eFF Programmers’ Guide

Large scale excited electron dynamics

Julius Su and William A. Goddard III (advisor)

April 29, 2008
### 6 Function prototypes and data structures

#### 6.1 eff access

- double GetTotalPE() .................................................. 17
- double GetNucleusPE(int i) ....................................... 17
- double GetElectronPE(int i) ...................................... 18
- void GetNuclearForce(int i, double *fx, double *fy, double *fz) .................................................. 18
- void GetElectronForce(int i, double *fx, double *fy, double *fz, double *fr) ........................................ 18
- void GetNuclearPosition(int i, double *x, double *y, double *z) .................................................. 18
- void SetNuclearPosition(int i, double x, double y, double z) .................................................. 18
- void GetElectronPosition(int i, double *x, double *y, double *z, double *r) ........................................ 18
- void SetElectronPosition(int i, double x, double y, double z, double r) .................................................. 18
- double GetNuclearCharge(int i) .................................... 18
- int GetElectronSpin(int i) ........................................ 18
- void SetElectronSpin(int i, int spin) ........................................ 18
- int NumNuclei() .................................................. 19
- int NumElectrons() .................................................. 19
- void AddNuclearForce(int i, double fx, double fy, double fz) .................................................. 19
- void AddElectronForce(int i, double fx, double fy, double fz, double fr) ........................................ 19
- void AddNucleusPE(int i, double energy) .................................................. 19
- void AddElectronPE(int i, double energy) .................................................. 19
- void SetNucFX(int i, double val) .................................... 19
- void SetNucFY(int i, double val) .................................... 19
- void SetNucFZ(int i, double val) .................................... 19
- void SetElecFX(int i, double val) .................................... 19
- void SetElecFY(int i, double val) .................................... 19
- void SetElecFZ(int i, double val) .................................... 19
- void SetElecFR(int i, double val) .................................... 19

#### 6.2 eff bounds

- void InitializeBounds() .................................................. 20
- int isWrapX() .................................................. 20
- int isWrapY() .................................................. 20
- int isWrapZ() .................................................. 20
- double GetVolume() .................................................. 21
- void ApplyPositionBounds() .................................................. 21

#### 6.3 eff constants

#### 6.4 eff create

- void Initialize(int maxnuclei, int maxelectrons) .................................................. 21
- int AddNucleus(double x, double y, double z, double q) .................................................. 21
- int AddElectron(double x, double y, double z, int spin, double re) .................................................. 21
6.5 eff_driver

6.5.1 void RunSinglePoint()

6.5.2 void RunMinimize()

6.5.3 void MinimizeCallback(int num_iterations, int num_fevals, double f_val, double g_sq)

6.5.4 void RunDynamics()

6.5.5 void AnnealSpins(double start_temperature, double end_temperature, int numiter)

6.6 eff_dynamics

6.6.1 Data structures

6.6.2 void SetDynamicsLogfile(FILE *fp)

6.6.3 void AllocateDynamics()

6.6.4 void SaveDynamics()

6.6.5 void RevertDynamics()

6.6.6 void InitializeDynamics()

6.6.7 void InitializeRandomVelocities(double temp)

6.6.8 void NullCMMotion()

6.6.9 void Dynamics(enum ThermostatType thermostat, double temperature)

6.6.10 void UpdateDynamicsPositions()

6.6.11 void UpdateDynamicsVelocities(enum ThermostatType thermostat, double temperature)

6.6.12 void ApplyNoseHoover(double temperature)

6.6.13 void ApplyNoseHooverNuclei(double temperature)

6.6.14 void GetDynamicsForces()

6.6.15 void GetDynamicsPositions()

6.6.16 void SetDynamicsPositions()

6.6.17 void UpdateDynamicsVirial()

6.6.18 void SetTimeStep(double dt)

6.6.19 double GetTemperature()

6.6.20 void SetNuclearVelocity(int i, double vx, double vy, double vz)

6.6.21 void SetElectronVelocity(int i, double vx, double vy, double vz, double vr)

6.6.22 void GetNuclearVelocity(int i, double *vx, double *vy, double *vz)

6.6.23 void GetElectronVelocity(int i, double *vx, double *vy, double *vz, double *vr)

6.6.24 void SetNuclearMass(int i, double mx, double my, double mz)

6.6.25 void SetElectronMass(int i, double mx, double my, double mz, double mr)

6.6.26 void GetNuclearMass(int i, double *mx, double *my, double *mz)

6.6.27 void GetElectronMass(int i, double *mx, double *my, double *mz, double *mr)
6.6.28 double GetTotalKE() ............................................. 27
6.6.29 double GetNucleusKE(int i) ..................................... 27
6.6.30 double GetElectronKE(int i) ...................................... 27
6.6.31 double GetElectronTranslationKE(int i) ....................... 28
6.7.1 void SetupExternalField(double Ex0, double Ey0, double
Ez0, double Efreq, double Epacket_duration) ..................... 28
6.7.2 void GetExternalField(double t, double *Ex, double *Ey,
double *Ez) ............................................................. 28
6.7.3 void ApplyExternalField(double t) ................................. 28
6.8.1 void jacobi(double *matrix, double eigenvalue[], double
*eigenvector, int n) ..................................................... 29
6.9.1 double erf(double x) .................................................. 29
6.9.2 double erfoverx(double x) .......................................... 29
6.9.3 double erfoverx1(double x, double *df) ......................... 29
6.9.4 double poly01(double x) ............................................ 30
6.9.5 double poly02(double x) ............................................ 30
6.9.6 double poly1(double x) ............................................. 30
6.9.7 double poly2(double x) ............................................. 30
6.10.1 Data structures ...................................................... 31
6.10.2 void AllocateEwald() .............................................. 31
6.10.3 void InitializeEwald(double Lx, double Ly, double Lz, dou-
ble max_alpha, double max_r, double max_k, double nuc_alpha) 
31
6.10.4 void SetEwaldCharges(double nuc_alpha) ....................... 31
6.10.5 double TotalCharge() .............................................. 32
6.10.6 double TotalVolume() .............................................. 32
6.10.7 void AddEwaldEnergyForce(double *fx, double *fy, dou-
ble *fz, double *falpha, double *energy) ......................... 32
6.10.8 void EwaldVirial(double *ewald_energy, double *ewald_falpha) 
32
6.10.9 void UpdateEwaldEnergy() ........................................ 32
6.10.10 void InitializeKSpace() ......................................... 32
6.10.11 void InitializeRSpace() ........................................ 33
6.10.12 void KSpaceEnergy(double *fx, double *fy, double *fz,
double *falpha, double *energy) ................................... 33
6.10.13 void RSpaceEnergy(double *fx, double *fy, double *fz,
double *falpha, double *energy) ................................... 33
6.10.14 void SelfEnergy(double *falpha, double *energy) ............ 34
6.10.15 void UniformChargeEnergy(double *falpha, double *en-
ergy) ................................................................. 34
6.10.16 double EwaldRCutoff(double precision, double alpha_cutoff,
double min_alpha) ................................................... 34
6.10.17 double EwaldKCutoff(double precision, double alpha_cutoff) 
34
6.11 void freeze ............................................................ 34
6.11.1 Data structures ........................................ 35
6.11.2 void InitializeFreezes() ............................... 35
6.11.3 void ClearFreezes() ..................................... 35
6.11.4 void AddFreezeNucX(int i) ............................. 35
6.11.5 void AddFreezeNucY(int i) ............................. 35
6.11.6 void AddFreezeNucZ(int i) ............................. 35
6.11.7 void AddFreezeElecX(int i) ............................ 35
6.11.8 void AddFreezeElecY(int i) ............................ 35
6.11.9 void AddFreezeElecZ(int i) ............................ 35
6.11.10 void AddFreezeElecR(int i) .......................... 35
6.11.11 void ApplyFreezes() ................................... 35

6.12 eff_global ................................................. 36
   6.12.1 Nucleus data .......................................... 36
   6.12.2 Electron data ......................................... 36
   6.12.3 Parameters ........................................... 37

6.13 eff_initial ................................................ 37
   6.13.1 Data structures ....................................... 39
   6.13.2 void AllocateInitialVelocities(int max_nuc_v, int max_elec_v) 39
   6.13.3 void AllocateInitialMasses(int max_nuc_m, int max_elec_m) 39
   6.13.4 int GetNumNucV() .................................. 39
   6.13.5 int GetNumElecV() .................................. 39
   6.13.6 int GetNumNucM() .................................. 39
   6.13.7 int GetNumElecM() .................................. 39
   6.13.8 void AddNucV(int i, double vx, double vy, double vz) ........ 40
   6.13.9 void AddElecV(int i, double vx, double vy, double vz, double vr) ........................................ 40
   6.13.10 void GetNucV(int idx, int *i, double *vx, double *vy, double *vz) ........................................ 40
   6.13.11 void GetElecV(int idx, int *i, double *vx, double *vy, double *vz, double *vr) ........................................ 40
   6.13.12 void AddNucM(int i, double mx, double my, double mz) .... 40
   6.13.13 void AddElecM(int i, double mx, double my, double mz, double mr) ........................................ 40
   6.13.14 void GetNucM(int idx, int *i, double *mx, double *my, double *mz) ........................................ 40
   6.13.15 void GetElecM(int idx, int *i, double *mx, double *my, double *mz, double *mr) ........................................ 40

6.14 eff_input .................................................. 41
   6.14.1 int CountLines(FILE *fp, char *header) ............... 41
   6.14.2 void ProcessInputFile(FILE *fp) .................... 41
   6.14.3 void ProcessNucleus(char *str) ...................... 41
   6.14.4 void ProcessElectron(char *str) .................... 41
   6.14.5 void ProcessNucVelocities(char *str) ............... 41
   6.14.6 void ProcessElecVelocities(char *str) ............... 41
   6.14.7 void ProcessNucMasses(char *str) ................... 42
   6.14.8 void ProcessElecMasses(char *str) ................... 42
6.14.9 void ProcessParam(char *str) .......................... 42
6.14.10 void ProcessRestraints(char *str) ..................... 42
6.14.11 void NuclElecIndex(char *str, int *index) ............ 42
6.14.12 int string_match(char *str1, char *str2) ............. 42
6.14.13 void StripNewline(char *str) ......................... 42
6.14.14 void ParseString(char *buffer, int *argc, char **argv) 42

6.15 eff_listint .............................................. 43
6.15.1 Data structure ......................................... 43
6.15.2 LIST_INT *new_list_int() .............................. 43
6.15.3 void delete_list_int(LIST_INT *list) ................... 44
6.15.4 void add_list_int(LIST_INT *list, int newelement) ... 44
6.15.5 void clear_list_int(LIST_INT *list) .................... 44

6.16 eff_listptr .............................................. 44
6.17 eff_minimize ............................................. 44
6.17.1 Data structures ........................................ 44
6.17.2 void AllocateMinimize(enum MinimizeType minmethod) .... 45
6.17.3 int MinStorageRequirements(enum MinimizeType minmethod) 45
6.17.4 void CalcContractFG(double *x, double *f, double *g) ... 45
6.17.5 enum MinimizeResult Minimize(enum MinimizeType s_minmethod, double eps, double acc, int max_numsteps, int print_every,
  void (*MinCallback)(int, int, double, double)) ........... 46
6.17.6 void GetMinimizeForces(double *forces) ................ 46
6.17.7 void GetInitialPosition(double *positions) ............. 46
6.17.8 void SetMinimizePositions(double *positions) .......... 46

6.18 eff_output ............................................... 46
6.18.1 void OutputEnergyForces(FILE *out_fp) .................. 47
6.18.2 void OutputPositions(FILE *out_fp) ..................... 47
6.18.3 void OutputVelocities(FILE *out_fp) .................... 47
6.18.4 void OutputEnergy(FILE *out_fp) ....................... 47
6.18.5 void OutputTimeElapsed(FILE *out_fp) ................... 47
6.18.6 void OutputRestartFile(char *restart_filename) ......... 47
6.18.7 void OutputFixedHeader(FILE *out_fp) .................. 47
6.18.8 void OutputDynamicsMasses(FILE *out_fp) ............... 47
6.18.9 void OutputParams(char *tag, FILE *out_fp) ............ 47
6.18.10 void PrintAnalyticForces() ............................ 48
6.18.11 void PrintNumericalForces() ........................... 48

6.19 eff_params ............................................... 48
6.19.1 Data structures ........................................ 48
6.19.2 void InitializeParameters() ............................ 49
6.19.3 void AddParameter(char *name, void (*fp)(int, char **)) 49
6.19.4 void ExecuteParameter(char *name, int argc, char **argv) 49
6.19.5 void SetDefaultParameters() ........................... 49
6.19.6 Parsing functions ....................................... 49
6.19.7 void ReadValue(char *str, double *val) ................. 51

6.20 eff_pressure ............................................. 51
6.20.1 Data structures ........................................ 51
6.20.2 void ClearVirial() ........................................... 52
6.20.3 void AddForceVirial(double rx, double ry, double rz, double rr, double fx, double fy, double fz, double fr, enum ElectronType electron_type) ........................................ 52
6.20.4 void AddSizeForceVirial(double r, double f, enum ElectronType electron_type) ........................................ 52
6.20.5 void AddRadialForceVirial(double r, double f, enum ElectronType electron_type) ........................................ 52
6.20.6 void AddPotentialEnergyVirial(double energy, enum ElectronType electron_type) ........................................ 52
6.20.7 void AddKineticEnergyVirial(double energy, enum ElectronType electron_type) ........................................ 52
6.20.8 double GetRigidPEPressure(double volume) ............... 52
6.20.9 double GetFlexiblePEPressure(double volume) .............. 52
6.20.10 double GetRigidKEPressure(double volume) ............... 53
6.20.11 double GetFlexibleKEPressure(double volume) ............ 53
6.21 eff_properties ................................................... 53
6.21.1 void CalcDipole(double *dipole_x, double *dipole_y, double *dipole_z) ........................................ 53
6.21.2 double ElectronDensity(double x, double y, double z) ... 53
6.22 eff_restrain ..................................................... 53
6.22.1 Data structures ............................................... 54
6.22.2 void InitializeRestraints() .................................. 55
6.22.3 void ClearRestraints() ....................................... 55
6.22.4 void AddCoordinateRestraint(int idx, double x, double y, double z) ........................................ 55
6.22.5 void AddDistanceRestraint(int idx1, int idx2, double dist) ........................................ 55
6.22.6 void AddAngleRestraint(int idx1, int idx2, int idx3, double angle) ........................................ 55
6.22.7 void AddDihedralRestraint(int idx1, int idx2, int idx3, int idx4, double angle) ........................................ 55
6.22.8 void AddLineRestraint(int *indices, int numindices) ..... 55
6.22.9 void AddPlaneRestraint(int *indices, int numindices) .... 56
6.22.10 void AddTetherRestraint(int *indices, int numindices) ... 56
6.22.11 double AddApplyRestraints() ............................... 56
6.22.12 void ApplyTetherVelocityRestraints() ..................... 56
6.22.13 void AddsNucElecForce(int idx, double fx, double fy, double fz) ........................................ 56
6.22.14 void GetNucElecPosition(int idx, double *x, double *y, double *z) ........................................ 56
6.22.15 void GetNucElecForce(int idx, double *fx, double *fy, double *fz) ........................................ 56
6.22.16 void GetNucElecMass(int idx, double *mx, double *my, double *mz) ........................................ 56
6.22.17 void GetNucElecVelocity(int idx, double *vx, double *vy, double *vz) ........................................ 57
6.22.18 void SetNucElecVelocity(int idx, double vx, double vy,
  double vz). .................................. 57
6.22.19 void NucElecString(char *str, int idx). ............... 57
6.22.20 double ApplyCoordinateRestraint(int idx, double x0, dou-
  ble y0, double z0, double *dist). ............... 57
6.22.21 double ApplyDistanceRestraint(int idx1, int idx2, double
  dist0, double *dist). .......................... 57
6.22.22 double ApplyAngleRestraint(int idx1, int idx2, int idx3,
  double angle0, double *angle). ................. 57
6.22.23 double ApplyDihedralRestraint(int idx1, int idx2, int idx3,
  int idx4, double d0, double *angle). .......... 57
6.22.24 double ApplyLineRestraint(int *indices, int numindices,
  double *dists). ................................ 58
6.22.25 double ApplyPlaneRestraint(int *indices, int numindices,
  double *dists). ................................ 58
6.22.26 double ApplyTetherRestraint(int *indices, int numindices,
  double *dists). ................................ 58
6.22.27 double ApplyTetherRestraintVelocity(int *indices, int nu-
  mindices). ..................................... 58
6.22.28 void OutputRestraints(FILE *out_fp). .................. 58
6.22.29 void OutputRestartRestraints(FILE *restart_fp) ...... 58

6.23 eff

6.23.1 int conmin(int n, double *x, double *f, double *g, int
  *ifun, int *iter, double eps, int mxfun, double *w, int iout,
  int ndim, double acc, int numth, void (*calcf)(double *
  double *, double *, double *), void (*update_func)(int, int, dou-
  ble, double)). .................................. 59
6.23.2 double dmin(double a, double b). ...................... 60
6.23.3 double dmax(double a, double b). ...................... 60

6.24 eff
timing

6.24.1 void StartTimer(). ................................ 60
6.24.2 void StopTimer(). ................................ 60
6.24.3 double CPUTimeElapsed(). ........................... 60
6.24.4 double TotalTimeElapsed(). ......................... 60

6.25 eff
update

6.25.1 Data structures .................................. 62
6.25.2 void InitializeEnergyUpdater(). ..................... 62
6.25.3 int LeftStencil(int numbins, int pbc). ............. 62
6.25.4 int RightStencil(int numbins, int pbc). ............ 63
6.25.5 void AssignParticleBins(). ........................ 63
6.25.6 void UpdateEnergyForces(). ......................... 63
6.25.7 void UpdateKineticEnergy(). ....................... 63
6.25.8 void UpdateElecAndPauli(int bin1, int bin2, int wrap_x,
  int wrap_y, int wrap_z). ...................... 63
6.25.9 void UpdatePauliPeriodic(). ....................... 63
6.25.10 void UpdateWidthRestraint(). .................... 63
6.26  eff_util .......................................................... 64
  6.26.1  void error(const char *format, ...) ....................... 64
  6.26.2  int rand_int(int max) ...................................... 64
  6.26.3  double rand_uni() ......................................... 64
  6.26.4  double rand_gauss() ...................................... 64
  6.26.5  void lcase(char *str) ................................... 64
  6.26.6  double GetMass(int Z) ................................... 64
  6.26.7  char *GetType(int Z) ..................................... 64
  6.26.8  int GetZ(char *str) ...................................... 64
  6.26.9  void GetBasename(char *str, char *basename) ............ 64
  6.26.10 void GetExtension(char *str, char *extension) .......... 65
  6.26.11 void rand_seed(unsigned int seed) ....................... 65
  6.26.12 double min(double x, double y) ......................... 65
List of Tables

2.1 eFView dependencies ............................................. 4
6.1 Parameter global data structure, part I .......................... 37
6.2 Parameter global data structure, part II .......................... 38
6.3 Parameter parsing functions and parameter names ................ 50
6.4 Arguments for Shanno’s conmin function in eff_shanno ........ 59
6.5 Error codes for Shanno’s conmin function in eff_shanno ......... 59
Chapter 1

License and dependencies

1.1 License

The eFF and eFFview software, as well as the accompanying utility scripts, are distributed under the following free software license (MIT/X11 license):

Copyright (c) 2008 Julius Su and William A. Goddard III, California Institute of Technology.

