Instrumentum Project Overview

Daniel Stubbs

October 17, 2021*

This document is devoted to a discussion of the structure and use of the Instrumentum software, which consists of approximately 5,600 lines of C++ code. The project's name comes from the Latin noun *instrumentum*, meaning tool, resource or equipment. The Instrumentum source code is available from the site `www.github.com/stubbsda/instrumentum` and is licensed under version 3.0 of the GNU public license (GPL).

As the class hierarchy which is shown in Figure 1 illustrates, the principal classes of the Instrumentum project are `Grid` and `Molecule`. The `Molecular_Assembler` class handles most of the program's file operations, including reading the run-time parameters, creating the SQLite database file (if necessary) as well as the multi-threading for this software. This thread-

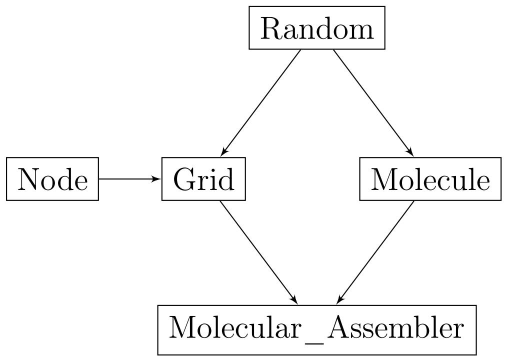

Figure 1: Instrumentum class hierarchy, where a directed edge indicates the target class makes use of the source class via composition

ing is carried out using the C++14 standard, with one particular method,

*Corresponding to version 1.1.1 of the Instrumentum software.

`run`, of the `Molecular_Assembler` class used to initialize the parallel execution. It is this method which creates instances of the `Grid` and `Molecule` classes to build the hydrocarbon skeleton of the small organic molecule and then subject it to a sequence of operations such as ring aromatization, heteroatom substitution, double bond creation and so forth in order to enhance its fit with the pharmacophore and its synthetic feasibility. The `Random` class provides the needed pseudo-random number generation needed by the main classes to create a great diversity of molecular structures while the `Node` class represents the vertices that are assembled into a diamond-like network by the `Grid` class for the hydrocarbon skeleton of the molecule.

The Instrumentum program has a large collection of run-time parameters which are passed to the program by its unique argument, the name of an XML file containing them. A sample parameter file, `parameters.xml`, is included with the software and can be modified using any text editor. A further option is the use of an included Python 3 script, `editor.py`, which makes use of the Tkinter library to provide a graphical interface for creating and modifying such an XML parameter file. The run-time parameters used for creating a given set of molecules are recorded in the same SQLite database where the molecules themselves are written, so that they can be reused for the generation of additional molecules using the same parameters if they have shown themselves to be particularly fructuous.

The available run-time parameters are shown in Table 1, coloured according to their nature. Black indicates the global parameters, orange those used for the molecular assembly, blue the pharmacophore parameters, green the parameters which control the hydrocarbon skeleton, red the rationalization of this skeleton and finally light blue for the desaturation parameters. The Boolean parameters should be set to the value 1 (true) or 0 (false) while the percentage parameters must lie between zero and one. Most of the numerical parameters are dimensionless, the only exceptions being `BondLength` and `PharmacophoreRadius` which are expressed in angstroms. The `DatabaseFile` parameter is the name of the SQLite file to which the program's output will be written, in the current working directory; if the database doesn't exist it will be created. The number of threads must be positive and should be equal to the number of CPU cores on the computer where the software is executed; the number of molecules to be created must also be positive but need not be an integer multiple of the number of threads. As the goal of the Instrumentum software is to create *small* organic molecules, consisting of no more than forty to fifty heavy atoms, the default value of the

