Pyphant - A Python framework for modelling reusable data processing tasks

Klaus Zimmermann Lorenz Quack
Andreas W. Liehr
Servicegroup Scientific Data Processing[*]
Freiburg Materials Research Center
University of Freiburg

Europython2006-paper.tex 1114 2006-07-18 10:21:44Z obi

We are presenting the Python framework *Pyphant* for the creation and application of data flow models. The central idea of this approach is to encapsulate each data processing step in one unit which we call *worker*. A worker receives input via *sockets* and provides the results of its data processing via *plugs*. These can be inserted into other workers' sockets. The resulting directed graph is called *recipe*. Classes for these objects comprise the Pyphant core. To implement actual processing steps Pyphant relies on third party plug-ins which extend the basic worker class and can be distributed in so-called *Pyphant-worker-archives* (*PWA*).

On top of the core Pyphant offers a data exchange layer on basis of scipy arrays and PIL images which facilitates the interoperability of the workers. A third layer comprises textual and graphical user interfaces. The latter allows for the interactive construction of recipes, the former for the batch processing of data.

Our contribution discusses the Pyphant framework and presents an example recipe for determining the length scale of aggregated polymeric phases building an amphiphilic conetwork from an Atomic Force Microscopy (AFM) phase mode image.

[*]e-mail: `servicegruppe.wissinfo@fmf.uni-freiburg.de`

1 Introduction

Working as computer scientist in an interdisciplinary scientific community often means adapting a previously developed data processing algorithm to the very special context of a new project. An example might be given by image processing [1]. Consider you already have developed an algorithm which determines the particle size distribution of a certain blend of materials on basis of an Atomic Force Microscopy (AFM) measurement. Given the measurement of a different material you probably have to apply different pre-processing steps to the primary data and adapt filter parameters like thresholding values in order to match the characteristics of the new sample. If you think of a programming environment which assists the adaption of this data analysis algorithm you will very quickly consider a flow-based programming paradigm. This ansatz was invented in the late sixties [2] and is quite established which can be seen from the variety of commercial and OpenSource tools applying flow-based programming in the context of visual programming languages [3]. Concerning data analysis several flow-based environments have been implemented in Python, just to mention the visualization tool ViPEr [4] or the Modular toolkit for Data Processing (MDP) [5]. It also has been demonstrated, that Python is perfectly suited to integrate several different software tools e.g. for computation and visualization into a consistent data analysis environment [6].

Inspired from these approaches and having in mind that quite different data analysis tools ranging from standard algorithms of statistics or image processing up to specialized tools developed in the context of material research [7, 8, 9] have to be integrated into a consistent visual programming environment, we started to think about the presented Python framework. A major prerequisite was, that the resulting environment should be suitable not only for the creative work of the specialized scientist but also for standardised data processing of the daily laboratory routine or a large scale data analysis campaign computed in a grid computing environment [10]. This balancing act results in the Pyphant framework enabling the fast integration of software modules into so-called *workers*, which receive input data via *sockets* and provide their cached results via *plugs*. The data analysis algorithms are composed as directed graphs within the Graphical User Interface (GUI) *wxPyphant*. The interactive evaluation of the algorithm is established on basis of an extensible set of *Visualizers* interfacing the matplotlib library [11]. And finally the algorithm can be saved as *Pyphant Recipe Archive* (*PRA*) which provides a Command Line Interface (CLI).

The article starts with an overview of the Pyphant framework and continues with a real life example demonstrating the estimation of the length scale of a phase separated polymer blend. In a second part we are discussing the technical details of the Pyphant implementation. Finally we are summarising our work.

2 Framework

Pyphant is a layered, plugin-based framework suitable for the modelling and execution of a wide range of data processing tasks. It is built on the idea that many computing

