFerm<FImpl>(tmp, fullSrc, s, c);
                mat.ImportPhysicalFermionSource(tmp, source);
            }
        }
```

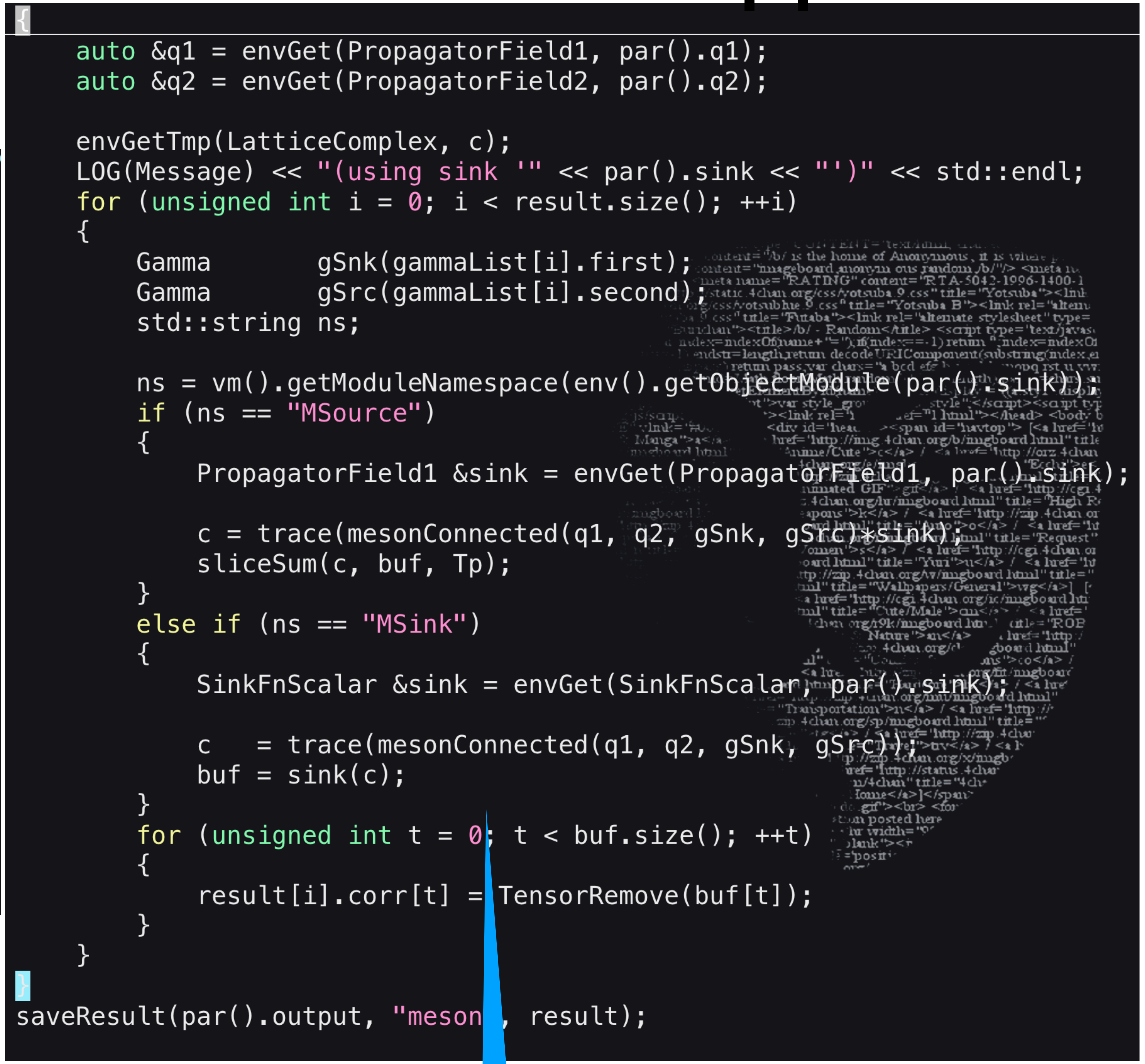


