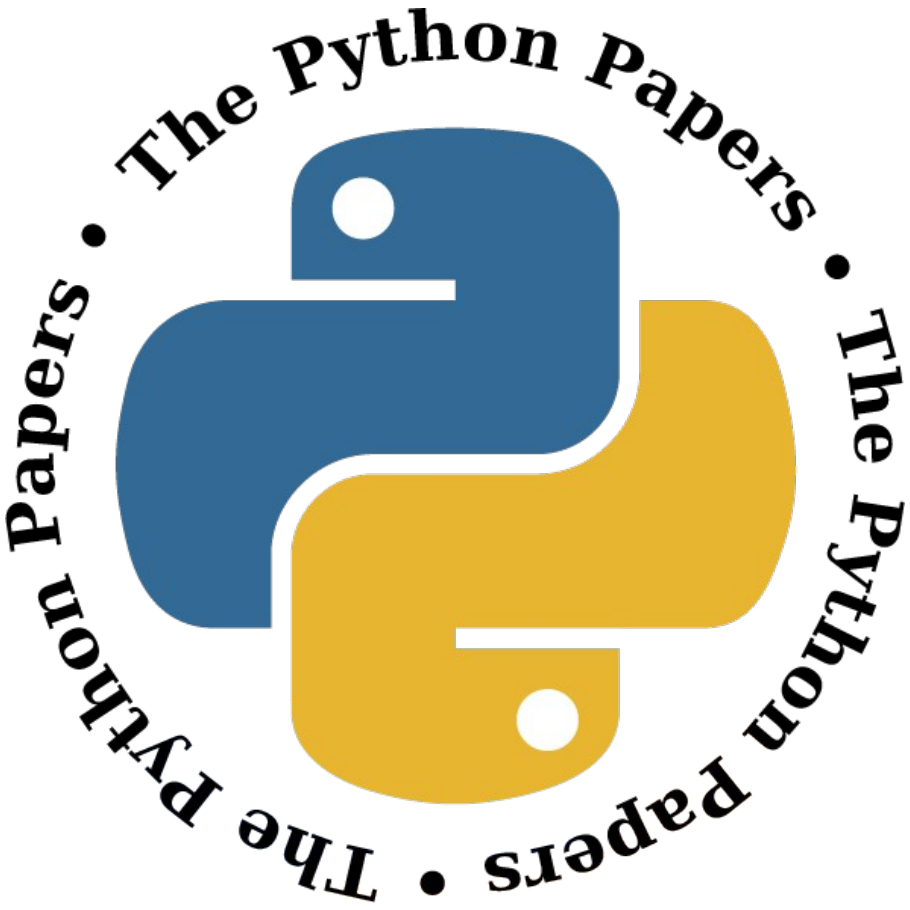
The Python Papers



Volume 2, Issue 3
pythonpapers.org

Journal Information

The Python Papers

ISSN: 1834-3147

Editors

Referencing Information

Articles from this edition of this journal may be referenced as follows:

Author, "Title" (2007) *The Python Papers*, Volume N, Issue M, pp. m:n

e.g. Maurice Ling, "Firebird Database Backup by Serialized Database Table Dump" (2007) *The Python Papers*, Volume 2, Issue 1, pp. 7:15.

Copyright Statement

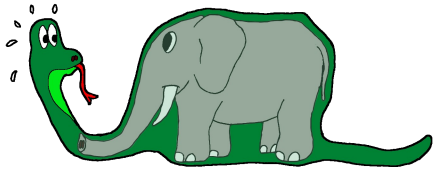
The Python Papers was first published in 2006 in Melbourne, Australia.

Referees

An academic peer-review was performed on all academic articles. A list of reviewers will be published in each December issue. This has been done in order to ensure the anonymity of reviewers for each paper.

Pyphant – A Python Framework for Modelling Reusable Information Processing Tasks

Klaus Zimmermann, Lorenz Quack and Andreas W. Liehr



We are presenting the Python framework “Pyphant” for the creation and application of information flow models. The central idea of this approach is to encapsulate each data processing step in one unit which we call a **worker**. A worker receives input via **sockets** and provides the results of its data processing via **plugs**. These can be connected to other workers' sockets. The resulting directed graph is called a **recipe**. Classes for these objects comprise the Pyphant core. To implement the actual processing steps, Pyphant relies on third-party plug-ins which extend the basic worker class and can be distributed as Python eggs. On top of the core, Pyphant offers an information exchange layer which facilitates the interoperability of the workers, using Numpy objects. A third layer comprises textual and graphical user interfaces. The former allows for the batch processing of data and the latter allows for the interactive construction of recipes.

This paper 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.

