Project Documentation

Title:

Implementation of GUI Prototype for MD-CUBE using QT

Table of Contents

Introduction

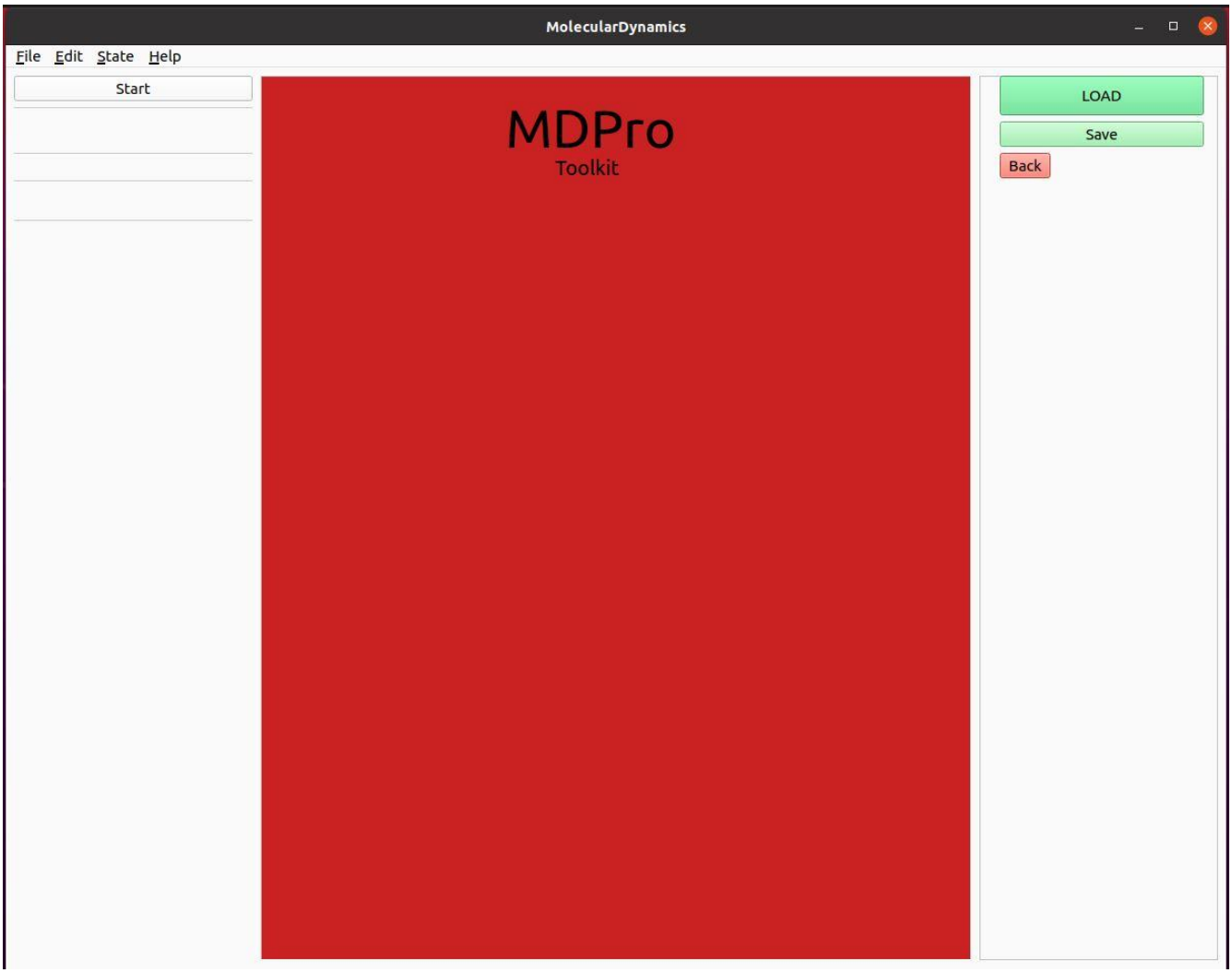
The Molecular Dynamics Application is a Qt-based software designed to facilitate molecular dynamics simulations. It provides a user-friendly graphical user interface (GUI) for editing input files, which define the parameters and settings for the simulation. This documentation outlines the key features and functionalities of the application.