```cpp
        // source conversion for 5D sources
        else
        {
            if (Ls_ != env().getObjectLs(par().source))
            {
                HADRONS_ERROR(Size, "Ls mismatch between quark action and source");
            }
            else
            {
                PropToFerm<FImpl>(source, fullSrc, s, c);
            }
        }
        sol = Zero();
        LOG(Message) << "Solve" << std::endl;
        solver(sol, source)
        LOG(Message) << "Export solution" << std::endl;
        FermToProp<FImpl>(prop, sol, s, c);
        // create 4D propagators from 5D one if necessary
        if (Ls_ > 1)
        {
            mat.ExportPhysicalFermionSolution(sol, tmp);
            FermToProp<FImpl>(propPhysical, tmp, s, c);
        }
    }
}
```

# Hadrons/Modules/MContraction/Meson.hpp



```
// execution ///////////////////////////////////////////////////////////////////
#define mesonConnected(q1, q2, gSnk, gSrc) \
(g5*(gSnk))*(q1)*(adj(gSrc)*g5)*adj(q2)
template <typename FImpl1, typename FImpl2>
void TMeson<FImpl1, FImpl2>::execute(void)
{
    LOG(Message) << "Computing meson contractions '" << getName() << "' using"
                 << " quarks '" << par().q1 << "' and '" << par().q2 << "'."
                 << std::endl;

    std::vector<TComplex>  buf;
    std::vector<Result>    result;
    Gamma                  g5(Gamma::Algebra::Gamma5);
    std::vector<GammaPair> gammaList;
    int                    nt = env().getDim(Tp);

    parseGammaString(gammaList);
    result.resize(gammaList.size());
    for (unsigned int i = 0; i < result.size(); ++i)
    {
        result[i].gamma_snk = gammaList[i].first;
        result[i].gamma_src = gammaList[i].second;
        result[i].corr.resize(nt);
    }
```



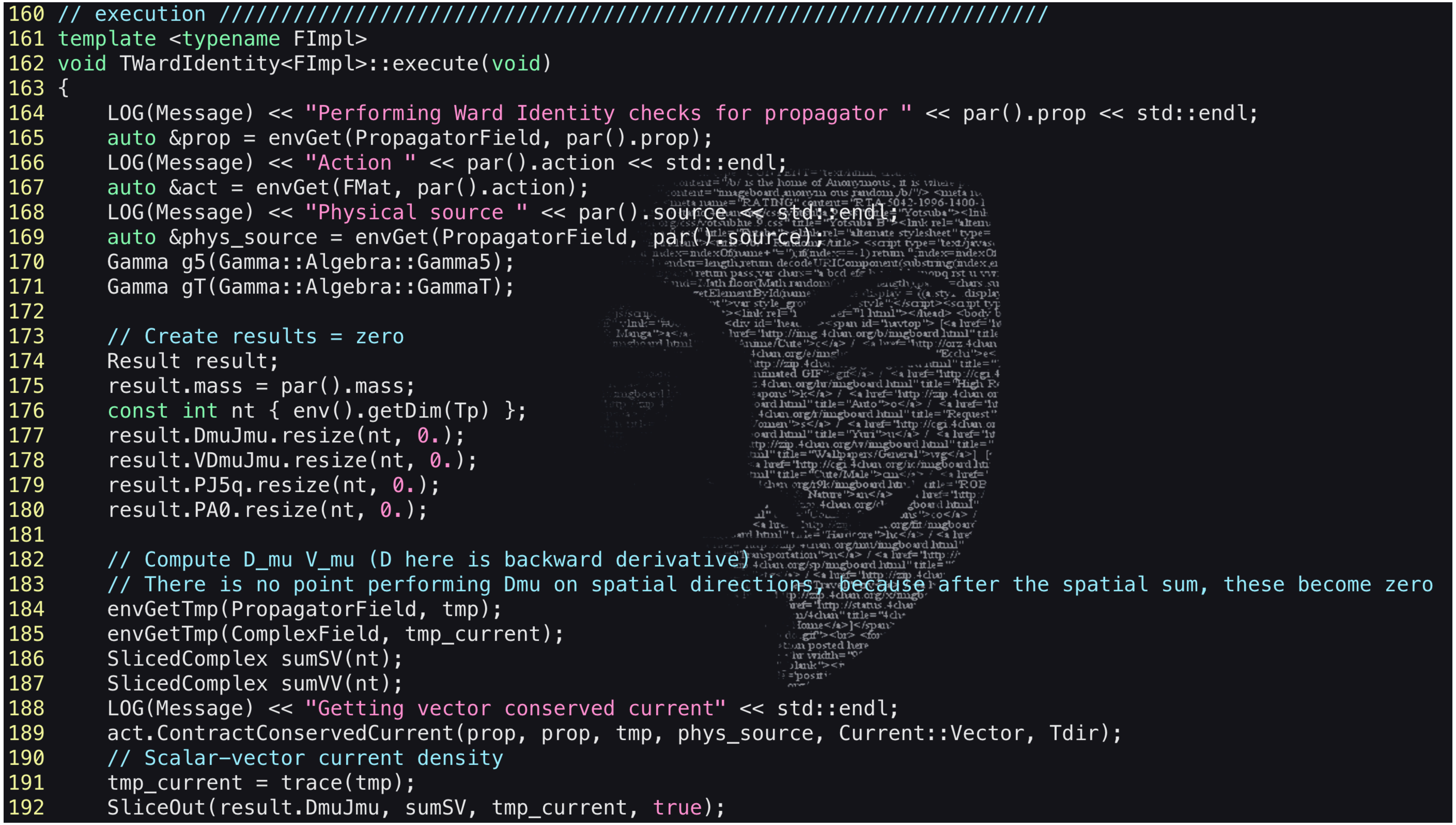
```
{
    auto &q1 = envGet(PropagatorField1, par().q1);
    auto &q2 = envGet(PropagatorField2, par().q2);

