```
#ifndef ALGOHELPER_H
#define ALGOHELPER_H

template<typename T>
class calcsum
{
  private:
    T sum;
  public:
    calcsum() : sum(0) {}
    calcsum(const calcsum &csum) : sum(csum.sum) {}

    void operator()(const T& x) { sum += x; }
    T result() { return sum; }
};

#endif
```

```
/** @file boundary.h
 * @brief Interface for wrapping and exchanging boundaries
 *
 *  This interface is used to exchange the boundaries of distribution
 *  functions and scalar fields. It can be implemented to define
 *  periodic boundaries or exchange data with other processes.
 *
 *  See the page @ref indices for a discussion on the numerical
 *  ranges of the fields
 */
//---------------------------------------------------------------------------
#ifndef BOUNDARY_H
#define BOUNDARY_H

#include "mpulse.h"
#include "globals.h"
#include "rebuild.h"

//---------------------------------------------------------------------------
//Boundary

/** @brief Interface for wrapping and exchanging boundaries .
 *
 *  This interface is used to exchange the boundaries of distribution
 *  functions and scalar fields. It can be implemented to define
 *  periodic boundaries or exchange data with other processes.
 *  This is the (abstract) base class of all boundary classes.
 *
 *  See the page @ref indices for a discussion on the numerical
 *  ranges of the fields
 */
class Boundary : public Rebuildable {
  public:
    /// Default constructor,
    Boundary() {}
    /** @brief Virtual destructor
     *
     *  We need a virtual destructor because the class has
     *  virtual methods
     */
    virtual ~Boundary() {}

    /** Initialize the boundary
     */
    virtual void init() = 0;

    /** @brief Exchange the boundaries of a field function
     *  in the x-direction
     */
    virtual void exchangeX(DataGrid3d &field) = 0;

    /** @brief Exchange the boundaries of a field function
     *  in the y-direction
     */
    virtual void exchangeY(DataGrid3d &field) = 0;

    /** @brief Exchange the boundaries of a field function
     *  in the z-direction
     */
    virtual void exchangeZ(DataGrid3d &field) = 0;

    /// Return the average of a single value over all the processes
    virtual double AvgReduce(double) const = 0;

    /// Return the average of a single value over all the processes
    virtual double SumReduce(double) const = 0;
```

```
    /// Return the maximum of a single value over all the processes
    virtual double MaxReduce(double) const = 0;

    /// Return the lower bound of the field
    virtual const GridIndex &RegionLow() const = 0;

    /// Return the upper bound of the field
    virtual const GridIndex &RegionHigh() const = 0;

    /// Return true if this is the master process and false otherwise
    virtual bool master() const = 0;

    /// Return the process number
    virtual int procnum() const = 0;

    ///get a unique Id
    virtual int getUniqueId() const = 0;
};

//Boundary


#endif //BOUNDARY_H
```

```
#ifndef MPULSE_CPML_BORDER_H
#define MPULSE_CPML_BORDER_H

#include "mpulse.h"
#include "rebuild.h"
#include "currentsim.h"
#include "currents.h"

class Storage;

class CPMLBorder : public CurrentSim
{
  public:
    virtual ~CPMLBorder() {}
    void initCurrents(Storage *storage, FieldSolver *solver);
  protected:
    ParameterMap* MakeParamMap (ParameterMap* pm = NULL);
  private:

    void initCoefficients(Storage *storage);

    int thickness;
    double kappaMax;
    double aMax;
    double sigmaMax;
};

class CPMLBorderCurrent : public Current
{
  public:
    CPMLBorderCurrent( int thickness_, Direction dir_, bool isH_,
                       double kappaMax_, double aMax_, double sigmaMax_, double eps_);


    virtual ~CPMLBorderCurrent() {}
  protected:
    bool reverse;
    int thickness;

    int dim;
    int transverse1, transverse2;

    Direction dir;
    bool isH;
    int lowOffset;
    int highOffset;

    int zerolayer;

    double kappaMax;
    double aMax;
    double sigmaMax;
    double eps;

    DataLine bCoeff;
    DataLine cCoeff;


    void makeCoeff(Storage *storage);
};

class CPMLBorderECurrent : public CPMLBorderCurrent
{
  public:
    CPMLBorderECurrent( int thickness_, Direction dir_,
                        double kappaMax_, double aMax_, double sigmaMax_, double eps_);
```

```
    virtual ~CPMLBorderECurrent() {}

    void initStorage(Storage *storage);

    void stepSchemeInit(double dt);
    void stepScheme(double dt);
  protected:

    DataGrid *pB[3];
    DataGrid *pPsi[2];
    double dx;
};

class CPMLBorderHCurrent : public CPMLBorderCurrent
{
  public:
    CPMLBorderHCurrent( int thickness_, Direction dir_,
                        double kappaMax_, double aMax_, double sigmaMax_, double eps_);

    virtual ~CPMLBorderHCurrent() {}

    void initStorage(Storage *storage);

    void stepSchemeInit(double dt);
    void stepScheme(double dt);
  protected:
    DataGrid *pE[3];
    DataGrid *pPsi[2];
    double dx;
};


#endif
```

```
#ifndef MPULSE_CURRENTS_H
#define MPULSE_CURRENTS_H

#include "mpulse.h"
#include "rebuild.h"

class Storage;

class Current
{
  protected:
    DataGrid *pJx;
    DataGrid *pJy;
    DataGrid *pJz;
  public:
    virtual ~Current() {}
    virtual void initStorage(Storage *storage_) = 0;

    virtual void stepSchemeInit(double dt) = 0;
    virtual void stepScheme(double dt) = 0;

    bool isValid() const { return (pJx!=0) && (pJy!=0) && (pJz!=0); }

    const DataGrid *getJx() { return pJx; }
    const DataGrid *getJy() { return pJy; }
    const DataGrid *getJz() { return pJz; }

    struct CurrentInvalidPredicate
    {
      bool operator()(const Current *bc) { return !(bc->isValid()); }
    };
};

#endif
```

```
#ifndef MPULSE_CURRENTSIM_H
#define MPULSE_CURRENTSIM_H

#include "mpulse.h"
#include "rebuild.h"

class Storage;
class FieldSolver;

class CurrentSim : public Rebuildable
{
  public:
    virtual ~CurrentSim() {}
    virtual void initCurrents(Storage *storage_, FieldSolver *solver) = 0;

/*    struct InitCurrentSimFunctor
    {
      Storage *storage;
      FieldSolver *solver;
      InitStorageFunctor(Storage *storage_, FieldSolver *solver_)
        : storage(storage_), solver(solver_) {}
      void operator()(CurrentSim *sim) { sim->initCurrents(storage, solver); }
    };
*/
};

#endif
```

```
#ifndef MPULSE_DIAGNOSTIC_H
#define MPULSE_DIAGNOSTIC_H

#include <iostream>
#include <list>
#include <string>
#include "stlpwrapper.h"
#include "rebuild.h"
//-------------------------------------------------------------------------
/** @file diagnostic.h
 * @brief Interface for diagnostic tasks.
 *
 * This interface can be used to implement different types of diagnostics.
 * The simplest form is implemented as SimpleDiagnostic, which writes the fields
 * to an output stream of arbitrary type.
 * The DiagnosticInterface is closely related to the DiagnosticManager. When an
 * instance of the interface is created it will register itself with the
 * DiagnosticManager. This then takes the responsibility of calling the execute
 * method of the DiagnosticInterface.
 */

//-------------------------------------------------------------------------
//DiagnosticInterface
/** @brief Interface for diagnostic tasks.
 *
 * This interface can be used to implement different types of diagnostics.
 * The DiagnosticInterface is closely related to the DiagnosticManager. When an
 * instance of the interface is created it will register itself with the
 * DiagnosticManager. This then takes the responsibility of calling the execute
 * method of the DiagnosticInterface.
 * This the abstract base class for all Diagnostic Interface Types.
 */
class DiagnosticInterface : public Rebuildable {
  private:
    /// The file name into which to write
    std::string fname;

    /** A string specifying whether to append to the file or whether to
     * write into a new file at each turn.
     * Only the first letter is checked for equality to 'y'
     */
    std::string append;

    /// each interval timesteps the actual output is performed
    int interval;

    ///timestep
    int t;
  public:
          ///constructor
    DiagnosticInterface();

    ///execute, calls the open, write and close method to perform the output
    void execute();
  protected:
         ///prototype of the open method
    virtual void open(const std::string &)=0;

    ///prototype of the write method
    virtual void write()=0;

    ///prototype of the close method
    virtual void close()=0;

    ///returns "false" if not overwritten by the derived class
    virtual bool singleOut() { return false; }
```

```
    /** can be overwritten do perform some calculation on all
     *  processes, independent on singleOut
     */
    virtual void calculate() {}

    ///create the parameter map
    virtual ParameterMap* MakeParamMap (ParameterMap* pm = NULL);
  private:
          ///if "true" append the output to the stream, if "false" each  write to a new fil
e
    bool appending();

          ///returns the name of the output file in a string
    std::string parsedFileName();
};
//DiagnosticINterface
//---------------------------------------------------------------------------
///wrapped pointer to DignosticInterfaces
typedef PtrWrapper<DiagnosticInterface> pDiagnosticInterface;
///list of wrapped pointers to DiagnosticInterfaces which are registered in the manager
typedef std::list<pDiagnosticInterface> DiagList;
//---------------------------------------------------------------------------
//DiagnosticManager
///@todo meyers singleton?! see the .cpp for this idea
/** @brief The diagnostic manager class
  *
  * Every instance of DiagnosticInterface registers here.
  * DiagnosticManager then takes care of calling the execute members of each interface.
  */
class DiagnosticManager {
  private:
        //should be deleted if meyers singleton is put to work
        ///the one and only instance of diagnosticManager
      static DiagnosticManager *theManager;
        ///list of the registered interfaces (pointers to these)
      DiagList diags;
  public:
        ///returns a reference to this instance
      static DiagnosticManager& instance();
        ///add a interface to the list
      void addDiagnostic(DiagnosticInterface*);
        ///call execute for each item of the list
      void execute();
  private:
        ///default constructor, is private to enforce singleton behavior
      DiagnosticManager();
        ///copy constructor, is private to enforce singleton behavior
      DiagnosticManager(DiagnosticManager&);
};
//DiagnosticManager
//---------------------------------------------------------------------------
//SimpleDiagnsotic for writing to streams

/** @brief a simple diagnostic interface, derived from DiagnosticInterface
  *
  * This can be used for writing results to files or streams for further use.
  */
template<class Type, class StreamType>
class SimpleDiagnostic : public DiagnosticInterface {
  private:
          /// Type of field
    Type *field;

    /// Stream to write to
    StreamType output;
```

```
    ///"true" if diagnostic is performed globaly
    bool single_out;
  public:
          ///default constructor
    SimpleDiagnostic() { single_out=false; }

    ///destrcutor
    ~SimpleDiagnostic();

    ///set the pointer to the field
    void setField(Type*);
  protected:
          ///open a file stream,
    void open(const std::string &);

    ///write diagnostics
    void write();

    ///close the file
    void close();

    ///returns the single_out member
    bool singleOut() { return single_out; }
  public:
          ///set the single_out member
    void setSingleOut(bool single_out_) { single_out = single_out_; }
};
//SimpleDiagnostic
//---------------------------------------------------------------------------
// including the template members
#include "diagnostic.t"
//---------------------------------------------------------------------------
#endif //DIAGNOSTIC_H
```

```
#ifndef FACTOR_H
#define FACTOR_H

#include <vector>

void equalFactors
  (
    int number,
    int nfact,
    std::vector<int> &factors,
    std::vector<int> &weights
  );

#endif
```

```
#ifndef MPULSE_FDTD_DISP_H
#define MPULSE_FDTD_DISP_H

#include "fieldsolver.h"
#include "fielddiag.h"
#include "mpulse.h"
#include "currents.h"

class Storage;

class FDTD_Dispersion : public FieldSolver
{
  private:
    DataGrid *pEx;
    DataGrid *pEy;
    DataGrid *pEz;
    DataGrid *pBx;
    DataGrid *pBy;
    DataGrid *pBz;

    // Lorentz polarization for three poles
    DataGrid *pPx[3];
    DataGrid *pPy[3];
    DataGrid *pPz[3];

    // Lorentz polarization from the previous timestep
    DataGrid *pPxp[3];
    DataGrid *pPyp[3];
    DataGrid *pPzp[3];

    Storage *storage;

    typedef std::list<Current*> CurrentList;
    CurrentList currents;

  public:
    void initStorage(Storage *storage_);

    void stepSchemeInit(double dt);
    void stepScheme(double dt);
  protected:
    /// build parametermap
    ParameterMap* MakeParamMap (ParameterMap* pm = NULL);

  private:
    void stepD(double dt);
    void stepB(double dt);

    /// value of eps_0*eps_infty (note that mu_0 = 0)
    double eps;

    /// value of eps_0*chi^(3)
    double chi;

    /// value of Delta epsilon_p^2 for the three Lorentz poles
    double LEps2[3];
    /// value of omega_p^2 for the three Lorentz poles
    double LOm2[3];
};

#endif
```

```
#ifndef MPULSE_FDTD_NL_H
#define MPULSE_FDTD_NL_H

#include "fieldsolver.h"
#include "fielddiag.h"
#include "mpulse.h"

class Storage;

class FDTD_Nonlinear : public FieldSolver
{
  private:
    DataGrid *pEx;
    DataGrid *pEy;
    DataGrid *pEz;
    DataGrid *pBx;
    DataGrid *pBy;
    DataGrid *pBz;

    Storage *storage;
  public:
    void initStorage(Storage *storage_);

    void stepSchemeInit(double dt);
    void stepScheme(double dt);
  protected:
    /// build parametermap
    ParameterMap* MakeParamMap (ParameterMap* pm = NULL);

  private:
    void stepD(double dt);
    void stepB(double dt);

    /// value of eps_0*eps_infty (note that mu_0 = 0)
    double eps;

    /// value of eps_0*chi^(3)
    double chi;

};

#endif
```

```
#ifndef MPULSE_FDTD_PLAIN_H
#define MPULSE_FDTD_PLAIN_H

#include "fieldsolver.h"
#include "mpulse.h"

class Storage;

class FDTD_Plain : public FieldSolver
{
  private:
    DataGrid *pEx;
    DataGrid *pEy;
    DataGrid *pEz;
    DataGrid *pBx;
    DataGrid *pBy;
    DataGrid *pBz;

    Storage *storage;
  public:
    void initStorage(Storage *storage_);

    void stepSchemeInit(double dt);
    void stepScheme(double dt);

  private:
    void stepD(double dt);
    void stepB(double dt);

};

#endif
```

```
#ifndef MPULSE_FDTD_PLRC_H
#define MPULSE_FDTD_PLRC_H

#include "fieldsolver.h"
#include "fielddiag.h"
#include "mpulse.h"
#include "currents.h"
#include "currentsim.h"

#include <complex>

class Storage;

class FDTD_PLRCCore
{
  public:
    void coreInitStorage(Storage *storage_);
  protected:
    DataGrid *pEx;
    DataGrid *pEy;
    DataGrid *pEz;
    DataGrid *pBx;
    DataGrid *pBy;
    DataGrid *pBz;

    DataLine *pKappaEdx;
    DataLine *pKappaEdy;
    DataLine *pKappaEdz;

    DataLine *pKappaHdx;
    DataLine *pKappaHdy;
    DataLine *pKappaHdz;

    DataLine *pStretchEax;
    DataLine *pStretchEay;
    DataLine *pStretchEaz;

    DataLine *pStretchEbx;
    DataLine *pStretchEby;
    DataLine *pStretchEbz;

    DataLine *pStretchHax;
    DataLine *pStretchHay;
    DataLine *pStretchHaz;

    DataLine *pStretchHbx;
    DataLine *pStretchHby;
    DataLine *pStretchHbz;

    // Real part of the accumulator for three poles
    DataGrid *pPsiRx[3];
    DataGrid *pPsiRy[3];
    DataGrid *pPsiRz[3];

    // Imaginary part of the accumulator for three poles
    DataGrid *pPsiIx[3];
    DataGrid *pPsiIy[3];
    DataGrid *pPsiIz[3];

    Storage *storage;

    typedef std::list<Current*> CurrentList;
    CurrentList currents;
    CurrentList magCurrents;

    typedef std::list<CurrentSim*> CurrentSimList;
```

```
    CurrentSimList currentSims;

    struct PLRCData
    {
      std::complex<double> dchi0[3];
      std::complex<double> dxi0[3];
      std::complex<double> Crec[3];
      double sumChi0;
      double sumXi0;
    };

    PLRCData plrcData;

    /// value of eps_infty (note that eps_0 = mu_0 = 1)
    double eps;

    /// value of Delta epsilon_p for the three Lorentz poles
    double LEps[3];
    /// value of delta_p^2 for the three Lorentz poles
    double LDelta[3];
    /// value of omega_p^2 for the three Lorentz poles
    double LOm2[3];
  private:

    template<class StorageHolder>
    struct InitStorageFunctor
    {
      Storage *storage;
      InitStorageFunctor(Storage *storage_) : storage(storage_) {}
      void operator()(StorageHolder *holder) { holder->initStorage(storage); }
    };



};

class FDTD_PLRCLinCore : public FDTD_PLRCCore
{
  protected:
    void plrcStepD(double dt,
                   int i, int j, int k,
                   double dx, double dy, double dz,
                   double Jx, double Jy, double Jz);

    void plrcStepB(double dt,
                   int i, int j, int k,
                   double dx, double dy, double dz,
                   double Jx, double Jy, double Jz);

    ParameterMap* CustomParamMap (ParameterMap* pm = NULL);
};

class FDTD_PLRCNonlinCore : public FDTD_PLRCCore
{
  protected:
    void plrcStepD(double dt,
                   int i, int j, int k,
                   double dx, double dy, double dz,
                   double Jx, double Jy, double Jz);

    void plrcStepB(double dt,
                   int i, int j, int k,
                   double dx, double dy, double dz,
                   double Jx, double Jy, double Jz);

    ParameterMap* CustomParamMap (ParameterMap* pm = NULL);
```

```
    /// value of chi^(3)
    double chi;

};

template<class PLRCImplementation>
class FDTD_PLRCSolver : public FieldSolver, public PLRCImplementation
{
  public:
    void initStorage(Storage *storage_);
    void addCurrent(Current *current);
    void addMagCurrent(Current *current);
    void stepSchemeInit(double dt);
    void stepScheme(double dt);
  protected:
    typedef PLRCImplementation Implementation;
    /// build parametermap
    ParameterMap* MakeParamMap (ParameterMap* pm = NULL);

  private:
    DataGrid *pJx;
    DataGrid *pJy;
    DataGrid *pJz;

    DataGrid *pMx;
    DataGrid *pMy;
    DataGrid *pMz;

    void stepD(double dt);
    void stepB(double dt);

    void initAccumulator(double dt);
    void sumCurrents();
    void sumMagCurrents();

    typedef typename Implementation::CurrentList CurrentList;
    typedef typename Implementation::CurrentSimList CurrentSimList;

};

typedef FDTD_PLRCSolver<FDTD_PLRCLinCore> FDTD_PLRCLin;
typedef FDTD_PLRCSolver<FDTD_PLRCNonlinCore> FDTD_PLRCNonlin;

#include "fdtd_plrc.t"

#endif
```

```
#ifndef MPULSE_FIELDDIAG_H
#define MPULSE_FIELDDIAG_H

#include "mpulse.h"
#include "hdfstream.h"
#include "diagnostic.h"
#include "storage.h"

#include <fstream>

class FieldDiag : public SimpleDiagnostic<DataGrid,HDFostream>
{
  public:
    void fetchField(Storage &storage);
  private:
    std::string fieldId;
  protected:
    typedef SimpleDiagnostic<DataGrid,HDFostream> ParentType;
    ParameterMap* MakeParamMap (ParameterMap* pm = NULL);
};


class FieldExtraDiag : public DiagnosticInterface {
  public:
    virtual void setStorage(Storage *storage) = 0;
    virtual void init() {}
};

class FieldEnergyDiag : public FieldExtraDiag {
  private:
          /// Type of field
    Storage *storage;

    double energy;

    /// Stream to write to
    std::ofstream output;

  public:
          ///default constructor
    FieldEnergyDiag();

    ///destrcutor
    ~FieldEnergyDiag();

    ///set the pointer to the field
    void setStorage(Storage *storage);
  protected:
          ///open a file stream,
    void open(const std::string &);

    ///write diagnostics
    void write();

    ///close the file
    void close();

    ///returns the single_out member
    bool singleOut() { return true; }

    /// calculate the field energy
    void calculate();
};


struct AllFieldDiag
```

```
{
  AllFieldDiag() {}
  typedef std::list<FieldDiag*> DiagList;
  typedef std::list<FieldExtraDiag*> ExtraDiagList;
  DiagList fields;
  ExtraDiagList fieldextras;
};

#endif
```

```
#ifndef MPULSE_FIELDSIM_H
#define MPULSE_FIELDSIM_H

#include "rebuild.h"
#include "init.h"
#include "storage.h"
#include "fielddiag.h"
#include "optfield.h"

class FieldSolver;

class FieldSimulation : public Storage, public Rebuildable
{
  private:
    double dt;

    typedef std::list<OptField*> OptFieldList;
    OptFieldList optfields;

  public:
    /** @brief Default constructor */
    FieldSimulation ();

    /** @brief Destructor, deletes boundary */
    ~FieldSimulation ();

    /** @brief Initializes the FieldSimulation
     */
    void init ();

    /** @brief Perform one timestep.
     */
    void execute();

// ------------- Rebuildable -------------

  protected:
    ///build parametermap, containing boundary, species and general parameters
    ParameterMap* MakeParamMap (ParameterMap* pm = NULL);

    AllFieldDiag fieldDiag;

    FieldSolver *solver;

    FieldSimInit *initializer;

};

#endif
```

```
#ifndef MPULSE_FIELDSOLVER_H
#define MPULSE_FIELDSOLVER_H

#include "rebuild.h"

class Storage;
class Current;

class FieldSolver : public Rebuildable
{
  public:
    virtual void initStorage(Storage *storage_) = 0;

    virtual void addCurrent(Current *current)
    {
      std::cerr << "================================================================
=\n";
      std::cerr << "============================    ERROR    ========================
=\n";
      std::cerr << "================================================================
=\n";
      std::cerr << "    Additional currents cannot be used when using this Field Solver
 \n;";
      std::cerr << "================================================================
=\n\n";
      exit(-1);
    }

    virtual void addMagCurrent(Current *current)
    {
      std::cerr << "================================================================
=\n";
      std::cerr << "============================    ERROR    ========================
=\n";
      std::cerr << "================================================================
=\n";
      std::cerr << "Additional magnetic currents cannot be used when using this Field Solve
r\n;";
      std::cerr << "================================================================
=\n\n";
      exit(-1);
    }

    virtual void stepSchemeInit(double dt) = 0;
    virtual void stepScheme(double dt) = 0;
};

#endif
```

```
#ifndef MPULSE_FREQDIAG_H
#define MPULSE_FREQDIAG_H

#include "mpulse.h"
#include "storage.h"
#include "fieldsim.h"

#include <fstream>
#include <string>

class FrequencyDiag : public FieldExtraDiag {
  private:
         /// Type of field
    Storage *storage;

    double frequency;
    int count;
    int lastcount;

    int x, y, z;
    double dt;

    double lastval;
    double firstzero;

    std::string field;

    /// Stream to write to
    std::ofstream output;

  public:
         ///default constructor
    FrequencyDiag();

    ///destrcutor
    ~FrequencyDiag();

    void init();

    ///set the pointer to the field
    void setStorage(Storage *storage);
  protected:
         ///open a file stream,
    void open(const std::string &);

    ///write diagnostics
    void write();

    ///close the file
    void close();

    ///returns the single_out member
    bool singleOut() { return true; }

    /// calculate the field energy
    void calculate();

    ParameterMap* MakeParamMap (ParameterMap* pm);
};

#endif
```

```
#ifndef MPULSE_GAUSSINIT_H
#define MPULSE_GAUSSINIT_H

#include "init.h"

class PlaneGaussPulseInit : public FieldSimInit
{
  public:
    void init(Storage &fields);
  protected:
    /// build parametermap
    ParameterMap* MakeParamMap (ParameterMap* pm = NULL);
  private:
    int kx, ky, kz;
    double bx, by, bz;
};

#endif
```

```
#ifndef MPULSE_GLOBALS_H
#define MPULSE_GLOBALS_H

#include "mpulse.h"
#include "rebuild.h"

#include <cmath>

/** @file globals.h
 * @brief global parameter class
 *
 * Defines a class for storing global Parameters
 */

/** A class for storing parameters, inherits Rebuildable.
 * Not all parameters might be needed for all types of simulation.