[*This paper was originally presented at Europython 2006 and has been updated for this publication. Full acknowledgements are included at the end of this article. -Ed*]

1. Introduction

Working as a 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 that of image processing[7]. Consider that you have previously developed an algorithm, which determines the particle size distribution of a certain blend of materials on the basis of an Atomic Force Microscopy (AFM) measurement. Given the measurement of a different material, it is likely that you would have to apply different processing steps in order to match the characteristics of the new sample. If you consider a programming environment which assists the adaptation of this data analysis algorithm, you will very quickly consider a flow-based programming paradigm. This approach was invented in the late sixties[8] and has become established. This can be seen from the variety of commercial and open-source tools applying flow-based programming in the context of visual programming languages[9]. As regards data analysis, several flow-based environments have been implemented in Python, including ViPEr[10] and the Modular Toolkit for Data Processing

7. John C. Russ. *The Image Processing Handbook*. CRC Press, Boca Raton, 4 edition, 2002.
8. 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.
9. Wikipedia. *Visual programming language*. http://en.wikipedia.org/wiki/Visual_programming_language
10. 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

(MDP)[11]. It has also been demonstrated that Python is well-suited to integrate several different software tools, e.g., for computation and visualization into a consistent data analysis environment[12].

Inspired by 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 materials research[13] [14] [15], have to be included into a consistent visual programming environment, we started to think about the Pyphant framework. A major prerequisite was that the resulting environment should be suitable not only for the creative work of a specialized scientist but also for standardized data processing in a daily laboratory routine or a large-scale data analysis campaign, computed in a grid computing environment. An attempt to balance these needs resulted 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 GUI **wxPyphant**[16]. The interactive evaluation of the algorithm is established using an extensible set of **visualisers**. Finally, the algorithm and its intermediary result can be saved in the Hierarchical Data Format HDF5[17]. The resulting file is also the basis for Command Line Interfaces (CLI), which can be tailored as Python scripts.

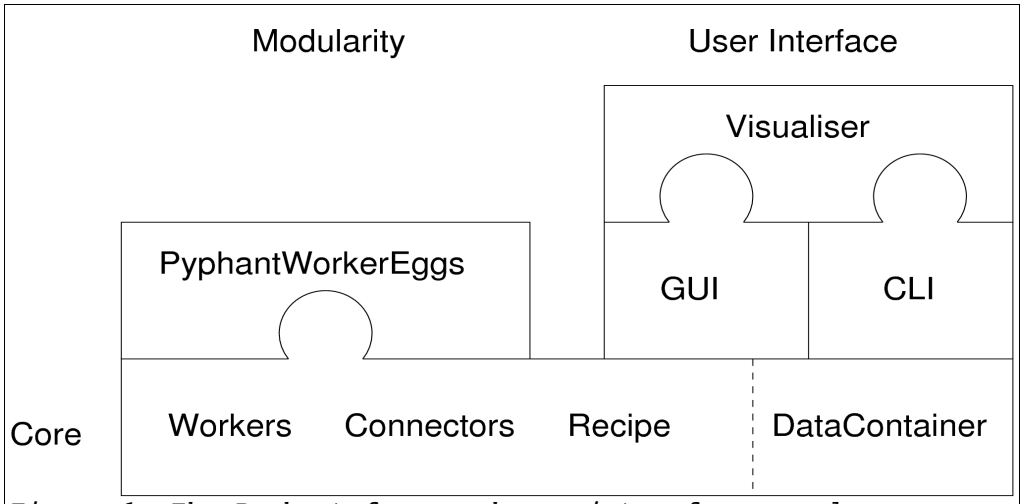


Figure 1. The Pyphant framework consists of a core layer comprising worker, connector, recipe and DataContainer objects. Workers for specialised data processing tasks can be provided by PyphantWorkerEggs. User interfaces are provided in the form of the GUI wxPyphant and a command-line interface for the individual recipe.

11. Pietro Berkes and Tiziano Zito. *Modular toolkit for Data Processing (MDP)*. http://mdp- toolkit.sourceforge.net, 2006.
12. 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*, pp 401-412, 1999.
13. J. Honerkamp and J. Weese. *A nonlinear regularization method for the calculation of relaxation spectra*. Rheologica Acta, 32(65):73, 1993.
14. T. Roths, M. Marth, J. Weese, and J. Honerkamp. *A generalized regularization method for nonlinear ill- posed problems enhanced for nonlinear regularization terms*. Computer Physics Communication, 139:279-296, 2001.
15. 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.
16. Servicegroup Scientific Information Processing, http://pyphant.sourceforge.net
17. The HDF Group (THG). *Hdf5 (hierarchical data format 5) software library and utilities*. http://hdf.ncsa.uiuc.edu/HDF5, 2006.

This paper begins 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.

2. Framework

Pyphant is a layered, plugin-based framework suitable for the modelling and execution of a wide range of information processing tasks. It is built on the idea that many computing 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 a recipe, following the famous textbook **Numerical Recipes**[18]. In this context the development of a data analysis algorithm can be pictured as the composition of a meal from various 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 which are usually 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 amongst workers, we added this container format to enhance the interoperability of the various workers.

Above the core we find the user interface layer (UI-layer), which comes in two flavours. We have implemented a simple GUI called wxPyphant which realises the visual programming paradigm. Pyphant also facilitates the easy creation of a command-line interface (CLI) for a specific recipe. This feature is very useful for a daily laboratory routine or the analysis of large data sets, e.g. if large scale data mining is performed in a grid computing environment.

18. 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.