Features

Input File

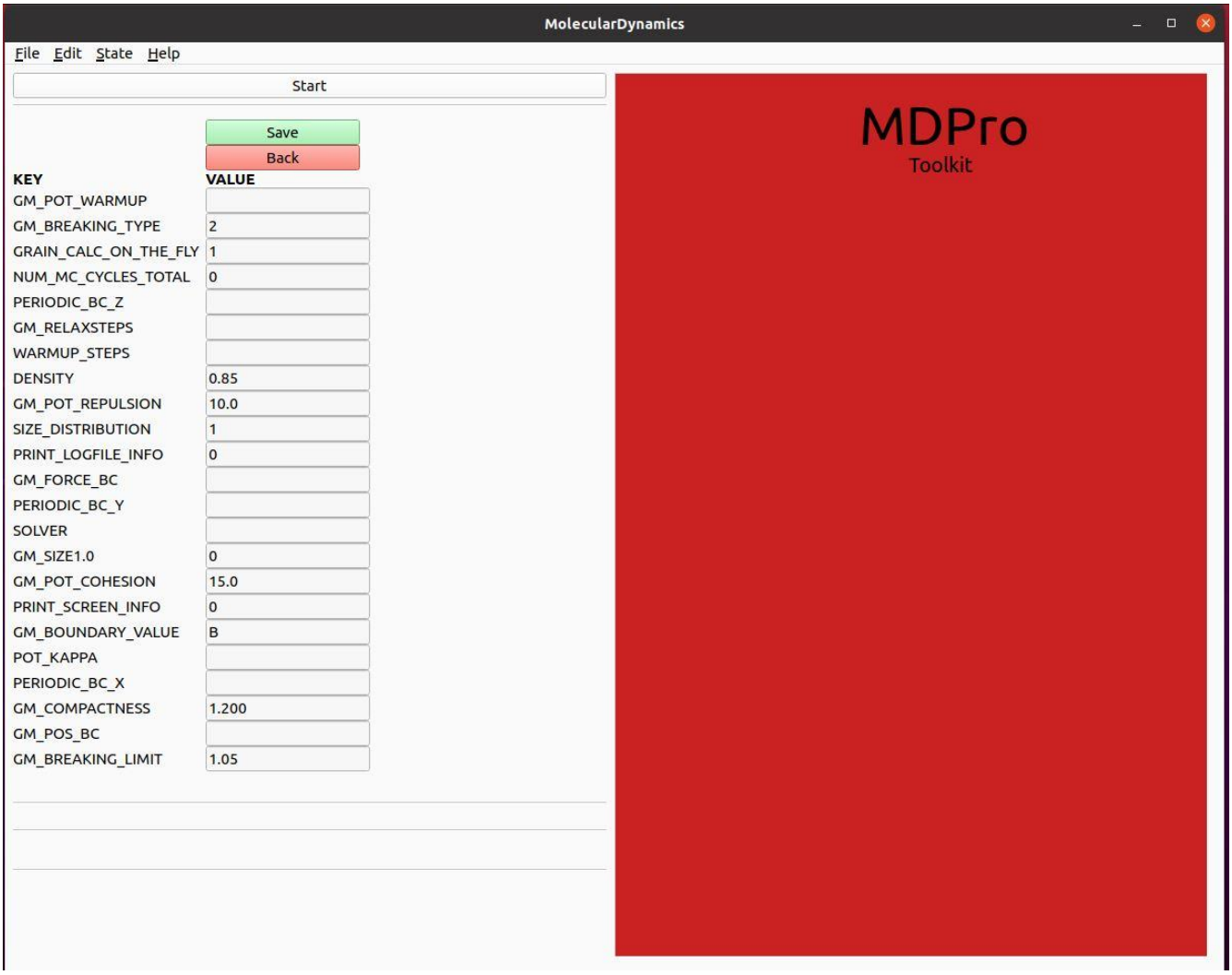
The application utilizes a binary input file format consisting of multiple keys and their corresponding values. Each key can have an integer (INT) or double (floating-point) value associated with it. The input file supports a maximum of 30 keys.

The GUI provides an interface to upload the binary input file from local storage.



Display and edit

The software provides an editor form to display and edit the content of the input file. The editor presents the keys and values in a user-friendly manner, allowing easy modification of the simulation parameters.



Update

Values associated with the keys can be adjusted. When modifications are made to the input file in the editor, the application provides a "Save" option. Upon clicking "Save," a warning message is displayed, asking whether the user wants to overwrite the original file. If the user selects "No," there is an option to save the modified file as a new copy.

File Storage

The modified file, whether saved as a new copy or overwritten, is stored on the local disk. The application allows users to specify the destination directory for the saved files.

Code explanation

Code to Overwrite file.

```
void overwriteFile()
{
    QFile file(global_file);
```

```
    if (file.open(QIODevice::WriteOnly | QIODevice::Text)) {
        QTextStream stream(&file);
        for (auto it = dataHash.constBegin(); it != dataHash.constEnd(); ++it) {
            stream <<  it.key() << " " << it.value() << "\n";
        }

        file.close();
    }
    QMessageBox::information(nullptr, "File Saved", "The file has been successfully
saved.");
}
```

The `overwriteFile()` function is responsible for overwriting a file with the contents of a `dataHash` container. It opens the file specified by the `global_file` path in write-only mode and as a text file. If the file is successfully opened, it creates a `QTextStream` object to write the key-value pairs from `dataHash` to the file, separating each key and value with a space and ending each line with a newline character. Finally, the function closes the file and displays an informational message box indicating that the file has been successfully saved.

