Part I

Theory

Part II

Results

Appendices

A
Dirac Notation

Calculations involving sums over inner products of orthogonal states are common in Quantum Mechanics. This due to the fact that eigenfunctions of Hermitian operators, which is the kind of operators which represent observables[1], are necessarily orthogonal[2]. These inner-products will in many cases result in either zero or one, i.e. the *Kronecker-delta* function δ_{ij}; explicitly calculating the integrals is unnecessary.

Dirac notation is a notation in which quantum states are represented as abstract components of a *Hilbert space*, i.e. an inner product space. This implies that the inner-product between two states are represented by these states alone, without the integral over a specific basis, which makes derivations a lot cleaner and general in the sense that no specific basis is needed.

Extracting the abstract state from a wave function is done by realizing that the wave function can be written as the inner product between the position basis eigenstates $|x\rangle$ and the abstract quantum state $|\psi\rangle$

$$\psi(x) = \langle r, \psi \rangle \equiv \langle x|\psi\rangle = \langle x| \times |\psi\rangle \, .$$

The notation is designed to be simple. The right hand side of the inner product is called a *ket*, while the left hand side is called a *bra*. Combining both of them leaves you with an inner product bracket, hence Dirac notation is commonly referred to as *bra-ket* notation.

To demonstrate the simplicity introduced with this notation, imagine a coupled two-level spin-$\frac{1}{2}$ system in the following state

$$\begin{aligned} |\chi\rangle &= N\Big[\,|\uparrow\downarrow\rangle - i\,|\downarrow\uparrow\rangle\,\Big] && \text{(A.1)} \\ \langle\chi| &= N\Big[\,\langle\uparrow\downarrow| + i\,\langle\downarrow\uparrow|\,\Big] && \text{(A.2)} \end{aligned}$$

Using the fact that both the $|\chi\rangle$ state and the two-level spin states should be orthonormal, the normalization factor can be calculated without explicitly setting up any integrals

$$\begin{aligned}
\langle\chi|\chi\rangle &= N^2\Big[\langle\uparrow\downarrow| + i\,\langle\downarrow\uparrow|\Big]\Big[|\uparrow\downarrow\rangle - i\,|\downarrow\uparrow\rangle\Big] \\
&= N^2\Big[\langle\uparrow\downarrow\,|\,\uparrow\downarrow\rangle + i\,\langle\downarrow\uparrow\,|\,\uparrow\downarrow\rangle - i\,\langle\uparrow\downarrow\,|\,\downarrow\uparrow\rangle + \langle\downarrow\uparrow\,|\,\downarrow\uparrow\rangle\Big] \\
&= N^2\Big[1+0-0+1\Big] \\
&= 2N^2 \\
&= 1,
\end{aligned}$$

This implies the trivial solution $N = 1/\sqrt{2}$. With this powerful notation at hand, important properties such as the *completeness relation* of a set of states can be shown. A standard strategy is to start by expanding one state $|\phi\rangle$ in a complete set of different states $|\psi_i\rangle$:

$$\begin{aligned}
|\phi\rangle &= \sum_i c_i\,|\psi_i\rangle \\
\langle\psi_k|\phi\rangle &= \sum_i c_i\,\underbrace{\langle\psi_k|\psi_i\rangle}_{\delta_{ik}} \\
&= c_k \\
|\phi\rangle &= \sum_i \langle\psi_i|\phi\rangle\,|\psi_i\rangle \\
&= \left[\sum_i |\psi_i\rangle\,\langle\psi_i|\right]|\phi\rangle
\end{aligned}$$

which implies that

$$\sum_i |\psi_i\rangle\,\langle\psi_i| = \mathbb{1} \tag{A.3}$$

for any complete set of orthonormal states $|\psi_i\rangle$. Calculating the corresponding identity for a continuous basis like e.g. the position basis yields

$$\begin{aligned}
\int |\psi(x)|^2 dx &= 1 \qquad \text{(A.4)} \\
\int |\psi(x)|^2 dx &= \int \psi^*(x)\psi(x)\mathrm{d}x \\
&= \int \langle\psi|x\rangle\,\langle x|\psi\rangle\,\mathrm{d}x \\
&= \langle\psi|\Big[\int |x\rangle\,\langle x|\,dx\Big]|\psi\rangle\,. \qquad \text{(A.5)}
\end{aligned}$$

Combining eq. A.4 and eq. A.5 with the fact that $\langle\psi|\psi\rangle = 1$ yields the identity

$$\int |x\rangle\,\langle x|\,dx = \mathbb{1}. \tag{A.6}$$

Looking back at the introductory example, this identity is exactly what is extracted when a wave function is described as an inner product instead of an explicit function.