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
1.2 eFF dependencies

The eFF software incorporates code derived from the following sources:


- **Chebyshev expansion.** Chebyshev polynomial expansion and bilinear mapping from J. Schonfelder, *Math. Comp.* 32(144):1232-40 (1978), evaluated using a code adapted from the public domain SLATEC/FNLIB routine csevl.f by W. Fullerton, which uses the Clenshaw method to perform the expansion.

- **Eigenvalue/eigenvector solver.** The Jacobi routine was translated from an algorithm by H. Rutishauser in *Numerische Mathematik* 9: 1-10 (1966).

1.3 eFFview dependencies

The eFFView software links to the following software libraries:

- **OpenGL and OpenGL utilities** by Silicon Graphics (no license needed for applications).

- **OpenGL utility toolkit (GLUT)** by Mark Kilgard (freely redistributable). Patched versions to support scroll wheel use in Mac OS X (Renaud Blanch) and Windows (www.realtech.net); freeglut used for Linux (freeglut.sourceforge.net).

- **OpenGL extension wrangler library (GLEW)** by Milan Ikits and Marcello Magallon (freely redistributable).

- **libpng** by Guy Eric Schalnat, Andreas Dilger, Glenn Randers-Pehrson, and others (freely redistributable).

- **zlib** by Jean-loup Gailly and Mark Adler (freely redistributable).
Chapter 2

Recompiling the software

We have successfully compiled eFF and eFFview on Windows (Microsoft Visual Studio 2005), Mac OS X 10.5 (gcc), and Red Hat Enterprise Linux 5 (gcc and Intel C). We list below the steps we used.

2.1 Windows

1. Compile eFF. Open the file `\eff\distro\eff\eFF.sln` in Visual Studio, and click on Build .. Build Solution (F7).

2. Compile eFFview. Open the file `\eff\distro\effview\eFFview.sln` in Visual Studio, and click on Build .. Build Solution (F7).

2.2 Mac OS X and Linux

Here we assume the root directory of the distribution is `/Software/eff_distro`.

1. Compile eFF:

   $ cd ~/Software/eff_distro/eff/src
   $ make

2. Compile eFFview:

   $ cd ~/Software/eff_distro/effview/src
   $ make

2.3 eFFview external dependencies

We have taken care to include with the distribution most of the external libraries needed to run and compile the eFFview program, such as GLUT, GLEW, and
libpng. They are linked in as either dynamic libraries whose pathnames are encoded into the binary directly, as in Mac OS X and Windows; or as static libraries, as in Linux.

The exception is the OpenGL and GLU (OpenGL Utility) libraries, which are built in to most modern operating systems.

The existing make system handles all of the dependencies properly, and should require no additional modification from the user. Nonetheless we have provided a list of eFFview dependencies in Table 2.1 not including OpenGL and GLU.

<table>
<thead>
<tr>
<th>Package</th>
<th>Include</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Windows</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLUT</td>
<td>./glut</td>
<td>./glut/glut32.lib, glut32.dl</td>
</tr>
<tr>
<td>GLEW</td>
<td>./glew/include</td>
<td>./glew/lib/glew32.lib</td>
</tr>
<tr>
<td>libpng</td>
<td>./libpng/</td>
<td>./libpng/libpng.lib</td>
</tr>
<tr>
<td>zlib</td>
<td>./libpng/zlib</td>
<td>./libpng/zlib/zlib.lib</td>
</tr>
<tr>
<td><strong>Mac OS X</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLUT</td>
<td>GLUT.framework</td>
<td></td>
</tr>
<tr>
<td>GLEW</td>
<td>./glew/include</td>
<td>./glew/lib/GLEW.dylib, -lglew</td>
</tr>
<tr>
<td>libpng</td>
<td>./libpng/include</td>
<td>./libpng/lib/libpng.dylib, -lpng</td>
</tr>
<tr>
<td><strong>Linux</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GLUT</td>
<td>./freeglut/include</td>
<td>./freeglut/lib/libglut.a</td>
</tr>
<tr>
<td>GLEW</td>
<td>./glew/include</td>
<td>./glew/lib/libGLEW.a</td>
</tr>
<tr>
<td>libpng</td>
<td>./libpng/include</td>
<td>./libpng/lib/libpng12.a</td>
</tr>
<tr>
<td>zlib</td>
<td>./zlib/include</td>
<td>./zlib/lib/libz.a</td>
</tr>
</tbody>
</table>

Table 2.1: eFFview dependencies
Chapter 3

To modify current features

3.1 Pauli potential

To modify the Pauli potential, change the inline function `PauliElecElec()` in `eff_update`, which computes the Pauli repulsion between two electrons. Be sure to update the analytic gradients as well as the energy. The function takes as input `samespin`, a flag which is 1 if the two spins are the same; `rc`, the distance between the two electrons; and `re1` and `re2`, the sizes of the two electrons. The function outputs an `energy`, and forces `frc`, `fre1`, and `re2` that are the negative gradients of energy with respect to distance and electron sizes. Arguments `i` and `j` are the indices of the electrons, but they are not used.

No other changes need to be made. As a note, the `PauliElecElec()` is called by `UpdatePauliPeriodic()` for Ewald-type calculations, and `UpdateElecAndPauli()` for all other calculations.

3.2 Kinetic energy

To modify the kinetic energy function, change the inline function `UpdateKineticEnergy()` in `eff_update`, which computes the kinetic energy of all the electrons.

The function does not take any arguments or return any values, but instead modifies directly the global data structure values `elec[i].energy`, the individual electron energy, and `elec[i].fr`, the individual electron radial force. If the kinetic energy depends on other coordinates, the forces or negative gradients relative to those coordinates need to be added too.

The function also calls `AddSizeForceVirial()` from `eff_pressure` to update the potential energy portion of the pressure virial, $-r_i \cdot f_i$. Be sure to properly update this value. If the kinetic energy depends on other coordinates, the more general virial update function `AddForceVirial()` should be used instead.
3.3 Non-Ewald electrostatics

To modify the pairwise electrostatics functions, change any of three inline functions: \texttt{ElecNucNuc()}, for nucleus-nucleus repulsion; \texttt{ElecNucElec()}, for nucleus-electron attraction; and \texttt{ElecElecElec()}, for electron-electron repulsion. The arguments of these functions are similar to those in the Pauli potential described above. Beyond these functions, no other changes need to be made.

The current functions describe the interactions between Gaussian charge distributions, which are screened Coulomb potentials of the form:

\[ V = \frac{\text{erf}(ax)}{x} \]

Functions such as \texttt{ierfoverx1()} compute the error function ratio and the first derivative. If one wishes to describe interactions involving other charge distributions, other auxiliary functions will be needed.

In our experience, the electrostatic energy has been the most difficult component of eFF to modify successfully. The Gaussian functions lack nuclear cusps and die off too quickly, but attempts to introduce these features by changing the electrostatic expressions have failed – there is a delicate cancellation of errors that is easily upset.

3.4 Ewald electrostatics

To modify the Ewald sum, change the real space sum \texttt{RSpaceEnergy()}, the reciprocal space sum \texttt{KSpaceEnergy()}, the self interaction \texttt{SelfEnergy()}, and the uniform background/charge interaction \texttt{UniformChargeEnergy()}.

All of these functions take as input the arrays \texttt{ewald\_q}, representing the charges, and \texttt{ewald\_alpha}, representing the exponents of the Gaussian charges. They output an energy, and its gradients with respect to position and exponent, \( f_x, f_y, f_z \), and \( f_{alpha} \).

If the goal is to use different kinds of charge distributions, new expressions for all of the above sums need to be derived. This includes an expression for the Fourier transform of the charge distribution, in order to compute the reciprocal space sum:

\[ E_{k-space} = \frac{1}{2V} \int dk \frac{4\pi}{k^2} |\rho(k)|^2 \]

In the current eFF, \( \rho(k) \) is a simple single Gaussian function, since the Fourier transform of a Gaussian is another Gaussian. The Fourier transform of other charge distributions may contain multiple terms, making the sum tedious to evaluate. In such cases, it may be useful to use other approaches to solving the Poisson equation, such as a Fast Fourier Transform on a reciprocal-space grid, or a multigrid or conjugate-gradient solver over a real-space grid. Such approaches have also been used to handle interactions more general than \( V = -1/r \), as in the Poisson-Boltzmann equation.
On the other hand, it may be difficult for grid-based algorithms to handle continuously varying electron sizes. The Ewald algorithm we have implemented, which handles charges of variable size, is slow partly because quantities such as \(e^{-k^2/4\alpha}\) are not precomputed, as they would be in a more traditional Ewald algorithm with fixed charge sizes. There may be a way to interpolate between precomputed values of this exponential function to make our algorithm more efficient.

As long as the interactions are all \(-1/r\), the pressure virial has the simple form \(\sum_i r_i \cdot f_i = U\), where \(U\) is the potential energy, and no further changes need to be made to \texttt{EwaldVirial()}. 

### 3.5 Pairwise cutoff function

To modify the pairwise cutoff function, change the inline functions \texttt{cutoff()} and \texttt{dcutoff()}, which are the cutoff function and its derivative respectively.

The function must have \(f(0) = 1\) and \(f(1) = 0\), and several of the higher order derivatives at the boundaries should be zero as well. The current cutoff function is a seventh-order spline; other lower-order splines are included in the code and commented out.

### 3.6 Integration algorithm

To modify the integration algorithm, change the functions \texttt{UpdateDynamicsPositions()} and \texttt{UpdateDynamicsVelocities()} in \texttt{eff\_dynamics}. These two functions implement the two equations in the velocity Verlet algorithm:

\[
\begin{align*}
x_{i+1} &= v_i \Delta t + \frac{1}{2m} f_i \Delta t^2 \\
v_{i+1} &= m \Delta t (f_i + f_{i+1})
\end{align*}
\]

The positions are stored in \texttt{dyn\_x}, the velocities are stored in \texttt{dyn\_v}, and the forces are stored in \texttt{dyn\_f}. The program currently keeps one previous iteration of forces as well, \texttt{dyn\_prev\_f}.

To allocate additional arrays, modify \texttt{AllocateDynamics()}. Be sure also to duplicate these arrays for the functions \texttt{SaveDynamics()} and \texttt{RevertDynamics()}, used by the adaptive step size procedure in \texttt{eff\_driver} to roll back to a previous iteration when needed.

### 3.7 Thermostat

Currently two thermostats are implemented, Nosé-Hoover in \texttt{ApplyNoseHooverNuclei()} and the Andersen thermostat as a part of \texttt{UpdateDynamicsVelocities()}.

The Nosé-Hoover thermostat could be improved by turning it into a chain thermostat. Currently, the temperature degrees of freedom are represented
by a single damping factor $\zeta$, which is modulated up and down to adjust the instantaneous temperature to match the desired temperature:

$$\Delta f_i = -\zeta m_i v_i$$
$$\Delta \zeta = \Delta t \left( \sum_i \frac{1}{2} m_i v_i^2 - \frac{3}{2} N k_B T \right)$$

In a chain thermostat, there are several damping factors $\zeta_1 \ldots \zeta_N$. $\zeta_1$ acts as a damping factor for the entire system, $\zeta_2$ acts as a damping factor for $\zeta_1$, $\zeta_3$ for $\zeta_2$ and so on.

To add such a thermostat, it would be best to make a separate function to update a set of thermostat global variables, and to call it at the same point the \texttt{ApplyNoseHooverNuclei()} function is called. The function would also apply the damping factor to the overall system, as it does now.

It may be useful to thermostat the electron degrees of freedom as well as the nuclear degrees of freedom.
Chapter 4

To add new features

4.1 Add a new parameter

To add a new parameter, first make an entry for it in the params data structure in eff_global:

```c
typedef struct {
    ...
    double new_param;
} PARAMS;
```

Then create a parsing function for it in eff_params:

```c
void ParamNewParam(int argc, char **argv) {
    if (argc != 3) error("Usage: new_param = [new parameter]\n");
}
```

In the above example, the parameter appears in the input .cfg file as “new_param = [value]”. However, variations are possible – there may be more arguments, the arguments may be mixtures of strings and integers – which require a more complex parsing function.

Next, associate the parsing function with a parameter name in AddParameters() in eff_params:

```c
void InitializeParameters() {
    ...
    AddParameter("new_param", ParamNewParam);
    ...
}
```
Finally, place an entry in `OutputFixedHeader()` in `eff_output` so that the parameter is printed out at the start of the `.eff` output file and `.cfg.restart` restart file:

```c
void OutputParams(char *tag, FILE *out_fp)
{
    ...
    fprintf(out_fp, "%s\tnew_param = %f\n", tag, params.new_param);
    ...
}
```

Now the parameter can be accessed as `params.new_param` in any file that includes `eff_global.h`.

### 4.2 Add a new restraint

To add a new restraint, first make a new data structure for it in `eff_restraints`:

```c
typedef struct
{
    int idx1, idx2, idx3;
    double value;
} RES_NEW;
```

Here we have assumed that the restraint operates between three particles. If it operates between a variable number of particles, use a structure of the following form instead:

```c
typedef struct
{
    int *indices;
    int numindices;
    double value;
} RES_NEW;
```

Next, declare a new global variable of type `LIST_PTR *` to serve as a restraint list, and add statements to initialize and clear it in `InitializeRestraints()` and `ClearRestraints()`:

```c
LIST_PTR *restraint_new; // in the global scope
restraint_new = new_list_ptr(); // in InitializeRestraints()
clear_list_ptr(restraint_new); // in ClearRestraints()
```

Then create a function for adding the restraint to the restraint list:

```c
void AddNewRestraint(int idx1, int idx2, int idx3, double value)
{
    RES_NEW *ptr = (RES_NEW *) malloc(sizeof(RES_NEW));
```
ptr->idx1 = idx1;
ptr->idx2 = idx2;
ptr->idx3 = idx3;
ptr->value = value;
add_list_ptr(restraint_new, ptr);
}

If the restraint operates on a variable of particles, use AddLineRestraint() as a model for your function instead.

Now we are ready to set up the function which actually enforces the restraint through use of a restraining potential, usually harmonic. Create a function called ApplyNewRestraint():

```c
double ApplyNewRestraint(int idx1, int idx2, int idx3,
                           double value0, double *value)
{
    // Gets the positions of the particles
double x1, y1, z1, x2, y2, z2, x3, y3, z3;
GetNucElecPosition(idx1, &x1, &y1, &z1);
GetNucElecPosition(idx2, &x2, &y2, &z2);
GetNucElecPosition(idx3, &x3, &y3, &z3);

    // Computes the forces on the particles
double fx1, fy1, fz1, fx2, fy2, fz2, fx3, fy3, fz3;
...

    if (value) *value = restraint_measure;

    // Applies forces to the particles
AddNucElecForce(idx1, fx1, fy1, fz1);
AddNucElecForce(idx2, fx2, fy2, fz2);
AddNucElecForce(idx3, fx3, fy3, fz3);
}
```

Next add the above function to the main ApplyRestraints() function, making sure it is inserted above the ApplyTetherRestraints() function, which needs to be applied last to work.

```c
for (ptr = restraint_new->head; ptr != 0; ptr = ptr->next)
{
    RES_NEW *element = (RES_NEW *) ptr->element;
    restraints_energy += ApplyNewRestraint(
        element->idx1, element->idx2, element->idx3,
        element->value, 0);
}
```

Finally, add a similar loop to the OutputRestartRestraints() function:
for (ptr = restraint_new->head; ptr != 0; ptr = ptr->next)
{
    RES_NEW *element = (RES_NEW *) ptr->element;
    NucElecString(str1, element->idx1);
    NucElecString(str2, element->idx2);
    NucElecString(str3, element->idx3);
    fprintf(restart_fp, "new %s %s %s %f\n", str1, str2, str3, element->value);
}

4.3 Add a new property to calculate

To calculate a new property, add a function to eff_properties. Then use functions defined in eff_access to get nuclear and electron positions and forces, or functions in eff_dynamics to get velocities and masses.

If the property is only to be computed once, at the end of the calculation, add the property function to OutputEnergy() in eff_output.

Otherwise, if the property is to be computed and printed out every print_every iterations, create a new function in eff_output that outputs the property:

```c
void OutputNewProperty(FILE *out_fp)
{
    fprintf(out_fp, "[new_property]\t%f\n", CalcNewProperty());
}
```

Use the instructions in Section 4.1 to add a new parameter called “output_new_property”, added in eff_global as type “enum OutputType”. Then the user will be able to set the parameter to NONE, ALL, or END to determine how often the property will be printed.

