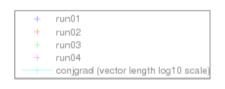


A general-purpose program structure for variational Monte-Carlo calculation

Kyrre Ness Sjøbæk 16th of April, 2009



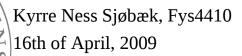


0.8

Outline

- Object-orientation and inheritance with C++
- Basic overview of program structure
 - Wavefunction classes
 - Algorithm classes
 - × Glue
- Ideas for improvement for my code
- Tips & tricks
 - * How to split a big C/C++ program into several files





Object-orientation and inheritance with C++

What is object orientation

Why and when to use object orientation

Inheritance

C++ syntax

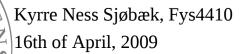


What is object orientation

- Classes = new types datatypes specialized for a task, that may perform operations
- A good way to keep state that needs to be shared between many functions
 - (alternative: Global variables, massive argument lists)
- Makes debugging and code reuse simpler:
 - Program is composed of several (mostly) independent self-contained pieces

Why and when to use object orientation (and when not to)

- Use object orientation when:
 - You can separate your code into logically separate sections
 - When writing a big, complicated program
 - Working on the same program for an extended period of time, or many collaborating with many people
- Don't use it when:
 - Writing a small "script"
 - Don't jump in and out of class methods to add two numbers
- Remember:
 - Programmers are (usually) slower than a computer
 - Comments don't make your program(ming) slower!
 - These rules are meant to be broken

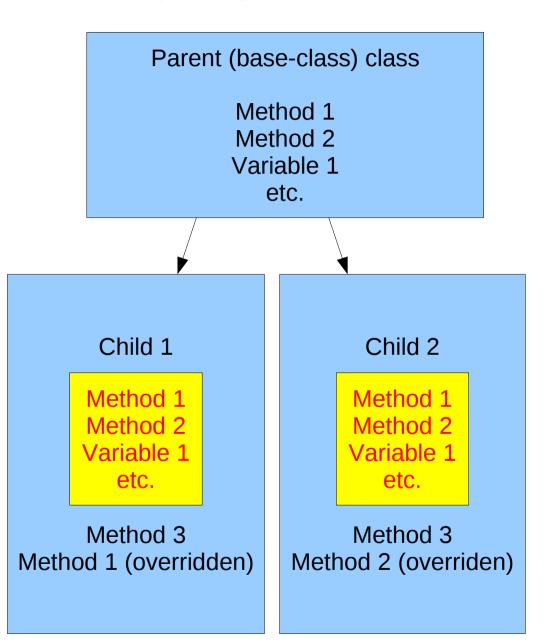


C++ syntax

```
//Header file
class myclass {
public:
    myclass();
    myclass(int arg1, double** arg2,...);
    void method1(int arg1);
    ~myclass();
protected:
    double** matrix;
private:
                                       //Implementation
    int some_private_variable;
                                       #include "header.hpp"
};
                                       myclass::myclass() {
                                           //Body of 1st constructor
                                       myclass::myclass(int arg1, void arg2, ...) {
                                           //Body of 2<sup>nd</sup> constructor
      Remember
         the ";"!
                                       void myclass::method1(int arg1) {
                                           //Body of method1
                                       myclass::~myclass() {
                                           //Body of destructor
```

Inheritance – what is it?

- You can make several new classes "inherit" old classes
- They then get copies of the methods and variables in the parent class
- In addition they may define their own methods and variables, or override methods in the base class
- You may use a parent pointer to hold any child, while accessing functionality declared in parent