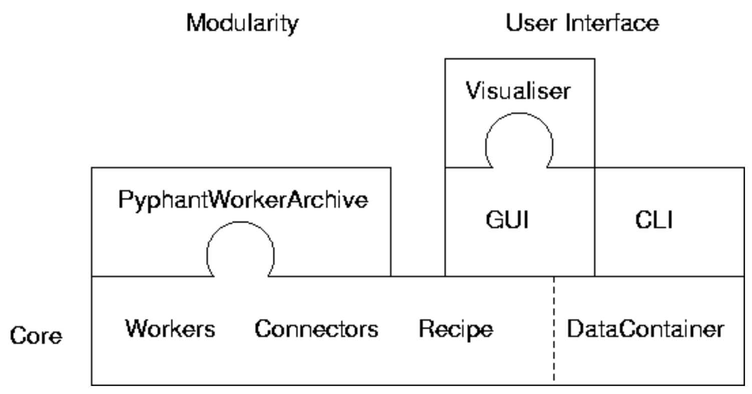

Figure 1: The Pyphant framework consists of a core layer comprising worker, connector, recipe, and DataContainer objects. Workers for specialized data processing tasks can be provided by PythonWorkerArchives. User Interfaces (UIs) are given at hand in form of the Graphical User Interface (GUI) wxPyphant and a Command Line Interface (CLI) for the individual recipe.

algorithms can be structured into a graph of distinct steps. In Pyphant those steps are represented by so-called workers, which also form the nodes in the directed graph. Such an algorithm is called recipe following the famous textbook *Numerical Recipes* [12] and conceiving the development of a data analysis algorithm as the composition of a meal from certain available ingredients.

Fig. 1 shows an overview of the structure of the Pyphant framework. At its base we find the core. Apart from the workers and the recipe we have the connectors which are used to model the edges of our graph and usually are members of the workers. Pyphant's core is completed by the DataContainer class. While the most basic incarnation of a Pyphant application does not impose any restriction on the data format exchanged among workers, we added this container format to enhance the interoperability of the various workers.

On top of the core we find the User-Interface-layer (UI-layer), which comes in two flavours. We have implemented a simple Graphical User Interface (GUI) called wx-Python which realizes the visual programming paradigm. Also provided is a Command Line Interface (CLI) which interfaces individual recipes, such that a certain GUI-crafted recipe becomes a standalone tool. This feature is very useful concerning the daily laboratory routine or the analysis of large data sets. Especially if such a large scale data mining should be performed in a Grid-Computing environment like the Black Forrest Grid [10].

That's all for the Pyphant framework. However, it would not be useful if it wasn't for the plugins which do the actual work. The most important kind of plugin is the worker plugin. We will cover this topic in the technical part of the paper. For now it suffices to notice that workers are bundled in PyphantWorkerArchives (PWA). Another kind of plugin is the visualization plugin which is used by the GUI to visualize data in a suitable format.

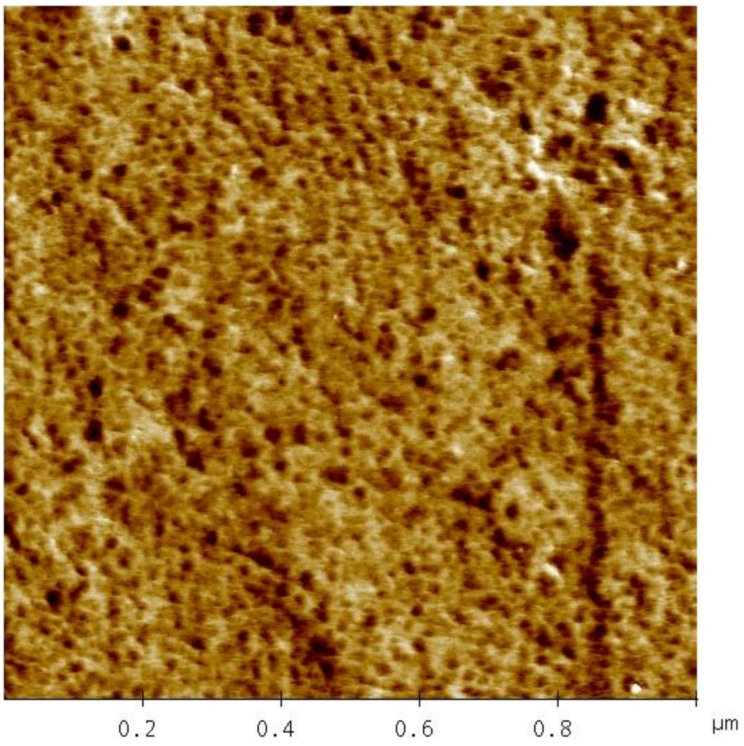

Figure 2: Atomic Force Microscopy (AFM) phase mode image of an amphiphilic poly(2-hydroxyethyl acrylate)-*l*-poly(dimethylsiloxane) (PHEA-*l*-PDMS) conetwork with 23 wt % PDMS. PHEA shows light and PDMS dark.

Now that you have a rough idea of Pyphant, let's start with a real life example of a Pyphant application.

