```
#ifndef BOND_H
#define BOND_H

#include<vector>
#include<complex>
#include<iostream>
#include<fstream>
#include<algorithm>
#include<sstream>

#if defined PHONONS && !defined ELASTICONLY
const int NDMAT=NSUBL+NMODE;
#else
const int NDMAT=NSUBL;
#endif
const int NDMAT2=NDMAT*NDMAT;

const int NS=1; // remove this


  //#g++ -I/data/sylju/include -L/data/sylju/lib FFTWtest.C -lfftw3 -lm;
#include <fftw3.h>
#include "matrixroutines.h"

#ifdef LONGDOUBLE
typedef fftwl_plan FFTWPLAN;
typedef fftwl_complex FFTWCOMPLEX;
#define FFTWEXECUTE(x) fftwl_execute(x)
#define FFTWDESTROYPLAN(x) fftwl_destroy_plan(x)
#elif defined FLOAT
typedef fftwf_plan FFTWPLAN;
typedef fftwf_complex FFTWCOMPLEX;
#define FFTWEXECUTE(x) fftwf_execute(x)
#define FFTWDESTROYPLAN(x) fftwf_destroy_plan(x)
#elif defined QUAD
typedef fftwq_plan FFTWPLAN;
typedef fftwq_complex FFTWCOMPLEX;
#define FFTWEXECUTE(x) fftwq_execute(x)
#define FFTWDESTROYPLAN(x) fftwq_destroy_plan(x)
#else
typedef fftw_plan FFTWPLAN;
typedef fftw_complex FFTWCOMPLEX;
#define FFTWEXECUTE(x) fftw_execute(x)
#define FFTWDESTROYPLAN(x) fftw_destroy_plan(x)
#endif



struct NumberList
{
NumberList(int n=NSUBL,realtype a=0.):v(n,a){}
  friend ostream& operator<<(ostream& os,const NumberList& d){for(unsigned int i=0; i<d.v.s
ize(); i++){ os << d.v[i] << " ";} return os;}
  vector<realtype> v;
};

bool operator==(NumberList& l,NumberList& r)
{
  bool isequal=true;
  for(unsigned int i=0; i<l.v.size(); i++){isequal &= (l.v[i]==r.v[i]); if(!isequal){break;
}}
  return isequal;
}



class Driver
{
```

```
 public:
  Driver(Rule&);
  ~Driver()
    {
      FFTWDESTROYPLAN(A1q_to_A1r);
      FFTWDESTROYPLAN(A1r_to_A1q);
      FFTWDESTROYPLAN(A2q_to_A2r);
      FFTWDESTROYPLAN(A2r_to_A2q);
      FFTWDESTROYPLAN(Bq_to_Br);
      FFTWDESTROYPLAN(Br_to_Bq);
      FFTWDESTROYPLAN(F1r_to_F1q);
      FFTWDESTROYPLAN(F1q_to_F1r);
      FFTWDESTROYPLAN(F2r_to_F2q);
      FFTWDESTROYPLAN(F2q_to_F2r);
    }
  realtype CalculateT(int);
  void CalculateTs(vector<realtype>&);

#ifdef PHONONS
  vector<realtype> CalculateEpsilonsOverT();
#endif

  realtype CalculateFreeEnergy(const realtype);
  vector<obstype> CalculateSpinOrderPars(realtype);
  vector<obstype> CalculateOrderPars(realtype,int,int);
  vector<obstype> CalculateAlphas(realtype);

  //void Convolve(const bool);
  //  realtype CalculateSecondDerivative(const realtype T,const int k);

  void ConstructKinvq();
  void ComputeDq(const bool,const bool);
  void ComputeSelfEnergy(const bool);

  void SetQsToZero(); // routine to set some q's to zero in self-energy
  void MakeRandomSigma();

  void MakeSymmetric(VecMat<complex<realtype>>&);

  void SolveSelfConsistentEquation(vector<realtype> Delta);

  void Solve(const NumberList delta,const bool pinfo)
  {
    if(TRACE) cout << "Starting Solve with Delta= " << delta << " Printinfo= " << pinfo <<
endl;
    logfile << "Starting Solver with Delta= " << delta << " Printinfo= " << pinfo << endl;

    for(int s=0; s<NSUBL; s++){Delta[s]=delta.v[s];}

#ifdef PHONONS
    for(int s=0; s<NELASTIC; s++){epsilon[s]=0.01;} // default starting value 0.01
#endif


    Printinfo=pinfo;
    //    Sigma = rule.GetInitialState(); // copy the Initial guess for Sigma
    SolveSelfConsistentEquation(Delta);
    if(TRACE) cout << "Done Solve" << endl;
  }


 private:
  Rule& rule;

  int dim;              // the number of dimensions
  vector<int> dims;     // the size of each dimension

  const int Vq; // number of q-space sites.
```

```
  const realtype invVq;
  const realtype invSqrtVq;

  vector<realtype> Delta;


#ifdef PRESERVESYMMETRY
  int TransformationPeriod;
  vector<int> TransformationTable;
#endif

  bool Printinfo; // set this to print out correlation functions
  int lineid;

  realtype mineigenvalue; // for storing the minimum SigmaE value
  realtype currT; // for storing the current value of the temperature

  VecMat<complex<realtype>>& Jq;
  // The actual storage areas
  VecMat<complex<realtype>> A1;  // holds Kq,Kinvq
  VecMat<complex<realtype>> A2;  // holds Sigmaq,
  VecMat<complex<realtype>> B;  // holds Dq,Dinvq

  vector<complex<realtype>> F1q ; // holds intermediate Fourier-transform
  vector<complex<realtype>> F2q ; // holds intermediate Fourier-transform


#ifdef FFTS_INPLACE
  VecMat<complex<realtype>>& A1r; // holds Kinvr
  VecMat<complex<realtype>>& A2r; // holds Sigmar
  VecMat<complex<realtype>>& Br; // holds Dr
  vector<complex<realtype>>& F1r; //
  vector<complex<realtype>>& F2r; //
#else
  VecMat<complex<realtype>> A1r;  // holds Kinvr
  VecMat<complex<realtype>> A2r;  // holds Sigmar
  VecMat<complex<realtype>> Br;  // holds Dr
  vector<complex<realtype>> F1r; //
  vector<complex<realtype>> F2r; //
#endif



  FFTWPLAN A1q_to_A1r;
  FFTWPLAN A1r_to_A1q;
  FFTWPLAN A2q_to_A2r;
  FFTWPLAN A2r_to_A2q;

  FFTWPLAN Bq_to_Br;
  FFTWPLAN Br_to_Bq;

  FFTWPLAN F1q_to_F1r;
  FFTWPLAN F1r_to_F1q;

  FFTWPLAN F2q_to_F2r;
  FFTWPLAN F2r_to_F2q;


  // the following references are used for readability of the code


  VecMat<complex<realtype>>& Kq;     // points to the A1 array
  VecMat<complex<realtype>>& Kinvq;  // points to the A1 array
  VecMat<complex<realtype>>& Kinvr;  // points to the A1 array

  VecMat<complex<realtype>>& Sigmar; // points to the A2 array
  VecMat<complex<realtype>>& Sigmaq; // points to the A2 array
```

```
  VecMat<complex<realtype>>& Dq;     // points to the B array
  VecMat<complex<realtype>>& Dinvq;  // points to the B array
  VecMat<complex<realtype>>& Dr;     // points to the B array

#ifdef PHONONS
  VecMat<complex<realtype>>& f; // holds the vertex information
  VecMat<complex<realtype>>& g; // points to rule

  vector<VecMat<complex<realtype>>*> gel;
#endif
  vector<realtype> epsilon;  // amplitude of elastic modes
};


Driver::Driver(Rule& r):rule(r),dim(la.D()),dims(la.SiteqDims()),Vq(la.SiteqVol()),invVq(st
atic_cast<realtype>(1.)/Vq),invSqrtVq(1./sqrt(Vq)),Delta(NSUBL),Printinfo(false),lineid(0),
Jq(r.Jq),
  A1(Vq,NMAT,NMAT),A2(Vq,NMAT,NMAT),B(Vq,NDMAT,NDMAT),F1q(Vq),F2q(Vq),
