Part I

Theory

1

Generalization and Optimization

There is a big difference in strategy between writing code for a specific problem, and creating a general solver. A general Quantum Monte-Carlo solver involves several layers of complexity, such as support for different potentials, single particle bases, sampling models, etc., which may easily lead to a combinatoric disaster if the planning is not done right.

This chapter will begin by covering the underlaying assumptions regarding the modeled systems in Section 1.1.1 before listing the generalization goals in Section 1.1.2 and the optimization goals in Section 1.1.3. The strategies used to reach the generalization goals based on the assumptions will then be covered in Section 1.2, followed by several optimization schemes, some of which are a direct consequence of the initial assumptions, and some of which are more general.

1.1 Underlying Assumptions and Goals

Thousands of lines of code should be written once and for all. In industrial computational projects it is custom to plan every single part of the program before the coding begins, simply due to the fact that large scale object oriented frame works demands it. Brute-force coding massive frameworks almost exclusively result in unforeseen consequences, rendering the code difficult to expand, disorganized, if not completely defect. The code used in this thesis has been completely restructured four times. This section will cover the assumptions made and the goals set in the planning stages preliminary to the coding process.

1.1.1 Assumptions

The code scheme was laid down based on the following assumptions

(i) The particles of the simulated systems are either all fermions or all bosons.

(ii) The Hamiltonian is spin - and time independent.

(iii) The trial wave function of a fermionic system is a single determinant.

(iv) A bosonic system is modeled by all particles being in the same assumed single particle ground state.

The second assumption implies that the Slater determinant can be split into parts corresponding to

different spin eigenvalues. The time-independence is a requirement on the QMC solver. These topics has been covered in Chapter **??**.

1.1.2 Generalization Goals

The implementation should be general for:

(i) Fermions and Bosons.

(ii) Anisotropic- and isotropic diffusion, i.e. Brute Force - or Importance sampling.

(iii) Different gradient descent algorithms.

(iv) Any Jastrow factor.

(v) Any error estimation algorithm.

(vi) Any single particle basis, including expanded single particle bases.

(vii) Any combination of any potentials.

In addition, the following constraint is set on solvers:

(viii) Full numerical support for all values involving derivatives.

The challenge is, despite the vast array of combinations, to preserve simplicity and structure as layers of complexity are added. If-testing inside the solvers in order to achieve generalization is considered less optimal, and is only used if no other solution is apparent/exists.

1.1.3 Optimization Goals

Designed for the CPU, runtime optimizations are favored over memory optimizations. The following list may appear short, but every step brings immense amounts of complexity to the implementation

(i) Identical values should never be re-calculated.

(ii) Generalization should not be achieved through if-tests in repeated function calls, but rather through polymorphism (see Section **??**).

(iii) Linear scaling of runtime vs. the number of CPUs for large simulations.

1.2 Specifics Regarding Generalization

Generalization is achieved through the use of deep object orientation, i.e. polymorphism, rather than if-testing. These concepts are described in Section **??**. The assumptions listed in Section 1.1.1 are applied if not otherwise is stated.

For details regarding the implementation of methods, see the code in [1].

1.2.1 Generalization Goals (i)-(vii)

As discussed in Section **??**, the mathematical difference between fermions and bosons (of importance to QMC) is how the many-body wave functions are constructed from the single-particle basis. In the case of fermions, the expression is given in terms of two Slater determinants, while for bosons, it is simply the product of all states.

```
double Fermions::get_spatial_wf(const Walker* walker) {
    using namespace arma;

    //Spin up times Spin down (determinants)
    return det(walker->phi(span(0, n2 - 1), span())) * det(walker->phi(span(n2, n_p - 1),
         span()));
}
```

```
double Bosons::get_spatial_wf(const Walker* walker) {

    double wf = 1;

    //Using the phi matrix as a vector in the case of bosons.
    //Assuming all particles to occupy the same single particle state (neglecting
        permutations).
    for (int i = 0; i < n_p; i++){
      wf *= walker->phi(i);
    }

    return wf;
}
```

Fermion/Boson overloaded pure virtual methods exist for all methods involving evaluation of the many body wave function, e.g. the spatial ratio and the Laplacian sum. When the `QMC` solver asks the `System*` object for a spatial ratio, depending on whether Fermions or Bosons are loaded run-time, the Fermion or Boson spatial ratio is evaluated.

This way of splitting the system class also takes care of optimization goal (ii) in Section 1.1.3 regarding no use of repeating if-tests.

Similar polymorphic splitting is introduced in the following classes:

- `Orbitals` Hydrogen orbitals, harmonic oscillator, etc.
- `BasisFunctions` Stand-alone single particle wave functions. Initialized by `Orbitals`.
- `Sampling` Brute force - or importance sampling.
- `Diffusion` Isotropic or Fokker-Planck. Automatically selected by `Sampling`.
- `ErrorEstimator` Simple or Blocking.
- `Jastrow` Padé Jastrow - or no Jastrow factor.
- `QMC` VMC or DMC.
- `Minimizer` ASGD. Support for adding additional minimizers.

Implementing e.g. a new Jastrow Factor is done by simply creating a new subclass of `Jastrow`. The QMC solver does not need to change to adapt to the new implementation. For more details, see Section **??**. The splitting done in `QMC` is done to avoid rewriting a lot of general QMC code.

A detailed description of the generalization of potentials, i.e. generalization goal (vii), is given in Section **??**.

1.2.2 Generalization Goal (vi) and Expanded bases

An expanded single particle basis is implemented as a subclass of the `Orbitals` superclass. It is designed as a wrapper to an `Orbitals` subclass, e.g. harmonic oscillator, containing basis elements $\phi_\alpha(r_j)$. In addition to these elements, the class has a set of expansion coefficients $C_{\gamma\alpha}$ in which the new basis elements are constructed

$$\psi_\gamma^{\text{Exp.}}(r_j) = \sum_{\alpha=0}^{B-1} C_{\gamma\alpha}\phi_\alpha(r_j) \tag{1.1}$$

where B is the size of the expanded basis.

```
class ExpandedBasis : public Orbitals {

...

protected:

    int basis_size;
    arma::mat coeffs;
    Orbitals* basis;

    //Hartree-Fock
    void calculate_coefficients();

};
```

In order to calculate the expansion coefficients, a *Hartree-Fock* solver has been implemented. For a given single particle basis loaded in the `Orbitals* basis` member, everything that is needed in order to obtain an expanded basis, i.e. calculate the coefficients, is expressions for the one-body - and two-body interaction elements.

The implementation of Eq. (1.1) into the expanded basis class is achieved by overloading the original `Orbitals::phi` virtual member function

```
double ExpandedBasis::phi(const Walker* walker, int particle, int q_num) {

    double value = 0;

    //Dividing basis_size by half assuming a two-level system.
    for (int m = 0; m < basis_size/2; m++) {
        value += coeffs(q_num, m) * basis->phi(walker, particle, m);
    }

    return value;

}
```

The Hartree-Fock implementation at the present time is bugged, and has not been a focus for the thesis. The implementation has merely been done to lay the foundation in case future Master students are to expand upon the code. Hence Hartree-Fock theory will not be covered in detail, however, introductory theory can be found in Refs. [2, 3].

1.2.3 Generalization Goal (viii)

Functionality for evaluating derivatives numerically is important for two reasons; the first being debugging, the second being the cases where no closed-form expressions for the derivatives can be obtained or become too expensive to evaluate.

As an example, `Orbitals::dell_phi`, which returns a single particle derivative, is virtual, and can be overloaded to call the numerical derivative implementation `Orbitals::num_diff`.The same goes for the Jastrow factor and the variational derivatives in the minimizers. Similar implementations exist for the Laplacian. An alternative to numerically evaluate the single particle wave function derivatives would be to perform the derivative on the full many-body wave function, however, this would demand another layer of polymorphism and make the code all-around less intuitive.

The implemented numerical derivatives are finite difference schemes with error proportional to the squared step length.

1.3 Optimizations due to a Single two-level Determinant

Assumption (iii) unlocks the possibility to optimize the expressions involving the Slater-determinant dramatically. Similar optimizations for bosons due to assumption (iv) are considered trivial and will not be covered in detail. See the code in [1] for details regarding bosons.

The full trial wave function Ψ_T is given by Eq. (**??**). The function arguments will be skipped in order to clean up the expressions. Written in terms of a spatial function $|D|$, which is split into a spin up -and spin down part by Eq. (**??**), and a Jastrow function J, the trial wave function reads

$$\Psi_T = |D^{\uparrow}||D^{\downarrow}|J \tag{1.2}$$

The Quantum Force becomes

$$\begin{aligned}
\mathbf{F}_i &= 2\frac{\nabla_i \left(|D^\uparrow||D^\downarrow|J\right)}{|D^\uparrow||D^\downarrow|J} \\
&= 2\left(\frac{\nabla_i|D^\uparrow|}{|D^\uparrow|} + \frac{\nabla_i|D^\downarrow|}{|D^\downarrow|} + \frac{\nabla_i J}{J}\right) \\
&= 2\left(\frac{\nabla_i|D^\alpha|}{|D^\alpha|} + \frac{\nabla_i J}{J}\right) \qquad (1.3)
\end{aligned}$$

where α is the spin configuration of particle i. The counterpart to α is independent of particle i and will be zero in the expression above.