C++ syntax

double myChildClass::method2(double arg1) { // Body of method2 in myChildClass

// belonging to myChildClass, and

// You may here manipulate class variables

// public/protected variables from parent

```
//Header file
class myclass {
public:
    myclass();
                                            //Called by default before
                                            //children constructors runs
    myclass(int arg1, double** arg2,...); //... Unless the children "calls"
                                            //this one explicitly
    void method1(int arg1);
    ~myclass();
protected:
    double** matrix;
private:
    int some_private_variable;
};
class myChildClass : public myclass {
public:
    myChildClass(); //Argless constructor
    myChildClass(int arg1, double** arg2, double arg3) :
        myclass(arg1, arg2), childvar1(arg3) {};
                                                              // Call base parent constructor,
                                                              // set childvar1
    double method2(double arg1);
                                                //Implementation
Private:
                                                #include "header.hpp"
    double childvar1;
};
                                                myChildClass::myChildClass() {
                                                    // Body of 1<sup>st</sup> child constructor
```



Interfaces / "abstract" classes

- An interface is a class with "undefined" method
- Serves as a template for other class to inherit
- A pointer of the interface type may then be used to access all methods & variables defined in the interface
 - You cannot have an object of an interface type
- In addition to the "purely virtual" functions, there may be "normal" helper functions
- Example: All algorithms need a method "runAlgo()".
 - Declare this in an interface for algoritms
 - Implement it in the inheriting class



```
//Header
#ifndef HEADER HPP
                                 C++ syntax
#define HEADER HPP
class myInterface {
    public:
    virtual void runAlgo() = 0;
};
class implementation : public myInterface {
                                              //Implementation
    public:
    void runAlgo();
                                              #include "header.hpp"
    double specialFunction();
                                              void implementation::runAlgo() {
};
                                                  //Implementation of runAlgo()
#endif
                                              Double implementation::specialFunction() {
                                                  //Implementation of specialFunction()
       //Usage
                                                  return 42;
       #include "header.hpp"
                                              }
       int main() {
           //Create an implementation object, use a generic myInterface pointer to store it
           myInterface* interfacePointer = new implementation();
           //Call runAlgo() in the implementation
           pointer->runAlgo();
           //Doesn't work (undefined what happens...):
           interfacePointer->specialFunction();
           //Correct: cast to implementation type first
           ((implementation*)interfacePointer)->specialFunction();
           delete interfacePointer;
```

Virtual keyword

- A warning about virtual methods: As the program has to figure out where in memory the function lives each time it is called, calls to virtual methods are a bit slower than "normal" methods.
 - Don't use virtual methods for very small methods that are called bazillions of times (but if you have a couple of calls to the math library etc. it doesn't matter)
- Virtual methods are the only methods that can be completely overridden

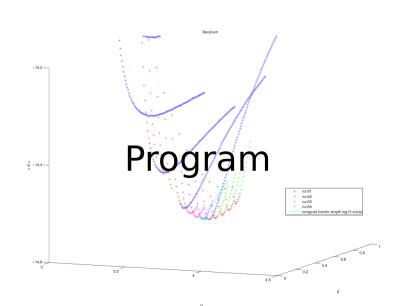
Basic overview of program structure

Description of physical system:
Wavefunction



Numerical algorithm







Wavefunction

- Represents the physical system under study
- Interface for basic operations
 - Algorithms use only the functions defined in the interface (getWf(), getRatio() etc.)
- Many implementing classes (helium1, hydrogen_1s, ...)
- Special implementation: Wavefunction_Slater
 - An interface
 - Handles "statefull" wavefunctions

Implementations neon, beryllium, ...