Do a search for OutputPositions in eff_driver, and insert lines of the form:

```c
if (params.output_new_property == ALL) OutputNewProperty(out_fp);
```

into the file following the example of OutputPositions, which appears in Run-SinglePoint(), MinimizeCallback(), RunMinimize(), and RunDynamics().

4.4 Add more degrees of freedom

To add degrees of freedom to the electrons, start by putting extra parameters into the NUCLEUS and ELECTRON typedefs in eff_global.

Next, change the routines ElecNucElec(), ElecElecElec(), and Pauli-ElecElec() in eff_update. There will need to be a way to keep the wavefunction normalized as well.
If the intent is to represent the electrons as linear combinations of primitive functions, it is straightforward enough to expand the kinetic energy and electrostatic sums into terms involving the primitives. An `Overlap()` function could be defined to aid normalization.

For the Pauli function, one needs to settle on a balance between implicit and explicit orthogonalization. The current functional works via implicit orthogonalization – it makes no attempt to generate orthogonal wavefunctions, but instead estimates the kinetic energy increase that would be expected upon orthogonalization.

With more degrees of freedom, however, there is possibility of changing the functions explicitly so that the wavefunctions are orthogonal. We have had some success using the SNOPT constrained optimizer by Gill, Murray, and Saunders for this purpose. One must be careful not to double-count the orthogonalization energy in that case. The exchange energy term also then becomes more prominent, and may be useful to calculate.

Too much variational flexibility can also lead to linear dependence issues. Two s floating gaussians can generate a p like function in the limit that they are near each other – however, this can lead to other numerical problems. This problem can be avoided somewhat by setting appropriate constraints, for example, mandating that primitives have exponents that are in fixed ratios to each other.

The semiclassical equations of motion in `eff_dynamics` will need to be updated as well, for example with the time dependent variational approach proposed by E. J. Heller, *J. Chem. Phys.* 64(1), 63-73 (1976).

In `eff_minimize`, the functions `GetMinimizeForces()`, `GetInitialPosition()`, and `SetMinimizePositions()` provide a mapping from the minimization arrays to the eFF system, and should be updated. In `eff_dynamics`, the functions `GetDynamicsForces()`, `GetDynamicsPositions()`, and `SetDynamicsPositions()` serve a similar role and should be updated as well.

### 4.4.1 Our experience

We experimented with using sums of Gaussians to make eFF more accurate. We found that the potential energy surface of H\textsubscript{2} could be made to approach the Hartree-Fock potential surface within a few kcal/mol, using only two floating Gaussian primitives per electron.

However, for systems with more than two electrons, double-counting in the Pauli potential and linear dependencies became problematic. Although there has been success in the literature with using multiple floating Gaussians to represent lone pairs, e.g. A. H. Pakari and F. M. Khalesifard, *J. Mol. Struct. Theochem* 340:175-183 (1995), those approaches have relied on “hand-positioned” Gaussian functions with very limited degrees of freedom. Once these systems were allowed to relax, we observed unphysical geometries and bond energies, as well as linear-dependence issues.

In general, we found it more effective to keep the simple single Gaussian description of electrons, and to add physically motivated terms to the eFF
expression to account for other effects, such as the $p$ character of electrons at a certain range from the nucleus, or electron correlation between opposite spin electrons in close proximity to each other. We have had a good deal of success with this approach, and we believe we have only begun to explore the potential of where these developments could lead.

By keeping the electrons simple and accounting for more complex effects in an implicit way, the equations of motion remain simple, the storage requirements remain light, and eFF remains fast.

4.5 Make nuclei floating Gaussians

In the current eFF scheme, nuclei are represented as classical particles. However, protons are light enough that they have a significant quantum delocalization, which can affect proton transfer rates, as in the kinetic isotope effect, and also affect excited electron motions coupled to proton motions.

In the nuclear-electron orbital method proposed by S. P. Webb, T. Iordanov, and S. Hammes-Schiffer, *J. Chem. Phys.* 117(9), 4106-4118 (2002), hydrogen nuclei as well as electrons are represented by sums of Gaussian functions. The analogous approach in eFF would be to make nuclei as well as electrons floating Gaussian functions.

One way to do this would be to extend the definition of electrons to include non-classical nuclei. These particles would have a kinetic energy inversely proportional to their mass, and would not interact via the Pauli potential (they could, but it would only come about at very small nucleus-nucleus separations).

First, enter a static mass $m$ into the ELECTRON typedef in eff_global. Then modify the function UpdateKineticEnergy() in eff_update, changing the kinetic energy from

$$E_{ke} = \frac{3}{2} \cdot \frac{1}{r_e^2}$$

to

$$E_{ke} = \frac{3}{2m} \cdot \frac{1}{r_e^2}$$

An additional restraining radial potential may also be needed to limit the proton size.

Then, modify the function ElecElecPauli so that the interaction does not apply to non-classical nuclei. Finally, change functions such as ProcessInputFile(), ProcessNucleus(), and ProcessElectrons() in eff_input to read in non-classical nuclei from the input file; functions in eff_output would need to be changed too.

4.6 Create and destroy particles dynamically

To create and destroy particles dynamically, we first need to change the data structure used to hold nucleus and electron data from a fixed array to a linked list.
To do this, redeclare the variables nuc and elec in eff_global as type LIST_PTR:

LIST_PTR *nuc, *elec

Rewrite the functions Initialize(), AddNucleus(), and AddElectron() in eff_create to work with the linked lists, using the functions new_list_ptr() and add_list_ptr() from eff_listptr. Also, use clear_list_ptr() and delete_list_ptr() to clear and deallocate the lists at the end of the program.

Next, modify the functions UpdateEnergyForces(), UpdateKineticEnergy(), UpdatePauliPeriodic(), UpdateWidthRestraint(), and AssignParticleBins() in eff_update so that they iterate over the linked list instead of over the array elements. For example, a loop to iterate over the nuclei:

```c
for (ptr = nuc->head; ptr != 0; ptr = ptr->next)
{
    NUC *nuc_ptr = (NUC *) ptr->element;
    ...
}
```

The functions in eff_ewald will need to be updated in the same way too.

Change the accessor functions in eff_access so instead of taking a nuclear or electron index as input, they take a pointer from the linked list. All the functions that reference eff_access functions will then need to be changed to loop over linked-list iterators, e.g. functions in eff_dynamics and eff_minimize. The alternative is to preserve the indexes, but to map the indexes onto linked list elements.

Finally, add a function to eff_listptr to delete elements from a linked list. Now it should be possible to create and delete particles in the middle of a simulation.

There are other issues to consider as well, such as energy conservation, making sure new particles do not overlap old ones too much, etc. Also when outputting the particle positions, energies, etc. in the .eff output file, it would be useful for trajectory analysis to be able to keep track of which particles were the same in between frames. For this purpose it would be useful to associate particles with unique indices or tags that would be printed out along with the per-particle information.
Chapter 5

To access hidden features

5.1 Check analytic derivatives

The functions `PrintAnalyticForces()` and `PrintNumericalForces()` in `eff_output` will print out to the standard output all the nuclear and electron forces, evaluated analytically and numerically through finite differences. Commented-out calls to these functions appear in `eff_driver`. When developing new energy expressions, it is useful to check that the analytic derivatives are correct by comparing the output of these two functions.

5.2 Use spin annealing

The function `AnnealSpins()` in `eff_driver` randomly swaps opposite spins in a Metropolis Monte Carlo fashion while keeping all particle positions fixed, using a temperature ramp profile specified through the arguments. A commented-out call to this function appear in `RunSinglePoint()` in `eff_driver`. We have used this routine to anneal the spin configuration of lithium solid to a lower potential energy state.
Chapter 6

Function prototypes and data structures

6.1 eff_access

This file defines accessor functions that should be used to get and set coordinates, energies, and forces of electrons and nuclei.

For example, these functions are called in eff_driver to print the energy of the system during dynamics runs, eff_bounds to wrap nuclear/electron positions to fit within the system bounds, eff_efield to apply electric field forces to the system, eff_properties to compute dipole moments and electron densities, eff_restrain to add restraint forces, and eff_dynamics and eff_minimize to perform dynamics and minimization.

There is some overhead associated with updating particle properties through function calls. In the minimization and dynamics routine, the overhead is reduced by copying particle coordinates and forces to a separate set of arrays, and performing multiple operations on the intermediate arrays before writing the positions and forces back to the main system. This preserves modularity at the cost of using additional memory. Such a policy makes it simple to use different minimization or dynamics routines, or to add extra degrees of freedom to the particles.

An exception is made in eff_update, where the underlying electron and nuclei data structures are accessed directly, bypassing the accessor functions.

The energies and forces are updated only after a call to UpdateEnergyForces() in eff_update. The nuclei indices range from 0 to NumNuclei() – 1, and the electron indices range from 0 to NumElectrons() – 1.

6.1.1 double GetTotalPE()

Returns the total potential energy of the system.
6.1.2  double GetNucleusPE(int i)
Returns the potential energy associated with nucleus i. Pairwise energies between two particles are split evenly between the two particles.

6.1.3  double GetElectronPE(int i)
Returns the potential energy associated with electron i. Pairwise energies between two particles are split evenly between the two particles.

6.1.4  void GetNuclearForce(int i, double *fx, double *fy, double *fz)
Gets the force (fx, fy, fz) acting on nucleus i.

6.1.5  void GetElectronForce(int i, double *fx, double *fy, double *fz, double *fr)
Gets the force (fx, fy, fz, fr) acting on electron i.

6.1.6  void GetNuclearPosition(int i, double *x, double *y, double *z)
Gets the position (x, y, z) of nucleus i.

6.1.7  void SetNuclearPosition(int i, double x, double y, double z)
Sets the position of nucleus i to (x, y, z).

6.1.8  void GetElectronPosition(int i, double *x, double *y, double *z, double *r)
Gets the position (x, y, z) and size r of electron i.

6.1.9  void SetElectronPosition(int i, double x, double y, double z, double r)
Sets the position of electron i to x, y, z and its size to r.

6.1.10 double GetNuclearCharge(int i)
Returns the charge of nucleus i.

6.1.11 int GetElectronSpin(int i)
Returns the spin of electron i.
6.1.12 void SetElectronSpin(int i, int spin)
Sets the spin of electron i to spin.

6.1.13 int NumNuclei()
Returns the number of nuclei.

6.1.14 int NumElectrons()
Returns the number of electrons.

6.1.15 void AddNuclearForce(int i, double fx, double fy, double fz)
Adds a force (fx, fy, fz) to nucleus i.

6.1.16 void AddElectronForce(int i, double fx, double fy, double fz, double fr)
Adds a force (fx, fy, fz, fr) to nucleus i.

6.1.17 void AddNucleusPE(int i, double energy)
Adds energy to the potential energy of nucleus i.

6.1.18 void AddElectronPE(int i, double energy)
Adds energy to the potential energy of electron i.

6.1.19 void SetNucFX(int i, double val)
Sets the x-component of the force on nucleus i to val. The functions AddNuclearForce and AddElectronForce are preferred; this function is used only in eff_freeze to null certain degrees of freedom.

6.1.20 void SetNucFY(int i, double val)
Sets the y-component of the force on nucleus i to val.

6.1.21 void SetNucFZ(int i, double val)
Sets the z-component of the force on nucleus i to val.

6.1.22 void SetElecFX(int i, double val)
Sets the x-component of the force on electron i to val.
6.1.23  void SetElecFY(int i, double val)
Sets the y-component of the force on electron \( i \) to \( \text{val} \).

6.1.24  void SetElecFZ(int i, double val)
Sets the z-component of the force on electron \( i \) to \( \text{val} \).

6.1.25  void SetElecFR(int i, double val)
Sets the r-component of the force on electron \( i \) to \( \text{val} \).

6.2  eff
bounds
This file contains functions that help to enforce rectangular periodic boundary conditions. The main function is \texttt{ApplyPositionBounds}, which wraps particles that have strayed outside the boundaries back to the interior of the unit cell.

Minimum image convention functions should properly be in this file, but are instead included as inline functions in \texttt{eff_update} for performance reasons. In \texttt{eff_update}, the functions \texttt{BoundDX}, \texttt{BoundDY}, and \texttt{BoundDZ} modify interatomic position vectors with the minimum image convention, so that the distance between two particles in a unit cell is always taken to be the shortest one subject to periodic boundaries.

6.2.1  void InitializeBounds()
Precomputes the box lengths and volume, and determines whether the individual dimensions of the box are periodic or not, based on the parameters \texttt{params.x_bound[]}, \texttt{params.y_bound[]}, \texttt{params.z_bound[]}, and \texttt{params.periodic}.

6.2.2  int isWrapX()
Returns whether the X dimension is periodic or not. Used internally, and in \texttt{eff_update} to determine whether a minimum image correction needs to be applied.

6.2.3  int isWrapY()
Returns whether the Y dimension is periodic or not.

6.2.4  int isWrapZ()
Returns whether the Z dimension is periodic or not.
6.2.5 double GetVolume()
Returns the volume of the rectangular box.

6.2.6 void ApplyPositionBounds()
Enforces periodic boundaries by mapping particles that have strayed outside the
unit cell back inside the unit cell. Used in eff_minimize and eff_dynamics.

6.3 eff_constants
Physical constants and conversion factors.

6.4 eff_create
In this version of eFF, the number of particles stays fixed throughout the simula-
tion. The nuclei and electrons are created once at the start of simulation through
two steps. First the Initialize function is called to dynamically allocate space
for the particles. Then the functions AddNucleus and AddElectron are used
to add the particles to the simulation space.

In this implementation, all of the functions in eff_create are called in eff_input, when the input .cfg file is read. The program counts the num-
ber of nuclei and electrons before adding them one at a time, so that exactly
the amount of space needed is allocated.

It may be useful to add the ability to create and destroy particles during the
simulation. Currently simple arrays hold the nuclear and electron coordinates.
In a more advanced implementation, a structure such as a linked list could be
used instead.

6.4.1 void Initialize(int maxnuclei, int maxelectrons)
Allocate enough space for maxnuclei nuclei and maxelectrons electrons.

6.4.2 int AddNucleus(double x, double y, double z, double q)
Add a nucleus at position (x, y, z) and charge q. If the number of nuclei exceeds
the allocated space, the program will stop with an error.

6.4.3 int AddElectron(double x, double y, double z, int spin, double re)
Adds an electron at position (x, y, z) with spin spin and size re.
6.5 eff_driver

This file runs the main loop of the program. First, the main procedure allocates memory for the elements contained in the input file: for the nuclei and electrons via `eff_create`, for user-set individual masses and velocities via `eff_initial`, and for constraints and restraints via `eff_freeze` and `eff_restrain`. The program peeks into the input file to see how much space needs to be allocated for each section.

Then the program sets up the parameters via `eff_param` and loads in their default values.

Next, the program reads in values from the input file via `eff_input`, sets up the output file via `eff_output`, and prints out any information it can prior to actually running the simulation.

With the parameters read in, the program initializes the boundaries via `eff_bounds`, the electric field via `eff_efield`, and the Ewald solver via `eff_ewald` if `periodic = true` has been specified. When setting up Ewald, the program also may auto-set the cutoff parameters `ewald_r_cutoff` and `ewald_k_cutoff` to achieve the precision specified in `ewald_log_precision`.

Then the program branches off to either a single point calculation, using `RunSinglePoint()`: a minimization, using `RunMinimize()`: or a dynamics run using `RunDynamics()`.

Finally, the main loop prints out elapsed time, updates the restart file as needed, and reclaims previously allocated memory.

6.5.1 void RunSinglePoint()

Runs a single point calculation by calling `eff_update` to compute an energy, and reporting it back via `eff_output`.

6.5.2 void RunMinimize()

In the minimization routine, the main loop is delegated to `eff_minimize` which in turn is driven through `eff_shanno`. The driver’s role is limited to providing a callback function that outputs energies, positions, forces, etc. via `eff_output` every several iterations.

6.5.3 void MinimizeCallback(int num_iterations, int num_fevals, double f_val, double g_sq)

Callback function for minimizer, provided to `eff_minimize`, which updates it every `print_every` iterations with the progress of the minimization: number of iterations `num_iterations`, number of function evaluations `num_fevals`, the value of the function `f_val`, and the gradient squared `g_sq`. The function in turn prints an update out to the output `.eff` file, and goes through `eff_output` to prints out position, energies, forces, etc.
6.5.4 void RunDynamics()

In the dynamics routine, the main loop is actually present in eff_driver, and it takes care of initializing masses and velocities; outputting energies, positions, and forces via eff_output regularly; and using an adaptive step size if energy is not conserved. The primary calls are through eff_dynamics, which is in charge of allocating memory for dynamics, taking single time steps, and implementing a thermostat.

6.5.5 void AnnealSpins(double start_temperature, double end_temperature, int numiter)

This routine is commented out. When called, it randomly swaps opposite spins in a Metropolis Monte Carlo fashion while keeping all particle positions fixed. The temperature is varied from start_temperature to end_temperature over numiter iterations. We used this routine to anneal the spin configuration of lithium solid to a lower potential energy state.

6.6 eff_dynamics

This file implements a velocity Verlet algorithm to propagate nuclear and electron dynamics. The function Dynamics() performs a single iteration of dynamics using a timestep set by SetTimeStep(). The propagation of positions and velocities occurs in the functions UpdateDynamicsPositions() and UpdateDynamicsVelocities(). Two thermostats are provided as well, Nose-Hoover in the ApplyNoseHooverNuclei() routine, and Andersen as a modification of the UpdateDynamicsVelocities() function.

In this eFF implementation, particle velocities and masses are stored separately as arrays in eff_dynamics, and are not part of the global data structure that stores nuclear and electron coordinates. A separate set of accessor functions provides access to masses and velocities, as well as temperatures, kinetic energy, and the time step. There is also a function UpdateDynamicsVirial() that computes the kinetic energy virial contribution to pressure, as it is velocity dependent.

To improve modularity, all the positions and forces in the system are copied to one-dimensional arrays dyn_x and dyn_f, and the integrator acts on those arrays. This additional translation layer makes it easy to add new degrees of freedom, change the underlying data structure for representing nuclei and electrons, or to program different integrators. A similar approach is used in eff_minimize.

