*ISPRS Int. J. Geo-Inf.* **2013**, *2*, 201-219; doi:10.3390/ijgi2010201

OPEN ACCESS

ISPRS International Journal of **Geo-Information**

**ISSN 2220-9964**

www.mdpi.com/journal/ijgi

*Article*

# Pygrass: An Object Oriented Python Application Programming Interface (API) for Geographic Resources Analysis Support System (GRASS) Geographic Information System (GIS)

**Pietro Zambelli [1,*], Sören Gebbert [2] and Marco Ciolli [1]**

[1] Department of Civil, Environmental and Mechanical Engineering, University of Trento, via Mesiano 77, Trento 38123, Italy; E-Mail: marco.ciolli@ing.unitn.it

[2] Thünen Institute of Climate-Smart Agriculture, Bundesallee 50, Braunschweig 38116, Germany; E-Mail: soerengebbert@googlemail.com

* Author to whom correspondence should be addressed; E-Mail: pietro.zambelli@ing.unitn.it; Tel.: +39-46-128-2696.

*Received: 1 January 2013; in revised form: 21 January 2013 / Accepted: 21 February 2013 / Published: 11 March 2013*

---

**Abstract:** PyGRASS is an object-oriented Python Application Programming Interface (API) for Geographic Resources Analysis Support System (GRASS) Geographic Information System (GIS), a powerful open source GIS widely used in academia, commercial settings and governmental agencies. We present the architecture of the PyGRASS library, covering interfaces to GRASS modules, vector and raster data, with a focus on the new capabilities that it provides to GRASS users and developers. Our design concept of the module interface allows the direct linking of inputs and outputs of GRASS modules to create process chains, including compatibility checks, process control and error handling. The module interface was designed to be easily extended to work with remote processing services (Web Processing Service (WPS), Web Service Definition Language (WSDL)/Simple Object Access Protocol (SOAP)). The new object-oriented Python programming API introduces an abstract layer that opens the possibility to use and access transparently the efficient raster and vector functions of GRASS that are implemented in C. The design goal was to provide an easy to use, but powerful, Python interface for users and developers who are not familiar with the programming language C and with the GRASS C-API. We demonstrate the capabilities, scalability and performance of PyGRASS with several dedicated tests and benchmarks. We compare and discuss the results of the benchmarks with dedicated C implementations.

**Keywords:** GRASS; python; GIS processing

## 1. Introduction

Geographic Information Systems (GIS) have the capabilityto integrate heterogeneous digital data, giving the opportunity to public administration, industry and research to provide basic and advanced data analysis and modeling for a wide range of disciplines [1]. The Geographic Resources Analysis Support System (GRASS) supports the creation, modification and processing of 2D and 3D raster and vector layers. It provides a topological vector model and true three dimensional coordinates for vector features. GRASS is characterized by stability, an efficient application programming interface (API) written in C, and a large number of GIS functions and modules [2]. GRASS provides a large number of models and algorithms that, after substantial testing and trouble shooting, have proven to be very reliable. Its capabilities to process geographical information have been testified by many research and technical papers [3–14].

GRASS GIS has a modular design. The core functionalities are implemented in shared libraries using the programming language C and can be accessed via the GRASS C-API. This API provides read and write access to raster, 3D raster and vector data, as well as the handling of projection information, spatial and attribute database management, spline interpolation, mathematical and numerical functionalities and visualization functionalities; see Table 1. Spatial algorithms and models are implemented as small stand-alone programs, called modules, that make use of the C-API. The implementation of GRASS modules follows the UNIX concept. Hence, each module in GRASS has a dedicated purpose and is efficiently implemented. Modules can be combined, similar to the UNIX tool concept. Since the early days of GRASS in the 80s, the UNIX shell was used to combine GRASS modules and UNIX tools to script repetitive tasks and to implement complex spatial analysis and processing algorithms. This concept results in a large amount of over 400 modules. Most of them are implemented in C. A sufficient amount is implemented as scripts using either POSIX (Portable Operating System Interface; defines a standard operating system interface and environment, including a command interpreter (or shell), and common utility programs to support applications portability at the source code level. Scripts are POSIX-based until version 6 of GRASS GIS.) or Python, as in the latest development version 7 of GRASS.

Many GIS software have chosen Python as a main language for users (see Table 2), because it is available on many platforms, and it seems to be a good compromise between simplicity (syntax, low learning curve), flexibility (multi-paradigm programming) and power (due to rich scientific libraries). GRASS developers have chosen Python to replace POSIX for scripting modules [2]. For this purpose, a Python scripting library was implemented, providing a Python API to the GRASS modules. However, the current Python scripting API does not provide any further improvement to the POSIX approach than managing process chains using the standard Python library.

**Table 1.** GRASS C API consists of 2,698 C functions that are available through the ctypes library, divided into 10 different fields. This data is derived from the official GRASS source code.

| *ctypes* | n. funcs | n. structs | n. vars |
|---|---|---|---|
| gis | 501 | 20 | 175 |
| raster | 372 | 26 | 37 |
| vector | 344 | 66 | 147 |
| dbmi | 333 | 19 | 100 |
| ogsf | 331 | 33 | 151 |
| raster3d | 245 | 20 | 52 |
| gmath | 130 | 1 | 21 |
| display | 120 | 15 | 21 |
| imagery | 100 | 15 | 16 |
| nviz | 82 | 8 | 30 |
| date | 63 | 1 | 17 |
| vedit | 22 | 49 | 43 |
| cluster | 19 | 3 | 10 |
| stats | 19 | 0 | 10 |
| proj | 17 | 9 | 16 |
| arraystats | 0 | 1 | 10 |
| | 2,698 | 286 | 856 |