3 Image Processing Example

In this real life example we will explain all steps needed to estimate the width distribution of an aggregated polymer phase from an Atomic Force Microscopy (AFM) phase mode image. The example starts with loading the primary data, preprocessing the data and finally determining the size of the detected features. A possible evaluation step is also discussed.

Fig. 2 shows an AFM phase mode image of an amphiphilic poly(2-hydroxyethyl acrylate)-*l*-poly(dimethylsiloxane) (PHEA-*l*-PDMS) conetwork with 23 wt % PDMS [13]. In this visualization the PHEA and PDMS phase show light and dark, respectively. The question is to determine the width of the PDMS phase.

The complete Pyphant recipe is depicted as snapshot of the GUI in Fig. 3. On the right hand side of the GUI the toolbox of available workers is visualized. Each worker can be placed by drag and drop on the canvas. Clearly visible are the individual workers as white boxes which are connected by arrows pointing from the plug of one worker to the socket of another worker. The colour of the connectors indicate different types of

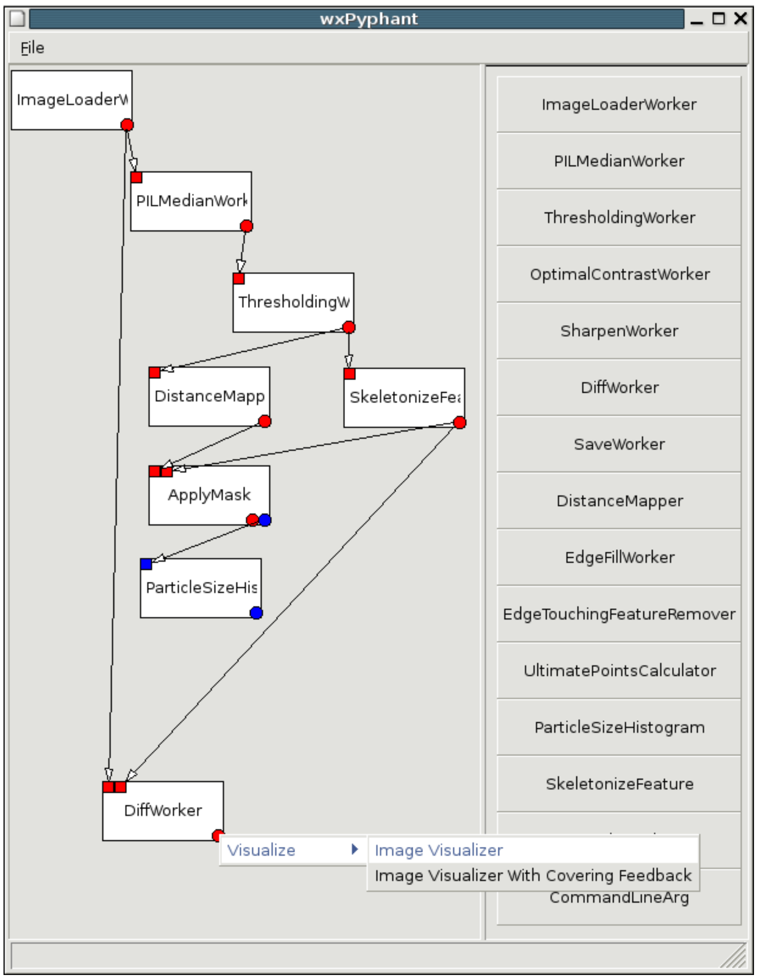

Figure 3: Pyphant recipe composed in the wxPyphant GUI. Workers are visualized as white boxes with sockets placed in their upper left corner and available plugs localised in their lower right corner. By right-clicking a plug a context menu with visualization plugins is provided.

DataContainer. Red indicates an array/image DataContainer, while blue denotes an array/matrix DataContainer. Please note the context menu emerging from the plug of the DiffWorker. It enables the interactive examination of the computed results by interfacing the matplotlib library [11] via visualisation plugins. Let's have a short look at the algorithm:

1. Loading the image
   This is pretty straight forward. Pyphant provides an ImageLoaderWorker which simply loads an image file from the location given in the workers configuration. The respective dialog can be opened by right-clicking the worker. This scheme holds for all configuration dialogs of all worker. The loaded image is provided as gray-scale image at the red plug. As the worker internaly uses the Python Imaging Library (PIL) [14] it supports a great variaty of file formats.