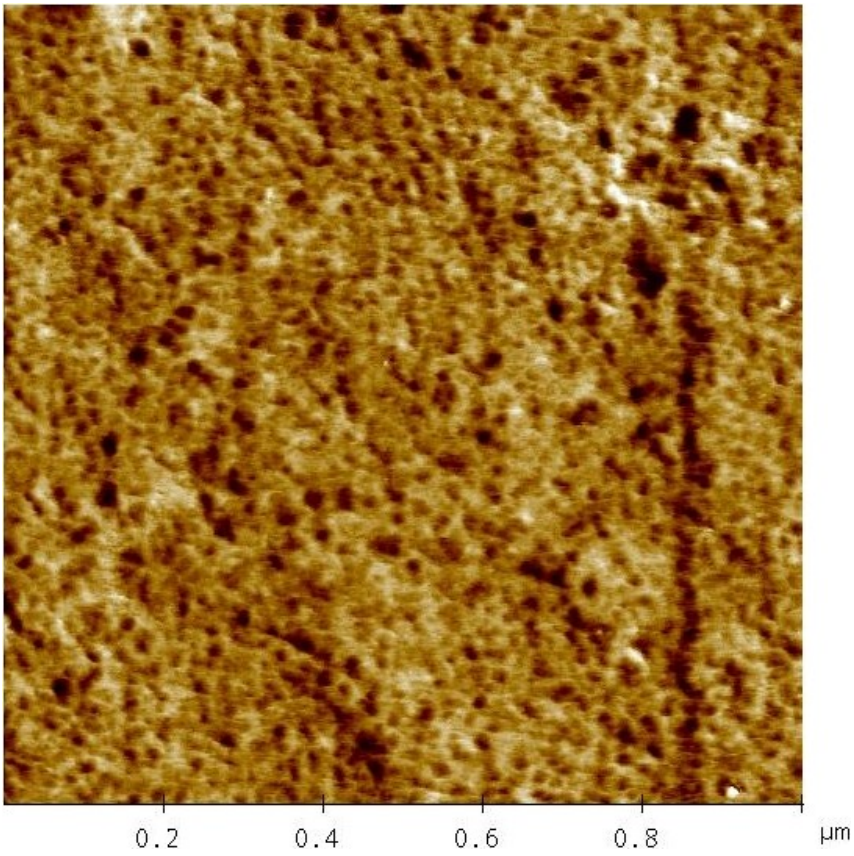


Figure 2. Atomic Force Microscopy (AFM) phase mode image1,14of an amphiphilic poly(2-hydroxyethyl acrylate)-l-poly(dimethylsolixane) (PHEA-l-PDMS) conetwork with 23 wt% PDMS. PHEA shows light and PDMS dark.

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. For now it suffices to notice that workers are bundled in PyphantWorkerEggs. Another kind of plugin is the visualisation plugin which is used to visualise data in a suitable way.

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 the steps needed to estimate the width distribution of an aggregated polymer phase from an AFM phase mode image. The example starts with loading the primary data, pre-processing 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(dimethylsolixane) (PHEA-*l*-PDMS) conetwork with 23 wt% PDMS[19]. In this visualization

19. Nico Bruns, Jonas Scherble, Laura Hartmann, Ralf Thomann, Bela Ian, Rolf Mülhaupt and Joerg C. Tiller. *Nanophase Separated Amphiphilic Conetwork Coatings and Membranes*. Macromolecules 38 pp 2431-2438, 2005.

the PHEA and PDMS show light and dark respectively. The task is to determine the with of the PDMS phase.