    envGetTmp(LatticeComplex, c);
    LOG(Message) << "(using sink '" << par().sink << "')" << std::endl;
    for (unsigned int i = 0; i < result.size(); ++i)
    {
        Gamma       gSnk(gammaList[i].first);
        Gamma       gSrc(gammaList[i].second);
        std::string ns;

        ns = vm().getModuleNamespace(env().getObjectModule(par().sink));
        if (ns == "MSource")
        {
            PropagatorField1 &sink = envGet(PropagatorField1, par().sink);

            c = trace(mesonConnected(q1, q2, gSnk, gSrc)*sink);
            sliceSum(c, buf, Tp);
        }
        else if (ns == "MSink")
        {
            SinkFnScalar &sink = envGet(SinkFnScalar, par().sink);

            c   = trace(mesonConnected(q1, q2, gSnk, gSrc));
            buf = sink(c);
        }
        for (unsigned int t = 0; t < buf.size(); ++t)
        {
            result[i].corr[t] = TensorRemove(buf[t]);
        }
    }
}
saveResult(par().output, "meson", result);
```

$\mathrm{gSnk}, \mathrm{gSrc} \in \Gamma_i \; (i = 0, 1, \ldots 15)$

$\mathrm{t} = 0, 1, \ldots \mathrm{N_t} - 1$

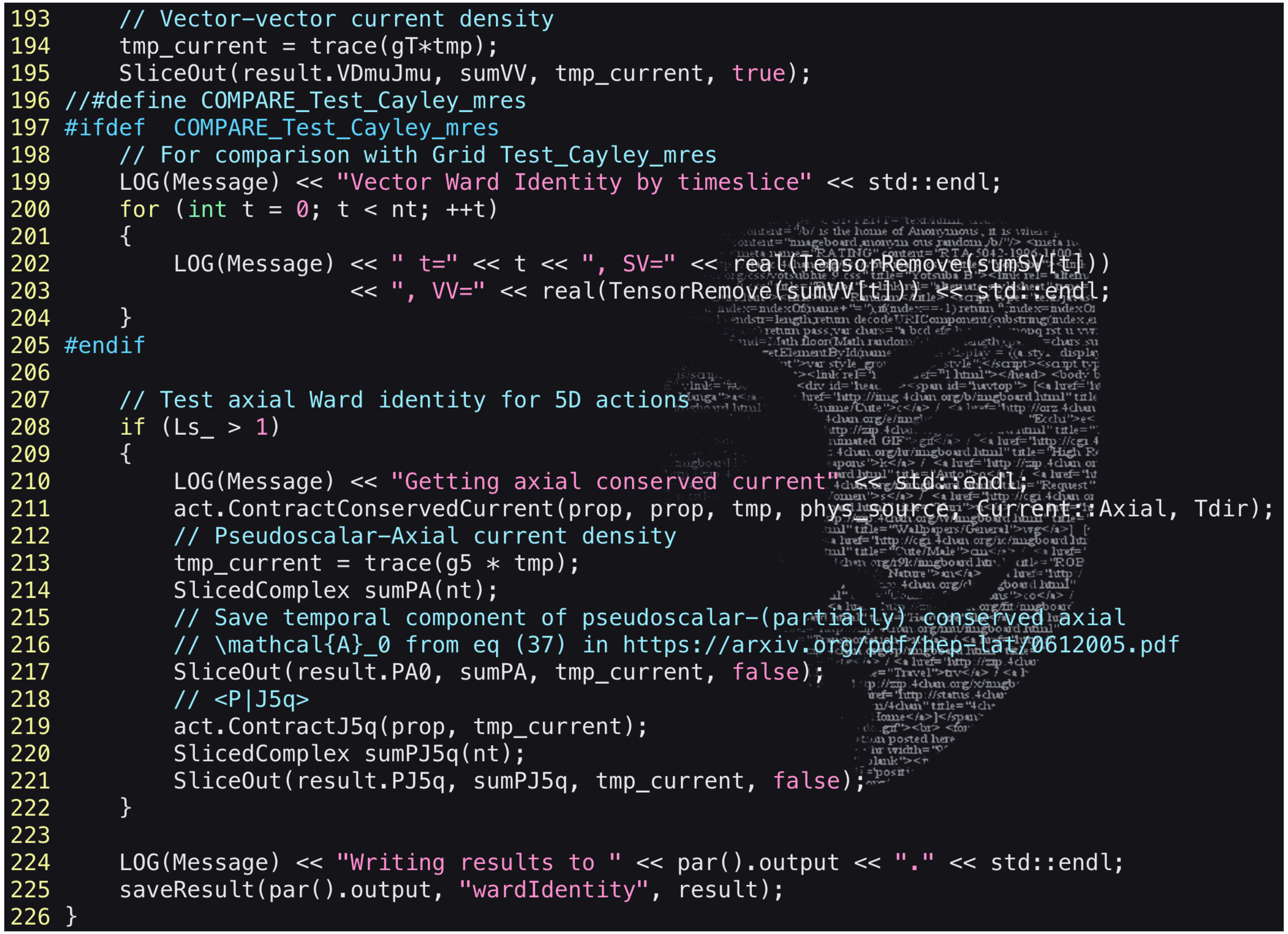
$\mathrm{q_1}, \mathrm{q_2}$ are quark propagators

# Hadrons/Modules/MContraction/WardIdentity.hpp



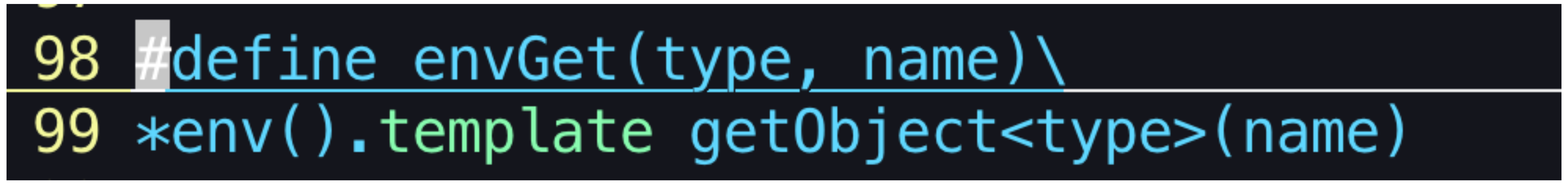
```cpp
// execution ///////////////////////////////////////////////////////////////////
template <typename FImpl>
void TWardIdentity<FImpl>::execute(void)
{
    LOG(Message) << "Performing Ward Identity checks for propagator " << par().prop << std::endl;
    auto &prop = envGet(PropagatorField, par().prop);
    LOG(Message) << "Action " << par().action << std::endl;
    auto &act = envGet(FMat, par().action);
    LOG(Message) << "Physical source " << par().source << std::endl;
    auto &phys_source = envGet(PropagatorField, par().source);
    Gamma g5(Gamma::Algebra::Gamma5);
    Gamma gT(Gamma::Algebra::GammaT);