**Table 2.** Table with a comparison of the most used GIS software.

| Software | write | script | license | use | OS | Web site |
|---|---|---|---|---|---|---|
| ArcGIS | C++ | python | proprietary | desktop | Windows | http://www.esri.com/software/arcgis |
| AutoCAD Map | C/C++ | AutoLisp | proprietary | desktop | Windows | http://usa.autodesk.com/autocad-map-3d |
| Geoserver | java | python(dev) | GPL | server | Windows, Mac, Linux | http://geoserver.org |
| GRASS7 | C | python | GPL | desktop | Windows, Mac, Linux | http://grass.osgeo.org |
| gvSIG | Java | jython | GPL | desktop | Windows, Mac, Linux | http://www.gvsig.org |
| IDRISI | COM | python | proprietary | desktop | Windows | http://clarklabs.org |
| ILWIS | C++ | python | GPL | desktop | Windows | http://www.ilwis.org |
| Geomedia | C/C++ | python | proprietary | desktop | Windows | http://www.intergraph.com |
| MapInfo | C/Basic | MapBasic, python | proprietary | desktop | Windows | http://www.mapinfo.com |
| Mapserver | C/C++ | python | X/MIT | server | Windows, Mac, Linux | http://www.mapserver.org |
| QGIS | C++ | python | GPL | desktop | Windows, Mac, Linux | http://qgis.org |
| Saga-GIS | C++ | python | GPL | desktop | Windows, Mac, Linux | http://www.saga-gis.org |
| Udig | Java | groovy | LGPL | desktop | Windows, Mac, Linux | http://udig.refractions.net |

Most of the GIS software and tools provide a large number of high-level algorithms to cover different GIS processing needs. Few GIS open the capabilities to the users to access the lower functionalities, such as iteration between the geometry features of a vector map, or iterate row by row to a raster map using a higher level language.

GRASS modules must be implemented in C to access the low level functionality. To overcome this limitation and to reach a broader development community, a ctypes interface was introduced to GRASS version 7. This interface allows access to the low level GRASS C-API in Python. However, the creation of new modules writing in C or using the C API with Python through the ctypes interface is not a trivial task and is generally a very time-consuming activity. This happens because the writer must be a competent C programmer (manage the computer memory, work with pointers, *etc.*) and because of how the GRASS library works internally. Hence, an intensive study of the large GRASS C-API is required.

The goal of this work is to implement an intuitive and easy to use object-oriented layer around the GRASS C-API, hiding its complexity, but providing a more powerful development environment to solve complex GIS data analysis and model problems. An additional task is the replacement of parts from the existing Python script API with more efficient and powerful object-oriented approaches. In this way, we can provide access to the capability of the C-API of GRASS for power users and geo-scientists that are not familiar with C and the C-API of GRASS.

The idea of PyGRASS was born from the experience of the authors who wished to expand GRASS capabilities implementing a tool that gives the freedom to approach the GIS problem from a different perspective, opening the software developer's approach to the GIS users and trying to maintain relative simplicity. The PyGRASS library provides a simple, object-oriented higher level interface that transforms each GRASS module into an object by interpreting its XML interface description, trying to simplify the syntax and enforcing the script activity. The object-oriented layer around the GRASS C-API, PyGRASS, implements several classes to access vector and raster data, covering several complex features that are only available in the GRASS C-API, like support for the vector topology or the use of the raster cache for fast random read and write access. In addition, PyGRASS simplifies the interoperability with all related geospatial software and tools provided by a Python interface.

The development of PyGRASS has been sponsored by Google's Summer of Code program (2012), and the code produced has been integrated in the latest development version, GRASS7.

## 2. Methodology

The PyGRASS library is written in Python and makes use of modules from the Python standard library [15], like: *sys*, *fnmatch*, *collections*, *sqlite3*, as well as from the third party Python library, NumPy [16]. NumPy is a package for scientific computing. It is already a dependence of GRASS. An optional library is *psycopg2* [17], which is used to interface the PyGRASS vector attribute handling with the PostgreSQL database.

The PyGRASS library was developed taking into account four main aspects:

- *consistency*—the library shall adhere to norms and architecture commonly found in both Python and GRASS, in order to avoid confusion for users who are only familiar with one of the above;
- *simplicity*—the library must be simple and intuitive, without hiding access to the lower level functionality, indeed, providing a seamless user experience between the low level C-API of GRASS with a higher level object-oriented Python approach;
- *flexibility*—the library must be flexible, both allowing the use of existing GRASS modules and giving to each user the freedom to implement his own logic, using more detailed and fine-grained programming tools;
- *performance*—the library must be fast, considering both the development and the CPU time. GRASS C API functions are heavily used by *PyGRASS* every time that it is possible.

The library is split in two parts: the first is more related to the script activity and the GRASS modules; the second is focused on the programming aspects and the C API of GRASS.

To improve the existing script API of GRASS, PyGRASS considers each GRASS module as an object with input parameters, output parameters and flags. When the object is "instantiate", the

`Module` class parses the XML interface description generated from the GRASS modules through the `--interface-description` flag to know which parameters and flags are defined. For each parameter, the metadata is analyzed. The metadata specifies if a parameter is required or optional, if it is an input or an output, what type it is (raster, vector, string, float, *etc.*) and many more. This information allows the class to check the correctness of the parameters and provides the capability to suggest the correct ones. The identification of inputs and outputs allows the implementation of process chains. The interface design of this class was chosen to support the implementation of local and remote process execution services, which may be added in the future. To implement an interface to a Web Processing Service (WPS), the definition of complex inputs, complex outputs and literals must be known to generate the XML execute request. The same is true for remote process execution services based on WSDL/SOAP. The `Module` class provides all required module specific information by design.

The current Python script API defines several functions to manage the GRASS module: the `make_command` returns a list of strings with the command options from a dictionary of keys and values. The `start_command` is a GRASS-oriented interface to `subprocess.Popen` (a module process creation and management provided by the Python standard library), that internally uses the `make_command` function. All the other `run/pipe/feed/read/write/parse` command functions are specialized wrappers of the `start_command`.

The `Module` class of PyGRASS gathers all these features in a single object, connecting directly the inputs and the outputs of GRASS modules; see Listing 1 in the Appendix.