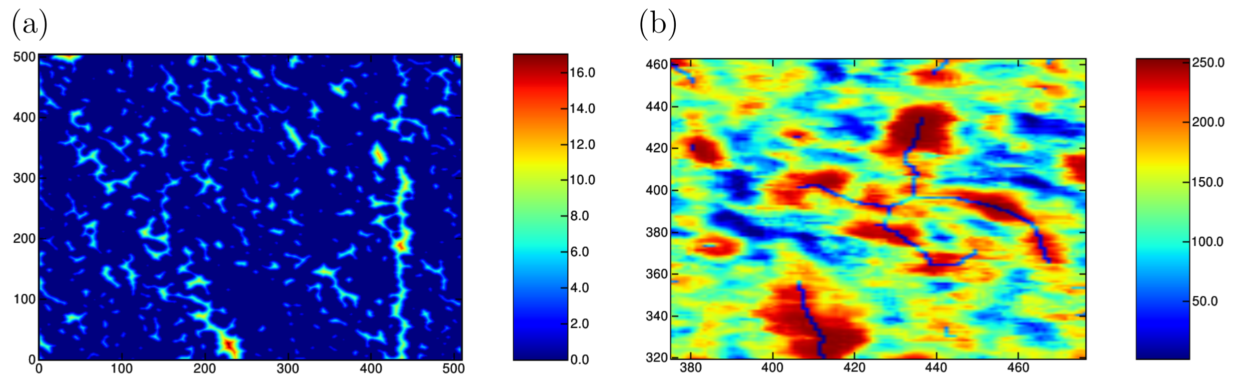

Figure 4: (a) Visualization of DistanceMapper result. The feature size is colour coded in units of pixel. (b) Display detail of difference between AFM image and skeleton of found features. Here the PDMS phase shows red, while the skeleton is represented by blue lines.

2. Removing noise
   Next we want to remove noise from the image. For this task the PILMedianWorker is applied. It can be configured by the size of the applied kernel and the number of smoothing runs. Here we have chosen a 5×5 kernel and five smoothing runs.

3. Applying a threshold
   Now we want to seperate the dark features which represent the PDMS phase from the background. This is achived through the ThresholdingWorker. It compares every pixel of the smoothed image with a given threshold and returns a binary image such that the pixel of comprising features are set to 0x00, while the pixels of the background are set to 0xFF. In this example the threshold is set to 90. The threshold is chosen, such that the fraction of the image being covered by features corresponds to the volume fraction of PDMS of the sample.

4. Measuring the size of the features
   To this point we have a binary image representing the features we are taking into account. Now we would like to determine their size by calculating the distance of each pixel to the nearest background pixel [1, S. 427ff.]. This task is done by the DistanceMapper. The resulting grey-scale image is shown in Fig. 4a with artifical colours.

5. Morphological transform
   In order to retrieve the width of the features, they are skeletonized. This is achieved by iteratively removing the outer pixels of each feature, until the inner core pixels remain [15, Chapter 25].

6. Checking result of skeleton computation
   The skeleton of the features can be compared with the primary data by feeding both images to the DiffWorker. By right clicking the plug of the DiffWorker a

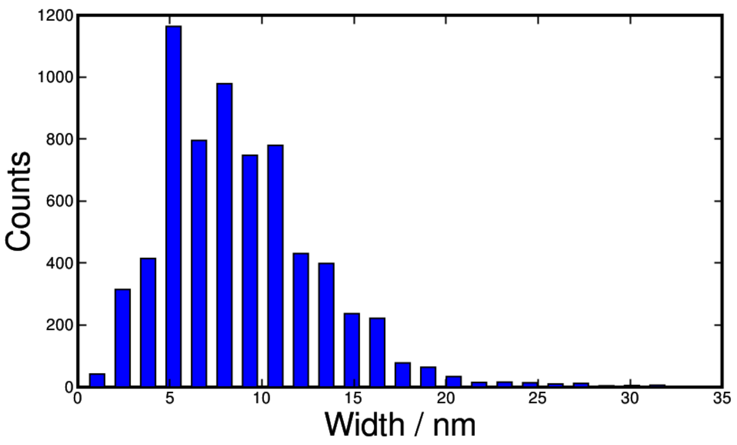

Figure 5: Width distribution of the PDMS phase. The result is obtained from the AFM phase mode image depicted in Fig. 2 with the Pyphant recipe shown in Fig. 3. PILMedianWorker: 5×5 kernel, 5 runs. ThresholdWorker: threshold 90.

visualisation method can be chosen from the context menu. A display detail of the result is depicted in Fig. 4b.