6.6.1 Data structures

The arrays dyn_x, dyn_v, dyn_f, and dyn_prev_f store positions, velocities, forces, and previous forces respectively as one-dimensional arrays of length
$3N_{\text{nuc}} + 4N_{\text{elec}}$, where $N_{\text{nuc}}$ is the number of nuclei and $N_{\text{elec}}$ is the number of electrons.

The degrees of freedom are stored as follows:

\begin{align*}
nuc(1) & -- fx, fy, fz \\
nuc(2) & -- fx, fy, fz \\
... \\
elec(1) & -- fx, fy, fz, fr \\
elec(2) & -- fx, fy, fz, fr \\
...
\end{align*}

so that the nuclei occupy indices from $0 \ldots N_{\text{nuc}} - 1$ and the electrons occupy indices from $N_{\text{nuc}} \ldots N_{\text{nuc}} + N_{\text{elec}} - 1$.

The functions \texttt{GetDynamicsPositions()} and \texttt{GetDynamicsForces()} translate positions and forces from the eFF system into the dynamics arrays, and the function \texttt{SetDynamicsPositions()} does the reverse operation.

The adaptive step size routine uses a spare set of arrays \texttt{save\_dyn\_x}, \texttt{save\_dyn\_v}, \texttt{save\_dyn\_f}, and \texttt{save\_dyn\_prev\_f} to “roll back” an iteration when energy is not properly conserved. It does this using the functions \texttt{SaveDynamics()} and \texttt{RevertDynamics()}.

6.6.2 \texttt{void SetDynamicsLogfile(FILE *fp)}

Sets the output logfile file handle to \texttt{fp}.

6.6.3 \texttt{void AllocateDynamics()}

Allocates space for the dynamics arrays discussed in the Data Structures section. Calls \texttt{NumNuclei()} and \texttt{NumElectrons()} via \texttt{eff\_access} to determine how big the arrays should be.

6.6.4 \texttt{void SaveDynamics()}

Save the current dynamics arrays to a set of backup arrays (starting with \texttt{save\_}), for possible restoration later. Used by the adaptive step size procedure in \texttt{eff\_driver}.

6.6.5 \texttt{void RevertDynamics()}

Restores the current state to the dynamics state saved in \texttt{SaveDynamics()}. 6.6.6 \texttt{void InitializeDynamics()}

Initializes the positions, velocities, masses, and forces for time zero. Must be called before \texttt{Dynamics()} can be valid.
6.6.7 void InitializeRandomVelocities(double temp)

Initializes the velocities of the nuclei to the given temperature temp using a Maxwell-Boltmann distribution. Random numbers are from eff_util. Random velocities can be applied to the electrons as well by uncommenting out a portion of the code. Calls NullCMMotion() to remove net translational drift.

6.6.8 void NullCMMotion()

Nulls out the net center-of-mass motion to prevent translational drift.

6.6.9 void Dynamics(enum ThermostatType s_thermostat, double temperature)

Performs a single time step of dynamics using the functions UpdateDynamicsPositions() and UpdateDynamicsVelocities(). Calls eff_update to get energies and forces, applies the Nose-Hoover or Andersen thermostat as needed, applies constraints and restraints and external electric fields, and updates the kinetic contribution to the pressure virial. Also moves positions and forces between the real system and the internal dynamics arrays.

6.6.10 void UpdateDynamicsPositions()

Propagates the particle positions with the velocity Verlet algorithm:

\[ x_{i+1} = v_i \Delta t + \frac{1}{2m} f_i \Delta t^2 \]

This routine is called by Dynamics() and should not be called separately.

6.6.11 void UpdateDynamicsVelocities(enum ThermostatType thermostat, double temperature)

Propagates the particle velocities with the velocity Verlet algorithm:

\[ v_{i+1} = m \Delta t (f_i + f_{i+1}) \]

Also applies the Andersen thermostat if it is selected by the variable thermostat, which randomly replaces the velocity of the particle with one chosen from a Maxwell-Boltzmann distribution of velocities at the given temperature. This routine is called by Dynamics() and should not be called separately.

6.6.12 void ApplyNoseHoover(double temperature)

Applies the Nose-Hoover thermostat at the given temperature:

\[ \Delta f_i = -\zeta m_i v_i \]

\[ \Delta \zeta = \Delta t \left( \sum_i \frac{1}{2} m_i v_i^2 - \frac{3}{2} Nk_B T \right) \]
Currently the function \texttt{ApplyNoseHooverNuclei()} is used instead.

\textbf{6.6.13 \texttt{void ApplyNoseHooverNuclei(double temperature)}}

Applies the Nose-Hoover thermostat to the nuclei only. Currently this is the version called by \texttt{Dynamics()}. This routine should not be called separately.

\textbf{6.6.14 \texttt{void GetDynamicsForces()}}

Copies nuclear and electron forces from the eFF system to the dynamics array \texttt{dyn.f}. Uses \texttt{eff.access}.

\textbf{6.6.15 \texttt{void GetDynamicsPositions()}}

Copies nuclear and electron positions from the eFF system to the dynamics array \texttt{dyn.x}.

\textbf{6.6.16 \texttt{void SetDynamicsPositions()}}

Copies nuclear and electron positions from the dynamics array \texttt{dyn.x} to the eFF system.

\textbf{6.6.17 \texttt{void UpdateDynamicsVirial()}}

Updates the pressure virial with the kinetic energy of the nuclei and electrons, placing separate contributions into rigid versus flexible pressures.

\textbf{6.6.18 \texttt{void SetTimeStep(double dt)}}

Sets the time step of the Verlet integrator to \texttt{dt}.

\textbf{6.6.19 \texttt{double GetTemperature()}}

Computes the temperature as $\sum_{all} \frac{1}{2} m v^2 / 2 k_B N_{\text{nuc}}$, where we sum over all degrees of freedom, but assume that the temperature is $\frac{3}{2} k_B T$ per nucleus.

\textbf{6.6.20 \texttt{void SetNuclearVelocity(int i, double vx, double vy, double vz)}}

Sets the velocity of nucleus \texttt{i} to (\texttt{vx}, \texttt{vy}, \texttt{vz}). Used in \texttt{eff.driver} to set individual initial velocities.

\textbf{6.6.21 \texttt{void SetElectronVelocity(int i, double vx, double vy, double vz, double vr)}}

Sets the velocity of electron \texttt{i} to (\texttt{vx}, \texttt{vy}, \texttt{vz}, \texttt{vr}).

26
6.6.22  void GetNuclearVelocity(int i, double *vx, double *vy, double *vz)
gets the velocity of nucleus i in (vx, vy, vz).

6.6.23  void GetElectronVelocity(int i, double *vx, double *vy, double *vz, double *vr)
gets the velocity of electron i in (vx, vy, vz, vr).

6.6.24  void SetNuclearMass(int i, double mx, double my, double mz)
sets the mass of nucleus i to (mx, my, mz). Used in eff_driver to set individual masses.

6.6.25  void SetElectronMass(int i, double mx, double my, double mz, double mr)
sets the mass of electron i to (mx, my, mz, mr).

6.6.26  void GetNuclearMass(int i, double *mx, double *my, double *mz)
gets the mass of nucleus i in (mx, my, mz). Not yet used.

6.6.27  void GetElectronMass(int i, double *mx, double *my, double *mz, double *mr)
gets the mass of electron i in (mx, my, mz, mr).

6.6.28  double GetTotalKE()
returns the total kinetic energy of the system $\sum_{all} \frac{1}{2} m v^2$.

6.6.29  double GetNucleusKE(int i)
returns the kinetic energy of nucleus i:

$$ T = \frac{1}{2} (m_x v_x^2 + m_y v_y^2 + m_z v_z^2) $$

6.6.30  double GetElectronKE(int i)
returns the kinetic energy of electron i:

$$ T = \frac{1}{2} (m_x v_x^2 + m_y v_y^2 + m_z v_z^2 + m_r v_r^2) $$
6.6.31 double GetElectronTranslationKE(int i)

Returns the translational kinetic energy of electron i:

\[
T = \frac{1}{2} (m_x v_x^2 + m_y v_y^2 + m_z v_z^2)
\]

6.7 eff_efield

This file applies an electric field to the system that is constant in space, but may vary with time, either as a continuous sine wave or as a wave packet.

6.7.1 void SetupExternalField(double Ex0, double Ey0, double Ez0, double Efreq, double Epacket_duration)

Initializes the electric field with an amplitude (Ex0, Ey0, Ez0) in volts-per-centimeter, a frequency Efreq in Hertz, and a wave packet duration Epacket_duration in femtoseconds. The quantities are converted into the proper internal units using the constants V_PER_CM and T0_IN_FS.

6.7.2 void GetExternalField(double t, double *Ex, double *Ey, double *Ez)

This function outputs the electric field vector (Ex, Ey, Ez) at time t.

If Efreq and Epacket_duration are both zero, the electric field is constant over time. If the frequency is nonzero, but the wave packet duration still zero, it is a sine wave; otherwise if both the frequency and the wave packet duration are nonzero it is a wave packet.

This function is called by ApplyExternalField and is only used inside eff_efield.

6.7.3 void ApplyExternalField(double t)

Applies a uniform electric field to the system, taking the field from GetExternalField.

6.8 eff_eigen

This file computes the eigenvalues and eigenvectors of a matrix using Jacobi rotations. The routine was translated from an algorithm by H. Rutishauser in Numerische Mathematik 9, 1-10 (1966).
6.8.1 void jacobi(double *matrix, double eigenvalue[], double *eigenvector, int n)

Computes the eigenvalues and eigenvectors of a dimension n matrix. Used in 
\texttt{eff_restrain} to find least-squares lines and planes.

6.9 \texttt{eff_erf}

This file computes the function $\text{erf}(x)/x$ and its first and second derivatives if 
desired. These functions are used in \texttt{eff_update} and \texttt{eff_ewald} to compute the 
electrostatic interactions between Gaussian distributions of charge.

The error function is defined as

$$\text{erf}(x) = \frac{2}{\pi} \int_0^x e^{-t^2} dt$$

We compute the function using a Chebyshev polynomial expansion and bi-
linear mapping from J. Schonfelder, Math. Comp. 32(144):1232-40 (1978), and 
a routine adapted from the public domain SLATEC/FNLIB routine csevl.f by 
W. Fullerton, which uses the Clenshaw method to perform the expansion.

The user may select how many terms of the expansion to use using the 
constants \texttt{ERF\_TERMS1} (for $x < 2$), \texttt{ERF\_TERMS2} (for $x \geq 2$), and 
\texttt{DERF\_TERMS} (for the first derivative, $x < 2$).

The functions are replicated in \texttt{eff_update} as inline functions for perfor-
mance reasons.

6.9.1 double erfoverx(double x)

Returns $\text{erf}(x)/x$ for any $x > 0$. Calls the Chebyshev expansion \texttt{poly01()} when 
$x < 2$ and the expansion \texttt{poly02()} when $x \geq 2$.

6.9.2 double erfoverx1(double x, double *df)

Returns $\text{erf}(x)/x$ and its first derivative for any $x > 0$. It requires less work 
to calculation the function and its derivative together than to compute them 
separately. Versus \texttt{erfoverx()}, the routine makes an added call to the built-in 
exponential function \texttt{exp()} when $x > 2$ and to the Chebyshev expansion \texttt{poly1()} 
when $x \geq 2$.

6.9.3 double erfoverx2(double x, double *df, double *df2)

Returns $\text{erf}(x)/x$ and its first and second derivatives for any $x > 0$. Versus 
\texttt{erfoverx1()}, the routine makes an additional call to the polynomial \texttt{poly2()} 
when $x < 2$. Currently not used, but could be pressed into service if analytic 
second derivatives were desired.
6.9.4 double poly01(double x)  
Auxiliary polynomial $P(x)$ (defined in the paper) for $x < 2$.

6.9.5 double poly02(double x)  
Auxiliary polynomial $P(x)$ for $x \geq 2$.

6.9.6 double poly1(double x)  
Auxiliary polynomial $P'(x)$ for $x < 2$.

6.9.7 double poly2(double x)  
Auxiliary polynomial $P''(x)$ for $x < 2$.

6.10 eff_ewald

This file computes the electrostatic interaction energy between a collection of periodic Gaussian charge densities:

$$
\rho_i(r) = q_i \sum_{R_L} e^{-\alpha_i (r - r_i - R_L)^2}
$$

where $R_L$ is a set of periodic lattice vectors.

The standard Ewald sum procedure operates on point charges. We have implemented a generalized procedure that works on charges of finite width. The key change we make is to define a threshold exponent $\alpha_{\text{max}}$. When $\alpha < \alpha_{\text{max}}$, i.e. the charges are diffuse, we sum the electrostatic energy entirely in reciprocal space:

$$
E_{k\text{-space}} = \frac{1}{2V} \int d\mathbf{k} \frac{4\pi}{k^2} |\rho(\mathbf{k})|^2
$$

The periodicity of $\rho$ in real space reduces this integral to a lattice sum which converges quickly in $k$ when $\alpha_{\text{max}}$ is sufficiently small.

For $\alpha \geq \alpha_{\text{max}}$, we divide the charges into two portions:

$$
\rho = \rho_1 + \rho_2
$$

$$
\rho_1 = Ne^{-\alpha r^2} - N_{\text{max}}e^{-\alpha_{\text{max}} r^2}
$$

$$
\rho_2 = N_{\text{max}}e^{-\alpha_{\text{max}} r^2}
$$

We sum $\rho_1$ over a real space lattice and sum $\rho_2$ over a reciprocal space lattice. Since $\rho_1$ is neutral for large $r$ and localized to a small region of real space, the real space sum converges quickly in $r$. Since $r \rho_2$ is diffuse for a small enough $\alpha_{\text{max}}$, the reciprocal space sum converges quickly in $k$.

The routines EwaldRCutoff() and EwaldKCutoff() choose real and reciprocal space cutoffs to achieve a desired precision goal, guided by the known
asymptotic behavior of the Gaussian distributions. We have chosen a default $\alpha_{\text{max}}$ that optimizes the overall CPU time, assuming that these automatic cut-offs are in use.

The final Ewald energy is

$$E = E_{k-\text{space}} + E_{r-\text{space}} + E_{\text{self}} + E_{\text{uniform}}$$

The self energy is the energy of a Gaussian charge with itself – this quantity must be subtracted out, since in reality, charges do not repel themselves. The uniform energy represents the interaction of each Gaussian charge with the uniform neutralizing background charge.

### 6.10.1 Data structures

The program converts the eFF nuclei and electrons into Gaussian charge densities. Nuclei are represented as very sharply peaked Gaussians whose width is set by $\text{ewald}_r\_\text{nuc}$. Hence all the arrays have a dimension equal to the total number of nuclei and electrons.

The arrays $\text{ewald}_x$, $\text{ewald}_y$, $\text{ewald}_z$ contain the charge density positions; $\text{ewald}_\alpha$ contains the exponents; and $\text{ewald}_q$ the charges.

The force arrays $\text{ewald}_fx$, $\text{ewald}_fy$, $\text{ewald}_fz$, $\text{ewald}_f\alpha$, store the negative derivatives of the energy relative to $x$, $y$, $z$, and $\alpha$. The energy array $\text{ewald}_\text{energy}$ stores the energy per particle.

The arrays $\text{ewald}_S\_\text{real}$ and $\text{ewald}_S\_\text{imag}$ represent the charge density in reciprocal space, which is a complex number. The arrays $\text{ewald}_S\_\text{cap}\_\text{real}$ and $\text{ewald}_S\_\text{cap}\_\text{imag}$ are analogous, but for $\alpha = \alpha_{\text{max}}$.

The arrays $\text{lattice}_kx$, $\text{lattice}_ky$, $\text{lattice}_kz$, and $\text{lattice}_k^2$ prestore reciprocal lattice points and the arrays $\text{lattice}_r_x$, $\text{lattice}_r_y$, and $\text{lattice}_r_z$ prestore real space lattice points. Those arrays are filled once at the start of the program.

### 6.10.2 void AllocateEwald()

Allocates space for the Ewald arrays discussed above.

### 6.10.3 void InitializeEwald(double Lx, double Ly, double Lz, double max_\alpha, double max_r, double max_k, double nuc_\alpha)

Sets the rectangular periodic box to have size $(Lx, Ly, Lz)$; the variable $\alpha_{\text{max}}$ to be $\text{max}_\alpha$, the real space and reciprocal space cutoffs to be $\text{max}_r$ and $\text{max}_k$, and the exponent of the nuclei to be $\text{nuc}_\alpha$.

### 6.10.4 void SetEwaldCharges(double nuc_\alpha)

Converts the eFF nuclei and electrons into Gaussian charge distributions suitable for use in eff_ewald.
6.10.5  double TotalCharge()
Returns the total charge.

6.10.6  double TotalVolume()
Returns the total volume of the unit cell.

6.10.7  void AddEwaldEnergyForce(double *fx, double *fy, double *fz, double *falpha, double *energy)
Converts the energies and forces computed by the Ewald procedure into energies and forces that act on eFF nuclei and electrons, and adds them directly to the global data structure. It would be more appropriate to use eff access instead. The Ewald arrays taken as input are ewald_fx, ewald_fy, ewald_fz, ewald_falpha, and ewald_energy.

Called several times in UpdateEwaldEnergy(). Note that the arrays such as ewald_energy are completely overwritten between calls. If this is not done, it is important to zero them, so that energies are not added twice to the eFF system.

6.10.8  void EwaldVirial(double *ewald_energy, double *ewald_falpha)
Computes the Ewald contribution to the pressure virial. It can be shown that for particles interacting through an $-1/r$ potential, the virial is just the negative of the potential energy. This holds true if the charge density is uniformly scaled. If we assume the electron size stays rigid, we must make an additional correction that depends on ewald_falpha.