The PyGRASS `Module` class simplifies the Python syntax as much as possible to be competitive with the POSIX module interface. It supports backward compatibility syntax and enhances the API to provide a tool that manages the users errors and returns the list of the valid options. Moreover, the PyGRASS library gives the capability to pass text to a command as input (*stdin*), to catch the text output (*stdout*) and the error message (*stderr*) of a command. Finally, PYGRASS allows users to manage (*i.e.*, terminate, kill, wait) the process.

The PyGRASS library introduces an Object-Oriented (OO) Python API to GRASS, which implements for each GIS/GRASS entity one or more classes. The classes use the C structures and functions through the existing *ctypes* interface. *Ctypes* is a Python library; it provides C compatible data types and allows calling functions in DLLs or shared libraries. It can be used to wrap these libraries in pure Python. Our higher level Python interface uses *ctypes* to integrate the underlying GRASS C-API structures and functions in an object-oriented framework, but at the same time, trying to respect the GRASS work-flow and nomenclature to conform with the C-API. The object-specific *ctypes* pointer to the underlying C structures are available under the attribute name that starts with `c_*` . This allows the user to access the lower level GRASS C-API structures directly using the *ctypes* interface. These classes allow one to face the problem to be confronted in a more abstract way. A high-level object-oriented approach can help users to face the problem, even if they are not familiar with the implementation details of the C-API level, speeding up the design, writing, prototyping and debugging phase.

*Architecture of the Library*

The PyGRASS library follows the main GRASS structure and is divided in four parts. Each part implements a set of dedicated classes. See Figures 1 and 2 for a general overview of the library.

**Figure 1.** Module, Raster and GIS classes.

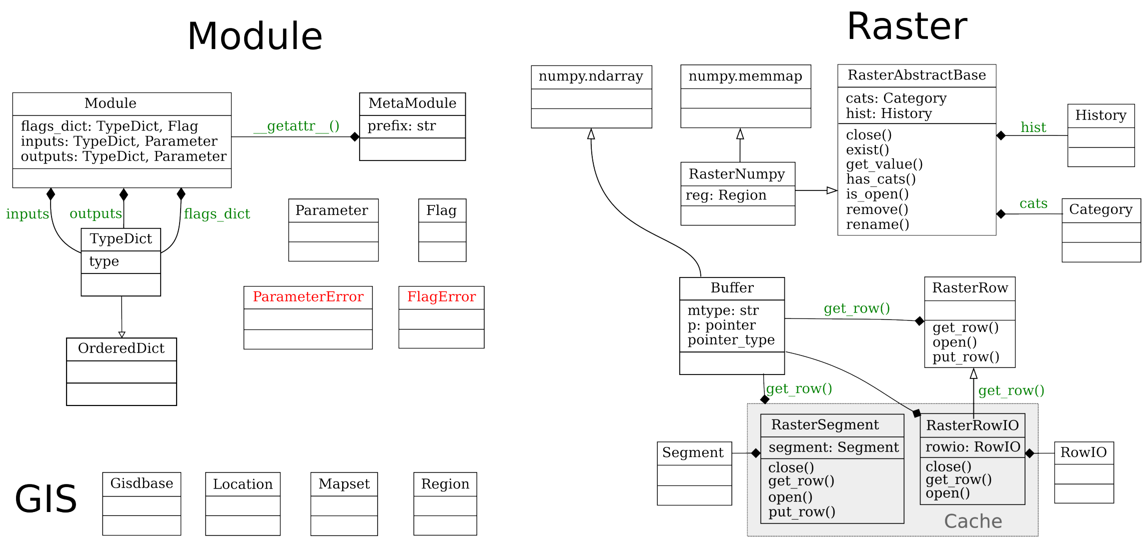
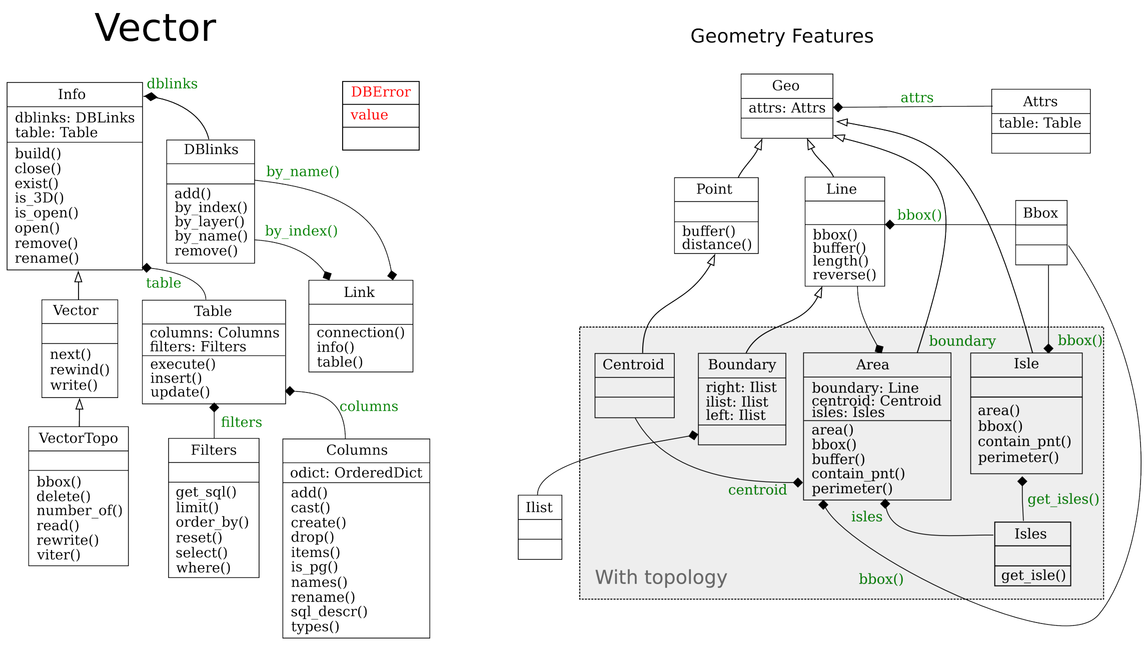


**Figure 2.** Vector class.



- `modules` contains the classes `Module`, `MetaModule` and `Parameter`. These classes are designed to substitute the previous POSIX-based scripting approach (see Listing 2) and replace parts of the existing Python script API (see Listing 3).
  The object-oriented architecture of the PyGRASS library allows users to interact with GRASS modules as Python objects. These objects allow direct access of module attributes like: `name`, `description` and `keywords`. The `input` and `output` options are implemented using a

dedicated parameter class. Instances of this class are stored either in an input dictionary or an output dictionary within the module object Listing 4. Inputs and outputs can be referenced by their name in the dictionaries or as attributes of the dictionary objects. They can be connected to each other to create process chains; see Listing 1. The type check system of the parameter class assures that output options can only be connected with input options of different modules when they have the same type. Hence, the PyGRASS module library will raise an error in case the output of a vector module was connected with a raster input of a second module.