 */
class Globals : public Rebuildable {
  private:
    bool master;
    int uniqueId;
    /// global cell count x
    int GridX;
    /// global cell count y
    int GridY;
    /// global cell count z
    int GridZ;
    /// Size of a single grid cell in x-direction
    double GridDX;
    /// Size of a single grid cell in y-direction
    double GridDY;
    /// Size of a single grid cell in z-direction
    double GridDZ;
    /// time step
    double DT;
    /// total number of time steps
    int TotalTime;

    /// used for initilaizing the Fields
    GridIndex GridLow;
    /// used for initilaizing the Fields
    GridIndex GridHigh;

    ///will be set "true" if its a restart
    bool IsRestart;

    ///pointer to global parameters, will be set to "this" in constructor
    static Globals *globals;

    /// set to "true" if parameters class is initialized
    bool initialized;

    ///static member, number of the parameters main was called with
    static int Argc;
    /// static member, first argument main was called with
    static char **Argv;
  protected:
      ///Create the parameter map
    virtual ParameterMap* MakeParamMap (ParameterMap* pm = NULL);
  public:

    /// constructor, sets up a not initialized parameters object, restart is "false", sets
globals to "this"
    Globals();
    /// initialize and set initialized to "true"
    void init();
```

```
    /// accessor method, returns GridX
    int gridX() { return GridX; }
    /// accessor method, returns GridY
    int gridY() { return GridY; }
    /// accessor method, returns GridZ
    int gridZ() { return GridZ; }

    /// accessor method, returns grid spacing in x-direction
    double gridDX() { return GridDX; }
    /// accessor method, returns grid spacing in y-direction
    double gridDY() { return GridDY; }
    /// accessor method, returns grid spacing in z-direction
    double gridDZ() { return GridDZ; }

    /// accessor method, returns cell volume
    double volumeQuant() { return fabs(GridDX*GridDY*GridDZ); }

    /// accessor method, returns time step
    double dt() { return DT; }
    /// accessor method, returns total number of time steps
    int totalTime() { return TotalTime; }

    ///accessor method, returns isRestart
    bool isRestart() { return IsRestart; }
    ///sets isRestart to its argument
    void setRestart(bool IsRestart_) { IsRestart = IsRestart_; }

    ///accessor method, returns master
    bool isMaster() { return master; }
    ///sets master to its argument
    void setMaster(bool master_) { master = master_; }

    ///accessor method, returns master
    int getUniqueId() { return uniqueId; }
    ///sets master to its argument
    void setUniqueId(int uniqueId_) { uniqueId = uniqueId_; }

    ///accessor method, returns GridLow
    const GridIndex& gridLow() { return GridLow; }
    ///accessor method, returns GridHigh
    const GridIndex& gridHigh() { return GridHigh; }

    ///accessor method, eturns argc
    static int getArgc() { return Argc; }
    ///sets argc to its argument
    static void setArgc(int Argc_) { Argc = Argc_; }
    ///accessor method, returns argv
    static char **getArgv() { return Argv; }
    ///stes argv to its argument
    static void setArgv(char** Argv_) { Argv = Argv_; }

    ///if parameters is not initialized, initialize. in any case: return parameters
    static Globals &instance() {
      if ( !(globals->initialized) ) globals->init();
      return *globals;
    }

    ///rebuild
    std::string Rebuild(std::istream& in);
};


#endif // MPULSE_GLOBALS_H
```

```
#ifndef HDFSTREAM_H
#define HDFSTREAM_H
//---------------------------------------------------------------------------

#include <H5LT.h>

#include <schnek/matrix.h>

//---------------------------------------------------------------------------
  /** @file hdfstream.h
    * @brief IO class for HDF
    *
    * IO classes for handling HDF files and streams
    */
//---------------------------------------------------------------------------
//HDFstream
/** @brief IO class for handling HDF files
  *
  * This is the abstract base class for HDF-IO- classes.
  * Implements the basic operations on HDFstreams as virtual methods.
  */
class HDFstream {
  protected:
    /// HDF5 File id
    hid_t       file_id;

    /// HDF5 Error status
    herr_t      status;

    /// name of the datablock to be read or written
    std::string blockname;
    /// counter for the sets with a given blockname read from or written to the file
    int sets_count;

  public:
    /// constructor
    HDFstream();
    ///copy constructur
    HDFstream(const HDFstream&);
    /// destructor
    virtual ~HDFstream();

    /// open  file
    virtual int open(const char*)=0;

    /// close file
    virtual void close();

    /// return true=1 if data are still available
    virtual bool good() const;

    void setBlockName(std::string blockname_);
    /// assign
    HDFstream& operator = (const HDFstream&);
  protected:
    std::string getNextBlockName();
};

//HDFstream
//-------------------------------------------------------------
//HDFistream
/** @brief Input stream for HDF files */
class HDFistream : public HDFstream {
  public:
    /// constructor
    HDFistream();
```

```
    /// copy constructor */
    HDFistream(const HDFistream&);

    /// constructor, opens HDF file "fname", selects first dataset
    HDFistream(const char* fname);

    /// opens HDF file "fname", selects first dataset
    virtual int open(const char*);

    /// stream input operator for a schnek::Matrix
    template<typename TYPE, int RANK>
    HDFistream& operator>>(schnek::Matrix<TYPE, RANK>& m);
};
//HDFistream
//--------------------------------------------------------------------------
//HDFostream
/** @brief output stream for HDF files */
class HDFostream : public HDFstream {
  public:
    /// constructor
    HDFostream();

    /// copy constructor
    HDFostream(const HDFostream&);

    /// constructor, opens HDF file "fname"
    HDFostream(const char* fname);

    /// open file
    virtual int open(const char*);

    /// stream output operator for a matrix
    template<typename TYPE, int RANK>
    HDFostream& operator<< (const schnek::Matrix<TYPE, RANK>&);
};

template<typename TYPE>
struct H5DataType{
  static const hid_t type;
};



//HDFostream
//--------------------------------------------------------------------------

#include "hdfstream.t"

//--------------------------------------------------------------------------
#endif // HDFSTREAM_H
```

```
#ifndef MPULSE_INCSOURCE_H
#define MPULSE_INCSOURCE_H

#include "mpulse.h"
#include "rebuild.h"
#include "currentsim.h"
#include "currents.h"

class Storage;
class IncidentSourceCurrent;

class IncidentSource : public CurrentSim
{
  public:
    virtual ~IncidentSource() {}
    void initCurrents(Storage *storage, FieldSolver *solver);

  protected:
    virtual IncidentSourceCurrent *makeECurrent(int distance_, Direction dir_) = 0;
    virtual IncidentSourceCurrent *makeHCurrent(int distance_, Direction dir_) = 0;

    ParameterMap* MakeParamMap (ParameterMap* pm = NULL);
  private:

    int distance;
};

class IncidentSourceCurrent : public Current
{
  public:
    IncidentSourceCurrent(int distance_, Direction dir_, bool isH_);

    virtual ~IncidentSourceCurrent() {}
  protected:
    bool reverse;
    int distance;

    int dim;
    int transverse1, transverse2;

    Direction dir;
    bool isH;
    int lowOffset;
    int highOffset;
};

class IncidentSourceECurrent : public IncidentSourceCurrent
{
  public:
    IncidentSourceECurrent(int distance_, Direction dir_);

    virtual ~IncidentSourceECurrent() {}

    void initStorage(Storage *storage);
  protected:
    void setCurrents(GridIndex i, Vector H);

    DataGrid *pPsi[2];
    double dx;
};

class IncidentSourceHCurrent : public IncidentSourceCurrent
{
  public:
    IncidentSourceHCurrent(int distance_, Direction dir_);
```

```
    virtual ~IncidentSourceHCurrent() {}

    void initStorage(Storage *storage);
  protected:
    void setCurrents(GridIndex i, Vector E);

    DataGrid *pPsi[2];
    double dx;
};


#endif
```

```
#ifndef MPULSE_INIT_H
#define MPULSE_INIT_H

#include "rebuild.h"

class Storage;

class FieldSimInit : public Rebuildable
{
  public:
    virtual void init(Storage &fields) = 0;
};

#endif
```

```
#ifndef MPI_BOUND_H
#define MPI_BOUND_H

#include "boundary.h"

#ifndef SINGLE_PROCESSOR
#include <mpi.h>

//------------------------------------------------------------------------------
//MPIPeriodicSplitBoundary

/** @brief a boundary class for mutliple processor runs
  *
  * Is design to be exchanged via the MPI protocol
  * This implementation splits only the x-axis into rectangles.
  */
class MPIPeriodicSplitXBoundary : public Boundary {
  protected:
    /// The number of processes
    int ComSize;

    /// The rank of the current process
    int ComRank;

    /// The Comm object referring to the cartesian process grid
    MPI_Comm comm;

    /// The coordinates of this process
    int mycoord;

    int leftcoord;  ///< The rank of the left neighbour process
    int rightcoord; ///< The rank of the right  neighbour process

    /** @brief The size of the array that needs to be exchanged,
     *  when the exchangeX method is called
     */
    int exchSize;

    double *sendarr; ///< buffer holding the data to be send (size: exchSize)
    double *recvarr; ///< buffer holding the received data (size: exchSize)

    /// The size of the scalar fields when reducing
    int scalarSize;

    ///The position of the lower corner of the local piece of the grid
    GridIndex Low;
    ///<The position of the lower corner of the local piece of the dgrid
    GridIndex High;
  public:
    ///default constructor
    MPIPeriodicSplitXBoundary();

    /// Virtual destructor deleting all the allocated arrays
    ~MPIPeriodicSplitXBoundary();

    ///initialize
    void init();

    /** @brief Exchanges the boundaries in x-direction.
     *  The two outmost simulated cells are sent and the surrounding
     *  two ghost cells are filled with values
     */
    void exchangeX(DataGrid3d &field);

    /** @brief Exchanges the boundaries in y-direction.
     *  The two outmost simulated cells are sent and the surrounding
```

```
     *  two ghost cells are filled with values
     */
    void exchangeY(DataGrid3d &field);

    /** @brief Exchanges the boundaries in z-direction.
     *  The two outmost simulated cells are sent and the surrounding
     *  two ghost cells are filled with values
     */
    void exchangeZ(DataGrid3d &field);

    /** @brief Use MPIALLReduce to calculate the sum and then divide
     *  by the number of processes.
     */
    double AvgReduce(double val);

    /** @brief Use MPIALLReduce to calculate the maximum
     */
    double MaxReduce(double val);

    /// Returns the global lower bound of the distribution function
    const GridIndex &RegionLow()const;

    /// Returns the global upper bound of the distribution function
    const GridIndex &RegionHigh()const;

    /// The process with the rank zero is designated master process
    bool master() const { return ComRank==0; }

    /// Returns the comm rank as given by mpi
    int procnum() const { return ComRank; }

    ///returns an ID, which is identical with the coordinates
    int getUniqueId() const { return mycoord; }
};
//MPIPeriodicSplitXBoundary
//--------------------------------------------------------------------------
//MPIPeriodicSplitXYBoundary

/** @brief a boundary class for mutliple processor runs
 *
 * Is designed to be exchanged via the MPI protocol.
 * Here splitting is performed in both spatial directions.
 */
class MPIPeriodicSplitXYZBoundary : public Boundary {
  protected:
    /// The number of processes
    int ComSize;

    /// The rank of the current process
    int ComRank;

    /// The Comm object referring to the cartesian process grid
    MPI_Comm comm;


    int xprevcoord; ///< The rank of the left neighbour process in x
    int xnextcoord; ///< The rank of the right neighbour process in x

    int yprevcoord; ///< The rank of the left neighbour process in y
    int ynextcoord; ///< The rank of the right neighbour process in y

    int zprevcoord; ///< The rank of the left neighbour process in z
    int znextcoord; ///< The rank of the right neighbour process in z

    ///dimensions
    int dims[3];
```

```
    /// The cartesian coordinates of this process
    int mycoord[3];

    /** @brief The size of the array that needs to be exchanged,
     *  when the exchangeX or the exchangeY method is called
     */
    int exchSize[3];

    double *sendarrx; ///< send Buffer for exchanging data in x-direction (size: exchSize[0
])
    double *recvarrx; ///< receive Buffer for exchanging data in x-direction (size: exchSiz
e[0])

    double *sendarry; ///< send Buffer for exchanging data in y-direction (size: exchSize[1
])
    double *recvarry; ///< receive Buffer for exchanging data in y-direction (size: exchSiz
e[1])

    double *sendarrz; ///< send Buffer for exchanging data in z-direction (size: exchSize[2
])
    double *recvarrz; ///< receive Buffer for exchanging data in z-direction (size: exchSiz
e[2])

    /// The size of the scalar fields when reducing
    int scalarSize;

    /// The positions of the lower corner of the local piece of the grid
    GridIndex Low;
    ///<The positions of the upper corner of the local piece of the grid
    GridIndex High;

  public:
    ///default constructor
    MPIPeriodicSplitXYZBoundary();

    /// Virtual destructor deleting all the allocated arrays
    ~MPIPeriodicSplitXYZBoundary();

    ///initialize
    void init();

    /** @brief Exchanges the boundaries in x-direction.
     *
     *  The two outmost simulated cells are sent and the surrounding
     *  two ghost cells are filled with values
     */
    void exchangeX(DataGrid3d &field);

    /** @brief Exchanges the boundaries in y-direction.
     *
     *  The two outmost simulated cells are sent and the surrounding
     *  two ghost cells are filled with values
     */
    void exchangeY(DataGrid3d &field);

    /** @brief Exchanges the boundaries in z-direction.
     *
     *  The two outmost simulated cells are sent and the surrounding
     *  two ghost cells are filled with values
     */
    void exchangeZ(DataGrid3d &field);

    /// Use MPIALLReduce to calculate the sum and then divide by the number of processes.
    double AvgReduce(double val);

    /// Use MPIALLReduce to calculate the maximum
```

```
    double MaxReduce(double val);

    /// Returns the global lower bound of the distribution function
    const GridIndex &RegionLow()const;

    /// Returns the global upper bound of the distribution function
    const GridIndex &RegionHigh()const;

    /// The process with the rank zero is designated master process
    bool master() const { return ComRank==0; }

    /// Returns the comm rank as given by mpi
    int procnum() const { return ComRank; }
    ///returns an ID, which consists of the Dimensions and coordinates
    int getUniqueId() const {
      return dims[2]*(dims[1]*mycoord[0] + mycoord[1]) + mycoord[2];
    }
    /// returns "true" since this is periodic in the x-direction
    virtual bool periodicX() { return true; }
    /// returns "true" since this is periodic in the y-direction
    virtual bool periodicY() { return true; }
};
//MPIPeriodicSplitXYBoundary
//-------------------------------------------------------------------------
#endif // multiple processor

#endif
```

```
#ifndef MPULSE_MPULSE_H
#define MPULSE_MPULSE_H

#include <schnek/matrix.h>
#include <schnek/fixedarray.h>

typedef schnek::Matrix<double, 1> DataGrid1d;
typedef schnek::Matrix<double, 2> DataGrid2d;
typedef schnek::Matrix<double, 3> DataGrid3d;

typedef schnek::FixedArray<double, 3> Vector;

typedef DataGrid1d::IndexType GridIndex1d;
typedef DataGrid2d::IndexType GridIndex2d;
typedef DataGrid3d::IndexType GridIndex3d;

static const size_t Dimension = 3;
typedef DataGrid1d DataLine;
typedef DataGrid3d DataGrid;
typedef GridIndex3d GridIndex;

enum Direction {north, south, west, east, up, down};

#endif // MPULSE_MPULSE_H
```

```
#ifndef MPULSE_OPTFIELDS_H
#define MPULSE_OPTFIELDS_H

#include "rebuild.h"

class Storage;

class OptField : public Rebuildable
{
  public:
    virtual void initStorage(Storage *storage_) = 0;

    virtual void stepSchemeInit(double dt) = 0;
    virtual void stepScheme(double dt) = 0;
};

#endif
```

```
#ifndef MPULSE_PARAMETER_H
#define MPULSE_PARAMETER_H

/** @file parameter.h
 * @brief Classes reading input files, storing parameters
 *
 *  The classes in this file are used for reading parameters from
 *  an input file. The parameters are stored in a parameter map
 */
//---------------------------------------------------------------------------
#include <map>
#include <string>
#include <iostream>
#include "stlpwrapper.h"
//---------------------------------------------------------------------------
//forward declarations
class Rebuildable;
class Parameter;
//---------------------------------------------------------------------------
///wrapped pointer to parameter
typedef PtrWrapper<Parameter> WParameter;
/// map strings against wrapped pointers to parameter ojects (used by all derivations of of
 Rebuildable)
typedef std::map<std::string,WParameter> ParameterMap;
//---------------------------------------------------------------------------
//Parameter
/** @brief An abstract base class for parameters.
 *
 *  There is only one abstract method: Rebuild. This method has to rebuild
 *  the data from the input file and return the next token of the
 *  in a string.
 *  @todo Is the < operator really needed?
 */
class Parameter {
  public:
    /// Default constructor
    Parameter () {};
    /// Virtual destructor
    virtual ~Parameter () {};

    /** @brief This method has to rebuild the data
     *
     *  from the input file and return the next token of th input file
     *  in a string.
     *
     * This method needs to be overwritten by derived classes.
     */
    virtual std::string Rebuild (std::istream& in) = 0;

    /// Always returns true since sorting is irrelevant
    friend bool operator< (const Parameter& left, const Parameter& right) {
      return true;
    };
};

//---------------------------------------------------------------------------
//ParameterValue
/** @brief Single value parameter of some type.
 *
 *  This reads in a single value of some type TYPE from the input file
 *  and stores it under a given address. If the parameter is not encountered
 *  in the input, a default value is set.
 */
template<class TYPE>
class ParameterValue : public Parameter {
  private:
```

```
    TYPE* pValue;   ///< Pointer to the variable to set
    TYPE Default;   ///< A default value
  public:
    /// Constructor, takes a pointer to the variable and a default value
    ParameterValue (TYPE* _pValue, TYPE _Default) : pValue(_pValue) {
      SetDefault(_Default);
    };
    /// Virtual destructor
    virtual ~ParameterValue () {};

    /// Sets the default value for TYPE
    void SetDefault (TYPE _Default) {
      Default = _Default;
      *pValue = Default;
    };

    /// Reads the value for TYPE from the stream and returns the next token
    virtual std::string Rebuild(std::istream& in);
};

//---------------------------------------------------------------------------
//ParameterRebuild
/** @brief Base class for reading a subclass of the Rebuildable class from the input.
 *
 *  A Rebuildable class is read from the input file by invoking the creating the
 *  object and calling its Rebuildable::Rebuild method. The parameters of the newly
 *  built Rebuildable have to be enclosed in curly brackets.
 */
template<class Type, class BaseType>
class ParameterRebuild : public Parameter {
  public:
  protected:
    BaseType **value; ///< Pointer to the pointer of the object
    typedef std::list<BaseType*> BaseList;
    BaseList *values;
  public:
    /// Constructor takes a pointer to the parent Rebuildable
    ParameterRebuild (BaseType **value_)
      : value(value_),
        values(NULL)
    {
      *value = NULL;
    }

    /// Constructor takes a pointer to the parent Rebuildable
    ParameterRebuild (BaseList *values_)
      : value(NULL),
        values(values_) {}

    /// Destructor
    virtual ~ParameterRebuild () {}

    /** @brief Creates a new Rebuildable, calls its
     * Rebuildable::Rebuild method
     */
    virtual std::string Rebuild (std::istream& in);
    /// Abstract method that should return a pointer to a new task
    virtual BaseType* NewInstance ()  { return new Type; }
};

//---------------------------------------------------------------------------
//including the implementations for template members
#include "parameter.t"
//---------------------------------------------------------------------------
#endif // PARAMETER_H
```

```
#ifndef PERIODIC_BOUND_H
#define PERIODIC_BOUND_H

#include "boundary.h"

//-------------------------------------------------------------------------
//SinglePeriodicBoundary

/** @brief Implements Boudary to supply a periodic system running on
 *  a single processor
 */
class SinglePeriodicBoundary : public Boundary {
  public:
    /// No initialization needs to be done
    void init() {};
    /// Wraps the boundaries in x-direction
    void exchangeX(DataGrid3d &field);
    /// Wraps the boundaries in y-direction
    void exchangeY(DataGrid3d &field);
    /// Wraps the boundaries in z-direction
    void exchangeZ(DataGrid3d &field);

    /// There is no average to be calculated
    double AvgReduce(double val) const { return val; }
    /// There is no sum to be calculated
    double SumReduce(double val) const { return val; }
    /// There is no maximum to be calculated
    double MaxReduce(double val) const { return val; }
    /// Returns the global lower bound of the distribution function
    const GridIndex &RegionLow() const;
    /// Returns the global upper bound of the distribution function
    const GridIndex &RegionHigh() const;
    /// There is only one process, so master always returns true
    bool master() const { return true; }
    /// The process number is always zero
    int procnum() const { return 0; }
    /// The unique id number is always zero
    int getUniqueId() const { return 0; }
};
//SinglePeriodicBoundary

#endif
```

```
#ifndef MPULSE_PLASMACURRENT_H
#define MPULSE_PLASMACURRENT_H

#include "rebuild.h"
#include "currents.h"

class Storage;

class PlasmaCurrent : public Current, public Rebuildable
{
  protected:
    DataGrid *pEx;
    DataGrid *pEy;
    DataGrid *pEz;
    DataGrid *pRho;
    Storage *storage;

    /// e/m for the charge carriers
    double em;

    /// friction coefficient
    double gamma;
  public:
    PlasmaCurrent();

    void initStorage(Storage *storage_);

    void stepSchemeInit(double dt) {}
    void stepScheme(double dt);

  protected:
    ParameterMap* MakeParamMap (ParameterMap* pm = NULL);
};

#endif
```

```
#ifndef MPULSE_PLASMADENSITY_H
#define MPULSE_PLASMADENSITY_H

#include "rebuild.h"
#include "optfield.h"

class Storage;

class PlasmaDensity : public OptField
{
  protected:
    DataGrid *pEx;
    DataGrid *pEy;
    DataGrid *pEz;
    DataGrid *pRho;
    Storage *storage;

    /// avalanche ionization
    double nu;

    /// multi photon ionization
    double mpa;

    /// exponent for MPA
    int K;
  public:
    void initStorage(Storage *storage_);

    void stepSchemeInit(double dt) { stepScheme(dt/2); }
    void stepScheme(double dt);

  protected:
    /// build parametermap
    ParameterMap* MakeParamMap (ParameterMap* pm = NULL);
};

#endif
```

```
#ifndef MPULSE_PROCESS_H
#define MPULSE_PROCESS_H
//------------------------------------------------------------------------------
#include "rebuild.h"
#include "parameter.h"
#include "globals.h"
#include "fieldsim.h"

//------------------------------------------------------------------------------
/** @file process.h
  * @brief contains the process classes
  *
  * Here the process classes are defined, which are the global objects of this code.
  */

/** @brief The process class
  *
  * This class has only one instance and controls setup, rebuilding, initializing and
  * executing of all objects of the programm. First it calls all the Rebuild methods,
  * builds the correct objects and then distributes the execution to the execxte() members
  * of the appropiate classes. It also prints global (error) message on the screen.
  */
class Process : public Rebuildable {
  private:
    ///parameters
    Globals *globals;

    /// pointer to process
    static Process *process;

    /// pointer to the field simulation class
    FieldSimulation *fieldSim;

    /// time step
    int time;
  public:
       //------------------------------------------------------------------------
       //constructor & destructor
    /** @brief constructor,
      *
      * sets static member process to "this" and time step to 0. Since process is
      * static, the lifetime of the dynamical allocated Process object is prolonged
      * indefinetly. Which means, it has to be deleted in finalize()
      */
    Process()
    {
      process = this;
      time = 0;
    }

    ///destructor, deletes boundary
    ~Process();

    /// returns time
    int getTime() { return time; }

    /// returns reference to current instance of the process class
    static Process& instance() { return *process; }

    /** @brief Initialize the fields and species
     *
     * Call the initialize members of each items in the SpeciesList species and
     * the field from parameters.
     */
    void init();
```

```
    /** @brief Run the process.