Equally for the Laplacian used in the local energy, the result becomes

$$E_L = \sum_i \frac{1}{\Psi_T}\nabla_i^2\Psi_T + V \qquad (1.4)$$

$$\begin{aligned}
\frac{1}{\Psi_T}\nabla_i^2\Psi_T &= \frac{1}{|D^\uparrow||D^\downarrow|J}\nabla_i^2|D^\uparrow||D^\downarrow|J \\
&= \frac{\nabla_i^2|D^\uparrow|}{|D^\uparrow|} + \frac{\nabla_i^2|D^\downarrow|}{|D^\downarrow|} + \frac{\nabla_i^2 J}{J} \\
&\quad + 2\frac{\left(\nabla_i|D^\uparrow|\right)\left(\nabla_i|D^\downarrow|\right)}{|D^\uparrow||D^\downarrow|} + 2\frac{\left(\nabla_i|D^\uparrow|\right)\left(\nabla_i J\right)}{|D^\uparrow|J} + 2\frac{\left(\nabla_i|D^\downarrow|\right)\left(\nabla_i J\right)}{|D^\downarrow|J} \\
&= \frac{\nabla_i^2|D^\alpha|}{|D^\alpha|} + \frac{\nabla_i^2 J}{J} + 2\frac{\nabla_i|D^\alpha|}{|D^\alpha|}\frac{\nabla_i J}{J} \qquad (1.5)
\end{aligned}$$

Finally, the expression for the R_ψ ratio for Metropolis is

$$\begin{aligned}
R_\psi &= \frac{\Psi_T^{\text{new}}}{\Psi_T^{\text{old}}} \\
&= \frac{|D^\uparrow|^{\text{new}}|D^\downarrow|^{\text{new}}J^{\text{new}}}{|D^\uparrow|^{\text{old}}|D^\downarrow|^{\text{old}}J^{\text{old}}} \\
&= \frac{|D^\alpha|^{\text{new}}}{|D^\alpha|^{\text{old}}}\frac{J^{\text{new}}}{J^{\text{old}}} \qquad (1.6)
\end{aligned}$$

where either the spin up or the spin down determinant is unchanged by the step, i.e. $|D^{\overline{\alpha}}|^{\text{new}} = |D^{\overline{\alpha}}|^{\text{old}}$, where $\overline{\alpha}$ denotes the opposite spin of α.

From these expressions it is clear that the dimensionality of the calculations is halved by splitting the Slater determinants. Calculation the determinant of a $N \times N$ matrix is $\mathcal{O}(N^2)$ floating point operations (flops). This implies a speedup of four in estimating the determinants alone.

1.4 Optimizations due to Single-particle Moves

Moving one particle at the time (see the diffusion algorithm in Fig. **??**), means changing only a single row in the Slater-determinant at the time. Changes to a single row implies that many *co-factors* remain unchanged. Since all of the expressions deduced in the previous section contains ratios of the spatial wave functions, expressing these determinants in terms of their co-factors should reveal a cancellation of terms.

1.4.1 Optimizing the Slater ratios

The inverse of the Slater-matrix is given in terms of its *adjugate* by the following relation [4] (spin-configuration parameter α will be skipped for now)

$$D^{-1} = \frac{1}{|D|}\text{adj}D$$

The adjugate of a matrix is the transpose of the cofactor matrix C. The expression in terms of matrix element equations reads

$$\begin{aligned} D^{-1}_{ij} &= \frac{C_{ji}}{|D|} && (1.7)\\ D_{ji} &= \phi_i(r_j) && (1.8) \end{aligned}$$

Moreover, the determinant can be expressed as a *cofactor expansion* around row j (Kramer's rule)

$$|D| = \sum_i D_{ji}C_{ji}. \tag{1.9}$$

The spatial part of the R_ψ ratio is obtained by inserting Eq. (1.9) into Eq. (1.6)

$$R_S = \frac{\sum_i D^{\text{new}}_{ji}C^{\text{new}}_{ji}}{\sum_i D^{\text{old}}_{ji}C^{\text{old}}_{ji}} \tag{1.10}$$

Let j represent the moved particle. The j'th column of the cofactor matrix is unchanged when the particle moves (column j depends on every column but its own). In other words

$$C^{\text{new}}_{ji} = C^{\text{old}}_{ji} = (D^{\text{old}}_{ij})^{-1}|D^{\text{old}}|, \tag{1.11}$$

where the inverse relation of Eq. (1.7) has been used. Inserting this into Eq. (1.10) yields

$$\begin{aligned} R_S &= \frac{|D^{\text{old}}|}{|D^{\text{old}}|}\frac{\sum_i D^{\text{new}}_{ji}(D^{\text{old}}_{ij})^{-1}}{\sum_i D^{\text{old}}_{ji}(D^{\text{old}}_{ij})^{-1}}\\ &= \frac{\sum_i D^{\text{new}}_{ji}(D^{\text{old}}_{ij})^{-1}}{I_{jj}} \end{aligned}$$

The diagonal element of the identity matrix is by definition unity. Inserting this fact combined with the relation from Eq. (1.8) yields the optimized expression for the ratio

$$R_S = \sum_i \phi_i(r^{\text{new}}_j)(D^{\text{old}}_{ij})^{-1} \tag{}$$

where j is the currently moved particle. The sum i spans the Slater matrix whose spin value matches that of particle j.

Similar reductions can be applied to all the Slater ratio expressions from the previous section, see refs. [5, 2]:

$$\frac{\nabla_i |D|}{|D|} = \sum_k \nabla_i \phi_k(r_i^{\text{new}})(D_{ki}^{\text{new}})^{-1} \tag{1.13}$$

$$\frac{\nabla_i^2 |D|}{|D|} = \sum_k \nabla_i^2 \phi_k(r_i^{\text{new}})(D_{ki}^{\text{new}})^{-1} \tag{1.14}$$

where k spans the Slater matrix whose spin values match that of the moved particle. Unlike for the ratio, $N/2$ of the gradients needs to be recalculated once a particle is moved (with N being the number of particles). This is due to the fact that it is the new Slater inverse that is used, and not the old.

Closed form expressions for the derivatives and Laplacians of the single particle may be implemented and accessed when calling these functions to avoid expensive numerical calculations. See Appendices D, E and F for a tabulation of closed form expressions. Appendix C presents an efficient framework for obtaining these expressions.

1.4.2 Optimizing the Inverse

One might question the efficiency of calculating inverse matrices compared to brute force estimation of the determinants. However, as for the ratio in Eq. (1.12), using co-factor expansions, an updating algorithm which dramatically decreases the cost of calculating the inverse of the new Slater matrix can be implemented.

Letting i denote the currently moved particle, the new inverse is given in terms of the old by the following expression [5, 2]

$$\tilde{I}_{ij} = \sum_l D_{il}^{\text{new}} (D_{lj}^{\text{old}})^{-1} \tag{1.15}$$

$$(D_{kj}^{\text{new}})^{-1} = (D_{kj}^{\text{old}})^{-1} - \frac{1}{R_S}(D_{ji}^{\text{old}})^{-1} \tilde{I}_{ij} \qquad j \neq i \tag{1.16}$$

$$(D_{ki}^{\text{new}})^{-1} = \frac{1}{R_S}(D_{ki}^{\text{old}})^{-1} \qquad \text{else} \tag{1.17}$$

This reduces the cost of calculating the inverse by an order of magnitude down to $\mathcal{O}(N^2)$.

Further optimization can be achieved by calculating the $\tilde{I}$ vector for particle i prior to performing the loop over k and j. Again, this loop should only update the inverse Slater matrix whose spin value correspond to that of the moved particle.

1.4.3 Optimizing the Padé Jastrow factor Ratio

As done with the Green's function ratio in Eq. (**??**), the ratio between two Jastrow factors are best calculating as exponentiation the logarithm

$$\log \frac{J^{\text{new}}}{J^{\text{old}}} = \sum_{k<j=1}^{N} \frac{a_{kj} r_{kj}^{\text{new}}}{1+\beta r_{kj}^{\text{new}}} - \frac{a_{kj} r_{kj}^{\text{old}}}{1+\beta r_{kj}^{\text{old}}} \tag{1.18}$$

$$\equiv \sum_{k<j=1}^{N} g_{kj}^{\text{new}} - g_{kj}^{\text{old}} \tag{1.19}$$

The relative distances r_{kj} behave much like the cofactors in Section 1.4.1: Changing r_i only changes r_{ij}, that is

$$r_{kj}^{\text{new}} = r_{kj}^{\text{old}} \qquad k \neq i \tag{1.20}$$

which inserted into Eq. (1.19) yields

$$\begin{aligned} \log \frac{J^{\text{new}}}{J^{\text{old}}} &= \sum_{k<j\neq i} g_{kj}^{\text{old}} - g_{kj}^{\text{old}} + \sum_{j=1}^{N} g_{ij}^{\text{new}} - g_{ij}^{\text{old}} \\ &= \sum_{j=1}^{N} a_{ij} \left(\frac{r_{ij}^{\text{new}}}{1+\beta r_{ij}^{\text{new}}} - \frac{r_{ij}^{\text{old}}}{1+\beta r_{ij}^{\text{old}}} \right) \end{aligned} \tag{1.21}$$

Exponentiating both sides reveals the final optimized ratio

$$\frac{J^{\text{new}}}{J^{\text{old}}} = \exp\left[\sum_{j=1}^{N} a_{ij} \left(\frac{r_{ij}^{\text{new}}}{1+\beta r_{ij}^{\text{new}}} - \frac{r_{ij}^{\text{old}}}{1+\beta r_{ij}^{\text{old}}} \right) \right] \tag{}$$

1.5 Optimizing the Padé Jastrow Derivative Ratios

The shape of the Padé Jastrow factor is general in the sense that its shape is independent of the system at hand. Calculating closed form expressions for the derivatives is then a process which can be done once and for all.

1.5.1 The Gradient

Using the notation of Eq. (1.19), the x component of the Padé Jastrow gradient ratio for particle i is

$$\frac{1}{J}\frac{\partial J}{\partial x_i} = \frac{1}{\prod_{k<l} \exp g_{kl}} \frac{\partial}{\partial x_i} \prod_{k<l} \exp g_{kl} \tag{1.23}$$

Using the product rule, only terms with k or l equal to i survive the differentiation. In addition, the terms independent of i will cancel the corresponding terms in the denominator. In other words,

$$\begin{aligned}\frac{1}{J}\frac{\partial J}{\partial x_i} &= \sum_{k\neq i}\frac{1}{\exp g_{ik}}\frac{\partial}{\partial x_i}\exp g_{ik} \\ &= \sum_{k\neq i}\frac{1}{\exp g_{ik}}\exp g_{ik}\frac{\partial g_{ik}}{\partial x_i} \\ &= \sum_{k\neq i}\frac{\partial g_{ik}}{\partial x_i} \\ &= \sum_{k\neq i}\frac{\partial g_{ik}}{\partial r_{ik}}\frac{\partial r_{ik}}{\partial x_i},\end{aligned} \tag{1.24}$$

where

$$\begin{aligned}\frac{\partial g_{ik}}{\partial r_{ik}} &= \frac{\partial}{\partial r_{ik}}\left(\frac{a_{ik}r_{ik}}{1+\beta r_{ik}}\right) \\ &= \frac{a_{ik}}{1+\beta r_{ik}} - \frac{a_{ik}r_{ik}}{(1+\beta r_{ik})^2}\beta \\ &= \frac{a_{ik}(1+\beta r_{ik}) - a_{ik}\beta r_{ik}}{(1+\beta r_{ik})^2} \\ &= \frac{a_{ik}}{(1+\beta r_{ik})^2},\end{aligned} \tag{1.25}$$

and

$$\begin{aligned}\frac{\partial r_{ik}}{\partial x_i} &= \frac{\partial}{\partial x_i}\sqrt{(x_i-x_k)^2+(y_i-y_k)^2+(z_i-z_k)^2} \\ &= \frac{1}{2}2(x_i-x_k)/\sqrt{(x_i-x_k)^2+(y_i-y_k)^2+(z_i-z_k)^2} \\ &= \frac{x_i-x_k}{r_{ik}}.\end{aligned} \tag{1.26}$$