B
DCViz: Visualization of Data

With a code framework increasing in complexity comes an increasing need for tools to ease the interface between the code and the developer(s). In computational science, a must-have tool is a tool for efficient visualization of data; there is only so much information a single number can hold. To supplement the QMC code, a visualization tool named DCViz (**D**ynamic **C**olumn data **Vi**suali**z**er) has been developed.

The tool is written in Python, designed to plot data stored in columns. The tool is not designed explicitly for the QMC framework, and has been successfully applied to codes by several Master students at the time of this thesis. The plot library used is *Matplotlib*[3] with a graphical user interface coded using *PySide*[4]. The data can be plotted dynamically at a specified interval, and designed to be run parallel to the main application, e.g. DMC.

DCViz is available at `https://github.com/jorgehog/DCViz`

B.1 Basic Usage

The application is centered around the `mainloop()` function, which handles the extraction of data, the figures and so on. The virtual function `plot()` is where the user specifies how the data is transformed into specified figures by creating a subclass which overloads it. The superclass handles everything from safe reading from dynamically changing files, efficient and safe re-plotting of data, etc. automatically. The tool is designed to be simple to use by having a minimalistic interface for implementing new visualization classes. The only necessary members to specify in a new implementation is described in the first three sections, from where the remaining sections will cover additional support.

The figure map

Represented by the member variable `figMap`, the figure map is where the user specifies the figure setup, that is, the names of the main-figures and their sub-figures. Consider the following figure map:

```python
figMap = {"mainFig1": ["subFig1", "subFig2"], "mainFig2": []}
```

This would cause DCViz to create two main-figures `self.mainFig1` and `self.mainFig2`, which can be accessed in the plot function. Moreover, the first main-figure will contain two sub-figures accessible through `self.subFig1` and `self.subFig2`. These sub-figures will be stacked vertically if not `stack="H"` is specified, in which they will be stacked horizontally.

The name tag

Having to manually combine a data file with the correct subclass implementation is annoying, hence DCViz is designed to automate this process. Assuming a dataset to be specified by a unique pattern of characters, i.e. a *name tag*, this name tag can be tied to a specific subclass implementation, allowing DCViz to automatically match a specific filename with the correct subclass. Name tags are

```
 nametag = "DMC_out_\d+\.dat"
```

The name tag has *regular expressions* (regex) support, which in the case of the above example allows DCViz to recognize any filename starting with "DMC_out_" followed by any integer and ending with ".dat" as belonging to this specific subclass. This is a necessary functionality, as filenames often differ between runs, that is, the filename is specified by e.g. a time step, which does not fit an absolute generic expression. The subclasses must be implemented in the file `DCViz_classes.py` in order for the automagic detection to work.

To summarize, the name tag invokes the following functionality

```
import DCvizWrapper, DCViz_classes

#DCViz automagically executes the mainloop for the
#subclass with a nametag matching 'filename'
DCVizWrapper.main(filename)

#This would be the alternative, where 'specific_class' needs to be manually selected.
specificClass = DCViz_classes.myDCVizClass #myDCVizClass matches 'filename'
specificClass(filename).mainloop()
```

The plot function

Now that the figures and the name tag has been specified, all that remains for a fully functional DCViz instance is the actual plot function

```
def plot(self, data)
```

where `data` contains the data harvested from the supplied filename. The columns can then be accessed easily by e.g.

```
col1, col2, col3 = data
```

which can then in turn be used in standard Matplotlib functions with the figures from `figMap`.

Additional (optional) support

Additional parameters can be overloaded for additional functionality

`nCols`	The number of columns present in the file. Will be automatically detected unless the data is stored in binary format.
`skipRows`	The number of initial rows to skip. Will be automatically detected unless the data is stored as a single column.
`skipCols`	The number of initial columns to skip. Defaults to zero.
`armaBin`	Boolean flag. If set to true, the data is assumed to be stored in Armadillo's binary format (doubles). Number of columns and rows will be read from the file header.
`fileBin`	Boolean flag. If set to true, the data is assumed to be stored in binary format. The number of columns must be specified.

The $\LaTeX$support is enabled if the correct packages is installed.