The PyGRASS module library introduces special parameters to allow fine-grain control over the GRASS processes. These special parameters end with the '_' character to avoid a mix-up with option names. The first two special parameters—`run_` and `finish_`—are used to manage the process. The parameter definition `run_=True` will execute the process immediately, and `finish_=True` will wait until the process terminates; see Listing 5. Other special parameters that were added are: `stdin_`, `stdout_` and `stderr_`. The parameters `stdin_` and `stdout_` are used to connect the textual inputs and outputs from different modules to create a process pipeline; `stdin_` is used to pass the textual output (`stdout_`) from one process to another; see Listing 6;

- `vector` contains the classes `Vector` without the GRASS topology and `VectorTopo` with the GRASS topology.

  The `Vector` class allows the user to access the non-topological geometry features of a vector map in sequential order; see Listing 7. The class `VectorTopo` was designed to access topological and non-topological geometry features of a vector map in random order. This class allows one to iterate among specific feature types; see Listings 8 and 9. Writing is supported in booth classes in sequential order. However, already written features can be updated in the topological access class. The following classes are designed to represent vector features: `Point`, `Line`, `Centroid`, `Boundary`, `Isle` and `Area`. Instances of these classes are usually created when features are read from vector maps by the `Vector` and `VectorTopo` classes. To manage multiple connections with vector attribute SQL databases, the classes `DBLinks` and `Link` were designed. Attribute tables can be created, accessed and modified with the `Table` class. The `Filter` class provides several methods to work with the data without the need to know SQL. To access the content of the attributes table from a geometry feature, the `Attrs` class was designed.
- `raster` contains the classes (`RasterRow`, `RasterRowIO`, `RasterSegment` and `RasterNumpy`).

  Each class uses a different GRASS C-library to grant a specific kind of access to raster maps. All the raster classes share common methods to open a map, read raster values or raster rows, get raster information and write metadata, like categories and history. Similarly to vector, a similar syntax has been used to instantiate, open and close a raster object.

  The `RasterRow` class reads the content of the raster map row by row and writes it in a sequential mode, row after row; see Listings 10 and 11.

  The `RasterRowIO` class implements a row cache that allows users to read raster rows randomly by keeping a number of rows in the main memory. This caching mechanism avoids heavy I/O (input/output) hard-disk usage in specific tasks, like moving window operations, or cell

neighborhood analysis. Similarly to the `RasterRowIO`, the `RasterSegment` class provides access to a tile cache. The tile cache is an uncompressed representation of a raster map that will be created at the point of initialization. The access to the uncompressed file is based on tiles that are cached in the main memory for fast random read and write access through the `Segment` class. With the `RasterSegment` class, it is possible to read and write the pixel value randomly at the same time in the same map.
The `RasterNumpy` class inherits from a `numpy.memmap` class and allows users to interact with the map as a NumPy matrix.

- `gis` contains GRASS management classes, like `Gisdbase`, `Location` and `Mapset`, that help users to interact with the GRASS environment; see Listing 12. The `Region` class manages the computational region of GRASS that directly affects 2D and 3D raster processing, as well as several vector processing algorithms; see Listing 13.

The PyGRASS library assures that the memory management is fully handled by python. All structures from the GRASS C-API that are used by PyGRASS are *ctypes* objects or get deleted in the class destructor's and, therefore, are handled by the Python garbage collector. The user must not take care of memory allocation and deletion directly.

## 3. Results

In this section, we compare different solutions of simple GIS tasks using standard GRASS tools and PyGRASS. The machine used for the benchmark was a laptop with an Intel Core i7 3610QM processor with 2.30 GHz and 6 Mb L3 Cache. The system has 24 Gb DDR3@1333Mhz of RAM and a solid state disk (SSD) of 250 Gb as the system driver. The installed operating system (OS) is GNU/Linux 3.7.5 (×86_64) on the SSD. The GRASS 7 development version used for the benchmark has the revision number `r54812`. The GRASS data are stored on a secondary hard disk of 750 Gb at 7,200 rpm.

Concerning the script activity, PyGRASS improves mainly the syntax and changes how users can interact with GRASS modules. We measured small performance loss when executing GRASS modules using the PyGRASS module interface compared to the POSIX approach, which go from 1% up to 12%, due to the average load of the system. We did not expect a large performance difference, since Python and POSIX are basically using the same OS (operating system) functions to spawn processes.

On the contrary, the new API added by PyGRASS must be tested (all the benchmark tests used in this paper are available at https://github.com/zarch/pygrass-benchmark) to identify its strengths, weaknesses and scalability. Each test, excluding the biggest region (with $10^{10}$ cells), has been repeated five times. There are only small differences between each measured run time, resulting in a small standard deviation. Hence, we think that the final results of our benchmark are representative.

The first test compares two simple procedures, one written using PyGRASS (`RasterRow` and `VectorTopo`) (see Listing 14) and the other using the programming language C (see Listing 15 and for the results, see Table 3). The test takes as inputs a vector point map and a raster map. It creates a new vector point map that includes all vector points from the input map. A new attribute table is created and linked with the vector map, which contains a column with the sampled values of the raster map. The procedure is applied to five different random vector point maps, to be independent from the spatial distribution of the vector points. Moreover, the tests have been executed using different region extents and number of points, to test the scalability of the different solutions. Both procedures are conceptually identical and share most of the GRASS C-API functions. The only difference is the database access, where PyGRASS uses the Python driver instead of the C-API of GRASS.

