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

Kyrre Ness Sjøbæk
16th of April, 2009

Code, slides and report may be found at:
http://folk.uio.no/kyrrens

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
  - Makefile

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

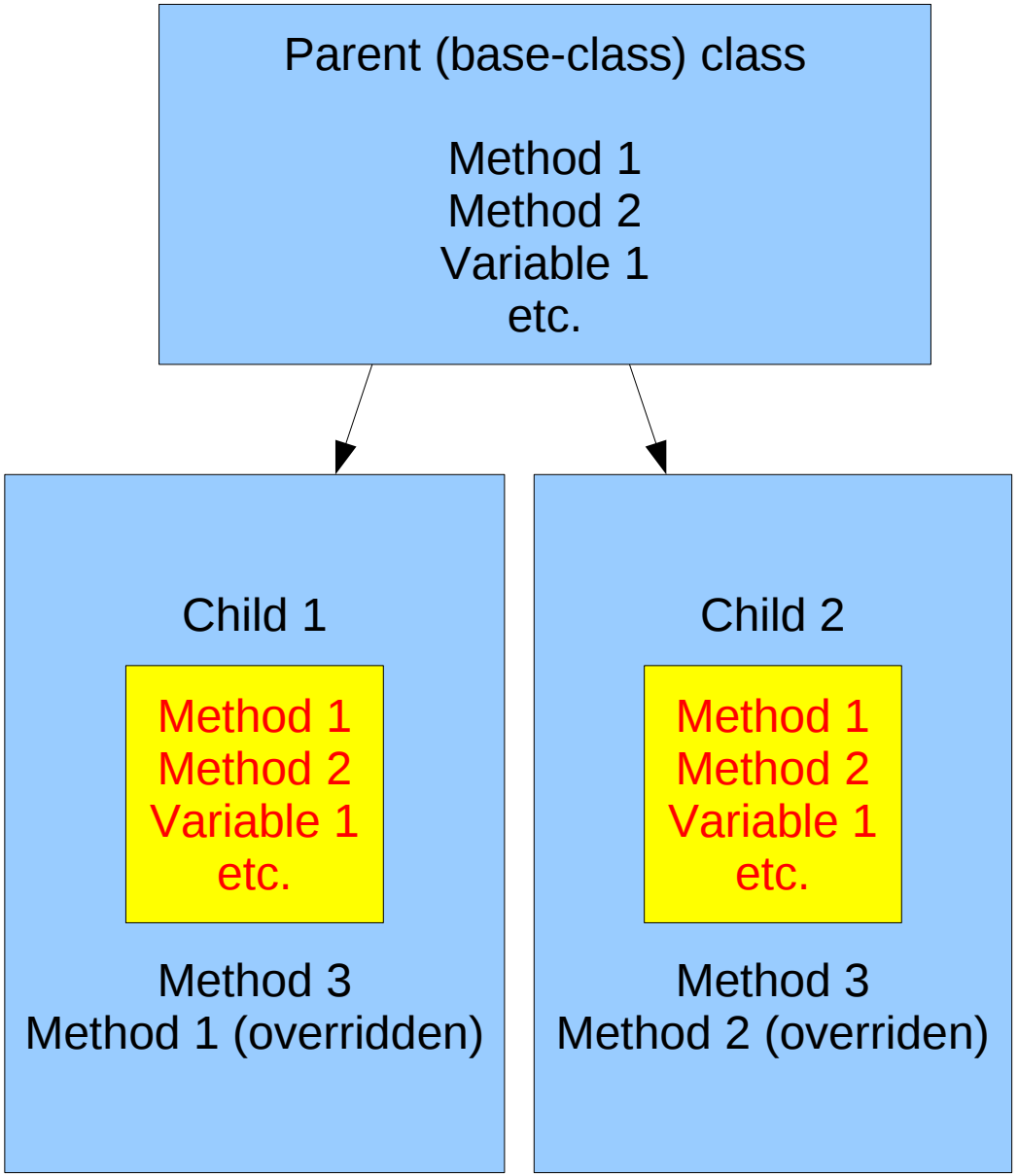
```
//Header file
class myclass {
public:
    myclass();
    myclass(int arg1, double** arg2,...);
    void method1(int arg1);
    ~myclass();
protected:
    double** matrix;
private:
    int some_private_variable;
};
```