An example

```python
#DCViz_classes.py

from DCViz_sup import DCVizPlotter #Import the superclass

class myTestClass(DCVizPlotter): #Create a new subclass
    nametag =  'testcase\d\.dat' #filename with regex support

    #1 figure with 1 subfigure
    figMap = {'fig1': ['subfig1']}

    #skip first row (must be supplied since the data is 1D).
    skipRows = 1

    def plot(self, data):
        column1 = data[0]

        self.subfig1.set_title('I have $\LaTeX$ support!')

        self.subfig1.set_ylim([-1,1])

        self.subfig1.plot(column1)

        #exit function
```

Families

A specific implementation can be flagged as belonging to a family of similar files, that is, files in the same directory matching the same name tag. Flagging a specific DCViz subclass as a family is achieved by setting the class member variable `isFamilyMember` to true. When a family class is initialized with a file, DCViz scans the file's folder for additional matches to this specific class. If several matches are found, all of these are loaded into the `data` object given as input to the plot function. In this case `data[i]` contains the column data of file i.

To keep track of which file a given data-set was loaded from, a list `self.familyFileNames` is created, where element i is the filename corresponding to `data[i]`. To demonstrate this, consider the following example

```python
isFamilyMember = True
def plot(self, data)

    file1, file2 = data
    fileName1, fileName2 = self.familyFileNames

    col1_1, col_2_1 = file1
    col1_2, col_2_2 = file2
    #...
```

A class member string `familyName` can be overridden to display a more general name in the auto-detection feedback.

Families are an important functionality in the cases where the necessary data is spread across several files. For instance, in the QMC library, the radial distributions of both VMC and DMC are needed in order to generate the plots shown in figure ?? of the results chapter. These results may be generated in separate runs, which implies that they either needs to be loaded as a family, or be concatenated beforehand. Which dataset belongs to VMC and DMC can be extracted from the list of family file names.

All the previous mentioned functionality is available for families.

Family example

```python
#DCViz_classes.py

from DCViz_sup import DCVizPlotter

class myTestClassFamily(DCVizPlotter):
    nametag =  'testcaseFamily\d\.dat' #filename with regex support

    #1 figure with 3 subfigures
    figMap = {'fig1': ['subfig1', 'subfig2', 'subfig3']}

    #skip first row of each data file.
    skipRows = 1

    #Using this flag will read all the files matching the nametag
    #(in the same folder.) and make them aviable in the data arg
    isFamilyMember = True
    familyName = "testcase"

    def plot(self, data):

        mainFig = self.fig1
        mainFig.suptitle('I have $\LaTeX$ support!')
        subfigs = [self.subfig1, self.subfig2, self.subfig3]

        #Notice that fileData.data is plotted (the numpy matrix of the columns)
        #and not fileData alone, as fileData is a 'dataGenerator' instance
        #used to speed up file reading. Alternatively, data[:] could be sent
        for subfig, fileData in zip(subfigs, data):
            subfig.plot(fileData.data)
            subfig.set_ylim([-1,1])
```

loading e.g. `testcaseFamily0.dat` would automatically load `testcaseFamily1.dat` etc. as well.

Dynamic mode

Dynamic mode in DCViz is enabled on construction of the object

```python
DCVizObj = myDCVizClass(filename, dynamic=True)
DCVizObj.mainloop()
```

This flag lets the mainloop know that it should not stop after the initial plot is generated, but rather keep on reading and plotting the file(s) until the user ends the loop with either a keyboard-interrupt (which is caught and safely handled), or in the case of using the GUI, with the stop button.

In order to make this functionality more CPU-friendly, a `delay` parameter can be adjusted to specify a pause period in between re-plotting.

Saving figures to file

The generated figures can be saved to file by passing a flag to the constructor

```python
DCVizObj = myDCVizClass(filename, toFile=True)
DCVizObj.mainloop()
```

In this case, dynamic mode is disabled and the figures will not be drawn on screen, but rather saved in a subfolder of the supplied filename's folder called `DCViz_out`.