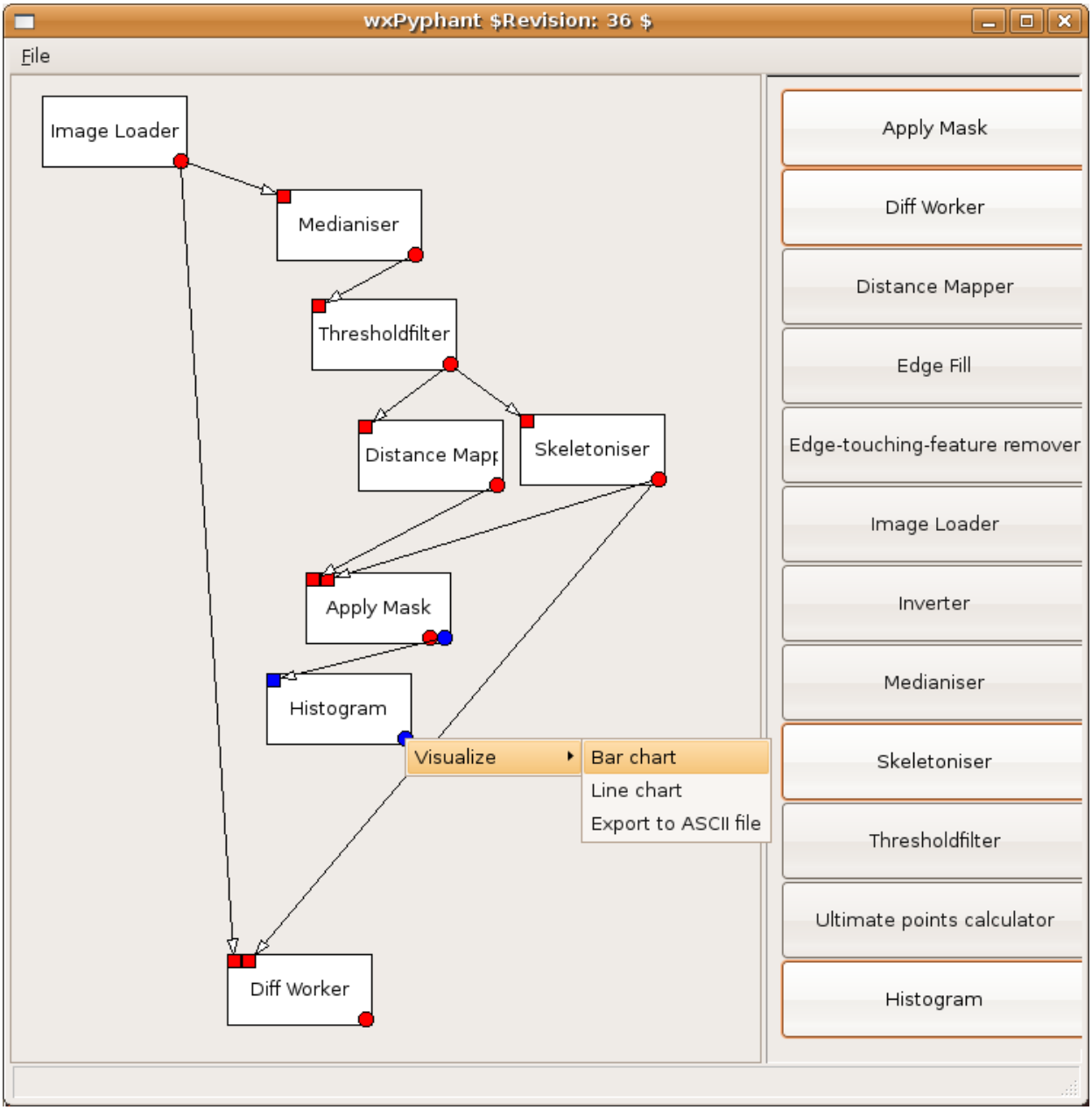


Figure 3. Pyphant recipe composed in the wxPyphant GUI. Workers are visualised as white boxes with sockets placed in their upper-left corner and available plugs placed in their lower-right corner. By right-clicking a plug, a context menu with visualisation plugins is provided.

The complete Pyphant recipe is depicted as a snapshot of the GUI in Fig. 3. On the right-hand side of the GUI, the toolbox of available workers is visualised. Each worker can be placed by drag-and-drop on the canvas. The individual workers are clearly visible 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 DataContainer. Red indicates a FieldContainer, while blue denotes a SampleContainer.

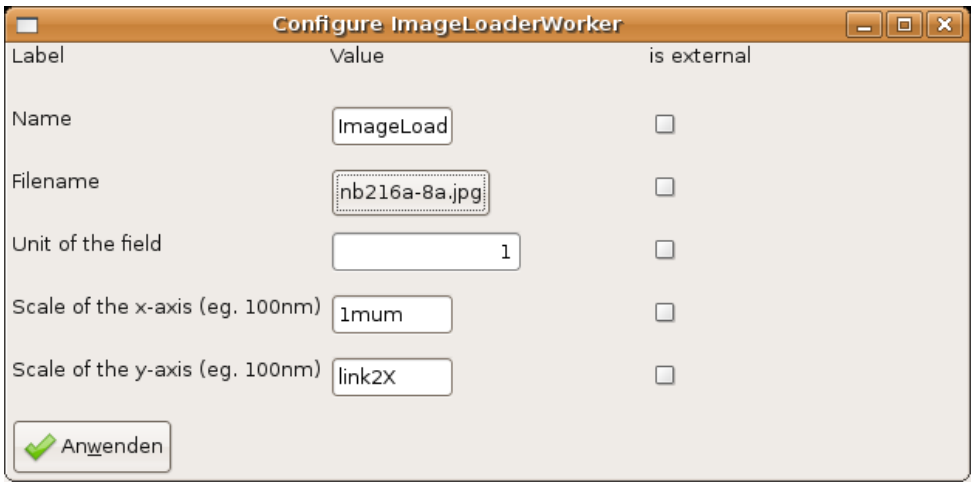


Figure 4. Configuration dialog for the ImageLoaderWorker which is automatically constructed from the definition of the ImageLoader class. It enables the interactive adjustment of all parameters, including the physical dimensions and units of the investigated sample. Note that the is external check-boxes are an experimental feature which allows the respective parameter to be set via a socket.

Please note the context menu emerging from the plug of the HistogramWorker. It enables the interactive examination of the computed results via visualisation plugins. Let's have a short look at the algorithm:

1. Loading the image

   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 (Fig. 4). This scheme holds for all configuration dialogs of all workers. The loaded image is provided as a gray-scale image at the red plug. As the worker internally uses the Python Imaging Library (PIL[20]), it supports a wide variety of file formats.

2. Removing noise

   Next we want to remove noise from the image. For this task, the PILMedianWorker is applied, which implements a standard median filter[21]. It can be configured by the size of the applied kernel and the number of smoothing runs. Here we have chosen a 5x5 kernel and five smoothing runs.

