Documentation Molecular Mechanics Project

* 1. **Modules**
  2. **Constants**

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

* 1. **EnergyCalculations**

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

* 1. **HelperCalculations**

This module contains helper functions for geometric and bonding calculations.

* 1. **IdentifyInteractions**

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

* 1. **InputOutput**

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

* 1. **MetropolisAlgorithm**

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

* 1. **MinimizationExperiment**

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

* 1. **Types**

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

* 1. **VectorMath**

This module contains procedures to perform calculations between two vectors.

1. **User-Derived Data Types**

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

* 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)
  1. **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)
  1. **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)
  1. **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)
  1. **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)

1. **Procedures**

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

* 1. **Constants**

-

* 1. **EnergyCalculations**
     1. function CalculateStretchEnergy(atoms, bonds) result(stretchEnergy)
  + Calculates the stretch (bond) energy of a molecule.
    1. function CalculateBendEnergy(atoms, angles) result(bendEnergy)
  + Calculates the bending energy of a molecule.
    1. function CalculateTorsionEnergy(atoms, dihedrals) result(torsionEnergy)
  + Calculates the torsional (dihedral) energy of a molecule.
    1. function CalculateNonBondedEnergy(atoms, separationCount) result(nonbondedEnergy)
  + Calculates the nonbonded (Van der Waals and electrostatic) energy of a molecule.
    1. function CalculateTotalEnergy(stretchEnergy, bendEnergy, torsionEnergy, nonbondedEnergy) result(totalEnergy)
  + Calculates the total energy of a molecule by summing all energy components.
  1. **HelperCalculations**
     1. function CalculateDistance(atomA, atomB) result(distanceAB)
  + Calculates the Eucledian distance between two atoms.
    1. function CalculateAngle(atomA, atomB, atomC) result(angleABC)
  + Calculates the angle formed by three atoms.
    1. function CalculateDihedralAngle(atomA, atomB, atomC, atomD) result(dihedralAngle)
  + Calculates the dihedral angle formed by four atoms.
    1. logical function CheckIfBondExists(atomIndex1, atomIndex2, bonds)
  + Checks if a covalent bond is known to exist between two atoms.
  1. **IdentifyInteractions**
     1. subroutine IdentifyBonds(atoms, bonds, nAtoms, bondCount, CC\_BondCount, CH\_BondCount)
  + Identifies bonds and counts the different types of bonds (CC, CH).
    1. subroutine IdentifyAngles(atoms, bonds, angles, angleCount, CCC\_count, CCH\_count, HCH\_count)
  + Identifies angles and counts the different types of angles (CCC, CCH, HCH).
    1. 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).
    1. 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).
    1. subroutine UpdateBondInfo(bonds, bondCount, atomA, atomB, kValue, r0Value)
  + Updates bond information with new atom indices, force constant, and equilibrium distance.
    1. subroutine UpdateAngleInfo(angles, angleCount, atomA, atomB, atomC, kValue)
  + Updates angle information with new atom indices and force constant.
    1. subroutine UpdateBondsAllocation(bonds, bondCount)
  + Allocates or reallocates the bonds array based on the current bond count.
    1. subroutine UpdateAnglesAllocation(angles, angleCount)
  + Allocates or reallocates the bonds array based on the current angle count.
    1. subroutine UpdateDihedralsAllocation(dihedrals, dihedralCount)
  + Allocates or reallocates the dihedrals array based on the current dihedral count.
    1. 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.
    1. function CheckForDuplicateAngle(angles, angleCount, atomA, atomB, atomC) result(alreadyExists)
  + Checks if a given angle is already listed in the angles array.
    1. function CheckForDuplicateDihedral(dihedrals, dihedralCount, atomA, atomB, atomC, atomD) result(alreadyExists)
  + Checks for the existence of a dihedral angle to prevent duplicates.
    1. function CheckIfNonbonded(atom1Index, atom2Index, bonds, angles) result(areNonbonded)
  + Determines if two atoms are nonbonded considering direct bonds and two-bond separations.
  1. **InputOutput**
     1. subroutine ReadMolecule(filename, atoms)
  + Reads molecular data from a file and initializes atom properties.
    1. subroutine WriteOutput(filename, 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.
    1. subroutine ConvertToUpperCase(str)
  + Converts a string to uppercase.
  1. **MetropolisAlgorithm**
     1. subroutine MetropolisMinimization(atoms, bonds, angles, dihedrals, separationCount, T, r, maxIter, tolerance, hasConverged, iterTaken)
  + Performs energy minimization using the Metropolis algorithm.
  1. **MinimizationExperiment**
     1. subroutine RunMinimizationExperiment(inputFilename, outputFilename, T, r, tolerance, maxIter)
  + Runs a molecular energy minimization experiment by applying the Metropolis algorithm.
  1. **Types**

-

* 1. **VectorMath**
     1. function Create3DVector(atom1, atom2) result(vector)
  + Creates a 3D vector pointing from atom1 to atom2.
    1. function DotProduct(vecA, vecB) result(dotProd)
  + Calculates and returns the dot product of two vectors.
    1. function CrossProduct(vecA, vecB) result(crossVec)
  + Calculates and returns the cross product of two vectors.

1. **Interdependencies**

**4.1. Data Flow Diagram**

The interdependencies between the different modules are presented in **Figure 1**. **Afbeelding met tekst, schermopname, Lettertype, diagram

Automatisch gegenereerde beschrijving**

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

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

1. **Input and Output**
   1. **Input**

The input file is expected to be a XYZ file which includes the Cartesian coordinates of a molecule that contains both carbon and hydrogen atoms and that only contains single bonds.

* 1. **Output**

…

- Input/Output handling

GitHub:

* Code (updated: main, …)
* Documentation (as PDF!)
* Examples Input and output
* Update Readme: introduction + GitHub directory / file structures
* Remove test files
* Check nonbonded energy calculations
* Main program description
* Remove logical function
* Remove epsilon0
* Public/private nagaan