**Table 3.** Table with the benchmark results, reported in seconds, using different computational extents and a different number of vector points.

| number of cells | $10^2$ | $10^4$ | $10^6$ | $10^8$ | $10^{10}$ |
|---|---|---|---|---|---|
| number of points | 10 | $10^2$ | $10^3$ | $10^4$ | $10^5$ |
| **Vector and Raster** | | | | | |
| sample (PyGRASS) | 2.21 | 4.23 | 23.87 | 218.63 | 12670.27 |
| v.sample2 (C-API) | 3.03 | 5.48 | 31.67 | 266.54 | 13304.67 |
| **Raster** | | | | | |
| RasterRow | 0.046 | 0.431 | 4.53 | 74.46 | 4303.24 |
| r.mapcalc | 0.078 | 0.525 | 5.83 | 170.43 | 5347.95 |

One indication that our approach is easier to handle than the C implementation is that the PyGRASS version is considerably shorter (48 lines) than the C version (102 lines). A PyGRASS library, it is noted that there is a marginal advantage in speed compared to it's C counterpart. The speed gain over the C version it is probably due to the slower driver adopted by the GRASS C API of the vector attribute database.

In a further test we compared the performance of the PyGRASS `RasterRow` implementation (see Listing 16), with `r.mapcalc` using a simple raster map algorithm (see Listing 17). The algorithm stores only those pixels in a new raster map that have a value that it is greater than 50. Again, the PyGRASS version is slightly faster than the GRASS module. The good performance of PyGRASS is caused by our design approach that uses NumPy for row computation tasks. The performance will drop dramatically in case we implement the same algorithm in Python comparing cell by cell without using the optimized NumPy approach.

## 4. Discussion and Benchmarks

The PyGRASS `Module` class adds some useful features that were not available with the previous Python script API; these features have a time cost, because they require exporting the GRASS module in XML, parsing the XML and instantiating the object, checking that all the parameters are correct and then executing. The time cost for these operations is around 0.2 s, but generally, the execution time of a GRASS module requires much more time; therefore, in most of the cases, we can neglect this time loss.

Concerning the new approach introduced by PyGRASS, the performance depends mainly on the features that are used. For example, updating the column attribute with the value of area with PyGRASS requires almost the same time, around 0.24 s for PyGRASS and 0.26 s using the `v.to.db` module.

Using the `RasterRow` class to compute areas that satisfy a condition, with a region of 16,000 rows and 14,000 columns, it is slightly faster (27.42 s) than using `r.mapcalc` (35.49 s) if the row is used as a NumPY array:

Using the r.mapcalc module:

Using the PyGRASS `RasterRow` class without using the NumPy array makes the execution seven-times slower than using the GRASS `r.mapcalc` module; below the Python code (992.5 s) is reported.

Using r.mapcalc (144.2 s):

The example above highlights that it is not convenient to replace an existing GRASS module with a new one written in PyGRASS, because the user has to write more code and because the GRASS native modules are generally faster. The big advantage of using the PyGRASS library is the object-oriented access to the GRASS C-API functionality.

Without the need to extract information from the output string of the module, in this kind of operation, the PyGRASS library is faster compared with modules and with the existing Python functions: for example, to get the list of the raster map contained in a `Mapset` with PyGRASS takes (608 ns).

Using the Python function `list_grouped` in the GRASS core takes (0.1273 s).

The same good results are obtained with the `Region` object; with PyGRASS, it takes (211 ns).

Using the Python function `region` in the GRASS core takes (0.1056 s).

The PyGRASS library can help to substitute all the commands in the GRASS Python `script` library that need to wrap and interpret the output of a GRASS module.

## 5. Conclusions

An increasing number of GIS software uses the Python language to provide a powerful scripting interface. An easy to use, but powerful, python interface can help to efficiently exploit the capabilities of a GIS software. Such an interface can be effectively used to integrate different GIS, statistical, geospatial tools and programming languages in a GIS to expand its overall capabilities.

The PyGRASS library tries to open a new perspective to power users and scientists that use GRASS GIS. It provides a Python interface that is able to compete with the simplicity of POSIX to write procedures with existing GRASS modules, as well as a powerful object-oriented interface to deal and experiment with GIS problems at a lower level.

The new `Module` class, introduced by PyGRASS, provides a single interface to all GRASS modules and can be extended to work with Web Processing Services (WPS), Web Services Description Language (WSDL) and Simple Object Access Protocol (SOAP) services or other remote execution services. The design concept of the `Module` class allows the direct linking of inputs and outputs of GRASS modules to create process chains, including compatibility checks, process control and error handling.

The new Object-Oriented Python programming API introduces an abstract layer that opens the possibility for the users who are not familiar with C and with GRASS C-API, to use and access transparently the efficient C functions of GRASS. Our tests show that algorithms implemented with PyGRASS are comparable in terms of performance with an equivalent C implementation. Hence, our approach wraps the underlying GRASS C libraries efficiently. It needs much less line of code to implement an algorithm in PyGRASS than in C. Moreover, it shows that specific Python strengths, for example, the database Python interface, can be used to gain a speed improvement over specific C-implementations in GRASS. The PyGRASS library has been designed to integrate new methods or to inherit from an existing class to extend the GRASS functionalities, providing new tools for prototyping complex scientific algorithms.

Some of the functionalities provided by PyGRASS are also available in other software, like Postgresql/Postgis, R, shapely, *etc*. However, switching to them requires changing the GIS working environment. That means installing, configuring, learning the new tools and converting from one format to another. The PyGRASS library does not force the users to learn and switch between different languages (C, SQL, R, Python, BASH, *etc*.) and tools to carry out their work.