     *
     * Which means: For the desired number of timesteps call the execute() members of
     * the objects stored in the speciesList
     */
    void run();

// ------------- Rebuildable -------------

  protected:
    ///build parametermap, containing boundary, species and general parameters
    ParameterMap* MakeParamMap (ParameterMap* pm = NULL);
  public:
    ///rebuild from file, call Rebuildable::Rebuild and makeProcess
    std::string Rebuild (std::istream& in);
};

#endif //MPULSE_PROCESS_H
```

```
#ifndef MPULSE_PULSEINIT_H
#define MPULSE_PULSEINIT_H

#include "init.h"

class GaussPulseInit : public FieldSimInit
{
  public:
    void init(Storage &fields);
  protected:
    /// build parametermap
    ParameterMap* MakeParamMap (ParameterMap* pm = NULL);
  private:
    double kz;
    double r0;
    double z0;
    double zc;
    double C;
    double bx, by;
};

#endif
```

```
#ifndef MPULSE_REBUILD_H
#define MPULSE_REBUILD_H

/** @file task.h
 *  @brief the fundametal class Rebuildable
 *
 *  Contains the declarations of the fundamental Rebuildable class
 */
//---------------------------------------------------------------------------
#include <list>
#include <vector>
#include <string>
#include "parameter.h"
//---------------------------------------------------------------------------

/** @brief fundamental base class for all the rebuildable objects
 *
 * All rebuildable objects inherit from this base class.
 *
 */
class Rebuildable {
  public:
    ///default constructor
    Rebuildable () {}
    /// virtual destructor
    virtual ~Rebuildable () {}

  protected:
    /** @brief Register the parameters needed for the Task
     *
     *  A derived class should overwrite this whenever it needs additional
     *  parameters from the setup file. It should then ALWAYS call the
     *  MakeParamMap of its superclass.
     *  If none present allocate new parameter map and return pointer to it.
     *  If a existing map is passed, do nothing.
     */
    virtual ParameterMap* MakeParamMap (ParameterMap* pm = NULL);
  public:
    /** @brief Rebuilds the task from the setup. This normally does not need
     *  to be overwritten.
     *
     *  Rebuild always returns the next token that does not belong to the
     *  object setup. Calls the rebuild methods of the objects stored in the parameter map.
     *  Will also remove all comments from the instream.
     */
    virtual std::string Rebuild (std::istream& in);
    ///virtual method,  does nothing unless overwritten by derived classes
    virtual void finalize() {};
};

#endif // MPULSE_REBUILD_H
```

```
#ifndef MPULSE_INCSOURCE_H
#define MPULSE_INCSOURCE_H

#include "incsource.h"

class PlaneWaveSource : public IncidentSource
{
  public:
    ~PlaneWaveSource() {}
  protected:
    IncidentSourceCurrent *makeECurrent(int distance_, Direction dir_);
    IncidentSourceCurrent *makeHCurrent(int distance_, Direction dir_);

    ParameterMap* MakeParamMap (ParameterMap* pm = NULL);

    double kx, ky, kz;
    double Hx, Hy, Hz;
    double ramp;

};

class PlaneWaveSourceECurrent : public IncidentSourceECurrent
{
  public:
    PlaneWaveSourceECurrent(int distance_, Direction dir_, Vector k_, Vector E_, Vector H_)
;

    ~PlaneWaveSourceECurrent() {}

    void stepSchemeInit(double dt);
    void stepScheme(double dt);

  private:
    Vector k;
    Vector E;
    Vector H;

    double dt;
    double om;
};

class PlaneWaveSourceHCurrent : public IncidentSourceHCurrent
{
  public:
    PlaneWaveSourceHCurrent(int distance_, Direction dir_, Vector k_, Vector E_, Vector H_)
;

    ~PlaneWaveSourceHCurrent() {}

    virtual void stepSchemeInit(double dt);
    virtual void stepScheme(double dt);

  private:
    Vector k;
    Vector E;
    Vector H;

    double dt;
    double om;
};

#endif
```

```
// -*- C++ -*-
// $Id: stlpwrapper.h,v 1.6 2006/10/23 09:45:34 hs Exp $


/** @file stlpwrapper.h
 *  @brief simple pointer wrapper
 *
 *  Contains a pointer wrapper derived from Mhumit Khan stl.html
 *  @todo implement a proper smart pointer
 */


// the user still has to delete the objects from the heap

#ifndef STLPWRAPPER_H
#define STLPWRAPPER_H

/** @brief A simple pointer wrapper
 *
 *  Wraps a pointer, so that it can be stored in an STL container.
 *  The current implementation has however little advantages.
 *  Care has to be taken, for the destructor does not delete dynamic allocated objects!
 *  Users still have to delete these objects from the heap.
 */
template <class TYPE>
class PtrWrapper {
  private:
    /// The pointer
    TYPE* m_pObj;
  public:
    /// Constructor with a pointer to the TYPE, default is NULL pointer
    PtrWrapper(TYPE* pObj = 0) : m_pObj(pObj) {};
    /// Copy constructor
    PtrWrapper(const PtrWrapper<TYPE>& wrapper) : m_pObj(wrapper.m_pObj) {}

    /// Assignment operator
    PtrWrapper<TYPE>& operator= (const PtrWrapper<TYPE>& wrapper) {
      m_pObj = wrapper.m_pObj;
      return *this;
    }

    /// Destructor doesn't free pointer
    ~PtrWrapper() {};

    /// Typecast to the original pointer type
    operator const TYPE* () const { return m_pObj; }
    /// Typecast to the original pointer type
    operator TYPE* () { return m_pObj; }
    /// Typecast to the original pointer type
    TYPE* operator->() { return m_pObj; }
    /// Explicit method for the typecast
    TYPE* pObj () const { return m_pObj; };

    /// Compares the two objects by address
    friend bool operator== (const PtrWrapper<TYPE>& left, const PtrWrapper<TYPE>& right) {
      return (left.m_pObj == right.m_pObj);
    };

    /// Compares the two objects
    friend bool operator!= (const PtrWrapper<TYPE>& left, const PtrWrapper<TYPE>& right) {
      return (left.m_pObj != right.m_pObj);
    };

};

#endif // STLPWRAPPER_H
```

```
#ifndef MPULSE_STORAGE_H
#define MPULSE_STORAGE_H

#include "mpulse.h"
#include "boundary.h"
#include <map>
#include <list>
#include <string>

class Boundary;


/** Storage Base class for storing the fields
 */
class Storage
{
  public:
    /** Construct the fields with zero size*/
    Storage();

    /** Destructor (empty) */
    ~Storage();

    /** Resizes the fields to span from low to high */
    void resize(GridIndex low_, GridIndex high_);

  protected:
    /** The low index of the fields */
    GridIndex low;
    /** The high index of the fields */
    GridIndex high;

    /** The grid delta x */
    double dx;
    /** The grid delta y */
    double dy;
    /** The grid delta z */
    double dz;

  public:

    DataGrid &getGrid(const std::string &gridid);
    bool hasGrid(const std::string &gridid);

    template<class Func>
    void forAllGrids(const Func &func);

    DataGrid *addGrid(const std::string &gridid);
    void addToGroup(const std::string &groupid, const std::string &gridid);

    DataGrid &getBorderLayer(const std::string &gridid, Direction dir);
    bool hasBorderLayer(const std::string &gridid, Direction dir);
    DataGrid *addBorderLayer(const std::string &gridid, Direction dir, int thickness, int d
istance=0);

    DataLine &getLine(const std::string &lineid);
    bool hasLine(const std::string &lineid);
    DataLine *addLine(const std::string &lineid, int orientation);

    /** Apply the boundary condition to the electric field*/
    void applyBoundary(const std::string &groupid);

    const GridIndex &getLow() { return low; }
    const GridIndex &getHigh() { return high; }

    double getDx() { return dx; }
```

```
    double getDy() { return dy; }
    double getDz() { return dz; }

  protected:
    Boundary *boundary;

  private:
    typedef std::map<std::string, DataGrid*> GridMap;
    typedef std::map<std::string, DataLine*> LineMap;
    typedef std::list<std::string> IdList;
    GridMap grids;
    GridMap gridsN;
    GridMap gridsS;
    GridMap gridsE;
    GridMap gridsW;
    GridMap gridsU;
    GridMap gridsD;

    LineMap lines;

    std::map<std::string,IdList> groups;

    template<class Func>
    void forAllGrids(GridMap &gm, Func &func);

    bool getBorderExtent(Direction dir, int thickness, int distance, GridIndex &blow, GridI
ndex &bhigh);

    struct GridDeleter
    {
      void operator()(std::string, DataGrid* grid);
    };

    struct GridResizer
    {
      const GridIndex &low, &high;
      GridResizer(const GridIndex &low_, const GridIndex &high_);
      void operator()(std::string, DataGrid* grid);
    };
  public:
    const Boundary& getBoundary() const { return *boundary; }
};



#endif // MPULSE_STORAGE_H
```

```
#include <cmath>
#ifndef UTIL_H
#define UTIL_H

/** @file util.h
  * @brief helper functions
  *
  * Defines some helper function templates for mathematical operations
  */

#if ! defined (__GNUG__) && ! defined (__HP_aCC)
/** returns maximum of two T's */
template<class T>
inline T max(T a, T b)
{
  return a > b ? a:b;
}
/** returns maximum of two T's */
template<class T>
inline T min(T a, T b)
{
  return a < b ? a:b;
}
/** returns absolute value */
template<class T>
inline T abs(T a)
{
  return (a > 0) ? a : -a;
}
#endif

/**returns square root */
template<class T>
inline T sqr(T a)
{
  return a*a;
}

/** returns the norm of a*/
inline double norm(double a)
{
  return a*a;
}

/** returns norm of a */
inline float norm(float a)
{
  return a*a;
}

/** returns sign of a*/
template<class T>
inline T sign(T a, T b)
{
  return (b >= 0) ? abs(a) : -abs(a);
}

/** returns logarithm to base 2 */
template<class T>
inline T log2(T a)
{
  const double inv_log2 = 1/log(2.);
  return T(inv_log2*log(a));
}

template<class T>
```

```
    inline T ipow(T x, unsigned int e)
{
  T r(1);
  for (unsigned int i=0; i<e; ++i)
    r *= x;
  return r;
}

#endif
```

```
#ifndef MPULSE_WAVEINIT_H
#define MPULSE_WAVEINIT_H

#include "init.h"

class PlaneWaveInit : public FieldSimInit
{
  public:
    void init(Storage &fields);
  protected:
    /// build parametermap
    ParameterMap* MakeParamMap (ParameterMap* pm = NULL);
  private:
    int kx, ky, kz;
    double bx, by, bz;
};

#endif
```

```
#include "mpulse.h"
#include "cpml_border.h"
#include "fieldsolver.h"
#include "globals.h"
#include "storage.h"

#include <vector>

// define TESTING_CORRECTIONS
// define TESTING_CORRECTIONS_B
#define TESTING_CORRECTIONS_C

//=================================================================
//==========  CPMLBorder
//=================================================================


void CPMLBorder::initCurrents(Storage *storage, FieldSolver *solver)
{
  solver->addCurrent(
    new CPMLBorderECurrent(thickness, north, this->kappaMax, this->aMax, this->sigmaMax, 1)
  );
  solver->addCurrent(
    new CPMLBorderECurrent(thickness, south, this->kappaMax, this->aMax, this->sigmaMax, 1)
  );
  solver->addCurrent(
    new CPMLBorderECurrent(thickness, east, this->kappaMax, this->aMax, this->sigmaMax, 1)
  );
  solver->addCurrent(
    new CPMLBorderECurrent(thickness, west, this->kappaMax, this->aMax, this->sigmaMax, 1)
  );
  solver->addCurrent(
    new CPMLBorderECurrent(thickness, up, this->kappaMax, this->aMax, this->sigmaMax, 1)
  );
  solver->addCurrent(
    new CPMLBorderECurrent(thickness, down, this->kappaMax, this->aMax, this->sigmaMax, 1)
  );

  solver->addMagCurrent(
    new CPMLBorderHCurrent(thickness, north, this->kappaMax, this->aMax, this->sigmaMax, 1)
  );
  solver->addMagCurrent(
    new CPMLBorderHCurrent(thickness, south, this->kappaMax, this->aMax, this->sigmaMax, 1)
  );
  solver->addMagCurrent(
    new CPMLBorderHCurrent(thickness, east, this->kappaMax, this->aMax, this->sigmaMax, 1)
  );
  solver->addMagCurrent(
    new CPMLBorderHCurrent(thickness, west, this->kappaMax, this->aMax, this->sigmaMax, 1)
  );
  solver->addMagCurrent(
    new CPMLBorderHCurrent(thickness, up, this->kappaMax, this->aMax, this->sigmaMax, 1)
  );
  solver->addMagCurrent(
    new CPMLBorderHCurrent(thickness, down, this->kappaMax, this->aMax, this->sigmaMax, 1)
  );

  initCoefficients(storage);

}

void CPMLBorder::initCoefficients(Storage *storage)
{
  // initialize Kappas here

  double dt = Globals::instance().dt();
```

```
GridIndex glow  = Globals::instance().gridLow();
GridIndex ghigh = Globals::instance().gridHigh();

GridIndex low  = storage->getLow();
GridIndex high = storage->getHigh();

std::vector<DataLine *> pKappaEdk(3);
std::vector<DataLine *> pKappaHdk(3);

std::vector<DataLine *> pStretchEa(3);
std::vector<DataLine *> pStretchEb(3);

std::vector<DataLine *> pStretchHa(3);
std::vector<DataLine *> pStretchHb(3);

pKappaEdk[0] = storage->addLine("KappaEdx", 0);
pKappaEdk[1] = storage->addLine("KappaEdy", 1);
pKappaEdk[2] = storage->addLine("KappaEdz", 2);

pKappaHdk[0] = storage->addLine("KappaHdx", 0);
pKappaHdk[1] = storage->addLine("KappaHdy", 1);
pKappaHdk[2] = storage->addLine("KappaHdz", 2);

pStretchEa[0] = storage->addLine("StretchEax", 0);
pStretchEa[1] = storage->addLine("StretchEay", 1);
pStretchEa[2] = storage->addLine("StretchEaz", 2);

pStretchEb[0] = storage->addLine("StretchEbx", 0);
pStretchEb[1] = storage->addLine("StretchEby", 1);
pStretchEb[2] = storage->addLine("StretchEbz", 2);

pStretchHa[0] = storage->addLine("StretchHax", 0);
pStretchHa[1] = storage->addLine("StretchHay", 1);
pStretchHa[2] = storage->addLine("StretchHaz", 2);

pStretchHb[0] = storage->addLine("StretchHbx", 0);
pStretchHb[1] = storage->addLine("StretchHby", 1);
pStretchHb[2] = storage->addLine("StretchHbz", 2);

std::cerr << "Field Size Edx " << pKappaEdk[0]->getLow()[0] << ", "
                               << pKappaEdk[0]->getHigh()[0] <<std::endl;

std::cerr << "Field Size Edy " << pKappaEdk[1]->getLow()[0] << ", "
                               << pKappaEdk[1]->getHigh()[0] <<std::endl;

std::cerr << "Field Size Edz " << pKappaEdk[2]->getLow()[0] << ", "
                               << pKappaEdk[2]->getHigh()[0] <<std::endl;

std::cerr << "Field Size Hdx " << pKappaHdk[0]->getLow()[0] << ", "
                               << pKappaHdk[0]->getHigh()[0] <<std::endl;

std::cerr << "Field Size Hdy " << pKappaHdk[1]->getLow()[0] << ", "
                               << pKappaHdk[1]->getHigh()[0] <<std::endl;

std::cerr << "Field Size Hdz " << pKappaHdk[2]->getLow()[0] << ", "
                               << pKappaHdk[2]->getHigh()[0] <<std::endl;

for (int dim = 0; dim<3; ++dim)
{
  std::cerr << "Dim " << dim << std::endl;
  if (low[dim]<glow[dim]+thickness)
  {
    (*pKappaEdk[dim]) = 1.0;
    (*pStretchEa[dim]) = 1.0;
    (*pStretchEb[dim]) = 1.0;
```

```
      (*pKappaHdk[dim]) = 1.0;
      (*pStretchHa[dim]) = 1.0;
      (*pStretchHb[dim]) = 1.0;

      for (int i=0; i<=thickness; ++i)
      {
        double x  = 1 - double(i)/double(thickness);
        double x3 = x*x*x;
        double sigma = this->sigmaMax * x3;
        (*pKappaEdk[dim])(low[dim]+i) = 1 + (this->kappaMax - 1)*x3;
#ifndef TESTING_CORRECTIONS_C
        (*pStretchEa[dim])(low[dim]+i) = (1-0.5*sigma*dt/1.) / (1+0.5*sigma*dt/1.);
        (*pStretchEb[dim])(low[dim]+i) = (1/1.) / (1+0.5*sigma*dt/1.);
#endif
      }
#ifdef TESTING_CORRECTIONS
      for (int i=0; i<=thickness; ++i)
#else
      for (int i=0; i<thickness; ++i)
#endif
      {
        double x  = 1 - (double(i)+0.5)/double(thickness);
#ifdef TESTING_CORRECTIONS
        x  = 1 - double(i)/double(thickness);
#endif
        double x3 = x*x*x;
        double sigma = this->sigmaMax * x3;
        (*pKappaHdk[dim])(low[dim]+i) = 1 + (this->kappaMax - 1)*x3;
#ifndef TESTING_CORRECTIONS_C
        (*pStretchHa[dim])(low[dim]+i) = (1-0.5*sigma*dt/1.) / (1+0.5*sigma*dt/1.);
        (*pStretchHb[dim])(low[dim]+i) = (1/1.) / (1+0.5*sigma*dt/1.);
#endif
      }
    }

    if (high[dim]>ghigh[dim]-thickness)
    {
      for (int i=0; i<=thickness; ++i)
      {
        double x  = 1 - double(i)/double(thickness);
        double x3 = x*x*x;
        double sigma = this->sigmaMax * x3;
        (*pKappaHdk[dim])(high[dim]-i) = 1 + (this->kappaMax - 1)*x3;
#ifndef TESTING_CORRECTIONS_C
        (*pStretchHa[dim])(high[dim]-i) = (1-0.5*sigma*dt/1.) / (1+0.5*sigma*dt/1.);
        (*pStretchHb[dim])(high[dim]-i) = (1/1.) / (1+0.5*sigma*dt/1.);
#endif
      }
#ifdef TESTING_CORRECTIONS
      for (int i=0; i<=thickness; ++i)
#else
      for (int i=0; i<thickness; ++i)
#endif

      {
        double x  = 1 - (double(i)+0.5)/double(thickness);
#ifdef TESTING_CORRECTIONS
        x  = 1 - double(i)/double(thickness);
#endif
        double x3 = x*x*x;
        double sigma = this->sigmaMax * x3;
        (*pKappaEdk[dim])(high[dim]-i) = 1 + (this->kappaMax - 1)*x3;
#ifndef TESTING_CORRECTIONS_C
        (*pStretchEa[dim])(high[dim]-i) = (1-0.5*sigma*dt/1.) / (1+0.5*sigma*dt/1.);
        (*pStretchEb[dim])(high[dim]-i) = (1/1.) / (1+0.5*sigma*dt/1.);
```

```
#endif
      }
    }
  }
}

ParameterMap* CPMLBorder::MakeParamMap (ParameterMap* pm)
{
  pm = CurrentSim::MakeParamMap(pm);

  (*pm)["d"] = WParameter(new ParameterValue<int>(&this->thickness,8));
  (*pm)["kappaMax"] = WParameter(new ParameterValue<double>(&this->kappaMax,15));
  (*pm)["aMax"] = WParameter(new ParameterValue<double>(&this->aMax,0.25));
  (*pm)["sigmaMax"] = WParameter(new ParameterValue<double>(&this->sigmaMax,3));
  return pm;
}

//=============================================================
//==========  CPMLBorderCurrent
//=============================================================


CPMLBorderCurrent::CPMLBorderCurrent( int thickness_, Direction dir_, bool isH_,
                                      double kappaMax_, double aMax_, double sigmaMax_, dou
ble eps_)
  : thickness(thickness_), dir(dir_), isH(isH_),
    kappaMax(kappaMax_), aMax(aMax_), sigmaMax(sigmaMax_), eps(eps_)
{
  switch (dir)
  {
    case east:
    case west:  dim = 0;
                transverse1 = 1;
                transverse2 = 2;
                break;
    case north:
    case south: dim = 1;
                transverse1 = 0;
                transverse2 = 2;
                break;
    case up:
    case down:  dim = 2;
                transverse1 = 0;
                transverse2 = 1;
                break;
  }
}

void CPMLBorderCurrent::makeCoeff(Storage *storage)
{
  double dt = Globals::instance().dt();

  GridIndex low  = pJx->getLow();
  GridIndex high = pJx->getHigh();

  switch (dir)
  {
    case east:
    case north:
    case up:    reverse = false; break;
    case west:
    case south:
    case down:  reverse = true;  break;
  }

  switch (dir)
```

```
  {
    case east:  zerolayer = high[0]; break;
    case west:  zerolayer = low[0]; break;
    case north: zerolayer = high[1]; break;
    case south: zerolayer = low[1]; break;
    case up:    zerolayer = high[2]; break;
    case down:  zerolayer = low[2]; break;
  }


  int lowk  = low[dim];
  int highk = high[dim];

  bCoeff.resize(lowk, highk);
  cCoeff.resize(lowk, highk);

  double offset = 0.0;
  lowOffset = 0;
  highOffset = 0;

  if (isH && !reverse)
  {
    offset = 0.5;
    lowOffset = 1;
  }

  if (!isH && reverse)
  {
    offset = 0.5;
    highOffset = 1;
  }
#ifdef TESTING_CORRECTIONS
    offset = 0;
    highOffset = 0;
#endif


  for (int k=0; k<=highk-lowk; ++k)
  {
    double x = 1 - (double(k)+offset)/double(thickness);
    double x3 = x*x*x;

    int pos = reverse ? (highk-k) : (lowk+k);

    double sigma = x3*sigmaMax;
    double kappa = 1 + (kappaMax - 1)*x3;
    double a = aMax*(1-x);

    double testing_f = 1.0;
#ifdef TESTING_CORRECTIONS_B
    testing_f = -1.0;
#endif
    double b = exp(-testing_f*(sigma/kappa + a)*dt/eps);
    double c = sigma*(b-1)/(kappa*(sigma+kappa*a));

    bCoeff(pos) = b;
    cCoeff(pos) = c;
  }

}

//===============================================================
//==========  CPMLBorderECurrent
//===============================================================

CPMLBorderECurrent::CPMLBorderECurrent( int thickness_, Direction dir_,
```

```
                                        double kappaMax_, double aMax_, double sigmaMax_, d
ouble eps_)
  : CPMLBorderCurrent(thickness_,dir_,false,kappaMax_,aMax_,sigmaMax_,eps_)
{}

void CPMLBorderECurrent::initStorage(Storage *storage)
{
  int realThickness;

  switch (dir)
  {
    case east:
    case north:
    case up:    realThickness = thickness-1; break;
    case west:
    case south:
    case down:  realThickness = thickness; break;
  }
#ifdef TESTING_CORRECTIONS
        realThickness = thickness;
#endif

  pJx = storage->addBorderLayer("CPMLPsiEx", dir, realThickness);
  pJy = storage->addBorderLayer("CPMLPsiEy", dir, realThickness);
  pJz = storage->addBorderLayer("CPMLPsiEz", dir, realThickness);

  switch (dir)
  {
    case east:
    case west:
      pPsi[0] = pJy;
      pPsi[1] = pJz;
      pB[0] = &storage->getGrid("By");