3. Applying a threshold

   Now we want to separate the dark features which represent the PDMS phase from the background. This is achieved through the ThresholdingWorker. It compares every pixel of the smoothed image with a given threshold and returns a binary image such that the pixels which comprise 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

20. Secret Labs AB. *Python Imaging Library (PIL)*. http://www.pythonware.com/products/pil
21 cf. [1], p. 152ff

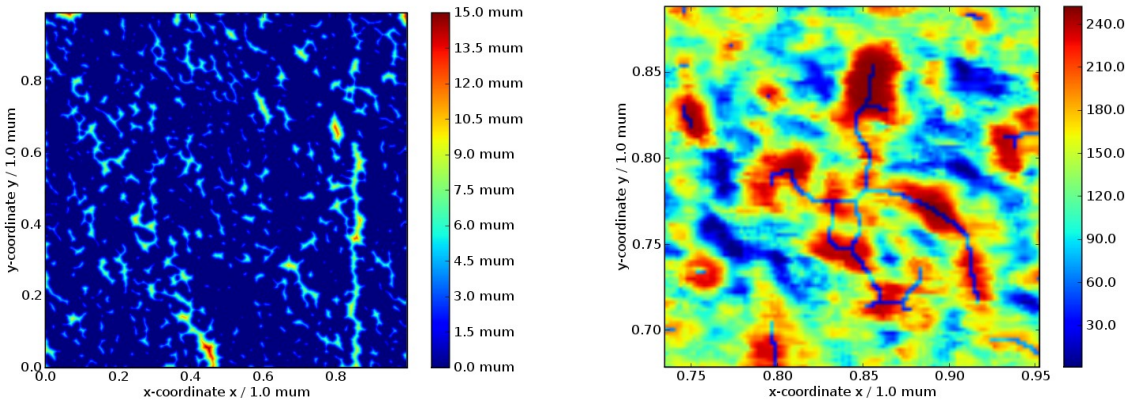


Figure 5. (a) Visualisation of DistanceMapper result. The feature size is colour-coded. (b) Display detail of difference between AFM image and skeleton of found features. Here the PDMS phase shows red while the skeleton is represented by blue lines.

By now we have a binary image representing the features we are taking into account. Next we would like to determine their size by calculating the distance of each pixel to the nearest background pixel[22]. This task is done by the DistanceMapper. The resulting gray-scale image is shown in Fig. 5a with pseudo-colours. Note the correct labelling of the colour palette indicating the distance of each feature pixel from the background.

5. Morphological transform

   In order to retrieve the width of the features, they are skeletonised. This is achieved by iteratively removing the outer pixels of each feature until the core pixels remain[23].

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. A display detail of the result is depicted in Fig. 5b.

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 an Nx3 table representation. Here, each skeleton pixel is specified by its spatial position and the respective feature width.

8. Computing the histogram

22 cf. [1], p. 427ff.
23. Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, San Diego, 1997, Chapter 25. http://www.dspguide.com/pdfbook.htm

By now, the recipe produces the information we are interested in. The only thing left to do is 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. 6. The width distribution of the PDMS phase determined by the

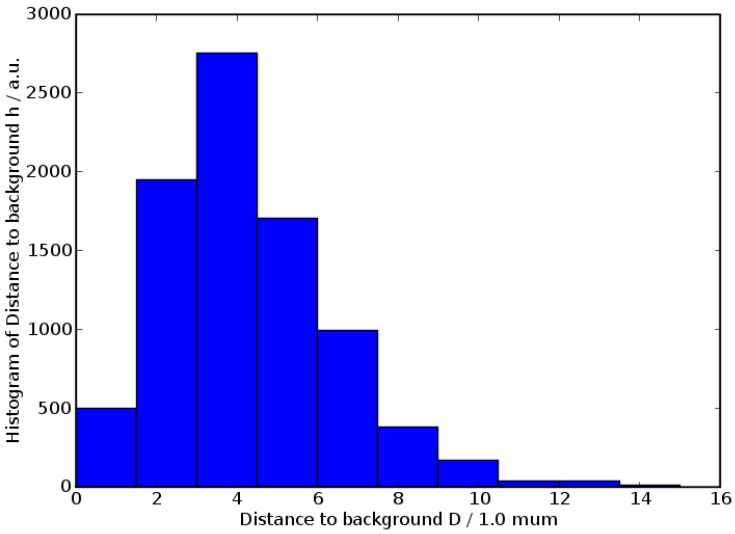


Figure 6. Width distribution of the PDMS phase. The result obtained from the AFM phase mode image depicted in Fig. 2 with the Pyphant recipe shown in Fig. 3. PILMedianWorker: 5x5 kernel, 5 runs. ThresholdWorker: threshold 90.

Pyphant recipe matches the results of Bruns et al[24].

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. It is designed to maximise worker interoperability.

4.1 The Worker

4.1.1 The DiffWorker – a Practical Example

Listing 1 shows the DiffWorker. It takes two images and provides their difference image. First, some general attributes are declared (Lst 1, 1ine.44).

API is a hint to Pyphant on how to invoke the worker.

VERSION is related to the semantics of the worker. It should be increased whenever it is altered.

REVISION identifies the specific revision of the worker in the subversion repository.

name is the name of the worker as presented to the user. This would be the place for internationalisation (i18n). Omission of this information indicates an abstract worker, i.e. one that will not be presented to the user. An example for this is the fundamental Worker class.