Vavefunctio

Kyrre Ness Sjø

16th of April, 2

S

```
Wavefunction
# num particles : int
# num params : int
- h : const double
- h2i : double
+ Wavefunction()
+ aetWf(r : double**) : double
+ local energy(r : double**) : double
+ print params()
+ get num particles(): int
+ get num params(): int
+ get partPsi over psi(r:double**):double*
# local energy kinetic numeric(r : double**) : double
# local energy electron(r : double**) : double
# local energy central(r : double**) : double
```

```
Wavefunction_Slater
# slater1 : Slater*
# slater2 : Slater*
# jastrow : Jastrow*
# particle lastmoved : int
# r new : double*
# r prev : double*
# particle_tried : int
# r now : double**
# R : double
# p2 : int
# Z : int
# WFname : char*
+ Wavefunction Slater()
+ initialSet(r : double**)
+ getWf(r : double**) : double
+ local energy(): double
+ local_energy(r : double**) : double
+ print params()
+ get_partPsi_over_psi() : double*
+ get partPsi over psi(r : double**) : double*
+ getRatio(r_new : double*, particle : int) : double1
+ getRatio() : double
+ tryMove(r_new : double*, particle : int)
+ rollback()
+ acceptMove()
+ getQuantumForce() : double**
# getDeriv slater(p : int) : double*
# local energy kinetic(): double
```

```
hydrogen 1s

    alpha : double

    numeric : bool

- Z : const int
+ hydrogen 1s(alpha : double, numeric : bool)
+ getWf(r : double**) : double
+ local_energy(r : double**) : double
+ print params()
+ get partPsi over psi(r : double**) : double*
               helium1 nocor

    alpha : double

 - Z : const int
 + helium1 nocor(alpha : double)
 + getWf(r : double**) : double
 + local energy(r : double**) : double
 + print params()
 + getAlpha(): double
 + get partPsi over psi(r : double**) : double*
                   helium1
  alpha : double
 - beta : double
 - Z : const int
 + helium1(alpha : double, beta : double)
 + getWf(r : double**) : double
 + local_energy(r : double**) : double
 + print params()
 + getAlpha(): double
 + getBeta() : double
 + get partPsi over psi(r : double**) : double*
               beryllium nocor

    alpha : double

       + beryllium nocor(alpha : double)
       + getAlpha() : double
                   beryllium

    alpha : double

    - beta : double
    + beryllium(alpha : double, beta : double)
    + getAlpha(): double
    + getBeta() : double
```

neon alpha : double

+ neon(alpha : double, beta : double) + getAlpha(): double

+ getBeta() : double

- beta : double

Jastro



+ initialSet(r : double**) + getWf(r : double**) : double + local energy(): double + local_energy(r : double**) : double + print params() + get partPsi over psi() : double* + get partPsi over psi(r : double**) : double* + getRatio(r_new : double*, particle : int) : double1 + getRatio() : double + tryMove(r new : double*, particle : int) + rollback() + acceptMove() + getQuantumForce() : double** # getDeriv slater(p : int) : double* # local energy kinetic(): double

#slater1 0..1 Slater dim : int orbitals : Orbital** matrix : double** inverse : double** orbitals_p : Orbital** orbitals_m : Orbital** h_alpha: double CGMready : bool particle : int r new : double* matrix_new : double* R : double r now : double** + Slater(dim : int, orbitals : Orbital**, r : double**) + initialUpdate(r : double**) + update() + getRatio(r_new : double*, particle : int) : double + getDet(r : double**) : double + getDeriv(particle : int) : double* + getLaplace(particle : int) : double + get partPsi over psi(): double + CGMsetup(orbitals p : Orbital**, orbitals m : Orbital**, h alpha : double) + printMatrices() + print params() det(mat : double**, d : int) : double setup matrix(orbs : Orbital**, r : double**, mat : double**)

0. orbitals orb1s Orbital alpha : double + getWf(r : double*) : double + orb1s(alpha : double) + getDeriv(r : double*) : double* + getWf(r : double*) : double + getLaplace(r : double*) : double + getDeriv(r : double*) : double* + print params() + getLaplace(r : double*) : double # calcR(r : double*) : double + print params() orb2p orb2s alpha : double alpha : double dir : int + orb2s(alpha : double) + orb2p(alpha : double, dir : int) + getWf(r : double*) : double + getWf(r : double*) : double + getDeriv(r : double*) : double* + getDeriv(r : double*) : double* + getLaplace(r : double*) : double

+ print params()

+ getLaplace(r : double*) : double

+ print_params()