      pB[1] = &storage->getGrid("Bz");
      pB[2] = &storage->getGrid("Bx");
      dx = Globals::instance().gridDX();
      break;
    case north:
    case south:
      pPsi[0] = pJz;
      pPsi[1] = pJx;
      pB[0] = &storage->getGrid("Bz");
      pB[1] = &storage->getGrid("Bx");
      pB[2] = &storage->getGrid("By");
      dx = Globals::instance().gridDY();
      break;
    case up:
    case down:
      pPsi[0] = pJx;
      pPsi[1] = pJy;
      pB[0] = &storage->getGrid("Bx");
      pB[1] = &storage->getGrid("By");
      pB[2] = &storage->getGrid("Bz");
      dx = Globals::instance().gridDZ();
      break;
  }

  if (pJx) makeCoeff(storage);
  else
  {
    std::cerr << "No E Current here: Direction " << dir << " thickness " <<realThickness <<
 " \n";
  }

}
```

```
void CPMLBorderECurrent::stepSchemeInit(double dt)
{}

void CPMLBorderECurrent::stepScheme(double dt)
{
  GridIndex low  = pPsi[0]->getLow();
  GridIndex high = pPsi[0]->getHigh();

  low[dim] += lowOffset;
  high[dim] -= highOffset;

  DataGrid &Psi0 = *pPsi[0];
  DataGrid &Psi1 = *pPsi[1];
  DataGrid &B0 = *pB[0];
  DataGrid &B1 = *pB[1];
  DataGrid &B2 = *pB[2];

  GridIndex ind;

  int t1 = transverse1;
  int t2 = transverse2;

  int sign = 1; //reverse?-1:1;

  ind[dim] = zerolayer;
  for (ind[t1]=low[t1]; ind[t1]<=high[t1]; ++ind[t1])
    for (ind[t2]=low[t2]; ind[t2]<=high[t2]; ++ind[t2])
    {
      B0(ind[0], ind[1], ind[2]) = 0;
      B1(ind[0], ind[1], ind[2]) = 0;
      B2(ind[0], ind[1], ind[2]) = 0;
    }

  for (ind[0]=low[0]; ind[0]<=high[0]; ++ind[0])
    for (ind[1]=low[1]; ind[1]<=high[1]; ++ind[1])
      for (ind[2]=low[2]; ind[2]<=high[2]; ++ind[2])
      {
        int j = ind[dim];
        GridIndex indm(ind);
        --indm[dim];

        Psi0(ind[0], ind[1], ind[2])
          = bCoeff(j)*Psi0(ind[0], ind[1], ind[2])
            - sign*cCoeff(j)*(B1(ind[0], ind[1], ind[2])-B1(indm[0], indm[1], indm[2]))/dx;
        Psi1(ind[0], ind[1], ind[2])
          = bCoeff(j)*Psi1(ind[0], ind[1], ind[2])
            + sign*cCoeff(j)*(B0(ind[0], ind[1], ind[2])-B0(indm[0], indm[1], indm[2]))/dx;
      }
}


//================================================================
//==========  CPMLBorderHCurrent
//================================================================

CPMLBorderHCurrent::CPMLBorderHCurrent( int thickness_, Direction dir_,
                                        double kappaMax_, double aMax_, double sigmaMax_, d
ouble eps_)
  : CPMLBorderCurrent(thickness_,dir_,true,kappaMax_,aMax_,sigmaMax_,eps_)
{}

void CPMLBorderHCurrent::initStorage(Storage *storage)
{
  int realThickness;

  switch (dir)
```

```
  {
    case east:
    case north:
    case up:    realThickness = thickness; break;
    case west:
    case south:
    case down:  realThickness = thickness-1; break;
  }
#ifdef TESTING_CORRECTIONS
        realThickness = thickness;
#endif

  pJx = storage->addBorderLayer("CPMLPsiBx", dir, realThickness);
  pJy = storage->addBorderLayer("CPMLPsiBy", dir, realThickness);
  pJz = storage->addBorderLayer("CPMLPsiBz", dir, realThickness);

  switch (dir)
  {
    case east:
    case west:
      pPsi[0] = pJy;
      pPsi[1] = pJz;
      pE[0] = &storage->getGrid("Ey");
      pE[1] = &storage->getGrid("Ez");
      pE[2] = &storage->getGrid("Ex");
      dx = Globals::instance().gridDX();
      break;
    case north:
    case south:
      pPsi[0] = pJz;
      pPsi[1] = pJx;
      pE[0] = &storage->getGrid("Ez");
      pE[1] = &storage->getGrid("Ex");
      pE[2] = &storage->getGrid("Ey");
      dx = Globals::instance().gridDY();
      break;
    case up:
    case down:
      pPsi[0] = pJx;
      pPsi[1] = pJy;
      pE[0] = &storage->getGrid("Ex");
      pE[1] = &storage->getGrid("Ey");
      pE[2] = &storage->getGrid("Ez");
      dx = Globals::instance().gridDZ();
      break;
  }

  if (pJx) makeCoeff(storage);
  else
  {
    std::cerr << "No H Current here: Direction " << dir << " thickness " <<realThickness <<
 " \n";
  }
}

void CPMLBorderHCurrent::stepSchemeInit(double dt)
{
  stepScheme(0.5*dt);
}

void CPMLBorderHCurrent::stepScheme(double dt)
{
  GridIndex low  = pPsi[0]->getLow();
  GridIndex high = pPsi[0]->getHigh();

  low[dim] += lowOffset;
```

```
  high[dim] -= highOffset;

  DataGrid &Psi0 = *pPsi[0];
  DataGrid &Psi1 = *pPsi[1];
  DataGrid &E0 = *pE[0];
  DataGrid &E1 = *pE[1];
  DataGrid &E2 = *pE[2];

  GridIndex ind;

  int t1 = transverse1;
  int t2 = transverse2;

  int sign = 1; //reverse?-1:1;

  ind[dim] = zerolayer;
  for (ind[t1]=low[t1]; ind[t1]<=high[t1]; ++ind[t1])
    for (ind[t2]=low[t2]; ind[t2]<=high[t2]; ++ind[t2])
    {
      E0(ind[0], ind[1], ind[2]) = 0;
      E1(ind[0], ind[1], ind[2]) = 0;
      E2(ind[0], ind[1], ind[2]) = 0;
    }

  for (ind[0]=low[0]; ind[0]<=high[0]; ++ind[0])
    for (ind[1]=low[1]; ind[1]<=high[1]; ++ind[1])
      for (ind[2]=low[2]; ind[2]<=high[2]; ++ind[2])
      {
        int j = ind[dim];
        GridIndex indp(ind);
        ++indp[dim];

        Psi0(ind[0], ind[1], ind[2])
          = bCoeff(j)*Psi0(ind[0], ind[1], ind[2])
            - sign*cCoeff(j)*(E1(indp[0], indp[1], indp[2])-E1(ind[0], ind[1], ind[2]))/dx;

        Psi1(ind[0], ind[1], ind[2])
          = bCoeff(j)*Psi1(ind[0], ind[1], ind[2])
            + sign*cCoeff(j)*(E0(indp[0], indp[1], indp[2])-E0(ind[0], ind[1], ind[2]))/dx;
      }
}
```

```
#include "diagnostic.h"
#include <sstream>
#include "process.h"
#include "globals.h"
#include "boundary.h"

DiagnosticInterface::DiagnosticInterface() {
  DiagnosticManager::instance().addDiagnostic(this);
  t=0;
}

void DiagnosticInterface::execute() {
  // perform calculations first
  calculate();

  if (singleOut() && !(Globals::instance().isMaster()) ) return;

  //if its the first call of execute() and appending is true open the file
  if ((0==t) && appending()) open(fname);

  //if timestep is a multiple of interval perform output
  if ( (t % interval) == 0 )
  {
    //if not appending open a new file
    if (!appending()) open(parsedFileName());
    write();
    //if not appending close the file
    if (!appending()) close();
  }
  ++t;
}

ParameterMap* DiagnosticInterface::MakeParamMap(ParameterMap* pm)
{
  pm = Rebuildable::MakeParamMap(pm);
  (*pm)["file"] = WParameter(
    new ParameterValue<std::string>(&fname, "")
  );
  (*pm)["append"] = WParameter(
    new ParameterValue<std::string>(&append, "n")
  );
  (*pm)["interval"] = WParameter(
    new ParameterValue<int>(&interval, 100)
  );
  return pm;
}

bool DiagnosticInterface::appending() {
  return 'y' == append[0];
}

std::string DiagnosticInterface::parsedFileName() {
  std::string parsed=fname;
  //look up boundary and ID of process
  std::ostringstream comrankstr;
  comrankstr << Globals::instance().getUniqueId();
  std::string comrank = comrankstr.str();
  //look up the global time step
  std::ostringstream tstepstr;
  tstepstr << Process::instance().getTime();
  std::string tstep = tstepstr.str();
  //replace placeholders with the appropiate variables
  size_t pos;
  pos = parsed.find("#p");
  if (pos != std::string::npos) parsed.replace(pos,2,comrank);
```

```
  pos = parsed.find("#t");
  if (pos != std::string::npos) parsed.replace(pos,2,tstep);

  return parsed;
}

DiagnosticManager *DiagnosticManager::theManager = NULL;

DiagnosticManager::DiagnosticManager() {}

void DiagnosticManager::addDiagnostic(DiagnosticInterface *diag)
{
  diags.push_back(pDiagnosticInterface(diag));
}

void DiagnosticManager::execute()
{
  for (
    DiagList::iterator it=diags.begin();
    it != diags.end();
    ++it
  )
  {
    (*it)->execute();
  }
}

DiagnosticManager& DiagnosticManager::instance()
{
  if (theManager==NULL) theManager = new DiagnosticManager;
  return *theManager;
}


/* meyers singelton should work here:
 * DiagnosticManager& DiagnosticManager::instance()
 * {
 *      static DiagnosticManager theManager ;
 *      return theManager;
 * }
 */
```

```
#include "factor.h"
#include "algohelper.h"

#include <list>
#include <vector>
#include <iostream>
#include <cmath>
#include <algorithm>

void makePrimes(std::list<int> &primes, int max);
void factorize(int number, std::list<int> &primes, std::list<int> &factors);
double distribute
  (
    std::vector<int> &factors,
    std::list<int> allfact,
    std::vector<int> &weights
  );

void equalFactors
  (
    int number,
    int nfact,
    std::vector<int> &factors,
    std::vector<int> &weights
  )
{
  std::list<int> primes;
  std::list<int> allfact;
  makePrimes(primes, int(floor(sqrt(number))));
  factorize(number, primes, allfact);

  factors.resize(nfact);

  factors[0] = allfact.front();
  for (int i=1; i<nfact; ++i) factors[i] = 1;

  allfact.pop_front();
  if (allfact.empty()) return;
  distribute(factors, allfact, weights);
}

void makePrimes(std::list<int> &primes, int max)
{
  std::vector<bool> isprime(max+1,true);
  primes.clear();
  for (int i=2; i<=max; ++i)
  {
    if (isprime[i])
    {
      primes.push_back(i);
      int prod = 2*i;
      for (prod = 2*i; prod<=max; prod += i)
      {
        isprime[prod] = false;
      }
    }
  }
}

void factorize(int number, std::list<int> &primes, std::list<int> &factors)
{
  if (number == 1) return;
  std::list<int>::iterator it = primes.begin();

  while (it != primes.end())
```

```
  {
    int p = *it;
    if ((number % p) == 0)
    {
      factorize(number/p, primes, factors);
      factors.push_front(p);
      return;
    }
    ++it;
  }
  factors.push_front(number);
}

double distribute
  (
    std::vector<int> &factors,
    std::list<int> allfact,
    std::vector<int> &weights
  )
{
  int f = allfact.front();
  allfact.pop_front();

  if (allfact.empty())
  {
    std::vector<double> dfactors;
    dfactors.resize(factors.size());

    int imin = 0;
    double facmin = 0;

    for (size_t i=0; i<factors.size(); ++i)
    {
      dfactors[i] = double(factors[i])/double(weights[i]);
      if ((i==0) || (dfactors[i] < facmin))
      {
        imin = i;
        facmin = dfactors[i];
      }
    }

    dfactors[imin] *= f;
    factors[imin] *= f;

    return std::for_each(dfactors.begin(), dfactors.end(), calcsum<double>()).result();

  }
  else
  {
    std::vector<int> best_factors = factors;
    best_factors[0] *= f;
    double bestSum = distribute(best_factors, allfact, weights);

    for (size_t i=1; i<factors.size(); ++i)
    {
      std::vector<int> t_factors = factors;
      t_factors[i] *= f;
      double sum = distribute(t_factors, allfact, weights);
      if (sum < bestSum)
      {
        best_factors = t_factors;
        bestSum = sum;
      }
    }

    factors = best_factors;
```

```
    return bestSum;
  }
}

/*
// Testing program for factors
int main()
{
  std::vector<int> factors;
  int number = 64;
  int nfact = 3;

  std::vector<int> weights(nfact);
  weights[0] = 32;
  weights[1] = 32;
  weights[2] = 256;

  equalFactors(number, 3, factors, weights);

  std::cout << number << " = ";
  for (int i = 0; i < nfact; ++i)
  {
    std::cout << factors[i] << "*";
  }
    std::cout << std::endl;
}
*/
```

```
#include "fdtd_disp.h"
#include "storage.h"
#include "plasmacurrent.h"
#include <cmath>

void FDTD_Dispersion::initStorage(Storage *storage_)
{
  storage = storage_;

  pEx = storage->addGrid("Ex");
  pEy = storage->addGrid("Ey");
  pEz = storage->addGrid("Ez");

  pBx = storage->addGrid("Bx");
  pBy = storage->addGrid("By");
  pBz = storage->addGrid("Bz");

  pPx[0] = storage->addGrid("P1x");
  pPy[0] = storage->addGrid("P1y");
  pPz[0] = storage->addGrid("P1z");

  pPx[1] = storage->addGrid("P2x");
  pPy[1] = storage->addGrid("P2y");
  pPz[1] = storage->addGrid("P2z");

  pPx[2] = storage->addGrid("P3x");
  pPy[2] = storage->addGrid("P3y");
  pPz[2] = storage->addGrid("P3z");

  pPxp[0] = storage->addGrid("P1xp");
  pPyp[0] = storage->addGrid("P1yp");
  pPzp[0] = storage->addGrid("P1zp");

  pPxp[1] = storage->addGrid("P2xp");
  pPyp[1] = storage->addGrid("P2yp");
  pPzp[1] = storage->addGrid("P2zp");

  pPxp[2] = storage->addGrid("P3xp");
  pPyp[2] = storage->addGrid("P3yp");
  pPzp[2] = storage->addGrid("P3zp");

  storage->addToGroup("E", "Ex");
  storage->addToGroup("E", "Ey");
  storage->addToGroup("E", "Ez");

  storage->addToGroup("B", "Bx");
  storage->addToGroup("B", "By");
  storage->addToGroup("B", "Bz");

  for
  (
    CurrentList::iterator it = currents.begin();
    it != currents.end();
    ++it
  )
  {
    (*it)->initStorage(storage);
  }

}

void FDTD_Dispersion::stepSchemeInit(double dt)
{
  stepB(0.5*dt);
```

```
  for
  (
    CurrentList::iterator it = currents.begin();
    it != currents.end();
    ++it
  )
  {
    (*it)->stepSchemeInit(dt);
  }
}

void FDTD_Dispersion::stepScheme(double dt)
{
  stepD(dt);
  stepB(dt);

  for
  (
    CurrentList::iterator it = currents.begin();
    it != currents.end();
    ++it
  )
  {
    (*it)->stepScheme(dt);
  }
}

void FDTD_Dispersion::stepD(double dt)
{
  DataGrid &Ex = *pEx;
  DataGrid &Ey = *pEy;
  DataGrid &Ez = *pEz;

  DataGrid &Bx = *pBx;
  DataGrid &By = *pBy;
  DataGrid &Bz = *pBz;

  GridIndex low = storage->getLow();
  GridIndex high = storage->getHigh();

  double dx = storage->getDx();
  double dy = storage->getDy();
  double dz = storage->getDz();

  double dt2 = dt*dt;

  for (int i=low[0]+1; i<=high[0]; ++i)
    for (int j=low[1]+1; j<=high[1]; ++j)
      for (int k=low[2]+1; k<=high[2]; ++k)
  {
    double ex = Ex(i,j,k);
    double ey = Ey(i,j,k);
    double ez = Ez(i,j,k);

    double Px = 0;
    double Py = 0;
    double Pz = 0;

    double Pxp = 0;
    double Pyp = 0;
    double Pzp = 0;

    for (int n=0;n<3;++n)
    {
      double &px = pPx[n]->operator()(i,j,k);
      double &py = pPy[n]->operator()(i,j,k);
```

```
      double &pz = pPz[n]->operator()(i,j,k);

      double &pxp = pPxp[n]->operator()(i,j,k);
      double &pyp = pPyp[n]->operator()(i,j,k);
      double &pzp = pPzp[n]->operator()(i,j,k);

      double a = dt2*LOm2[n];
      double b = a*LEps2[n];
      double pxn = (2-a)*px - pxp + b*ex;
      double pyn = (2-a)*py - pyp + b*ey;
      double pzn = (2-a)*pz - pzp + b*ez;

      pxp = px;
      pyp = py;
      pzp = pz;

      px = pxn;
      py = pyn;
      pz = pzn;

      Px += px;
      Py += py;
      Pz += pz;

      Pxp += pxp;
      Pyp += pyp;
      Pzp += pzp;
    }

//    double E2 = ex*ex + ey*ey + ez*ez;

    // after this Dx, Dy and Dz actually contain D - P_L

    double Dx = /* (eps + chi*E2)* */ ex + Pxp - Px
      + dt*(
          (Bz(i,j,k) - Bz(i,j-1,k))/dy
        - (By(i,j,k) - By(i,j,k-1))/dz
      );

    double Dy = /* (eps + chi*E2)* */ ey + Pyp - Py
      + dt*(
          (Bx(i,j,k) - Bx(i,j,k-1))/dz
        - (Bz(i,j,k) - Bz(i-1,j,k))/dx
      );

    double Dz = /* (eps + chi*E2)* */ ez + Pzp - Pz
      + dt*(
          (By(i,j,k) - By(i-1,j,k))/dx
        - (Bx(i,j,k) - Bx(i,j-1,k))/dy
      );


//    double E = sqrt(E2);
//    double D = sqrt(Dx*Dx + Dy*Dy + Dz*Dz);
//
//    // Scalar Newton iteration because (D-P) parallel to E
//    double Eold;
//    do {
//      Eold = E;
//      E2 = E*E;
//      E = (2*chi*E2*E + D) / (3*chi*E2 + eps);
//    } while (fabs(E-Eold) > 1e-9);
//
//    Ex(i,j,k) = E*Dx/D;
//    Ey(i,j,k) = E*Dy/D;
//    Ez(i,j,k) = E*Dz/D;
```

```
    Ex(i,j,k) = Dx;
    Ey(i,j,k) = Dy;
    Ez(i,j,k) = Dz;

  }
  storage->applyBoundary("E");
}


void FDTD_Dispersion::stepB(double dt)
{
  DataGrid &Ex = *pEx;
  DataGrid &Ey = *pEy;
  DataGrid &Ez = *pEz;

  DataGrid &Bx = *pBx;
  DataGrid &By = *pBy;
  DataGrid &Bz = *pBz;

  GridIndex low = storage->getLow();
  GridIndex high = storage->getHigh();

  double dx = storage->getDx();
  double dy = storage->getDy();
  double dz = storage->getDz();

  for (int i=low[0]; i<high[0]; ++i)
    for (int j=low[1]; j<high[1]; ++j)
      for (int k=low[2]; k<high[2]; ++k)
  {
    Bx(i,j,k) = Bx(i,j,k)
      + dt*(
          (Ey(i,j,k+1) - Ey(i,j,k))/dz
        - (Ez(i,j+1,k) - Ez(i,j,k))/dy
      );

    By(i,j,k) = By(i,j,k)
      + dt*(
          (Ez(i+1,j,k) - Ez(i,j,k))/dx
        - (Ex(i,j,k+1) - Ex(i,j,k))/dz
      );

    Bz(i,j,k) = Bz(i,j,k)
      + dt*(
          (Ex(i,j+1,k) - Ex(i,j,k))/dx
        - (Ey(i+1,j,k) - Ey(i,j,k))/dx
      );
  }
  storage->applyBoundary("B");
}


ParameterMap* FDTD_Dispersion::MakeParamMap (ParameterMap* pm)
{
  pm = FieldSolver::MakeParamMap(pm);

  (*pm)["eps"] = WParameter(new ParameterValue<double>(&eps,1.0));
  (*pm)["chi"] = WParameter(new ParameterValue<double>(&chi,0.1));

  (*pm)["L1eps2"] = WParameter(new ParameterValue<double>(&LEps2[0],0.0));
  (*pm)["L2eps2"] = WParameter(new ParameterValue<double>(&LEps2[1],0.0));
  (*pm)["L3eps2"] = WParameter(new ParameterValue<double>(&LEps2[2],0.0));

  (*pm)["L1Om2"] = WParameter(new ParameterValue<double>(&LOm2[0],0.0));
  (*pm)["L2Om2"] = WParameter(new ParameterValue<double>(&LOm2[1],0.0));
```

```
  (*pm)["L3Om2"] = WParameter(new ParameterValue<double>(&LOm2[2],0.0));

//  (*pm)["plasma"] = WParameter(
//       new ParameterRebuild<PlasmaCurrent, Current>(&currents)
//  );

  return pm;
}
```

```
#include "fdtd_nl.h"
#include "storage.h"
#include <cmath>

void FDTD_Nonlinear::initStorage(Storage *storage_)
{
  storage = storage_;

  pEx = storage->addGrid("Ex");
  pEy = storage->addGrid("Ey");
  pEz = storage->addGrid("Ez");

  pBx = storage->addGrid("Bx");
  pBy = storage->addGrid("By");
  pBz = storage->addGrid("Bz");

  storage->addToGroup("E", "Ex");
  storage->addToGroup("E", "Ey");
  storage->addToGroup("E", "Ez");

  storage->addToGroup("B", "Bx");
  storage->addToGroup("B", "By");
  storage->addToGroup("B", "Bz");
}

void FDTD_Nonlinear::stepSchemeInit(double dt)
{
  stepB(0.5*dt);
}

void FDTD_Nonlinear::stepScheme(double dt)
{
  stepD(dt);
  stepB(dt);
}

void FDTD_Nonlinear::stepD(double dt)
{
  DataGrid &Ex = *pEx;
  DataGrid &Ey = *pEy;
  DataGrid &Ez = *pEz;

  DataGrid &Bx = *pBx;
  DataGrid &By = *pBy;
  DataGrid &Bz = *pBz;

  GridIndex low = storage->getLow();
  GridIndex high = storage->getHigh();

  double dx = storage->getDx();
  double dy = storage->getDy();
  double dz = storage->getDz();

  for (int i=low[0]+1; i<=high[0]; ++i)
    for (int j=low[1]+1; j<=high[1]; ++j)
      for (int k=low[2]+1; k<=high[2]; ++k)
  {
    double ex = Ex(i,j,k);
    double ey = Ey(i,j,k);
    double ez = Ez(i,j,k);

    double E2 = ex*ex + ey*ey + ez*ez;

    double Dx = (eps + chi*E2)*ex
      + dt*(
```

```
          (Bz(i,j,k) - Bz(i,j-1,k))/dy
        - (By(i,j,k) - By(i,j,k-1))/dz
      );

    double Dy = (eps + chi*E2)*ey
      + dt*(
          (Bx(i,j,k) - Bx(i,j,k-1))/dz
        - (Bz(i,j,k) - Bz(i-1,j,k))/dx
      );

    double Dz = (eps + chi*E2)*ez
      + dt*(
          (By(i,j,k) - By(i-1,j,k))/dx
        - (Bx(i,j,k) - Bx(i,j-1,k))/dy
      );

    double E = sqrt(E2);
    double D = sqrt(Dx*Dx + Dy*Dy + Dz*Dz);

    // Newton iteration
    double Eold;
    do {
      Eold = E;
      E2 = E*E;
      E = (2*chi*E2*E + D) / (3*chi*E2 + eps);
    } while (fabs(E-Eold) > 1e-9);

    Ex(i,j,k) = E*Dx/D;
    Ey(i,j,k) = E*Dy/D;
    Ez(i,j,k) = E*Dz/D;

  }
  storage->applyBoundary("E");
}


void FDTD_Nonlinear::stepB(double dt)
{
  DataGrid &Ex = *pEx;
  DataGrid &Ey = *pEy;
  DataGrid &Ez = *pEz;

  DataGrid &Bx = *pBx;
  DataGrid &By = *pBy;
  DataGrid &Bz = *pBz;

  GridIndex low = storage->getLow();
  GridIndex high = storage->getHigh();

  double dx = storage->getDx();
  double dy = storage->getDy();
  double dz = storage->getDz();

  for (int i=low[0]; i<high[0]; ++i)
    for (int j=low[1]; j<high[1]; ++j)
      for (int k=low[2]; k<high[2]; ++k)
  {
    Bx(i,j,k) = Bx(i,j,k)
      + dt*(
          (Ey(i,j,k+1) - Ey(i,j,k))/dz
        - (Ez(i,j+1,k) - Ez(i,j,k))/dy
      );

    By(i,j,k) = By(i,j,k)
      + dt*(
          (Ez(i+1,j,k) - Ez(i,j,k))/dx
```

```
        - (Ex(i,j,k+1) - Ex(i,j,k))/dz
      );

    Bz(i,j,k) = Bz(i,j,k)
      + dt*(
          (Ex(i,j+1,k) - Ex(i,j,k))/dx
        - (Ey(i+1,j,k) - Ey(i,j,k))/dx
      );
  }
  storage->applyBoundary("B");
}


ParameterMap* FDTD_Nonlinear::MakeParamMap (ParameterMap* pm)
{
  pm = FieldSolver::MakeParamMap(pm);

  (*pm)["eps"] = WParameter(new ParameterValue<double>(&eps,1.0));
  (*pm)["chi"] = WParameter(new ParameterValue<double>(&chi,0.1));
  return pm;
}
```

```
#include "fdtd_plain.h"
#include "storage.h"

void FDTD_Plain::initStorage(Storage *storage_)
{
  storage = storage_;

  pEx = storage->addGrid("Ex");
  pEy = storage->addGrid("Ey");
  pEz = storage->addGrid("Ez");

  pBx = storage->addGrid("Bx");
  pBy = storage->addGrid("By");
  pBz = storage->addGrid("Bz");

  storage->addToGroup("E", "Ex");