Remember the ";"!

```
//Implementation
#include "header.hpp"

myclass::myclass() {
    //Body of 1st constructor
}
myclass::myclass(int arg1, void arg2, ...) {
    //Body of 2nd constructor
}
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



*) Method = function in a class

C++ syntax

```
//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);
Private:
    double childvar1;
};
```

```
//Implementation
#include “header.hpp”

myChildClass::myChildClass() {
    // Body of 1st child constructor
}

double myChildClass::method2(double arg1) {
    // Body of method2 in myChildClass
    // You may here manipulate class variables
    // belonging to myChildClass, and
    // public/protected variables from parent
}
```

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

C++ syntax

```
//Header
#ifndef HEADER_HPP
#define HEADER_HPP

class myInterface {
     public:
     virtual void runAlgo() = 0;
};

class implementation : public myInterface {
     public:
     void runAlgo();
     double specialFunction();
};

#endif
```

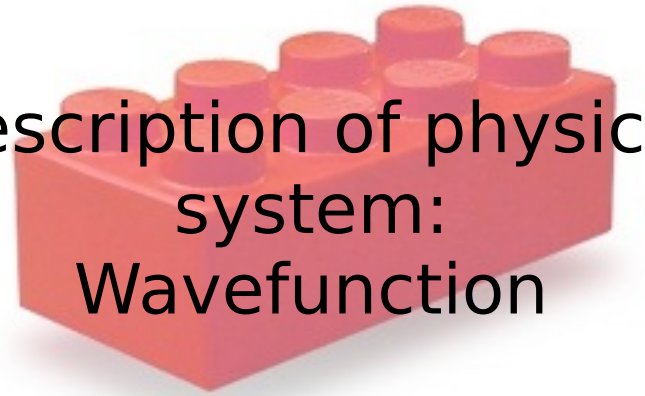
```
//Implementation
#include “header.hpp”

void implementation::runAlgo() {
     //Implementation of runAlgo()
}
Double implementation::specialFunction() {
     //Implementation of specialFunction()
     return 42;
}
```