B.1.1 The Terminal Client

The `DCVizWrapper.py` script is designed to be called from the terminal with the path to a datafile specified as command line input. From here it automatically selects the correct subclass based on the filename:

```
jorgen@teleport:~$ python DCVizWrapper.py ./ASGD_out.dat

[ Detector ] Found subclasses 'myTestClass', 'myTestClassFamily', 'EnergyTrail',
                              'Blocking', 'DMC_OUT', 'radial_out', 'dist_out',
                              'R_vs_E', 'E_vs_w', 'testBinFile', 'MIN_OUT'
[  DCViz   ] Matched [ASGD_out.dat] with [MIN_OUT]
[  DCViz   ] Press any key to exit
```

If the option `-d` is supplied, dynamic mode is activated:

```
jorgen@teleport:~$ python DCVizWrapper.py ./ASGD_out.dat -d

[ Detector ] Found subclasses ......
[  DCViz   ] Matched [ASGD_out.dat] with [MIN_OUT]
[  DCViz   ] Interrupt dynamic mode with CTRL+C
^C[  DCViz   ] Ending session...
```

Saving figures through the terminal client is done by supplying the flag `-f` to the command line together with a folder `aDir`, whose content will then be traversed recursively. For every file matching a DCViz name tag, the file data will be loaded and its figure(s) saved to `aDir/DCViz_out/`. In case of family members, only one instance needs to be run (they would all produce the same image), hence "family portraits" are taken only once:

```
jorgen@teleport:~$ python DCVizWrapper.py ~/scratch/QMC_SCRATCH/ -f

[ Detector ] Found subclasses ......
[  DCViz   ] Matched [ASGD_out.dat] with [MIN_OUT]
[  DCViz   ] Figure(s) successfully saved.
[  DCViz   ] Matched [dist_out_QDots2c1vmc_edge3.05184.arma] with [dist_out]
[  DCViz   ] Figure(s) successfully saved.
[  DCViz   ] Matched [dist_out_QDots2c1vmc_edge3.09192.arma] with [dist_out]
[  DCViz   ] Family portait already taken, skipping...
[  DCViz   ] Matched [radial_out_QDots2c1vmc_edge3.05184.arma] with [radial_out]
[  DCViz   ] Figure(s) successfully saved.
[  DCViz   ] Matched [radial_out_QDots2c1vmc_edge3.09192.arma] with [radial_out]
[  DCViz   ] Family portait already taken, skipping...
```

The terminal client provides extremely efficient and robust visualization of data. When e.g. blocking data from 20 QMC runs, the automated figure saving functionality is gold.

B.1.2 The Application Programming Interface (API)

DCViz has been developed to interface nicely with any Python script. Given a path to the data file, all that is needed in order to visualize it is to include the wrapper function used by the terminal client:

```python
import DCVizWrapper as viz
dynamicMode = False #or true

...
#Generate some data and save it to the file myDataFile (including path)

#DCVizWrapper.main() automatically detects the subclass implementation
#matching the specified file. Thread safe and easily interruptable.
viz.main(myDataFile, dynamic=dynamicMode, toFile=toFile)
```

If on the other hand the data needs to be directly visualized without saving it to file, the pure API function `rawDataAPI` can be called directly with a numpy array `data`. If the plot should be saved to file, this can be enabled by supplying an arbitrary file-path (e.g. `/home/me/superDuper.png`) and setting `toFile=True`.

```python
from DCViz_classes import myDCVizClass

#Generate some data
myDCVizObj = myDCVizClass(saveFileName, toFile=ToFile)
myDCVizObj.rawDataAPI(data)
```

The GUI

The script `DCVizGUI.py` sets up a GUI for visualizing data using DCViz. The GUI is implemented using PySide (python wrapper for Qt), and is designed to be simple. Data files are loaded from an open-file dialog (`Ctrl+s` for entire folders or `Ctrl+o` for individual files), and will appear in a drop-down menu once loaded labeled with the corresponding class name. The play button executes the main loop of the currently selected data file. Dynamic mode is selected though a check-box, and the pause interval is set by a slider (from zero to ten seconds). Dynamic mode is interrupted by pressing the stop button. Warnings can be disabled through the configuration file. See figure B.1 for a screenshot of the GUI in action.

The GUI can be opened from any Python script by calling the `main` function (should be threaded if used as part of another application). If a path is supplied to the function, this path will be default in all file dialogues. Defaults to the current working directory.

The following is a tiny script executing the GUI for a QMC application. If no path is supplied at the command line, the default path is set to the scratch path.

```python
import sys, os
from pyLibQMC import paths #contains all files specific to the QMC library

#Adds DCVizGUI to the Python path
sys.path.append(os.path.join(paths.toolsPath, "DCViz", "GUI"))

import DCVizGUI

if __name__ == "__main__":

    if len(sys.argv) > 1:
        path = sys.argv[1]
    path = paths.scratchPath

    sys.exit(DCVizGUI.main(path))
```

The python script responsible for starting the QMC program and setting up the environments for simulations in this thesis automatically starts the GUI in the simulation main folder, which makes the visualizing the simulation extremely easy.