Combining these expressions yields

$$\frac{1}{J}\frac{\partial J}{\partial x_i} = \sum_{k\neq i}\frac{a_{ik}}{r_{ik}}\frac{x_i-x_k}{(1+\beta r_{ik})^2} \tag{1.27}$$

When changing Cartesian variable in the differentiation, the only change to the expression is that the corresponding Cartesian variable changes in the numerator of Eq. (1.27). In other words, generalizing to the full gradient is done by substituting the Cartesian difference with the position vector difference.

$$\frac{\nabla_i J}{J} = \sum_{k\neq i=1}^{N}\frac{a_{ik}}{r_{ik}}\frac{\vec{r}_i-\vec{r}_k}{(1+\beta r_{ik})^2} \tag{1.28}$$

1.5.2 The Laplacian

The same strategy used to obtain the closed form expression for the gradient in the previous section can be applied to the Laplacian. The full calculation is done in ref. [2]. The expression becomes

$$\frac{\nabla_i^2 J}{J} = \left|\frac{\nabla_i J}{J}\right|^2 + \sum_{k\neq i=1}^{N}\left(\frac{d-1}{r_{ik}}\frac{\partial g_{ik}}{\partial r_{ik}} + \frac{\partial^2 g_{ik}}{\partial r_{ik}^2}\right) \tag{1.29}$$

where d is the number of dimensions, arising due to the fact that the Laplacian, unlike the gradient, is a summation of contributions from all dimensions. A simple differentiation of Eq. (1.25) with respect to r_{ik} yields

$$\frac{\partial^2 g_{ik}}{\partial r_{ik}^2} = -\frac{2a_{ik}\beta}{(1+\beta r_{ik})^3} \tag{1.30}$$

Inserting Eq. (1.25) and Eq. (1.30) into Eq. (1.29) yields

$$\begin{aligned}\frac{\nabla_i^2 J}{J} &= \left|\frac{\nabla_i J}{J}\right|^2 + \sum_{k\neq i=1}^{N}\left(\frac{d-1}{r_{ik}}\frac{a_{ik}}{(1+\beta r_{ik})^2} - \frac{2a_{ik}\beta}{(1+\beta r_{ik})^3}\right)\\ &= \left|\frac{\nabla_i J}{J}\right|^2 + \sum_{k\neq i=1}^{N} a_{ik}\frac{(d-1)(1+\beta r_{ik}) - 2\beta r_{ik}}{r_{ik}(1+\beta r_{ik})^3}\end{aligned}$$

which when cleaned up results in

$$\frac{\nabla_i^2 J}{J} = \left|\frac{\nabla_i J}{J}\right|^2 + \sum_{k\neq i} a_{ik}\frac{(d-3)(\beta r_{ik}+1)+2}{r_{ik}(1+\beta r_{ik})^3} \tag{1.31}$$

The local energy calculation needs the sum of the Laplacians for all particles (see Eq. (1.4)). In other words, the quantity of interest becomes

$$\sum_i \frac{\nabla_i^2 J}{J} = \sum_i \left[\left|\frac{\nabla_i J}{J}\right|^2 + \sum_{k\neq i}^{N} a_{ik}\frac{(d-3)(\beta r_{ik}+1)+2}{r_{ik}(1+\beta r_{ik})^3}\right] \tag{1.32}$$

Due to the symmetry of r_{ik}, the second term count equal values twice. Further optimization can thus be achieved by calculation only terms where $k > i$, and multiply the sum by two. Bringing it all together yields

$$\sum_i \frac{\nabla_i^2 J}{J} = \sum_i \left|\frac{\nabla_i J}{J}\right|^2 + 2\sum_{k>i} a_{ik}\frac{(d-3)(\beta r_{ik}+1)+2}{r_{ik}(1+\beta r_{ik})^3} \tag{}$$

1.6 Tabulating Recalculated Data

The optimizations covered in this section will exclusively arise from point (i) in section 1.1.3. Avoiding recalculating expressions also include exploiting symmetries, such as was done in the Padé Jastrow Laplacian in Eq. (1.33).

Code examples are presented to narrow the gap between presented optimizations and practical implementations in order to demonstrate that most optimizations does not require much additional code.

1.6.1 The relative distance matrix

In the discussions regarding the optimization of the Jastrow ratio, it became clear that moving one particle only changed N of the relative distances. Storing these values in a matrix r_{rel}, the row and column representing the moved particle can be updated, ensuring that the relative distances are calculated once and for all.

$$r_{\text{rel}} = r_{\text{rel}}^T = \begin{pmatrix} 0 & r_{12} & r_{13} & \cdots & r_{1N} \\ & 0 & r_{23} & \cdots & r_{2N} \\ & & \ddots & \ddots & \vdots \\ & \cdots & & 0 & r_{(N-1)N} \\ & & & & 0 \end{pmatrix}. \tag{1.34}$$

```
1  void Sampling::update_pos(const Walker* walker_pre, Walker* walker_post, int particle)
       const {
2
3      ...
4
5      //Updating the part of the r_rel matrix which is changed by moving the [particle]
6      for (int j = 0; j < n_p; j++) {
7          if (j != particle) {
8              walker_post->r_rel(particle, j) = walker_post->r_rel(j, particle)
9                      = walker_post->calc_r_rel(particle, j);
10         }
11     }
12
13     ...
14
15 }
```

Functions such as `Coulomb::get_potential_energy` and all of the Jastrow functions can then simply access these matrix elements.

Similar storage has been done for the squared distance vector in all cases and the length of the distance vector in case of atomic orbitals.

1.6.2 The Slater related matrices

Apart from the inverse, whose optimization was covered in Section 1.4.2, calculating the single particle wave functions and its gradients are the most expensive operations of the QMC algorithm.

Storing these function values in a matrices representing the Slater Matrix and its derivatives will ensure that these values never gets recalculated.

$$D \equiv \left[D^{\uparrow}\, D^{\downarrow}\right] = \begin{bmatrix} \phi_1(r_0) & \phi_1(r_1) & \cdots & \phi_1(r_N) \\ \phi_2(r_0) & \phi_2(r_1) & \cdots & \phi_2(r_N) \\ \vdots & \vdots & & \vdots \\ \phi_{N/2}(r_0) & \phi_{N/2}(r_1) & \cdots & \phi_{N/2}(r_N) \end{bmatrix} \tag{1.35}$$

$$\nabla D \equiv \left[\nabla D^{\uparrow}\, \nabla D^{\downarrow}\right] = \begin{bmatrix} \nabla\phi_1(r_0) & \nabla\phi_1(r_1) & \cdots & \nabla\phi_1(r_N) \\ \nabla\phi_2(r_0) & \nabla\phi_2(r_1) & \cdots & \nabla\phi_2(r_N) \\ \vdots & \vdots & & \vdots \\ \nabla\phi_{N/2}(r_0) & \nabla\phi_{N/2}(r_1) & \cdots & \nabla\phi_{N/2}(r_N) \end{bmatrix} \tag{1.36}$$

1.6.3 Avoiding spin tests

Since the slater determinant is split by spin eigenvalues, the same splitting occurs in the inverse, the Slater matrix etc. The brute force implementation is to perform an if-test to decide which of the matrices to access. In the case of a two-level system we get

$$\begin{array}{ll} i < N/2 & D^{\uparrow}(i) \\ i \geq N/2 & D^{\downarrow}(i - N/2) \end{array} \tag{1.37}$$

However, simply concatenating the Slater related matrices solves the entire problem. This has already been done in Eq. (1.35) and Eq. (1.36). Applying this to the inverse yields

$$D^{-1} \equiv \left[(D^{\uparrow})^{-1} \, (D^{\downarrow})^{-1} \right] \tag{1.38}$$

When the index representing the moved particle succeeds $N/2 - 1$, the values representing the opposite spin is automatically accessed, which means that no if-tests is required in order to sort the spin splitting. In e.g. the updating algorithm for the inverse, simply keeping track of whether to start at $k = 0$ or $k = N/2$ solves the problem. This "start" parameter can be calculated once every particle move, and can then be used throughout the program.

1.6.4 The Padé Jastrow gradient

Consider Eq. (1.28). Just as for the Jastrow Laplacian, there are (anti)symmetries in the expression, which implies an optimized way of calculating the gradient. However, unlike the Laplacian, the gradient is split into components, which makes the exploitation of symmetries a little less straight-forward.