√√\ Kyrre Ness Sjøba 16th of April, 200 S

Jastrow # num_particles : int # r now : double** # r inter now : double** # particle : int # r new : double* # r inter new : double* + Jastrow(num particles : int) + ~ Jastrow() + initialUpdate(r : double**) #jastrow + update() + getRatio(r : double*, particle : int) : double + getVal(r : double**) : double + get partPsi_over_psi() : double + getDeriv(particle : int) : double* + getLaplace(particle : int) : double + print_params() # r_inter(r : double**, p1 : int, p2 : int) : double # r inter(r1 : double*, r2 : double*) : double # update tempstore(r update : double*, particle : int) # flip_tempstore()

JastrowDummy

+ getRatio(r : double*, particle : int) : doubl

+ JastrowDummy(num particles : int)

+ initialUpdate(r : double**)

+ getVal(r : double**) : double

+ get_partPsi_over_psi() : double

+ getDeriv(particle : int) : double*

+ getLaplace(particle : int) : double

+ update()

+ print_params()

JastrowExp # np2 : int

factor : jasFuncExp* factor p : jasFuncExp*

factor_m : jasFuncExp* h_beta : double

CGMready: bool factors now : double** factors_new : double*

+ JastrowExp(num_particles : int, factor : jasFuncExp*)

+ ~ JastrowExp()

+ initialUpdate(r : double**)

+ update()

alpha : double

beta : double

+ getAlpha() : double

+ getBeta() : double

alpha : double

+ getAlpha(): double

+ getBeta() : double

beta : double

+ beryllium(alpha : double, beta : double)

neon

+ neon(alpha : double, beta : double)

+ getRatio(r : double*, particle : int) : double

+ getVal(r : double**) : double

+ get partPsi over psi() : double

+ getDeriv(particle : int) : double*

+ getLaplace(particle : int) : double

+ print params()

+ CGMsetup(factor p : jasFuncExp*, factor m : jasFuncExp*, h beta : double)

spinEx_decide(p1 : int, p2 : int) : bool

-factor 0..1

jasFuncExp

+ getValue(r_inter : double, spinEx : bool) : double

+ getDeriv(r1 : double*, r2 : double*, spinEx : bool) : double*

+ getLaplace(r_inter : double, spinEx : bool) : double

+ print_params()

r inter(r1 : double*, r2 : double*) : double

jasFuncExp1 beta : double

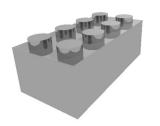
+ jasFuncExp1(beta : double)

+ getValue(r inter : double, spinEx : bool) : double

+ getDeriv(r1 : double*, r2 : double*, spinEx : bool) : double* + getLaplace(r inter : double, spinEx : bool) : double

+ print params()