#ifdef FFTS_INPLACE
  A1r(A1),A2r(A2),Br(B),F1r(F1q),F2r(F2q),
#else
  A1r(Vq,NMAT,NMAT),A2r(Vq,NMAT,NMAT),Br(Vq,NDMAT,NDMAT),F1r(Vq),F2r(Vq),
#endif
  Kq(A1),
  Kinvq(A1),Kinvr(A1r),Sigmar(A2r),Sigmaq(A2),
  Dq(B),Dinvq(B),Dr(Br)
#ifdef PHONONS
  ,f(r.Getf()),g(r.g)
  ,epsilon(NELASTIC)
#else
  ,epsilon(0)
#endif
{
  if(TRACE) cout << "Initializing solver " << endl;

  //Setting up fftw_plans, in-place ffts:
  /*
  fftw_plan fftw_plan_many_dft(int rank, const int *n, int howmany,
                               fftw_complex *in, const int *inembed,
                               int istride, int idist,
                               fftw_complex *out, const int *onembed,
                               int ostride, int odist,
                               int sign, unsigned flags);
  */
  //  unsigned flags=FFTW_MEASURE;
  unsigned flags=( Vq > 1000 ? FFTW_PATIENT: FFTW_ESTIMATE);
  //unsigned flags=FFTW_PATIENT;

  logfile << "Making FFT plans" << endl;
  FFTWCOMPLEX* A1_ptr  =reinterpret_cast<FFTWCOMPLEX*>(A1.start());
  FFTWCOMPLEX* A1r_ptr =reinterpret_cast<FFTWCOMPLEX*>(A1r.start());
  FFTWCOMPLEX* A2_ptr  =reinterpret_cast<FFTWCOMPLEX*>(A2.start());
  FFTWCOMPLEX* A2r_ptr =reinterpret_cast<FFTWCOMPLEX*>(A2r.start());
  FFTWCOMPLEX* B_ptr  =reinterpret_cast<FFTWCOMPLEX*>(B.start());
  FFTWCOMPLEX* Br_ptr =reinterpret_cast<FFTWCOMPLEX*>(Br.start());

  FFTWCOMPLEX* F1q_ptr=reinterpret_cast<FFTWCOMPLEX*>(&F1q[0]);
  FFTWCOMPLEX* F1r_ptr=reinterpret_cast<FFTWCOMPLEX*>(&F1r[0]);
  FFTWCOMPLEX* F2q_ptr=reinterpret_cast<FFTWCOMPLEX*>(&F2q[0]);
  FFTWCOMPLEX* F2r_ptr=reinterpret_cast<FFTWCOMPLEX*>(&F2r[0]);


#ifdef LONGDOUBLE
  A1q_to_A1r = fftwl_plan_many_dft(dim,&dims[0],NMAT2 ,A1_ptr ,0,NMAT2 ,1,A1r_ptr,0,NMAT2 ,
1,-1,flags);
  A1r_to_A1q = fftwl_plan_many_dft(dim,&dims[0],NMAT2 ,A1r_ptr,0,NMAT2 ,1,A1_ptr ,0,NMAT2 ,
1,+1,flags);
  A2q_to_A2r = fftwl_plan_many_dft(dim,&dims[0],NMAT2 ,A2_ptr ,0,NMAT2 ,1,A2r_ptr,0,NMAT2 ,
```

```
1,-1,flags);
  A2r_to_A2q = fftwl_plan_many_dft(dim,&dims[0],NMAT2 ,A2r_ptr,0,NMAT2 ,1,A2_ptr ,0,NMAT2 ,
1,+1,flags);
  Bq_to_Br   = fftwl_plan_many_dft(dim,&dims[0],NDMAT2,B_ptr  ,0,NDMAT2,1,Br_ptr ,0,NDMAT2,
1,-1,flags);
  Br_to_Bq   = fftwl_plan_many_dft(dim,&dims[0],NDMAT2,Br_ptr ,0,NDMAT2,1,B_ptr  ,0,NDMAT2,
1,+1,flags);
  F1q_to_F1r = fftwl_plan_many_dft(dim,&dims[0],1,F1q_ptr,0,1,1,F1r_ptr,0,1,1,-1,flags);
  F1r_to_F1q = fftwl_plan_many_dft(dim,&dims[0],1,F1r_ptr,0,1,1,F1q_ptr,0,1,1,+1,flags);
  F2q_to_F2r = fftwl_plan_many_dft(dim,&dims[0],1,F2q_ptr,0,1,1,F2r_ptr,0,1,1,-1,flags);
  F2r_to_F2q = fftwl_plan_many_dft(dim,&dims[0],1,F2r_ptr,0,1,1,F2q_ptr,0,1,1,+1,flags);

#elif defined FLOAT
  A1q_to_A1r = fftwf_plan_many_dft(dim,&dims[0],NMAT2 ,A1_ptr ,0,NMAT2 ,1,A1r_ptr,0,NMAT2 ,
1,-1,flags);
  A1r_to_A1q = fftwf_plan_many_dft(dim,&dims[0],NMAT2 ,A1r_ptr,0,NMAT2 ,1,A1_ptr ,0,NMAT2 ,
1,+1,flags);
  A2q_to_A2r = fftwf_plan_many_dft(dim,&dims[0],NMAT2 ,A2_ptr ,0,NMAT2 ,1,A2r_ptr,0,NMAT2 ,
1,-1,flags);
  A2r_to_A2q = fftwf_plan_many_dft(dim,&dims[0],NMAT2 ,A2r_ptr,0,NMAT2 ,1,A2_ptr ,0,NMAT2 ,
1,+1,flags);
  Bq_to_Br   = fftwf_plan_many_dft(dim,&dims[0],NDMAT2,B_ptr  ,0,NDMAT2,1,Br_ptr ,0,NDMAT2,
1,-1,flags);
  Br_to_Bq   = fftwf_plan_many_dft(dim,&dims[0],NDMAT2,Br_ptr ,0,NDMAT2,1,B_ptr  ,0,NDMAT2,
1,+1,flags);
  F1q_to_F1r = fftwf_plan_many_dft(dim,&dims[0],1,F1q_ptr,0,1,1,F1r_ptr,0,1,1,-1,flags);
  F1r_to_F1q = fftwf_plan_many_dft(dim,&dims[0],1,F1r_ptr,0,1,1,F1q_ptr,0,1,1,+1,flags);
  F2q_to_F2r = fftwf_plan_many_dft(dim,&dims[0],1,F2q_ptr,0,1,1,F2r_ptr,0,1,1,-1,flags);
  F2r_to_F2q = fftwf_plan_many_dft(dim,&dims[0],1,F2r_ptr,0,1,1,F2q_ptr,0,1,1,+1,flags);

#else
  A1q_to_A1r = fftw_plan_many_dft(dim,&dims[0],NMAT2 ,A1_ptr ,0,NMAT2 ,1,A1r_ptr,0,NMAT2 ,1
,-1,flags);
  A1r_to_A1q = fftw_plan_many_dft(dim,&dims[0],NMAT2 ,A1r_ptr,0,NMAT2 ,1,A1_ptr ,0,NMAT2 ,1
,+1,flags);
  A2q_to_A2r = fftw_plan_many_dft(dim,&dims[0],NMAT2 ,A2_ptr ,0,NMAT2 ,1,A2r_ptr,0,NMAT2 ,1
,-1,flags);
  A2r_to_A2q = fftw_plan_many_dft(dim,&dims[0],NMAT2 ,A2r_ptr,0,NMAT2 ,1,A2_ptr ,0,NMAT2 ,1
,+1,flags);
  Bq_to_Br   = fftw_plan_many_dft(dim,&dims[0],NDMAT2,B_ptr  ,0,NDMAT2,1,Br_ptr ,0,NDMAT2,1
,-1,flags);
  Br_to_Bq   = fftw_plan_many_dft(dim,&dims[0],NDMAT2,Br_ptr ,0,NDMAT2,1,B_ptr  ,0,NDMAT2,1
,+1,flags);
  F1q_to_F1r = fftw_plan_many_dft(dim,&dims[0],1,F1q_ptr,0,1,0,F1r_ptr,0,1,0,-1,flags);
  F1r_to_F1q = fftw_plan_many_dft(dim,&dims[0],1,F1r_ptr,0,1,0,F1q_ptr,0,1,0,+1,flags);
  F2q_to_F2r = fftw_plan_many_dft(dim,&dims[0],1,F2q_ptr,0,1,0,F2r_ptr,0,1,0,-1,flags);
  F2r_to_F2q = fftw_plan_many_dft(dim,&dims[0],1,F2r_ptr,0,1,0,F2q_ptr,0,1,0,+1,flags);
#endif

  logfile << "Done making FFT plans" << endl;


#ifdef PRESERVESYMMETRY
  TransformationPeriod=lattice.TransformationPeriod;
  TransformationTable=lattice.TransformationTable;
#endif


  if(TRACE) cout << "Done initializing solver " << endl;
};


realtype Driver::CalculateT(int sl)
{
  if(TRACE) cout << "Starting CalculateT for sublattice: " << sl << endl;
  realtype sumalpha=0.;
  vector<realtype> alpha(NSPIN);
  for(int s=0; s<NSPIN; s++)
```

```
    {
      int m=mindx(s,sl); // make the composite index.
      alpha[s] = NFAKESPINTRACE*real(Sumq(Kinvq,m,m))/(2.*Vq);
      sumalpha += alpha[s];
    }

  if(TRACE)
    {
      cout << "T sums: ";
      for(int s=0; s<NSPIN; s++){ cout << "alpha[" << s << "]=" << alpha[s] << " ";}
      cout << endl;
    }

  realtype T = 1./sumalpha;

  if(TRACE) cout << "Done CalculateT "  << T << endl;
  return T;
}


void Driver::CalculateTs(vector<realtype>& Ts)
{
  if(TRACE) cout << "Starting CalculateTs " << endl;
  for(int s=0; s<NSUBL; s++)
    {
      Ts[s]=CalculateT(s);
    }
}


#ifdef PHONONS
vector<realtype> Driver::CalculateEpsilonsOverT()
{
  if(TRACE) cout << "Starting CalculateEpsilonsOverT" << endl;

  vector<realtype> epsoverT(NELASTIC);
  for(int i=0; i<NELASTIC; i++)
    {
      realtype mui= rule.elasticeigenvalues[i];
      realtype sum=0.;
      if(mui != 0.)