```
//Usage
#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

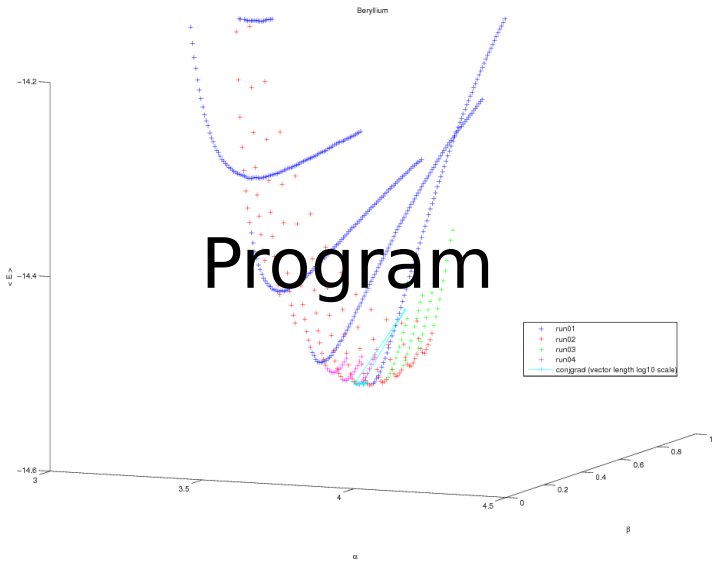
+

Numerical algorithm

+

Some code that “glue” them together

=

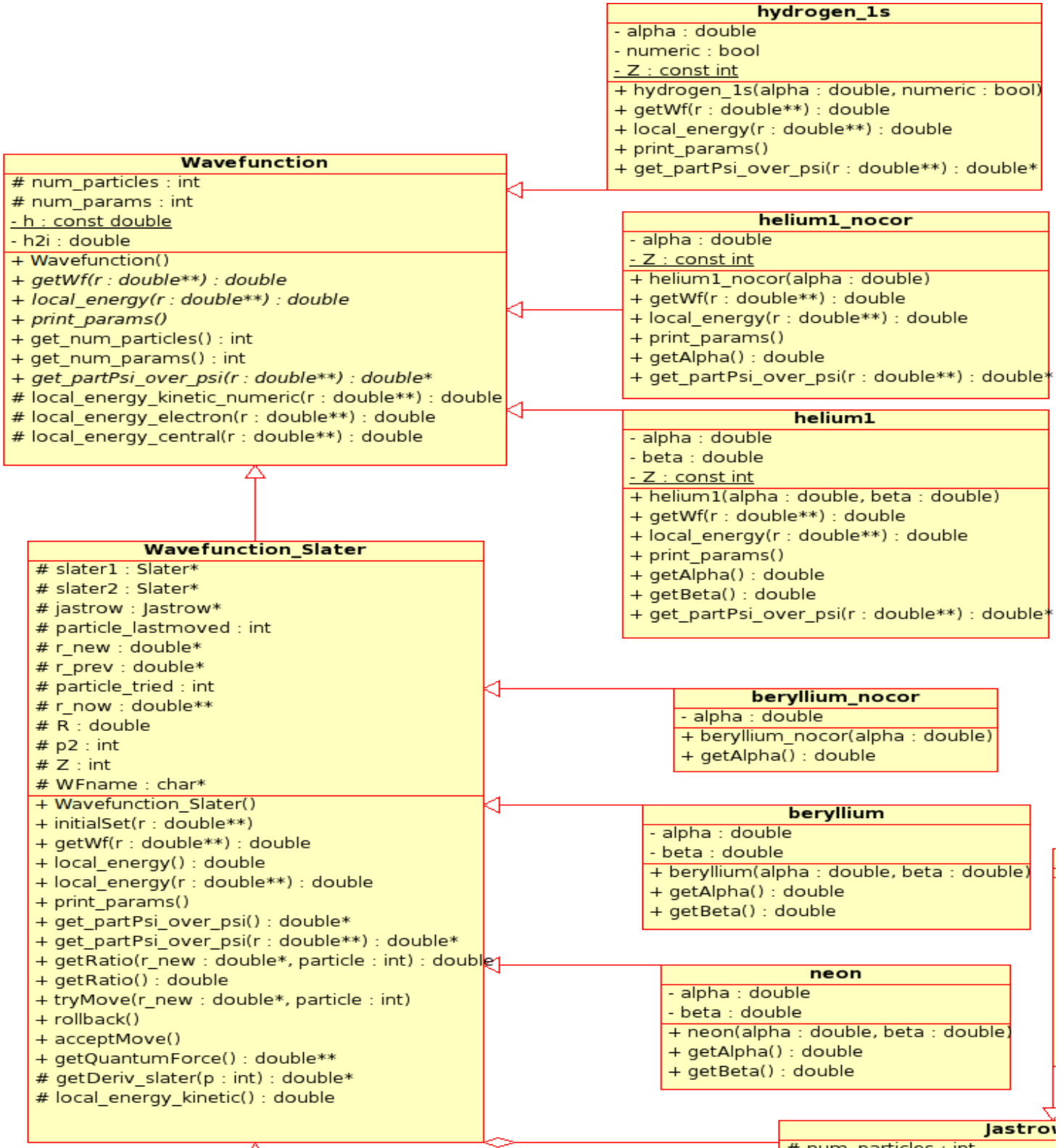


Program

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`, ...