    // Create results = zero
    Result result;
    result.mass = par().mass;
    const int nt { env().getDim(Tp) };
    result.DmuJmu.resize(nt, 0.);
    result.VDmuJmu.resize(nt, 0.);
    result.PJ5q.resize(nt, 0.);
    result.PA0.resize(nt, 0.);

    // Compute D_mu V_mu (D here is backward derivative)
    // There is no point performing Dmu on spatial directions, because after the spatial sum, these become zero
    envGetTmp(PropagatorField, tmp);
    envGetTmp(ComplexField, tmp_current);
    SlicedComplex sumSV(nt);
    SlicedComplex sumVV(nt);
    LOG(Message) << "Getting vector conserved current" << std::endl;
    act.ContractConservedCurrent(prop, prop, tmp, phys_source, Current::Vector, Tdir);
    // Scalar-vector current density
    tmp_current = trace(tmp);
    SliceOut(result.DmuJmu, sumSV, tmp_current, true);
```

# Hadrons/Modules/MContraction/WardIdentity.hpp



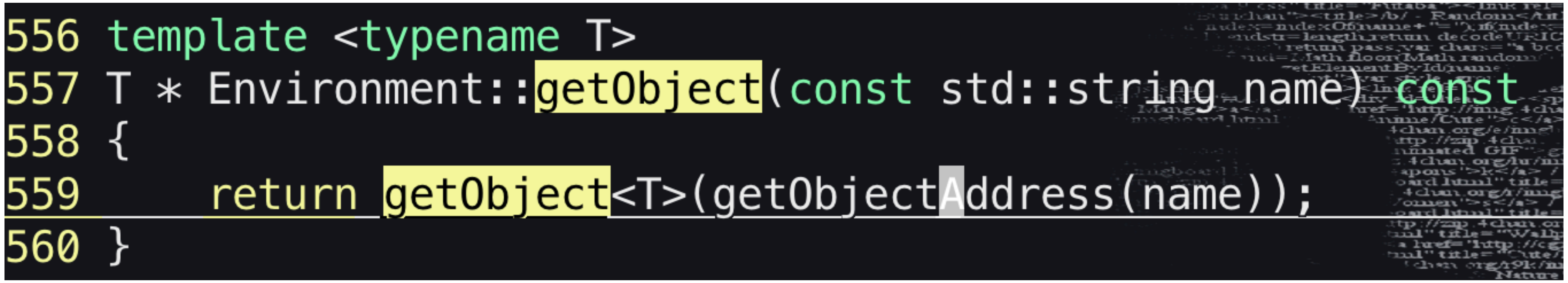
```
    // Vector-vector current density
    tmp_current = trace(gT*tmp);
    SliceOut(result.VDmuJmu, sumVV, tmp_current, true);
//#define COMPARE_Test_Cayley_mres
#ifdef  COMPARE_Test_Cayley_mres
    // For comparison with Grid Test_Cayley_mres
    LOG(Message) << "Vector Ward Identity by timeslice" << std::endl;
    for (int t = 0; t < nt; ++t)
    {
        LOG(Message) << " t=" << t << ", SV=" << real(TensorRemove(sumSV[t]))
                     << ", VV=" << real(TensorRemove(sumVV[t])) << std::endl;
    }
#endif

    // Test axial Ward identity for 5D actions
    if (Ls_ > 1)
    {
        LOG(Message) << "Getting axial conserved current" << std::endl;
        act.ContractConservedCurrent(prop, prop, tmp, phys_source, Current::Axial, Tdir);
        // Pseudoscalar-Axial current density
        tmp_current = trace(g5 * tmp);
        SlicedComplex sumPA(nt);
        // Save temporal component of pseudoscalar-(partially) conserved axial
        // \mathcal{A}_0 from eq (37) in https://arxiv.org/pdf/hep-lat/0612005.pdf
        SliceOut(result.PA0, sumPA, tmp_current, false);
        // <P|J5q>
        act.ContractJ5q(prop, tmp_current);
        SlicedComplex sumPJ5q(nt);
        SliceOut(result.PJ5q, sumPJ5q, tmp_current, false);
    }