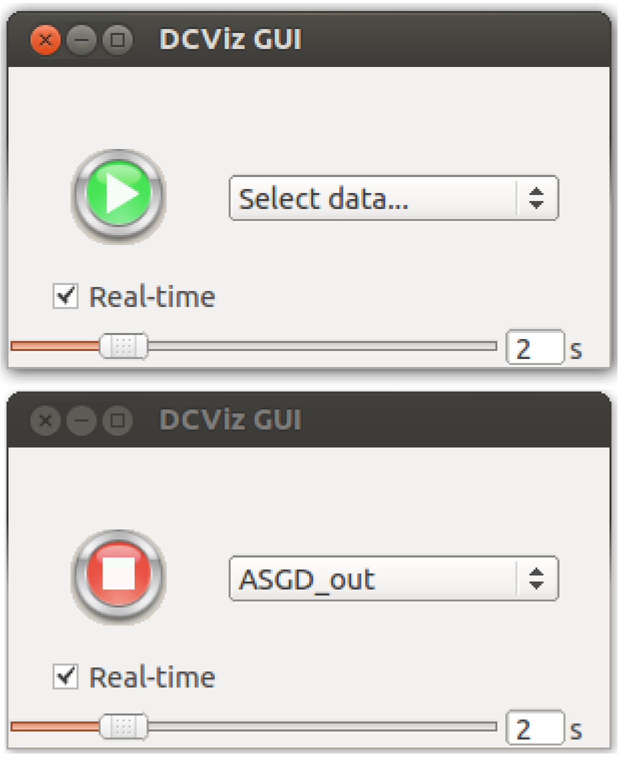

Figure B.1: Two consequent screen shots of the GUI. The first (top) is taken directly after the detector is finished loading files into the drop-down menu. The second is taken directly after the job is started.

Alternatively, DCVizGUI.py can be executed directly from the terminal with an optional default path as first command line argument.

The following is the terminal feedback supplied from opening the GUI

```
 .../DCViz/GUI$ python DCVizGUI.py
[ Detector ]:  Found subclasses 'myTestClass', 'myTestClassFamily', 'EnergyTrail',
'Blocking', 'DMC_OUT', 'radial_out', 'dist_out', 'testBinFile', 'MIN_OUT'
[   GUI    ]:  Data reset.
```

Selecting a folder from the open-folder dialog initializes the detector on all file content

```
[ Detector ]:  matched [  DMC_out.dat  ] with [    DMC_OUT    ]
[ Detector ]:  matched [  ASGD_out.dat ] with [    MIN_OUT    ]
[ Detector ]:  matched [blocking_DMC_out.dat] with [    Blocking    ]
[ Detector ]:  'blocking_MIN_out0_RAWDATA.arma' does not match any DCViz class
[ Detector ]:  'blocking_DMC_out_RAWDATA.arma' does not match any DCViz class
[ Detector ]:  matched [blocking_VMC_out.dat] with [    Blocking    ]
```

Executing a specific file selected from the drop-down menu starts a threaded job, hence several non-dynamic jobs can be ran at once. The limit is set to one dynamic job pr. application due to the high CPU cost (in case of a low pause timer).

The terminal output can be silenced through to configuration file to not interfere with the standard output of an application. Alternatively, the GUI thread can redirect its standard output to file.

C

Auto-generation with SymPy

"SymPy is a Python library for symbolic mathematics. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible. SymPy is written entirely in Python and does not require any external libraries."

- The SymPy Home Page [5]

This appendix will be focused on using SymPy to calculate closed form expressions for single particle wave functions needed to optimize the calculations of the Slater gradient and Laplacian. For systems of many particles, it is crucial to have these expressions in order for the code to remain efficient.

Calculating these expressions by hand is not human, given that the complexity of the expressions is proportional to the magnitude of the quantum number, which again scales with the number of particles. In the case of a 56 particle Quantum Dot, the number of unique derivatives involved in the simulation is 112.

C.1 Usage

SymPy is, as described in the introductory quote, designed to be simple to use. This section will cover the basics needed to calculate gradients and Laplacians, auto-generating C++ - and Latex code.