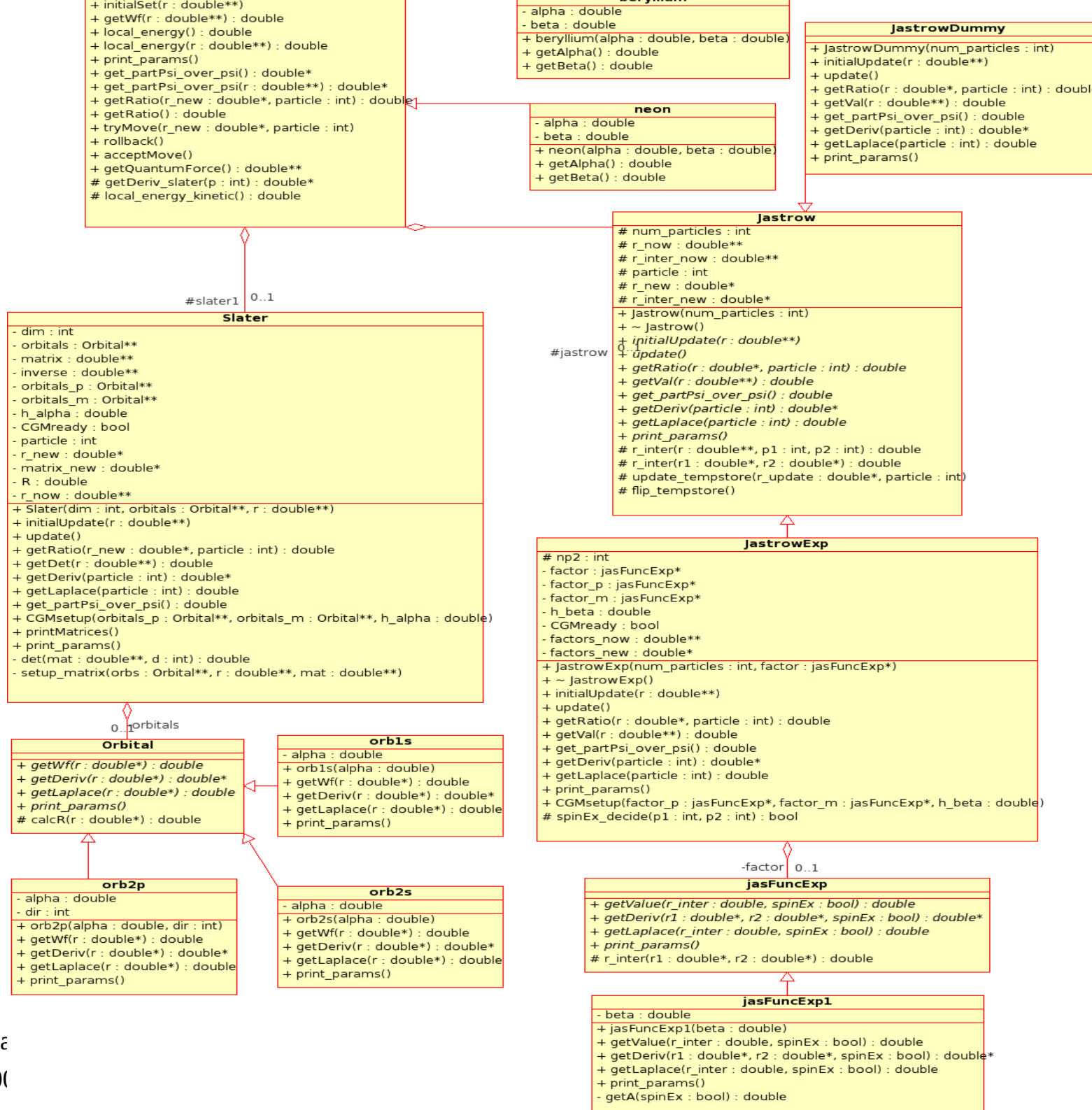
Wavefunction



Kyrre Ness Sjø
16th of April, 2

Wavefunction

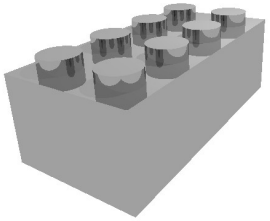
Kyrre Ness Sjøba
16th of April, 20(

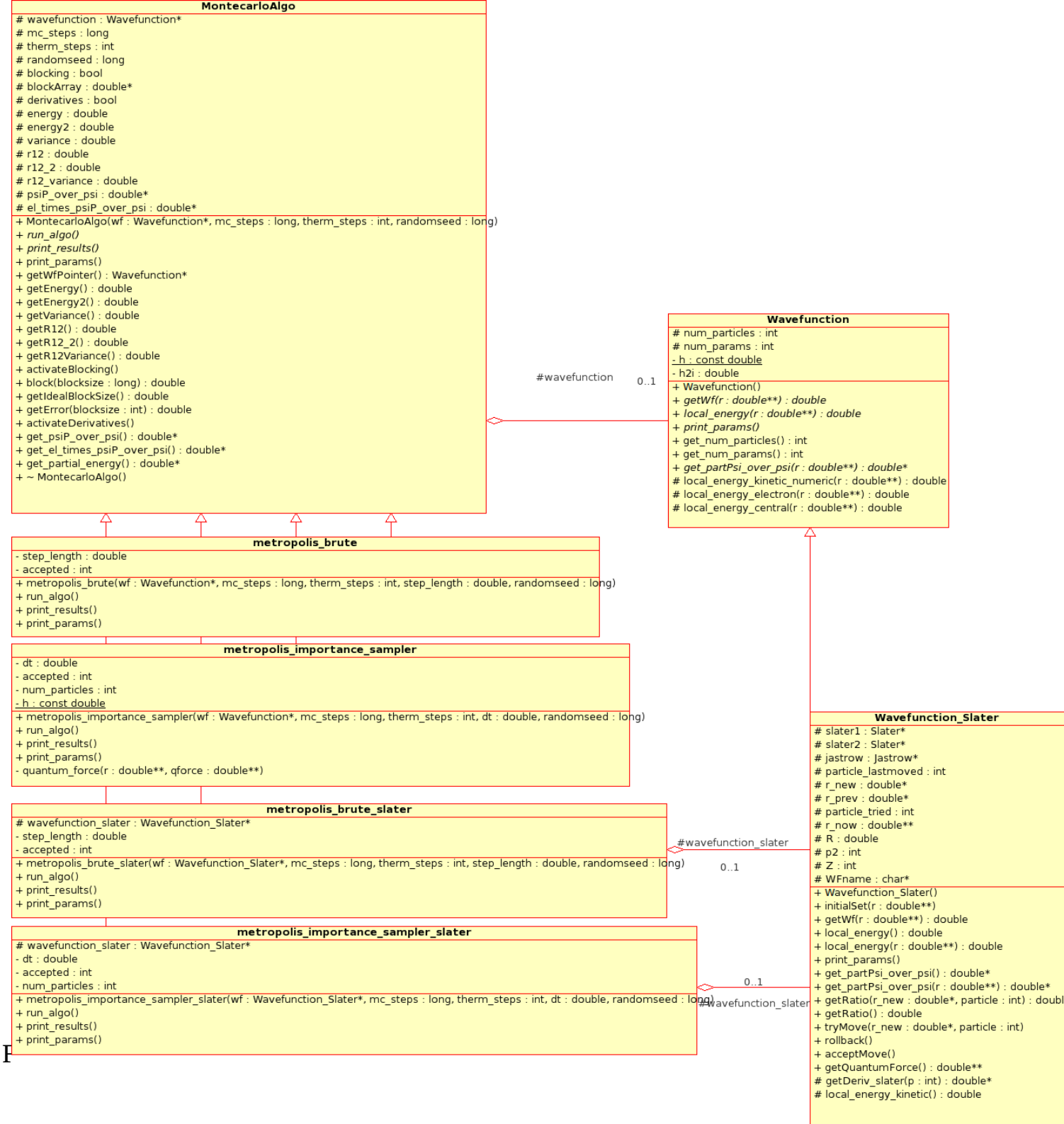


Numerical algorithm

- 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`

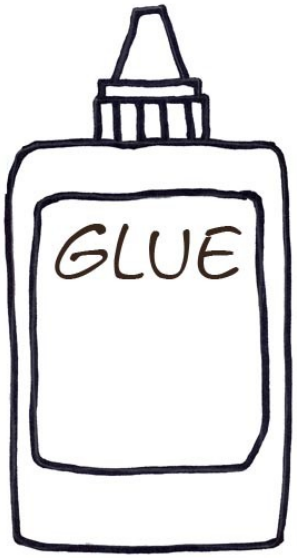
Algorithms





Kyrre Ness Sjøbæk, P

16th of April, 2009



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, 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 functions, 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 -O3 -c file.cpp
  ```

  This yields an “object” (`.o`) file
- Linking of several object files:

  ```
  g++ -Wall -O3 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 -O3
LONGSTRING = lib1.o lib2.o

#Special target “all”
all : program1 program2 lib

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

#Compile program 2
program2 :
	(similar to above)

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

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”