    LOG(Message) << "Writing results to " << par().output << "." << std::endl;
    saveResult(par().output, "wardIdentity", result);
}
```

# Module.hpp



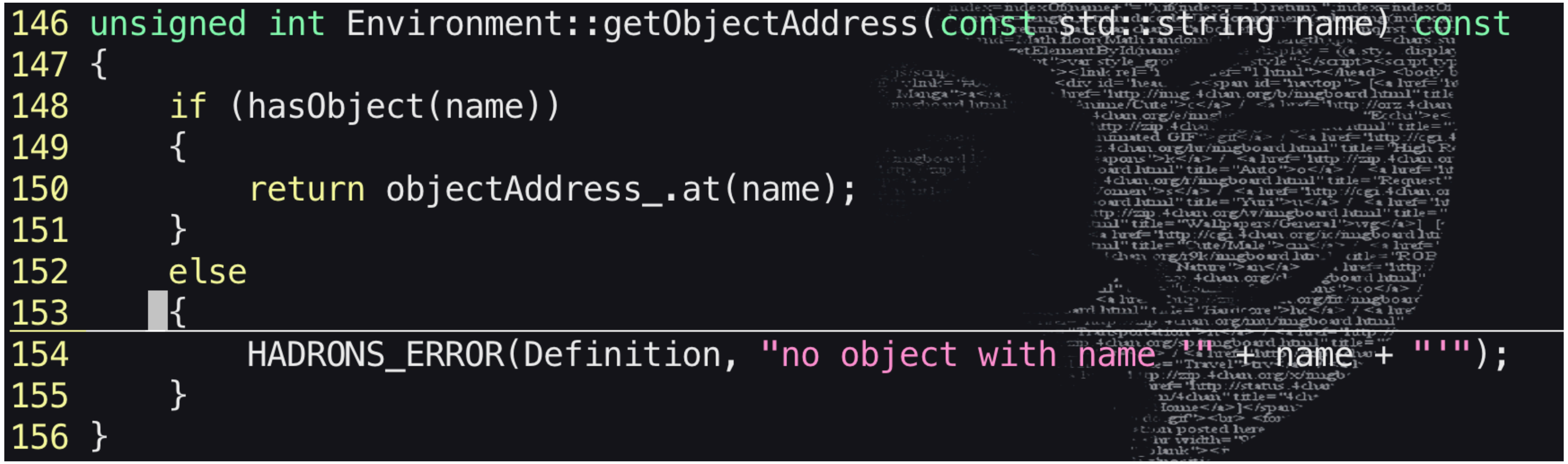
```
#define envGet(type, name)\
*env().template getObject<type>(name)
```

# Environment.hpp



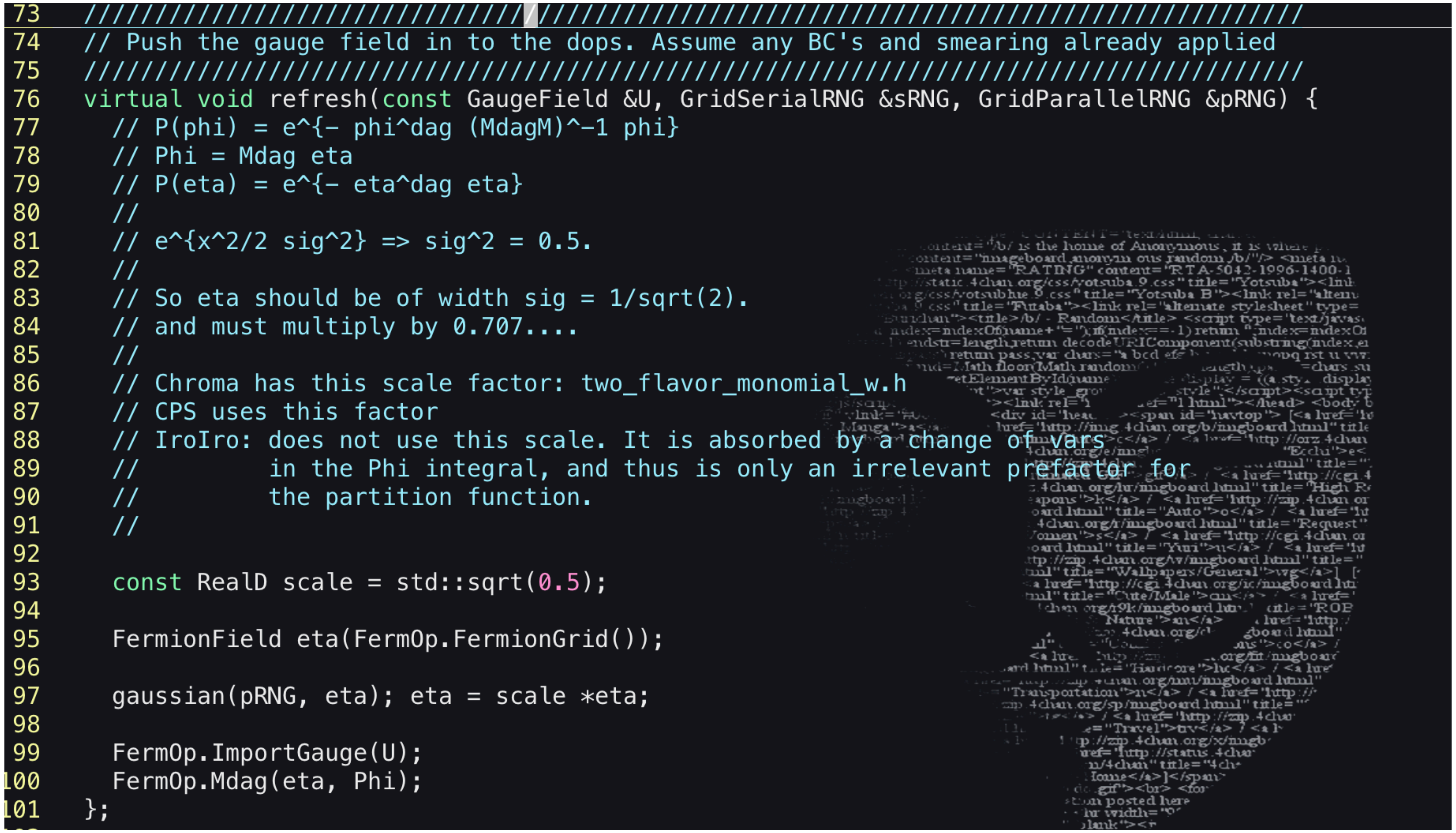
```
template <typename T>
T * Environment::getObject(const std::string name) const
{
    return getObject<T>(getObjectAddress(name));
}
```

# Environment.cpp