7. Determining the width of the features
The skeleton of the features is applied as a mask to the distance map. This results in a skeleton image, where the brightness of each skeleton pixel corresponds to the width of the feature at the respective position. While this image is provided by the red plug, the blue plug returns the result as $N \times 3$ matrix representation. Here each skeleton pixel is specified by its lateral position and the respective feature width.

8. Computing the histogram
By now the recipe produces the information we are interessted in. The only thing left to do is to compute a histogram from the data provided. This is done by the HistogramWorker. The resulting histogram presenting the length scale of the PDMS phase is shown in Fig. 5. The width distribution of the PDMS phase determined by the Pyphant recipe matches the results of Bruns et. al. [13].

4 The Pyphant Core

In this section we will describe the Pyphant core in greater detail. First we will show an example of a worker, then the worker base class in general. Next we will describe the connection facilities that link the workers into the recipe and the efficient computing model this suggests. Finally we discuss the DataContainer which is the preferred data exchange class, designed for a maximum of worker interoperability.

4.1 The Worker

4.1.1 The DiffWorker - a Practical Example

```
 1 from pyphant.core import (Worker, Connectors,
 2                           DataContainer)
 3
 4 import ImageChops
 5
 6 def createWorker(recipe, annotations={}):
 7     return DiffWorker(recipe, annotations)
 8
 9 WORKER_INFO = Worker.WorkerInfo("DiffWorker", createWorker)
10
11 class DiffWorker(Worker.Worker):
12     _sockets = [ ("image1", Connectors.TYPE_IMAGE),
13                  ("image2", Connectors.TYPE_IMAGE) ]
14
15     @Worker.plug(Connectors.TYPE_IMAGE)
16     def diffImages(self, image1, image2):
17         im1=image1.getSliceAsImage()
18         im2=image2.getSliceAsImage()
19         result=ImageChops.difference(im1,im2)
20         return DataContainer.DataContainer(result)
```

Listing 1: DiffWorker

Listing 1 shows the DiffWorker. It takes two images and provides their difference image. The actual work is performed by PIL[14] (Lst. 1, l. 19). First of all the sockets are declared. These are the input facilities of the worker. They are declared with a name and a type. The latter is used to provide visual hints to the user, the former to identify the socket later (Lst. 1, l.16). Note that there is no declaration of output facilities. Those are referred to as plugs and are immediately coupled with a calculation method. This can be observed in line 15 et sqq. There we define diffImages, a common method that takes two arguments. Their names coincide with the aforementioned socket names. The author of the worker is not required to deal specially with his input. He simply declares a method and Pyphant takes care of the data-handling necessary. All he has to do is prefix his plug with the Worker.plug decorator, declaring the return type.

In front of the actual class definition we see a factory function (Lst. 1, l.6) and the WORKER_INFO constant, that carries a reference to the factory. This constant must be provided by every worker module and is used by the framework to identify the worker prior to its instantiation, for example in the toolbar you see on the right hand side of Fig. 3.

4.1.2 The General Worker

In section 4.1.1 we have presented an example for a simple worker. Let's take a look at excerpts from the worker module:

```python
def plug(returnType):
    def setPlug(plug):
        setattr(plug, 'isPlug', True)
        setattr(plug, 'returnType', returnType)
        return plug
    return setPlug
```

Listing 2: Plug decorator of Worker module

This is the plug decorator. It is merely used as a marker, adding some meta information to the plug method. These are employed by the WorkerFactory metaclass (Lst. 3) for finding the plugs, which are subsequently utilised to construct the plug instances, among the methods of a worker.

```python
class WorkerFactory(type):
    def __init__(cls, name, bases, cdict):
        cls._plugs=[]
        for f in filter(lambda key : isPlug(key, cdict), cdict):
            cls._plugs.append((f, cdict[f]))
        super(WorkerFactory, cls).__init__(name, bases, cdict)
```

Listing 3: WorkerFactory metaclass of Worker module

Next we have the Worker class itself. Of special interest is the construction. At runtime of the Worker.__init__ method every Worker instance will have three attributes: _sockets, _plugs and _params. All of them are lists, describing the respective, requested objects. You have seen the _sockets list being filled in Listing 1. The list of plugs is constructed by the WorkerFactory metaclass on basis of the plug decorators (Lst. 3). The _params list contains parameter descriptions, and is filled like the socket list if the worker has parameters. Actually parameters are a special type of socket but we will come back to this in Section 4.2.