C.1.1 Symbolic Algebra

In order for SymPy to recognize e.g. `x` as a symbol, that is, a *mathematical variable*, special action must be made. In contrast to programming variables, symbols are not initialized to a value. Initializing symbols can be done in several ways, the two most common are listed below

```
In [1]: from sympy import Symbol, symbols

In [2]: x = Symbol('x')

In [3]: y, z = symbols('y z')

In [4]: x*x+y
Out[4]: 'x**2 + y'
```

The `Symbol` function handles single symbols, while `symbols` can initialize several symbols simultaneously. The string argument might seem redundant, however, this represents the *label* displayed using print functions, which is neat to control. In addition, key word arguments can be sent to the symbol functions, flagging variables as e.g. positive, real, etc.

```
In [1]: from sympy import Symbol, symbols, im

In [2]: x2 = Symbol('x^2', real=True, positive=True) #Flagged as real. Note the label.

In [3]: y, z = symbols('y z') #Not flagged as real

In [4]: x2+y #x2 is printed more nicely given a describing label
Out[4]: 'x^2 + y'

In [5]: im(z) #Imaginary part cannot be assumed to be anything.
Out[5]: 'im(z)'

In [6]: im(x2) #Flagged as real, the imaginary part is zero.
Out[6]: 0
```

C.1.2 Exporting C++ and Latex Code

Exporting code is extremely simple: SymPy functions exist in the `sympy.printing` module, which simply takes a SymPy expression on input and returns the requested code-style equivalent. Consider the following example

```
In [1]: from sympy import symbols, printing, exp

In [2]: x, x2 = symbols('x x^2')

In [3]: printing.ccode(x*x*x*x*exp(-x2*x))
Out[3]: 'pow(x, 4)*exp(-x*x^2)'

In [4]: printing.ccode(x*x*x*x)
Out[4]: 'pow(x, 4)'

In [5]: print printing.latex(x*x*x*x*exp(-x2))
\frac{x^{4}}{e^{x^{2}}}
```

The following expression is the direct output from line five compiled in Latex

$$\frac{x^{4}}{e^{x^{2}}}$$

C.1.3 Calculating Derivatives

The $2s$ orbital from hydrogen (not normalized) is chosen as an example for this section

$$\phi_{2s}(\vec{r}) = (Zr - 2)e^{-\frac{1}{2}Zr} \tag{C.1}$$

$$r^2 = x^2 + y^2 + z^2 \tag{C.2}$$

Calculating the gradients and Laplacian is very simply by using the `sympy.diff` function

```
 1 In [1]: from sympy import symbols, diff, exp, sqrt
 2
 3 In [2]: x, y, z, Z = symbols('x y z Z')
 4
 5 In [3]: r = sqrt(x*x + y*y + z*z)
 6
 7 In [4]: r
 8 Out[4]: '(x**2 + y**2 + z**2)**(1/2)'
 9
10 In [5]: phi = (Z*r - 2)*exp(-Z*r/2)
11
12 In [6]: phi
13 Out[6]: '(Z*(x**2 + y**2 + z**2)**(1/2) - 2)*exp(-Z*(x**2 + y**2 + z**2)**(1/2)/2)'
14
15 In [7]: diff(phi, x)
16 Out[7]: '-Z*x*(Z*(x**2 + y**2 + z**2)**(1/2) - 2)*exp(-Z*(x**2 + y**2 + z**2)**(1/2)/2)
       /(2*(x**2 + y**2 + z**2)**(1/2)) + Z*x*exp(-Z*(x**2 + y**2 + z**2)**(1/2)/2)/(x**2 +
       y**2 + z**2)**(1/2)'
```

Now, this looks like a nightmare. However, SymPy has great support for simplifying expressions through factorization, collecting, substituting etc. The following code demonstrated this quite nicely

```
 1 ...
 2
 3 In [6]: phi
 4 Out[6]: '(Z*(x**2 + y**2 + z**2)**(1/2) - 2)*exp(-Z*(x**2 + y**2 + z**2)**(1/2)/2)'
 5
 6 In [7]: from sympy import factor, Symbol, printing
 7
 8 In [8]: R = Symbol('r') #Creates a symbolic equivalent of the mathematical r
 9
10 In [9]: diff(phi, x).factor() #Factors out common factors
11 Out[9]: '-Z*x*(Z*(x**2 + y**2 + z**2)**(1/2) - 4)*exp(-Z*(x**2 + y**2 + z**2)**(1/2)/2)
       /(2*(x**2 + y**2 + z**2)**(1/2))'
12
13 In [10]: diff(phi, x).factor().subs(r, R) #replaces (x^2 + y^2 + z^2)^(1/2) with r
14 Out[10]: '-Z*x*(Z*r - 4)*exp(-Z*r/2)/(2*r)'
15
16 In [11]: print printing.latex(diff(phi, x).factor().subs(r, R))
17 - \frac{Z x \left(Z r -4\right)}{2 r e^{\frac{1}{2} Z r}}
```