The PyGRASS library allows GIS modelers and scientists to use the C-API of GRASS, with a high level interface, providing a tool that gives the freedom to approach the GIS problem from a different perspective. In this way, users and scientists can combine the GRASS modules with the GRASS C-API functions and algorithms. Therefore, PyGRASS is able to simplify the approach to develop a new GIS model, using one program (GRASS) and one language (Python) to cover the different GIS aspects, increasing the productivity and allowing geo-scientist to focus on the study of the problem they have selected and not on the study of the tools and languages used.

Moreover, the PyGRASS library can be used as a tool to facilitate use and integration with other GIS/statistical software and libraries (not only open source). The common language among different software and the object-oriented structure should make communication and procedure/data exchange easier.

The PyGRASS library, together with the GRASS GIS temporal framework, can provide a comprehensive high performance spatio-temporal GIS framework for GI-Scientists.

The PyGRASS seems to be ideal to be applied in complex case studies, like air quality monitoring from wireless sensor networks, and to build decision support systems to evaluate the assessment of sustainable forest energy.

## Acknowledgments

We are grateful to the Google Summer of Code project that supported the idea and the implementation of the PyGRASS module. I would like to thank my GSoC co-mentors, Luca Delucchi, Martin Landa and Markus Metz, for the very valuable help and support. We would like to thank the GRASS software community, who have begun to actively support the PyGRASS module with good ideas, bug fixes and tests. Lastly, we would like to thank the core GRASS Development Team that strongly supported the project, providing suggestions and clarifying how GRASS works.

## References

1. Foody, G. GIS: Biodiversity applications. *Prog. Phys. Geogr.* **2008**, *32*, 223–235.
2. Neteler, M.; Bowman, M.H.; Landa, M.; Metz, M. GRASS GIS: A multi-purpose open source GIS. *Environ. Modell. Softw.* **2012**, *31*, 124–130.
3. Sacchelli, S.; Zambelli, P.; Zatelli, P.; Ciolli, M. Biomasfor—An open source holistic model for the assessment of sustainable forest bioenergy. *iForest* **2013**, in press.
4. Hofierka, J.; Zlocha, M. A new 3-D solar radiation model for 3-D city models. *Trans. GIS* **2012**, *16*, 681–690.

5. Preatoni, D.; Tattoni, C.; Bisi, F.; Masseroni, E.; D'Acunto, D.; Lunardi, S.; Grimod, I.; Tosi, G. Open source evaluation of kilometric indexes of abundance. *Ecol. Inform.* **2012**, *7*, 35–40.
6. Tattoni, C.; Rizzoli, F.; Pedrini, P. Can LiDAR data improve bird habitat suitability models? *Ecol. Modell.* **2012**, *245*, 103–110.
7. Zambelli, P.; Lora, C.; Spinelli, R.; Tattoni, C.; Vitti, A.; Zatelli, P.; Ciolli, M. A GIS decision support system for regional forest management to assess biomass availability for renewable energy production. *Environ. Modell. Softw.* **2012**, *38*, 203–213.
8. Vettorato, D.; Geneletti, D.; Zambelli, P. Spatial comparison of renewable energy supply and energy demand for low-carbon settlements. *Cities* **2011**, *28*, 557–566.
9. Li, X.; Di, L.; Han, W.; Zhao, P.; Dadi, U. Sharing geoscience algorithms in a Web service-oriented environment (GRASS GIS example). *Comput. Geosci.* **2010**, *36*, 1060–1068.
10. Tattoni, C.; Ciolli, M.; Ferretti, F.; Cantiani, M.G. Monitoring spatial and temporal pattern of Paneveggio forest (Northern Italy) from 1859 to 2006. *iForest* **2010**, *1*, 72–80.
11. Zambelli, P.; Lora, C.; Ciolli, M.; Spinelli, R.; Tattoni, C.; Vitti, A.; Zatelli, P. A FOSS4G Model to Estimate Forest Extraction Methods and Biomass Availability for Renewable Energy Production. In Proceedings of FOSS4G, Barcelona, Spain, 6–9 September 2010.
12. Okabe, A.; Satoh, T.; Sugihara, K. A kernel density estimation method for networks, its computational method and a GIS-based tool. *Int. J. Geogr. Inf. Sci.* **2009**, *23*, 7–32.
13. Vettorato, D.; Zambelli, P. Estimation of Energy Sustainability at Local Scale : An Approach Based on Innovative Analytical and Mapping Tools and Multicriteria Analysis. In Proceedings of 45th ISOCARP Congress, Porto, Portugal, 18–22 October 2009; pp. 1–12.
14. Ciolli, M.; de Franceschi, M.; Rea, R.; Vitti, A.; Zardi, D.; Zatelli, P. Development and application of 2D and 3D GRASS modules for simulation of thermally driven slope winds. *Trans. GIS* **2004**, *8*, 191–209.
15. Van Rossum, G. *Python Library Reference*; Report CS-R9524; Pub-CWI: Amsterdam, The Netherlands, 1995.
16. Jones, E.; Oliphant, T.; Peterson, P.; Community, S. *SciPy: Open Source Scientific Tools for Python*. Avaiable online: http://www.scipy.org (accessed on 7 March 2013).
17. Varrazzo, D. *Psycopg Community*. *Psycopg–PostgreSQL Database Adapter for Python*. Avaiable online: http://initd.org/psycopg (accessed on 7 March 2013).

## Appendix

Some small samples of code are provided below to show how modelers, scientists and developers could interact with the PyGRASS library. If the code starts with `>>>` , this indicates a python interactive section with the terminal. To use and test the PyGRASS extension, the reader needs to install the latest development version of GRASS7. Furthermore, all the following examples are using the maps contained in the free available GRASS demonstration dataset *North Carolina* (http://grass.osgeo.org/sampledata/north_carolina/nc_basic_spm_grass7.tar.gz).