6.10.9  void UpdateEwaldEnergy()
Main Ewald procedure. Calls SetEwaldCharges() to get Gaussian charges from the eFF system, computes the sum of self, uniform, real space, and reciprocal space energies, and then calls AddEwaldEnergyForces() to apply the calculated energies and forces back to the eFF system. Also calls Ewald-Virial() to update the pressure virial.

6.10.10 void InitializeKSpace()
Allocates and precomputes the points of the reciprocal space lattice, filling the arrays lattice_kx, lattice_ky, lattice_kz, and lattice_k2 (magnitude). Uses the k-space cutoff parameter ewald_max_k. The point (0, 0, 0) is excluded. This routine is run only once, inside InitializeEwald().
6.10.11 void InitializeRSpace()

Allocates and precomputes the points of the real space lattice, filling the arrays `lattice_rx`, `lattice_ry`, and `lattice_rz`. Uses the r-space cutoff parameter `ewald_max_r`. The point (0, 0, 0) is excluded. This routine is run only once, inside `InitializeEwald()`.

6.10.12 void KSpaceEnergy(double *fx, double *fy, double *fz, double *falpha, double *energy)

Computes the reciprocal space energy:

\[ E_{k\text{-space}} = \frac{2\pi}{V} \sum_k \rho(k) \rho_{\text{max}}(k) \]

where

\[ \rho(k) = \sum_i q_i e^{-k^2/4\alpha_i} e^{-ik \cdot r_i} \]

\[ \rho_{\text{max}}(k) = \begin{cases} \rho(k) |_{\alpha_i = \alpha_{\text{max}}} & \text{for } \alpha_i > \alpha_{\text{max}} \\ \rho(k) & \text{otherwise} \end{cases} \]

Places the energy in the variable `energy`. Also computes the negative gradients and places them in the variables `fx`, `fy`, `fz`, and `falpha`.

6.10.13 void RSpaceEnergy(double *fx, double *fy, double *fz, double *falpha, double *energy)

Computes the real space energy:

\[ E_{r\text{-space}} = \sum_{R_L} \sum_{i \neq j} E_{ij} (r_{ij} - R_L) \]

where

\[ E_{ij} = \frac{1}{r_{ij}} \text{erf} \sqrt{\frac{\alpha_i \alpha_j}{\alpha_i + \alpha_j}} r_{ij} \]

\[ -\frac{1}{r_{ij}} \text{erf} \sqrt{\frac{\alpha_i \alpha_{\text{max}}}{\alpha_i + \alpha_{\text{max}}}} r_{ij} \]

Only interactions between a charge and another charge with \( \alpha > \alpha_{\text{max}} \) are considered. Places the energy in the variable `energy`. Also computes the negative gradients and places them in the variables `fx`, `fy`, `fz`, and `falpha`.

33
6.10.14 void SelfEnergy(double *falpha, double *energy)
Computes the self-energy:

\[ E_{\text{self}} = -\frac{1}{2} \sum_i q_i^2 \sqrt{\frac{\alpha_i}{\pi}} \sqrt{\frac{\alpha_i \alpha_{\text{max}}}{\alpha_i + \alpha_{\text{max}}}} \]

if \( \alpha_i < \alpha_{\text{max}} \)

or

\[ \sqrt{\frac{\alpha_i + \alpha_{\text{max}}}{\alpha_i \alpha_{\text{max}}}} \]

otherwise

Places the energy in the variable energy. Also computes the negative gradient with respect to the exponent and places it in the variable falpha.

6.10.15 void UniformChargeEnergy(double *falpha, double *energy)
Computes the interaction of each charge with a uniform neutralizing background:

\[ E_{\text{uniform}} = \frac{1}{4} Q \sum_i \frac{\pi}{V} \left( \frac{1}{\alpha_{\text{max}}} - \frac{1}{\alpha_i} \right) \]

for \( \alpha_i > \alpha_{\text{max}} \)

Places the energy in the variable energy. Also computes the negative gradient with respect to the exponent and places it in the variable falpha.

6.10.16 double EwaldRCutoff(double precision, double alpha_cutoff, double min_alpha)
Computes the real-space lattice cutoff needed to achieve the negative log10 precision goal precision \( p \), assuming \( \alpha_{\text{max}} = \alpha_{\text{cutoff}} \) and that we want to describe charges that have an exponent as small as \( \alpha_{\text{min}} = \text{min}_\alpha \):

\[ r_{\text{cutoff}} = \sqrt{\frac{-\ln 10 \cdot p + 3}{\alpha_{\text{max}}/\left(1 + \alpha_{\text{max}}/\alpha_{\text{min}}\right)}} \]

6.10.17 double EwaldKCutoff(double precision, double alpha_cutoff)
Computes the reciprocal-space lattice cutoff needed to achieve the negative log10 precision goal precision \( p \), assuming \( \alpha_{\text{max}} = \alpha_{\text{cutoff}} \):

\[ k_{\text{cutoff}} = \sqrt{4\alpha_{\text{max}} \left(-\ln 10 \cdot p + 5\right)} \]

6.11 eff_freeze
This file keeps track of and applies coordinate constraints. When a coordinate is constrained, the forces corresponding to that coordinate are set to zero at every minimization or dynamics step via the function ApplyFreezes().

New constraints are added via functions such as AddFreezeNucX(); these functions are called in eff_input when in the input file the character # appears after a nuclear or electron coordinate.
6.11.1  Data structures
The constraints themselves are dynamically allocated and stored in a linked list
declared via eff_listint. Thus the number of constraints that can be added
is limited only by the system’s memory. Separate lists are used to store con-
straints on different coordinates: freeze_nuc_x, freeze_nuc_y, freeze_nuc_z,
freeze_elec_x, freeze_elec_y, freeze_elec_z, freeze_elec_r.

6.11.2  void InitializeFreezes()
Initializes the linked lists containing the constraints. Uses eff_listint.

6.11.3  void ClearFreezes()
Deallocates the constraint lists via eff_listint.

6.11.4  void AddFreezeNucX(int i)
Adds a constraint to the x-coordinate of nucleus i.

6.11.5  void AddFreezeNucY(int i)
Adds a constraint to the y-coordinate of nucleus i.

6.11.6  void AddFreezeNucZ(int i)
Adds a constraint to the z-coordinate of nucleus i.

6.11.7  void AddFreezeElecX(int i)
Adds a constraint to the x-coordinate of electron i.

6.11.8  void AddFreezeElecY(int i)
Adds a constraint to the y-coordinate of electron i.

6.11.9  void AddFreezeElecZ(int i)
Adds a constraint to the z-coordinate of electron i.

6.11.10 void AddFreezeElecR(int i)
Adds a constraint to the r-coordinate of electron i.

6.11.11 void ApplyFreezes()
Applies all of constraints by setting to zero the forces on any coordinates present
in the constraint lists. Called in eff_dynamics and eff_minimize.

35
6.12 eff_global

Contains global data structures shared by many source files: nucleus data, electron data, and parameters.

To keep the program modular, the nucleus and electron data should only be accessed through the routines in eff_access and eff_create. However two source files break this guideline and access the global data structures nuc[] and elec[] directly: eff_update and eff_ewald. If changes are made to the nucleus and electron data structures, routines in those source files should be updated as well.

Every user-defined parameter is stored as a global variable accessible to all routines in the program. There have been some efforts to restrict the usage of these parameters to eff_driver but they have been only somewhat successful.

6.12.1 Nucleus data

typedef struct 
{
  double x, y, z, q;
  double fx, fy, fz, energy;
} NUCLEUS;

The variables x, y, z, and q are the nuclei position and charge. The variables fx, fy, fz, and energy are the nuclear forces and single particle energy.

An extern variable of type (NUCLEUS *) is defined in eff_global, which is allocated as an array with NumNuclei() entries in eff_create. Hence any source file which includes eff_global has access to variables of the form nuc[i].*

6.12.2 Electron data

typedef struct 
{
  double x, y, z, r;
  int spin;
  double fx, fy, fz, fr, energy;
} ELECTRON;

The variables x, y, z, and r are the electron positions and size. The variable spin is the electron spin, and can be either plus or minus one. The variables fx, fy, fz, fr, and energy are the electron forces and single particle energy.

An extern variable of type (ELECTRON *) is defined in eff_global, which is allocated as an array with NumElectrons() entries in eff_create. Hence any source file which includes eff_global has access to variables of the form elec[i].*
6.12.3 Parameters

Tables 6.1 and 6.2 lists the variables are defined inside the typedef PARAMS. In some cases, enumerated types are used and their entries are hidden here; however, their definitions in the header file are self-explanatory.

An extern variable of type PARAMS is defined in eff_global. Hence any source file which includes eff_global has access to variables of the form params.*.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Cutoffs</strong></td>
<td>General pair cutoff</td>
<td>10000.0</td>
</tr>
<tr>
<td>double taper_cutoff</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Boundaries</strong></td>
<td>Rectangular box xmin, xmax</td>
<td>-5000, 5000</td>
</tr>
<tr>
<td>double x_bound[2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>double y_bound[2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>double z_bound[2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>periodic</td>
<td>Ewald bounds, min image, none?</td>
<td>none</td>
</tr>
<tr>
<td><strong>Ewald</strong></td>
<td>-log 10 desired Ewald precision</td>
<td>-6.0</td>
</tr>
<tr>
<td>double ewald_log_precision</td>
<td>Real/reciprocal space switchover</td>
<td>3.54</td>
</tr>
<tr>
<td>double ewald_re_cutoff</td>
<td>Automatically set r and k-space cutoffs?</td>
<td>yes</td>
</tr>
<tr>
<td>int ewald_autoset</td>
<td>Manually set r-space cutoff</td>
<td>7.0</td>
</tr>
<tr>
<td>double ewald_r_cutoff</td>
<td>Manually set k-space cutoff</td>
<td>8.0</td>
</tr>
<tr>
<td>double ewald_k_cutoff</td>
<td>Radius of nucleus in Ewald</td>
<td>1e-10</td>
</tr>
<tr>
<td>double ewald_muc_r</td>
<td>Maximum electron size Ewald can describe</td>
<td>4.5</td>
</tr>
<tr>
<td>double ewald_max_re</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Minimization</strong></td>
<td>Conjugate gradient or Newton-Raphson?</td>
<td>CG</td>
</tr>
<tr>
<td>min</td>
<td>Freeze nuclei and/or electrons?</td>
<td>no</td>
</tr>
<tr>
<td>min_freeze</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Dynamics</strong></td>
<td>None, Nose-Hoover, or Anderson</td>
<td>none</td>
</tr>
<tr>
<td>thermostat</td>
<td>Andersen collision rate</td>
<td>0.1</td>
</tr>
<tr>
<td>double andersen_coupling</td>
<td>Nose-Hoover effective mass</td>
<td>1.0</td>
</tr>
<tr>
<td>double nose_hoover_coupling</td>
<td>Initial T for rand. velocities and thermostat</td>
<td>0</td>
</tr>
<tr>
<td>double start_temperature</td>
<td>End temperature ramp for thermostat</td>
<td>0</td>
</tr>
<tr>
<td>double end_temperature</td>
<td>Time step for integrator</td>
<td>0.005</td>
</tr>
<tr>
<td>double dt</td>
<td>On or off?</td>
<td>off</td>
</tr>
<tr>
<td>adaptive_step_size</td>
<td>Energy conservation goal</td>
<td>0.0001</td>
</tr>
<tr>
<td>double adaptive_energy_number</td>
<td>Number of subdivisions to attempt</td>
<td>5</td>
</tr>
<tr>
<td>int adaptive_num_tries</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.1: Parameter global data structure, part I

6.13 eff_initial

This file keeps track of individual initial velocities defined in the muc_velocities and elec_velocities section of the .cfg input file. It does the same for the individual masses defined in the muc_masses and elec_masses section as well. This
### Updating

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int print_every</td>
<td>Output information every print_every steps</td>
<td>50</td>
</tr>
<tr>
<td>int num_steps</td>
<td>Total number of dynamics or minimization steps to take</td>
<td>10000</td>
</tr>
</tbody>
</table>

### Action

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>calc</td>
<td>Single point, dynamics, or minimization</td>
<td>single point</td>
</tr>
</tbody>
</table>

### Output

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>output_position</td>
<td>None, every step, or only at end</td>
<td>all</td>
</tr>
<tr>
<td>output_velocity</td>
<td>None, every step, or only at end</td>
<td>all</td>
</tr>
<tr>
<td>output_energy_forces</td>
<td>None, every step, or only at end</td>
<td>none</td>
</tr>
<tr>
<td>output_restart</td>
<td>None, every step, or only at end</td>
<td>all</td>
</tr>
<tr>
<td>output_restraints</td>
<td>None, every step, or only at end</td>
<td>all</td>
</tr>
</tbody>
</table>

### Masses

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>double electron_mass</td>
<td>Dynamics mass of electron to use</td>
<td>1.0</td>
</tr>
</tbody>
</table>

### Random numbers

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned int rand_seed</td>
<td>Pseudorandom number seed</td>
<td>10000</td>
</tr>
</tbody>
</table>

### External field

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>double Ex, Ey, Ez</td>
<td>Amplitude of electric field</td>
<td>0, 0, 0</td>
</tr>
<tr>
<td>double Efreq</td>
<td>Frequency of electric field (0 for constant)</td>
<td>0</td>
</tr>
<tr>
<td>double Epacket_duration</td>
<td>Length of wave packet (0 for continuous)</td>
<td>0</td>
</tr>
</tbody>
</table>

### Electron size limit

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>int size_limit</td>
<td>Size limit is on or off?</td>
<td>on</td>
</tr>
<tr>
<td>double size_limit_stiffness</td>
<td>Constant in harmonic restraining potential</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 6.2: Parameter global data structure, part II.
file performs bookkeeping only, and does not actually set the initial velocities or masses.

Accessor functions are provided to add and retrieve the entries. Entries are added in `eff_input` by functions such as `ProcessNucVelocities()` when the input .cfg file is read in.

Entries are retrieved in `eff_driver` by the `RunDynamics()` function, which then calls `eff_dynamics` to initialize the velocities and masses. Note that this is done after the default velocities and masses have been set.

### 6.13.1 Data structures

The initial velocities and masses are stored in fixed arrays, unlike the constraints in `eff_freezes` or the restraints in `eff_restraints`, which are stored in linked lists. Hence the number of initial velocities and masses must be known ahead of time, so that the arrays can be allocated.

The arrays used are `nuc_v_array`, which contains components of `(vx, vy, vz)`; `elec_v_array`, which contains components of `(vx, vy, vz, vr)`; `nuc_m_array`, which contains components of `(mx, my, mz)`; and `elec_m_array`, which contains components of `(mx, my, mz, mr)`.

### 6.13.2 void AllocateInitialVelocities(int max_nuc_v, int max_elec_v)

Allocates the initial velocity arrays with `max_nuc_v` nuclear velocities and `max_elec_v` electron velocities.

### 6.13.3 void AllocateInitialMasses(int max_nuc_m, int max_elec_m)

Allocates the individual mass arrays with `max_nuc_m` nuclear masses and `max_elec_m` electron masses.

### 6.13.4 int GetNumNucV()

Returns the number of nuclear velocities stored.

### 6.13.5 int GetNumElecV()

Returns the number of electron velocities stored.

### 6.13.6 int GetNumNucM()

Returns the number of nuclear masses stored.

### 6.13.7 int GetNumElecM()

Returns the number of electron masses stored.
6.13.8 void AddNucV(int i, double vx, double vy, double vz)

Adds a nuclear velocity \((vx, vy, vz)\) corresponding to nucleus \(i\).

6.13.9 void AddElecV(int i, double vx, double vy, double vz, double vr)

Adds an electron velocity \((vx, vy, vz, vr)\) corresponding to electron \(i\).

6.13.10 void GetNucV(int idx, int *i, double *vx, double *vy, double *vz)

Gets stored entry \(idx\), filling up the variables \(i\), the nucleus index, and \((vx, vy, vz)\), the nucleus velocity.

6.13.11 void GetElecV(int idx, int *i, double *vx, double *vy, double *vz, double *vr)

Gets stored entry \(idx\), filling up the variables \(i\), the electron index, and \((vx, vy, vz, vr)\), the electron velocity.

6.13.12 void AddNucM(int i, double mx, double my, double mz)

Adds a nuclear mass \((mx, my, mz)\) corresponding to nucleus \(i\).

6.13.13 void AddElecM(int i, double mx, double my, double mz, double mr)

Adds an electron mass \((mx, my, mz, mr)\) corresponding to electron \(i\).

6.13.14 void GetNucM(int idx, int *i, double *mx, double *my, double *mz)

Gets stored entry \(idx\), filling up the variables \(i\), the nucleus index, and \((mx, my, mz)\), the nucleus mass.

6.13.15 void GetElecM(int idx, int *i, double *mx, double *my, double *mz, double *mr)

Gets stored entry \(idx\), filling up the variables \(i\), the electron index, and \((mx, my, mz, mr)\), the electron mass.
6.14 eff_input

Reads in the .cfg input file, which contains calculation parameters, nuclear positions and charges, electron positions and sizes and spins, nuclear and electron initial velocities, nuclear and electron individual masses, and restraints.

Once the data is read, updates the eFF system using functions in eff_params, eff_create, eff_freezes eff_initial, and eff_restraints.

Also contains auxiliary functions to match strings, strip newlines, and parse strings.

6.14.1 int CountLines(FILE *fp, char *header)

Returns the number of lines under a given header, such as nuclei. Used to allocate array sizes before the data has been read, required in eff_create and eff_initial.

6.14.2 void ProcessInputFile(FILE *fp)

Reads in the .cfg file line-by-line. Determines which section each line belongs to, and sends it to the appropriate processing function, e.g. ProcessParam() or ProcessNucleus(). Also removes the newline character.

6.14.3 void ProcessNucleus(char *str)

From a string, reads in the nucleus position and charge, and adds it to the eFF system using AddNucleus() in eff_create. Also looks for the character # appended to any coordinate and calls functions in eff_freeze if necessary.

6.14.4 void ProcessElectron(char *str)

From a string, reads in the electron position, spin, and radius, and adds it to the eFF system using AddElectron() in eff_create. Also looks for the character # appended to any coordinate and calls functions in eff_freeze if necessary.