For every entry in those lists a corresponding connector instance is created as member of the worker. This way the Worker class functions as a factory for its own connectors.

4.2 The Connectors

The Connectors module defines the type constants and the FullSocketError, which is raised when someone attempts to insert a plug into an already used socket.

Apart from that this is of course the place to find the connector classes, i.e. Connector, Socket, and Plug.

```
13
14 class Connector(object):
15     def __init__(self, worker, name, type=DEFAULT_DATA_TYPE):
16         self.worker=worker
17         self.name=name
18         self.type=type
19         self._isExternal=True
20     def _getIsExternal(self):
21         return self._isExternal
22     def _setIsExternal(self, isExternal):
23         if isExternal!=self.isExternal:
24             self._isExternal=isExternal
25             self.worker.connectorsExternalizationStateChanged(self)
26     isExternal=property(_getIsExternal, _setIsExternal)
```

Listing 4: Connector class of Connectors module

In Listing 4 you see the Connector class. It is the base class for sockets and plugs. As you can see every connector carries a reference to its worker (Lst. 4, l.16) as well as an identifying name (l.17). Furthermore there is the isExternal property (Lst. 4, l.19). It denotes whether the connector is exposed to input from outside the worker. It might not be external if either a default value is available, a socket is not needed or a plug is not available due to the specific configuration of a worker.

The socket plays host to at most one plug and keeps track of the connection. If a connection is broken or the respective plug becomes invalid the socket will invalidate itself and its worker, which in turn invalidates all its plugs. This way the invalidation propagates through the recipe until all concerned workers are informed.

Finally we have the Plug. It is perhaps the most interesting connector since it is the one responsible for the multithreading.

```
class Computer(threading.Thread):
    def __init__(self, method):
        threading.Thread.__init__(self)
        self.method=method
        self.result=None
    def run(self):
        if self.method:
            self.result=self.method()
```

Listing 5: Computer class of connectors module

```
def createWrapper(method):
    args,varargs,varkw,defaults=inspect.getargspec(method)
    sockets=args[1:]
    name=method.func_name+'PyphantWrapper'
    l='def '+name+'(method=method):\n'
    for s in sockets:
        l+='\t'+s+'=Computer(method.im_self.getSocket("'+s+'").
            getResult)\n'
    for s in sockets:
        l+='\t'+s+'.start()\n'
    for s in sockets:
        l+='\t'+s+'.join()\n'
    l+='\treturn method( '#<-That Space is very important
    for s in sockets:
        l+=s+'='+s+'.result ,'
    l=l[:-1]+')\n'
    exec l
    return eval(name)
```

Listing 6: createWrapper helper of connectors module

In order to accomplish threading the framework needs a little help from

1. the Computer class (Listing 5) and
2. the createWrapper method (Listing 6).

While the *Computer* class encapsulates the thread running the various calculation tasks the *createWrapper* method is used to create at construction time of the plug a matching wrapper for the calculation method of the worker. To this end it constructs a method that starts one thread for every socket used by the plug and joins them back with the main thread prior to calling the plug itself with the fetched results as its arguments.

When the plug is queried via its getResult method it checks for an already available result, and only generates a new one when necessary, also handling the required locking transparently.

4.3 The Pyphant Execution Model

How is a Pyphant recipe executed? Actually it is not so much executed as evaluated. The naive approach to execution might be to determine an execution order for the graph, then executing each node in a top to bottom order. Pyphant instead starts from the bottom node(s) and fetches the required results of previous calculations. This way only needed results are calculated. For example the UltimatePointsCalculator provides two results:

1. An image, that shows the found extrema visually, and

2. a mere list of the found extrema.

While in a pure computer oriented recipe the list might be needed for further processing it can be convenient to have an immediate visual feedback on the success of the operation, for example to determine the usefulness of ones image preprocessing. However, only the requested result is calculated, thus saving time and computing power by avoiding the calculation of the entire node.

Furthermore this order of execution allows for an easy caching of already computed results: When a plug is queried it simply provides the last computed result without even bothering the worker unless it has been invalidated meanwhile.

