The basics of the DMC code

Jørgen Høgberget

What can it do?

Quantities of interest

- Ground state energies and densities.
- Energy distributions.

Implemented Systems

- Harmonic oscillator systems (2D, 3D, doublewells)
- Atomic systems (Atoms, homonuclear diatomic molecules)

What can it do?

Quantities of interest

- Ground state energies and densities.
- Energy distributions.

Implemented Systems

- Harmonic oscillator systems (2D, 3D, doublewells)
- Atomic systems (Atoms, homonuclear diatomic molecules)

Underlying goals and assumptions

In a nutshell

Ab-initio, Efficiency, Transparency

Specifics: Assumptions

- Single Slater-determinant ansatz.
- Single-parameter Jastrow correlation function.

Underlying goals and assumptions

In a nutshell

Ab-initio, Efficiency, Transparency

Specifics: Assumptions

- Single Slater-determinant ansatz.
- Single-parameter Jastrow correlation function.

Why ab-initio?

- Avoid self-consistency in multi-scale modelling.
- System acts independent of our aims of the computation - nature is not influenced by our intentions.

Why ab-initio?

- Avoid self-consistency in multi-scale modelling.
- System acts independent of our aims of the computation - nature is not influenced by our intentions.
- Fundamental modelling: High academic eigenvalue.

Why ab-initio?

- Avoid self-consistency in multi-scale modelling.
- System acts independent of our aims of the computation - nature is not influenced by our intentions.
- Fundamental modelling: High academic eigenvalue.
- Downside: It is harder to solve equations when you cannot assume to know the answer (even if you do).

Why ab-initio?

- Avoid self-consistency in multi-scale modelling.
- System acts independent of our aims of the computation - nature is not influenced by our intentions.
- Fundamental modelling: High academic eigenvalue.
- Downside: It is harder to solve equations when you cannot assume to know the answer (even if you do).

Efficiency vs. generalization and precision

The precision of DMC (given a model Hamiltonian) is, to first approximation, limited by one thing: **The trial wave function**.

General MB. WF. construction:

SP-basis $\rightarrow$ Det. Basis $\rightarrow$ Combine Determinants

Efficiency vs. generalization and precision

The precision of DMC (given a model Hamiltonian) is, to first approximation, limited by one thing: **The trial wave function**.

General MB. WF. construction:

$$\text{SP-basis} \quad \rightarrow \quad \text{Det. Basis} \quad \rightarrow \quad \text{Combine Determinants}$$

Efficiency vs. generalization and precision

The precision of QMC (given a model Hamiltonian) is, to first approximation, limited by one thing: **The trial wave function**.

Compromise to ensure efficiency and “ideal scaling“

SP-basis $\rightarrow$ Det. ~~Basis~~ $\not\rightarrow$ ~~Combine Determinants~~

Start from the bottom up

- Implement single particle wave functions $\Phi_j(\vec{r}_i)$.

```
class BasisFunctions {
public:
    BasisFunctions();

    virtual double eval(const Walker* walker, int i) = 0;
};
```

```
//n_x = n_y = n_z = 1
double HarmonicOscillator3D_15::eval(const Walker* walker, int i) {

    x = walker->r(i, 0);
    y = walker->r(i, 1);
    z = walker->r(i, 2);

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

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

}
```

- Implement single particle wave functions $\Phi_j(\vec{r}_i)$.
- Extensive bases should be generated with the supplied *SymPy*-script (read Appendix C in the Thesis).
- Setup the determinant.

- Implement single particle wave functions $\Phi_j(\vec{r}_i)$.
- Extensive bases should be generated with the supplied *SymPy*-script (read Appendix C in the Thesis).
- Setup the determinant.

```
class Orbitals {
protected:

    BasisFunctions** basis_functions;
    BasisFunctions*** del_basis_functions;
    BasisFunctions** lapl_basis_functions;

    //Important: All basis elements and the orbital wrapper
    //SHARE parameter references
    virtual void set_parameter(double parameter, int n) = 0;

public:

    Orbitals(int n_p, int dim);

    virtual double      phi(const Walker* walker, int i, int q);
    virtual double  del_phi(const Walker* walker, int i, int q, int d);
    virtual double lapl_phi(const Walker* walker, int i, int q);

};
```

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

double Orbitals::del_phi(const Walker* walker, int particle, int q_num, int d)
    return del_basis_functions[d][q_num]->eval(walker, particle);
}