Defining

$$\mathrm{d}\mathbf{J}_{ik} \equiv \frac{a_{ik}}{r_{ik}} \frac{\vec{r}_i - \vec{r}_k}{(1 + \beta r_{ik})^2} = -\mathrm{d}\mathbf{J}_{ki}, \tag{1.39}$$

the gradient can be written in a more compact form

$$\frac{\nabla_i J}{J} = \sum_{k \neq i = 1}^{N} \mathrm{d}\mathbf{J}_{ik}. \tag{1.40}$$

As for the relative distances, storing the elements and exploiting the symmetry properties, only half the total elements needs to be calculated.

$$\mathrm{d}J \equiv \begin{pmatrix} 0 & \mathrm{d}\mathbf{J}_{12} & \mathrm{d}\mathbf{J}_{13} & \cdots & \mathrm{d}\mathbf{J}_{1N} \\ & 0 & \mathrm{d}\mathbf{J}_{23} & \cdots & \mathrm{d}\mathbf{J}_{2N} \\ & & \ddots & \ddots & \vdots \\ & (-) & & 0 & \mathrm{d}\mathbf{J}_{(N-1)N} \\ & & & & 0 \end{pmatrix} = -\mathrm{d}J^T. \tag{1.41}$$

```
void Pade_Jastrow::get_dJ_matrix(Walker* walker, int i) const {

    for (int j = 0; j < n_p; j++) {
        if (j == i) continue;

        b_ij = 1.0 + beta * walker->r_rel(i, j);
        factor = a(i, j) / (walker->r_rel(i, j) * b_ij * b_ij);
        for (int k = 0; k < dim; k++) {
            walker->dJ(i, j, k) = (walker->r(i, k) - walker->r(j, k)) * factor;
            walker->dJ(j, i, k) = -walker->dJ(i, j, k);
        }
    }
}
```

Calculating the gradient is now only a matter of summing the rows of the matrix in Eq. (1.41). Further optimization can be achieved by realizing that the function has access to the gradient of the previous iteration

$$\frac{\nabla_i J^{\text{old}}}{J^{\text{old}}} = \sum_{k\neq i=1}^{N} \mathrm{d}\mathbf{J}_{ik}^{\text{old}} \tag{1.42}$$

$$\frac{\nabla_i J^{\text{new}}}{J^{\text{new}}} = \sum_{k\neq i=1}^{N} \mathrm{d}\mathbf{J}_{ik}^{\text{new}} \tag{1.43}$$

By moving particle p in QMC, only a single row and column of the $\mathrm{d}J$ matrix changes. Assuming that $i \neq p$, only a single term from the new matrix is required

$$\frac{\nabla_{i\neq p} J^{\text{new}}}{J^{\text{new}}} = \sum_{k\neq i\neq p} \mathrm{d}\mathbf{J}_{ik}^{\text{old}} + \mathrm{d}\mathbf{J}_{ip}^{\text{new}} \tag{1.44}$$

$$= \left[\sum_{k\neq i\neq p} \mathrm{d}\mathbf{J}_{ik}^{\text{old}} + \mathrm{d}\mathbf{J}_{ip}^{\text{old}}\right] - \mathrm{d}\mathbf{J}_{ip}^{\text{old}} + \mathrm{d}\mathbf{J}_{ip}^{\text{new}} \tag{1.45}$$

$$= \frac{\nabla_i J^{\text{old}}}{J^{\text{old}}} - \mathrm{d}\mathbf{J}_{ip}^{\text{old}} + \mathrm{d}\mathbf{J}_{ip}^{\text{new}} \tag{1.46}$$

reducing the calculation to three flops. For the case $i = p$, the entire sum as in Eq. (1.43) must be calculated. This is demonstrated in the following code

```
void Pade_Jastrow::get_grad(const Walker* walker_pre, Walker* walker_post, int p) const {
    double sum;

    for (int i = 0; i < n_p; i++) {
        if (i == p) {

            //for i == p the entire sum needs to be calculated
            for (int k = 0; k < dim; k++) {

                sum = 0;
                for (int j = 0; j < n_p; j++) {
                    sum += walker_post->dJ(p, j, k);
                }

                walker_post->jast_grad(p, k) = sum;
            }

        } else {
```

```
19
20              //for i != p only one term differ from the old and the new matrix
21              for (int k = 0; k < dim; k++) {
22                  walker_post->jast_grad(i, k) = walker_pre->jast_grad(i, k)
23                          + walker_post->dJ(i, p, k) - walker_pre->dJ(i, p, k);
24              }
25          }
26      }
27  }
```

Due to dJ being a 3-dimensional matrix (tensor), this optimization scales very well with increasing number of particles.

1.6.5 The single-particle Wave Functions

In order to evaluate a specific single particle wave function, a quantum number q and a position in space r is needed. A QMC walker holds the position of all particles, so in addition, the particle number i needs to be supplied. In the case of the gradient of the single particle wave functions, the dimension parameter representing x, y or z also needs to be supplied.

For systems of many particles, the function call `Orbitals::phi(walker, i, qnum)` needs to figure out which expression is related to which quantum number. The brute force implementation is to simply if-test on the quantum number, and calculate the corresponding expression inside the correct if-test.

```
1  double AlphaHarmonicOscillator::phi(const Walker* walker, int particle, int q_num) {
2
3      //Ground state of the harmonic oscillator
4      if (q_num == 0){
5        return exp(-0.5*w*walker->get_r_i2(i));
6      }
7
8      ...
9
10 }
```

This is an inefficient solution when the number of particles are big. A more efficient implementation is to implement the single particle wave function expressions as `BasisFunctions` subclasses, which holds only one pure virtual member function `BasisFunctions::eval()` which takes on input the particle number i and the walker. The class itself is defined by the quantum number q.

The following is an example of the 2D harmonic oscillator single particle wave function for quantum number $q = 1$ (see Appendix D for ϕ_1)

```
1  double HarmonicOscillator_1::eval(const Walker* walker, int i) {
2
3      y = walker->r(i, 1);
4
5      //y*exp(-k^2*r^2/2)
6
7      H = y;
8      return H*exp(-0.5*w*walker->get_r_i2(i));
9
10 }
```

These objects representing single particle wave functions can be loaded into an array in such a way that element q corresponds to the `BasisFunctions` object representing this quantum number, e.g. `basis_functions[1] = new HarmonicOscillator_1()`. The new `Orbitals` function then becomes an in-lined call to an array instead of an endless stream of if-tests:

```
double Orbitals::phi(const Walker* walker, int particle, int q_num) {
    return basis_functions[q_num]->eval(walker, particle);
}
```

All discussed optimizations thus far have been general in the sense that they do not depend on the shape of the single particle wave functions. There should, however, be room for optimizations in the basis functions, as long as these are applied locally within each class where the explicit shape of the orbitals are absolute.

As discussed previously, only a single column in the Slater related matrices from Section 1.6.2 needs to be updated when a particle is moved. This implies that terms which are independent of the quantum number can be calculated once for every particle instead of once for every quantum number. These terms often come in shape of exponentials, and are thus conserved though derivatives, implying that these terms are not only shared between quantum numbers locally within each Slater matrix, but also shared between the matrices.

Looking at the single particle states for harmonic oscillator and hydrogen listed in Appendix D - F, the exponential factor is indeed independent of the quantum number in all the terms. Referring to the quantum number independent terms as $\overline{Q}_i$, the expressions are

$$\overline{Q}_i^{\text{H.O.}} = e^{-\frac{1}{2}\alpha\omega r_i^2} \tag{1.47}$$

$$\overline{Q}_i^{\text{Hyd.}} = e^{-\frac{1}{n}\alpha Z r_i} \tag{1.48}$$

For hydrogen, there is a dependence on the principle quantum number n, however, for e.g $n = 2$, 20 terms share this exponential factor. Calculating it once and for all saves 19 exponential calls pr. particle pr. walker pr. cycle etc. resulting in a dramatic speedup.

The implementation is very simple; upon moving a particle, the virtual function `Orbitals::set_qnum_indie_terms` is called, updating the value of e.g. an `exp_factor` pointer shared by all the loaded `BasisFunctions` objects and the `Orbitals` class:

```
void AlphaHarmonicOscillator::set_qnum_indie_terms(const Walker * walker, int i) {

   //k2 = alpha*omega
   *exp_factor = exp(-0.5 * (*k2) * walker->get_r_i2(i));
}

void hydrogenicOrbitals::set_qnum_indie_terms(Walker* walker, int i) {

    //waler::calc_r_i() calculates |r_i| such that walker::get_r_i() can be used
    walker->calc_r_i(i);

    //k = alpha*Z
    double kr = -(*k) * walker->get_r_i(i);

    //Calculates only the exponentials needed based on the number of particles
    *exp_factor_n1 = exp(kr);
    if (n_p > 2) *exp_factor_n2 = exp(kr / 2);
    if (n_p > 10) *exp_factor_n3 = exp(kr / 3);
    if (n_p > 28) *exp_factor_n4 = exp(kr / 4);

}
```

The `BasisFunctions` objects share the pointer to the correct exponential factor with the orbital class. These exponential factors can then simply be accessed instead of being recalculated numerous times

```
double HarmonicOscillator_1::eval(const Walker* walker, int i) {

    y = walker->r(i, 1);

    //y*exp(-k^2*r^2/2)

    H = y;
    return H*(*exp_factor);

}

double lapl_hydrogenic_0::eval(const Walker* walker, int i) {

    //k*(k*r - 2)*exp(-k*r)/r

    psi = (*k)*((*k)*walker->get_r_i(i) - 2)/walker->get_r_i(i);
    return psi*(*exp_factor);

}
```

As will be presented in the result section, this optimization represents an enormous speedup for many-particle simulations.

For Quantum Dots, 112 `BasisFunctions` objects are needed for a 56-particle simulation Applying the currently discussed optimization reduces the number of exponential calls from 84 to 1 pr. particle pr. walker pr. cycle etc., which for an average DMC calculation results in $6 \cdot 10^1 1$ saved exponential calls. Generally, the number of exponential calls are reduced by a factor $\frac{N}{2}(N+1)$.

1.7 CPU Cache Optimization

The *CPU cache* is a limited amount of memory directly connected to the CPU, designed to reduce the average time to access memory. Simply speaking, standard memory is slower than the CPU cache, as bits have to travel through the motherboard before it can be fed to the CPU (a so called *bus*).

Which values are held in the CPU cache is controlled by the compiler, however, if programmed poorly, the compiler will not be able to handle the cache storage optimally. Optimization tools such as `O3` exist in order to work around this, however, keeping the cache in mind from the beginning of the coding process may result in a much faster code. In the case of the QMC code, the most optimal use of the cache would be to have all the active walkers in the cache at all times.

The memory is sent to the cache as arrays, which means that storing walker data sequentially in memory is the way to go in order to make take full use of the processor cache. If objects are declared as pointers, which is the case when declaring matrices of general sizes, the memory layout is uncontrollable. This fact renders a QMC solver for a general number of particles hard to optimize with respect to the cache.

The alternative is to statically declare the matrices, i.e. declare matrices with a fixed size known at compile time. An apparent work-around would be to declare the matrices to the maximum possible simulation size independent of the actual simulation size. This would however waste a ton of memory in case of the walkers, and would render most applications impossible to run at a standard computer. However, in the case of the basis function arrays described in the previous section, the memory waste is not so large, and hence this is done in the code used in this thesis.

A second alternative would be to re-compile the code every time the system variables are changed. This is not optimal in the case of a generalized solver.