  storage->addToGroup("E", "Ey");
  storage->addToGroup("E", "Ez");

  storage->addToGroup("B", "Bx");
  storage->addToGroup("B", "By");
  storage->addToGroup("B", "Bz");
}

void FDTD_Plain::stepSchemeInit(double dt)
{
  stepB(0.5*dt);
}

void FDTD_Plain::stepScheme(double dt)
{
  stepD(dt);
  stepB(dt);
}

void FDTD_Plain::stepD(double dt)
{
  DataGrid &Ex = *pEx;
  DataGrid &Ey = *pEy;
  DataGrid &Ez = *pEz;

  DataGrid &Bx = *pBx;
  DataGrid &By = *pBy;
  DataGrid &Bz = *pBz;

  GridIndex low = storage->getLow();
  GridIndex high = storage->getHigh();

  double dx = storage->getDx();
  double dy = storage->getDy();
  double dz = storage->getDz();

  for (int i=low[0]+1; i<=high[0]; ++i)
    for (int j=low[1]+1; j<=high[1]; ++j)
      for (int k=low[2]+1; k<=high[2]; ++k)
  {
    Ex(i,j,k) = Ex(i,j,k)
      + dt*(
          (Bz(i,j,k) - Bz(i,j-1,k))/dy
        - (By(i,j,k) - By(i,j,k-1))/dz
      );

    Ey(i,j,k) = Ey(i,j,k)
      + dt*(
          (Bx(i,j,k) - Bx(i,j,k-1))/dz
```

```
        - (Bz(i,j,k) - Bz(i-1,j,k))/dx
      );

    Ez(i,j,k) = Ez(i,j,k)
      + dt*(
          (By(i,j,k) - By(i-1,j,k))/dx
        - (Bx(i,j,k) - Bx(i,j-1,k))/dy
      );
  }
  storage->applyBoundary("E");
}


void FDTD_Plain::stepB(double dt)
{
  DataGrid &Ex = *pEx;
  DataGrid &Ey = *pEy;
  DataGrid &Ez = *pEz;

  DataGrid &Bx = *pBx;
  DataGrid &By = *pBy;
  DataGrid &Bz = *pBz;

  GridIndex low = storage->getLow();
  GridIndex high = storage->getHigh();

  double dx = storage->getDx();
  double dy = storage->getDy();
  double dz = storage->getDz();

  for (int i=low[0]; i<high[0]; ++i)
    for (int j=low[1]; j<high[1]; ++j)
      for (int k=low[2]; k<high[2]; ++k)
  {
    Bx(i,j,k) = Bx(i,j,k)
      + dt*(
          (Ey(i,j,k+1) - Ey(i,j,k))/dz
        - (Ez(i,j+1,k) - Ez(i,j,k))/dy
      );

    By(i,j,k) = By(i,j,k)
      + dt*(
          (Ez(i+1,j,k) - Ez(i,j,k))/dx
        - (Ex(i,j,k+1) - Ex(i,j,k))/dz
      );

    Bz(i,j,k) = Bz(i,j,k)
      + dt*(
          (Ex(i,j+1,k) - Ex(i,j,k))/dx
        - (Ey(i+1,j,k) - Ey(i,j,k))/dx
      );
  }
  storage->applyBoundary("B");
}
```

```
#include "fdtd_plrc.h"
#include "storage.h"
#include <cmath>
#include <complex>
#include <algorithm>


//=============================================================
//==========  FDTD_PLRCCore
//=============================================================

void FDTD_PLRCCore::coreInitStorage(Storage *storage_)
{
  storage = storage_;

  pEx = storage->addGrid("Ex");
  pEy = storage->addGrid("Ey");
  pEz = storage->addGrid("Ez");

  pBx = storage->addGrid("Bx");
  pBy = storage->addGrid("By");
  pBz = storage->addGrid("Bz");

  pKappaEdx = storage->addLine("KappaEdx", 0);
  pKappaEdy = storage->addLine("KappaEdy", 1);
  pKappaEdz = storage->addLine("KappaEdz", 2);

  pStretchEax = storage->addLine("StretchEax", 0);
  pStretchEay = storage->addLine("StretchEay", 1);
  pStretchEaz = storage->addLine("StretchEaz", 2);

  pStretchEbx = storage->addLine("StretchEbx", 0);
  pStretchEby = storage->addLine("StretchEby", 1);
  pStretchEbz = storage->addLine("StretchEbz", 2);

  pKappaHdx = storage->addLine("KappaHdx", 0);
  pKappaHdy = storage->addLine("KappaHdy", 1);
  pKappaHdz = storage->addLine("KappaHdz", 2);

  pStretchHax = storage->addLine("StretchHax", 0);
  pStretchHay = storage->addLine("StretchHay", 1);
  pStretchHaz = storage->addLine("StretchHaz", 2);

  pStretchHbx = storage->addLine("StretchHbx", 0);
  pStretchHby = storage->addLine("StretchHby", 1);
  pStretchHbz = storage->addLine("StretchHbz", 2);

  pPsiRx[0] = storage->addGrid("PsiR1x");
  pPsiRy[0] = storage->addGrid("PsiR1y");
  pPsiRz[0] = storage->addGrid("PsiR1z");

  pPsiRx[1] = storage->addGrid("PsiR2x");
  pPsiRy[1] = storage->addGrid("PsiR2y");
  pPsiRz[1] = storage->addGrid("PsiR2z");

  pPsiRx[2] = storage->addGrid("PsiR3x");
  pPsiRy[2] = storage->addGrid("PsiR3y");
  pPsiRz[2] = storage->addGrid("PsiR3z");

  pPsiIx[0] = storage->addGrid("PsiI1x");
  pPsiIy[0] = storage->addGrid("PsiI1y");
  pPsiIz[0] = storage->addGrid("PsiI1z");

  pPsiIx[1] = storage->addGrid("PsiI2x");
  pPsiIy[1] = storage->addGrid("PsiI2y");
```

```
  pPsiIz[1] = storage->addGrid("PsiI2z");

  pPsiIx[2] = storage->addGrid("PsiI3x");
  pPsiIy[2] = storage->addGrid("PsiI3y");
  pPsiIz[2] = storage->addGrid("PsiI3z");

  storage->addToGroup("E", "Ex");
  storage->addToGroup("E", "Ey");
  storage->addToGroup("E", "Ez");

  storage->addToGroup("B", "Bx");
  storage->addToGroup("B", "By");
  storage->addToGroup("B", "Bz");

  std::for_each(currents.begin(), currents.end(),  InitStorageFunctor<Current>(storage));
  std::remove_if(currents.begin(), currents.end(), Current::CurrentInvalidPredicate());

  std::for_each(magCurrents.begin(), magCurrents.end(),  InitStorageFunctor<Current>(storag
e));
  std::remove_if(magCurrents.begin(), magCurrents.end(), Current::CurrentInvalidPredicate()
);

}

//=============================================================
//==========  FDTD_PLRCLinCore
//=============================================================

void FDTD_PLRCLinCore::plrcStepD(double dt,
                                 int i, int j, int k,
                                 double dx, double dy, double dz,
                                 double Jx, double Jy, double Jz)
{
  double &ex = (*pEx)(i,j,k);
  double &ey = (*pEy)(i,j,k);
  double &ez = (*pEz)(i,j,k);

  double kappaEdx = (*pKappaEdx)(i)*dx;
  double kappaEdy = (*pKappaEdy)(j)*dy;
  double kappaEdz = (*pKappaEdz)(k)*dz;

  double stretchEax = (*pStretchEax)(i);
  double stretchEay = (*pStretchEay)(j);
  double stretchEaz = (*pStretchEaz)(k);

  double stretchEbx = (*pStretchEbx)(i);
  double stretchEby = (*pStretchEby)(j);
  double stretchEbz = (*pStretchEbz)(k);

  double Psix = 0;
  double Psiy = 0;
  double Psiz = 0;

  for (int n=0;n<3;++n)
  {
    double &pxr = pPsiRx[n]->operator()(i,j,k);
    double &pyr = pPsiRy[n]->operator()(i,j,k);
    double &pzr = pPsiRz[n]->operator()(i,j,k);

    double &pxi = pPsiIx[n]->operator()(i,j,k);
    double &pyi = pPsiIy[n]->operator()(i,j,k);
    double &pzi = pPsiIz[n]->operator()(i,j,k);

    std::complex<double> D = plrcData.dchi0[n]-plrcData.dxi0[n];

    std::complex<double> px = D*ex + std::complex<double>(pxr,pxi);
```

```
    std::complex<double> py = D*ey + std::complex<double>(pyr,pyi);
    std::complex<double> pz = D*ez + std::complex<double>(pzr,pzi);

    Psix += std::real(px);
    Psiy += std::real(py);
    Psiz += std::real(pz);

    px = plrcData.dxi0[n]*ex + plrcData.Crec[n]*px;
    py = plrcData.dxi0[n]*ey + plrcData.Crec[n]*py;
    pz = plrcData.dxi0[n]*ez + plrcData.Crec[n]*pz;

    pxr = std::real(px);
    pyr = std::real(py);
    pzr = std::real(pz);

    pxi = std::imag(px);
    pyi = std::imag(py);
    pzi = std::imag(pz);

  }

  double denomX = eps + stretchEbx*(plrcData.sumChi0 - plrcData.sumXi0);
  double denomY = eps + stretchEby*(plrcData.sumChi0 - plrcData.sumXi0);
  double denomZ = eps + stretchEbz*(plrcData.sumChi0 - plrcData.sumXi0);

  double numerX = stretchEax*eps - stretchEbx*plrcData.sumXi0;
  double numerY = stretchEay*eps - stretchEby*plrcData.sumXi0;
  double numerZ = stretchEaz*eps - stretchEbz*plrcData.sumXi0;

//    double E2 = ex*ex + ey*ey + ez*ez;

  // after this Dx, Dy and Dz actually contain D - P_L

//  if (fabs(Jx+Jy+Jz) > 1e-40) std::cerr << "Current " << Jx << " " << Jy << " " << Jz <<
std::endl;

  double exn =
    (
      numerX*ex
      + stretchEbx*(
          dt*(
            ((*pBz)(i,j,k) - (*pBz)(i,j-1,k))/kappaEdy
          - ((*pBy)(i,j,k) - (*pBy)(i,j,k-1))/kappaEdz
          + Jx
        )
        + Psix
      )
    ) / denomX;

  double eyn =
    (
      numerY*ey
      + stretchEby*(
          dt*(
            ((*pBx)(i,j,k) - (*pBx)(i,j,k-1))/kappaEdz
          - ((*pBz)(i,j,k) - (*pBz)(i-1,j,k))/kappaEdx
          + Jy
        )
        + Psiy
      )
    ) / denomY;

  double ezn =
    (
      numerZ*ez
      + stretchEbz*(
```

```
          dt*(
            ((*pBy)(i,j,k) - (*pBy)(i-1,j,k))/kappaEdx
          - ((*pBx)(i,j,k) - (*pBx)(i,j-1,k))/kappaEdy
          + Jz
        )
        + Psiz
      )
    ) / denomZ;

  ex = exn;
  ey = eyn;
  ez = ezn;
}

void FDTD_PLRCLinCore::plrcStepB(double dt,
                                 int i, int j, int k,
                                 double dx, double dy, double dz,
                                 double Jx, double Jy, double Jz)
{

  double kappaHdx = (*pKappaHdx)(i)*dx;
  double kappaHdy = (*pKappaHdy)(j)*dy;
  double kappaHdz = (*pKappaHdz)(k)*dz;

  double stretchHax = (*pStretchHax)(i);
  double stretchHay = (*pStretchHay)(j);
  double stretchHaz = (*pStretchHaz)(k);

  double stretchHbx = (*pStretchHbx)(i);
  double stretchHby = (*pStretchHby)(j);
  double stretchHbz = (*pStretchHbz)(k);


  (*pBx)(i,j,k) = stretchHax*(*pBx)(i,j,k)
    + stretchHbx*dt*(
        ((*pEy)(i,j,k+1) - (*pEy)(i,j,k))/kappaHdz
      - ((*pEz)(i,j+1,k) - (*pEz)(i,j,k))/kappaHdy
     - Jx
    );

  (*pBy)(i,j,k) = stretchHay*(*pBy)(i,j,k)
    + stretchHby*dt*(
        ((*pEz)(i+1,j,k) - (*pEz)(i,j,k))/kappaHdx
      - ((*pEx)(i,j,k+1) - (*pEx)(i,j,k))/kappaHdz
     - Jy
    );

  (*pBz)(i,j,k) = stretchHaz*(*pBz)(i,j,k)
    + stretchHbz*dt*(
        ((*pEx)(i,j+1,k) - (*pEx)(i,j,k))/kappaHdy
      - ((*pEy)(i+1,j,k) - (*pEy)(i,j,k))/kappaHdx
     - Jz
    );
}

ParameterMap* FDTD_PLRCLinCore::CustomParamMap (ParameterMap* pm)
{
  return pm;
}

//=============================================================
//==========  FDTD_PLRCNonlinCore
//=============================================================

void FDTD_PLRCNonlinCore::plrcStepD(double dt,
                                    int i, int j, int k,
```

```
                                    double dx, double dy, double dz,
                                    double Jx, double Jy, double Jz)
{
  double &ex = (*pEx)(i,j,k);
  double &ey = (*pEy)(i,j,k);
  double &ez = (*pEz)(i,j,k);

  double kappaEdx = (*pKappaEdx)(i)*dx;
  double kappaEdy = (*pKappaEdy)(j)*dy;
  double kappaEdz = (*pKappaEdz)(k)*dz;

  double Psix = 0;
  double Psiy = 0;
  double Psiz = 0;

  for (int n=0;n<3;++n)
  {
    double &pxr = pPsiRx[n]->operator()(i,j,k);
    double &pyr = pPsiRy[n]->operator()(i,j,k);
    double &pzr = pPsiRz[n]->operator()(i,j,k);

    double &pxi = pPsiIx[n]->operator()(i,j,k);
    double &pyi = pPsiIy[n]->operator()(i,j,k);
    double &pzi = pPsiIz[n]->operator()(i,j,k);

    std::complex<double> D = plrcData.dchi0[n]-plrcData.dxi0[n];

    std::complex<double> px = D*ex + std::complex<double>(pxr,pxi);
    std::complex<double> py = D*ey + std::complex<double>(pyr,pyi);
    std::complex<double> pz = D*ez + std::complex<double>(pzr,pzi);

    Psix += std::real(px);
    Psiy += std::real(py);
    Psiz += std::real(pz);

    px = plrcData.dxi0[n]*ex + plrcData.Crec[n]*px;
    py = plrcData.dxi0[n]*ey + plrcData.Crec[n]*py;
    pz = plrcData.dxi0[n]*ez + plrcData.Crec[n]*pz;

    pxr = std::real(px);
    pyr = std::real(py);
    pzr = std::real(pz);

    pxi = std::imag(px);
    pyi = std::imag(py);
    pzi = std::imag(pz);

  }


  double denom = eps + plrcData.sumChi0 - plrcData.sumXi0;
  double numer = eps - plrcData.sumXi0;

  double E2 = ex*ex + ey*ey + ez*ez;

//    double E2 = ex*ex + ey*ey + ez*ez;

  // after this Dx, Dy and Dz actually contain D - P_L

  double cx =
    (
      (numer + chi*E2)*ex

      + dt*(
          ((*pBz)(i,j,k) - (*pBz)(i,j-1,k))/kappaEdy
        - ((*pBy)(i,j,k) - (*pBy)(i,j,k-1))/kappaEdz
```

```
        + Jx
      )
      + Psix
    ) / denom;

  double cy =
    (
      (numer + chi*E2)*ey
      + dt*(
          ((*pBx)(i,j,k) - (*pBx)(i,j,k-1))/kappaEdz
        - ((*pBz)(i,j,k) - (*pBz)(i-1,j,k))/kappaEdx
        + Jy
      )
      + Psiy
    ) / denom;

  double cz =
    (
      (numer + chi*E2)*ez
      + dt*(
          ((*pBy)(i,j,k) - (*pBy)(i-1,j,k))/kappaEdx
        - ((*pBx)(i,j,k) - (*pBx)(i,j-1,k))/kappaEdy
         + Jz
      )
      + Psiz
    ) / denom;

  double E = sqrt(E2);
  double A = chi/denom;
  double C = sqrt(cx*cx + cy*cy + cz*cz);

  // Newton iteration
  double Eold;
  do {
    Eold = E;
    E2 = E*E;
    E = (2*A*E2*E + C) / (3*A*E2 + 1);
  } while (fabs(E-Eold) > 1e-9);

  ex = E*cx/C;
  ey = E*cy/C;
  ez = E*cz/C;

}

void FDTD_PLRCNonlinCore::plrcStepB(double dt,
                                    int i, int j, int k,
                                    double dx, double dy, double dz,
                                    double Jx, double Jy, double Jz)
{

  double kappaHdx = (*pKappaHdx)(i)*dx;
  double kappaHdy = (*pKappaHdy)(j)*dy;
  double kappaHdz = (*pKappaHdz)(k)*dz;

  (*pBx)(i,j,k) = (*pBx)(i,j,k)
    + dt*(
        ((*pEy)(i,j,k+1) - (*pEy)(i,j,k))/kappaHdz
      - ((*pEz)(i,j+1,k) - (*pEz)(i,j,k))/kappaHdy
      + Jx
    );

  (*pBy)(i,j,k) = (*pBy)(i,j,k)
    + dt*(
        ((*pEz)(i+1,j,k) - (*pEz)(i,j,k))/kappaHdx
      - ((*pEx)(i,j,k+1) - (*pEx)(i,j,k))/kappaHdz
```

```
       + Jy
    );

  (*pBz)(i,j,k) = (*pBz)(i,j,k)
    + dt*(
        ((*pEx)(i,j+1,k) - (*pEx)(i,j,k))/kappaHdy
      - ((*pEy)(i+1,j,k) - (*pEy)(i,j,k))/kappaHdx
       + Jz
    );
}

ParameterMap* FDTD_PLRCNonlinCore::CustomParamMap (ParameterMap* pm)
{
  (*pm)["chi"] = WParameter(new ParameterValue<double>(&chi,0.1));
  return pm;
}
```

```
#include "fielddiag.h"
#include "process.h"
#include "globals.h"
#include "boundary.h"

ParameterMap* FieldDiag::MakeParamMap (ParameterMap* pm)
{
  pm = ParentType::MakeParamMap(pm);

  (*pm)["field"] = WParameter(new ParameterValue<std::string>(&fieldId,""));
  return pm;
}

void FieldDiag::fetchField(Storage &storage)
{
  this->setField(&(storage.getGrid(fieldId)));
}

FieldEnergyDiag::FieldEnergyDiag() : storage(0) {}

FieldEnergyDiag::~FieldEnergyDiag() {}

void FieldEnergyDiag::setStorage(Storage *storage_)
{
  storage = storage_;
}

void FieldEnergyDiag::open(const std::string &fname){
  output.open(fname.c_str());
}


void FieldEnergyDiag::write()
{
  int time = Process::instance().getTime();

  output << time << " " << energy << std::endl;
}

void FieldEnergyDiag::close()
{
  output.close();
}

void FieldEnergyDiag::calculate()
{
  DataGrid &Ex = storage->getGrid("Ex");
  DataGrid &Ey = storage->getGrid("Ey");
  DataGrid &Ez = storage->getGrid("Ez");
  DataGrid &Bx = storage->getGrid("Bx");
  DataGrid &By = storage->getGrid("By");
  DataGrid &Bz = storage->getGrid("Bz");

  GridIndex low = storage->getLow();
  GridIndex high = storage->getHigh();

  energy = 0;

  for (int i=low[0]+1; i<high[0]; ++i)
    for (int j=low[1]+1; j<high[1]; ++j)
      for (int k=low[2]+1; k<high[2]; ++k)
      {
        double ex = Ex(i,j,k);
        double ey = Ey(i,j,k);
        double ez = Ez(i,j,k);
        double bx = Bx(i,j,k);
```

```
        double by = By(i,j,k);
        double bz = Bz(i,j,k);
        energy += ex*ex + ey*ey + ez*ez + bx*bx + by*by + bz*bz;
      }

  const Boundary &bound = storage->getBoundary();
  energy = bound.SumReduce(energy);
  Globals &glob = Globals::instance();
  energy *= glob.gridDX() * glob.gridDY() * glob.gridDZ();
}
```

```
#include "fieldsim.h"

#include "fdtd_plain.h"
#include "fdtd_nl.h"
#include "fdtd_disp.h"
#include "fdtd_plrc.h"

#include "globals.h"
#include "process.h"
#include "periodic_bound.h"
#include "mpi_bound.h"

#include "waveinit.h"
#include "gaussinit.h"
#include "pulseinit.h"

#include "freqdiag.h"



FieldSimulation::FieldSimulation()
  : initializer(0)
{

}

FieldSimulation::~FieldSimulation()
{
}

void FieldSimulation::init()
{
  this->boundary->init();

  dt = Globals::instance().dt();

  std::cout << "                              *** RESIZING ***" << std::endl;

  //resize
  resize(this->boundary->RegionLow(),this->boundary->RegionHigh());
  solver->initStorage(this);

  //get the global grid spacing
  this->dx = Globals::instance().gridDX();
  this->dy = Globals::instance().gridDY();
  this->dz = Globals::instance().gridDZ();

  typedef AllFieldDiag::DiagList::iterator Iter;
  typedef AllFieldDiag::ExtraDiagList::iterator ExtraIter;

  for (
    Iter it = fieldDiag.fields.begin();
    it != fieldDiag.fields.end();
    ++it
  )
  {
    (*it)->fetchField(*this);
  }

  for (
    ExtraIter xit = fieldDiag.fieldextras.begin();
    xit != fieldDiag.fieldextras.end();
    ++xit
  )
  {
    (*xit)->setStorage(this);
```

```
    (*xit)->init();
  }

  if(!initializer)
  {
    std::cout << "  NO INITIALIZER SPECIFIED!!!" << std::endl;
    std::cout << "    Starting with zero fields" << std::endl;
  }
  else {
    initializer->init(*this);
    if (!Globals::instance().isRestart())
      solver->stepSchemeInit(dt);
  }
}

void FieldSimulation::execute()
{
  solver->stepScheme(dt);
}

ParameterMap* FieldSimulation::MakeParamMap (ParameterMap* pm) {
  pm = Rebuildable::MakeParamMap(pm);

  (*pm)["fdtd-plain"] = WParameter(
      new ParameterRebuild<FDTD_Plain, FieldSolver>(&solver)
  );

  (*pm)["fdtd-nl"] = WParameter(
      new ParameterRebuild<FDTD_Nonlinear, FieldSolver>(&solver)
  );

  (*pm)["fdtd-disp"] = WParameter(
      new ParameterRebuild<FDTD_Dispersion, FieldSolver>(&solver)
  );

  (*pm)["fdtd-plrc-lin"] = WParameter(
      new ParameterRebuild<FDTD_PLRCLin, FieldSolver>(&solver)
  );

  (*pm)["fdtd-plrc-nonlin"] = WParameter(
      new ParameterRebuild<FDTD_PLRCNonlin, FieldSolver>(&solver)
  );


  (*pm)["single_periodic"] = WParameter(
      new ParameterRebuild<SinglePeriodicBoundary, Boundary>(&boundary)
  );

#ifndef SINGLE_PROCESSOR
  (*pm)["mpi_slice_periodic"] = WParameter(
      new ParameterRebuild<MPIPeriodicSplitXBoundary, Boundary>(&boundary)
  );

  (*pm)["mpi_block_periodic"] = WParameter(
      new ParameterRebuild<MPIPeriodicSplitXYZBoundary, Boundary>(&boundary)
  );
#endif


  (*pm)["wave_init"] = WParameter(
      new ParameterRebuild<PlaneWaveInit, FieldSimInit>(&initializer)
  );

  (*pm)["gauss_init"] = WParameter(
      new ParameterRebuild<PlaneGaussPulseInit, FieldSimInit>(&initializer)
  );
```

```
  (*pm)["pulse_init"] = WParameter(
      new ParameterRebuild<GaussPulseInit, FieldSimInit>(&initializer)
  );

  (*pm)["fielddiag"] = WParameter(
      new ParameterRebuild<FieldDiag, FieldDiag>(&fieldDiag.fields)
  );

  (*pm)["energy"] = WParameter(
      new ParameterRebuild<FieldEnergyDiag, FieldExtraDiag>(&fieldDiag.fieldextras)
  );

  (*pm)["frequency"] = WParameter(
      new ParameterRebuild<FrequencyDiag, FieldExtraDiag>(&fieldDiag.fieldextras)
  );

  return pm;
}
```

```
#include "freqdiag.h"
#include "process.h"
#include "globals.h"
#include "boundary.h"

FrequencyDiag::FrequencyDiag() : storage(0) {}

FrequencyDiag::~FrequencyDiag() {}

void FrequencyDiag::setStorage(Storage *storage_)
{
  storage = storage_;
}

void FrequencyDiag::open(const std::string &fname){
  output.open(fname.c_str());
}


void FrequencyDiag::write()
{
  if (count>lastcount)
  {
    lastcount = count;
    int time = Process::instance().getTime();
    output << time << " " << frequency << std::endl;
  }
}

void FrequencyDiag::close()
{
  output.close();
}

void FrequencyDiag::init()
{
  std::cerr << "FrequencyDiag::init()\n";
  dt=Globals::instance().dt();
  count = -1;
  lastcount = 0;
  lastval = 0;
}

void FrequencyDiag::calculate()
{
  if (!storage->hasGrid(field)) return;

  DataGrid &G = storage->getGrid(field);
  double val = 0.15*G(x,y,z) + 0.85*lastval;

  if ((val*lastval<=0) && (fabs(val)*fabs(lastval)>0))
  {
    double d = lastval/(lastval-val);
    double time = (Process::instance().getTime() - 1 + d)*dt;
    if (count<0)
    {
      firstzero = time;
      count = 0;
    }
    else
    {
      ++count;
      frequency = (0.5*count) / (time-firstzero);
    }

  }
```

```
  lastval = val;

}

ParameterMap* FrequencyDiag::MakeParamMap (ParameterMap* pm) {
  pm = FieldExtraDiag::MakeParamMap(pm);

  (*pm)["field"] = WParameter(
    new ParameterValue<std::string>(&field, "Ex")
  );

  (*pm)["x"] = WParameter(
    new ParameterValue<int>(&x, 1)
  );

  (*pm)["y"] = WParameter(
    new ParameterValue<int>(&y, 1)
  );

  (*pm)["z"] = WParameter(
    new ParameterValue<int>(&z, 1)
  );

  return pm;
}
```

```
#include "gaussinit.h"
#include "globals.h"
#include "storage.h"



void PlaneGaussPulseInit::init(Storage &fields)
{
  std::cerr << "=== INITIALIZING Gaussian Pulse ===\n";
  double ex = ky*bz-kz*by;
  double ey = kz*bx-kx*bz;
  double ez = kx*by-ky*bx;
  double bmag = sqrt(bx*bx + by*by + bz*bz);
  double factor = -bmag/sqrt(ex*ex + ey*ey + ez*ez);

  ex *= factor;
  ey *= factor;
  ez *= factor;

  GridIndex gridLow = Globals::instance().gridLow();
  GridIndex gridHigh = Globals::instance().gridHigh();