        {
          for(int qi=0; qi<Vq; qi++)
            {
              SMatrix<complex<realtype>> tmp(NMAT,NMAT);
              tmp=Kinvq[qi];
              tmp *= (*rule.gelptrs[i])[qi];
              sum += NFAKESPINTRACE*real(tr(tmp));
            }
          sum *= -1./(2.*Vq*mui);
        }
      epsoverT[i]= sum;
    }

  if(TRACE) cout << "Done CalculateEpsilonsOverT "  << endl;
  return epsoverT;
}
#endif


vector<obstype> Driver::CalculateSpinOrderPars(realtype T)
{
  if(TRACE) cout << "Starting CalculateSpinOrderPars" << endl;

  vector<obstype> opars(NSPINOBSERVABLES);
  for(int j=0; j<NSPINOBSERVABLES; j++)
    {
      obstype sum=0.;
```

```
      KernelFunction* f=spinobservables[j];

      for(int qi=0; qi<Vq; qi++)
        {
          for(int m1=0; m1<Kinvq.Nrows; m1++)
            for(int m2=0; m2<Kinvq.Ncols; m2++)
              sum+= (*f)(qi,m1,m2)*Kinvq(qi,m1,m2);
        }
      opars[j]=0.5*T*invVq*sum;
    }

  if(TRACE) cout << "Done CalculateSpinOrderPars" << endl;
  return opars;
}



vector<obstype> Driver::CalculateOrderPars(realtype T,int m1,int m2)
{
  if(TRACE) cout << "Starting CalculateOrderPars" << endl;

  vector<obstype> opars(NOBSERVABLES);
  for(int j=0; j<NOBSERVABLES; j++)
    {
      obstype sum=0.;
      vector<obstype>& f=rule.GetIrrep(j);

      for(int i=0; i<Vq; i++)
        {
          sum+= f[i]*Kinvq(i,m1,m2);
        }
      opars[j]=0.5*T*invVq*sum;
    }

  if(TRACE) cout << "Done CalculateOrderPars" << endl;
  return opars;
}

#ifdef PRESERVESYMMETRY
void Driver::MakeSymmetric(VecMat<complex<realtype>>& m)
{
  if(TRACE) cout << "Starting MakeSymmetric" << endl;
  if( NSUBL !=1){ cout << "MakeSymmetric works only for NSUBL=1" << endl; exit(1);}

  complex<realtype>* mstart=m.start(); // This only works for NSUBL=1

  //  cout << "m=" << m << endl;

  VecMat<complex<realtype>> temp(m);
  complex<realtype>* tempstart=temp.start(); // This only works for NSUBL=1

  //  cout << "temp=" << temp << endl;

  int p=1; // p=0 is the identity transformation
  vector<int> ThisT(TransformationTable);

  //cout << "Period: " << TransformationPeriod << endl;
  //cout << "Vq=" << Vq << endl;


  while( p < TransformationPeriod)
    {
      for(int i=0; i<Vq; i++)
        {
          int q=ThisT[i];
          tempstart[i]+=mstart[q];
          ThisT[i] = TransformationTable[q]; // update ThisT for the next period
        }
```

```
      p++;
    }

  // Average over all transformations and transfer the result to the input array
  realtype invperiod=1./TransformationPeriod;
  for(int i=0; i<Vq; i++){ mstart[i]=tempstart[i]*invperiod;}

  if(TRACE) cout << "End of MakeSymmetric" << endl;

}
#endif



// Free energy per unit volume
// we use the convention \nu=\nup
realtype Driver::CalculateFreeEnergy(realtype T)
{
  if(TRACE) cout << "Starting CalculateFreeEnergy " << endl;

  const int Ns=NSPIN*NFAKESPINTRACE;

  realtype f=0;

  if(TRACE) cout << "--- T  = " << T << endl;
  // constants:
  //realtype betaf_constants= -( 0.5*NSUBL*log(2.*Vq) + 0.5*NSUBL*(NS-1)*log(PI));
  realtype betaf_constants = -0.5*NSUBL*log(Vq) -0.5*NSUBL*(Ns-2)*log(PI) - 0.5*NSUBL*log(T
WOPI);


  if(TRACE) cout << "betaf_constants  = " << betaf_constants << endl;

  f += T*betaf_constants;

  //temperature factors
  realtype betaf_Tdep = 0.5*NSUBL*(Ns-2)*log(1./T);

  if(TRACE) cout << "betaf_Tdep  = " << betaf_Tdep << endl;

  f += T*betaf_Tdep;


#ifdef PHONONS
  //elastic modes
  realtype betaf_elastic = 0;
  for(int i=0; i<NELASTIC; i++){ betaf_elastic += 0.5*epsilon[i]*epsilon[i]*rule.elasticeig
envalues[i];}

  if(TRACE) cout << "betaf_elastic  = " << betaf_elastic << endl;

  f += T*betaf_elastic;

  // constants:
  realtype betaf_phononconst = -0.5*2.*NMODE*log(TWOPI);
  if(TRACE) cout << "betaf_phononconst  = " << betaf_phononconst << endl;

  f += T*betaf_phononconst;

#if !defined ELASTICONLY
  //the phonon spectrum
  realtype betaf_phonons = 2.*rule.GetSumLogOmegaoverV(); // 2 both X^2 and P^2

  if(TRACE) cout << "betaf_phonons  = " << betaf_phonons << endl;

  f += T*betaf_phonons;
#endif
```

```
#endif

  //Must correct the Delta values for the subtraction of the minimum from SigmaE
  for(int i=0; i<NSUBL; i++) f += -(Delta[i]-mineigenvalue);

  if(TRACE) cout << "betaf_delta      = " << -(Delta[0]-mineigenvalue)/T << " (Delta= " <<
Delta[0] << " mineig: " << mineigenvalue << ")" << endl;

  realtype betaf_logKinvq   = -0.5*invVq*NFAKESPINTRACE*SumLogDet(Kinvq);

  if(TRACE) cout << "betaf_logKinvq   =  " << betaf_logKinvq << endl;
  f += T*betaf_logKinvq;

  ComputeDq(false,true); // excludeqzero=false, preserveinput=true not to jeopardize Keff

  realtype betaf_logD       = -0.5*invVq*SumLogDet(Dq);
  if(TRACE) cout << "betaf_logD       =  " << betaf_logD << endl;
  f += T*betaf_logD;

  ComputeSelfEnergy(true); // ensure that Kinvq is the same.

  realtype betaf_KinvqSigma = -0.5*invVq*NFAKESPINTRACE*SumTr(Kinvq,Sigmaq);
  if(TRACE) cout << "betaf_KinvqSigma = " << betaf_KinvqSigma << endl;
  f += T*betaf_KinvqSigma;

  // the correction to the saddle-point are already taken into account
  return f;
}


/* The old free energy routine
// Free energy per unit volume
// we use the convention \nu=\nup
realtype Driver::CalculateFreeEnergy(realtype T)
{
  if(TRACE) cout << "Starting CalculateFreeEnergy " << endl;

  realtype f=0;

  if(TRACE) cout << "--- T  = " << T << endl;
  // constants:
  realtype betaf_constants= -( 0.5*NSUBL*log(2.*Vq) + 0.5*NSUBL*(NS-1)*log(PI));
  if(TRACE) cout << "betaf_constants  = " << betaf_constants << endl;

  f += T*betaf_constants;

  //Must correct the Delta values for the subtraction of the minimum from SigmaE
#ifdef SOFTCONSTRAINT
  for(int i=0; i<NSUBL; i++) f += -(Delta[i]-mineigenvalue)*(Delta[i]-mineigenvalue)/(4.*g*
T);
#endif
  for(int i=0; i<NSUBL; i++) f += -(Delta[i]-mineigenvalue);

  if(TRACE) cout << "betaf_delta      = " << -(Delta[0]-mineigenvalue)/T << " (Delta= " <<
Delta[0] << " mineig: " << mineigenvalue << ")" << endl;

  // NSUBL*log(T) is needed because T is not part of Kinvq in the program
  realtype betaf_logKinvq   = -0.5*NS*(invVq*SumLogDet(Kinvq) + NSUBL*log(T));
  //  cout << "invVqq*SumLogDet= " << invVq*SumLogDet(Kinvq) << " log(T)= " << log(T) << en
dl;
  if(TRACE) cout << "betaf_logKinvq   =  " << betaf_logKinvq << endl;
          f += T*betaf_logKinvq;

  ComputeDq(false,true); // excludeqzero=false, preserveinput=true not to jeopardize Keff

  // NSUBL*2.*log(T) needed because T is not part of Dq in the program
  realtype betaf_logD       = -0.5*invVq*( SumLogDet(Dq) - Vq*NSUBL*2.*log(T) );
  if(TRACE) cout << "betaf_logD       =  " << betaf_logD << endl;
```

```
  f += T*betaf_logD;

  ComputeSelfEnergy(true); // ensure that Kinvq is the same.

  realtype betaf_KinvqSigma = -0.5*NS*invVq*SumTr(Kinvq,Sigmaq);
  if(TRACE) cout << "betaf_KinvqSigma = " << betaf_KinvqSigma << endl;
  f += T*betaf_KinvqSigma;

  // the correction to the saddle-point are already taken into account
  return f;
}
*/

// ComputeDq() computes the renormalized constraint propagator
//
// Input: Kinvq (stored in A1).
// Output: Dq (stored in B)
//
// The routine uses in-place FFTs so info contained in A and B are modified
// On output:
// A1 = Kinvr, unless preserveinput=true, then A1=Kinvq
// B = Dq
//
// FFTW omits volume prefactors in fourier transforms, we adopt the convention that the Fou
rier transform
// is without prefactors in going from q->r, and the prefactor 1/Vq is inserted on going fr
om r->q.
// This means that using FFTW for transforming q->r->q, one should divide the result by Vq.

void Driver::ComputeDq(const bool excludeqzero=true, const bool preserveinput=false)
{
#ifdef PHONONS
  if(NSUBL !=1){ cout << "Error:ComputeDq is not implemented for NSUBL !=1, exiting." << en
dl; exit(1);}
#endif

  if(TRACE) cout << "Starting ComputeDq, excludeqzero=" << excludeqzero << ",preserveinput=
" << preserveinput << endl;

  if(TRACE){SanityCheck(Kinvq,"Kinvq, Initializing ComputeDq");}


  FFTWEXECUTE(A1q_to_A1r); // Kinvq->Kinvr, stored in Ar, after: Ar=Kinvr*Sqrt(Vq)
  Kinvr *= invSqrtVq; // Ar=Kinvr


#ifdef FORCEINVERSIONSYMMETRY
  MakeReal(Kinvr);  // is this needed here?