Another feature of this execution approach is the simple implementation of multi-threading. In case a plug has no or only an invalidated result Pyphant retrieves the data from every socket used by that plug in parallel, each in its own thread. Thus a non-trivial recipe automatically leads to a pseudo-parallized execution within the restrictions imposed on Pyphant by the *global interpreter lock* [16].

4.4 The DataContainer

Most often when talking about data, we are talking about physical quantities. While there is a wide variety thatof, they share a couple of properties:

- They have a dimension unit like m, kg, s.
- Discrete data sets are usually composed of one or more abscissae and one or more ordinates, because for the same set of sampling points different physical quantities have been measured.
- Slices of the data are of interest. For example it is common to look at slices of a Computed Tomography to examine it.

These considerations led us to the idea of a common data exchange class, which found its first incarnation in the DataContainer.

The primary data format is the scipy array. One DataContainer comprises one scipy array. Any data used within a Pyphant application based on the DataContainer must be convertible to scipy arrays. In addition further formats may be supported. At the moment we support PIL images as a secondary data format for two dimensional arrays. The DataContainer provides a data property that always yields the array, a getSliceAsImage and a setImageAsSlice method. The idea for the future is to allow the extraction of arbitrary slices from the possibly multi-dimensional array. For now they simply treat the whole, two dimensional array as an image. The two formats are transparently and lazily converted, i.e. if the data is requested as an image it is converted to the image and all changes made to the images are kept there until the array is asked for and vice-versa. This is done to save computing time in the case where a couple of consecutive steps only deal with images until one worker converts it to the array which is used henceforth.

5 The User Interfaces

We employed a clean encapsulation of the core of Pyphant. This allows for a variety of User Interfaces (UI). For now we have implemented a simple Graphical User Interface (GUI) based on the wxPython toolkit and a Command Line Interface (CLI). You already got a glimpse at the GUI in Section 3. In this section we will elaborate on the technicalities of the GUI, which also gives a good example of a Pyphant application. Then we will discuss the CLI, which is designed to facilitate the application of one specific recipe to a set of similar data, e.g. to analyse the images of different runs of the same experiment.

5.1 wxPyphant - the Graphical User Interface

A screenshot of the Graphical User Interface (GUI) is seen in Fig. 3. On the left hand side of the client area of the window you see the canvas. This is were you put the desired workers and link them. The arrangement of the workers and their connections are conducted via an intuitive drag and drop interface. To get them on the canvas in the first place you drag the desired worker from the toolbar on the right onto the canvas. What happens behind the curtains is the following: The GUI has acquired a WORKER_INFO for every known worker and had constructed a corresponding representative in the toolbar. When this is dragged onto the canvas the factory method provided by that WORKER_INFO is called in order to construct a worker of that kind. Then a corresponding GUI object is created and integrated into the recipe. While the GUI as a whole is based on the wxPython toolkit, the canvas is based on the Object Graphics Library (OGL), which in its latest form is part of wxPython.

Apart from the construction of recipes wxPyphant allows for the immediate inspection of intermediate results by right clicking on the appropriate plugs. Upon a right click a menu is shown, that offers all suitable visualisations. These can be provided by visualisation plugins and are filtered according to the data type of the plug. Finally the GUI features the saving and loading of recipes. This is realised by simple pickling with a little bit of extra care with regards to method handling.

The WORKER_INFO objects are collected on startup of the GUI by the following auto discovery mechanism: By default Pyphant has a home directory which is called *.pyphant* and is located in the users home directory. Inside Pyphant's home directory there is a *workers* directory which contains an arbitrary number of *PyphantWorkerArchives* (PWA). By definition these are .tar.bz2 files, that contain a config.xml file which lists the provided workers. These are loaded and registered with a central management class, the so-called WorkerRepository using the respective WORKER_INFO constants.

5.2 The Command Line Interface

The Command Line Interface (CLI) is given a pickled recipe as argument and loads the recipe, by unpickling it. In a second step all nodes of the recipe are identified that have no plugs but only sockets. The results of these workers can be retrieved by calling the special method .execute on them. Workers of this kind are referred to as sinks. They are the most important distinction between recipes intended only for the GUI and those intended for the command line. To obtain a useful command line recipe one needs to include at least one sink. Unfortunately at the moment the CLI must be adapted for every new recipe in order to add the handling of command line parameters e.g. for changing an image filename. However this is expected to be resolved soon.

Thanks to the encapsulation of the core the CLI does not depend on the availability of a graphical environment at all, which allows to deploy recipes visually crafted on workstations into a powerful computing environment. This is especially important for more complex tasks, where the computation easily can take days.

