Melanie Messih
13362933
19-03-2024

Documentation Molecular Mechanics Project

1. Modules

1.1. Constants

This module defines physical and molecular constants for molecular simulation calculations.

1.2. EnergyCalculations

This module contains procedures to calculate energy components from a molecule's structure.

1.3. HelperCalculations

This module contains helper functions for geometric and bonding calculations.

1.4. IdentifyInteractions

This module contains procedures to identify existent bond patterns in a molecule.

1.5. InputOutput

This module contains subroutines for reading molecular data from files and writing output.

1.6. MetropolisAlgorithm

This module contains a subroutine to minimize an energy value using a Metropolis algorithm.

1.7. MinimizationExperiment

This module coordinates a comprehensive molecular minimization experiment by optimizing energy configurations.

1.8. Types

This module defines the types for atoms, bonds, angles, dihedrals, and nonbonded interactions.

1.9. VectorMath

This module contains procedures to perform calculations between two vectors.

2. User-Derived Data Types

The user-derived data types can be found in the Types module.

2.1. Atom

Defines the Atom type for representing individual atoms and their properties. Contains properties unique for a specific atom type (C or H).

- **type** (character: 'C' or 'H')
- **x, y, z** (real(kind=8): Cartesian coordinates)
- **R_nb** (real(kind=8): Lennard-Jones parameter)
- **epsilon_nb** (real(kind=8): Lennard-Jones parameter)
- **q_nb** (real(kind=8): partial charge)

2.2. Bond

Defines the Bond type for representing covalent bonds between atoms. Contains properties unique for a specific bond type (CC or CH).

- **atom1, atom2** (integer: indices of the 2 atoms involved)
- **k** (real(kind=8): force constant)
- **r0** (real(kind=8): equilibrium bond length)

2.3. Angle

Defines the Angle type for representing angles formed by three atoms. Contains properties unique for a specific angle type (CCC, CCH or HCH).

- **atom1, atom2, atom3** (integer: indices of the 3 atoms involved)
- **k** (real(kind=8): force constant)

2.4. Dihedral

Defines the Dihedral type for representing torsional angles formed by four atoms. Contains properties unique for a specific dihedral angle type (CCCC, CCCH or HCCH).

- **atom1, atom2, atom3, atom4** (integer: indices of the 4 atoms involved)

2.5. NonBonded

Defines the NonBonded type for representing nonbonded interactions between atoms. HCCH). Contains properties unique for a specific nonbonded pair (CC, CH or HH).

- **atom1, atom2** (integer: indices of the 2 atoms involved)

3. Procedures

The different subroutines and functions used in this project are listed per module below.

3.1. Constants

-

3.2. EnergyCalculations

3.2.1. function CalculateStretchEnergy(atoms, bonds) result(stretchEnergy)
- Calculates the stretch (bond) energy of a molecule.

3.2.2. function CalculateBendEnergy(atoms, angles) result(bendEnergy)
- Calculates the bending energy of a molecule.

3.2.3. function CalculateTorsionEnergy(atoms, dihedrals) result(torsionEnergy)
- Calculates the torsional (dihedral) energy of a molecule.

3.2.4. function CalculateNonBondedEnergy(atoms, separationCount) result(nonbondedEnergy)
- Calculates the nonbonded (Van der Waals and electrostatic) energy of a molecule.

3.2.5. function CalculateTotalEnergy(stretchEnergy, bendEnergy, torsionEnergy, nonbondedEnergy) result(totalEnergy)
- Calculates the total energy of a molecule by summing all energy components.

3.3. HelperCalculations

3.3.1. function CalculateDistance(atomA, atomB) result(distanceAB)
- Calculates the Eucledian distance between two atoms.

3.3.2. function CalculateAngle(atomA, atomB, atomC) result(angleABC)
- Calculates the angle formed by three atoms.

3.3.3. function CalculateDihedralAngle(atomA, atomB, atomC, atomD) result(dihedralAngle)
- Calculates the dihedral angle formed by four atoms.

3.3.4. logical function CheckIfBondExists(atomIndex1, atomIndex2, bonds)
- Checks if a covalent bond is known to exist between two atoms.

3.4. IdentifyInteractions

3.4.1. subroutine IdentifyBonds(atoms, bonds, nAtoms, bondCount, CC_BondCount, CH_BondCount)
- Identifies bonds and counts the different types of bonds (CC, CH).

3.4.2. subroutine IdentifyAngles(atoms, bonds, angles, angleCount, CCC_count, CCH_count, HCH_count)
- Identifies angles and counts the different types of angles (CCC, CCH, HCH).

3.4.3. subroutine IdentifyDihedrals(atoms, bonds, dihedrals, dihedralCount, CCCC_count, CCCH_count, HCCH_count)
- Identifies dihedral angles and counts the different types of angles (CCCC, CCCH, HCCH).

3.4.4. subroutine IdentifyNonBondedSeparations(atoms, bonds, angles, separationMatrix, CC_nbCount, HC_nbCount, HH_nbCount)
- Identifies nonbonded interactions between atom pairs and counts the different types (CC, HC, HH).

3.4.5. subroutine UpdateBondInfo(bonds, bondCount, atomA, atomB, kValue, r0Value)
- Updates bond information with new atom indices, force constant, and equilibrium distance.

3.4.6. subroutine UpdateAngleInfo(angles, angleCount, atomA, atomB, atomC, kValue)
- Updates angle information with new atom indices and force constant.

3.4.7. subroutine UpdateBondsAllocation(bonds, bondCount)
- Allocates or reallocates the bonds array based on the current bond count.