  MakeInversionTransposedSymmetric(Kinvr);
#endif

  // first compute the constraint block

  complex<realtype> tmp(0);

  for(int m1=0; m1<NSUBL; m1++)
    for(int m2=m1; m2<NSUBL; m2++)
      {
        for(int r=0; r<Vq; r++)
          {
            complex<realtype> tt(0.);
            for(int s1=0; s1<NSPIN; s1++)
              for(int s2=0; s2<NSPIN; s2++)
                {
                  int alpha=mindx(s1,m2); // alpha_s = m2
                  int delta=mindx(s2,m1); // delta_s = m1

                  tmp=Kinvr(r,alpha,delta);
```

```
                  tt += tmp*tmp;
                }
            F1r[r]=0.5*NFAKESPINTRACE*tt;
          }
        /*
        cout << "m1=" << m1 << " m2=" << m2 << endl;
        cout << "F1r" << endl;
        for(int r=0; r<Vq; r++) cout << F1r[r] << " ";
        cout << endl;
        */
        FFTWEXECUTE(F1r_to_F1q); // F1r->F1q
        /*
        cout << "F1q" << endl;
        for(int q=0; q<Vq; q++) cout << F1q[q] << " ";
        cout << endl;
        */

        for(int qi=0; qi<Vq; qi++)
          {
            Dinvq(qi,m1,m2)=F1q[qi];
            if(m2 != m1) Dinvq(qi,m2,m1)=conj(F1q[qi]);
          }
      }

  //  cout << "Dinvq after constraint-constraint part " << Dinvq << endl;

#ifdef PHONONS
  // the offdiagonal phonon-constraint part
#ifdef CPOSITIVE
  realtype multiplier=2.;
#else
  realtype multiplier=1.;
#endif

  for(int m1=0; m1<NSUBL; m1++) // m1 must be 0 here because phonons NSUBL=1
    for(int c=0; c<NC; c++)
      {
        for(int r=0; r<Vq; r++)
          {
            SMatrix<complex<realtype>> my(NMAT,NMAT);
            my  = g[c];
            my *= Kinvr[r];
            int mrpc = la.rAdd(la.GetInversionIndx(r),clist[c]); // index of -r+c

            my *= Kinvr[mrpc];

            F1r[r] = NFAKESPINTRACE*tr(my);
          }

        // do fourier-transform of theta
        FFTWEXECUTE(F1r_to_F1q);

        for(int qi=0; qi<Vq; qi++)
          {
            for(int m2=NSUBL; m2<NDMAT; m2++)
              {
                int n2=m2-NSUBL;

                if(c==0){Dinvq(qi,m1,m2)=0;}
                Dinvq(qi,m1,m2)+= multiplier*0.5*invSqrtVq*F1q[qi]*f(qi,c,n2)*conj(expi(la.
qr(qi,clist[c])));
                /*
                      cout << "F1q[" << qi << "]=" << F1q[qi] << " F1q[-" << qi << "]=" <<
F1q[la.GetInversionIndx(qi)]
                      << "f(" << qi << "," << c << "," << n2 << ")=" << f(qi,c,n2)
                      << "f(-" << qi << "," << c << "," << n2 << ")=" << f(la.GetInversionI
ndx(qi),c,n2) << endl;
                */
```

```
            }
          }
      }

  // cout << "before setting using inversion: Dinvq=" << Dinvq << endl;

  // Set the constraint-phonon part.
  for(int qi=0; qi<Vq; qi++)
    for(int m1=0; m1<NSUBL; m1++)
      for(int m2=NSUBL; m2<NDMAT; m2++)
        {
          Dinvq(la.GetInversionIndx(qi),m2,m1)=Dinvq(qi,m1,m2);
        }

  //cout << "after phonon-constraint: Dinvq=" << Dinvq << endl;


  // finally the phonon-phonon part
#ifdef CPOSITIVE
    for(int c2=0; c2<NC; c2++)
      for(int c4=0; c4<NC; c4++)
        {
          Triplet myivec=clist[c2]+clist[c4];

          for(int r=0; r<Vq; r++)
            {
              SMatrix<complex<realtype>> my(NMAT,NMAT);

              my  = g[c4];
              my *= Kinvr[r];
              my *= g[c2];

              int mrpc2c4=la.rAdd(la.GetInversionIndx(r),myivec);

              my *= Kinvr[mrpc2c4];

              F1r[r]=NFAKESPINTRACE*tr(my);
            }
          FFTWEXECUTE(F1r_to_F1q); // F1r->F1q;


          for(int m1=NSUBL; m1<NDMAT; m1++)
            for(int m2=m1; m2<NDMAT; m2++)
              {

                int n1=m1-NSUBL;
                int n2=m2-NSUBL;

                for(int qi=0; qi<Vq; qi++)
                  {
                    if(c2==0 && c4==0){Dinvq(qi,m1,m2);}
                    Dinvq(qi,m1,m2)+=-invVq*F1q[qi]*conj(f(qi,c2,n1))*f(qi,c4,n2)*conj(expi
(la.qr(qi,clist[c4])));
                  }
              }


          myivec=clist[c2]-clist[c4];

          for(int r=0; r<Vq; r++)
            {
              SMatrix<complex<realtype>> my(NMAT,NMAT);

              my  = g[c4];
              my.Transpose();
              my *= Kinvr[r];
              my *= g[c2];
```

```
              int mrc2mc4=la.rAdd(la.GetInversionIndx(r),myivec);

              my *= Kinvr[mrc2mc4];

              F1r[r]=NFAKESPINTRACE*tr(my);
            }
          FFTWEXECUTE(F1r_to_F1q); // F1r->F1q;


          for(int m1=NSUBL; m1<NDMAT; m1++)
            for(int m2=m1; m2<NDMAT; m2++)
              {

                int n1=m1-NSUBL;
                int n2=m2-NSUBL;

                for(int qi=0; qi<Vq; qi++)
                  {
                    Dinvq(qi,m1,m2)+=-invVq*F1q[qi]*conj(f(qi,c2,n1))*f(qi,c4,n2);
                  }
              }
        }
#else
  for(int c2=0; c2<NC; c2++)
    for(int c4=0; c4<NC; c4++)
      {
        Triplet myivec=clist[c2]+clist[c4];

        for(int i=0; i<Vq; i++)
          {
            SMatrix<complex<realtype>> my(NMAT,NMAT);

            my  = g[c4];
            my *= Kinvr[i];
            my *= g[c2];

            int newindx=la.rAdd(la.GetInversionIndx(i),myivec);

              my *= Kinvr[newindx];

              F1r[i]=0.5*NFAKESPINTRACE*tr(my);
          }
        FFTWEXECUTE(F1r_to_F1q); // F1r->F1q;


        for(int m1=NSUBL; m1<NDMAT; m1++)
          for(int m2=m1; m2<NDMAT; m2++)
            {

              int n1=m1-NSUBL;
              int n2=m2-NSUBL;

              for(int qi=0; qi<Vq; qi++)
                {
                  if(c2==0 && c4==0){Dinvq(qi,m1,m2);}
                  Dinvq(qi,m1,m2)+=-invVq*F1q[qi]*conj(f(qi,c2,n1))*f(qi,c4,n2)*conj(expi(l
a.qr(qi,clist[c4])));
                }
            }
      }
#endif



  // Set the remainding phonon-phonon part.
  for(int qi=0; qi<Vq; qi++)
```

```
    for(int m1=NSUBL; m1<NDMAT; m1++)
      for(int m2=m1; m2<NDMAT; m2++)
        {
          Dinvq(qi,m2,m1)=conj(Dinvq(qi,m1,m2));
        }


  // add the bare phonon part

  realtype barepart=1./currT; // the one-half is included in the def. of propagator
  for(int qi=0; qi<Vq; qi++)
    {
      for(int n=0; n<NMODE; n++)
        {
          int m=n+NSUBL;
          Dinvq(qi,m,m) += barepart;
        }
    }
#endif

  //  cout << "after phonon-phonon: Dinvq=" << Dinvq << endl;

  // FORCING
  MakeMixedHermitian(Dinvq,NSUBL,NSUBL);
  MakeInversionTransposedSymmetric(Dinvq);


  Chomp(Dinvq);

  if(TRACE){SanityCheck(Dinvq,"Dinvq, after constructing it",false);}

  MatrixInverse(Dinvq); // B = Dq

  if(TRACE){SanityCheck(Dq,"Dq, after inverting Dinvq",false);}

  if(excludeqzero) Setqzerotozero(Dq);


  if(preserveinput)
    {
      FFTWEXECUTE(A1r_to_A1q);  // Kinvr->Kinvq, so after transform: Aq=Kinvq*sqrt(Vq)
      Kinvq *= invSqrtVq; // Aq=Kinvq;

#ifdef PRESERVESYMMETRY
      MakeSymmetric(Kinvq);
#endif
      MakeHermitian(Kinvq);
    }

  if(TRACE){SanityCheck(Dq,"Dq, at end of ComputeDq",false);}

  if(TRACE) cout << "Done with ComputeDq " << endl;
}


// ComputeSelfEnergy() computes the self-energy from the self-consistent equations
//
// Input: Kinv_q (stored in A1).
// Output: Sigma_q (stored in A2)
//
// The routine uses in-place FFTs so info contained in A and B are modified
// On output:
// A1 = Kinvr, unless preserveinput=true, then A1=Kinvq
// A2 = Sigmaq
//
// FFTW omits volume prefactors in fourier transforms, so it just carries out the sums
// Therefore volume factors must be included explicitly.
void Driver::ComputeSelfEnergy(const bool preserveinput=false)
```

```
{
  if(TRACE) cout << "Starting ComputeSelfEnergy" << endl;

  if(TRACE){SanityCheck(Kinvq,"Kinvq, at start of ComputeSelfEnergy",true);}


  ComputeDq(true,false); // exclude zero mode, do not preserve input here, Ar must contain
Kinvr for ComputeSelfEnergy to work.