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

Idea: Fill the basis function vectors in the subclass constructor, and you're good to go.

```
HaromonicOscillator::HarmonicOscillator{
    ...

    basis_functions[0] = new HarmonicOscillator3D_0(...);
    basis_functions[1] = new HarmonicOscillator3D_1(...);
    basis_functions[2] = new HarmonicOscillator3D_2(...);
    ...

    del_basis_functions[0][0] = new del_HarmonicOscillator3D_0_x(...);
    del_basis_functions[1][0] = new del_HarmonicOscillator3D_0_y(...);
    del_basis_functions[2][0] = new del_HarmonicOscillator3D_0_z(...);
    del_basis_functions[0][1] = new del_HarmonicOscillator3D_1_x(...);
    del_basis_functions[1][1] = new del_HarmonicOscillator3D_1_y(...);
    del_basis_functions[2][1] = new del_HarmonicOscillator3D_1_z(...);
    ...

    lapl_basis_functions[0] = new lapl_HarmonicOscillator3D_0(...);
    lapl_basis_functions[1] = new lapl_HarmonicOscillator3D_1(...);
    ...

}
```

- Implement single particle wave functions $\Phi_j(\vec{r}_i)$.
- Extensive bases should be generated with the supplied *SymPy* script (read Appendix C in the Thesis).
- Setup the determinant.
- Luckily this is done automatically by the SymPy script.. Such a manual coding opens up possibility for seting up *any* of the determinants, not just the "ground state".

Overcomplicated?

- Implement single particle wave functions $\Phi_j(\vec{r}_i)$.
- Extensive bases should be generated with the supplied *SymPy* script (read Appendix C in the Thesis).
- Setup the determinant.
- Luckily this is done automatically by the SymPy script.. Such a manual coding opens up possibility for seting up *any* of the determinants, not just the "ground state".

Overcomplicated?

Perhaps it can be simplified? Doesn't matter really. This way of structuring makes implementations of expanded **SP** bases and molecules trivial.

- Implement single particle wave functions $\Phi_j(\vec{r}_i)$.
- Extensive bases should be generated with the supplied *SymPy* script (read Appendix C in the Thesis).
- Setup the determinant.
- Luckily this is done automatically by the SymPy script.. Such a manual coding opens up possibility for seting up *any* of the determinants, not just the ”ground state“.

Overcomplicated?
Perhaps it can be simplified? Doesn't matter really. This way of structuring makes implementations of expanded **SP** bases and molecules trivial.

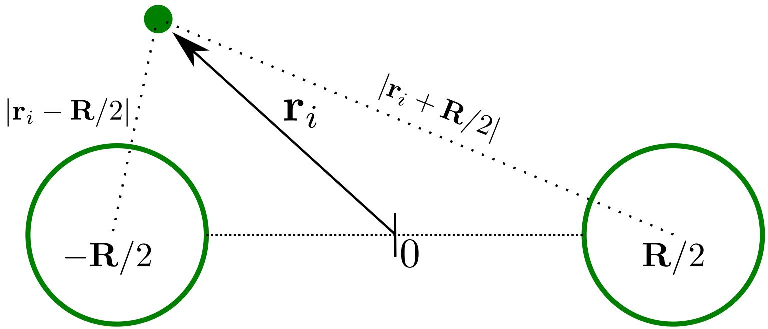
Expanded Bases

```
class ExpandedBasis : public Orbitals {
public:
    ExpandedBasis(...);

    double phi(...);
    double del_phi(...);
    double lapl_phi(...);

protected:
    arma::mat coeffs;

    Orbitals* basis;

};
```

Expanded Bases

```
double ExpandedBasis::phi(const Walker* walker, int i, int q) {

    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, m) * basis->phi(walker, i, m);
    }

    return value;

}
```

Can be loaded into the QMC machinery as any other Orbital instance.


$|\mathbf{r}_i - \mathbf{R}/2|$
$\mathbf{r}_i$
$|\mathbf{r}_i + \mathbf{R}/2|$
$-\mathbf{R}/2$
0
$\mathbf{R}/2$

$$\widehat{\mathbf{H}}_{\mathrm{Mol.}}(\mathbf{r},\mathbf{R}) = \sum_{i=1}^{N}\left[-\frac{1}{2}\nabla_i^2 - \frac{Z}{|\mathbf{r}_i + \mathbf{R}/2|} - \frac{Z}{|\mathbf{r}_i - \mathbf{R}/2|}\right] + \frac{Z^2}{R} + \sum_{i<j}\frac{1}{r_{ij}}.$$