Next, the sockets are declared. These are the input facilities of the worker. They are declared

24 cf. [19]

with a name and a type (Lst 1, line 48). The name is used to identify the socket. The type is used to provide visual hints to the user. Note that there is no declaration of output facilities. Those are referred to as plugs and are immediately coupled with a calculation method. From the implementors point of view, the leading parameters are ordinary parameters. If the worker is part of a recipe, Pyphant will call the method at a good time with appropriate arguments. The last parameter, subscriber, is special because it acts as a simple feedback facility for the progress meter. Therefore, the method should write information about its completion in percent at meaningful points in the calculation. Pyphant will supply a property-like object at runtime that is used to inform the user about the overall progress of the calculation.

Note that the names of the parameters coincide with the aforementioned socket names. This is necessary for Pyphant to figure out which socket should be associate with which parameter. The author of the worker is otherwise not required to deal specially with the input. The author simply declares a method and Pyphant takes care of all the data-handling necessary. All that is required is to prefix the plug with the Worker.plug decorator, declaring the return type (Lst 1, 1ine 51).

```
40 from pyphant.core import (Worker , Connectors)
41 import copy, scipy
42
43 class DiffWorker (Worker.Worker) :
44     API = 2
45     VERSION = 1
46     REVISION = "$Revision:␣28␣$"[11:-1]
47     name=" Diff␣Worker "
48     _sockets = [ ("image1" , Connectors.TYPE_IMAGE) ,
49         ("image2" , Connectors.TYPE_IMAGE) ]
50
51     @Worker.plug(Connectors.TYPE_IMAGE)
52     def diffImages(self, image1, image2, subscriber=0) :
53         im1=image1.data
54         im2=image2.data
55         diff=scipy.absolute(im1-im2)
56         result=copy.deepcopy(image1)
57         result.data=diff
58         result.seal()
```

Listing 1: The DiffWorker class which is included in Pyphant's ImageProcessing toolbox

4.1.2 The Worker Module

In section 4.1.1 we have presented an example for a simple worker, which like all of Pyphant's *productive* workers, inherits from the Worker class of the eponymous module (Lst 1, line 43). A small but important function, which the Worker module provides, is the plug decorator (Lst 2). It is merely used as a marker which adds the attributes `isPlug` and `returnType` to the plug-deploying method of the respective worker (e.g. Lst 1, line 51). This meta-information is used by the WorkerFactory metaclass (Lst 3) for gathering all plugs in the list `_plugs`.

```
51 def plug(returnType) :
52     def setPlug(plug) :
```

```
53              setattr(plug, 'isPlug', True)
54              setattr(plug, 'returnType', returnType)
55              return plug
56 return setPlug
```

Listing 2: Plug decorator of Worker module

Next, we have the worker class itself. Of special interest is the construction. At runtime, the `Worker.__init__` method will, for every worker instance, set the 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, line 69). 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 `Worker.__init__` creates a corresponding connector instance as a member of the Worker instance.

```
64 class WorkerFactory(type):
65       workerRegistry=WorkerRegistry.WorkerRegistry.getInstance()
66       log=logging.getLogger("WorkerFactory")
67       def __init__(cls, name, bases, cdict):
68            cls._plugs=[]
69            for f in filter(lambda key : identifyPlugs(key, cdict), cdict):
70                 cls._plugs.append ((f,cdict[ f ]))
71            super(WorkerFactory, cls).__init__(name, bases, cdict)
72            try:
73                 WorkerFactory.workerRegistry.registerWorker(WorkerInfo (cls.name, cls))
74            except (AttributeError):
75                 WorkerFactory.log.warning("Ignoring␣worker␣"+name+"␣due␣
                       to␣missing␣name␣attribute.")
```

Listing 3: WorkerFactory metaclass of Worker module.

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, the connector classes are also found here (i.e. Connector, Socket and Plug).

```
54 class Connector(object):
55      def __init__(self, worker, name, type=DEFAULT_DATA_TYPE) :
56            self.worker=worker
57            self.na me=name
58           self.type=type
59            self._isExternal=True
60      def _getIsExternal(self):
61            return self._isExternal
62      def _setIsExternal(self, isExternal):
63            if isExternal != self._isExternal:
64                  self._isExternal=isExternal
65                  self.worker.connectorsExternalizationStateChanged(self)
66      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, line 56) as well as an identifying name (line 57). Furthermore, there is the `_isExternal` property (line 59) which denotes whether the connector is exposed to input from outside the worker.

The Socket takes the connecting plugs and keeps track of the connection. If any connection is broken, or the respective plug becomes invalid, the socket will invalidate itself and its worker, which in turn invalidates all of 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 multithreading. 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 at the construction time of the plug to create 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 a plug is queried via its `getResult` method, it checks for an already available result, generates a new one if necessary and handles the required locking transparently.

```
68 class Computer(threading.Thread):
69     def __init__(self, method, **kwargs):
70         threading.Thread.__init__(self)
71         self.method=method
72         self.kwargs=kwargs
73         self.result=None
74     def run(self):
75         if self.method:
76             self.result=self.method(subscriber=self.kwargs["subscriber"])
```

Listing 5: Computer class of connectors module.