### *Modules*

**Listing 1**. Direct inputs/outputs.

```
# -*- coding: utf-8 -*-
import subprocess as sub
from pygrass.modules import Module


colout = Module("r.colors.out", map="elevation", stdout_=sub.PIPE)

col = Module("r.colors")
col.inputs.map = "field"
col.inputs.stdin = colout.outputs.stdout
col.inputs.rules = '-'
col.run()
```

**Listing 2**. The syntax is similar to POSIX.

```
# -*- coding: utf-8 -*-
from grass.pygrass.modules import raster as r

# cmd: r.info map=elevation
r.info(map='elevation')

# cmd: r.slope.aspect elevation=elevation slope=slope aspect=aspect --overwrite
r.slope_aspect(elevation='elevation', slope='slope', aspect='aspect', overwrite=True)

# cmd: r.mapcalc 'slope_gt100 = if(slope > 100, slope)' --overwrite
r.mapcalc('slope_gt100 = if(slope > 100, slope)', overwrite=True)
```

**Listing 3**. Backward compatibility.

```
# -*- coding: utf-8 -*-
from grass.pygrass.modules import Module as run_command

run_command('r.info', map='elevation')
```

**Listing 4**. Module as an object.

```
>>> from grass.pygrass.modules import Module
>>> slp = Module('r.slope.aspect')                        # instantiate
>>> slp.name
'r.slope.aspect'
>>> slp.description
'Aspect is calculated counterclockwise from east.'
>>> slp.inputs['elevation']
Parameter <elevation> (required:yes, type:raster, multiple:no)
>>> slp.inputs.elevation = 'elevation'                # set parameter value
>>> slp.inputs.slope = 'slope'
>>> slp.run()                          # finally execute the 'slp' module
```

**Listing 5**. Run and finish.

```
>>> from grass.pygrass.modules import Module
>>> slp = Module('r.slope.aspect')
>>> slp(elevation='elevation', slope='slp', aspect='asp',
...   format='percent', overwrite=True, run_=False) # only set the parameters
>>> slp(elevation='elevation', slope='slp', aspect='asp',
...   format='percent', overwrite=True, finish_=False)   # start the process
>>> slp.popen.wait() # .kill() manage the process
```

**Listing 6**. Stdin.

```
# -*- coding: utf-8 -*-
from pygrass.modules import raster as r

rules = """0 red
1 green
2 blue
end"""

r.colors(map='rtest', rules='-', stdin_=rules)
```

***Vectors***

**Listing 7**. Vector class.

```
>>> from grass.pygrass.vector import Vector
>>> schools = Vector('schools')
>>> schools.open('r')
>>> schools.title    # Vector attributes: name, organization, person, map_date
'Wake County schools (points map)'
>>> school = schools.next()              # access to the geometry features
>>> school
Point(633649.285674, 221412.944348)
>>> for school in schools:        # or simply iterate through the vector map
...   print school
...
POINT(628787.129283, 223961.620521)
POINT(629900.710134, 222915.798505)
POINT(630686.456623, 224447.772161)
...
>>> schools.close()
```

**Listing 8**. VectorTopo class.

```
>>> from grass.pygrass.vector import VectorTopo
>>> geology = VectorTopo('geology')
>>> geology.open('r')
>>> geology.title    # Vector attributes: name, organization, person, map_date
'North Carolina geology map (polygon map)'
>>> for g in geology:                # or iterate through the vector map
...   print g
...
LINESTRING(...)
LINESTRING(...)
...
POINT(...)
POINT(...)
...
>>> big = [a for a in geology.viter('areas')    # iterate only a geometry type
...     if a.alive() and a.area() >= 10**8]
>>> geology.close()
```

**Listing 9**. Write a new vector map.

```
# -*- coding: utf-8 -*-
from pygrass.vector import VectorTopo

# instantiate and open the tunnels map
tunnels = VectorTopo('tunnels')
tunnels.open('r')

```

```
# instantiate a new map
new = VectorTopo('newvect')

# open a new vector map defining the column names and types, with:
new.open('w', tabcols=[(u'cat', int),
                       (u'tunnel', int),
                       (u'length', float)])

for tunnel in tunnels:
    # extract the first and the last point of the tunnel
    first, last = tunnel[0], tunnel[-1]
    # compute the tunnel length and divide the length for each point
    length = tunnel.length() / 2.
    #.write(geom_feature, attributes)
    new.write(first, (tunnel.cat, length))
    new.write(last, (tunnel.cat, length))
# then close all
new.close()
tunnels.close()
```

### *Rasters*

**Listing 10**. RasterRow class.

```
>>> from grass.pygrass.raster import RasterRow
>>> elev = RasterRow('elevation')
>>> elev.exist()                          # check if the map exist
True
>>> elev.name                            # return the raster name
'elevation'
>>> elev.mapset                         # return the raster mapset
'PERMANENT'
>>> elev.open('r')                              # open in read mode
>>> elev.is_open()                       # check if the map is open
True
>>> elev.range                            # return the map range
(55.578792572021484, 156.32986450195312)
>>> for row in elev[:5]:                        # get the first 5 rows
...   print(row[:3])               # show the first 3 columns of each row
...
[ 141.99613953 141.27848816 141.37904358]
[ 142.90461731 142.39450073 142.68611145]
[ 143.81854248 143.54707336 143.83972168]
[ 144.56524658 144.58493042 144.86477661]
[ 144.99488831 145.22894287 145.57142639]
>>> elev.close()
```

**Listing 11**. Write a new raster map.

```
# -*- coding: utf-8 -*-
from grass.pygrass.raster import RasterRow

# instantiate raster objects
old = RasterRow('elevation')
new = RasterRow('mapname1')

# open the maps
old.open('r')
new.open('w', mtype=old.mtype, overwrite=True)

# start a cycle
for row in old:
    new.put_row(row > 100)                                      # write row

# close the maps
new.close()
old.close()
```

### *GIS/GRASS*

**Listing 12**. Write a new raster map.

```
>>> from grass.pygrass.gis import Location, Mapset
>>> location = Location()
>>> location.mapsets()
['PERMANENT', 'user1']
>>> permanent = Mapset('PERMANENT')
>>> permanent.glist('rast', pattern='elev*')      # return a list with rasters
['elevation_shade', 'elevation']
```