Code to Load file

```
void MainWindow::on_pushButton_2_clicked()
{
    QString filename = QFileDialog::getOpenFileName(this,"Open binary file",
QDir::homePath());
    global_file = filename;
    QFile file(filename);

    std::unordered_map<std::string, std::string> dataMap;
    std::string line;
    if(!file.open(QFile::ReadOnly | QFile::Unbuffered)){
        QMessageBox::warning(this,"Error","File can not open");
    }

    QTextStream in(&file);


    while (!in.atEnd()) {
        QString line = in.readLine().trimmed();

        if (line.isEmpty() || line.startsWith('#') || line.startsWith('=')) {
            continue;  // Ignore empty lines and comment lines starting with '#' or '='
        }

        QStringList parts = line.split(' ');

        // Remove any empty parts
        parts.removeAll("");

        if (parts.size() >= 2) {
            QString key = parts[0];
            QString value = parts[1];
            dataHash[key] = value;
        }
    }
```

```
    file.close();
    QLayoutItem* item;
    QFormLayout* formLayout = ui->formLayout;
    while ((item = formLayout->takeAt(0)) != nullptr) {
        QWidget* widget = item->widget();
        if (widget != nullptr) {
            formLayout->removeWidget(widget);
            delete widget;
        }
        delete item;
    }
    QHashIterator<QString, QString> it(dataHash);
    QFormLayout *pb(ui->formLayout);
    QPushButton* saveButton = new QPushButton("Save");
    pb->addWidget(saveButton);

    QPushButton* backButton = new QPushButton("Back");
    pb->addWidget(backButton);
    QObject::connect(saveButton, &QPushButton::clicked,[this](){

        qDebug() << "-------------------------";
        QHashIterator<QString, QString> it(dataHash);
        while (it.hasNext()) {
            it.next();
            qDebug() << "Key:" << it.key() << ", Value:" << it.value();
        }
        QMessageBox::StandardButton reply;
        reply = QMessageBox::question(this, "File Overwrite", "The file already exists. Do
you want to overwrite it?",
                                      QMessageBox::Yes | QMessageBox::Cancel);
        if (reply == QMessageBox::Yes) {
            overwriteFile();
            ui->stackedWidget->setCurrentIndex(1);
            dataHash.clear();
        }
});

    QObject::connect(backButton, &QPushButton::clicked,[this](){

        ui->stackedWidget->setCurrentIndex(1);
        dataHash.clear();

    });

     while (it.hasNext()) {
        it.next();
        QString key = it.key();
        QString value = it.value();

     }
     // listeners
      QFormLayout* layout = ui->formLayout;
     for (auto it = dataHash.constBegin(); it != dataHash.constEnd(); ++it) {
        QLabel* lab = new QLabel(it.key());
        QLineEdit* lineEdit = new QLineEdit(it.value());
        layout->addRow(lab);
```

```
        layout->addRow(lineEdit);

        // Connect the textChanged signal of each QLineEdit to a custom slot
        QObject::connect(lineEdit, &QLineEdit::textChanged, [it](const QString& newValue)
{
            // Update the value in dataHash when the text in the QLineEdit changes
            dataHash[it.key()] = newValue;
            qDebug() << newValue << "aaa";
        });
     }
    ui->stackedWidget->setCurrentIndex(2);

}
```

The function is a member function of the `MainWindow` class and is responsible for loading and processing a binary file selected by the user.

First, it prompts the user to select a file using a file dialog and assigns the chosen filename to the filename variable. It then sets the `global_file` variable to the selected filename.

Next, it opens the file in read-only mode and checks if the file is successfully opened. If there's an error opening the file, it displays a warning message using a `QMessageBox`.

A `QTextStream` object is created to read the contents of the file. The function reads the file line by line until reaching the end. It trims each line to remove any leading or trailing whitespace.

Lines that are empty or start with '#' or '=' characters are ignored as they are considered comments or irrelevant. The remaining lines are split into parts using the space character as a delimiter.

If a line has at least two parts, the first part is treated as the key and the second part as the value. The key-value pair is then stored in the `dataHash` container.

After processing the file, the function closes it. It then removes any existing widgets from the `formLayout` in the UI, preparing it for the new key-value pairs.

Next, the function adds "Save" and "Back" buttons to the `formLayout` and connects their clicked signals to respective slots. The "Save" button's slot checks if the file already exists and asks the user if they want to overwrite it. If the user chooses to overwrite, the `overwriteFile()` function is called, the UI is set to the appropriate index, and the `dataHash` is cleared.

The "Back" button's slot sets the UI to the appropriate index and clears the `dataHash`.

The function then iterates over the `dataHash` container to retrieve the keys and values without performing any operation on them.

Finally, it sets up the UI by creating `QLabel` and `QLineEdit` widgets for each key-value pair in the `dataHash` container. It connects the `textChanged` signal of each `QLineEdit` to a custom slot, which updates the corresponding value in the `dataHash` container when the text is changed. Once all the widgets are set up, the UI is set to the appropriate index, indicating that the loading and processing of the file are complete.

Conclusion

The Molecular Dynamics Application provides a comprehensive set of features for editing input files and configuring molecular dynamics simulations. With its intuitive GUI editor, value manipulation widgets, save options, and file storage capabilities, the software aims to simplify the setup process for molecular dynamics simulations.