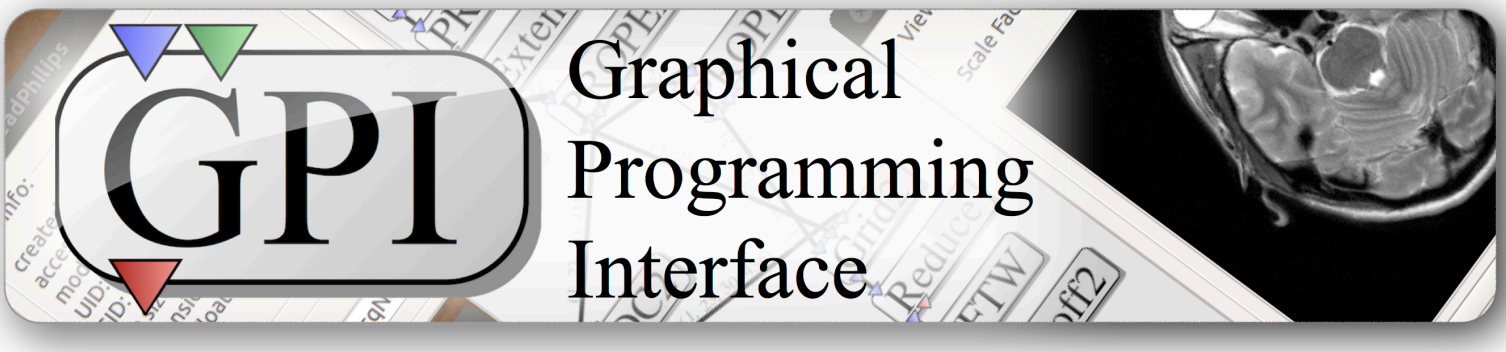

GPI 0.2 Release

Nicholas R. Zwart, Ph.D.

Summary:

The Graphical Programming Interface (GPI)[1] is a multi-platform, Python[2] based, application framework designed to modularize and link scientific algorithms, similar to projects such as AVS[3] and Vision[4]. This framework simplifies algorithm development and deployment by abstracting the GUI API from the algorithm code and by allowing rapid prototyping of new algorithms that are aggregates of existing elements or unit test components of a larger method. In contrast to similar projects, GPI has few dependencies, placing the feature development on the node-modules themselves.

User Interface:

Algorithm nodes are instantiated and connected via the canvas GUI (Fig. 1) to create workflows or processing pipelines as represented by a flow diagram. Each node has an associated interactive window that allows parameters to be modified (through sliders, text boxes, etc...) or for information (such as images, plots, or text) to be displayed. Changes, to the canvas or node windows, trigger new processing events that update each algorithm in realtime. This gives immediate feedback to the user when exploring data at each point of a method process (e.g. slicing through 3D data volumes).

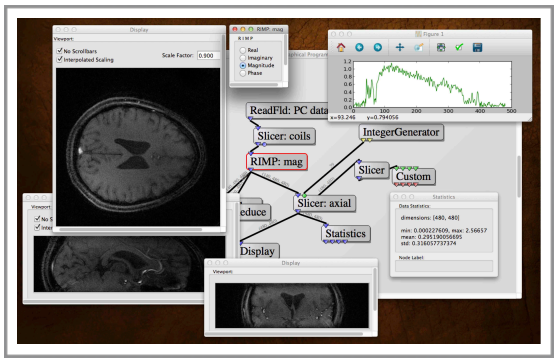

Fig. 1: A screen-capture of an active GPI session. The canvas (lowered window) shows a flow diagram of algorithm nodes. The node's interactive windows (raised windows) show input and output UI elements such as toggles, displays and plotters.