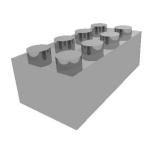
getA(spinEx : bool) : double



Numerical algorithm

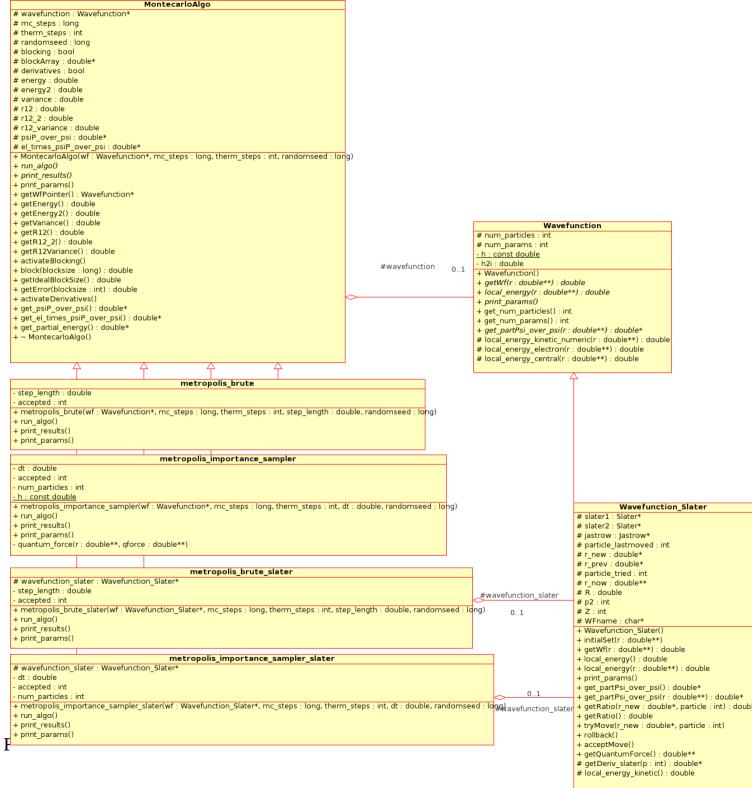
- Several possibilities for different algorithms with same purpose (simulation of the PDF, taking statistics)
- They share common operations (runAlgo() etc.)
- Common interface for these: MontecarloAlgo
- × Several implementations:

metropolis_brute, metropolis_brute_slater,
metropolis_importance_sampler,
metropolis_importance_sampler_slater



Algorithms







Glue

- Several programs that use the wavefunction and algorithm classes to do usefull things
 - Calculate energy in a set of points
 - Calculate energy and estimate error in a set of Δt
 - Use CGM to find minima
- Since MontecarloAlgo and Wavefunction / Wavefunction_Slater are interfaces, easy to change which wavefunction or algo we are working with

Ideas for improvement

- Sampler classes
- Generalization of functions to get wavefunction parameters
- Generalization of Wavefunction_Slater to handle other (non-atomic) Hamiltonians
- Implementation of real "rollback" support
- Smarter calculation of determinants of slater matrices
 - Maybe possible to have a "simple" scheme for calculating analytic Ψ_i/Ψ_i , in analogue to $\nabla\Psi/\Psi$ and

 $\nabla^2 \Psi / \Psi$?

Tips & tricks

- * How to split a big C/C++ program into several files
- * Makefile



How to split a big C/C++ program into several files

- Headers:Named .h or .hpp
- Contain definitions of functions, classes, global variables etc.
- Basic structure:
 #ifndef FILE_HPP
 #define FILE_HPP
 <stuff>
 #endif
- Often useful to put detailed comments that describes
 Junctions, variables etc. here

- Program code:Named .c or .cpp
- Contains the code that will be compiled
- #include one or more headers:
 - #include "header.hpp"
- Compilation of a single .cpp file:
 - g++ -Wall -03 -c file.cpp
 - This yields an "object" (.o) file
 - Linking of several object files:
 - g++ -Wall -03 file1.o file2.o -o progname

Makefile

- Big programs takes time to compile, and manually giving commands is error-prone
- Solution: Only compiled what is needed
- * Tool: make
- Make is controlled by a "makefile" in the directory

Makefile: Basic syntax

```
#This is a comment
```

#Definition of variables CPP = g++ -Wall -O3LONGSTRING = lib1.0 lib2.0

#Special target "all" all: program1 program2 lib

#Compile program 1 program1 : program1.o \${LONGSTRING} \${CPP} program1.0 \${LONGSTRING} -o program1 program1.o: program1.cpp \${CPP} -c program1.cpp

#Compile program 2 program2: (similar to above)

#Special target to compile targets lib1.0 lib2.0 lib: lib1.0 lib2.0

lib1.o: lib1.cpp lib2.hpp \${CPP} -c lib1.cpp lib2.o: lib2.cpp lib2.hpp \${CPP} -c lib2.cpp

Use: "make" in directory with makefile to run make with target "all"

"make program1" Run make with target "program1"

"make -C subdir target" Run make in subdirectory "subdir" with target "target"