6 Summary and Outlock

Pyphant is a flexible framework for the composition of data-flow models. It offers easy integration of new computing nodes and a multithreading execution of entire workflows without special burdens on the user. Its CLI allows for the application of carefully crafted recipes in a computing environment under the lack of graphical services or possibly the integration into completly different applications.

Concerning the application of the Pyphant framework we are planning to extend the *ImageProcessing* PWA by more tools and provide a PWA for solving ill-posed problems on basis of non-linear regularisation methods. Possibly the family of PWAs can be extended even further to entirely different projects in need of a similar GUI. Furthermore we are going to enhance the DataContainer as hinted in Sec. 4.4 and improve on the CLI in order to support command line parameters and their automatic coercion into the appropriate worker parameters.

Acknowledgement

The authors like to thank M. C. Röttger, J. Honerkamp, and Nico Bruns for fruitful discussions on the topic. The financial support by the German BMBF (Project No.: 03C0354A) is gratefully acknowledged.

References

[1] John C. Russ. *The Image Processing Handbook.* CRC Press, Boca Raton, 4 edition, 2002.

[2] John Paul Morrison. *Flow-based programming: A New Approach to Application Development.* VNR computer library. Van Nostrand Reinhold, New York, 1994. http://www.jpaulmorrison.com/fbp/index.shtml.

[3] Wikipedia. Visual programming language. http://en.wikipedia.org/wiki/Visual_programming_language.

[4] Michel F. Sanner, Daniel Stoffler, and Arthur J. Olson. ViPEr a Visual Programming Environment for Python. In *10th International Python Conference*, February 2002. `http://www.scripps.edu/~sanner/html/papers/IPC02.pdf`.

[5] Pietro Berkes and Tiziano Zito. Modular toolkit for Data Processing (MDP). http://mdp-toolkit.sourceforge.net, 2006.

[6] M. F. Sanner, B. S. Duncan, C. J. Carrillo, and A. J. Olson. Integrating Computation and Visualization for Biomolecular Analysis: An Example Using Python and AVS. In *Proc. Pacific Symposium in Biocomputing '99*, pages 401–412, 1999.

[7] J. Honerkamp and J. Weese. A nonlinear regularization method for the calculation of relaxation spectra. *Rheologica Acta*, 32(65):73, 1993.

[8] T. Roths, M. Marth, J. Weese, and J. Honerkamp. A generalized regularization method for nonlinear ill-posedproblems enhanced for nonlinear regularization terms. *Computer Physics Communication*, 139:279–296, 2001.

[9] M. Bohnert, R. Walther, T. Roths, and J. Honerkamp. A Monte Carlo-based model for steady-state diffuse reflectance spectrometry in human skin: estimation of carbon monoxide concentration in livor mortis. *Int J Legal Med*, 119:355–362, 2005.

[10] R. Backofen, H.-G. Borrmann, W. Deck, L. De Raedt, K. Desch, M. Diesmann, M. Geier, A. Greiner, W. R. Hess, J. Honerkamp, St. Jankowski, I. Krossing, A. W. Liehr, A. Karwath, R. Klöfkorn, R. Pesché, T. Potjans, M. C. Röttger, L-Schmidt-Thieme, G. Schneider, B. Voß, B. Wiebelt, P. Wienemann, and V.-H. Winterer. A bottom-up approach to grid-computing at a university: the black-forrest-grid initiative. *Praxis der Informationsverarbeitung und Kommunikation*, 29(2):81–89, 2006.

[11] John Hunter. Matplotlib. `http://matplotlib.sourceforge.net`, 2006.

[12] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C. The Art of Scientific Computing.* Cambridge University Press, Cambridge, 2 edition, 1996.

[13] Nico Bruns, Jonas Scherble, Laura Hartmann, Ralf Thomann, Béla Iván, Rolf Mühlhaupt, and Joerg C. Tiller. Nanophase Separated Amphiphilic Conetwork Coatings and Membranes. *Macromolecules*, 38:2431–2438, 2005.

[14] Secret Labs AB. Python Imaging Library (PIL). `http://www.pythonware.com/products/pil`.

[15] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing.* California Technical Publishing, San Diego, 1997. `http://www.dspguide.com/pdfbook.htm`.

[16] Peyton McCollough. Basic threading in python. `http://www.devshed.com/c/a/Python/Basic-Threading-in-Python`, 2005.