6.14.5 void ProcessNucVelocities(char *str)

From a string, reads in an initial nuclear velocity, and adds it to the eFF system via eff_initial.

6.14.6 void ProcessElecVelocities(char *str)

From a string, reads in an initial electron velocity, and adds it to the eFF system via eff_initial.
6.14.7  void ProcessNucMasses(char *str)
From a string, reads in an initial nuclear mass, and adds it to the eFF system via eff_initial.

6.14.8  void ProcessElecMasses(char *str)
From a string, reads in an initial electron mass, and adds it to the eFF system via eff_initial.

6.14.9  void ProcessParam(char *str)
From a string, reads in a parameter and its arguments, and sends it to ExecuteParameter() in eff_param. The source file eff_param contains a system to dynamically add commands and function pointers, and to match parameters to their associated functions.

6.14.10 void ProcessRestraints(char *str)
From a string, reads in a restraint, and adds it to the eFF system via eff_restraints.

6.14.11 void NucElecIndex(char *str, int *index)
Helper function for ProcessRestraints(). Recognizes the labels used to refer to nuclei and electrons, and converts them to a numerical form the program can use. In restraints, nuclei are referenced by their number (e.g. 5), and electrons by a number with the prefix e (e.g. e5).

6.14.12 int string_match(char *str1, char *str2)
Returns 1 if str1 and str2 match, disregarding case.

6.14.13 void StripNewline(char *str)
Removes the terminating newline character from str, if present.

6.14.14 void ParseString(char *buffer, int *argc, char **argv)
Tokenizes the input string buffer, recognizing both the tab character and space as separators, and fills argc, the number of substrings, and argv *, an array of pointers to the substrings. Both argc and argv must be defined and properly allocated before the function call.
6.15  eff_listint

This file contains functions to handle linked lists of integers, which the program uses in situations where a list is needed, but the number of entries is not known in advance.

Accessor functions are provided to create, add elements to, and destroy the lists. Reading elements from the linked list is accomplished by accessing the underlying data structure directly, and not through accessor functions.

The integer lists are used in eff_update to keep track of which particles are within different spatial bins, and in eff_freeze to keep track of coordinate constraints.

6.15.1  Data structure

The file defines two structure types, LIST_INT for the list itself, and LIST_INT_ELEMENT for the elements of the linked list.

LIST_INT contains pointers to the head and tail elements of the list:

```c
typedef struct {
    LIST_INT_ELEMENT *head;
    LIST_INT_ELEMENT *tail;
} LIST_INT;
```

LIST_INT_ELEMENT contains the list element itself, as well as a pointer to the next element of the list:

```c
typedef struct LIST_INT_ELEMENT {
    struct LIST_INT_ELEMENT *next;
    int element;
} LIST_INT_ELEMENT;
```

To read elements of a list that has already been created, the following loop can be used in any file that includes eff_listint:

```c
LIST_INT integer_list;
... (initialize and fill with elements) ...

LIST_INT_ELEMENT *ptr;
for (ptr = integer_list->head; ptr != 0; ptr = ptr->next)
{
    printf("Element %i\n", integer_list->element);
}
```

6.15.2  LIST_INT *new_list_int()

Creates a new integer list, e.g.

```c
LIST_INT integer_list = new_list_int();
```
6.15.3  void delete_list_int(LIST_INT *list)
Deallocates a list. Before calling this function, call clear_list_int to deallocate
the individual elements of the list.

6.15.4  void add_list_int(LIST_INT *list, int newelement)
Adds an element to the tail of the list, e.g.
    add_list_int(integer_list, 4);

6.15.5  void clear_list_int(LIST_INT *list)
Deallocates all the individual list items; the list is empty afterwards.

6.16  eff_listptr
This file performs the functions as eff_listint, but with elements of type (void *)
* instead of integers. Since the data structures and prototype functions are
identical in all but element type, they will not be repeated here. Void pointers
require a type cast function, e.g. (double *) ptr, before they are considered valid
pointers and can be accessed.

The pointer lists are used in eff_restrain to keep track of restraints that
appear in the restraints section of the .cfg file.

6.17  eff_minimize
This file interfaces with the Shanno minimization code in eff_shanno, initializ-
ing position and gradient arrays, updating gradient arrays with values from the
eFF system, and updating positions with values from the minimizer.

6.17.1  Data structures
As in eff_dynamics, all the nuclear and electronic degrees of freedom are
mapped to a single one-dimensional position array min_x:

    nuc(1) -- x, y, z
    nuc(2) -- x, y, z
...
    elec(1) -- x, y, z, log r
    elec(2) -- x, y, z, log r
...

Thus the array has dimension $3N_{nuc}+4N_{elec}$, where $N_{nuc}$ is the number of nuclei
and $N_{elec}$ is the number of electrons.

A key difference from the variable mapping versus eff_dynamics is that
log r is stored instead of r. This ensures that $r > 0$ even though the variables
range unconstrained from $-\infty$ to $\infty$ during the minimization. It also improves the efficiency of the minimizer when small electrons are involved, for example core electrons around high $Z$ nuclei.

The array $\text{min}_g$ stores the gradients of the total potential energy with respect to the variables in $\text{min}_f$.

Finally, an array $\text{min}\_\text{temp}$ is defined which serves as temporary storage for the Shanno minimizer. Its size depends on the minimization scheme selected. If $\text{min}_x$ and $\text{min}_g$ have dimension $N$, the storage array has size $5N + 2$ for the conjugate gradient optimizer, and size $N(N+7)/2$ for the Newton-Raphson optimizer.

6.17.2  void AllocateMinimize(enum MinimizeType minmethod)

Allocates space for the $\text{min}_x$, $\text{min}_g$, and $\text{min}_\text{temp}$ arrays. The space set aside for $\text{min}_\text{temp}$ depends on the minimization algorithm $\text{minmethod}$ selected.

6.17.3  int MinStorageRequirements(enum MinimizeType minmethod)

Computes the storage requirements for the $\text{min}_\text{temp}$ array, using the formula given in the Data Structures section, which is a function of the minimization algorithm $\text{minmethod}$ selected.

6.17.4  void CalcContractFG(double *x, double *f, double *g)

A callback function that gives the scalar function value $f$ and the function gradient vector $g$ at a position vector $x$. In eFF, the $f$ is the potential energy of the system, $g$ are the forces on the nuclei and electrons, and $x$ are the positions of the nuclei and electrons.

The function operates using the helper functions $\text{GetMinimizeForces()}$ to fetch eFF system forces, $\text{SetMinimizePositions()}$ to set eFF system positions, and $\text{UpdateEnergyForces()}$ from $\text{eff\_update}$ to compute the energy of the system and the forces on the particles.

The function also applies restraints, constraints, and external electric fields. Note that the restraint energy and forces are included explicitly in the minimization; however those energy components are not included in the sum $\text{GetTotalPE()}$ in $\text{eff\_update}$. 
6.17.5 enum MinimizeResult Minimize(enum MinimizeType s_minmethod, double eps, double acc, int max_numsteps, int print_every, void (*MinCallback)(int, int, double, double))

Interface function to the Shanno algorithm in eff_shanno. Passes along the method type, either conjugate gradient or Newton, an epsilon value \( \text{eps} \), the desired accuracy \( \text{acc} \), the maximum number of steps to take \( \text{max_numsteps} \), how often to print out energy and position information (\text{print_every}）， and the callback function \text{MinCallback} which is called every \text{print_every} steps with information on the iteration, function value, and gradient.

6.17.6 void GetMinimizeForces(double *forces)

Maps eFF system forces into the array \( \text{forces} \). Used by CalcContractFG() to determine the energy \( f \) and gradients \( g \) at a given position \( x \).

6.17.7 void GetInitialPosition(double *positions)

Maps the current eFF position into the array \( \text{positions} \). Used by Minimize() to start the Shanno minimizer off at the current system position.

6.17.8 void SetMinimizePositions(double *positions)

Maps positions from the array \( \text{positions} \) into the eFF system. Used by CalcContractFG() to determine the energy \( f \) and gradients \( g \) at a given position \( x \). The position \( x \) is assigned by the Shanno algorithm.

6.18 eff_output

This file provides functions for writing to output .eff files and restart .cfg.restart files.

Writing to .eff files is done via calls to individual functions such as OutputEnergyForces() and OutputVelocities(). This arrangement simplifies switching off and on different outputs, for example, by the parameters output_energy_forces and output_velocity.

For modularity, all of the writing to the .eff file should be done through eff_output. This rule is violated in a few places: (1) in eff_driver, minimization and dynamics iteration updates are written directly, as are adaptive step size updates, (2) in eff_restrain, information on the different restraints is written out directly, and (3) in eff_update, out of bounds warnings are written out directly.

Writing to the .cfg.restart file is done via a single call to OutputRestartFiles(). If the input file format is changed, this function should be changed too.
6.18.1 void OutputEnergyForces(FILE *out_fp)
Outputs individual particle energies and forces. See section ??.

6.18.2 void OutputPositions(FILE *out_fp)
Outputs individual particle positions. See section ??.

6.18.3 void OutputVelocities(FILE *out_fp)
Outputs individual particle velocities. See section ??.

6.18.4 void OutputEnergy(FILE *out_fp)
Outputs the dipole moment, total energy, and pressure at the end of the calculation. See section ??.

6.18.5 void OutputTimeElapsed(FILE *out_fp)
Outputs the elapsed time at the end of the calculation. See section ??.

6.18.6 void OutputRestartFile(char *restart_filename)
Outputs the restart file, given the filename restart_filename. The parameters are printed out with a call to OutputParams(), and all the other quantities are retrieved via accessor functions: nuclear and electron coordinates from eff_access, nuclear and electron velocities from eff_dynamics, individually specified masses from eff_initial, and restraints with a call to OutputRestartRestraints() in eff_restraints.

6.18.7 void OutputFixedHeader(FILE *out_fp)
Outputs the header at the start of the .eff file, which contains fixed quantities relating to the nuclei and electrons. See section ??.

6.18.8 void OutputDynamicsMasses(FILE *out_fp)
Outputs individually specified masses into the .eff file.

6.18.9 void OutputParams(char *tag, FILE *out_fp)
Outputs the calculation parameters. This function does double duty, it is used by OutputRestartFile() to write to .cfg.restart files, and by eff_driver to write to .eff files. The tag prepends a prefix to each line, and is set to [params] when writing to .eff files.

The parameters are read from the global data structure params.*. Some helper functions are used to translate enumerated data types into strings for
output. Their purpose is clear, and so their names are listed below without further elaboration: PeriodicTypeToString, MinimizeTypeToString, MinimizeFreezeTypeToString, ThermostatTypeToString, ActionTypeToString, OutputTypeToString, AdaptiveStepSizeTypeToString, and BoolToString.

6.18.10 void PrintAnalyticForces()

This function prints out to the standard output all the nuclear and electron forces. It is used for debugging purposes only, and is present in commented-out form in eff_driver.

6.18.11 void PrintNumericalForces()

This function prints out to the standard output all the nuclear and electron forces, evaluated using finite-difference numerical derivatives. It is used for debugging purposes only, and is present in commented-out form in eff_driver.

6.19 eff_params

This file parses calculation parameters that appear in the params section of the .cfg input file, and sets the corresponding variable in the global data structure params.*. It also sets default values for parameters via the function SetDefaultParameters().

The program provides a general scheme for adding parameters. The function AddParameters() associates each parameter name to a function which parses the parameter string. Multiple parameters can map onto a single parsing function, which can then contain switch statements to handle the multiple parameters.

Often parameters that are in the same logical category are parsed together. For example, the parameters x_bound, y_bound, and z_bound are parsed by ParamXYZBound, and the parameters ewald_log_precision and ewald_re_cutoff are parsed by ParamEwald. This division makes to code easier to understand than having a single global parsing function which handles all the parameters.

All of the parameters are added under the function InitializeParameters(), called in eff_driver. Once this is done, the function ExecuteParameter is called in eff_input as the parameters appear in the .cfg input file.

6.19.1 Data structures

The function AddParameters() associates each parameter name with a function pointer via a structure of type PARAM_FUNCTIONS:

typedef struct
{
    char name[255];
    void (*fp)(int, char **);
};
All of the name/function combinations are stored in a fixed-length array `param_functions[]`. Currently the array size is set at 100; this value should be increased as needed.

The parser functions take as arguments the integer `argc`, which are the number of words present on the parameter string, and the array of pointers `argv`, which point to the individual words present on the parameter string. For example, the string

```
x_bound = 5
```

will have `argc` equal to 3, and

```
argv[0] = "x_bound"
argv[1] = "="
argv[2] = "5"
```

### 6.19.2 void InitializeParameters()

Initializes the name/function parameter array `params_functions[]`, and associates all of the parameter names with their parsing functions by calling `AddParameter()`.

### 6.19.3 void AddParameter(char *name, void (*fp)(int, char **))

Associates the parameter `name` with the parsing function `fp`. Called repeatedly in `InitializeParameters()`.

### 6.19.4 void ExecuteParameter(char *name, int argc, char **argv)

Executes the parsing function matched to `name`, with the argument count `argc` and the argument list `argv`. Called by `eff_input`.

### 6.19.5 void SetDefaultParameters()

Sets the parameters to the default values listed in Table 6.1.

### 6.19.6 Parsing functions

Table 6.3 lists the parsing functions and the parameter names they handle. The purpose of the parsing functions is clear, and so no further elaboration is given.
<table>
<thead>
<tr>
<th>Parsing function</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParamCutoff</td>
<td>taper_cutoff</td>
</tr>
<tr>
<td>ParamXYZBound</td>
<td>x_bound, y_bound, z_bound</td>
</tr>
<tr>
<td>ParamPeriodic</td>
<td>periodic</td>
</tr>
<tr>
<td>ParamEwald</td>
<td>ewald_log_precision, ewald_re_cutoff</td>
</tr>
<tr>
<td></td>
<td>ewald_autoset, ewald_r_cutoff, ewald_k_cutoff</td>
</tr>
<tr>
<td></td>
<td>ewald_nuc_r, ewald_max_re</td>
</tr>
<tr>
<td>ParamMin</td>
<td>min</td>
</tr>
<tr>
<td>ParamMinFreeze</td>
<td>min_freeze</td>
</tr>
<tr>
<td>ParamThermostat</td>
<td>thermostat</td>
</tr>
<tr>
<td>ParamNoseHooverCoupling</td>
<td>nose_hoover_coupling</td>
</tr>
<tr>
<td>ParamAndersenCoupling</td>
<td>andersen_coupling</td>
</tr>
<tr>
<td>ParamTemperature</td>
<td>start_temperature, end_temperature</td>
</tr>
<tr>
<td>ParamAdaptiveStepSize</td>
<td>adaptive_step_size</td>
</tr>
<tr>
<td>ParamAdaptiveEnergy</td>
<td>adaptive_energy</td>
</tr>
<tr>
<td>ParamAdapativeNumTries</td>
<td>adaptive_num_tries</td>
</tr>
<tr>
<td>ParamDt</td>
<td>dt</td>
</tr>
<tr>
<td>ParamPrintEvery</td>
<td>print_every</td>
</tr>
<tr>
<td>ParamNumSteps</td>
<td>num_steps</td>
</tr>
<tr>
<td>ParamCalc</td>
<td>calc</td>
</tr>
<tr>
<td>ParamOutput</td>
<td>output_position, output_velocity,</td>
</tr>
<tr>
<td></td>
<td>output_energy_forces, output_restart,</td>
</tr>
<tr>
<td></td>
<td>output_restraints</td>
</tr>
<tr>
<td>ParamElectronMass</td>
<td>electron_mass</td>
</tr>
<tr>
<td>ParamRandSeed</td>
<td>rand_seed</td>
</tr>
<tr>
<td>ParamEField</td>
<td>e_field</td>
</tr>
<tr>
<td>ParamEFieldFreq</td>
<td>e_field_freq</td>
</tr>
<tr>
<td>ParamEFieldPacketDuration</td>
<td>e_field_packet_duration</td>
</tr>
<tr>
<td>ParamSizeLimit</td>
<td>size_limit</td>
</tr>
<tr>
<td>ParamSizeLimitStiffness</td>
<td>size_limit_stiffness</td>
</tr>
</tbody>
</table>

Table 6.3: Parameter parsing functions and parameter names.
6.19.7 void ReadValue(char *str, double *val)

A helper function for the parser functions. Translates the string str to a double val, with the provision that the strings -infinity and infinity are translated into the global constant INFINITY, defined in eff_global as 1000000.

6.20 eff_pressure

This file computes the instantaneous pressure of a periodic system using the virial expression

\[
P = \frac{dE}{dV} = \frac{Nk_B T}{V} + \frac{1}{3V} \sum_i r_i \cdot f_i
\]

\[
= \frac{1}{3V} \left( 2 \sum_i \frac{1}{2} m_i v_i^2 + \sum_i r_i \cdot f_i \right)
\]

\[
= \frac{1}{3V} (ke_{\text{virial}} + pe_{\text{virial}})
\]

where N is the number of particles, T is the instantaneous temperature, V is the volume, \( r_i \) and \( f_i \) are the particle positions and forces, and \( m_i \) and \( v_i \) are the particle masses and velocities.

The virial sum is built up in pieces throughout the calculation. Each time a force on a particle is calculated in eff_update, a function call of the form AddForceVirial() is made to update the potential energy virial. Each time a velocity is computed, an analogous call is made in eff_dynamics to AddKineticEnergyVirial() to update the kinetic energy virial. The potential energy virial for electrostatic sums has a particularly simple form, and is handled through AddPotentialEnergyVirial().

Two separate pressures are computed, one assuming that electron sizes change upon compression, and one that assumes that they do not. The program keeps track of these virial sums separately, and when the virial add functions are called, the argument electron_type specifies which sum to update. For example, when electron_type is rigid, no radial force contribution is included, as opposed to when electron_type is flexible.

6.20.1 Data structures

The program tracks the virial sums in the four variables rigid_ke_virial, rigid_pe_virial, flexible_ke_virial, and flexible_pe_virial.