Code Interface:

Each node on the canvas represents a Python interface to underlying algorithm code, which can be written in any language that can be bound to Python. The node interface allows UI elements to be added to its associated interactive window in a simple manner, separate from the algorithm. The core algorithm can be instantiated as a separate process for developing without being incorporated in the Python namespace, allowing the instantiation of unique extension modules (over multiple recompilations) without restarting the Python interpreter (GPI session). Developed modules can also be run as threads for faster shared memory use or as part of the application loop for new UI elements (plotters, displays, etc...). The finished algorithm can then be easily deployed to another system (e.g. to a reconstruction processor) as it contains no GPI specific code.

Basic Features:

Network Load/Save:

GPI networks can be saved to a file and re-instantiated later.

Drag-and-Drop:

External node code, data files, plugins, and network descriptions can be instantiated on the canvas by dragging the file from a file browser.

File associations:

Filetypes can be paired with their associated 'reader' nodes. This release is aware of .data/.list, .fld, and .raw.

Nodes:

The supplied node library contains various file readers and writers, signal processing (via numpy, scipy, etc...) libraries, a 2D image display, a plotter (via matplotlib), and many more possible configurations through the use of the 'Custom' node (an editable Python scripting module). This release has been packaged with a few demo networks that illustrate the potential uses of the supplied nodes.

Extensible:

New nodes can be easily generated in Python or in C++ (with a Python wrapper). The supplied C++ interface translates Numpy arrays to the Philips Reconstruction Platform array library. C++ code written with this library is then easily converted to run directly on the Philips scanning platform after prototyping.

Install & Launch:

This version is released as installable packages for Linux and OSX and a Linux virtual machine (Ubuntu 12.04). The recommended requirements for running GPI are:

- 8GB system memory
- 4-core processor
- 50GB of drive space
- VMWare Fusion / VMPlayer ≥ 5 (for the Linux VM).

Packaged Software:

GPI Framework:

- Anaconda 1.9
- Python 2.7 (via Anaconda)
- Qt 4.10
- PyQt4 4.10

Packaged GPI Node Dependencies:

- Scipy 0.13
- Numpy 1.8
- Matplotlib 1.3

Known Issues:

GPI is currently in a rapid state of development and as such the behavior is in constant flux. Program errors can be the result of the underlying GPI ***Framework*** or specific to a particular ***Node***. At the time of this release the following common occurrences have been identified (this is not an exhaustive list):

1) ***Framework:*** If node connections are made or disconnected during canvas processing the event loop can get stuck requiring the user to pause-then-unpause in order to trigger pending events.
2) ***Node:*** Clipping planes that are added to the GLViewer cannot be deleted. A new GLViewer must be instantiated to clear the plane objects.
3) ***OSX:***
    1) ***Canvas:*** On OSX Mavericks, the Terminal.app steals focus which hides the main GPI menu. To get the GPI menu back just click the Terminal.app icon, then click the canvas to regain focus.
    2) ***File Size:*** The underlying object serialization in OSX appears to be 32bit forcing all read/written files over 2GB to fail for .npy, .pickle, and .hdf5.
    3) ***Scipy/Numpy:*** Some scipy and numpy packages such as 'linalg' randomly fail (often with a malloc error) if run in a GPI_PROCESS on OSX. This seems to be an OSX specific issue. Run the node as a GPI_THREAD to get around this.

Feedback:

This release is offered without support. However, questions and comments sent to philips.gpi@gmail.com will be compiled into a FAQ for release at a later date.

Developers:

GPI is developed in the Keller Center for Imaging Innovation a Philips funded lab lead by **James Pipe**. **Nicholas Zwart** is the primary developer. Senior scientists at the Keller Center involved in GPI development include **Ryan Robison**, **Zhiqiang Li**, **Dinghui Wang** and **Yuchou Chang**. **Mike Schar** is a Philips scientist contributing to the project.

References:

[1] N. Zwart and J. Pipe, ISMRM Workshop on Data Sampling, Sedona, 2013 [2] G. van Rossum et al., CWI Quarterly, 4, 1991; [3] C. Upson et al., IEEE Comput. Graph., 9, 1989; [4] M. F. Sanner, Structure, 13, 2005.