  DataGrid &Ex = fields.getGrid("Ex");
  DataGrid &Ey = fields.getGrid("Ey");
  DataGrid &Ez = fields.getGrid("Ez");
  DataGrid &Bx = fields.getGrid("Bx");
  DataGrid &By = fields.getGrid("By");
  DataGrid &Bz = fields.getGrid("Bz");

  GridIndex low = fields.getLow();
  GridIndex high = fields.getHigh();

  double mx = 0.5*double(gridHigh[0] + gridLow[0]);
  double my = 0.5*double(gridHigh[1] + gridLow[1]);
  double mz = 0.5*double(gridHigh[2] + gridLow[2]);

  double nkx = kx/double(gridHigh[0] - gridLow[0] - 1);
  double nky = ky/double(gridHigh[1] - gridLow[1] - 1);
  double nkz = kz/double(gridHigh[2] - gridLow[2] - 1);

  int offset = 11;

  for (int i=low[0]+offset; i<=high[0]-offset; ++i)
    for (int j=low[1]+offset; j<=high[1]-offset; ++j)
      for (int k=low[2]+offset; k<=high[2]-offset; ++k)
      {
        double delta = 0.5;

        double pose = (i-mx)*(i-mx)*nkx*nkx
                    + (j-my)*(j-my)*nky*nky
                    + (k-mz)*(k-mz)*nkz*nkz;
        double posb = (i+delta-mx)*(i+delta-mx)*nkx*nkx
                    + (j+delta-my)*(j+delta-my)*nky*nky
                    + (k+delta-mz)*(k+delta-mz)*nkz*nkz;

        double ampe = exp(-pose);
        double ampb = exp(-posb);

/*        double pose = (i-mx)*nkx
                    + (j-my)*nky
                    + (k-mz)*nkz;
        double posb = (i+delta-mx)*nkx
                    + (j+delta-my)*nky
                    + (k+delta-mz)*nkz;

        double ampe = exp(-pose*pose);
```

```
        double ampb = exp(-posb*posb);
*/
        Ex(i,j,k) = ex*ampe;
        Ey(i,j,k) = ey*ampe;
        Ez(i,j,k) = ez*ampe;

        Bx(i,j,k) = bx*ampb;
        By(i,j,k) = by*ampb;
        Bz(i,j,k) = bz*ampb;
      }

}


ParameterMap* PlaneGaussPulseInit::MakeParamMap (ParameterMap* pm) {
  pm = Rebuildable::MakeParamMap(pm);
  (*pm)["kx"] = WParameter(new ParameterValue<int>(&kx,1));
  (*pm)["ky"] = WParameter(new ParameterValue<int>(&ky,1));
  (*pm)["kz"] = WParameter(new ParameterValue<int>(&kz,1));
  (*pm)["Bx"] = WParameter(new ParameterValue<double>(&bx,1));
  (*pm)["By"] = WParameter(new ParameterValue<double>(&by,1));
  (*pm)["Bz"] = WParameter(new ParameterValue<double>(&bz,1));

  return pm;
}
```

```
#include "globals.h"

//---------------------------------------------------------------------------
Globals *Globals::globals;
int Globals::Argc;
char **Globals::Argv;
//---------------------------------------------------------------------------
Globals::Globals() {
  globals = this;
  initialized = false;
  IsRestart = false;
  master = true;
  uniqueId = 0;
}
//---------------------------------------------------------------------------
void Globals::init()
{
  initialized = true;
}
//---------------------------------------------------------------------------
ParameterMap* Globals::MakeParamMap (ParameterMap* pm) {
  pm = Rebuildable::MakeParamMap(pm);
  (*pm)["grid-x"] = WParameter(new ParameterValue<int>(&GridX,128));
  (*pm)["grid-y"] = WParameter(new ParameterValue<int>(&GridY,128));
  (*pm)["grid-z"] = WParameter(new ParameterValue<int>(&GridZ,128));

  (*pm)["grid-dx"] = WParameter(new ParameterValue<double>(&GridDX,0.1));
  (*pm)["grid-dy"] = WParameter(new ParameterValue<double>(&GridDY,0.1));
  (*pm)["grid-dz"] = WParameter(new ParameterValue<double>(&GridDZ,0.1));

  (*pm)["dt"] = WParameter(new ParameterValue<double>(&DT,1e-2));

  (*pm)["T-total"] = WParameter(new ParameterValue<int>(&TotalTime,100000));

  return pm;
}

//---------------------------------------------------------------------------

std::string Globals::Rebuild(std::istream& in)
{
  std::string strToken = Rebuildable::Rebuild(in);

  GridLow = GridIndex(0,0,0);
  GridHigh = GridIndex(GridX+1,GridY+1,GridZ+1);

  return strToken;
}
```

```
#include "hdfstream.h"


HDFstream::HDFstream()
  : file_id(-1),
    status(0),
    blockname("data"),
    sets_count(0)
{}

HDFstream::HDFstream(const HDFstream& hdf)
  : file_id(hdf.file_id),
    status(hdf.status),
    blockname(hdf.blockname),
    sets_count(hdf.sets_count)
{}

HDFstream &HDFstream::operator=(const HDFstream& hdf)
{
  file_id = hdf.file_id;
  status = hdf.status;
  sets_count = hdf.sets_count;
  blockname = hdf.blockname;
  return *this;
}

HDFstream::~HDFstream()
{
  close();
}

void HDFstream::close()
{
  if (file_id >= 0) {
    H5Fclose (file_id);
  }
  file_id = -1;
}

bool HDFstream::good() const
{
  return (file_id>=0);
}

void HDFstream::setBlockName(std::string blockname_)
{
  blockname = blockname_;
  sets_count = -1;
}

std::string HDFstream::getNextBlockName()
{
  std::ostringstream bname;
  bname << blockname;

  if (sets_count<0) sets_count = 1;
  else
  {
    bname << sets_count++;
  }
  return bname.str();
}

// ---------------------------------------------------------------------

HDFistream::HDFistream()
```

```
  : HDFstream() {}

HDFistream::HDFistream(const char* fname)
  : HDFstream()
{
  open(fname);
}

HDFistream::HDFistream(const HDFistream& hdf)
  : HDFstream(hdf)
{}

int HDFistream::open(const char* fname)
{
  close();

  file_id = H5Fopen (fname, H5F_ACC_RDONLY, H5P_DEFAULT);

  sets_count = 0;
  return 1;
}


// ---------------------------------------------------------------------

HDFostream::HDFostream()
   : HDFstream()
{}

HDFostream::HDFostream(const HDFostream& hdf)
  : HDFstream(hdf)
{}

HDFostream::HDFostream(const char* fname)
   : HDFstream()
{
  open(fname);
}

int HDFostream::open(const char* fname)
{
  sets_count = 0;
  file_id = H5Fcreate (fname, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
  return file_id;
}

// ---------------------------------------------------------------------

template<>
const hid_t H5DataType<int>::type = H5T_NATIVE_INT;

template<>
const hid_t H5DataType<float>::type = H5T_NATIVE_FLOAT;

template<>
const hid_t H5DataType<double>::type = H5T_NATIVE_DOUBLE;
```

```
#include "mpulse.h"
#include "incsource.h"
#include "fieldsolver.h"
#include "globals.h"
#include "storage.h"

#include <vector>

//=============================================================
//==========  IncidentSource
//=============================================================


void IncidentSource::initCurrents(Storage *storage, FieldSolver *solver)
{
  solver->addCurrent(
    makeECurrent(distance, north)
  );
  solver->addCurrent(
    makeECurrent(distance, south)
  );
  solver->addCurrent(
    makeECurrent(distance, east)
  );
  solver->addCurrent(
    makeECurrent(distance, west)
  );
  solver->addCurrent(
    makeECurrent(distance, up)
  );
  solver->addCurrent(
    makeECurrent(distance, down)
  );

  solver->addMagCurrent(
    makeHCurrent(distance, north)
  );
  solver->addMagCurrent(
    makeHCurrent(distance, south)
  );
  solver->addMagCurrent(
    makeHCurrent(distance, east)
  );
  solver->addMagCurrent(
    makeHCurrent(distance, west)
  );
  solver->addMagCurrent(
    makeHCurrent(distance, up)
  );
  solver->addMagCurrent(
    makeHCurrent(distance, down)
  );
}


ParameterMap* IncidentSource::MakeParamMap (ParameterMap* pm)
{
  pm = CurrentSim::MakeParamMap(pm);

  (*pm)["d"] = WParameter(new ParameterValue<int>(&this->distance,8));
  return pm;
}

//=============================================================
//==========  CPMLBorderCurrent
//=============================================================
```

```
IncidentSourceCurrent::IncidentSourceCurrent(int distance_, Direction dir_, bool isH_)
  : distance(distance_), dir(dir_), isH(isH_)
{
  switch (dir)
  {
    case east:
    case west:  dim = 0;
                transverse1 = 1;
                transverse2 = 2;
                break;
    case north:
    case south: dim = 1;
                transverse1 = 2;
                transverse2 = 0;
                break;
    case up:
    case down:  dim = 2;
                transverse1 = 0;
                transverse2 = 1;
                break;
  }
}

//=============================================================
//==========  IncidentSourceECurrent
//=============================================================

IncidentSourceECurrent::IncidentSourceECurrent(int distance_, Direction dir_)
  : IncidentSourceCurrent(distance_, dir_, false)
{}

void IncidentSourceECurrent::initStorage(Storage *storage)
{

  pJx = storage->addBorderLayer("IncidentJx", dir, 1, distance);
  pJy = storage->addBorderLayer("IncidentJy", dir, 1, distance);
  pJz = storage->addBorderLayer("IncidentJz", dir, 1, distance);

  switch (dir)
  {
    case east:
    case west:
      pPsi[0] = pJy;
      pPsi[1] = pJz;
      dx = Globals::instance().gridDX();
      break;
    case north:
    case south:
      pPsi[0] = pJz;
      pPsi[1] = pJx;
      dx = Globals::instance().gridDY();
      break;
    case up:
    case down:
      pPsi[0] = pJx;
      pPsi[1] = pJy;
      dx = Globals::instance().gridDZ();
      break;
  }

}

void IncidentSourceECurrent::setCurrents(GridIndex i, Vector H)
{
```

```
  (*pPsi[0])(i[0], i[1], i[2]) =  H[transverse2];
  (*pPsi[1])(i[0], i[1], i[2]) = -H[transverse1];
}

//===========================================================
//==========  IncidentSourceHCurrent
//===========================================================

IncidentSourceHCurrent::IncidentSourceHCurrent(int distance_, Direction dir_)
  : CPMLBorderCurrent(distance_, dir_, true)
{}

void IncidentSourceHCurrent::initStorage(Storage *storage)
{

  pJx = storage->addBorderLayer("IncidentJx", dir, 1, distance);
  pJy = storage->addBorderLayer("IncidentJy", dir, 1, distance);
  pJz = storage->addBorderLayer("IncidentJz", dir, 1, distance);

  switch (dir)
  {
    case east:
    case west:
      pPsi[0] = pJy;
      pPsi[1] = pJz;
      dx = Globals::instance().gridDX();
      break;
    case north:
    case south:
      pPsi[0] = pJz;
      pPsi[1] = pJx;
      dx = Globals::instance().gridDY();
      break;
    case up:
    case down:
      pPsi[0] = pJx;
      pPsi[1] = pJy;
      dx = Globals::instance().gridDZ();
      break;
  }
}

void IncidentSourceHCurrent::setCurrents(GridIndex i, Vector E)
{
  (*pPsi[0])(i[0], i[1], i[2]) =  E[transverse2];
  (*pPsi[1])(i[0], i[1], i[2]) = -E[transverse1];
}
```

```
#include "mpi_bound.h"

#include "mpulse.h"
#include "globals.h"
#include "factor.h"

#include <sstream>
#include <fstream>

#ifndef SINGLE_PROCESSOR

/* ************************************************************
 *                  MPIPeriodicSplitXBoundary                 *
 ************************************************************/

MPIPeriodicSplitXBoundary::MPIPeriodicSplitXBoundary()
{}

void MPIPeriodicSplitXBoundary::init()
{
  Low = Globals::instance().gridLow();
  High = Globals::instance().gridHigh();

  MPI_Comm_size(MPI_COMM_WORLD,&ComSize);

  int periodic = true;

  MPI_Cart_create(MPI_COMM_WORLD,1,&ComSize,&periodic,true,&comm);
  MPI_Comm_rank(comm,&ComRank);

  MPI_Cart_coords(comm,ComRank,1,&mycoord);

  MPI_Cart_shift(comm,0,1,&leftcoord,&rightcoord);

  double width = (High[0]-2.)/double(ComSize);

  if (ComRank>0)
      Low[0] = int(width*mycoord);

  if (ComRank<(ComSize-1))
      High[0] = int(width*(mycoord+1))+1;

  exchSize = (High[1]-Low[1]+1);

  sendarr = new double[exchSize];
  recvarr = new double[exchSize];

  Globals::instance().setMaster(this->master());
  Globals::instance().setUniqueId(this->getUniqueId());
}

MPIPeriodicSplitXBoundary::~MPIPeriodicSplitXBoundary() {
    MPI_Finalize();
    delete[] sendarr;
    delete[] recvarr;
}

void MPIPeriodicSplitXBoundary::exchangeX(DataGrid3d &field)
{
  int xi;
  int yi;
  int zi;
  MPI_Status stat;

  int arr_ind = 0;
  xi = Low[0] + 1;
```

```
  for (yi = Low[1]; yi <= High[1]; ++yi)
    for (zi = Low[2]; zi <= High[2]; ++zi)
      sendarr[arr_ind++] = field(xi, yi, zi);

  MPI_Sendrecv(sendarr, exchSize, MPI_DOUBLE, leftcoord, 0,
               recvarr, exchSize, MPI_DOUBLE, rightcoord, 0,
               comm, &stat);

  arr_ind = 0;
  xi = High[0];

  for (yi = Low[1]; yi <= High[1]; ++yi)
    for (zi = Low[2]; zi <= High[2]; ++zi)
      field(xi,yi,zi) = recvarr[arr_ind++];

  arr_ind = 0;
  xi = High[0] - 1;

  for (yi = Low[1]; yi <= High[1]; ++yi)
    for (zi = Low[2]; zi <= High[2]; ++zi)
      sendarr[arr_ind++] = field(xi, yi, zi);

  MPI_Sendrecv(sendarr, exchSize, MPI_DOUBLE, rightcoord, 0,
               recvarr, exchSize, MPI_DOUBLE, leftcoord, 0,
               comm, &stat);

  arr_ind = 0;
  xi = Low[0];

  for (yi = Low[1]; yi <= High[1]; ++yi)
    for (zi = Low[2]; zi <= High[2]; ++zi)
      field(xi,yi,zi) = recvarr[arr_ind++];

}

void MPIPeriodicSplitXBoundary::exchangeY(DataGrid3d &field)
{
  const GridIndex3d &UBound = field.getHigh();
  const GridIndex3d &LBound = field.getLow();

  int xi;
  int zi;

  int my0=UBound[1], my1=my0-1;
  int ly0=LBound[1], ly1=ly0+1;

  for (xi = LBound[0]; xi <= UBound[0]; ++xi)
  {
    for (zi = LBound[2]; zi <= UBound[2]; ++zi)
    {
      field(xi, ly0, zi) = field(xi, my1, zi);
      field(xi, my0, zi) = field(xi, ly1, zi);
    }
  }
}

void MPIPeriodicSplitXBoundary::exchangeZ(DataGrid3d &field)
{
  const GridIndex3d &UBound = field.getHigh();
  const GridIndex3d &LBound = field.getLow();

  int xi;
  int yi;

  int mz0=UBound[2], mz1=mz0-1;
```

```
  int lz0=LBound[2], lz1=lz0+1;

  for (xi = LBound[0]; xi <= UBound[0]; ++xi)
  {
    for (yi = LBound[1]; yi <= UBound[1]; ++yi)
    {
      field(xi, yi, lz0) = field(xi, yi, mz1);
      field(xi, yi, mz0) = field(xi, yi, lz1);
    }
  }
}


double MPIPeriodicSplitXBoundary::AvgReduce(double val) {
  double result;
    //this collects results from all nodes, MPI_SUM returns the sum of all results
  MPI_Allreduce(&val, &result, 1, MPI_DOUBLE, MPI_SUM, comm);
  return result/double(ComSize);
}

double MPIPeriodicSplitXBoundary::MaxReduce(double val) {
  double result;
    //this collects results from all nodes, MPI_MAX returns the maximum of all results
  MPI_Allreduce(&val, &result, 1, MPI_DOUBLE, MPI_MAX, comm);
  return result;
}


const GridIndex &MPIPeriodicSplitXBoundary::RegionLow() const {
    return Low;
}

const GridIndex &MPIPeriodicSplitXBoundary::RegionHigh() const {
    return High;
}

/* ************************************************************
 *                 MPIPeriodicSplitXYBoundary                 *
 ************************************************************/

MPIPeriodicSplitXYZBoundary::MPIPeriodicSplitXYZBoundary()
{}


void MPIPeriodicSplitXYZBoundary::init()
{
  Low = Globals::instance().gridLow();
  High = Globals::instance().gridHigh();

  MPI_Comm_size(MPI_COMM_WORLD,&ComSize);

  int periodic[2] = {true, true};

  std::vector<int> box(3);
  for (int i=0; i<3; ++i)
  {
    box[i] = High[i]-Low[i]-1;
  }

  std::vector<int> eqDims;

  equalFactors(ComSize, 3, eqDims, box);

  std::copy(eqDims.begin(), eqDims.end(), dims);

  MPI_Cart_create(MPI_COMM_WORLD,3,dims,periodic,true,&comm);
```

```
  MPI_Comm_rank(comm,&ComRank);

  MPI_Cart_coords(comm,ComRank,3,mycoord);

  MPI_Cart_shift(comm,0,1,&xprevcoord,&xnextcoord);
  MPI_Cart_shift(comm,1,1,&yprevcoord,&ynextcoord);
  MPI_Cart_shift(comm,2,1,&zprevcoord,&znextcoord);

  double width[3];
  width[0] = (High[0]-1.)/double(dims[0]);
  width[1] = (High[1]-1.)/double(dims[1]);
  width[2] = (High[2]-1.)/double(dims[2]);

  for (int i=0; i<3; ++i)
  {
    if (mycoord[i]>0)
      Low[i] = int(width[i]*mycoord[i]);

    if (mycoord[i]<(dims[i]-1))
      High[i] = int(width[i]*(mycoord[i]+1))+1;

  }


  exchSize[0] = (High[1]-Low[1]+1)*(High[2]-Low[2]+1);
  exchSize[1] = (High[0]-Low[0]+1)*(High[2]-Low[2]+1);
  exchSize[2] = (High[0]-Low[0]+1)*(High[1]-Low[1]+1);

  sendarrx = new double[exchSize[0]];
  recvarrx = new double[exchSize[0]];
  sendarry = new double[exchSize[1]];
  recvarry = new double[exchSize[1]];
  sendarrz = new double[exchSize[2]];
  recvarrz = new double[exchSize[2]];

  Globals::instance().setMaster(this->master());
  Globals::instance().setUniqueId(this->getUniqueId());

  std::ostringstream S;
  S << "boundary"<<ComRank<<".dat"<<char(0);
  std::ofstream O(S.str().c_str());
  O << "Coord: " << mycoord[0] << " " << mycoord[1] << " " << mycoord[2] << "\n";
  O << "Dims: " << dims[0] << " " << dims[1] << " " << dims[2] << "\n";
  O << "Low: " << Low[0] << " " << Low[1] << " " << Low[2] << "\n";
  O << "High: " << High[0] << " " << High[1] << " " << High[2] << "\n";
  O.close();
}

MPIPeriodicSplitXYZBoundary::~MPIPeriodicSplitXYZBoundary()
{
    MPI_Finalize();
    delete[] sendarrx;
    delete[] recvarrx;
    delete[] sendarry;
    delete[] recvarry;
    delete[] sendarrz;
    delete[] recvarrz;
}

void MPIPeriodicSplitXYZBoundary::exchangeX(DataGrid3d &field)
{
  int xi;
  int yi;
  int zi;
  MPI_Status stat;
```

```
  int arr_ind = 0;
  xi = Low[0] + 1;

  for (yi = Low[1]; yi <= High[1]; ++yi)
    for (zi = Low[2]; zi <= High[2]; ++zi)
      sendarrx[arr_ind++] = field(xi, yi, zi);

  MPI_Sendrecv(sendarrx, exchSize[0], MPI_DOUBLE, xprevcoord, 0,
               recvarrx, exchSize[0], MPI_DOUBLE, xnextcoord, 0,
               comm, &stat);

  arr_ind = 0;
  xi = High[0];

  for (yi = Low[1]; yi <= High[1]; ++yi)
    for (zi = Low[2]; zi <= High[2]; ++zi)
      field(xi,yi,zi) = recvarrx[arr_ind++];

  arr_ind = 0;
  xi = High[0] - 1;

  for (yi = Low[1]; yi <= High[1]; ++yi)
    for (zi = Low[2]; zi <= High[2]; ++zi)
      sendarrx[arr_ind++] = field(xi, yi, zi);

  MPI_Sendrecv(sendarrx, exchSize[0], MPI_DOUBLE, xnextcoord, 0,
               recvarrx, exchSize[0], MPI_DOUBLE, xprevcoord, 0,
               comm, &stat);

  arr_ind = 0;
  xi = Low[0];

  for (yi = Low[1]; yi <= High[1]; ++yi)
    for (zi = Low[2]; zi <= High[2]; ++zi)
      field(xi,yi,zi) = recvarrx[arr_ind++];

}

void MPIPeriodicSplitXYZBoundary::exchangeY(DataGrid3d &field)
{
  int xi;
  int yi;
  int zi;
  MPI_Status stat;

  int arr_ind = 0;
  yi = Low[1] + 1;

  for (xi = Low[0]; xi <= High[0]; ++xi)
    for (zi = Low[2]; zi <= High[2]; ++zi)
      sendarry[arr_ind++] = field(xi, yi, zi);

  MPI_Sendrecv(sendarry, exchSize[1], MPI_DOUBLE, yprevcoord, 0,
               recvarry, exchSize[1], MPI_DOUBLE, ynextcoord, 0,
               comm, &stat);

  arr_ind = 0;
  yi = High[1];

  for (xi = Low[0]; xi <= High[0]; ++xi)
    for (zi = Low[2]; zi <= High[2]; ++zi)
      field(xi,yi,zi) = recvarry[arr_ind++];

  arr_ind = 0;
  yi = High[1] - 1;
```

```
  for (xi = Low[0]; xi <= High[0]; ++xi)
    for (zi = Low[2]; zi <= High[2]; ++zi)
      sendarry[arr_ind++] = field(xi, yi, zi);

  MPI_Sendrecv(sendarry, exchSize[1], MPI_DOUBLE, ynextcoord, 0,
               recvarry, exchSize[1], MPI_DOUBLE, yprevcoord, 0,
               comm, &stat);

  arr_ind = 0;
  yi = Low[1];

  for (xi = Low[0]; xi <= High[0]; ++xi)
    for (zi = Low[2]; zi <= High[2]; ++zi)
      field(xi,yi,zi) = recvarry[arr_ind++];

}

void MPIPeriodicSplitXYZBoundary::exchangeZ(DataGrid3d &field)
{
  int xi;
  int yi;
  int zi;
  MPI_Status stat;

  int arr_ind = 0;
  zi = Low[2] + 1;

  for (xi = Low[0]; xi <= High[0]; ++xi)
    for (yi = Low[1]; yi <= High[1]; ++yi)
      sendarrz[arr_ind++] = field(xi, yi, zi);

  MPI_Sendrecv(sendarrz, exchSize[2], MPI_DOUBLE, zprevcoord, 0,
               recvarrz, exchSize[2], MPI_DOUBLE, znextcoord, 0,
               comm, &stat);

  arr_ind = 0;
  zi = High[2];

  for (xi = Low[0]; xi <= High[0]; ++xi)
    for (yi = Low[1]; yi <= High[1]; ++yi)
      field(xi,yi,zi) = recvarrz[arr_ind++];

  arr_ind = 0;
  zi = High[2] - 1;

  for (xi = Low[0]; xi <= High[0]; ++xi)
    for (yi = Low[1]; yi <= High[1]; ++yi)
      sendarrz[arr_ind++] = field(xi, yi, zi);

  MPI_Sendrecv(sendarrz, exchSize[2], MPI_DOUBLE, znextcoord, 0,
               recvarrz, exchSize[2], MPI_DOUBLE, zprevcoord, 0,
               comm, &stat);

  arr_ind = 0;
  zi = Low[2];

  for (xi = Low[0]; xi <= High[0]; ++xi)
    for (yi = Low[1]; yi <= High[1]; ++yi)
      field(xi,yi,zi) = recvarrz[arr_ind++];
}


double MPIPeriodicSplitXYZBoundary::AvgReduce(double val) {
  double result;
  MPI_Allreduce(&val, &result, 1, MPI_DOUBLE, MPI_SUM, comm);
  return result/double(ComSize);
```

```
}

double MPIPeriodicSplitXYZBoundary::MaxReduce(double val) {
  double result;
  MPI_Allreduce(&val, &result, 1, MPI_DOUBLE, MPI_MAX, comm);
  return result;
}

const GridIndex &MPIPeriodicSplitXYZBoundary::RegionLow() const {
  return Low;
}

const GridIndex &MPIPeriodicSplitXYZBoundary::RegionHigh() const {
  return High;
}


#endif
```

```
#include "globals.h"
#include "process.h"

#ifndef SINGLE_PROCESSOR
#include "mpi.h"
#endif

#include <fstream>
#include <string>
#include <unistd.h>

// Print some initial diagnostics and start the simulation.
