Compiling AGAMA with MS Visual Studio

Here I explain how to get Vasiliev's AGAMA package (Action-based Galaxy Modelling Framework) running with Microsoft's freely-distributed 'Visual Studio' compiler. This is essential if you want to run AGAMA under Windows because it is impossible to compile either Python or the Gnu Scientific Library with the Gnu compilers, and a single compiler must compile all components of a code.

Instructions for installing GSL can be found at

https://solarianprogrammer.com/2020/01/26/getting-started-gsl-gnu-scientific-library-windows-macos-linux/

In a Linux/Gnu environment, AGAMA is embodied in shared library `agama.so`. Under VS it is contained in two files `agama.lib` and `agama.dll`. The former is what the linker uses, while the latter is required at runtime, so has to lie on your `path`. I have the following file `vscl.bat` on my `path` so I can compile and link a program `testit.cpp` that uses AGAMA by typing `vscl testit`

```
@echo off
cl.exe /c /openmp /I\u\c\agama\agama -EHsc %1.cpp
@if not %ERRORLEVEL% == 0 (goto end)
link %1.obj agama.lib /LIBPATH:\u\c\agama\agama\x64\release /NOIMPLIB
del %1.obj
del %1.exp
:end
```

What follows `/I` in the second line is the path to my AGAMA source code & what follows `LIBPATH:` in the fourth line is the path to `agama.lib`. That's where the VS IDE put it: I used the VS IDE to compile AGAMA rather than hacking the file `makefile.local` distributed with AGAMA and using `make`. The IDE handles access to the Gnu Scientific Library and Python if they were installed using the `vcpkg` utility. The last `del` command is necessary because VS insists on producing an 'export' library file `.exp` even when told to make an executable.

Installing AGAMA

Untar everything into your source directory and start the Visual Studio IDE and via the File tab open `agama.sln` (sln stands for 'solution') within the source directory. Then provided you've installed the Gnu Scientific Library within VS, you should be able to 'build agama' from within the 'build' tab of the VS IDL. VS is a bossy compiler so there will be zillions of warnings. It will even declare an error on correct code, such one using `fopen` rather than `fopen_s`. It's easier to identify any errors if you use the drop-down menu at the top centre of the error list to change to 'Build only'.

You may need to set some compiler options. At the bottom of the Project tab, click on Properties. Then click on C/C++ and by clicking on open MP support get Yes (/openmp)

After building AGAMA with Visual Studio Community 2019, Version 16.9.2 I tested the procedure for installing AGAMA by transferring the code to an older machine running Version 16.4.2. I encountered two problems: (i) min was said not to be a member of sdt – I fixed the problem by adding `#include <algorithm>` to `math_base.h`; (ii) the linker complained about twoPhase template instantiation. This problem was fixed by clicking Project→Properties→C/C++→Command line and typing in `/Zc:twoPhase-` at the bottom. This click sequence could also be used to establish what the compiler and linker flags should be if you prefer to hack `makefile.local` rather than using the IDE.

Among the many outputs of an error-free compilation will be `agama.dll`, which you need to copy to a folder that lies on your path, and `agama.lib`. The outputs will be in `x64\release` or `x64\debug` if you've set VS to debugging (top left centre of IDE). Here I'm assuming that you have a 64-bit machine and that you will be running the code from the 'x64 Native Tools

Command Prompt' that should be in your Visual Studio folder on your start menu. If you have a 32-bit machine, change the IDE from 'x64' to 'x86' at the top.

The following lines in a makefile

```
ASRC = \u\c\agama\agama
ALIB  = \u\c\agama\agama\x64\release\agama.lib
ADLL  = \u\c\agama\agama\x64\release\agama.dll
testit.exe: testit.cpp  $(ALIB) $(ADLL) other.obj
cl /openmp /I$(ASRC) -EHsc testit.cpp /link /out:testit.exe NOIMPLIB \
other.obj \libs\press.lib $(ALIB)
```

will update testit.exe when any of `testit.cpp`, `other.obj` or AGAMA is changed. `testit` will be linked to AGAMA, other.obj and the static library `\libs\press.lib`. The words before `/link` are instructions to the compiler, while those after are instructions to the linker.

For some reason VS doesn't put any symbols from the dll it is constructing into the `.lib` file unless explicitly instructed to do so. So any symbol you want to reference from code that's not in the AGAMA library has to have its name decorated by `__declspec(dllexport)`. I've done this for most routines in the AGAMA library by adding

```
#define EXP __declspec(dllexport)
```

in header files and then placing `EXP` at appropriate places. In the case of a class, one writes

```
class EXP some_class{..
```

which will cause VS to put into the `.lib` file objects and methods defined in the class definition. When a method's code is given outside the class definition, its name needs another `EXP`:

```
EXP double some_class::some_method(int i,..){..
```

even though the method was of course declared when the class was defined. Failure to decorate the name when the code is given generates a 'redefinition with different linkage' error. If the compiler can figure out the need for decoration, why the hell can't it add it without bothering me?

The syntax for a `struct` mirrors that of a `class`

```
struct EXP some_struct{..
```

Functions unconnected with a class are handled like methods define outside a class

```
EXP double do_something(double x,double y){..
```

The older of my versions of Visual Studio complained when the names of symbols in internal namespaces were decorated. Since these symbols should only be referenced from within the library, this should not be a problem. Anyway don't add `EXP` within internal namespaces.

If you reference a symbol in the library that's not had its name thus decorated, it will be missing from the `.lib` file and the linker will report 'unresolved symbols'. I haven't put `EXP`s in the `raga` files or `py_wrapper` because I haven't used these bits of code. I have run all the `test_xx` programs in the distributed `tests` folder and checked that the results are comparable to those obtained (on a different machine) with Gnu.

James Binney May 2021

Implementing the Python wrappers on Windows

Access to much of the AGAMA code from Python has been achieved via the PYBIND11 package. If you want to use our wrappers, your first step should be to install PYBIND11 by typing at a command prompt

```
pip install PYBIND11
```

Python responds to `import xx` by following the `PYTHONPATH` and there looking for a folder xx. If it finds this folder, it tries to open `__init__.py` within it. If it doesn't find the folder, it tries to open `xx.pyd` in the `PYTHONPATH`. So after installing PYBIND11, create a directory `agama` in the `PYTHONPATH` and copy into it our file `__init__.py`, which contains

```
import os
os.add_dll_directory(‘‘h:/u/c/agama/agama/x64/release’’)
from Py_agama import *
```

edit the bit in quotes to the path to the location of `agama.dll` and `gsl.dll` on your machine (if they are in separate folders, use two `os.add_dll` statements).

Copy our files Py_agama.cpp and Py_agama.sln into a folder and start a new project in VS by navigating to this folder and clicking on Py_agama.sln. Now build the solution. This should yield `Py_agama.pyd` in the `x64/release` folder. Copy this to the folder pointed to by PYTHONPATH and you are all set.

James Binney & Tom Wright June 2025