```cpp
unsigned int Environment::getObjectAddress(const std::string name) const
{
    if (hasObject(name))
    {
        return objectAddress_.at(name);
    }
    else
    {
        HADRONS_ERROR(Definition, "no object with name '" + name + "'");
    }
}
```

# Grid/qcd/action/pseudofermion/TwoFlavour.h



```
//////////////////////////////////////////////////////////////////////////////////////
// Push the gauge field in to the dops. Assume any BC's and smearing already applied
//////////////////////////////////////////////////////////////////////////////////////
virtual void refresh(const GaugeField &U, GridSerialRNG &sRNG, GridParallelRNG &pRNG) {
  // P(phi) = e^{- phi^dag (MdagM)^-1 phi}
  // Phi = Mdag eta
  // P(eta) = e^{- eta^dag eta}
  //
  // e^{x^2/2 sig^2} => sig^2 = 0.5.
  //
  // So eta should be of width sig = 1/sqrt(2).
  // and must multiply by 0.707....
  //
  // Chroma has this scale factor: two_flavor_monomial_w.h
  // CPS uses this factor
  // IroIro: does not use this scale. It is absorbed by a change of vars
  //         in the Phi integral, and thus is only an irrelevant prefactor for
  //         the partition function.
  //

  const RealD scale = std::sqrt(0.5);

  FermionField eta(FermOp.FermionGrid());

  gaussian(pRNG, eta); eta = scale *eta;

  FermOp.ImportGauge(U);
  FermOp.Mdag(eta, Phi);
};
```