  Sigmaq.SetToZero();


  for(int alpha=0; alpha<NMAT; alpha++)
    for(int delta=alpha; delta<NMAT; delta++)
      {
        //      const int alpha_s=spin(alpha);
        const int alpha_l=subl(alpha);

        //      const int delta_s=spin(delta);
        const int delta_l=subl(delta);

        // Term 1:
        for(int q=0; q<Vq; q++){F1q[q]=Dq(q,delta_l,alpha_l);}
        FFTWEXECUTE(F1q_to_F1r); // we take the Fourier volume factor X1=0

        for(int r=0; r<Vq; r++){F2r[r]=Kinvr(la.GetInversionIndx(r),alpha,delta)*F1r[r];}
        FFTWEXECUTE(F2r_to_F2q);

        for(int k=0; k<Vq; k++){Sigmaq(k,alpha,delta) += invSqrtVq*F2q[k];}

#if defined PHONONS && !defined ELASTICONLY
        if(NSUBL != 1){cout << "Error: PHONONS are not implemented for NSUBL!=1, exiting" <
< endl; exit(1);}


#ifdef CPOSITIVE
        // Term 2
        for(int c=0; c<NC; c++)
          {
            for(int q=0; q<Vq; q++)
              {
                F1q[q]=0;
                for(int n=0; n<NMODE; n++)
                  {
                    F1q[q]+=Dq(q,NSUBL+n,0)*f(q,c,n);
                  }
              }
            FFTWEXECUTE(F1q_to_F1r);


            for(int r=0; r<Vq; r++)
              {
                F2r[r]=0;
                for(int s=0; s<NSPIN; s++)
                  //              F2r[r]+=F1r[r]*conj(Kinvr(r,s,alpha))*g(c,s,delta); // Ki
nv(-r,alpha,s) = Kinv(r,s,alpha)
                    F2r[r]+=F1r[r]*Kinvr(r,s,alpha)*g(c,s,delta); // conj(Kinv(r,s,alpha))= K
inv(r,s,alpha)) ; real
              }

            FFTWEXECUTE(F2r_to_F2q);

            for(int k=0; k<Vq; k++){ Sigmaq(k,alpha,delta) += invVq*expi(la.qr(k,clist[c]))
*F2q[k];}

            for(int r=0; r<Vq; r++)
              {
                F2r[r]=0;
```

```
                int rpc=la.rAdd(r,clist[c]);
                for(int s=0; s<NSPIN; s++)
                  F2r[r]+=F1r[r]*Kinvr(rpc,s,alpha)*g(c,delta,s);
              }

            FFTWEXECUTE(F2r_to_F2q);

            for(int k=0; k<Vq; k++){ Sigmaq(k,alpha,delta) += invVq*F2q[k];}
          }


        // Term 3 (as term 2, but switch alpha and delta and take complex conj)
        for(int c=0; c<NC; c++)
          {
            for(int q=0; q<Vq; q++)
              {
                F1q[q]=0;
                for(int n=0; n<NMODE; n++)
                  {
                    F1q[q]+=Dq(q,NSUBL+n,0)*f(q,c,n);
                  }
              }
            FFTWEXECUTE(F1q_to_F1r);


            for(int r=0; r<Vq; r++)
              {
                F2r[r]=0;
                for(int s=0; s<NSPIN; s++)
                  //              F2r[r]+=F1r[r]*conj(Kinvr(r,s,alpha))*g(c,s,delta); // Ki
nv(-r,alpha,s) = Kinv(r,s,alpha)
                  F2r[r]+=F1r[r]*Kinvr(r,s,delta)*g(c,s,alpha); // conj(Kinv(r,s,alpha))= K
inv(r,s,alpha)) ; real
              }

            FFTWEXECUTE(F2r_to_F2q);

            for(int k=0; k<Vq; k++){ Sigmaq(k,alpha,delta) += invVq*conj(expi(la.qr(k,clist
[c]))*F2q[k]);}

            for(int r=0; r<Vq; r++)
              {
                F2r[r]=0;
                int rpc=la.rAdd(r,clist[c]);
                for(int s=0; s<NSPIN; s++)
                  F2r[r]+=F1r[r]*Kinvr(rpc,s,delta)*g(c,alpha,s);
              }

            FFTWEXECUTE(F2r_to_F2q);

            for(int k=0; k<Vq; k++){ Sigmaq(k,alpha,delta) += invVq*conj(F2q[k]);}
          }

        // Term4:
        for(int c1=0; c1<NC; c1++)
          for(int c2=0; c2<NC; c2++)
            {
              for(int q=0; q<Vq; q++)
                {
                  F1q[q]=0;

                  for(int n1=0; n1<NMODE; n1++)
                    for(int n2=0; n2<NMODE; n2++)
                      { F1q[q]+= conj(f(q,c1,n1))*f(q,c2,n2)*Dq(q,NSUBL+n2,NSUBL+n1)*expi(l
a.qr(q,clist[c1]));}
                }
              FFTWEXECUTE(F1q_to_F1r);
```

```
              for(int r=0; r<Vq; r++)
                {
                  F2r[r]=0;
                  for(int be=0; be<NSPIN; be++)
                    for(int ga=0; ga<NSPIN; ga++)
                      {
                        //                      F2r[r]+=g(c1,alpha,s1)*conj(Kinvr(r,s2,s1))
*g(c2,s2,delta); // Kinv(-r,s1,s2) = Kinv(r,s2,s1)^*
                        F2r[r]+=g(c1,alpha,be)*Kinvr(r,ga,be)*g(c2,ga,delta); // conj(Kinv(
r,s2,s1))=Kinv(r,s2,s1)
                      }
                  F2r[r] *= F1r[r];
                }

              FFTWEXECUTE(F2r_to_F2q);
              for(int k=0; k<Vq; k++){Sigmaq(k,alpha,delta)+= -invVq*invSqrtVq*F2q[k]*expi(
la.qr(k,clist[c1]))*expi(la.qr(k,clist[c2]));}


              for(int r=0; r<Vq; r++)
                {
                  int rpc2=la.rAdd(r,clist[c2]);
                  F2r[r]=0;
                  for(int be=0; be<NSPIN; be++)
                    for(int ga=0; ga<NSPIN; ga++)
                      {
                        F2r[r]+=g(c1,alpha,be)*Kinvr(rpc2,ga,be)*g(c2,delta,ga);
                      }
                  F2r[r] *= F1r[r];
                }

              FFTWEXECUTE(F2r_to_F2q);
              for(int k=0; k<Vq; k++){Sigmaq(k,alpha,delta)+= -invVq*invSqrtVq*F2q[k]*expi(
la.qr(k,clist[c1]));}


              for(int r=0; r<Vq; r++)
                {
                  int rpc1=la.rAdd(r,clist[c1]);
                  F2r[r]=0;
                  for(int be=0; be<NSPIN; be++)
                    for(int ga=0; ga<NSPIN; ga++)
                      {
                        F2r[r]+=g(c1,be,alpha)*Kinvr(rpc1,ga,be)*g(c2,ga,delta);
                      }
                  F2r[r] *= F1r[r];
                }

              FFTWEXECUTE(F2r_to_F2q);
              for(int k=0; k<Vq; k++){Sigmaq(k,alpha,delta)+= -invVq*invSqrtVq*F2q[k]*expi(
la.qr(k,clist[c2]));}

              for(int r=0; r<Vq; r++)
                {
                  int rpc1c2=la.rAdd(r,clist[c1]+clist[c2]);
                  F2r[r]=0;
                  for(int be=0; be<NSPIN; be++)
                    for(int ga=0; ga<NSPIN; ga++)
                      {
                        F2r[r]+=g(c1,be,alpha)*Kinvr(rpc1c2,ga,be)*g(c2,delta,ga);
                      }
                  F2r[r] *= F1r[r];
                }

              FFTWEXECUTE(F2r_to_F2q);
              for(int k=0; k<Vq; k++){Sigmaq(k,alpha,delta)+= -invVq*invSqrtVq*F2q[k];}
            }
```

```
#else
        // Term 2
        for(int c=0; c<NC; c++)
          {
            for(int q=0; q<Vq; q++)
              {
                F1q[q]=0;
                for(int n=0; n<NMODE; n++)
                  {
                    F1q[q]+=Dq(q,NSUBL+n,0)*f(q,c,n);
                  }
              }
            FFTWEXECUTE(F1q_to_F1r);


            for(int r=0; r<Vq; r++)
              {
                F2r[r]=0;
                for(int s=0; s<NSPIN; s++)
                  //              F2r[r]+=F1r[r]*conj(Kinvr(r,s,alpha))*g(c,s,delta); // Ki
nv(-r,alpha,s) = Kinv(r,s,alpha)
                  F2r[r]+=F1r[r]*Kinvr(r,s,alpha)*g(c,s,delta); // conj(Kinv(r,s,alpha))= K