3.4.8. subroutine UpdateAnglesAllocation(angles, angleCount)
- Allocates or reallocates the bonds array based on the current angle count.

3.4.9. subroutine UpdateDihedralsAllocation(dihedrals, dihedralCount)
- Allocates or reallocates the dihedrals array based on the current dihedral count.

3.4.10. function SelectAngleAtom(bondParameter, vertexAtom) result(nonVertexAtom)
- Returns the index of the non-vertex atom in a bond relative to a specified vertex atom of the angle.

3.4.11. function CheckForDuplicateAngle(angles, angleCount, atomA, atomB, atomC) result(alreadyExists)
- Checks if a given angle is already listed in the angles array.

3.4.12. function CheckForDuplicateDihedral(dihedrals, dihedralCount, atomA, atomB, atomC, atomD) result(alreadyExists)
- Checks for the existence of a dihedral angle to prevent duplicates.

3.4.13. function CheckIfNonbonded(atom1Index, atom2Index, bonds, angles) result(areNonbonded)
- Determines if two atoms are nonbonded considering direct bonds and two-bond separations.

3.5. InputOutput

3.5.1. subroutine ReadMolecule(filename, atoms)
- Reads molecular data from a file and initializes atom properties.

3.5.2. subroutine WriteEnergyOutput(inputFilename, CC_bondCount, CH_bondCount, CCC_count, CCH_count, HCH_count, CCCC_count, CCCH_count, HCCH_count,

CC_nbCount, HC_nbCount, HH_nbCount, i_stretchEnergy, i_bendEnergy, i_torsionEnergy, i_nbEnergy, i_totalEnergy, f_stretchEnergy, f_bendEnergy, f_torsionEnergy, f_nbEnergy, f_totalEnergy, iterationsTaken, maxIterations, hasConverged)

- Writes simulation results to a file, including interaction type counts and energy components.

3.5.3. subroutine WriteOptimizedGeometry(inputFilename, atoms)

- Writes the optimized geometry to an XYZ file.

3.5.4. subroutine ExtractMoleculeName(inputFilename, moleculeName)

- Extracts the molecule's name from the input filename.

3.5.5. subroutine ConvertToUpperCase(str)

- Converts a string to uppercase.

3.6. MetropolisAlgorithm

3.6.1. subroutine MetropolisMinimization(atoms, bonds, angles, dihedrals, separationCount, T, r, maxIter, tolerance, hasConverged, iterTaken)

- Performs energy minimization using the Metropolis algorithm.

3.7. MinimizationExperiment

3.7.1. subroutine RunMinimizationExperiment(inputFilename, T, r, tolerance, maxIter)

- Runs a molecular energy minimization experiment by applying the Metropolis algorithm.

3.8. Types

-

3.9. VectorMath

3.9.1 function Create3DVector(atom1, atom2) result(vector)

- Creates a 3D vector pointing from atom1 to atom2.

3.9.2 function DotProduct(vecA, vecB) result(dotProd)

- Calculates and returns the dot product of two vectors.

3.9.3 function CrossProduct(vecA, vecB) result(crossVec)

- Calculates and returns the cross product of two vectors.

4. Interdependencies

4.1. Data Flow Diagram

The interdependencies between the different modules are presented in **Figure 1**.

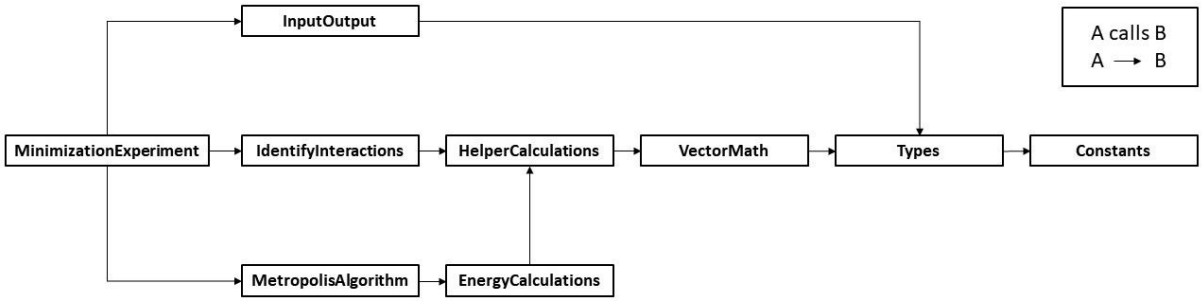

Figure 1. Data flow diagram of the modules in this Molecular Mechanics project.

The *MinimizationExperiment* module contains the subroutine RunMinimizationExperiment (inputFilename, T, r, tolerance, maxIter), which is called to run the full experiment in the main program.

5. Input and Output

5.1. Input

The input file is expected to be an xyz file which contains the Cartesian coordinates of a molecule that consists of both carbon and hydrogen atoms and that only contains single bonds. The filename is expected to contain the molecular formula of the molecule (e.g. c4h10.xyz).

5.2. Output

The first output file is a txt file that contains the simulation results, including the counts for the different interaction types (e.g. CC bonded interactions, HCH bending interactions, etc.) and the initial and final values of the energy components. The filename is an extension of the input filename (e.g. energy_minimization_output_c4h10.txt).

The second output file is an xyz file that contains the final Cartesian coordinates of the molecule. It is very similar to the input file, but contains the updated coordinates corresponding to the optimized geometry. The filename is an extension of the input filename (e.g. optimized_geometry_c4h10.xyz).