//---------------------------------------------------------------
//begin of main
int main (int argc, char** argv) {
  // if SINGLE_PROCESSOR is not defined, the MPI protocol is used

#ifndef SINGLE_PROCESSOR
  MPI_Init(&argc, &argv);
#endif

  // set  the static member of parameters
  Globals::setArgc(argc);
  // set  the static member of parameters
  Globals::setArgv(argv);

  //---------------------------------------------------------------
  // if specified get setup parameters from file
  // default setup file name, to be parsed if specified differently in the
  // argument
  std::string setupfilename = "setup.dat";
  if (argc > 1) setupfilename = argv[1];

  //buffer for current working directory
  char buf[200];
  //get current working directory
  getcwd(buf,200);
  std::cerr << "*********************************************************************************
****\n";
  std::cerr << "****************************** This is MPulse ****************************
****\n";
  std::cerr << "*********************************************************************************
****\n";
  std::cerr << "CWD is " << buf << "\n";

  //open setup file
  std::ifstream setupfile(setupfilename.c_str());
  std::cerr << "MAIN: Reading Inputfile: " << setupfilename.c_str() << " ...\n";
  //if "setup.dat" does not exist, display error message
  if (!setupfile)
  {
    std::cerr << "MAIN: Could not open: " << setupfilename.c_str() << " ...\n";
    exit(-1);
  }

  Process *process = new Process();
  process->Rebuild(setupfile);

  std::cerr << "                          *** MAIN:  INITIALIZING ***\n";

  //turn over control to process class, initialize...
  process->init();
  std::cout << "                              *** MAIN: RUNNING ***\n";

  //.. and run the process
  process->run();
```

```
  //delete the allocated process
  delete process;

  std::cerr << "************************** MPulse has finished **************************
\n";
  return 0;
}

// end of main
```

```
#include "periodic_bound.h"

#include "mpulse.h"
#include "globals.h"

/* ************************************************************
 *                  SinglePeriodicBoundary                    *
 ************************************************************/

void SinglePeriodicBoundary::exchangeX(DataGrid3d &field)
{
  const GridIndex3d &UBound = field.getHigh();
  const GridIndex3d &LBound = field.getLow();

  int yi;
  int zi;

  int mx0=UBound[0], mx1=mx0-1;
  int lx0=LBound[0], lx1=lx0+1;

  for (yi = LBound[1]; yi <= UBound[1]; ++yi)
  {
    for (zi = LBound[2]; zi <= UBound[2]; ++zi)
    {
      field(lx0, yi, zi) = field(mx1, yi, zi);
      field(mx0, yi, zi) = field(lx1, yi, zi);
    }
  }
}

void SinglePeriodicBoundary::exchangeY(DataGrid3d &field)
{
  const GridIndex3d &UBound = field.getHigh();
  const GridIndex3d &LBound = field.getLow();

  int xi;
  int zi;

  int my0=UBound[1], my1=my0-1;
  int ly0=LBound[1], ly1=ly0+1;

  for (xi = LBound[0]; xi <= UBound[0]; ++xi)
  {
    for (zi = LBound[2]; zi <= UBound[2]; ++zi)
    {
      field(xi, ly0, zi) = field(xi, my1, zi);
      field(xi, my0, zi) = field(xi, ly1, zi);
    }
  }
}

void SinglePeriodicBoundary::exchangeZ(DataGrid3d &field)
{
  const GridIndex3d &UBound = field.getHigh();
  const GridIndex3d &LBound = field.getLow();

  int xi;
  int yi;

  int mz0=UBound[2], mz1=mz0-1;
  int lz0=LBound[2], lz1=lz0+1;

  for (xi = LBound[0]; xi <= UBound[0]; ++xi)
  {
    for (yi = LBound[1]; yi <= UBound[1]; ++yi)
    {
```

```
      field(xi, yi, lz0) = field(xi, yi, mz1);
      field(xi, yi, mz0) = field(xi, yi, lz1);
    }
  }
}


const GridIndex &SinglePeriodicBoundary::RegionLow() const {
  return Globals::instance().gridLow();
}

const GridIndex &SinglePeriodicBoundary::RegionHigh() const {
  return Globals::instance().gridHigh();
}
```

```
#include "plasmacurrent.h"
#include "storage.h"

PlasmaCurrent::PlasmaCurrent() {}

void PlasmaCurrent::initStorage(Storage *storage_)
{
  storage = storage_;

  pEx = storage->addGrid("Ex");
  pEy = storage->addGrid("Ey");
  pEz = storage->addGrid("Ez");

  pJx = storage->addGrid("PlasmaJx");
  pJy = storage->addGrid("PlasmaJy");
  pJz = storage->addGrid("PlasmaJz");

  pRho = storage->addGrid("PlasmaDensity");
}

void PlasmaCurrent::stepScheme(double dt)
{
  DataGrid &Ex = *pEx;
  DataGrid &Ey = *pEy;
  DataGrid &Ez = *pEz;

  DataGrid &Jx = *pJx;
  DataGrid &Jy = *pJy;
  DataGrid &Jz = *pJz;
  DataGrid &Rho = *pRho;

  GridIndex low = storage->getLow();
  GridIndex high = storage->getHigh();

  double gdt = 1-gamma*dt;
  double emdt = em*dt;

  for (int i=low[0]; i<high[0]; ++i)
    for (int j=low[1]; j<high[1]; ++j)
      for (int k=low[2]; k<high[2]; ++k)
  {
    double &jx = Jx(i,j,k);
    double &jy = Jy(i,j,k);
    double &jz = Jz(i,j,k);
    double &rho = Rho(i,j,k);

    jx = jx*gdt - emdt*Ex(i,j,k)*rho;
    jy = jy*gdt - emdt*Ey(i,j,k)*rho;
    jz = jz*gdt - emdt*Ez(i,j,k)*rho;
  }

}


ParameterMap* PlasmaCurrent::MakeParamMap (ParameterMap* pm)
{
  pm = Rebuildable::MakeParamMap(pm);

  (*pm)["em"] = WParameter(new ParameterValue<double>(&em,1.0));
  (*pm)["gamma"] = WParameter(new ParameterValue<double>(&gamma,0.1));

  return pm;
}
```

```
#include "plasmadensity.h"
#include "storage.h"
#include "util.h"

PlasmaDensity::PlasmaDensity() {}

void PlasmaDensity::initStorage(Storage *storage_)
{
  storage = storage_;

  pEx = storage->addGrid("Ex");
  pEy = storage->addGrid("Ey");
  pEz = storage->addGrid("Ez");

  pRho = storage->addGrid("PlasmaDensity");
}

void PlasmaDensity::stepScheme(double dt)
{
  DataGrid &Ex = *pEx;
  DataGrid &Ey = *pEy;
  DataGrid &Ez = *pEz;

  DataGrid &Rho = *pRho;

  GridIndex low = storage->getLow();
  GridIndex high = storage->getHigh();

  double n = 0.5*dt*nu;
  double m = dt*mpa;

  for (int i=low[0]; i<high[0]; ++i)
    for (int j=low[1]; j<high[1]; ++j)
      for (int k=low[2]; k<high[2]; ++k)
  {
    double &ex = Ex(i,j,k);
    double &ey = Ey(i,j,k);
    double &ez = Ez(i,j,k);
    double &rho = Rho(i,j,k);

    double I = ex*ex + ey*ey + ez*ez;
    double IK = ipow(I,K);


    rho = ( rho*(1+n*I) + m*IK ) / (1 - n*I);
  }
}


ParameterMap* PlasmaCurrent::MakeParamMap (ParameterMap* pm)
{
  pm = Current::MakeParamMap(pm);

  (*pm)["nu"] = WParameter(new ParameterValue<double>(&nu,1.0));
  (*pm)["mpa"] = WParameter(new ParameterValue<double>(&mpa,1.0));
  (*pm)["K"] = WParameter(new ParameterValue<int>(&K,1.0));

  return pm;
}
```

```
#include "process.h"
#include "diagnostic.h"

Process *Process::process;

Process::~Process()
{
  if (fieldSim) delete fieldSim;
  if (globals) delete globals;
}

void Process::init()
{
  fieldSim->init();
}

void Process::run()
{
  int T = Globals::instance().totalTime();

  //display time step and call execute for all species +field
  for (time=0; time<T; ++time)
  {
    if (Globals::instance().isMaster())
      std::cout << "Cycle " << time << std::endl << std::flush;

    fieldSim->execute();

    DiagnosticManager::instance().execute();
  }
}

ParameterMap* Process::MakeParamMap (ParameterMap* pm) {
  pm = Rebuildable::MakeParamMap(pm);
  (*pm)["globals"]
      = WParameter(new ParameterRebuild<Globals, Globals>(&globals));

  (*pm)["fieldsim"]
      = WParameter(new ParameterRebuild<FieldSimulation, FieldSimulation>(&fieldSim));

  return pm;
}


std::string Process::Rebuild(std::istream& in)
{
  std::cerr << "Rebuilding Process ...\n";
  Rebuildable::Rebuild(in);

  if (NULL == globals) {
    std::cerr << "No globals specified! Must exit!\n";
    exit(-1);
  }
  if (NULL == fieldSim) {
    std::cerr << "No field simulation specified! Must exit!\n";
    exit(-1);
  }
  return "";
}
```

```
#include "pulseinit.h"
#include "globals.h"
#include "storage.h"



void GaussPulseInit::init(Storage &fields)
{
  std::cerr << "=== INITIALIZING Gaussian Pulse ===\n";
  double ex = by/kz;
  double ey = -bx/kz;

  double dx = Globals::instance().gridDX();
  double dy = Globals::instance().gridDY();
  double dz = Globals::instance().gridDZ();
  double dx2 = dx*dx;
  double dy2 = dy*dy;

  GridIndex gridLow = Globals::instance().gridLow();
  GridIndex gridHigh = Globals::instance().gridHigh();

  DataGrid &Ex = fields.getGrid("Ex");
  DataGrid &Ey = fields.getGrid("Ey");
  DataGrid &Ez = fields.getGrid("Ez");
  DataGrid &Bx = fields.getGrid("Bx");
  DataGrid &By = fields.getGrid("By");
  DataGrid &Bz = fields.getGrid("Bz");

  GridIndex low = fields.getLow();
  GridIndex high = fields.getHigh();

  double nkz = dz*kz;
  double rr0 = r0*r0;
  double zz0 = z0*z0;

  double xh = 0.5*double(gridHigh[0] + gridLow[0]);
  double yh = 0.5*double(gridHigh[1] + gridLow[1]);


  for (int i=low[0]; i<=high[0]; ++i)
    for (int j=low[1]; j<=high[1]; ++j)
    {
      double iie = i-xh;
      double jje = j-yh;
      double rre = (iie*iie*dx2 + jje*jje*dy2) / rr0;

      double iib = i+0.5-xh;
      double jjb = j+0.5-yh;
      double rrb = (iib*iib*dx2 + jjb*jjb*dy2) / rr0;

      for (int k=low[2]; k<=high[2]; ++k)
      {
        double nze = k*dz - zc;
        double nzb = (k+0.5)*dz - zc;

        double kze = k*nkz;
        double kzb = (k+0.5)*nkz;

        double ampe = cos(kze - C*rre)*exp(-nze*nze/zz0 - rre);
        double ampb = cos(kzb - C*rrb)*exp(-nzb*nzb/zz0 - rrb);

        Ex(i,j,k) = ex*ampe;
        Ey(i,j,k) = ey*ampe;
        Ez(i,j,k) = 0;

        Bx(i,j,k) = bx*ampb;
```

```
        By(i,j,k) = by*ampb;
        Bz(i,j,k) = 0;
      }
    }

}


ParameterMap* GaussPulseInit::MakeParamMap (ParameterMap* pm) {
  pm = Rebuildable::MakeParamMap(pm);
  (*pm)["k"] = WParameter(new ParameterValue<double>(&kz,1));
  (*pm)["r0"] = WParameter(new ParameterValue<double>(&r0,1));
  (*pm)["z0"] = WParameter(new ParameterValue<double>(&z0,1));
  (*pm)["zc"] = WParameter(new ParameterValue<double>(&zc,1));
  (*pm)["C"] = WParameter(new ParameterValue<double>(&C,1));
  (*pm)["Bx"] = WParameter(new ParameterValue<double>(&bx,1));
  (*pm)["By"] = WParameter(new ParameterValue<double>(&by,1));

  return pm;
}
```

```
#include "rebuild.h"
#include <algorithm>
//for the count algorithm
//-----------------------------------------------------------------------------
ParameterMap* Rebuildable::MakeParamMap (ParameterMap* pm) {
  //if not present, allocate
  if (NULL == pm) pm = new ParameterMap;
  return pm;
}
//ParameterMap is a std::map<std::string,WParameter> as found in parameter.h, while WParame
ter
// is a wrapped pointer to parameter
// for ParameterMap see parameter.h
//-----------------------------------------------------------------------------
//Rebuild
std::string Rebuildable::Rebuild (std::istream& in) {
  //set up parameter map
  ParameterMap* pm = MakeParamMap();
  // get first token
  std::string strToken;
  in >> strToken;
  //outer while loop
  //get next token until EOF or "}", remove commentary lines
  while ((strToken != "}") && (!in.eof())) {
    //-------------------------------------------------------------------------
    // loop for removing commentaries from stream
    while (strToken == "//") {
      char ch;
      do {
                    in.get(ch);
      } while ((ch != '\n') && (!in.eof()));
      //get next token unless EOF is encountered
      if (!in.eof()) in >> strToken;
    }
    //end of commentary loop
    //-------------------------------------------------------------------------
    //test on not (end of block)
    if ("}" != strToken) {
      //-----------------------------------------------------------------------
      //handling errors and unexpected tokens
      //if its not found in the parameter map, write it to strParam for displaying error me
ssages
      if (0 == pm->count(strToken)) {
        std::string strParam = strToken;
                        //get next token
                               in >> strToken;
                               //if a "{" token is encountered, skip the contents
                               // since this is a task, it should be skipped (?)
                               if (strToken == "{") {
                                 std::cerr << "Unknown Task " << strParam << ". Skipping..
. " << std::endl;

                                 int nLevel = 0;
                                 do {
                                   in >> strToken;
                                   if (strToken == "{") nLevel++;
                                   if (strToken == "}") nLevel--;
                                 } while (((strToken != "}") || (nLevel >= 0)) && (!in.eof
()));

                                 //if EOF encountered, display error message, else get nex
t token
                          if (in.eof())
                                   std::cerr << "Unexpected end of file." << std::endl;
                                 else
                                   in >> strToken;
                               }
                               else
```

```
                                    // Display name of parameter if it is not found in the ma
p
                                    // But do not break, since a correct one could follow
                                    std::cerr << "Unknown Parameter " << strParam << std::end
l;
      }
      //-------------------------------------------------------------------------
      // here the real rebuilding is done
      // this else belongs to if (0 == pm->count(strToken))!
      // if all is as expected, write data to the parameters
        // by calling its rebuild method
      else {
        // get the parameter object from the map via index operator
        // dereference pm[token] --> get wrapped pointer to parameter
          //--> cast to normal pointer to parameter
          // --> assign to new pointer to parameter
        // c-style cast! (reinterpret_cast<*parameter>(*pm) ?)