inv(r,s,alpha)) ; real
              }

            FFTWEXECUTE(F2r_to_F2q);

            for(int k=0; k<Vq; k++){ Sigmaq(k,alpha,delta) += invVq*expi(la.qr(k,clist[c]))
*F2q[k];}
          }

        // Term 3 (as term 2, but switch alpha and delta and take complex conj)
        for(int c=0; c<NC; c++)
          {
            for(int q=0; q<Vq; q++)
              {
                F1q[q]=0;
                for(int n=0; n<NMODE; n++){ F1q[q]+=Dq(q,NSUBL+n,0)*f(q,c,n);}
              }
            FFTWEXECUTE(F1q_to_F1r);

            for(int r=0; r<Vq; r++)
              {
                F2r[r]=0;
                for(int s=0; s<NSPIN; s++)
                  //              F2r[r]+=F1r[r]*conj(Kinvr(r,s,delta))*g(c,s,alpha); // Ki
nv(-r,alpha,s) = Kinv(r,s,alpha)
                  F2r[r]+=F1r[r]*Kinvr(r,s,delta)*g(c,s,alpha); // conj(Kinv(r,s,delta))= K
inv(r,s,delta)
              }
            FFTWEXECUTE(F2r_to_F2q);

            for(int k=0; k<Vq; k++){ Sigmaq(k,alpha,delta) += invVq*conj(expi(la.qr(k,clist
[c]))*F2q[k]);}
          }


        // Term 4:
        for(int c1=0; c1<NC; c1++)
          for(int c2=0; c2<NC; c2++)
            {
              for(int q=0; q<Vq; q++)
                {
                  F1q[q]=0;

                  for(int n1=0; n1<NMODE; n1++)
                    for(int n2=0; n2<NMODE; n2++)
                      { F1q[q]+= conj(f(q,c1,n1))*f(q,c2,n2)*Dq(q,NSUBL+n2,NSUBL+n1)*expi(l
```

```
a.qr(q,clist[c1]));}
                }
              FFTWEXECUTE(F1q_to_F1r);

              for(int r=0; r<Vq; r++)
                {
                  F2r[r]=0;
                  for(int s1=0; s1<NSPIN; s1++)
                    for(int s2=0; s2<NSPIN; s2++)
                      {
                        //                      F2r[r]+=g(c1,alpha,s1)*conj(Kinvr(r,s2,s1))
*g(c2,s2,delta); // Kinv(-r,s1,s2) = Kinv(r,s2,s1)^*
                        F2r[r]+=g(c1,alpha,s1)*Kinvr(r,s2,s1)*g(c2,s2,delta); // conj(Kinv(
r,s2,s1))=Kinv(r,s2,s1)
                      }
                  F2r[r] *= F1r[r];
                }

              FFTWEXECUTE(F2r_to_F2q);
              for(int k=0; k<Vq; k++){Sigmaq(k,alpha,delta)+= -invVq*invSqrtVq*F2q[k]*expi(
la.qr(k,clist[c1]))*expi(la.qr(k,clist[c2]));}
            }
#endif
#endif
      }

#ifdef PRESERVESYMMETRY
  MakeSymmetric(Sigmaq);
#endif
  MakeHermitian(Sigmaq,false);

  if(TRACE) SanityCheck(Sigmaq,"Sigmaq after explicitly constructing it");

  if(preserveinput)
    {
      FFTWEXECUTE(A1r_to_A1q); // Kinvr->Kinvq, so after transform: A1q=Kinvq*sqrt(Vq)
      Kinvq *= invSqrtVq;

#ifdef PRESERVESYMMETRY
      MakeSymmetric(Kinvq);
#endif
      MakeHermitian(Kinvq);
    }

  if(TRACE) cout << "Done with ComputeSelfEnergy " << endl;
}




// Routine to build Kinq from Jq and Sigmaq
//
//
void Driver::ConstructKinvq()
{
  if(TRACE) cout << "Starting ConstructKinvq" << endl;

  Kq =  Jq;

  if(TRACE) cout << "Initializing Kq" << endl;

  if(TRACE){SanityCheck(Kq,"Kq, after setting Jq");}

  Kq += Sigmaq;

  if(TRACE){SanityCheck(Kq,"Kq, after adding Sigmaq");}
```

```
#ifdef PHONONS
  if(TRACE) cout << "Adding elastic modes " << endl;
  for(int j=0; j<NELASTIC; j++)
    {
      VecMat<complex<realtype>> temp(*rule.gelptrs[j]);
      temp *= epsilon[j];
      Kq += temp;
    }
#endif


  if(TRACE){SanityCheck(Kq,"Kq, after adding Elastic modes");}

  mineigenvalue=SubtractMinimumEigenvalue(Kq);

  if(TRACE){SanityCheck(Kq,"Kq, after subtracting eigenvalues");}

  AddDelta(Kq,Delta);

  if(TRACE){SanityCheck(Kq,"Kq, after adding Delta");}

  // construct K1inv
  MatrixInverse(Kq); // K = Kinv_q

  // Force correct properties on the matrix
  MakeHermitian(Kinvq);
  MakeInversionTransposedSymmetric(Kinvq);

  if(TRACE){SanityCheck(Kinvq,"Kinvq, after inverting");}

  if(TRACE) cout << "Done with ConstructKinvq" << endl;
}



void Driver::MakeRandomSigma()
{
  if(TRACE) cout << "Starting MakeRandomSigma()" << endl;
  logfile << "Making a random initialization of the self-energy" << endl;

  realtype da = par[DA];

  for(int m=0; m<NMAT; m++)
    {
      for(int i=0; i<Vq; i++)
        {
          complex<realtype> c=da*complex<realtype>(RAN(),0.); // real positive value
          Sigmaq(i,m,m)=c;
        }
    }

  if(NMAT>1)
    {
      for(int m1=0; m1<NMAT; m1++)
        for(int m2=m1+1; m2<NMAT; m2++)
          {
            for(int i=0; i<Vq; i++)
              {
                complex<realtype> c=da*complex<realtype>(RAN(),RAN());
                Sigmaq(i,m1,m2)=c;
              }
          }
    }


  MakeHermitian(Sigmaq,false); // no warnings

#ifdef FORCEINVERSIONSYMMETRY
```

```
  MakeInversionTransposedSymmetric(Sigmaq);
  /*
  FFTWEXECUTE(A2q_to_A2r);
  MakeReal(Sigmar);
  FFTWEXECUTE(A2r_to_A2q);
  Sigmaq*=invVq;  // do not change magnitude
  */
#endif

#ifdef PRESERVESYMMETRY
  MakeSymmetric(Sigmaq);
#endif

  MakeHermitian(Sigmaq);

  MakeSpinSymmetric(Sigmaq,false);

  if(TRACE) SanityCheck(Sigmaq,"Sigmaq, at end of MakeRandomSigma");
  if(TRACE) cout << "Finished MakeRandomSigma()" << endl;
}


void Driver::SetQsToZero()
{
  logfile << "Setting the following qpts to zero in the self-energy" << endl;

  bool found=false;
  ifstream ifile("qstozero.in");

  while(ifile)
    {
      int qindx;
      ifile >> qindx;
      if(qindx >=0 && qindx <Vq && ifile)
        {
          found=true;
          Coord thisq=la.qPos(qindx);
          logfile << "(" << thisq << ")" << endl;
          Sigmaq(qindx,0,0)=complex<realtype>(0.,0.);
        }
    }
  if(!found) logfile << "None" << endl;
}



// assumes initialized Sigma and mu
void Driver::SolveSelfConsistentEquation(vector<realtype> Delta)
{
  if(TRACE) cout << "Starting SolveSelfConsistentEquation " << endl;

  lineid++;


#ifdef RANDOMINITIALIZATION
  MakeRandomSigma();
#else
  rule.InitializeSigma(Sigmaq); // Get initial values of Sigmaq
#endif
  SetQsToZero();

#ifdef PRESERVESYMMETRY
  MakeSymmetric(Sigmaq);
#endif

#ifdef BIASQS  // set some q's to zero in

#endif
```

```
  Chomp(Jq); // set very small entries to 0

  if(TRACE) SanityCheck(Jq,"Jq, input to SolveSelfConsistentEquation");


  //  if(TRACE) cout << "Min eigenvalue: " << FindMinimumEigenvalue(Jq) << endl;

  SubtractMinimumEigenvalue(Jq);

  Chomp(Jq); // set very small entries to 0

  if(TRACE) SanityCheck(Jq,"Jq, after subtracting minimum");

  /*
#ifdef PRINTTCONVERGENCE
  ostringstream sstrs; sstrs << "DELTA" << Delta << ".s.dat";
  ostringstream tstrs; tstrs << "DELTA" << Delta << ".t.dat";
  ostringstream ustrs; ustrs << "DELTA" << Delta << ".u.dat";
  string tfilename = tstrs.str();
  string sfilename = sstrs.str();
  string ufilename = ustrs.str();
  ofstream tfile(tfilename.c_str());
  ofstream sfile(sfilename.c_str());
  ofstream ufile(ufilename.c_str());
#endif
  */

  // Put together K


  realtype newT=-1.;
  realtype m2=0.; // magnetic order parameter squared.