(di)Molecules

```
double DiAtomCore::get_pot_E(const Walker* walker) const {

    double e_pot = 0;
    double com_corr, shared;

    double quarterR2 = 0.25*(*R)*(*R);
    for (int i = 0; i < n_p; i++) {

        shared = walker->get_r_i2(i)+ quarterR2;
        com_corr = (*R)*walker->r(i, 0);

        e_pot -= Z*(1./sqrt(shared + com_corr) + 1./sqrt(shared - com_corr));
    }

    e_pot += Z*Z/(*R);

    return e_pot;

}
```

(di)Molecules

```
class DiTransform : public Orbitals {
...

protected:
    double* R;

    Orbitals* nucleus1, nucleus2;
    Walker* walker_nucleus1, walker_nucleus2;

    //Wrap wrap wrap, it's christmas!
    double get_parameter(int n) {
        return nucleus1->get_parameter(n);
    }

    void set_parameter(double parameter, int n) {
        nucleus1->set_parameter(parameter, n);
        nucleus2->set_parameter(parameter, n);
    }
};
```

(di)Molecules

- A Walker instance for each nucleus ensures minimal overhead when transforming from atoms to molecules (precalculates distances etc.).
- An Orbital instance for each nucleus ensures that all optimizations from the single atom scheme can be carried to the molecular one (precalculates exponential factors etc.).

(di)Molecules

- A Walker instance for each nucleus ensures minimal overhead when transforming from atoms to molecules (precalculates distances etc.).
- An Orbital instance for each nucleus ensures that all optimizations from the single atom scheme can be carried to the molecular one (precalculates exponential factors etc.).
- QMC is not memory intensive. Matrices are of size $N \times N$ at max, where $N = 100$ is the world record.

(di)Molecules

- A Walker instance for each nucleus ensures minimal overhead when transforming from atoms to molecules (precalculates distances etc.).
- An Orbital instance for each nucleus ensures that all optimizations from the single atom scheme can be carried to the molecular one (precalculates exponential factors etc.).
- QMC is not memory intensive. Matrices are of size $N \times N$ at max, where $N = 100$ is the world record.

(di)Molecules

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

    walker->calc_r_i(i);

    //Apply the molecular transformation!
    walker_nucleus1->r.row(i) = walker->r.row(i);
    walker_nucleus2->r.row(i) = walker->r.row(i);

    walker_nucleus1->r(i, 0) += (*R) / 2;
    walker_nucleus2->r(i, 0) -= (*R) / 2;

    double shared = walker->get_r_i2(i) + 0.25 * (*R)*(*R);
    double comm_spec = walker->r(i, 0)*(*R);

    walker_nucleus1->r2(i) = shared + comm_spec;
    walker_nucleus2->r2(i) = shared - comm_spec;

    nucleus1->set_qnum_indie_terms(walker_nucleus1, i);
    nucleus2->set_qnum_indie_terms(walker_nucleus2, i);
```

Transforming SPWFs to molecular SPWFs

Plus and minus refers to the two nuclei, *H* refers to the standard hydrogen-like basis.

$$\begin{aligned}\phi^{+}_{nlm}(\mathbf{r}_i, \mathbf{R}) &= \phi^{\mathrm{H}}_{nlm}(\mathbf{r}_i + \mathbf{R}/2) + \phi^{\mathrm{H}}_{nlm}(\mathbf{r}_i - \mathbf{R}/2),\\ \phi^{-}_{nlm}(\mathbf{r}_i, \mathbf{R}) &= \phi^{\mathrm{H}}_{nlm}(\mathbf{r}_i + \mathbf{R}/2) - \phi^{\mathrm{H}}_{nlm}(\mathbf{r}_i - \mathbf{R}/2),\end{aligned}$$

which reads “electron surrounding first nucleus combined with electron surrounding second nucleus”.

Applying the transformation

```
double DiTransform::phi(const Walker* walker, int i, int q) {

    (void) walker;
    int sign = minusPower(q);

    return nucleus1->phi(walker_nucleus1, i, q / 2) +
            sign * nucleus2->phi(walker_nucleus2, i, q / 2);