This version of the expression is much more satisfying to the eye. The output from line 11 compiled in Latex is

$$- \frac{Z x \left(Z r -4\right)}{2 r e^{\frac{1}{2} Z r}}$$

SymPy has a general method for simplifying expressions `sympy.simplify`, however, this function is extremely slow and does not behave well on general expressions. SymPy is still young, so nothing can be expected to work perfectly. Moreover, in contrast to *Wolfram Alpha* and *Mathematica*, SymPy is open source, which means that much of the work, if not all of the work, is done by ordinary people on their spare time. The ill behaving simplify function is not really a great loss; full control for a Python programmer is never considered a bad thing, whether it is enforced or not.

Estimating the Laplacian is just a matter of summing double derivatives

```
...

In [12]: (diff(diff(phi, x), x) +
   ....:  diff(diff(phi, y), y) +
   ....:  diff(diff(phi, z), z)).factor().subs(r, R)
Out[12]: 'Z*(Z**2*x**2 + Z**2*y**2 + Z**2*z**2 - 10*Z*r + 16)*exp(-Z*r/2)/(4*r)'

In [13]: (diff(diff(phi, x), x) + #Not quite satisfying.
   ....:  diff(diff(phi, y), y) + #Let's collect the 'Z' terms.
   ....:  diff(diff(phi, z), z)).factor().collect(Z).subs(r, R)
Out[13]: 'Z*(Z**2*(x**2 + y**2 + z**2) - 10*Z*r + 16)*exp(-Z*r/2)/(4*r)'

In [14]: (diff(diff(phi, x), x) + #Still not satisfying.
   ....:  diff(diff(phi, y), y) + #The r^2 terms needs to be substituted as well.
   ....:  diff(diff(phi, z), z)).factor().collect(Z).subs(r, R).subs(r**2, R**2)
Out[14]: 'Z*(Z**2*r**2 - 10*Z*r + 16)*exp(-Z*r/2)/(4*r)'

In [15]: (diff(diff(phi, x), x) + #Let's try to factorize once more.
   ....:  diff(diff(phi, y), y) +
   ....:  diff(diff(phi, z), z)).factor().collect(Z).subs(r, R).subs(r**2, R**2).factor()
Out[15]: 'Z*(Z*r - 8)*(Z*r - 2)*exp(-Z*r/2)/(4*r)'
```

Getting the right factorization may come across as tricky, but with minimal training this poses no real problems.

C.2 Using the auto-generation Script

The superclass `orbitalsGenerator` aims to serve as a interface with the QMC C++ `BasisFunctions` class, automatically generating the C++ code containing all the implementations of the derivatives for the given single particle states. The single particle states are implemented in the generator by subclasses overloading system specific virtual functions which will be described in the following sections.

C.2.1 Generating Latex code

The following methods are user-implemented functions used to calculate the expressions which are in turn automagically converted to Latex code. Once they are implemented, the following code can be executed in order to create the latex output

```
orbitalSet = HO_3D.HOOrbitals3D(N=40) #Creating a 3D harm. osc. object
orbitalSet.closedFormify()
orbitalSet.TeXToFile(outPath)
```

The constructor

The superclass constructor takes on input the maximum number of particles for which expressions should be generated and the name of the orbital set, e.g. `hydrogenic`. Calling a superclass constructor from a subclass constructor is done in the following way

```
class hydrogenicOrbitals(orbitalGenerator):

    def __init__(self, M):

        super(hydrogenicOrbitals, self).__init__(M, "hydrogenic")
        #...
```

makeStateMap

This function takes care of the mapping of a set of quantum numbers, e.g. *nml* to a specific index *i*. The Python dictionary `self.stateMap` must be filled with values for every unique set of quantum numbers (not counting spin) in order for the Latex and C++ files to be created successfully. For the three-dimensional harmonic oscillator wave functions, the state map looks like this

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
n_x	0	0	0	1	0	0	0	1	1	2	0	0	0	0	1	1	1	2	2	3
n_y	0	0	1	0	0	1	2	0	1	0	0	1	2	3	0	1	2	0	1	0
n_z	0	1	0	0	2	1	0	1	0	0	3	2	1	0	2	1	0	1	0	0

setUpOrbitals

Within this function, the orbital elements corresponding to the quantum number mappings made in `makeStateMap` needs to be implemented in a matching order. The quantum numbers from `self.stateMap` are calculated prior to this function being called, and can thus be accessed in case they are needed, as is the case for the *n*-dependent exponential factor of the hydrogen-like orbitals.