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[6], are necessarily orthogonal[7]. 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

$$|\chi\rangle = N\Big[\,|\uparrow\downarrow\rangle - i\,|\downarrow\uparrow\rangle\,\Big] \tag{A.1}$$

$$\langle\chi| = N\Big[\,\langle\uparrow\downarrow| + i\,\langle\downarrow\uparrow|\,\Big] \tag{A.2}$$

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

$$\int |\psi(x)|^2 dx = 1 \tag{A.4}$$

$$\begin{aligned}
\int |\psi(x)|^2 dx &= \int \psi^*(x)\psi(x)\mathrm{d}x \\
&= \int \langle\psi|x\rangle\langle x|\psi\rangle\,\mathrm{d}x \\
&= \langle\psi|\left[\int |x\rangle\langle x|\,dx\right]|\psi\rangle\,.
\end{aligned} \tag{A.5}$$

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**sualiz**e**r) 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*[8] with a graphical user interface coded using *PySide*[9]. 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:

```
1 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 LaTeXsupport is enabled if the correct packages is installed.

An example

```
#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

```
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

```
#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

```
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

```
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:

```
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`.

```
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.

```
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.

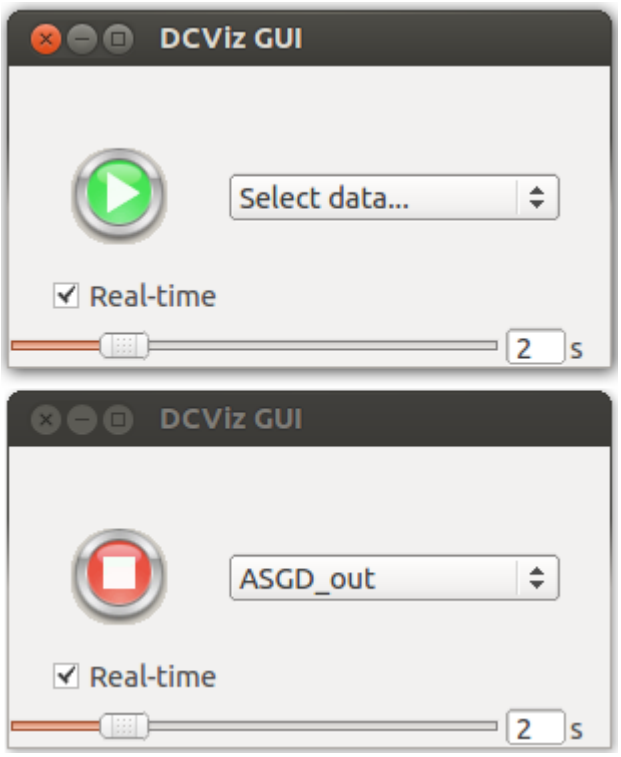

Figure B.1: Two consequent screen shots of the GUI. The first (top) is taken directly after the detector is finished loading files into the drop-down menu. The second is taken directly after the job is started.

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 [10]

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

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

In [2]: x, y, z, Z = symbols('x y z Z')

In [3]: r = sqrt(x*x + y*y + z*z)

In [4]: r
Out[4]: '(x**2 + y**2 + z**2)**(1/2)'

In [5]: phi = (Z*r - 2)*exp(-Z*r/2)

In [6]: phi
Out[6]: '(Z*(x**2 + y**2 + z**2)**(1/2) - 2)*exp(-Z*(x**2 + y**2 + z**2)**(1/2)/2)'

In [7]: diff(phi, x)
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

$$-\frac{Zx\left(Zr-4\right)}{2re^{\frac{1}{2}Zr}}$$

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, N):

        super(hydrogenicOrbitals, self).__init__(N, "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

```
1 def setupOrbitals(self):
2
3     for i, stateMap in self.stateMap.items():
4         nx, ny, nz = stateMap
5
6         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 cam be calculated by calling the `closedFormify()` function, however, unless the following member function is implemented, they are going to look messy.

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 is available. For hydrogen-like wave functions, the introductory 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 empty shells 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 and getCreturn

The empty shell of the `BasisFunctions::eval` functions in the `CPPbasis` class is implemented as below

```
self.evalShell = """
double __name__::eval(const Walker* walker, int i) {

__necessities__

    //__simpleExpr__

__preCalc__
    return __return__

}
"""
```

where `__preCalc__` is a generated C++ expression returned from `getCpre()`, and `__return__` is the returned C++ expression from `getCreturn()`. The commented `__simpleExpr__` will be replaced by the expression in nicely formatted SymPy output code. `__necessities__` is automatically detected by the script, and represents the Cartesian variable expressions needed by the expressions.

The functions take a SymPy generated expression on input, i.e. an orbital, gradient or Laplacian, and the corresponding index of the expression i. The reason these functions are split into a precalculation and a return expression is purely cosmetic. Consider the following example output for the hydrogen-like wave functions:

```
double dell_hydrogenic_9_y::eval(const Walker* walker, int i) {

    y = walker->r(i, 1);
    z = walker->r(i, 2);

    z2 = z*z;

    //-y*(k*(-r^2 + 3*z^2) + 6*r)*exp(-k*r/3)/(3*r)

    psi = -y*((*k)*(-walker->get_r_i2(i) + 3*z2) + 6*walker->get_r_i(i))/(3*walker->
        get_r_i(i));
    return psi*(*exp_factor);

}
```

The `*exp_factor` is the precalculated $n = 3$ exponential which is then simply multiplied by the non-exponential terms before being returned. The commented line is a clean version of the full expression. The required Cartesian components are retrieved prior to the evaluation.

The full implementation of `getCpre()` and `getCreturn()` for the hydrogen-like wave functions are given below

```
def getCReturn(self, expr, i):
     return "psi*(*exp_factor);"

def getCPre(self, expr, i):
     qNums = self.stateMap[i]
     return "    psi = %s;" % printing.ccode(expr/self.genericFactor(qNums))
```

makeOrbConstArg

Loading the generated `BasisFunctions` objects into the `Orbitals` object in the QMC code is rather a dull job, and is not designed to be done manually. The function `makeOrbConstArg` is designed to automate this process. This is best demonstrated by an example: Consider the following constructor of the hydrogen-like wave function's orbital class

```
basis_functions[0] = new hydrogenic_0(k, k2, exp_factor_n1);
basis_functions[1] = new hydrogenic_1(k, k2, exp_factor_n2);
//...
basis_functions[5] = new hydrogenic_5(k, k2, exp_factor_n3);
//..
basis_functions[14] = new hydrogenic_14(k, k2, exp_factor_n4);
//..
basis_functions[17] = new hydrogenic_17(k, k2, exp_factor_n4);
```

where `exp_factor_nk` represents $\exp(-Zr/k)$, which is saved as a pointer reference for reasons explained in Section 1.6.5. The same procedure is applied to the gradients and the Laplacians as well, leaving a

total of 90 sequential initializations. Everything needed in order to auto-generate the code is the following implementation

```
def makeOrbConstArgs(self, args, i):
    n = self.stateMap[i][0]
    args = args.replace('exp_factor', 'exp_fa-ctor_n%d' % n)
    return args
```

which tells the SymPy framework that the input arguments to e.g. element 1 is `(k, k2, exp_factor_n2)`, since the single particle orbital `self.phi[1]` has a principle quantum number $n = 2$. The input argument `args` is the default constructor arguments set up the the `initCPPbasis`, and is in the case of hydrogen-like wave functions `(k, k2, exp_factor)`.

The tables listed in Appendix D, E and F are all generated within seconds using this framework. The generated C++ code for these span 8975 lines not counting blank ones.

D

Harmonic Oscillator Orbitals 2D

Orbitals are constructed in the following fashion:

$$\phi(\vec{r})_{n_x,n_y} = H_{n_x}(kx)H_{n_y}(ky)e^{-\frac{1}{2}k^2r^2}$$

where $k = \sqrt{\omega\alpha}$, with ω being the oscillator frequency and α being the variational parameter.

$H_0(kx)$	1
$H_1(kx)$	$2kx$
$H_2(kx)$	$4k^2x^2 - 2$
$H_3(kx)$	$8k^3x^3 - 12kx$
$H_4(kx)$	$16k^4x^4 - 48k^2x^2 + 12$
$H_5(kx)$	$32k^5x^5 - 160k^3x^3 + 120kx$
$H_6(kx)$	$64k^6x^6 - 480k^4x^4 + 720k^2x^2 - 120$
$H_0(ky)$	1
$H_1(ky)$	$2ky$
$H_2(ky)$	$4k^2y^2 - 2$
$H_3(ky)$	$8k^3y^3 - 12ky$
$H_4(ky)$	$16k^4y^4 - 48k^2y^2 + 12$
$H_5(ky)$	$32k^5y^5 - 160k^3y^3 + 120ky$
$H_6(ky)$	$64k^6y^6 - 480k^4y^4 + 720k^2y^2 - 120$

Table D.1: Hermite polynomials used to construct orbital functions

$\phi_0 \to \phi_{0,0}$	
$\phi(\vec{r})$	1
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2 x$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2 y$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 2\right)$

Table D.2: Orbital expressions HOOrbitals : 0, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_1 \to \phi_{0,1}$	
$\phi(\vec{r})$	y
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2 xy$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\left(ky - 1\right)\left(ky + 1\right)$
$\nabla^2\phi(\vec{r})$	$k^2 y\left(k^2r^2 - 4\right)$

Table D.3: Orbital expressions HOOrbitals : 0, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_2 \to \phi_{1,0}$	
$\phi(\vec{r})$	x
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\left(kx - 1\right)\left(kx + 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2 xy$
$\nabla^2\phi(\vec{r})$	$k^2 x\left(k^2r^2 - 4\right)$

Table D.4: Orbital expressions HOOrbitals : 1, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_3 \to \phi_{0,2}$	
$\phi(\vec{r})$	$2k^2y^2 - 1$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2 x\left(2k^2y^2 - 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2 y\left(2k^2y^2 - 5\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 6\right)\left(2k^2y^2 - 1\right)$

Table D.5: Orbital expressions HOOrbitals : 0, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_4 \to \phi_{1,1}$	
$\phi(\vec{r})$	xy
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-y\left(kx - 1\right)\left(kx + 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-x\left(ky - 1\right)\left(ky + 1\right)$
$\nabla^2\phi(\vec{r})$	$k^2 xy\left(k^2r^2 - 6\right)$