}
```

We can reuse closed form expressions

$$
\begin{aligned}
\mathbf{j}\cdot\nabla_i\phi^{\pm}_{nlm}(\mathbf{r}_i,\mathbf{R}) &= \underbrace{\frac{\partial(y_i+R_y/2)}{\partial y_i}}_{1}\frac{\partial\phi^{\mathrm{H}}_{nlm}(\mathbf{r}_i+\mathbf{R}/2)}{\partial(y_i+R_y/2)} \\
&\pm \underbrace{\frac{\partial(y_i-R_y/2)}{\partial y_i}}_{1}\frac{\partial\phi^{\mathrm{H}}_{nlm}(\mathbf{r}_i-\mathbf{R}/2)}{\partial(y_i-R_y/2)} \\
&= \frac{\partial\phi^{\mathrm{H}}_{nlm}(\mathbf{r}_i+\mathbf{R}/2)}{\partial(y_i+R_y/2)} \pm \frac{\partial\phi^{\mathrm{H}}_{nlm}(\mathbf{r}_i-\mathbf{R}/2)}{\partial(y_i-R_y/2)} \\
&= \frac{\partial\phi^{\mathrm{H}}_{nlm}(\tilde{\mathbf{R}}^{+}_{\mathbf{i}})}{\partial\tilde{Y}^{+}_{i}} \pm \frac{\partial\phi^{\mathrm{H}}_{nlm}(\tilde{\mathbf{R}}^{-}_{\mathbf{i}})}{\partial\tilde{Y}^{-}_{i}},
\end{aligned}
$$

We can reuse closed form expressions

```
double DiTransform::lapl_phi(const Walker* walker, int particle, int q_num) {

    (void) walker;
    int sign = minusPower(q_num);

    return nucleus1->lapl_phi(walker_nucleus1, particle, q_num / 2) +
            sign * nucleus2->lapl_phi(walker_nucleus2, particle, q_num / 2);
}
```

Generalizable to N-atomic molecules?

YES.

Challenges:

- Current model breaks down around O_2.
- Need a better trial wave function.
- Multi-determinants are out of the question.
- More advanced Jastrow is out of the question.
- Expanded single particle states are gold.
- Slater orbitals are gold.

Generalizable to N-atomic molecules?

YES.

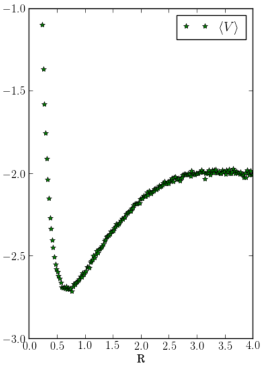
Challenges:

- Current model breaks down around O_2.
- Need a better trial wave function.
- Multi-determinants are out of the question.
- More advanced Jastrow is out of the question.
- Expanded single particle states are gold.
- Slater orbitals are gold.

Molecule	R	E_{VMC}	E_{DMC}	Expt.
H_2	1.4	-1.1551(3)	-1.1745(3)	-1.1746
$\vdots$				
O_2	2.282	-143.97(2)	-148.53(2)	-150.3268

Table: Refs. for R and Expt.: [3, 2, 1].

Parametrizations



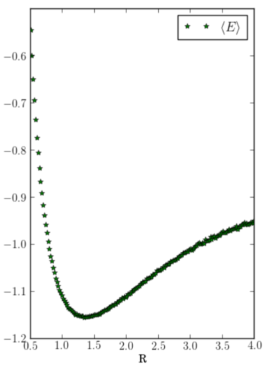


Figure: H_2

Parametrizations

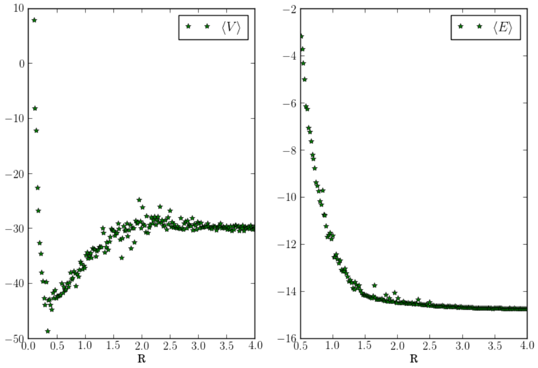


Figure: Li_2

S. Datta, S. A. Alexander, and R. L. Coldwell.
Properties of selected diatomics using variational Monte Carlo methods.
J. Chem. Phys., **120**:3642, 2004.

Claudia Filippi and C. J. Umrigar.
Multiconfiguration wave function for quantum Monte Carlo calculations of first-row diatomic molecules.
J. Chem. Phys, **105**:213, July 1996.

Moskowitz Kalos.
A new Look at Correlations in Atomic and Molecular Systems. Application of Fermion Monte Carlo Variational Method.
Int. J. Quant. Chem., **XX**:1107, 1981.