The variable electron_type can be set to the values rigid, flexible, or both. If the virial add functions are called with the parameter rigid, only the rigid virial is updated; with flexible, only the flexible virial is updated; and with both, both of the virials are updated.
6.20.2  void ClearVirial()
Resets the virial sums to zero.

6.20.3  void AddForceVirial(double rx, double ry, double rz, double rr, double fx, double fy, double fz, double fr, enum ElectronType electron_type)
Adds to the potential energy virial the term \( r_x f_x + r_y f_y + r_z f_z + r_r f_r \). The parameter \texttt{electron_type} determines which virial sums are updated.

6.20.4  void AddSizeForceVirial(double r, double f, enum ElectronType electron_type)
Adds to the potential energy virial the term \( r \cdot f \).

6.20.5  void AddRadialForceVirial(double r, double f, enum ElectronType electron_type)
Adds to the potential energy virial the term \( r \cdot f \). The same as the \texttt{AddSizeForceVirial}().

6.20.6  void AddPotentialEnergyVirial(double energy, enum ElectronType electron_type)
Adds to the potential energy virial the value \texttt{energy}. Used in computing the potential energy virial for Coulomb systems, where a simplification occurs: \( \sum_i r_i \cdot f_i = U \), where \( U \) is the potential energy.

6.20.7  void AddKineticEnergyVirial(double energy, enum ElectronType electron_type)
Adds to the kinetic energy virial the value \( 2 \cdot \texttt{energy} \).

6.20.8  double GetRigidPEPressure(double volume)
Returns the rigid potential energy component of pressure, which equals \( \frac{1}{3V} \cdot \texttt{rigid_pe_virial} \). Units are in GPa.

6.20.9  double GetFlexiblePEPressure(double volume)
Returns the flexible potential energy component of pressure, which equals \( \frac{1}{3V} \cdot \texttt{flexible_pe_virial} \). Units are in GPa.
6.20.10  double GetRigidKEPressure(double volume)
Returns the rigid kinetic energy component of pressure, which equals \( \frac{1}{3V} \cdot \text{rigid\_ke\_virial} \). Units are in GPa.

6.20.11  double GetFlexibleKEPressure(double volume)
Returns the flexible kinetic energy component of pressure, which equals \( \frac{1}{3V} \cdot \text{flexible\_ke\_virial} \). Units are in GPa.

6.21  eff\_properties
Computes properties of the system at the end of the calculation.

6.21.1  void CalcDipole(double *dipole\_x, double *dipole\_y, double *dipole\_z)
Computes the dipole moment of the system \((\text{dipole\_x}, \text{dipole\_y}, \text{dipole\_z})\) in atomic units.

6.21.2  double ElectronDensity(double x, double y, double z)
Returns the electron density at position \((x, y, z)\). Currently not used in the code.

6.22  eff\_restrain
This file track of and applies geometry restraints, for example keeping two atoms a fixed distance apart from each other. The restraints are enforced via the function \text{ApplyRestraints()}\), which acts by applying harmonic restraining potentials to sets of atoms. The function is called at every minimization or dynamics step.

These potentials have a minimum when the geometric condition is satisfied, and have steep walls to push the system toward the desired configuration. However, there may be cases where large energy barriers are present and opposing forces resist the applied restraint, and in the end the restraint may not be fully satisfied.

The function calls and data structures are similar to those in \text{eff\_freeze}. New restraints are added via functions such as \text{AddDistanceRestraint()}\); these functions are called in \text{eff\_input} when lines in the @restraints section are read.

The \textit{tether restraint} is a special kind of restraint, in that it averages and evenly distributes forces on a given a set of particles. In the case of dynamics,
the forces are scaled by particle masses so that the accelerations are equal for all particle in the set, and the initial velocities are averaged over the set as well.

Developers wishing to add new restraints should note that for the tether restraint to work properly, it must be applied last in the function ApplyRestraints().

Restraints can act on both nuclei and electrons together. In the input file, for example, a distance restraint setting the distance between nucleus 2 and electron 2 to 1.0 bohr is written as

@restraints
distance 2 e2 1.0

Internally, the program translates the indices “2” and “e2” into particle indices 2 and 2+Nnuclei via the function NucElecIndex() in eff_input. Helper functions in eff_restrain then take these particle indices as arguments. Examples include AddNucElecForce(), GetNucElecPosition(), and GetNucElecForce().

6.22.1 Data structures

The restraints are dynamically allocated and stored in a linked list declared via eff_listptr. Thus the number of restraints that can be added is limited only by the system’s memory. Separate lists are used to store different types of restraints: restraint_coordinate, restraint_distance, restraint_angle, restraint_dihedral, restraint_line, restraint_plane, and restraint_tether.

Data structures for the individual restraints containing particle indices and the restraint value. For example, the data structure for the dihedral restraint is as follows:

typedef struct
{
    int idx1, idx2, idx3, idx4;
    double angle;
} RES_DIHEDRAL;

In some cases, the restraint contains a variable number of particles. In these cases – restraint_line, restraint_plane, and restraint_tether – the data structure is defined as follows:

typedef struct
{
    int *indices;
    int numindices;
} RES_TETHER;

containing a pointer to an integer array indices, where the size of the array is numindices.

The linked list stores void pointers, which must be cast to the appropriate type using an explicit cast. For example, to apply the planar restraints, the following code is used:
LIST_PTR_ELEMENT *ptr;   // ptr->element has type (void *)
for (ptr = restraint_plane->head; ptr != 0; ptr = ptr->next) {
    RES_PLANE *element = (RES_PLANE *) ptr->element;
    restraints_energy += ApplyPlaneRestraint(element->indices, element->numindices, 0);
}

The harmonic restraints have spring constants $K_{COORD}$, $K_{DISTANCE}$, $K_{ANGLE}$, $K_{DIHEDRAL}$, $K_{LINE}$, and $K_{PLANE}$.

6.22.2 void InitializeRestraints()
Initializes empty linked lists for all the restraint types. Called once in eff_driver when the program starts.

6.22.3 void ClearRestraints()
Clears all the restraint lists. Called once in eff_driver when the program ends.

6.22.4 void AddCoordinateRestraint(int idx, double x, double y, double z)
Adds a restraint forcing particle $idx$ to coordinate $(x, y, z)$.

6.22.5 void AddDistanceRestraint(int idx1, int idx2, double dist)
Adds a restraint forcing the distance between particles $idx1$ and $idx2$ to $dist$.

6.22.6 void AddAngleRestraint(int idx1, int idx2, int idx3, double angle)
Adds a restraint forcing the angle between particles $idx1$, $idx2$, and $idx3$ to $angle$.

6.22.7 void AddDihedralRestraint(int idx1, int idx2, int idx3, int idx4, double angle)
Adds a restraint forcing the dihedral between particles $idx1$, $idx2$, $idx3$, and $idx4$ to $angle$.

6.22.8 void AddLineRestraint(int *indices, int numindices)
Adds a restraint forcing the numindices particles listed in indices onto a line. There must be at least three particles for this restraint to be valid.
6.22.9  void AddPlaneRestraint(int *indices, int numindices)

Adds a restraint forcing the numindices particles listed in indices onto a plane. There must be at least four particles for this restraint to be valid.

6.22.10 void AddTetherRestraint(int *indices, int numindices)

Adds a restraint that averages and evenly distributes the forces on the numindices particles listed in indices.

6.22.11 double ApplyRestraints()

Applies all of the added restraints to the eFF system. Does this by applying functions such as ApplyCoordinateRestraint() and ApplyDistanceRestraint() to each of the elements of the restraint lists. This function is called once per minimization or dynamics iteration.

6.22.12 void ApplyTetherVelocityRestraints()

Executes the function ApplyTetherVelocityRestraint() over all the stored tethers. This function is called once in the entire dynamics run, right after the initial velocities have been set.

6.22.13 void AddsNucElecForce(int idx, double fx, double fy, double fz)

Adds to particle idx the force (fx, fy, fz). The particle indices are discussed in the Data Structures section above.

6.22.14 void GetNucElecPosition(int idx, double *x, double *y, double *z)

Gets the position of particle idx and places it in (x, y, z).

6.22.15 void GetNucElecForce(int idx, double *fx, double *fy, double *fz)

Gets the force on particle idx and places it in (fx, fy, fz).

6.22.16 void GetNucElecMass(int idx, double *mx, double *my, double *mz)

Gets the mass of particle idx and places it in (mx, my, mz).
6.22.17  void GetNucElecVelocity(int idx, double *vx, double *vy, double *vz)

Gets the velocity of particle idx and places it in (vx, vy, vz).

6.22.18  void SetNucElecVelocity(int idx, double vx, double vy, double vz)

Sets the velocity of particle idx to (vx, vy, vz).

6.22.19  void NucElecString(char *str, int idx)

Translates the particle index idx to a string. For example, if the number of nuclei is 5, the index 2 would be translated to “2”, while the index 5 would be translated to “e1”.

6.22.20  double ApplyCoordinateRestraint(int idx, double x0, double y0, double z0, double *dist)

Applies a coordinate restraint to particle idx, forcing it to have position (x0, y0, z0). Returns the restraint energy. If dist is not NULL, places into dist the distance between the particle and (x0, y0, z0).

6.22.21  double ApplyDistanceRestraint(int idx1, int idx2, double dist0, double *dist)

Applies a distance restraint to particles idx1 and idx2, forcing them to be separated by distance dist0. Returns the restraint energy. If dist is not NULL, places into dist the distance between the two particles.

6.22.22  double ApplyAngleRestraint(int idx1, int idx2, int idx3, double angle0, double *angle)

Applies an angle restraint to particles idx1, idx2, and idx3 forcing them to subtend the angle angle0. Returns the restraint energy. If angle is not NULL, places into angle the angle between the three particles.

6.22.23  double ApplyDihedralRestraint(int idx1, int idx2, int idx3, int idx4, double d0, double *angle)

Applies an dihedral restraint to particles idx1, idx2, idx3, and idx4 forcing them to subtend the dihedral d0. Returns the restraint energy. If angle is not NULL, places into angle the dihedral between the four particles.
6.22.24 double ApplyLineRestraint(int *indices, int numindices, double *dists)

Applies a line restraint to the numindices particles listed in indices, forcing them to lie on a line. If dists is not NULL, places into the array the distance between the particles and the least-squares line, measured orthogonally.

6.22.25 double ApplyPlaneRestraint(int *indices, int numindices, double *dists)

Applies a plane restraint to the numindices particles listed in indices, forcing them to lie on a plane. If dists is not NULL, places into the array the distance between the particles and the least-squares plane, measured orthogonally.

6.22.26 double ApplyTetherRestraint(int *indices, int numindices, double *dists)

Applies a tether restraint to the numindices particles listed in indices, averaging and evenly distributing the forces on the particles. If a dynamics calculation is being performed, scales the average velocities by the mass of each particle, so that the accelerations are the same; also see ApplyTetherRestraintVelocity() used to set the initial velocities. If dists is not NULL, places zeros into the array (not used).

6.22.27 double ApplyTetherRestraintVelocity(int *indices, int numindices)

Applies a tether restraint to the numindices particles listed in indices, averaging and evenly distributes the velocities on the particles. This only needs to be done once, at the beginning of the dynamics simulation.

6.22.28 void OutputRestraints(FILE *out_fp)

Outputs value of restraints to the .eff output file, handle out_fp.

6.22.29 void OutputRestartRestraints(FILE *restart_fp)

Outputs value of restraints to the .cfg.restart restart file, handle restart_fp.

6.23 eff.shanno

We have converted the original Fortran subroutine into a C function, and modified the array indices so that they start at 0 instead of 1.

The subroutine can operate in two modes, a conjugate gradient mode requiring $7n$ words of temporary storage, or a quasi-Newton BFGS (Broyden-Fletcher-Goldfarb-Shanno) mode requiring $n^2/2 + 11n/2$ words of temporary storage.

The `conmin` subroutine is called by the `Minimize()` function in `eff_minimize`.

```c
6.23.1 int conmin(int n, double *x, double *f, double *g,
                int *ifun, int *iter, double eps, int mxfun, double
                *w, int iout, int mdim, double acc, int nmeth, void
                (*calcfg)(double *, double *, double *), void (*update_func)(int, int, double, double))
```

From the comments provided in the original `conmin` algorithm, the arguments are listed in Table 6.4 and the return value error codes are listed in Table 6.5.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>Number of variables in the function to be minimized</td>
</tr>
<tr>
<td>x</td>
<td>Vector containing the initial guess. Upon exiting, holds the final vector.</td>
</tr>
<tr>
<td>f</td>
<td>Upon exiting, holds the final (minimum) function value.</td>
</tr>
<tr>
<td>g</td>
<td>Upon exiting, holds the final gradient.</td>
</tr>
<tr>
<td>ifun</td>
<td>Upon exiting, holds the number of function/gradient evaluations.</td>
</tr>
<tr>
<td>iter</td>
<td>Upon exiting, holds the number of iterations.</td>
</tr>
<tr>
<td>eps</td>
<td>User-supplied convergence parameter.</td>
</tr>
<tr>
<td>mxfun</td>
<td>Maximum number of evaluations allowed.</td>
</tr>
<tr>
<td>w</td>
<td>Temporary storage vector.</td>
</tr>
<tr>
<td>iout</td>
<td>If 1, enables output.</td>
</tr>
<tr>
<td>mdim</td>
<td>Dimension of w.</td>
</tr>
<tr>
<td>acc</td>
<td>User-supplied machine accuracy.</td>
</tr>
<tr>
<td>nmeth</td>
<td>1 for BFGS, 0 for CG.</td>
</tr>
<tr>
<td>calcfg</td>
<td>Callback <code>func(x, f, g)</code> calculates function at x, and returns its value in f and the gradient in g.</td>
</tr>
<tr>
<td>update_func</td>
<td>Callback output(num_iterations, num_fevals, f_val, g_sq).</td>
</tr>
</tbody>
</table>

Table 6.4: Arguments for Shanno’s `conmin` function in `eff_shanno`.

<table>
<thead>
<tr>
<th>Return value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Algorithm has converged.</td>
</tr>
<tr>
<td>1</td>
<td>Maximum number of function evaluations used.</td>
</tr>
<tr>
<td>2</td>
<td>Linear search fails, is gradient wrong?</td>
</tr>
<tr>
<td>3</td>
<td>Search vector not a descent direction, roundoff error?</td>
</tr>
</tbody>
</table>

Table 6.5: Error codes for Shanno’s `conmin` function in `eff_shanno`. 59
6.23.2 double dmin(double a, double b)
Returns the minimum of $a$ and $b$. Used in conmin.

6.23.3 double dmax(double a, double b)
Returns the maximum of $a$ and $b$. Used in conmin.

6.24 eff_timing
This file implements a stopwatch used to determine how long a calculation takes to complete.

Two timers are available, CPUElapsed(), which has a resolution of less than a second but which overflows after 36 minutes, and TotalTimeElapsed(), which has a resolution of one second but should not overflow. Both values are printed out by eff_output at the end of the calculation.

6.24.1 void StartTimer()
Starts the stopwatch timer.

6.24.2 void StopTimer()
Stops the stopwatch timer.

6.24.3 double CPUElapsed()
CPU time elapsed in between the time StartTimer() and StopTimer() is called.

6.24.4 double TotalTimeElapsed()
User time elapsed in between the time StartTimer() and StopTimer() is called.

6.25 eff_update
This file computes the energy and forces associated with an eFF system. In eFF, the overall energy is a sum of a Hartree product kinetic energy, a Hartree product electrostatic energy, and an antisymmetrization (Pauli) correction:

$$E = E_{\text{ke}} + E_{\text{nuc-nuc}} + E_{\text{nuc-elec}} + E_{\text{elec-elec}} + E_{\text{Pauli}}$$
which can be broken down further as follows:

\[
E_{\text{ke}} = \sum_i \frac{3}{2} \frac{1}{s_i^2}
\]

\[
E_{\text{nuc-nuc}} = \sum_{i<j} \frac{Z_i Z_j}{R_{ij}}
\]

\[
E_{\text{nuc(i)-elec(j)}} = -\sum_{i,j} Z_i \frac{Z_j}{R_{ij}} \text{Erf}\left(\frac{\sqrt{2} R_{ij}}{s_j}\right)
\]

\[
E_{\text{elec(i)-elec(j)}} = \sum_{i<j} \frac{1}{r_{ij}} \text{Erf}\left(\frac{\sqrt{2} r_{ij}}{s_i + s_j}\right)
\]

\[
E_{\text{Pauli}} = \sum_{\sigma_i = \sigma_j} E(\uparrow\uparrow)_{ij} + \sum_{\sigma_i \neq \sigma_j} E(\uparrow\downarrow)_{ij}
\]

where \(E(\uparrow\downarrow)\) and \(E(\uparrow\downarrow)\) are the Pauli potential functions:

\[
E(\uparrow\uparrow)_{ij} = \left(\frac{s_{ij}^2}{1 - s_{ij}^2} + (1 - \rho) \frac{s_{ij}^2}{1 + s_{ij}^2}\right) \Delta T_{ij}
\]

\[
E(\uparrow\downarrow)_{ij} = \frac{\rho s_{ij}^2}{1 + s_{ij}^2} \Delta T_{ij}
\]

where \(\Delta T\) is a measure of the kinetic energy change upon antisymmetrization, and \(S\) is the overlap between two wave packets:

\[
\Delta T_{ij} = \frac{3}{2} \left(\frac{1}{s_i^2} + \frac{1}{s_j^2}\right) - \frac{2(3(s_i^2 + s_j^2) - 2s_{ij}^2)}{(s_i^2 + s_j^2)^2}
\]

\[
S_{ij} = \left(\frac{s_i/s_j + s_j/s_i}{s_i + s_j}\right)^{3/2} \exp\left(-\frac{r_{ij}^2}{s_i^2 + s_j^2}\right)
\]

where \(\rho = -0.2\), \(\bar{x}_{ij} = x_{ij} \cdot 1.125\), and \(\bar{s}_i = s_i \cdot 0.9\).