  vector<obstype> nobs(NOBSERVABLES); // nematic order parameters
  vector<obstype> nspinobs(NSPINOBSERVABLES); // different types of spin order parameters

  //  vector<obstype> nalphas(NOBSERVABLES); // alpha


  // construct Kinvq:

  ConstructKinvq();

  realtype oldT=CalculateT(0);

  currT=oldT; // set the current operating temperature

  vector<realtype> Ts(NSUBL); // A list of Ts

  if(TRACE)
    {
      cout << "Initial temperatures: " << endl;
      CalculateTs(Ts); // calculate all the temperatures

      streamsize ss=cout.precision();
      cout.precision(17);
      for(int i=0; i<NSUBL; i++)
        cout << Ts[i] << " ";
      cout << endl;
      cout.precision(ss); // restore precision
    }




  int iter=0;
  bool converged=false;
  bool pconverged=true; // convergence in the previous iteration,
```

```
  bool done=false;

  while(iter< par[MAXITER] && !done)
    {
      if(TRACE) cout << "New iteration: " << iter << endl;
      iter++;
      if(converged) pconverged=true; // record if the previous iteration had converged

      if(TRACE) SanityCheck(Kinvq,"Kinvq, at start of new iteration");


      ComputeSelfEnergy(); // careful with this, it overwrites K

#ifdef NOSELFENERGY
      for(int i=0; i<Vtot; i++){Sigmaq[i]=0.;}
#endif

      if(TRACE) SanityCheck(Sigmaq,"Sigmaq, after ComputeSelfEnergy");

      ConstructKinvq();

      if(TRACE) SanityCheck(Kinvq,"Kinvq, in iteration loop after making sigmaq");

      // convergence checks
      newT=CalculateT(0);

#ifdef PHONONS
      vector<realtype> epsoverT=CalculateEpsilonsOverT();
#endif

      if(TRACE)
        {
          cout << "iteration: " << iter << " T= " << newT << " oldT=" << oldT
               << " dev: " << fabs((newT-oldT)/oldT);
#ifdef PHONONS
          cout << " epsilon/T= ";
          for(int i=0; i<NELASTIC; i++) cout << epsoverT[i] << " ";
#endif
          cout << endl;
        }

      // write progress to logfile
      if(PRINTPROGRESS && iter%PRINTPROGRESSTICKLER==0)
        {
          logfile.precision(17);
          logfile << "iteration: " << iter << " T= " << newT << " oldT=" << oldT << " dev:
" << fabs((newT-oldT)/oldT);
#ifdef PHONONS
          logfile << " epsilon/T= ";
          for(int i=0; i<NELASTIC; i++) logfile << epsoverT[i] << " ";
#endif
          logfile << endl;
        }

      if( fabs((newT-oldT)/oldT) < par[TOLERANCE]) converged=true;

      const realtype inertia=0.5; // how much to resist changes: [0,1]

      currT= (1.-inertia)*newT+inertia*oldT;

      oldT=currT;

#ifdef PHONONS
      for(int i=0; i<NELASTIC; i++){epsilon[i]= (1.-inertia)*currT*epsoverT[i]+inertia*epsi
lon[i];}
#endif

      if(TRACE) cout << "converged= " << converged << " TOLERANCE:" << par[TOLERANCE] << en
```

```
dl;

      if(converged && pconverged){ done=true; continue;} // two iterations must fulfill con
v. crit.
    }

  if(TRACE) cout << "Final Kinv_q: " << Kinvq << endl;

  //  nalphas=CalculateAlphas(newT); // calculate alphas

  m2=NS*newT/(2.*Delta[0]*Vq); // calculate magnetic moment

  CalculateTs(Ts); // calculate the final temperatures

  // Print final Ts and epsilons
  logfile << "iteration: " << iter << " Ts: ";
  streamsize ss=logfile.precision();
  logfile.precision(17);
  for(int i=0; i<NSUBL; i++){logfile << Ts[i] << " ";}
#ifdef PHONONS
  logfile << " epsilon: ";
  for(int i=0; i<NELASTIC; i++) logfile << epsilon[i] << " ";
#endif
  logfile << " converged: " << (converged ? "true": "false") << endl;
  logfile.precision(ss);


#ifdef PRINTTCONVERGENCE
      tfile << setprecision(17) << newT << endl;
      sfile << setprecision(17) << nobs[2].real() << endl;
#endif


  if(converged) logfile << "Convergence reached after " << iter << " steps." << endl;

  //Use MAXITER as convergence criterion itself
  if(par[TOLERANCE]==0.)
    {
      logfile << "Exiting after MAXITER steps, using end result" << endl;
      converged=true;
    }


  if(!converged)
    {
      logfile << "reached MAXITER=" << par[MAXITER] << " iterations without converging, inc
rease MAXITER!" << endl;
    }
  else
    {

      //      lineid++;



      //      if(Printinfo)
        {

          /*
          vector<record> lv=FindLowestValues(K1,nvals);

          logfile << "largest values" << endl;
          for(int i=0; i<nvals; i++)
            {
              const int p=lv[i].pos;
              Coord q=lattice.qPos(p);
              Coord qoverpi=(1./PI)*q;
              logfile << q << " (" << qoverpi << ")  : " << 1./lv[i].value << endl;
```

```
            }


          logfile << "Printing info to " << MAXQNAME;
          ofstream maxq(MAXQNAME.c_str(),ios::app);
          Coord maximumq=lattice.qPos(lv[0].pos);
          maxq << setprecision(16) << newT << " " << maximumq << " " << endl;
          */


#ifdef REDUCEDOUTPUT

          if(PRINTQCORRS)
            {
              stringstream ss;
              ss << QCORRSFILENAME << "_" << lineid << ".dat";

              logfile << ss.str();

              //              ofstream qcorrfile(QCORRSFILENAME.c_str(),ios::app);
              ofstream qcorrfile(ss.str().c_str());

              realtype factor=newT*NS*0.5;
              complex<realtype>* start=Kinvq(0,0);

              if(BINARYOUTFILES)
                {
                  qcorrfile.write((char*) &lineid,sizeof(lineid)); // general format
                  qcorrfile.write((char*) &la.nindx_q,sizeof(la.nindx_q));
                  for(int i=0; i<la.nindx_q; i++)
                    {
                      complex<realtype> value=factor*start[la.indx_site_q[i]];
                      qcorrfile.write((char*) &value,sizeof(value));
                    }
                }
              else
                {
                  //              qcorrfile << setprecision(16) << lineid << " ";
                  for(int i=0; i<la.nindx_q; i++)
                    {
                      complex<realtype> value=factor*start[la.indx_site_q[i]];
                      qcorrfile << real(value) << " " << imag(value) << endl;;}
                }
              qcorrfile.close();
            }
#endif
          /*
          if(PRINTSIGMAE)
            {
              logfile << ", " << SIGMAEFILENAME;
              ofstream sigmaefile(SIGMAEFILENAME.c_str(),ios::app);

              realtype min=SubtractMinimum(Sigma);

              if(BINARYOUTFILES)
                {
                  sigmaefile.write((char*) &lineid,sizeof(lineid)); // general format
                  sigmaefile.write((char*) &lattice.nindx_q,sizeof(lattice.nindx_q));
                  for(int i=0; i<lattice.nindx_q; i++)
                    {
                      sigmaefile.write((char*) &Sigma[lattice.indx_site_q[i]],sizeof(Sigma[
0]));
                    }
                }
              else
                {
                  sigmaefile << setprecision(16) << lineid << " ";
                  for(int i=0; i<lattice.nindx_q; i++){sigmaefile << Sigma[lattice.indx_sit
```

```
e_q[i]] << " ";}
                  sigmaefile << endl;
                }
              sigmaefile.close();
            }

          if(PRINTRCORRS)
            {
              logfile << ", " << RCORRSFILENAME;
              vector<realtype> rcorr(Vf);
              RealSpaceCorrelationFunction(K1,newT,NS,dim,dims,rcorr);
              ofstream rcorrfile(RCORRSFILENAME.c_str(),ios::app);
              if(BINARYOUTFILES)
                {
                  rcorrfile.write((char*) &lineid,sizeof(lineid)); // general format
                  rcorrfile.write((char*) &lattice.nindx_r,sizeof(lattice.nindx_r));
                  for(int i=0; i<lattice.nindx_r; i++)
                    {
                      rcorrfile.write((char*) &rcorr[lattice.indx_site_r[i]],sizeof(rcorr[0
]));
                    }
                }
              else
                {
                  rcorrfile << setprecision(16) << lineid << " ";
                  for(int i=0; i<lattice.nindx_r; i++){rcorrfile << rcorr[lattice.indx_site
_r[i]] << " ";}
                }
              rcorrfile.close();

              ofstream mostremotefile(RCORRMOSTREMOTE.c_str(),ios::app);
              mostremotefile << setprecision(16) << newT << " " << rcorr[lattice.indx_most_
remote] << endl;
              mostremotefile.close();

            }
#else
      // Full output
          if(PRINTQCORRS)
            {
              logfile << ", " << QCORRSFILENAME;
              ofstream qcorrfile(QCORRSFILENAME.c_str(),ios::app);
              realtype factor=newT*NS*0.5;
              vector<realtype> qcorr(Vq);
              //              for(int i=0; i<Vq; i++) qcorr[i]=factor/K1[i];

              if(BINARYOUTFILES)
                {
                  qcorrfile.write((char*) &lineid,sizeof(lineid)); // general format
                  qcorrfile.write((char*) &Vq,sizeof(Vq));
                  qcorrfile.write((char*) &qcorr[0],Vq*sizeof(qcorr[0]));
                }
              else
                {
                  qcorrfile << setprecision(16) << lineid << " ";
                  for(int i=0; i<Vq; i++){ qcorrfile << qcorr[i] << " ";}
                  qcorrfile << endl;
                }
              qcorrfile.close();
            }

          if(PRINTSIGMAE)
            {
              logfile << ", " << SIGMAEFILENAME;
              ofstream sigmaefile(SIGMAEFILENAME.c_str(),ios::app);

              //              vector<realtype> min=SubtractMinimum(Sigma);
```

```
              if(BINARYOUTFILES)
                {
                  sigmaefile.write((char*) &lineid,sizeof(lineid)); // general format
                  sigmaefile.write((char*) &Vq,sizeof(Vq));
                  sigmaefile.write((char*) &Sigma[0],Vq*sizeof(Sigma[0]));
                }
              else
                {
                  sigmaefile << setprecision(16) << lineid << " ";
                  for(int i=0; i<Vq; i++){ sigmaefile << Sigma[i] << " ";}
                  sigmaefile << endl;
                }
              sigmaefile.close();
            }


          if(PRINTRCORRS)
            {
              logfile << ", " << RCORRSFILENAME;
              vector<realtype> rcorr(Vf);
              //              RealSpaceCorrelationFunction(K1,newT,NS,dim,dims,rcorr);
              ofstream rcorrfile(RCORRSFILENAME.c_str(),ios::app);
              if(BINARYOUTFILES)
                {
                  rcorrfile.write((char*) &lineid,sizeof(lineid)); // general format
                  rcorrfile.write((char*) &Vf,sizeof(Vf));
                  rcorrfile.write((char*) &rcorr[0],Vf*sizeof(rcorr[0]));
                }
              else
                {
                  rcorrfile << setprecision(16) << lineid << " ";
                  for(int i=0; i<Vf; i++){ rcorrfile << rcorr[i] << " ";}
                  rcorrfile << endl;
                }
              rcorrfile.close();

              ofstream mostremotefile(RCORRMOSTREMOTE.c_str(),ios::app);
              mostremotefile << setprecision(16) << newT << " " << rcorr[lattice.indx_most_
remote] << endl;
              mostremotefile.close();


            }
#endif // REDUCEDOUTPUT
          logfile << endl;
          */
        }



      for(int s=0; s<NSUBL; s++)
        {
          stringstream ss;
          ss << "td" << "_" << s << ".dat";

          ofstream outfile_a(ss.str().c_str(),ios::app);
          outfile_a << setprecision(16) << Ts[s] << " " << Delta[s] << endl;
          outfile_a.close();

          ss.str("");
          ss << "dt" << "_" << s << ".dat";

          ofstream outfile_b(ss.str().c_str(),ios::app);
          outfile_b << setprecision(16) << Delta[s] << " " << Ts[s] << endl;
          outfile_b.close();
        }

#ifdef PHONONS
```

```
      for(int i=0; i<NELASTIC; i++)
        {
          stringstream ss;
          ss << "teps" << "_" << i << ".dat";
          ofstream outfile_a(ss.str().c_str(),ios::app);
          outfile_a << setprecision(16) << Ts[0] << " " << epsilon[i] << endl;
          outfile_a.close();
        }
#endif

      /*

      for(int s1=0; s1<NSUBL; s1++)
        for(int s2=0; s2<NSUBL; s2++)
          {
            nobs=CalculateOrderPars(newT,s1,s2); // calculate order pars.