        Parameter *par = (Parameter*)(*pm)[strToken];
        // deference and call the rebuild method  of this object with the "in" filestream
          // parameter's rebuild member gets the next token from the file stream,
          // until a new block is encountered, see also parameter.h
        strToken = par->Rebuild(in);
      }
      //-------------------------------------------------------------------------
    }
  }
  //-----------------------------------------------------------------------------
  // end of outer while loop
  // if end of block reached, get next token
  if (strToken == "}") in >> strToken;
  //remove commentary from instream
  while (strToken == "//") {
    char ch;
    do {
      in.get(ch);
    } while ((ch != '\n') && (!in.eof()));
    //get next token until EOF
    if (!in.eof())
      in >> strToken;
    else
      break;
  }
  //-----------------------------------------------------------------------------
  // clean up
  // clean pm
  while (!pm->empty()) {
    ParameterMap::iterator iter = pm->begin();
    //c-style cast to normal, non-wrapped pointer to parameter
    delete ((Parameter*)(*iter).second);
    pm->erase(iter);
  }
  //deallocate pm
  delete pm;
  //-----------------------------------------------------------------------------
  //return token
  return strToken;
}
```

```
#include "sources.h"



IncidentSourceCurrent *PlaneWaveSource::makeECurrent(int distance_, Direction dir_)
{
  Vector k(kx,ky,kz);
  Vector H(Hx,Hy,Hz);

  Vector E(ky*Hz-kz*Hy, kz*Hx-kx*Hz, kx*Hy-ky*Hx);

  double bmag = sqrt(Hx*Hx + Hy*Hy + Hz*Hz);
  double factor = -bmag/sqrt(E[0]*E[0] + E[1]*E[1] + E[2]*E[2]);

  E *= factor;

  return new PlaneWaveSourceECurrent(distance_, dir_, k, E, H);

}

IncidentSourceCurrent *PlaneWaveSource::makeHCurrent(int distance_, Direction dir_)
{
  Vector k(kx,ky,kz);
  Vector H(Hx,Hy,Hz);

  Vector E(ky*Hz-kz*Hy, kz*Hx-kx*Hz, kx*Hy-ky*Hx);

  double bmag = sqrt(Hx*Hx + Hy*Hy + Hz*Hz);
  double factor = -bmag/sqrt(E[0]*E[0] + E[1]*E[1] + E[2]*E[2]);

  E *= factor;

  return new PlaneWaveSourceHCurrent(distance_, dir_, k, E, H);

}

ParameterMap* PlaneWaveSource::MakeParamMap (ParameterMap* pm)
{
  pm = IncidentSource::MakeParamMap(pm);

  (*pm)["kx"] = WParameter(new ParameterValue<double>(&this->kx,0));
  (*pm)["ky"] = WParameter(new ParameterValue<double>(&this->ky,0));
  (*pm)["kz"] = WParameter(new ParameterValue<double>(&this->kz,1));

  (*pm)["Hx"] = WParameter(new ParameterValue<double>(&this->Hx,1));
  (*pm)["Hy"] = WParameter(new ParameterValue<double>(&this->Hy,0));
  (*pm)["Hz"] = WParameter(new ParameterValue<double>(&this->Hz,0));

  (*pm)["ramp"] = WParameter(new ParameterValue<double>(&this->ramp,0.5));

  return pm;
}

PlaneWaveSourceECurrent::PlaneWaveSourceECurrent
  (
    int distance_,
    Direction dir_,
    Vector k_,
    Vector E_,
    Vector H_
  )
  : distance(distance_), dir(dir_), k(k_), E(E_), H(H_)
{
  dt = Globals::instance().dt();
  om = sqrt(k[0]*k[0] + k[1]*k[1] + k[2]*k[2]);
}
```

```
void PlaneWaveSourceECurrent::stepSchemeInit(double dt) {}

void PlaneWaveSourceECurrent::stepScheme(double dt)
{
  GridIndex low = pJx->getLow();
  GridIndex high = pJx->getHigh();

  double time = dt*Globals::instance().totalTime();

  double dx = Globals::instance().gridDX();
  double dy = Globals::instance().gridDY();
  double dz = Globals::instance().gridDZ();


  GridIndex i;
  for (int i[0]=low[0]; i[0]<=high[0]; ++i[0])
    for (int i[1]=low[1]; i[1]<=high[1]; ++i[1])
      for (int i[2]=low[2]; i[2]<=high[2]; ++i[2])
      {
        time_loc = time - (k[0]*i[0]*dx + k[1]*i[1]*dy + k[2]*i[2]*dz);

        double amp = sin(om*time_loc);
        if (time_loc<ramp) amp = 0;
        else if (time_loc<2*ramp) amp *= time_loc/ramp - 1;

        setCurrents(i,H*amp);
      }
}

PlaneWaveSourceHCurrent::PlaneWaveSourceHCurrent
  (
    int distance_,
    Direction dir_,
    Vector k_,
    Vector E_,
    Vector H_
  )
  : distance(distance_), dir(dir_), k(k_), E(E_), H(H_)

{
  dt = Globals::instance().dt();
  om = sqrt(k[0]*k[0] + k[1]*k[1] + k[2]*k[2]);
}


void PlaneWaveSourceHCurrent::stepSchemeInit(double dt) {}

void PlaneWaveSourceHCurrent::stepScheme(double dt)
{
  GridIndex low = pJx->getLow();
  GridIndex high = pJx->getHigh();

  double time = dt*Globals::instance().totalTime();

  double dx = Globals::instance().gridDX();
  double dy = Globals::instance().gridDY();
  double dz = Globals::instance().gridDZ();


  GridIndex i;
  for (int i[0]=low[0]; i[0]<=high[0]; ++i[0])
    for (int i[1]=low[1]; i[1]<=high[1]; ++i[1])
      for (int i[2]=low[2]; i[2]<=high[2]; ++i[2])
      {
        time_loc = time - (k[0]*(i[0]+0.5)*dx + k[1]*(i[1]+0.5)*dy + k[2]*(i[2]+0.5)*dz);
```

```
        double amp = sin(om*time_loc);
        if (time_loc<ramp) amp = 0;
        else if (time_loc<2*ramp) amp *= time_loc/ramp - 1;

        setCurrents(i,E*amp);
      }
}
```

```
#include "storage.h"
#include "globals.h"
#include "boundary.h"



Storage::Storage()
  : low(0,0,0),
    high(1,1,1),
    boundary(0),
    grids(),
    gridsN(),
    gridsS(),
    gridsE(),
    gridsW(),
    groups()
{}

Storage::~Storage()
{
  GridDeleter deleteGrid;
  forAllGrids(grids, deleteGrid);
  forAllGrids(gridsN, deleteGrid);
  forAllGrids(gridsS, deleteGrid);
  forAllGrids(gridsE, deleteGrid);
  forAllGrids(gridsW, deleteGrid);
}

void Storage::resize(GridIndex low_, GridIndex high_)
{
  low = low_;
  high = high_;
  GridResizer resizer(low,high);
  forAllGrids(grids, resizer);
  if (
     (gridsN.size() + gridsS.size()
    + gridsE.size() + gridsW.size()
    + gridsU.size() + gridsD.size()
    + lines.size() ) > 0
  )
  {
    std::cerr << "Resizing not implemented\n";
    exit(-1);
  }
}

DataGrid &Storage::getGrid(const std::string &gridid)
{
  return *(grids[gridid]);
}

bool Storage::hasGrid(const std::string &gridid)
{
  return (grids.count(gridid) > 0);
}

DataGrid *Storage::addGrid(const std::string &gridid)
{
  DataGrid *g = 0;
  if (grids.count(gridid) == 0)
  {
    g = new DataGrid(low, high);
    (*g) = 0;
    grids[gridid] = g;
  } else
  {
```

```
    g = grids[gridid];
  }
  return g;
}

DataGrid &Storage::getBorderLayer(const std::string &gridid, Direction dir)
{
  switch (dir)
  {
    case north: return *(gridsN[gridid]); break;
    case south: return *(gridsS[gridid]); break;
    case east:  return *(gridsE[gridid]); break;
    case west:  return *(gridsW[gridid]); break;
    case up:    return *(gridsU[gridid]); break;
    case down:  return *(gridsD[gridid]); break;
  }
  return *(gridsN[gridid]);
}

bool Storage::hasBorderLayer(const std::string &gridid, Direction dir)
{
  switch (dir)
  {
    case north: return (gridsN.count(gridid) > 0); break;
    case south: return (gridsS.count(gridid) > 0); break;
    case east:  return (gridsE.count(gridid) > 0); break;
    case west:  return (gridsW.count(gridid) > 0); break;
    case up:    return (gridsU.count(gridid) > 0); break;
    case down:  return (gridsD.count(gridid) > 0); break;
  }
  return false;
}

DataGrid *Storage::addBorderLayer(const std::string &gridid, Direction dir, int thickness,
int distance)
{
  DataGrid *g = 0;
  GridMap *gm;

//  std::cerr << "Adding border layer " << gridid << " " << dir << std::endl;
  switch (dir)
  {
    case north: gm = &gridsN; break;
    case south: gm = &gridsS; break;
    case east:  gm = &gridsE; break;
    case west:  gm = &gridsW; break;
    case up:    gm = &gridsU; break;
    case down:  gm = &gridsD; break;
  }

  if (gm->count(gridid) == 0)
  {
    GridIndex b_low, b_high;
    if (getBorderExtent(dir, thickness, distance, b_low, b_high))
    {
      g = new DataGrid(b_low, b_high);
      (*g) = 0;
      gm->operator[](gridid) = g;
    }
  } else
  {
    g = grids[gridid];
  }
  return g;
}
```

```
DataLine &Storage::getLine(const std::string &lineid)
{
  return *(lines[lineid]);
}

bool Storage::hasLine(const std::string &lineid)
{
  return (lines.count(lineid) > 0);
}


DataLine *Storage::addLine(const std::string &lineid, int orientation)
{
  DataLine *ln = 0;
  if (lines.count(lineid) == 0)
  {
    ln = new DataLine(low[orientation], high[orientation]);
    (*ln) = 0;
    lines[lineid] = ln;
  } else
  {
    ln = lines[lineid];
  }
  return ln;
}

template<class Func>
void Storage::forAllGrids(const Func &func)
{
  forAllGrids(grids, func);
}

template<class Func>
void Storage::forAllGrids(GridMap &gm, Func &func)
{
  for (GridMap::iterator it=gm.begin(); it!=gm.end(); ++it)
  {
    func(it->first, it->second);
  }
}


void Storage::addToGroup(
    const std::string &groupid,
    const std::string &gridid
  )
{
  groups[groupid].push_back(gridid);
}

void Storage::applyBoundary(const std::string &groupid)
{
  IdList &gr = groups[groupid];
  for (IdList::iterator it=gr.begin(); it!=gr.end(); ++it)
  {
    DataGrid &g = *(grids[*it]);
    boundary->exchangeX(g);
    boundary->exchangeY(g);
    boundary->exchangeZ(g);
  }
}



bool Storage::getBorderExtent
  (
```

```
    Direction dir,
    int thickness,
    int distance,
    GridIndex &blow,
    GridIndex &bhigh
  )
{
  bool haveBorder = false;
  GridIndex glow  = Globals::instance().gridLow();
  GridIndex ghigh = Globals::instance().gridHigh();

  blow = low + GridIndex(distance,distance,distance);
  bhigh = high - GridIndex(distance,distance,distance);

  switch (dir)
  {
    case west:
      if (low[0]<glow[0]+thickness+distance)
      {
        bhigh[0] = glow[0]+thickness-1+distance;
        haveBorder = true;
      }
      break;
    case south:
      if (low[1]<glow[1]+thickness+distance)
      {
        bhigh[1] = glow[1]+thickness-1+distance;
        haveBorder = true;
      }
      break;
    case down:
      if (low[2]<glow[2]+thickness+distance)
      {
        bhigh[2] = glow[2]+thickness-1+distance;
        haveBorder = true;
      }
      break;
    case east:
      if (high[0]>ghigh[0]-thickness-distance)
      {
        blow[0] = ghigh[0]-thickness+1-distance;
        haveBorder = true;
      }
      break;
    case north:
      if (high[1]>ghigh[1]-thickness-distance)
      {
        blow[1] = ghigh[1]-thickness+1-distance;
        haveBorder = true;
      }
      break;
    case up:
      if (high[2]>ghigh[2]-thickness-distance)
      {
        blow[2] = ghigh[2]-thickness+1-distance;
        haveBorder = true;
      }
      break;
  }
  /*
  std::cerr << "Function getBorderExtent: " << dir << " " << thickness << std::endl;
  std::cerr << "Grid Low " << glow[0] << " " << glow[1] << " " << glow[2] << std::endl;
  std::cerr << "Grid High " << ghigh[0] << " " << ghigh[1] << " " << ghigh[2] << std::endl;
  std::cerr << "Border Low " << blow[0] << " " << blow[1] << " " << blow[2] << std::endl;
  std::cerr << "Border High " << bhigh[0] << " " << bhigh[1] << " " << bhigh[2] << std::end
l;
```

```
  */
  return haveBorder;
}

void Storage::GridDeleter::operator()(std::string, DataGrid* grid)
{
  delete grid;
}

Storage::GridResizer::GridResizer(const GridIndex &low_, const GridIndex &high_)
  : low(low_), high(high_) {}

void Storage::GridResizer::operator()(std::string, DataGrid* grid)
{
  grid->resize(low,high);
}
```

```
#include "waveinit.h"
#include "globals.h"
#include "storage.h"



void PlaneWaveInit::init(Storage &fields)
{
  std::cerr << "=== INITIALIZING Plane Wave ===\n";
  double ex = ky*bz-kz*by;
  double ey = kz*bx-kx*bz;
  double ez = kx*by-ky*bx;
  double bmag = sqrt(bx*bx + by*by + bz*bz);
  double factor = -bmag/sqrt(ex*ex + ey*ey + ez*ez);

  ex *= factor;
  ey *= factor;
  ez *= factor;

  GridIndex gridLow = Globals::instance().gridLow();
  GridIndex gridHigh = Globals::instance().gridHigh();

  DataGrid &Ex = fields.getGrid("Ex");
  DataGrid &Ey = fields.getGrid("Ey");
  DataGrid &Ez = fields.getGrid("Ez");
  DataGrid &Bx = fields.getGrid("Bx");
  DataGrid &By = fields.getGrid("By");
  DataGrid &Bz = fields.getGrid("Bz");

  GridIndex low = fields.getLow();
  GridIndex high = fields.getHigh();

  double nkx = 2*M_PI*kx/double(gridHigh[0] - gridLow[0] - 1);
  double nky = 2*M_PI*ky/double(gridHigh[1] - gridLow[1] - 1);
  double nkz = 2*M_PI*kz/double(gridHigh[2] - gridLow[2] - 1);

  for (int i=low[0]; i<=high[0]; ++i)
    for (int j=low[1]; j<=high[1]; ++j)
      for (int k=low[2]; k<=high[2]; ++k)
      {
        double ampe = cos(i*nkx + j*nky + k*nkz);
        double ampb = cos((i+0.5)*nkx + (j+0.5)*nky + (k+0.5)*nkz);

        Ex(i,j,k) = ex*ampe;
        Ey(i,j,k) = ey*ampe;
        Ez(i,j,k) = ez*ampe;

        Bx(i,j,k) = bx*ampb;
        By(i,j,k) = by*ampb;
        Bz(i,j,k) = bz*ampb;
      }

}


ParameterMap* PlaneWaveInit::MakeParamMap (ParameterMap* pm) {
  pm = Rebuildable::MakeParamMap(pm);
  (*pm)["kx"] = WParameter(new ParameterValue<int>(&kx,1));
  (*pm)["ky"] = WParameter(new ParameterValue<int>(&ky,1));
  (*pm)["kz"] = WParameter(new ParameterValue<int>(&kz,1));
  (*pm)["Bx"] = WParameter(new ParameterValue<double>(&bx,1));
  (*pm)["By"] = WParameter(new ParameterValue<double>(&by,1));
  (*pm)["Bz"] = WParameter(new ParameterValue<double>(&bz,1));

  return pm;
}
```

```
template<class Type, class StreamType>
void SimpleDiagnostic<Type,StreamType>::open(const std::string &fname)
{
  output.open(fname.c_str());
}

template<class Type, class StreamType>
SimpleDiagnostic<Type,StreamType>::~SimpleDiagnostic()
{
  output.close();
}

template<class Type, class StreamType>
void SimpleDiagnostic<Type,StreamType>::write()
{
  output << *field;
}

template<class Type, class StreamType>
void SimpleDiagnostic<Type,StreamType>::close()
{
  output.close();
}

template<class Type, class StreamType>
void SimpleDiagnostic<Type,StreamType>::setField(Type *fld)
{
  field = fld;
}
```

```
#include "plasmacurrent.h"
#include "cpml_border.h"
#include <boost/function.hpp>
#include <boost/bind.hpp>

//=============================================================
//==========  FDTD_PLRCSolver
//=============================================================


template<class PLRCImplementation>
void FDTD_PLRCSolver<PLRCImplementation>::initStorage(Storage *storage_)
{
  for (typename CurrentSimList::iterator it=this->currentSims.begin();
        it != this->currentSims.end();
        ++it
      )
  {
    (*it)->initCurrents(storage_, this);
  }

  this->coreInitStorage(storage_);

  if (!(this->currents.empty()))
  {
    pJx = this->storage->addGrid("JxTotal");
    pJy = this->storage->addGrid("JyTotal");
    pJz = this->storage->addGrid("JzTotal");
    std::cerr << "Added Currents\n";
  }
  else
  {
    pJx = 0;
    pJy = 0;
    pJz = 0;
  }

  if (!(this->magCurrents.empty()))
  {
    pMx = this->storage->addGrid("MxTotal");
    pMy = this->storage->addGrid("MyTotal");
    pMz = this->storage->addGrid("MzTotal");
    std::cerr << "Added Magnetic Currents\n";
  }
  else
  {
    pMx = 0;
    pMy = 0;
    pMz = 0;
  }
}

template<class PLRCImplementation>
void FDTD_PLRCSolver<PLRCImplementation>::addCurrent(Current *current)
{
  this->currents.push_back(current);
}

template<class PLRCImplementation>
void FDTD_PLRCSolver<PLRCImplementation>::addMagCurrent(Current *current)
{
  this->magCurrents.push_back(current);
}

template<class PLRCImplementation>
void FDTD_PLRCSolver<PLRCImplementation>::stepSchemeInit(double dt)
```

```
{
  initAccumulator(dt);

  stepB(0.5*dt);

  for
  (
    typename CurrentList::iterator it = this->currents.begin();
    it != this->currents.end();
    ++it
  )
  {
    (*it)->stepSchemeInit(dt);
  }

  for
  (
    typename CurrentList::iterator it = this->magCurrents.begin();
    it != this->magCurrents.end();
    ++it
  )
  {
    (*it)->stepSchemeInit(dt);
  }
}

template<class PLRCImplementation>
void FDTD_PLRCSolver<PLRCImplementation>::stepScheme(double dt)
{
  for
  (
    typename CurrentList::iterator it = this->currents.begin();
    it != this->currents.end();
    ++it
  )
  {
    (*it)->stepScheme(dt);
  }
  stepD(dt);

  for
  (
    typename CurrentList::iterator it = this->magCurrents.begin();
    it != this->magCurrents.end();
    ++it
  )
  {
    (*it)->stepScheme(dt);
  }
  stepB(dt);
}

template<class PLRCImplementation>
void FDTD_PLRCSolver<PLRCImplementation>::stepD(double dt)
{
  GridIndex low = this->storage->getLow();
  GridIndex high = this->storage->getHigh();

  double dx = this->storage->getDx();
  double dy = this->storage->getDy();
  double dz = this->storage->getDz();


  /// value of beta_p for the three Lorentz poles
  double beta[3];
  /// value of gamma_p for the three Lorentz poles
```

```
  double gamma[3];

  std::complex<double> I(0,1);
  std::complex<double> phasor[3];

  std::complex<double> chi0[3];
  std::complex<double> xi0[3];

  this->plrcData.sumChi0 = 0;
  this->plrcData.sumXi0  = 0;

  for (int n=0;n<3;++n)
  {
    beta[n] = sqrt(this->LOm2[n] - this->LDelta[n]*this->LDelta[n]);
    gamma[n] = this->LEps[n]*this->LOm2[n]/beta[n];
    phasor[n] = this->LDelta[n] - I*beta[n];
    this->plrcData.Crec[n] = exp(-phasor[n]*dt);
    chi0[n] = (I*gamma[n]*(this->plrcData.Crec[n]-1.0)/phasor[n] );
    xi0[n] = I*gamma[n] *(this->plrcData.Crec[n]*(phasor[n]*dt + 1.0) - 1.0)
             / (phasor[n]*phasor[n]*dt);

    this->plrcData.dchi0[n] = chi0[n]*(1.0-this->plrcData.Crec[n]);
    this->plrcData.dxi0[n]  = xi0[n] *(1.0-this->plrcData.Crec[n]);

    this->plrcData.sumChi0 += std::real(chi0[n]);
    this->plrcData.sumXi0  += std::real(xi0[n]);
  }

  double jx(0), jy(0), jz(0);
  if (this->pJx != 0) sumCurrents();

  for (int i=low[0]+1; i<=high[0]; ++i)
    for (int j=low[1]+1; j<=high[1]; ++j)
      for (int k=low[2]+1; k<=high[2]; ++k)
      {
        if (this->pJx != 0)
        {
          jx = (*this->pJx)(i,j,k);
          jy = (*this->pJy)(i,j,k);
          jz = (*this->pJz)(i,j,k);
        }

        this->plrcStepD(dt, i, j, k, dx, dy, dz, jx, jy, jz);
      }


  this->storage->applyBoundary("E");
}


template<class PLRCImplementation>
void FDTD_PLRCSolver<PLRCImplementation>::initAccumulator(double dt)
{
  DataGrid &Ex = *this->pEx;
  DataGrid &Ey = *this->pEy;
  DataGrid &Ez = *this->pEz;

  GridIndex low = this->storage->getLow();
  GridIndex high = this->storage->getHigh();

  /// value of beta_p for the three Lorentz poles
  double beta[3];
  /// value of gamma_p for the three Lorentz poles
  double gamma[3];

  std::complex<double> I(0,1);
```

```
  std::complex<double> phasor[3];
  std::complex<double> xi0[3];

  for (int n=0;n<3;++n)
  {
    beta[n] = sqrt(this->LOm2[n] - this->LDelta[n]*this->LDelta[n]);
    gamma[n] = this->LEps[n]*this->LOm2[n]/beta[n];
    phasor[n] = this->LDelta[n] - I*beta[n];
    this->plrcData.Crec[n] = exp(-phasor[n]*dt);

    xi0[n] = I*gamma[n] *(this->plrcData.Crec[n]*(phasor[n]*dt + 1.0) - 1.0)
             / (phasor[n]*phasor[n]*dt);

    this->plrcData.dxi0[n]  = xi0[n] *(1.0-this->plrcData.Crec[n]);

  }

  for (int i=low[0]+1; i<=high[0]; ++i)
    for (int j=low[1]+1; j<=high[1]; ++j)
      for (int k=low[2]+1; k<=high[2]; ++k)
  {
    double ex = Ex(i,j,k);
    double ey = Ey(i,j,k);
    double ez = Ez(i,j,k);

    for (int n=0;n<3;++n)
    {
      double &pxr = this->pPsiRx[n]->operator()(i,j,k);
      double &pyr = this->pPsiRy[n]->operator()(i,j,k);
      double &pzr = this->pPsiRz[n]->operator()(i,j,k);

      double &pxi = this->pPsiIx[n]->operator()(i,j,k);
      double &pyi = this->pPsiIy[n]->operator()(i,j,k);
      double &pzi = this->pPsiIz[n]->operator()(i,j,k);

      std::complex<double> px = std::complex<double>(pxr,pxi);
      std::complex<double> py = std::complex<double>(pyr,pyi);
      std::complex<double> pz = std::complex<double>(pzr,pzi);

      px = this->plrcData.dxi0[n]*ex + this->plrcData.Crec[n]*px;
      py = this->plrcData.dxi0[n]*ey + this->plrcData.Crec[n]*py;
      pz = this->plrcData.dxi0[n]*ez + this->plrcData.Crec[n]*pz;

      pxr = std::real(px);
      pyr = std::real(py);
      pzr = std::real(pz);

      pxi = std::imag(px);
      pyi = std::imag(py);
      pzi = std::imag(pz);

    }
  }
}

template<class PLRCImplementation>
void FDTD_PLRCSolver<PLRCImplementation>::stepB(double dt)
{
  GridIndex low = this->storage->getLow();
  GridIndex high = this->storage->getHigh();

  double dx = this->storage->getDx();
  double dy = this->storage->getDy();
  double dz = this->storage->getDz();

  double jx(0), jy(0), jz(0);
```

```
  if (this->pMx != 0) sumMagCurrents();

  for (int i=low[0]; i<high[0]; ++i)
    for (int j=low[1]; j<high[1]; ++j)
      for (int k=low[2]; k<high[2]; ++k)
      {
        if (this->pMx != 0)
        {
          jx = (*this->pMx)(i,j,k);
          jy = (*this->pMy)(i,j,k);
          jz = (*this->pMz)(i,j,k);
        }

        this->plrcStepB(dt, i, j, k, dx, dy, dz, jx, jy, jz);
      }

  this->storage->applyBoundary("B");
}

template<class PLRCImplementation>
void FDTD_PLRCSolver<PLRCImplementation>::sumCurrents()
{
  DataGrid &jxT = *this->pJx;
  DataGrid &jyT = *this->pJy;
  DataGrid &jzT = *this->pJz;

  jxT = 0;
  jyT = 0;
  jzT = 0;

  for
  (
    typename CurrentList::iterator it = this->currents.begin();
    it != this->currents.end();
    ++it
  )
  {
    const DataGrid &jx = *(*it)->getJx();
    const DataGrid &jy = *(*it)->getJy();
    const DataGrid &jz = *(*it)->getJz();

    GridIndex low = jx.getLow();
    GridIndex high = jx.getHigh();
    for (int i=low[0]; i<high[0]; ++i)
      for (int j=low[1]; j<high[1]; ++j)
        for (int k=low[2]; k<high[2]; ++k)
        {
          jxT(i,j,k) += jx(i,j,k);
          jyT(i,j,k) += jy(i,j,k);
          jzT(i,j,k) += jz(i,j,k);
        }
  }
}

template<class PLRCImplementation>
void FDTD_PLRCSolver<PLRCImplementation>::sumMagCurrents()
{
  DataGrid &jxT = *this->pMx;
  DataGrid &jyT = *this->pMy;
  DataGrid &jzT = *this->pMz;

  jxT = 0;
  jyT = 0;
  jzT = 0;

  for
```

```
  (
    typename CurrentList::iterator it = this->magCurrents.begin();
    it != this->magCurrents.end();
    ++it
  )
  {
    const DataGrid &jx = *(*it)->getJx();
    const DataGrid &jy = *(*it)->getJy();
    const DataGrid &jz = *(*it)->getJz();

    GridIndex low = jx.getLow();
    GridIndex high = jx.getHigh();
    for (int i=low[0]; i<high[0]; ++i)
      for (int j=low[1]; j<high[1]; ++j)
        for (int k=low[2]; k<high[2]; ++k)
        {
          jxT(i,j,k) += jx(i,j,k);
          jyT(i,j,k) += jy(i,j,k);
          jzT(i,j,k) += jz(i,j,k);
        }
  }
}

template<class PLRCImplementation>
ParameterMap* FDTD_PLRCSolver<PLRCImplementation>::MakeParamMap (ParameterMap* pm)
{
  pm = FieldSolver::MakeParamMap(pm);
  pm = Implementation::CustomParamMap(pm);

  (*pm)["eps"] = WParameter(new ParameterValue<double>(&this->eps,1.0));

  (*pm)["L1eps"] = WParameter(new ParameterValue<double>(&this->LEps[0],0.0));
  (*pm)["L2eps"] = WParameter(new ParameterValue<double>(&this->LEps[1],0.0));
  (*pm)["L3eps"] = WParameter(new ParameterValue<double>(&this->LEps[2],0.0));

  (*pm)["L1delta"] = WParameter(new ParameterValue<double>(&this->LDelta[0],0.0));
  (*pm)["L2delta"] = WParameter(new ParameterValue<double>(&this->LDelta[1],0.0));
  (*pm)["L3delta"] = WParameter(new ParameterValue<double>(&this->LDelta[2],0.0));

  (*pm)["L1Om2"] = WParameter(new ParameterValue<double>(&this->LOm2[0],0.0));
  (*pm)["L2Om2"] = WParameter(new ParameterValue<double>(&this->LOm2[1],0.0));
  (*pm)["L3Om2"] = WParameter(new ParameterValue<double>(&this->LOm2[2],0.0));

//  (*pm)["plasma"] = WParameter(
//      new ParameterRebuild<PlasmaCurrent, Current>(&this->currents)
//  );

  (*pm)["cpml_border"] = WParameter(
      new ParameterRebuild<CPMLBorder, CurrentSim>(&this->currentSims)
  );

  return pm;
}
```

```
#include <sstream>



template<typename TYPE, int RANK>
HDFistream& HDFistream::operator>>(schnek::Matrix<TYPE, RANK>& m)
{
  std::string dset_name = getNextBlockName();

  long extent[2*RANK];

  typename schnek::Matrix<TYPE, RANK>::IndexType mlow;
  typename schnek::Matrix<TYPE, RANK>::IndexType mhigh;

  H5LTget_attribute_long(file_id,dset_name.c_str(),"extent",extent);

  for (int i=0; i<RANK; ++i)
  {
    mlow[i] = extent[2*i];
    mhigh[i] = extent[2*i+1];
  }

  m.resize(mlow,mhigh);

  TYPE *data = &(*m.begin());

  H5LTread_dataset (file_id, dset_name.c_str(), H5DataType<TYPE>::type, data);

  return *this;
}

template<typename TYPE, int RANK>
HDFostream& HDFostream::operator<< (const schnek::Matrix<TYPE, RANK>& m)
{
  std::string dset_name = getNextBlockName();

  typename schnek::Matrix<TYPE, RANK>::IndexType mdims = m.getDims();
  typename schnek::Matrix<TYPE, RANK>::IndexType mlow = m.getLow();
  typename schnek::Matrix<TYPE, RANK>::IndexType mhigh = m.getHigh();

  hsize_t dims[RANK];
  long extent[2*RANK];

  for (int i=0; i<RANK; ++i)
  {
    dims[i] = mdims[i];
    extent[2*i] = mlow[i];
    extent[2*i + 1] = mhigh[i];
  }

  const TYPE *data = &(*m.cbegin());

  H5LTmake_dataset(file_id, dset_name.c_str(), RANK, dims, H5DataType<TYPE>::type, data);
  H5LTset_attribute_long (file_id, dset_name.c_str(), "extent", extent, 2*RANK);

  return *this;
}
```

```
//the rebuild method template. called by Rebuildable objects
//gets *pValue from stream returns next token
template<class Type>
std::string ParameterValue<Type>::Rebuild (std::istream& in) {
  in >> *pValue;
  std::string strToken;
  in >> strToken;
  return strToken;
}


template<class Type, class BaseType>
std::string ParameterRebuild<Type,BaseType>::Rebuild (std::istream& in) {
  BaseType *val = NewInstance();
  if (value) (*value) = val;
  else if (values) values->push_back(val);


  // Task aus der Datei wiederherstellen
  std::string strToken;
  in >> strToken;
  if ((strToken != "{") || in.eof()) return strToken;
  std::string nextToken = val->Rebuild(in);
  val->finalize();
  return nextToken;
}
```