The main function \texttt{UpdateEnergyForces()} updates the energy and forces, calling other functions in \texttt{eff_update} to compute individual energy components, and then summing them all together. The final results can be retrieved using functions from \texttt{eff_access} such as \texttt{GetTotalPE()}, \texttt{GetNuclearForce()}, and \texttt{GetElectronForce()}.

For systems whose dimensions are larger than the pairwise cutoff \texttt{taper_cutoff}, eFF saves on computational cost by partitioning space into rectangular bins smaller than the cutoff length, and only considering interactions between nearest-neighbor bins. The routine \texttt{AssignParticleBins()} assigns the particles to the bins, and a routine inside \texttt{UpdateEnergyForces()} calls \texttt{UpdateElecAndPauli()} only over nearest-neighbor bins.

When such systems have periodic boundaries, the unit cell is so large that each particle can only “see” particles within the same unit cell. We treat the
interactions using the minimum image convention, and wrap around space so that bins on one edge of the system can interact with bins on the opposite edge of the system. Functions such as LeftStencil() and RightStencil() help manage this process.

To apply the minimum image convention, we use functions such BoundDX(), which convert interatomic distances within a periodic cell to a minimum image distance. In smaller periodic systems, each particle may see multiple images of itself and particles from other cells, and a more sophisticated algorithm is needed to sum long range forces. We sum electrostatic forces using the Ewald algorithm via UpdateEwaldEnergy() in eff_ewald. The Pauli potential is shorter range, and we assume the minimum image convention holds – we perform this sum using UpdatePauliPeriodic(). The kinetic energy in all cases is updated with the same function UpdateKineticEnergy().

Many functions in this source file are declared inline to improve performance, such as ierfoverx1(), which evaluates erf(x)/x. All gradients are computed analytically.

6.25.1 Data structures

We partition space into rectangular bins, and each bin \((n_x, n_y, n_z)\) has an index

\[ i = n_x + n_y \cdot \text{numbins}_x + n_z \cdot \text{numbins}_x \cdot \text{numbins}_y \]

To keep track of which particles are in which bin, we associate with each bin index \(i\) a list of nuclei indices \text{nuc\_bin}[i] and a list of electron indices \text{elec\_bin}[i]. The variables \text{nuc\_bin} and \text{elec\_bin} are arrays of pointers to integer lists, and have type (LIST_INT **).

6.25.2 void IntitializeEnergyUpdater()

Initializes the rectangular bins. The number of bins along a given direction is calculated as follows:

\[ \text{numbins}_x = \text{(int)} \left( \frac{\text{length}_x}{\text{params.taper_cutoff}} \right); \]
\[ \text{if} \ (\text{numbins}_x == 0) \ \text{numbins}_x = 1; \]

Hence the program tries to break space down into segments the size of the cutoff, but if fractional units are left over, the program stretches out the bin slightly so that all the bins fit evenly into space. Each bin is guaranteed to be at least the cutoff length in size along every dimension.

The program also determines whether the minimum image convention needs to be applied along the x, y, and z dimensions, based on the user-set parameters, and sets the flags \text{minimage}_x, \text{minimage}_y, and \text{minimage}_z accordingly.

6.25.3 int LeftStencil(int numbins, int pbc)

The left range of bins that can interact with a bin, given the number of bins numbins along a direction and whether or not periodic boundary conditions
are present along that direction \((\text{pbc})\). Usually equal to -1, so that a nearest neighbor bin can interact with a central bin, but some cases need to be handled specially.

6.25.4  **int RightStencil(int numbins, int pbc)**

Similar to **LeftStencil**, but represents the right range of bins, and is usually equal to +1.

6.25.5  **void AssignParticleBins()**

Assigns particles to the rectangular bins. Clears the lists, then for each particle determine which bin contains it, then adds the particles to the bins. If a particle doesn’t belong to any bin, prints out a warning. This function is called every time **UpdateEnergyForces()** is called.

6.25.6  **void UpdateEnergyForces()**

Updates the total potential energy, and the forces on individual particles. Begins by computing the kinetic energy, then branches into two cases.

If Ewald summation is being used, computes electrostatics via **UpdateEwaldEnergy()** in **eff_ewald**, and Pauli repulsion via **UpdatePauliPeriodic()**.

Otherwise, breaks space down into bins using **AssignParticleBins()**, and call **UpdateElecAndPauli()** over all nearest-neighbor bins that are expected to interact with each other, accounting for periodic boundary conditions if present.

6.25.7  **void UpdateKineticEnergy()**

Computes the total kinetic energy and associated radial forces.

6.25.8  **void UpdateElecAndPauli(int bin1, int bin2, int wrap_x, int wrap_y, int wrap_z)**

Computes the total electrostatic interaction and Pauli repulsion between particles in **bin1** and **bin2**. The flags **wrap_x**, **wrap_y**, and **wrap_z** control whether or not the minimum image convention is applied in the **x**, **y**, and **z** dimensions.

6.25.9  **void UpdatePauliPeriodic()**

Computes the Pauli repulsion for a periodic system using the minimum image convention.

6.25.10  **void UpdateWidthRestraint()**

Applies a restraint potential \(V = \frac{1}{2}k_s s^2\) to electrons that have a radius greater than half the size of the smallest rectangular box dimension.
6.26 eff_util

This file is a collection of miscellaneous utility functions to report errors, generate random numbers, convert strings from uppercase to lowercase, get information on elements, and handle file extensions.

6.26.1 void error(const char *format, ...)

Reports an error and exits the program. Called with the same arguments as printf:

```c
error("Error: 1 plus 1 is \%i\n", 3);
```

6.26.2 int rand_int(int max)

Returns an integer from 0 to max – 1.

6.26.3 double rand_uni()

Returns a double from 0 to 1 (will never be exactly 1), distributed uniformly.

6.26.4 double rand_gauss()

Returns a double from a Gaussian distribution, mean 0, standard deviation 1.

6.26.5 void lcase(char *str)

Converts the string str to a lower case string.

6.26.6 double GetMass(int Z)

Returns the atomic mass in amu of the element with nuclear charge Z.

6.26.7 char *GetType(int Z)

Returns the element symbol (H, He, Li, etc.) of the element with nuclear charge Z.

6.26.8 int GetZ(char *str)

Returns the nuclear charge Z of the element with element symbol str.

6.26.9 void GetBasename(char *str, char *basename)

Returns the basename of a filename basename.extension.
6.26.10  void GetExtension(char *str, char *extension)
Returns the extension of a filename basename.extension.

6.26.11  void rand_seed(unsigned int seed)
Seeds the pseudorandom generator with seed.

6.26.12  double min(double x, double y)
Returns the minimum of x and y.
Index

accessor functions, see eff_access
add_list_int(), 44
add_list_ptr(), 44
AddAngleRestraint(), 55
AddCoordinateRestraint(), 55
AddDihedralRestraint(), 55
AddDistanceRestraint(), 55
AddElecM(), 40
AddElectron(), 21
AddElectronForce(), 19
AddElectronPE(), 19
AddElecV(), 40
AddEwaldEnergyForce(), 32
AddForceVirial(), 52
AddFreezeElecR(), 35
AddFreezeElecX(), 35
AddFreezeElecY(), 35
AddFreezeElecZ(), 35
AddFreezeNucX(), 35
AddFreezeNucY(), 35
AddFreezeNucZ(), 35
AddKineticEnergyVirial(), 52
AddLineRestraint(), 55
AddNuclearForce(), 21
AddNucleus(), 21
AddNucleusPE(), 19
AddNucM(), 40
AddNucV(), 40
AddParameter(), 49
AddParameters(), 49
AddPlaneRestraint(), 56
AddPotentialEnergyVirial(), 52
AddRadialForceVirial(), 52
AddSizeForceVirial(), 52
AddSnuElecForce(), 56
AddTetherRestraint(), 56
AllocateDynamics(), 7, 24
AllocateEwald(), 31
AllocateInitialMasses(), 39
AllocateInitialVelocities(), 39
AllocateMinimize(), 45
AnnealSpins(), 23
ApplyAngleRestraint(), 57
ApplyCoordinateRestraint(), 56
ApplyDihedralRestraint(), 57
ApplyDistanceRestraint(), 57
ApplyExternalField(), 28
ApplyFreezes(), 35
ApplyLineRestraint(), 57
ApplyNoseHoover(), 25
ApplyNoseHooverNuclei(), 7, 26
ApplyPlaneRestraint(), 57
ApplyPositionBounds(), 21
ApplyRestraints(), 11, 56
ApplyTetherRestraint(), 57
ApplyTetherRestraints(), 11
AssignParticleBins(), 62

BFGS, see eff_shanno
boundary
minimum image, 61
boundary conditions, see eff_bounds

CalcContractFG(), 45
CalcDipole(), 53
callback function
minimizer, 45
Chebyshev polynomial, 29
clear_list_int, 44
ClearFreezes(), 35
ClearRestraints(), 10, 55
ClearVirial(), 52
conjugate gradient, see eff_shanno
commun(), 58
constraints, see eff_freeze
conversion factors, see eff_constants
CountLines(), 41
CPU time, see CUTimeElapsed() 60
CPUTimeElapsed(), 60
creating particles, see eff_create
data structures
dynamics, 23
Ewald, 31
freeze, 35
initial velocities and masses, 39
integer linked list, 43
minimize, 44
nuclei and electrons, 36
parameters, 48
partition space, 62
pressure virial, 51
restraints, 54
void pointer linked list, see eff_listptr
degrees of freedom
add new, 12
issues with too many, 13
delete_list_int(), 44
delete_list_ptr(), 44
dipole moment, see eff_properties
dmax(), 58
dmin(), 58
dyn_prev_f, 23
dyn_x/y/f, 23
dynamic particle creation, 14
dynamics, see eff_dynamics
Dynamics(), 25
eff_access(), 17
eff_bound(), 20
eff_constants, 21
eff_create, 21
eff_driver, 22
eff_dynamics, 23
eff_efield, 28
eff_eigen, 28
eff_eref, 29
eff_ewald, 30
energy/force calculation
see eff_update, 60
eff_initial, 37
eff_input, 41
eff_listint, 43
eff_listptr, 44
eff_minimize, 44
eff_output, 46
eff_params, 48
pressure, 51
eff_properties, 53
restrain, 53
self energy, 34
shape, 54
uniform charge, 54
modify Ewald, 6
modify non-Ewald, 6
energy/force calculation
see eff_update, 60
erfoverx(), 29
erfoverx1(), 29
erfoverx2(), 29
error function, see eff_eref
error(), 63
ewald_fx/ty/fz/alpha, 31
ewald_S_cap_real/imag, 31
ewald_S_real/imag, 31
ewald_x/y/z/alpha/q, 31
EwaldKCutoff, 34
EwaldRCutoff, 34
EwaldVirial(), 7, 32
ExecuteParameter(), 49
freeze_elec_{x/y/z/r}, 35
freeze_nuc_{x/y/z}, 33
frozen coordinates, see eff_freeze
GetBasename(), 64
GetDynamicsForces(), 26
GetDynamicsPositions(), 29
GetElecM(), 40
GetElectronForce(), 18
GetElectronKE(), 27
GetElectronMass(), 27
GetElectronPE(), 18
GetElectronPosition(), 18
GetElectronSpin(), 18
GetElectronTranslationKE(), 28
GetElectronVelocity(), 27
GetElecV(), 40
GetExtension(), 64
GetExternalField(), 28
GetFlexibleKEPressure(), 53
GetFlexiblePEPressure(), 52
GetInitialPosition(), 46
GetMass(), 64
GetMinimizeForces(), 46
GetNucElecForce(), 56
GetNucElecPosition(), 56
GetNuclearCharge(), 18
GetNuclearForce(), 18
GetNuclearMass(), 27
GetNuclearPosition(), 18
GetNuclearVelocity(), 27
GetNucleusKE(), 27
GetNucleusPE(), 18
GetNucV(), 40
GetNumElecM(), 39
GetNumElecV(), 39
GetNumNucM(), 39
GetNumNucV(), 39
GetRigidKEPressure(), 53
GetRigidPEPressure(), 52
GetTemperature(), 26
GetTotalKE(), 27
GetTotalPE(), 17
GetType(), 64
GetVolume(), 21
GetZ(), 64
global parameters, see eff_global
ierfervx1(), 6
initial velocities and masses, see eff_initial
Initialize(), 21
InitializeBounds(), 20
InitializeDynamics(), 24
InitializeEwald(), 31
InitializeFreezes(), 35
InitializeKSpace(), 32
InitializeParameters(), 49
InitializeRandomVelocities(), 25
InitializeRestraints(), 10, 55
InitializeRestraints(), 20
input file parser, see eff_input
integrate
IntitializeEnergyUpdater(), 62
isWrapX(), 20
isWrapY(), 20
isWrapZ(), 20
jacobi iteration, see eff_eigen
jacobi(), 29
K_ANGLE, 54
KCOORD, 55
K_DIHEDRAL, 55
K_DISTANCE, 55
K_LINE, 55
K_PLANE, 55
kinetic energy, 27
modify, 5
see eff_update, 60
KSpaceEnergy(), 61, 33
lattice_kx/y/z/2, 31
lattice_rx/y/z, 31
lcase(), 64
LeftStencil(), 62
license, 1
linked list
68
integer, see eff_listint
void pointers, see eff_listptr

main loop, see eff_driver
Maxwell-Boltzmann distribution, 25
min(), 64
min_x/f/g, 44
minimize, see eff_minimize
Minimize(), 46
MinimizeCallback(), 22
MinStorageRequirements(), 45

new_list_int(), 43
new_list_ptr(), 44
mic[], 36
NucElecIndex(), 42
NucElecString(), 56
nuclear-electron orbital, see protons
NullCMMotion(), 25
NumElectrons(), 19
numerical derivatives, 16 48
NumNuclei(), 19

orthogonalization
implicit vs explicit, 13
OutputEnergy(), 12 47
OutputEnergyForces(), 47
OutputFixedHeader(), 10 47
OutputParams(), 47
OutputPositions(), 47
OutputRestartFile(), 47
OutputRestartRestraints(), 12 58
OutputRestraints(), 57
OutputTimeElapsed(), 47
OutputVelocities(), 47

pairwise cutoff
modify, 7
ParamAdaptiveNumTries, 49
ParamAdaptiveEnergy, 49
ParamAdaptiveStepSize, 49
ParamAndersenCoupling, 49
ParamCalc, 49
ParamCutoff, 49
ParamDt, 49
ParamEField, 49
ParamEFieldFreq, 49
ParamEFieldPacketDuration, 49
ParamElectronMass, 49
parameter
add a new, 9
eff_params, see eff_params
table with defaults, 37
ParamEwald, 49
ParamMin, 49
ParamMinFreeze, 49
ParamNoseHooverCoupling, 49
ParamNumSteps, 49
ParamOutput, 49
ParamPeriodic, 49
ParamPrintEvery, 49
ParamRandSeed, 49
ParamSizeLimit, 49
ParamSizeLimitStiffness, 49
ParamTemperature, 49
ParamThermostat, 49
ParamXYZBound, 49
ParseString(), 42
parsing functions, 49
partition
rectangular cells, 61
Pauli potential
modify, 7
see eff_update, 60
PauliElecElec(), 5
physical constants, see eff_constants
poly01(), 30
poly02(), 30
poly1(), 30
poly2(), 30
pressure
eff_pressure, see eff_pressure
rigid vs flexible, 51
PrintAnalyticForces(), 48
PrintNumericalForces(), 48
ProcessElecMasses(), 48
ProcessElectron(), 41
ProcessElecVelocities(), 41
ProcessInputFile(), 41
ProcessNucleus(), 41
ProcessNucMasses(), 42
ProcessNucVelocities(), 41
ProcessParam(), 42
ProcessRestraints(), 42
properties, see eff_properties
property
add a new, 12
protons
  floating Gaussian, 14
rand_gauss(), 64
rand_int(), 64
rand_seed(), 64
rand_uni(), 64
ReadValue(), 51
restraint
  add a new, 10
  angle, 55
  coordinate, 55
  dihedral, 55
  distance, 55
  eff_restrain, see eff_restrain
  line, 55
  plane, 56
  tether, 53, 56
restraint labels, 42
RevertDynamics(), 7, 24
RightStencil(), 62
RSpaceEnergy(), 6, 33
RunDynamics(), 23
RunMinimize(), 22
RunSinglePoint(), 22
save_dyn_prev_f, 23
save_dyn_x/v/f, 23
SaveDynamics(), 7, 24
SelfEnergy(), 6, 34
SetDefaultParameters(), 49
SetDynamicsLogfile(), 24
SetDynamicsPositions(), 26
SetElecFR(), 20
SetElecFX(), 19
SetElecFY(), 20
SetElecFZ(), 20
SetEwaldCharges(), 31
SetMinimizePositions(), 46
SetNucFX(), 19
SetNucFY(), 19
SetNucFZ(), 19
SetNuclearMass(), 27
SetNuclearPosition(), 18
SetNuclearVelocity(), 26
SetTimeStep(), 26
SetupExternalField(), 28
Shanno minimizer, 45, see eff_shanno
spin annealing, 16
StartTimer(), 60
StopTimer(), 60
stopwatch, see eff_timing
string_match(), 42
StripNewline(), 42
temperature, 26
thermostat
  Andersen, 23
  making a chain, 8
  modify 7
  Nose-Hoover, 23, 25
TotalCharge(), 32
TotalTimeElapsed(), 60
TotalVolume(), 32
UniformChargeEnergy(), 6, 34
UpdateDynamicsPositions(), 7, 25
UpdateDynamicsVelocities(), 7, 25
UpdateDynamicsVirial(), 26
UpdateElecAndPauli(), 63
UpdateEnergyForces(), 63
UpdateEwaldEnergy(), 32
UpdateKineticEnergy(), 5, 63
UpdatePauliPeriodic(), 63
UpdateWidthRestraint(), 63
User time, see TotalTimeElapsed() 60
utility functions, see eff_util
velocity Verlet, see eff_dynamics
velocity Verlet algorithm, 25
writing files, see eff_output