The *i*'th orbital needs to be implemented in `self.orbitals[i]`, using the `x`, `y` and `z` variables defined in the superclass. For the three-dimensional harmonic oscillator, the function is simply

```python
def setupOrbitals(self):

    for i, stateMap in self.stateMap.items():
        nx, ny, nz = stateMap

        self.orbitals[i] = self.Hx[nx]*self.Hy[ny]*self.Hz[nz]*self.expFactor
```

where `self.Hx` and the exponential factor are implemented in the constructor. After the orbitals are created, the gradients and Laplacians are calculated.

simplifyLocal

As demonstrated in the previous example, SymPy expressions are messy when they are fresh out of the derivative functions. Since every system needs to be treated differently when it comes to cleaning up their expressions, this function are available. For hydrogen-like wave functions, the previous example's strategy can be applied up to the level of Neon. Going higher will require more advanced strategies for cleaning up the expressions.

The expression and the corresponding set of quantum numbers are given on input. In addition, there is an input argument `subs`, which if set to false should make the function return the expression in terms of `x`, `y` and `z` without substituting e.g. $x^2 + y^2 = r^2$.

genericFactor

A convenient function for returning generic parts of the expressions, i.e. the exponential parts. A set of quantum numbers are supplied on input in case the generic expression is dependent of these. In addition, a flag `basic` is supplied on input, which if set to true should, as in the simplify function, return the generic factor in Cartesian coordinates. This generic factor can then be taken out of the Latex expression and mentioned in the caption in order to clean up the expression tables.

__str__

This method is invoked by calling `str(obj)` on an arbitrary Python object `obj`. In the case of the orbital generator class, this string will serve as an introductory text to the latex output.

C.2.2 Generating C++ code

A class `CPPbasis` is constructed to supplement the orbitals generator class. This objects holds the raw shell of the C++ constructors and implementations. After the functions described in this section are implemented, the following code can be executed to generate the C++ files

```
orbitalSet = HO_3D.HOOrbitals3D(N=40) #Creating a 3D harm. osc. object
orbitalSet.closedFormify()
orbitalSet.TeXToFile(outPath)
orbitalSet.CPPToFile(outPath)
```

initCPPbasis

Sets up the variables in the `CPPbasis` object needed in order to construct the C++ file, such as the dimension, the name, the constructor input variables and the C++ class members. The following function is the implementation for the two-dimensional harmonic oscillator

```
def initCPPbasis(self):

    self.cppBasis.dim = 2

    self.cppBasis.setName(self.name)

    self.cppBasis.setConstVars('double* k',           #sqrt(k2)
                               'double* k2',          #scaled oscillator freq.
                               'double* exp_factor') #The exponential

    self.cppBasis.setMembers('double* k',
                             'double* k2',
                             'double* exp_factor',
                             'double H',              #The Hermite polynomial part
                             'double x',
                             'double y',
                             'double x2',             #Squared Cartesian coordinates
                             'double y2')
```

getCPre

`getCPre`	Given an expression and a its index i as input, set up the C++ pre-return calculation. E.g. `H = printing.ccode(expr/self.genericFactor(i));`
`getCreturn`	Given an expression and a its index i as input, set up the C++ return value. E.g. `return H*(*exp_factor);`
`makeOrbConstArg`	Used to set up the constructor input in the generated `Orbitals` constructor. Defaults to sending names equal to those used in the declaration, however, in e.g. the case of hydrogenic orbitals, different basis functions need different exponential factors. See `hydrogenic.py` for an example.

Implementing these functions will generate a Latex file listing the calculated expressions (see Appendix **??** and **??**), the constructor needed by the `Orbitals` subclass holding the generated orbitals and C++ header and source files containing the `BasisFunctions` implementation. In the code used in this thesis, 8975 lines of C++ code was auto-generated using SymPy (not counting blank lines).

Bibliography

[1] D. Griffiths, *Introduction to Quantum Mechanics*, 2nd ed. Pearson, 2005.

[2] G. Golub and C. Van Loan, *Matrix computations.* Johns Hopkins Univ Press, 1996, vol. 3.

[3] (2013, May) Matplotlib website. [Online]. Available: http://matplotlib.org

[4] (2013, May) PySide website. [Online]. Available: http://qt-project.org/wiki/Category:LanguageBindings::PySide

[5] (2013, May) SymPy website. [Online]. Available: http://sympy.org