**Listing 13**. Write a new raster map.

```
>>> from grass.pygrass.gis.region import Region
>>> region = Region()
>>> region.north
258500.0
>>> region.south
185000.0
>>> region.rows
7350
>>> region.nsres
10.0
>>> print region
projection: 99 (Lambert Conformal Conic)
zone:      0
datum:     nad83
ellipsoid: a=6378137 es=0.006694380022900787
north:     258500
south:     185000
west:      596670
east:      678330
nsres:     10
ewres:     10
rows:      7350
cols:      8166
cells:     60020100
```

### *Benchmark*

**Listing 14**. Write a new vector points map adding the raster value in the attribute table using PyGRASS.

```
# -*- coding: utf-8 -*-
import numpy as np
from grass.pygrass.gis.region import Region
from grass.pygrass.vector import VectorTopo
from grass.pygrass.raster import RasterRow
from grass.pygrass.functions import coor2pixel


def sample(vect_in_name, rast_in_name):
    """sample('point00', 'field')"""
    # instantiate the object maps
    vect_in = VectorTopo(vect_in_name)
    rast_in = RasterRow(rast_in_name)
    vect_out = VectorTopo('test_' + vect_in_name)

    # define the columns of the attribute table of the new vector map
    columns = [(u'cat',       'INTEGER PRIMARY KEY'),
               (rast_in_name,  'DOUBLE')]

    # open the maps
    vect_in.open('r')
    rast_in.open('r')
    vect_out.open('w', tab_cols=columns, link_driver='sqlite')

    # get the current region
    region = Region()

    # initialize the counter
    counter = 0
    data = []
    for pnt in vect_in.viter('points'):
        counter += 1
        # transform the spatial coordinates in row and col value
        x, y = coor2pixel(pnt.coords(), region)
        value = rast_in[int(x)][int(y)]
        data.append((counter, None if np.isnan(value) else float(value)))
        # write the geometry features
        vect_out.write(pnt)

    # write the attributes
    vect_out.table.insert(data, many=True)
    vect_out.table.conn.commit()

    # close the maps
    vect_in.close()
    rast_in.close()
    vect_out.close()
```

**Listing 15**. Write a new vector points map adding the raster value in the attribute table using C.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include <grass/gis.h>
#include <grass/raster.h>
#include <grass/glocale.h>
#include <grass/dbmi.h>
#include <grass/vector.h>


static void sample(char *input, char *rast, char *column, char *output) {

    struct Cell_head window;
    struct Map_info In, Out;
    int fdrast;
    DCELL value;
    G_get_window(&window);
    int line;
    int type;
    struct field_info *Fi;
    dbDriver *Driver;
    char buf[2000];
    dbString sql;
    DCELL *dcell_buf;

    /* Open input */
    Vect_set_open_level(2);
    Vect_open_old2(&In, input, "", "1");
    fdrast = Rast_open_old(rast, "");

    /* Open output */
    Vect_open_new(&Out, output, 0);
    Vect_hist_copy(&In, &Out);
    Vect_hist_command(&Out);

    /* Create table */
    Fi = Vect_default_field_info(&Out, 1, NULL, GV_1TABLE);
    Vect_map_add_dblink(&Out, Fi->number, Fi->name, Fi->table, Fi->key,
            Fi->database, Fi->driver);
    Driver = db_start_driver_open_database(Fi->driver,
            Vect_subst_var(Fi->database, &Out));
    sprintf(buf, "create table %s ( cat integer, rast_val double precision)",
            Fi->table);
    db_init_string(&sql);
    db_set_string(&sql, buf);
    db_execute_immediate(Driver, &sql);
    db_create_index2(Driver, Fi->table, Fi->key);
    db_grant_on_table(Driver, Fi->table, DB_PRIV_SELECT, DB_GROUP | DB_PUBLIC);

    /* Sample the raster map with vector points */
    struct line_pnts *Points = Vect_new_line_struct();
    struct line_cats *Cats = Vect_new_cats_struct();
    int nlines = Vect_get_num_lines(&In);
    int count = 0;

    dcell_buf = Rast_allocate_buf(DCELL_TYPE);

    db_begin_transaction(Driver);

    for (line = 1; line <= nlines; line++) {
        type = Vect_read_line(&In, Points, Cats, line);

        if (!(type & GV_POINT))
            continue;

        if(G_point_in_region(Points->x[0], Points->y[0]) == 0)
            continue;

        if (Rast_is_d_null_value(&value))
            continue;

        int row = Rast_northing_to_row(Points->y[0], &window);
        int col = Rast_easting_to_col(Points->x[0], &window);

        Rast_get_d_row(fdrast, dcell_buf, row);
        value = dcell_buf[col];

        /* Write value into the vector table */
        count++;
        Vect_reset_cats(Cats);
        Vect_cat_set(Cats, 1, count);
        Vect_write_line(&Out, GV_POINT, Points, Cats);

        sprintf(buf, "INSERT INTO %s VALUES ( %d, %e ); ", Fi->table, count,
                (double)value);
        db_set_string(&sql, buf);
        db_execute_immediate(Driver, &sql);
    }

    db_commit_transaction(Driver);
    db_close_database_shutdown_driver(Driver);

    Rast_close(fdrast);
    Vect_close(&In);
    Vect_build(&Out);
    Vect_close(&Out);

    exit(EXIT_SUCCESS);
}
```

**Listing 16**. Compute using the RasterRow class.

```
# -*- coding: utf-8 -*-
from grass.pygrass.raster import RasterRow

def ifcondition(mapname0, mapname1):
    # instantiate raster objects
    old = RasterRow(mapname0)
    new = RasterRow(mapname1)
    # open the maps
    old.open('r')
    new.open('w', mtype=old.mtype, overwrite=True)
    # start a cycle
    for row in old:
        true = row > 50
        new.put_row(row * true)
    # close the maps
    new.close()
    old.close()
```

**Listing 17**. Call the *r.mapcalc* module from Python

```
sub.Popen("r.mapcalc expression='test_mapcalc=if(field>50,field,0)' --o", shell=True).wait()
```