Table D.6: Orbital expressions HOOrbitals : 1, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_5 \rightarrow \phi_{2,0}$	
$\phi(\vec{r})$	$2k^2x^2 - 1$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2x\left(2k^2x^2 - 5\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2y\left(2k^2x^2 - 1\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 6\right)\left(2k^2x^2 - 1\right)$

Table D.7: Orbital expressions HOOrbitals : 2, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_6 \rightarrow \phi_{0,3}$	
$\phi(\vec{r})$	$y\left(2k^2y^2 - 3\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2y^2 - 3\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-2k^4y^4 + 9k^2y^2 - 3$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2 - 8\right)\left(2k^2y^2 - 3\right)$

Table D.8: Orbital expressions HOOrbitals : 0, 3. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_7 \rightarrow \phi_{1,2}$	
$\phi(\vec{r})$	$x\left(2k^2y^2 - 1\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\left(kx - 1\right)\left(kx + 1\right)\left(2k^2y^2 - 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2y^2 - 5\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2 - 8\right)\left(2k^2y^2 - 1\right)$

Table D.9: Orbital expressions HOOrbitals : 1, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_8 \rightarrow \phi_{2,1}$	
$\phi(\vec{r})$	$y\left(2k^2x^2 - 1\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2x^2 - 5\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\left(ky - 1\right)\left(ky + 1\right)\left(2k^2x^2 - 1\right)$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2 - 8\right)\left(2k^2x^2 - 1\right)$

Table D.10: Orbital expressions HOOrbitals : 2, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_9 \rightarrow \phi_{3,0}$	
$\phi(\vec{r})$	$x\left(2k^2x^2 - 3\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-2k^4x^4 + 9k^2x^2 - 3$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2x^2 - 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2 - 8\right)\left(2k^2x^2 - 3\right)$

Table D.11: Orbital expressions HOOrbitals : 3, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{10} \rightarrow \phi_{0,4}$	
$\phi(\vec{r})$	$4k^4y^4 - 12k^2y^2 + 3$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2x\left(4k^4y^4 - 12k^2y^2 + 3\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2y\left(4k^4y^4 - 28k^2y^2 + 27\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 10\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$

Table D.12: Orbital expressions HOOrbitals : 0, 4. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{11} \rightarrow \phi_{1,3}$	
$\phi(\vec{r})$	$xy\left(2k^2y^2 - 3\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-y\left(kx - 1\right)\left(kx + 1\right)\left(2k^2y^2 - 3\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-x\left(2k^4y^4 - 9k^2y^2 + 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2xy\left(k^2r^2 - 10\right)\left(2k^2y^2 - 3\right)$

Table D.13: Orbital expressions HOOrbitals : 1, 3. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{12} \rightarrow \phi_{2,2}$	
$\phi(\vec{r})$	$\left(2k^2x^2 - 1\right)\left(2k^2y^2 - 1\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2x\left(2k^2x^2 - 5\right)\left(2k^2y^2 - 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2y\left(2k^2x^2 - 1\right)\left(2k^2y^2 - 5\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 10\right)\left(2k^2x^2 - 1\right)\left(2k^2y^2 - 1\right)$

Table D.14: Orbital expressions HOOrbitals : 2, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{13} \rightarrow \phi_{3,1}$	
$\phi(\vec{r})$	$xy\left(2k^2x^2 - 3\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-y\left(2k^4x^4 - 9k^2x^2 + 3\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-x\left(ky - 1\right)\left(ky + 1\right)\left(2k^2x^2 - 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2xy\left(k^2r^2 - 10\right)\left(2k^2x^2 - 3\right)$

Table D.15: Orbital expressions HOOrbitals : 3, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{14} \rightarrow \phi_{4,0}$	
$\phi(\vec{r})$	$4k^4x^4 - 12k^2x^2 + 3$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2x\left(4k^4x^4 - 28k^2x^2 + 27\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2y\left(4k^4x^4 - 12k^2x^2 + 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 10\right)\left(4k^4x^4 - 12k^2x^2 + 3\right)$

Table D.16: Orbital expressions HOOrbitals : 4, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{15} \to \phi_{0,5}$	
$\phi(\vec{r})$	$y\left(4k^4y^4 - 20k^2y^2 + 15\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(4k^4y^4 - 20k^2y^2 + 15\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-4k^6y^6 + 40k^4y^4 - 75k^2y^2 + 15$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2 - 12\right)\left(4k^4y^4 - 20k^2y^2 + 15\right)$

Table D.17: Orbital expressions HOOrbitals : 0, 5. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{16} \to \phi_{1,4}$	
$\phi(\vec{r})$	$x\left(4k^4y^4 - 12k^2y^2 + 3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\left(kx - 1\right)\left(kx + 1\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(4k^4y^4 - 28k^2y^2 + 27\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2 - 12\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$

Table D.18: Orbital expressions HOOrbitals : 1, 4. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{17} \to \phi_{2,3}$	
$\phi(\vec{r})$	$y\left(2k^2x^2 - 1\right)\left(2k^2y^2 - 3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(2k^2x^2 - 5\right)\left(2k^2y^2 - 3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-\left(2k^2x^2 - 1\right)\left(2k^4y^4 - 9k^2y^2 + 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2 - 12\right)\left(2k^2x^2 - 1\right)\left(2k^2y^2 - 3\right)$

Table D.19: Orbital expressions HOOrbitals : 2, 3. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{18} \to \phi_{3,2}$	
$\phi(\vec{r})$	$x\left(2k^2x^2 - 3\right)\left(2k^2y^2 - 1\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\left(2k^2y^2 - 1\right)\left(2k^4x^4 - 9k^2x^2 + 3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(2k^2x^2 - 3\right)\left(2k^2y^2 - 5\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2 - 12\right)\left(2k^2x^2 - 3\right)\left(2k^2y^2 - 1\right)$

Table D.20: Orbital expressions HOOrbitals : 3, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{19} \to \phi_{4,1}$	
$\phi(\vec{r})$	$y\left(4k^4x^4 - 12k^2x^2 + 3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(4k^4x^4 - 28k^2x^2 + 27\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-\left(ky - 1\right)\left(ky + 1\right)\left(4k^4x^4 - 12k^2x^2 + 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2 - 12\right)\left(4k^4x^4 - 12k^2x^2 + 3\right)$

Table D.21: Orbital expressions HOOrbitals : 4, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{20} \to \phi_{5,0}$	
$\phi(\vec{r})$	$x\left(4k^4x^4 - 20k^2x^2 + 15\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-4k^6x^6 + 40k^4x^4 - 75k^2x^2 + 15$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(4k^4x^4 - 20k^2x^2 + 15\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2 - 12\right)\left(4k^4x^4 - 20k^2x^2 + 15\right)$

Table D.22: Orbital expressions HOOrbitals : 5, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{21} \to \phi_{0,6}$	
$\phi(\vec{r})$	$8k^6y^6 - 60k^4y^4 + 90k^2y^2 - 15$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2x\left(8k^6y^6 - 60k^4y^4 + 90k^2y^2 - 15\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2y\left(8k^6y^6 - 108k^4y^4 + 330k^2y^2 - 195\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 14\right)\left(8k^6y^6 - 60k^4y^4 + 90k^2y^2 - 15\right)$

Table D.23: Orbital expressions HOOrbitals : 0, 6. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{22} \to \phi_{1,5}$	
$\phi(\vec{r})$	$xy\left(4k^4y^4 - 20k^2y^2 + 15\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-y\left(kx - 1\right)\left(kx + 1\right)\left(4k^4y^4 - 20k^2y^2 + 15\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-x\left(4k^6y^6 - 40k^4y^4 + 75k^2y^2 - 15\right)$
$\nabla^2\phi(\vec{r})$	$k^2xy\left(k^2r^2 - 14\right)\left(4k^4y^4 - 20k^2y^2 + 15\right)$

Table D.24: Orbital expressions HOOrbitals : 1, 5. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{23} \to \phi_{2,4}$	
$\phi(\vec{r})$	$\left(2k^2x^2 - 1\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2x\left(2k^2x^2 - 5\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2y\left(2k^2x^2 - 1\right)\left(4k^4y^4 - 28k^2y^2 + 27\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 14\right)\left(2k^2x^2 - 1\right)\left(4k^4y^4 - 12k^2y^2 + 3\right)$

Table D.25: Orbital expressions HOOrbitals : 2, 4. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{24} \to \phi_{3,3}$	
$\phi(\vec{r})$	$xy\left(2k^2x^2 - 3\right)\left(2k^2y^2 - 3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-y\left(2k^2y^2 - 3\right)\left(2k^4x^4 - 9k^2x^2 + 3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-x\left(2k^2x^2 - 3\right)\left(2k^4y^4 - 9k^2y^2 + 3\right)$
$\nabla^2\phi(\vec{r})$	$k^2xy\left(k^2r^2 - 14\right)\left(2k^2x^2 - 3\right)\left(2k^2y^2 - 3\right)$

Table D.26: Orbital expressions HOOrbitals : 3, 3. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{25} \to \phi_{4,2}$	
$\phi(\vec{r})$	$\left(2k^2y^2-1\right)\left(4k^4x^4-12k^2x^2+3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2x\left(2k^2y^2-1\right)\left(4k^4x^4-28k^2x^2+27\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2y\left(2k^2y^2-5\right)\left(4k^4x^4-12k^2x^2+3\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2-14\right)\left(2k^2y^2-1\right)\left(4k^4x^4-12k^2x^2+3\right)$

Table D.27: Orbital expressions HOOrbitals : 4, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{26} \to \phi_{5,1}$	
$\phi(\vec{r})$	$xy\left(4k^4x^4-20k^2x^2+15\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-y\left(4k^6x^6-40k^4x^4+75k^2x^2-15\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-x\left(ky-1\right)\left(ky+1\right)\left(4k^4x^4-20k^2x^2+15\right)$
$\nabla^2\phi(\vec{r})$	$k^2xy\left(k^2r^2-14\right)\left(4k^4x^4-20k^2x^2+15\right)$

Table D.28: Orbital expressions HOOrbitals : 5, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{27} \to \phi_{6,0}$	
$\phi(\vec{r})$	$8k^6x^6-60k^4x^4+90k^2x^2-15$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2x\left(8k^6x^6-108k^4x^4+330k^2x^2-195\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2y\left(8k^6x^6-60k^4x^4+90k^2x^2-15\right)$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2-14\right)\left(8k^6x^6-60k^4x^4+90k^2x^2-15\right)$

Table D.29: Orbital expressions HOOrbitals : 6, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

E

Harmonic Oscillator Orbitals 3D

Orbitals are constructed in the following fashion:

$$\phi(\vec{r})_{n_x,n_y,n_z} = H_{n_x}(kx)H_{n_y}(ky)H_{n_z}(kz)e^{-\frac{1}{2}k^2r^2}$$

where $k = \sqrt{\omega\alpha}$, with ω being the oscillator frequency and α being the variational parameter.