```
88 def createWrapper(self, method):
89     args, varargs, varkw, defaults=inspect.getargspec(method)
90     sockets=args[:-1]
91     name=method.func_name+'PyphantWrapper'
92     l='def '+name+'(subscriber, method=method, process=self):\n'
93     for s in sockets:
94         l+='\t'+s+'=Computer(method.im_self.getSocket("'+s+'").getResult,
           subscriber=subscriber)\n'
95     for s in sockets:
96         l+='\t'+s+'.start()\n'
97     for s in sockets:
98         l+='\t'+s+'.join()\n'
99     l+='\tdef updater(percentage):\n'
```

```
100     l+='\t\tsubscriber.updateProcess(process,␣percentage)\n '
101     l+='\treturn method(␣subscriber=property(fset=updater),' # If
          no sockets are needed that comma will be erased, so do
          not add a space!
102     for s in sockets:
103         l+=s+'='+s+'.result,'
104         l=l[:-1]+')\n'
105     exec l
106     return eval(name)
```

Listing 6: createWrapper helper of connectors module.

4.3 The Pyphant Execution Model

How is a Pyphant recipe executed? Actually, it is not so much executed as evaluated. The naïve approach to execution might be to determine an execution order for the graph, then execute each node in a top-to-bottom order. Pyphant instead starts from the bottom node and fetches the required results of previous calculations. For example, the `UltimatePointsCalculator` provides two results:

1. An image that shows the found extrema visually; and
2. a 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 immediate visual feedback on the success of the operation, e.g., to determine the usefulness of the 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-parallelised execution within the restrictions imposed on Pyphant by the *global interpreter lock*[25].

4.4 Information Exchange and Visualisation

In most cases, the scientific community deals with normalised physical quantities in the form of pure numbers. This is of course very convenient because it allows the immediate application of a wide variety of methods and algorithms. The disadvantage is that the scientific information is stripped of its meaning and is reduced to pure data, such that the labels of a result-presenting graph have to be recompiled by hand if the primary data or the applied algorithm change.

Therefore we were seeking a self-descriptive data exchange format which encourages the annotation of data right from the start and enables the entrainment of the physical units involved in the algorithm, such that a well-labeled graph can be produced without effort. The result is the DataContainer module which reproduces the self-descriptiveness of the network Common Data Form (netCDF[26]) but is augmented in the following respects: once sealed, a

25. Peyton McCollough. *Basic threading in python*. http://www.devshed.com/c/a/Python/Basic-Threading-in-Python, 2005.
26. Unidata. *netcdf (network common data form)*. http://www.unidata.ucar.edu/software/netcdf, 2003

DataContainer is immutable and can be identified by its **emd5** attribute (Enhanced MD5). This is a unique identifier composed of information about the origin of the container, its type and its MD5 hash[27] to ensure its integrity. It is also used to store the container as part of a recipe in an hdf5 file. Note that Pyphant does not impose any restrictions on the data exchange format, such that the described DataContainer is *just* a convenient interface for exchanging scientific information between workers (e.g., Lst 1) and visualisers (e.g., Lst 7).

The module DataContainer provides the basic class DataContainer which has the following attributes:

`longname:` Notation of the data, e.g. 'electric field', which is used for the automatic annotation of charts.

`shortname:` Symbol of the physical variable in LaTeX notation, e.g. `E_\alpha`, which is also used for the automatic annotation of charts.

`id:` Identifier in Enhanced MD5 (emd5) format
`emd5://NODE/USER/DATETIME/MD5-HASH.TYPESTRING`
which is set by calling the method `seal` and indicates that the stored information is unchangeable.

`label:` Typical axis description, composed from the meta information of the DataContainer.

`data:` Data object, e.g. Numpy array[28].

On top of the DataContainer we have defined a `FieldContainer` and a `SampleContainer` class. The FieldContainer stores an *n*-dimensional array together with its unit and the coordinates of the independent variables, which are called dimensions and in turn are represented as FieldContainers. The SampleContainer combines different FieldContainers which have the same number of sample points to a table-like representation.

The FieldContainer is characterised by the following properties:

`data:` Numpy.array representing the sampled field.

`unit:` PhysicalQuantity object[29] denoting the unit of the sampled field.

`dimensions:` List of FieldContainer instances describing the dimensions of the sampled field.

`data:` Sampled field stored as Numpy.array.

`error:` Absolute error of the sampled field stored as Numpy.array.

A SampleContainer can be regarded as a table which is typically obtained from measurement campaigns. It stores different observations on the same subject per row, whereby each column comprises a quantity of the same kind. From a statistical point of view, each row is the realisation of a random variable (sample). A SampleContainer is constructed from a list of FieldContainers and provides the following attributes:

`data:` table of samples stored in a Numpy.recarray;

`desc:` description of Numpy.dtype of the recarray; and

`units:` list of PhysicalQuantities objects denoting the units of the columns.

An example of a visualiser using the meta information provided by the FieldContainer is given in the following listing. A graph compiled by this visualiser is depicted in Fig. 5. From line 40

27. R. Rivest. *The md5 message-digest algorithm*. http://tools.ietf.org/html/rfc1321, 1992
28. Travis E. Oliphant. *Guide to Numpy*. Trelgol Publishing, http://www.tramy.us, 2005
29. Konrad Hinsen. *Scientific python.* http://dirac.cnrs-orleans.fr/plone/software/scientificpython/, 2007

of the ImageVisualiser module, you can see that we utilise the pylab module[30] in order to visulise our results. However, in order to make a visualisation class available to Pyphant's GUI, it has to register itself via the `DataVisReg` registry (Lst 7, line 74), from which the context menus of the plugs are constructed by means of the respective `name` attributes (Lst 7, line 45).

While the graph itself is created by a simple call to pylab's `imshow` procedure, the majority of code is spent on annotating the graph: (Lst 7, line 57):

`11.52-55:` Determining the reference coordinate system from the dimension attribute of the FieldContainer

`11.58-59:` Labelling the axis with the meta information of the dimension attribute

`11.61-70:` Creating the colour bar, which maps the false colours of the image to the amplitude of the visualised field.