            stringstream ss;

            for(int j=0; j<NOBSERVABLES; j++)
              {
                ss.str("");
                ss << NAMESOFOBSERVABLES[j] << "_" << s1 << s2 << ".dat";

                ofstream outfile_a(ss.str().c_str(),ios::app);
                outfile_a << setprecision(16) << newT << " "
                          << real(nobs[j]) << " " << imag(nobs[j]) << endl;
                outfile_a.close();

                ss.str("");
                ss << NAMESOFOBSERVABLES[j] << "1_" << s1 << s2 << ".dat";

                ofstream outfile_b(ss.str().c_str(),ios::app);
                outfile_b << setprecision(16) << newT << " "
                          << abs(nobs[j]) << endl;
                outfile_b.close();

                ss.str("");
                ss << NAMESOFOBSERVABLES[j] << "2_" << s1 << s2 << ".dat";

                ofstream outfile_c(ss.str().c_str(),ios::app);
                outfile_c << setprecision(16) << newT << " "
                          << norm(nobs[j]) << endl;
                outfile_c.close();
              }
          }

      */

      {
        nspinobs=CalculateSpinOrderPars(newT); // calculate order pars.

        stringstream ss;

        for(int j=0; j<NSPINOBSERVABLES; j++)
          {
            ss.str("");
            ss << NAMESOFSPINOBSERVABLES[j] << ".dat";

            ofstream outfile_a(ss.str().c_str(),ios::app);
            outfile_a << setprecision(16) << newT << " "
                      << real(nspinobs[j]) << " " << imag(nspinobs[j]) << endl;
            outfile_a.close();

            ss.str("");
            ss << NAMESOFSPINOBSERVABLES[j] << ".abs.dat";

            ofstream outfile_b(ss.str().c_str(),ios::app);
```

```
            outfile_b << setprecision(16) << newT << " "
                      << abs(nspinobs[j]) << endl;
            outfile_b.close();

            ss.str("");
            ss << NAMESOFSPINOBSERVABLES[j] << ".norm.dat";

            ofstream outfile_c(ss.str().c_str(),ios::app);
            outfile_c << setprecision(16) << newT << " "
                      << norm(nspinobs[j]) << endl;
            outfile_c.close();
          }
      }




      logfile << "Magnetic moment: " << m2 << endl;

      stringstream ss;
      ss << "m2" << ".dat";

      ofstream outfilem(ss.str().c_str(),ios::app);
      outfilem << setprecision(16) << newT << " " << m2 << endl;
      outfilem.close();


      realtype f=CalculateFreeEnergy(newT);
      logfile << "Free energy: " << f << endl;

      ss.str("");
      ss << "tf" << ".dat";

      ofstream outfile(ss.str().c_str(),ios::app);
      outfile << setprecision(16) << newT << " " << f << endl;
      outfile.close();

      ss.str("");
      ss << "df" << ".dat";

      ofstream outfile2(ss.str().c_str(),ios::app);
      outfile2 << setprecision(16) << Delta[0] << " " << f << endl;
      outfile2.close();
    }

  if(TRACE) cout << "Done SolveSelfConsistentEquation " << endl;
}




class Simulation{
  friend ostream& operator<<(ostream& os,Simulation& s){
    os << endl; return os;}
 public:
  Simulation();
  void Run();
 private:
  Couplings couplings;
  Rule rule;

  Driver mysolver;
  vector<NumberList> Deltalist;
  vector<NumberList> Deltastoshowlist;
  vector<bool> Printinfolist;
};
```

```
Simulation::Simulation(): couplings(par,NC,NMAT),rule(couplings),mysolver(rule),Deltalist(0
),Deltastoshowlist(0),Printinfolist(0)
{
  if(TRACE) cout << "Initializing Simulation" << endl;

  ifstream parameterfile(PARAMETERFILENAME.c_str());
  if(!parameterfile)
    {
      if(TRACE)
        cout << "No file " << PARAMETERFILENAME << " found."
             << " Using Delta=" << par[DELTA] << endl;
      logfile << "No file " << PARAMETERFILENAME << " found."
              << " Using Delta=" << par[DELTA] << endl;

      NumberList myval(NSUBL,par[DELTA]);
      Deltalist.push_back(myval);
      Printinfolist.push_back(true);
    }
  else
    {
      if(TRACE) cout << "Reading " << PARAMETERFILENAME << " from disk" << endl;
      logfile << "Reading " << PARAMETERFILENAME << " from disk" << endl;

      string line;
      while (getline(parameterfile, line))
        {
          istringstream iss(line);
          realtype newval;
          if(!(iss >> newval)){ break;}

          NumberList newDelta(NSUBL,newval); // initialize with one value for all

          for(int i=1; i<NSUBL; i++)
            {
              if(!(iss >> newval)){ break;}
              newDelta.v[i]=newval;
            }
          Deltalist.push_back(newDelta);
          Printinfolist.push_back(false);
        }
    }
  if(TRACE) cout << "Deltalist has " << Deltalist.size() << " entries" << endl;
  logfile << "Deltalist has " << Deltalist.size() << " entries" << endl;


  ifstream parameterfile2(DELTASTOSHOWFILENAME.c_str());
  if(!parameterfile)
    {
      if(TRACE)
        cout << "No file " << DELTASTOSHOWFILENAME << " found."
             << " Using Delta=" << par[DELTA] << endl;
      logfile << "No file " << DELTASTOSHOWFILENAME << " found."
              << " Using Delta=" << par[DELTA] << endl;

      NumberList myval(NSUBL,par[DELTA]);
      Deltastoshowlist.push_back(myval);
    }
  else
    {
      if(TRACE) cout << "Reading " << DELTASTOSHOWFILENAME << " from disk" << endl;
      logfile << "Reading " << DELTASTOSHOWFILENAME << " from disk" << endl;

      string line;
      while (getline(parameterfile, line))
        {
          istringstream iss(line);
```

```
          realtype newval;
          if(!(iss >> newval)){ break;}

          NumberList newDelta(NSUBL,newval); // initialize with one value for all

          for(int i=1; i<NSUBL; i++)
            {
              if(!(iss >> newval)){ break;}
              newDelta.v[i]=newval;
            }
          Deltastoshowlist.push_back(newval);
        }

    }
  if(TRACE) cout << "Deltastoshowlist has " << Deltastoshowlist.size() << " entries" << end
l;
  logfile << "Deltastoshowlist has " << Deltastoshowlist.size() << " entries" << endl;


  //Search Deltastoshowlist and mark if it is present in Deltalist
  for(unsigned int j=0; j<Deltastoshowlist.size(); j++)
    {
      NumberList d=Deltastoshowlist[j];
      bool found=false;
      unsigned int indx=0;
      while( indx<Deltalist.size() && !found)
        {
          if(TRACE) cout  << indx << " d=" << d << " Deltalist " << Deltalist[indx] << endl
;
          if(Deltalist[indx]==d){found=true;}
          else{indx++;}
        }
      if(found){Printinfolist[indx]=true;}
    }

  if(TRACE)
    {
      cout << "Printinfolist" << endl;
      for(unsigned int i=0; i<Printinfolist.size(); i++)
        {
          cout << i << " " << Deltalist[i] << " " << "info: " << Printinfolist[i] << endl;
        }
    }

  if(TRACE) cout << "Done Initializing Simulation" << endl;
}


void Simulation::Run()
{
  if(TRACE) cout << "Starting Run" << endl;
  for(unsigned int i=0; i< Deltalist.size(); i++)
    {
      mysolver.Solve(Deltalist[i],Printinfolist[i]);
    }
  if(TRACE) cout << "Done Run" << endl;
}



#endif //BOND_H
```