$H_0(kx)$	1
$H_1(kx)$	$2kx$
$H_2(kx)$	$4k^2x^2 - 2$
$H_3(kx)$	$8k^3x^3 - 12kx$
$H_0(ky)$	1
$H_1(ky)$	$2ky$
$H_2(ky)$	$4k^2y^2 - 2$
$H_3(ky)$	$8k^3y^3 - 12ky$
$H_0(kz)$	1
$H_1(kz)$	$2kz$
$H_2(kz)$	$4k^2z^2 - 2$
$H_3(kz)$	$8k^3z^3 - 12kz$

Table E.1: Hermite polynomials used to construct orbital functions

$\phi_0 \to \phi_{0,0,0}$	
$\phi(\vec{r})$	1
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2 x$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2 y$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-k^2 z$
$\nabla^2\phi(\vec{r})$	$k^2\left(k^2r^2 - 3\right)$

Table E.2: Orbital expressions HOOrbitals3D : 0, 0, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_1 \to \phi_{0,0,1}$	
$\phi(\vec{r})$	z
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2 xz$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2 yz$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\left(kz - 1\right)\left(kz + 1\right)$
$\nabla^2\phi(\vec{r})$	$k^2 z\left(k^2r^2 - 5\right)$

Table E.3: Orbital expressions HOOrbitals3D : 0, 0, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_2 \to \phi_{0,1,0}$	
$\phi(\vec{r})$	y
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2 xy$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\left(ky - 1\right)\left(ky + 1\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-k^2 yz$
$\nabla^2\phi(\vec{r})$	$k^2 y\left(k^2r^2 - 5\right)$

Table E.4: Orbital expressions HOOrbitals3D : 0, 1, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_3 \to \phi_{1,0,0}$	
$\phi(\vec{r})$	x
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\left(kx - 1\right)\left(kx + 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2 xy$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-k^2 xz$
$\nabla^2\phi(\vec{r})$	$k^2 x\left(k^2r^2 - 5\right)$

Table E.5: Orbital expressions HOOrbitals3D : 1, 0, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_4 \to \phi_{0,0,2}$	
$\phi(\vec{r})$	$4k^2z^2 - 2$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-2k^2 x\left(2k^2z^2 - 1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-2k^2 y\left(2k^2z^2 - 1\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-2k^2 z\left(2k^2z^2 - 5\right)$
$\nabla^2\phi(\vec{r})$	$2k^2\left(k^2r^2 - 7\right)\left(2k^2z^2 - 1\right)$

Table E.6: Orbital expressions HOOrbitals3D : 0, 0, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_5 \rightarrow \phi_{0,1,1}$	
$\phi(\vec{r})$	yz
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2xyz$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-z\,(ky-1)\,(ky+1)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-y\,(kz-1)\,(kz+1)$
$\nabla^2\phi(\vec{r})$	$k^2yz\left(k^2r^2-7\right)$

Table E.7: Orbital expressions HOOrbitals3D : 0, 1, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_6 \rightarrow \phi_{0,2,0}$	
$\phi(\vec{r})$	$4k^2y^2-2$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-2k^2x\left(2k^2y^2-1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-2k^2y\left(2k^2y^2-5\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-2k^2z\left(2k^2y^2-1\right)$
$\nabla^2\phi(\vec{r})$	$2k^2\left(k^2r^2-7\right)\left(2k^2y^2-1\right)$

Table E.8: Orbital expressions HOOrbitals3D : 0, 2, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_7 \rightarrow \phi_{1,0,1}$	
$\phi(\vec{r})$	xz
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-z\,(kx-1)\,(kx+1)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2xyz$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-x\,(kz-1)\,(kz+1)$
$\nabla^2\phi(\vec{r})$	$k^2xz\left(k^2r^2-7\right)$

Table E.9: Orbital expressions HOOrbitals3D : 1, 0, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_8 \rightarrow \phi_{1,1,0}$	
$\phi(\vec{r})$	xy
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-y\,(kx-1)\,(kx+1)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-x\,(ky-1)\,(ky+1)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-k^2xyz$
$\nabla^2\phi(\vec{r})$	$k^2xy\left(k^2r^2-7\right)$

Table E.10: Orbital expressions HOOrbitals3D : 1, 1, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_9 \rightarrow \phi_{2,0,0}$	
$\phi(\vec{r})$	$4k^2x^2-2$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-2k^2x\left(2k^2x^2-5\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-2k^2y\left(2k^2x^2-1\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-2k^2z\left(2k^2x^2-1\right)$
$\nabla^2\phi(\vec{r})$	$2k^2\left(k^2r^2-7\right)\left(2k^2x^2-1\right)$

Table E.11: Orbital expressions HOOrbitals3D : 2, 0, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{10} \rightarrow \phi_{0,0,3}$	
$\phi(\vec{r})$	$z\left(2k^2z^2-3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xz\left(2k^2z^2-3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2yz\left(2k^2z^2-3\right)$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-2k^4z^4+9k^2z^2-3$
$\nabla^2\phi(\vec{r})$	$k^2z\left(k^2r^2-9\right)\left(2k^2z^2-3\right)$

Table E.12: Orbital expressions HOOrbitals3D : 0, 0, 3. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{11} \rightarrow \phi_{0,1,2}$	
$\phi(\vec{r})$	$y\left(2k^2z^2-1\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(2k^2z^2-1\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-\left(ky-1\right)\left(ky+1\right)\left(2k^2z^2-1\right)$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-k^2yz\left(2k^2z^2-5\right)$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2-9\right)\left(2k^2z^2-1\right)$

Table E.13: Orbital expressions HOOrbitals3D : 0, 1, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{12} \rightarrow \phi_{0,2,1}$	
$\phi(\vec{r})$	$z\left(2k^2y^2-1\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xz\left(2k^2y^2-1\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2yz\left(2k^2y^2-5\right)$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-\left(kz-1\right)\left(kz+1\right)\left(2k^2y^2-1\right)$
$\nabla^2\phi(\vec{r})$	$k^2z\left(k^2r^2-9\right)\left(2k^2y^2-1\right)$

Table E.14: Orbital expressions HOOrbitals3D : 0, 2, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{13} \rightarrow \phi_{0,3,0}$	
$\phi(\vec{r})$	$y\left(2k^2y^2-3\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(2k^2y^2-3\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-2k^4y^4+9k^2y^2-3$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-k^2yz\left(2k^2y^2-3\right)$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2-9\right)\left(2k^2y^2-3\right)$

Table E.15: Orbital expressions HOOrbitals3D : 0, 3, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{14} \rightarrow \phi_{1,0,2}$	
$\phi(\vec{r})$	$x\left(2k^2z^2-1\right)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\left(kx-1\right)\left(kx+1\right)\left(2k^2z^2-1\right)$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-k^2xy\left(2k^2z^2-1\right)$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-k^2xz\left(2k^2z^2-5\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2-9\right)\left(2k^2z^2-1\right)$

Table E.16: Orbital expressions HOOrbitals3D : 1, 0, 2. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{15} \rightarrow \phi_{1,1,1}$	
$\phi(\vec{r})$	xyz
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-yz\,(kx-1)\,(kx+1)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-xz\,(ky-1)\,(ky+1)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-xy\,(kz-1)\,(kz+1)$
$\nabla^2\phi(\vec{r})$	$k^2xyz\left(k^2r^2-9\right)$

Table E.17: Orbital expressions HOOrbitals3D : 1, 1, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{16} \rightarrow \phi_{1,2,0}$	
$\phi(\vec{r})$	$x\left(2k^2y^2-1\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\left(kx-1\right)\left(kx+1\right)\left(2k^2y^2-1\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2y^2-5\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-k^2xz\left(2k^2y^2-1\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2-9\right)\left(2k^2y^2-1\right)$

Table E.18: Orbital expressions HOOrbitals3D : 1, 2, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{17} \rightarrow \phi_{2,0,1}$	
$\phi(\vec{r})$	$z\left(2k^2x^2-1\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2xz\left(2k^2x^2-5\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2yz\left(2k^2x^2-1\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\left(kz-1\right)\left(kz+1\right)\left(2k^2x^2-1\right)$
$\nabla^2\phi(\vec{r})$	$k^2z\left(k^2r^2-9\right)\left(2k^2x^2-1\right)$

Table E.19: Orbital expressions HOOrbitals3D : 2, 0, 1. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{18} \rightarrow \phi_{2,1,0}$	
$\phi(\vec{r})$	$y\left(2k^2x^2-1\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2x^2-5\right)$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\left(ky-1\right)\left(ky+1\right)\left(2k^2x^2-1\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-k^2yz\left(2k^2x^2-1\right)$
$\nabla^2\phi(\vec{r})$	$k^2y\left(k^2r^2-9\right)\left(2k^2x^2-1\right)$

Table E.20: Orbital expressions HOOrbitals3D : 2, 1, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

$\phi_{19} \rightarrow \phi_{3,0,0}$	
$\phi(\vec{r})$	$x\left(2k^2x^2-3\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-2k^4x^4+9k^2x^2-3$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-k^2xy\left(2k^2x^2-3\right)$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-k^2xz\left(2k^2x^2-3\right)$
$\nabla^2\phi(\vec{r})$	$k^2x\left(k^2r^2-9\right)\left(2k^2x^2-3\right)$

Table E.21: Orbital expressions HOOrbitals3D : 3, 0, 0. Factor $e^{-\frac{1}{2}k^2r^2}$ is omitted.

F

Hydrogen Orbitals

Orbitals are constructed in the following fashion:

$$\phi(\vec{r})_{n,l,m} = L_{n-l-1}^{2l+1}\Big(\frac{2r}{n}k\Big)S_l^m(\vec{r})e^{-\frac{r}{n}k}$$

where n is the principal quantum number, $k = \alpha Z$ with Z being the nucleus charge and α being the variational parameter.

$$l = 0,\ 1,\ ...,\ (n-1)$$

$$m = -l,\ (-l+1),\ ...,\ (l-1),\ l$$

$\phi_0 \to \phi_{1,0,0}$	
$\phi(\vec{r})$	1
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kx}{r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{ky}{r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kz}{r}$
$\nabla^2\phi(\vec{r})$	$\frac{k(kr-2)}{r}$

Table F.1: Orbital expressions hydrogenicOrbitals : 1, 0, 0. Factor e^{-kr} is omitted.