Name	*Type*	*Default Value*
NumberMolecules	`long`	50000
DatabaseFile	`string`	NULL
BondLength	`double`	1.52
GridSize	`int`	17
NumberThreads	`int`	1
NumberInitial	`int`	3
NumberSecondary	`int`	3
NumberPath	`int`	3
NumberRationalize	`int`	3
NumberDesaturate	`int`	1
PharamcophoreRadius	`double`	3.5
NumberPharmacophores	`int`	3
InitialPercentage	`double`	0.5
MaximumAttempts	`int`	100
MaximumSecondary	`int`	100
QuaternaryCarbonAtoms	`int`	2
NumberRings	`int`	4
FourRingCarbonAtoms	`int`	0
PharmacophoreHardening	`bool`	1
PercentMethyl	`double`	0.4
MinimumRings	`int`	1
MaximumRings	`int`	6
StripAxialMethyls	`bool`	1
CreateFiveMemberRings	`bool`	1
CreateExotic	`bool`	0
CreateDoubleBonds	`bool`	1
CreateTripleBonds	`bool`	0
SubstituteOxygen	`bool`	1
SubstituteNitrogen	`bool`	1
SubstituteSulfur	`bool`	0
SubstituteFunctionalGroups	`bool`	0

Table 1: Run-Time Parameters

grid size, $17 \times 17 \times 9$ should be sufficient. This parameter sets the size of the grid in the x and y dimensions, while the z dimension is set to $\lfloor N/2 \rfloor + q$ where N is the value of the grid size and $q \equiv N \pmod{\lfloor N/2 \rfloor}$. Practically all of the integer parameters must be positive, though a handful may be zero (e.g. `FourRingCarbonAtoms` or `MinimumRings`), and there are such common sense restrictions as the requirement that the maximum number of rings allowed must not be less than the minimum number of rings.

Once the parameter file has been read, the SQLite database created (if necessary) and the run-time parameters written to it, the `assemble` method of the `Molecular_Assembler` class will then create the threads, each of which creates its own instance of the `Grid` and `Molecule` classes. Each of these has its own instantiation of the `Random` class, with the random number seed initialized to the product of the current time[1] and the thread index, starting from one. Each thread executes the `run` method of the `Molecular_Assembler` class, where a nested set of loops begins with a diamond grid of carbon atoms that are gradually sculpted by removing atoms while keeping the pharmacophore nodes connected to this pure carbon skeleton, then rationalized and finally the hydrogen atoms are added. At this point the current state of the grid is written to an instance of the `Molecule` class and then desaturated. If these desaturation operations are successful, the resulting molecule is written to a binary disk file unique to the thread. When this `run` method has created its share of the total number of molecules to be created the thread exits. Once all the threads have exited, the program will write the contents of these binary disk files to the SQLite database, which is a fairly time-consuming step in part due to its serial nature, though a high-speed storage medium like a solid-state disk can alleviate this to some extent.

The program's makefile should work without any need for modification on any modern Unix-based computer, assuming that it has a C++ compiler supporting the C++11 standard and the PugiXML and SQLite development libraries installed. By default the compiler is assumed to be GNU C++ compiler (`g++`) but this can easily be changed in the makefile if necessary. The only compilation flags used by the Instrumentum software are `VERBOSE`, which causes a large amount of diagnostic information to be written to the terminal, and `DEBUG`, which will ensure that the software performs a series of verifications and safety checks on various intermediate stages of the program

[1]Measured by seconds elapsed since the start of the Unix era, January 1, 1970

execution. Both flags are normally only of use when debugging or testing the software and so should not be used in production runs. A set of fairly aggressive optimization flags are available in the makefile, with the collective name `OPT`, that are suited to such production runs.

Once the Instrumentum program has been used to create a large SQLite database of molecules, the Python 3 script `post-processing.py` can be used to optimize the geometric structure of the molecules so that it is more realistic than the diamond grid used by the C++ code, as well as calculating the synthetic feasibility of the molecule assuming a program for this is available. The script runs inside an infinite loop, waking every three minutes to check if there are any new molecules in the database; if so, it uses the Python module PyBel — a Python wrapper for the OpenBabel software — to optimize the geometry of these molecules, one after another[2], performing 2500 iterations with the MMFF94 energy model, followed by the computation of the synthetic feasibility, a number between zero and one, of the molecule with its optimized geometry. The only argument to the Python script should be the name of the SQLite database file, assumed to be in the same directory as the Python script itself.

[2]Note that a database which supports row-level locking could perform these operations in parallel.