$\phi_1 \to \phi_{2,0,0}$	
$\phi(\vec{r})$	$kr-2$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kx(kr-4)}{2r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{ky(kr-4)}{2r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kz(kr-4)}{2r}$
$\nabla^2\phi(\vec{r})$	$\frac{k(kr-8)(kr-2)}{4r}$

Table F.2: Orbital expressions hydrogenicOrbitals : 2, 0, 0. Factor $e^{-\frac{1}{2}kr}$ is omitted.

$\phi_2 \to \phi_{2,1,0}$	
$\phi(\vec{r})$	z
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz}{2r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kyz}{2r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$\frac{-kz^2+2r}{2r}$
$\nabla^2\phi(\vec{r})$	$\frac{kz(kr-8)}{4r}$

Table F.3: Orbital expressions hydrogenicOrbitals : 2, 1, 0. Factor $e^{-\frac{1}{2}kr}$ is omitted.

$\phi_3 \to \phi_{2,1,1}$	
$\phi(\vec{r})$	x
$\vec{i} \cdot \nabla\phi(\vec{r})$	$\frac{-kx^2+2r}{2r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kxy}{2r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz}{2r}$
$\nabla^2\phi(\vec{r})$	$\frac{kx(kr-8)}{4r}$

Table F.4: Orbital expressions hydrogenicOrbitals : 2, 1, 1. Factor $e^{-\frac{1}{2}kr}$ is omitted.

$\phi_4 \rightarrow \phi_{2,1,-1}$	
$\phi(\vec{r})$	y
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kxy}{2r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$\frac{-ky^2+2r}{2r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kyz}{2r}$
$\nabla^2\phi(\vec{r})$	$\frac{ky(kr-8)}{4r}$

Table F.5: Orbital expressions hydrogenicOrbitals : 2, 1, -1. Factor $e^{-\frac{1}{2}kr}$ is omitted.

$\phi_5 \rightarrow \phi_{3,0,0}$	
$\phi(\vec{r})$	$2k^2r^2 - 18kr + 27$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kx\left(2k^2r^2-30kr+81\right)}{3r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{ky\left(2k^2r^2-30kr+81\right)}{3r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kz\left(2k^2r^2-30kr+81\right)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{k(kr-18)\left(2k^2r^2-18kr+27\right)}{9r}$

Table F.6: Orbital expressions hydrogenicOrbitals : 3, 0, 0. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_6 \rightarrow \phi_{3,1,0}$	
$\phi(\vec{r})$	$z\,(kr-6)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz(kr-9)}{3r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kyz(kr-9)}{3r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$\frac{3kr^2-kz^2(kr-9)-18r}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{kz(kr-18)(kr-6)}{9r}$

Table F.7: Orbital expressions hydrogenicOrbitals : 3, 1, 0. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_7 \rightarrow \phi_{3,1,1}$	
$\phi(\vec{r})$	$x\,(kr-6)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$\frac{3kr^2-kx^2(kr-9)-18r}{3r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kxy(kr-9)}{3r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz(kr-9)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{kx(kr-18)(kr-6)}{9r}$

Table F.8: Orbital expressions hydrogenicOrbitals : 3, 1, 1. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_8 \rightarrow \phi_{3,1,-1}$	
$\phi(\vec{r})$	$y\,(kr-6)$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\frac{kxy(kr-9)}{3r}$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$\frac{3kr^2-ky^2(kr-9)-18r}{3r}$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-\frac{kyz(kr-9)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{ky(kr-18)(kr-6)}{9r}$

Table F.9: Orbital expressions hydrogenicOrbitals : 3, 1, -1. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_9 \rightarrow \phi_{3,2,0}$	
$\phi(\vec{r})$	$-r^2+3z^2$
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\frac{x\left(k\left(-r^2+3z^2\right)+6r\right)}{3r}$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-\frac{y\left(k\left(-r^2+3z^2\right)+6r\right)}{3r}$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-\frac{z\left(k\left(-r^2+3z^2\right)-12r\right)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{k\left(-r^2+3z^2\right)(kr-18)}{9r}$

Table F.10: Orbital expressions hydrogenicOrbitals : 3, 2, 0. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_{10} \rightarrow \phi_{3,2,1}$	
$\phi(\vec{r})$	xz
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\frac{z\left(kx^2-3r\right)}{3r}$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-\frac{kxyz}{3r}$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-\frac{x\left(kz^2-3r\right)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{kxz(kr-18)}{9r}$

Table F.11: Orbital expressions hydrogenicOrbitals : 3, 2, 1. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_{11} \rightarrow \phi_{3,2,-1}$	
$\phi(\vec{r})$	yz
$\vec{i}\cdot\nabla\phi(\vec{r})$	$-\frac{kxyz}{3r}$
$\vec{j}\cdot\nabla\phi(\vec{r})$	$-\frac{z\left(ky^2-3r\right)}{3r}$
$\vec{k}\cdot\nabla\phi(\vec{r})$	$-\frac{y\left(kz^2-3r\right)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{kyz(kr-18)}{9r}$

Table F.12: Orbital expressions hydrogenicOrbitals : 3, 2, -1. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_{12} \to \phi_{3,2,2}$	
$\phi(\vec{r})$	$x^2 - y^2$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{x\left(k\left(x^2-y^2\right)-6r\right)}{3r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{y\left(k\left(x^2-y^2\right)+6r\right)}{3r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kz\left(x^2-y^2\right)}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{k\left(x^2-y^2\right)(kr-18)}{9r}$

Table F.13: Orbital expressions hydrogenicOrbitals : 3, 2, 2. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_{13} \to \phi_{3,2,-2}$	
$\phi(\vec{r})$	xy
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{y\left(kx^2-3r\right)}{3r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{x\left(ky^2-3r\right)}{3r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kxyz}{3r}$
$\nabla^2\phi(\vec{r})$	$\frac{kxy(kr-18)}{9r}$

Table F.14: Orbital expressions hydrogenicOrbitals : 3, 2, -2. Factor $e^{-\frac{1}{3}kr}$ is omitted.

$\phi_{14} \to \phi_{4,0,0}$	
$\phi(\vec{r})$	$k^3r^3 - 24k^2r^2 + 144kr - 192$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kx\left(k^3r^3-36k^2r^2+336kr-768\right)}{4r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{ky\left(k^3r^3-36k^2r^2+336kr-768\right)}{4r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kz\left(k^3r^3-36k^2r^2+336kr-768\right)}{4r}$
$\nabla^2\phi(\vec{r})$	$\frac{k(kr-32)\left(k^3r^3-24k^2r^2+144kr-192\right)}{16r}$

Table F.15: Orbital expressions hydrogenicOrbitals : 4, 0, 0. Factor $e^{-\frac{1}{4}kr}$ is omitted.

$\phi_{15} \to \phi_{4,1,0}$	
$\phi(\vec{r})$	$z\left(k^2r^2 - 20kr + 80\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz(kr-20)(kr-8)}{4r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kyz(kr-20)(kr-8)}{4r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$\frac{4k^2r^3-80kr^2-kz^2(kr-20)(kr-8)+320r}{4r}$
$\nabla^2\phi(\vec{r})$	$\frac{kz(kr-32)\left(k^2r^2-20kr+80\right)}{16r}$

Table F.16: Orbital expressions hydrogenicOrbitals : 4, 1, 0. Factor $e^{-\frac{1}{4}kr}$ is omitted.

$\phi_{16} \to \phi_{4,1,1}$	
$\phi(\vec{r})$	$x\left(k^2r^2 - 20kr + 80\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$\frac{4k^2r^3 - 80kr^2 - kx^2(kr-20)(kr-8) + 320r}{4r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$-\frac{kxy(kr-20)(kr-8)}{4r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kxz(kr-20)(kr-8)}{4r}$
$\nabla^2\phi(\vec{r})$	$\frac{kx(kr-32)\left(k^2r^2-20kr+80\right)}{16r}$

Table F.17: Orbital expressions hydrogenicOrbitals : 4, 1, 1. Factor $e^{-\frac{1}{4}kr}$ is omitted.

$\phi_{17} \to \phi_{4,1,-1}$	
$\phi(\vec{r})$	$y\left(k^2r^2 - 20kr + 80\right)$
$\vec{i} \cdot \nabla\phi(\vec{r})$	$-\frac{kxy(kr-20)(kr-8)}{4r}$
$\vec{j} \cdot \nabla\phi(\vec{r})$	$\frac{4k^2r^3 - 80kr^2 - ky^2(kr-20)(kr-8) + 320r}{4r}$
$\vec{k} \cdot \nabla\phi(\vec{r})$	$-\frac{kyz(kr-20)(kr-8)}{4r}$
$\nabla^2\phi(\vec{r})$	$\frac{ky(kr-32)\left(k^2r^2-20kr+80\right)}{16r}$

Table F.18: Orbital expressions hydrogenicOrbitals : 4, 1, -1. Factor $e^{-\frac{1}{4}kr}$ is omitted.

Bibliography

[1] J. Høgberget, *Git Repository: LibBorealis*, 2013. [Online]. Available: http://www.github.com/jorgehog/QMC2

[2] M. Hjorth-Jensen, "Computational Physics," 2010.

[3] I. Shavitt and R. J. Bartlett, *Many-body methods in chemistry and physics: MBPT and coupled-cluster theory*, ser. Cambridge molecular science series. Cambridge University Press, 2009. [Online]. Available: http://books.google.no/books?id=gU1eHAAACAAJ

[4] D. C. Lay, *Linear Algebra and its Applications*, 4th ed. Pearson, 2012.

[5] B. Hammond, J. W. A. Lester, and P. J. Reynolds, *Monte Carlo Methods in Ab Initio Quantum Chemistry*, S. Lin, Ed. World Scientific Publishing Co., 1994.

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

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

[8] (2013, May) Matplotlib website. [Online]. Available: http://matplotlib.org

[9] (2013, May) PySide website. [Online]. Available: http://qt-project.org/wiki/Category:LanguageBindings::PySide

[10] (2013, May) SymPy website. [Online]. Available: http://sympy.org