```
40 import pylab, scipy
41 from pyphant.core.Connectors import TYPE_IMAGE
42 from pyphant.wxgui2.DataVisReg import DataVisReg
43
44 class ImageVisualizer(object):
45       name='Image␣Visualizer'
46       def  __init__(self, fieldContainer):
47           self.fieldContainer=fieldContainer
48           self.execute()
49
50       def execute(self):
51           self.figure=pylab.figure()
52           xmin=scipy.amin(self.fieldContainer.dimensions[0].data)
53           xmax=scipy.amax(self.fieldContainer.dimensions[0].data)
54           ymin=scipy.amin(self.fieldContainer.dimensions[1].data)
55           ymax=scipy.amax(self.fieldContainer.dimensions[1].data)
56
57           pylab.imshow(self.fieldContainer.data, extent=(xmin , xmax,ymin, ymax))
58           pylab.xlabel(self.fieldContainer.dimensions[0].label)
59           pylab.ylabel(self.fieldContainer.dimensions[1].label)
60
61           class F(pylab.Formatter):
62                def __init__(self, container, args, **kwargs):
63                     self.container=container
64                def __call__(self, x, pos=None):
65                     try:
66                          return str(x*self.container.unit)
67                     except IndexError, error:
68                          return str(x)
69           ax=pylab.gca()
70           pylab.colorbar(format=F(self.fieldContainer))
71           pylab.ion()
72           pylab.show()
73
74 DataVisReg.getInstance().registerVisualizer(TYPE_IMAGE, ImageVisualizer)
```

Listing 7: Excerpt from the ImageVisualizer module showing the homonymous class, which is used for visualizing the results depicted in Fig. 5.

30. John Hunter. *Matplotlib*. http://matplotlib.sourceforge.net, 2006.

5 The User Interfaces

We employed a clean encapsulation of the core of Pyphant. This allows for a variety of user interfaces. For now, we have implemented a simple GUI based on the wxPython toolkit. 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 a short example on how to incorporate a Pyphant recipe into a simple Python script.

5.1 wxPyphant – the Graphical User Interface

A screenshot of the GUI is seen in Fig. 3. On the left hand side of the client area of the window you see the canvas. This is where you put the desired workers and link them. The available workers are discovered at startup of the Pyphant framework via the entry points of the respective Python eggs[31] and are presented at the right hand side of the wxPython window.

The arrangement of the workers and their connections are conducted via an intuitive drag-and-drop interface. By placing a worker onto the canvas, the following mechanism is triggered: the factory method provided by the corresponding WorkerInfo object is called in order to construct a worker of that kind. Then a corresponding GUI object is created and integrated into the recipe. Sockets are represented by coloured boxes at the upper-left corner of the workers, while plugs are represented by coloured circles in the lower-right corner. Plugs and sockets of the same colour can be connected by left-clicking the plug and releasing the mouse button over the socket. Apart from the construction of recipes, wxPyphant allows for the immediate inspection of intermediate results by right-clicking the appropriate plugs. Upon a right-click, a context menu is shown that offers all suitable visualisations. Finally, the GUI features the saving and loading of recipes to and from hdf5 files.

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.

5.2 Scripting with Pyphant Recipes

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

Concerning the recipe discussed in Sec 3, the following Python script demonstrates the simplicity of re-using a Pyphant recipe with modified parameters. The only modules to be imported are: the PyTablesPersister (Lst 8, line 1) for loading the recipe; and ImageVisualiser (Lst 8, line 2) for saving the result of the analysis as a Portable Network Graphics (PNG) image. Once the recipe has been loaded, the individual workers can be accessed by the `getWorkers` method of the recipe (Lst 8, line 8). A Worker can be configured by setting the `value` attribute of the respective parameters (Lst 8, line 10). In order to obtain the result from a specific worker, the `getResult` method of the respective plug has to be called (Lst 8, line 14). The resulting DataContainer can be used to initialise a suitable visualiser instance (Lst 8, line 17) whose diagram can be exported to a suitable graphic file format (Lst 8, line 18).

```
1 import pyphant.core.PyTablesPersister
2 from pyphant.visualizers.ImageVisualizer import ImageVisualizer
3
4 #Load recipe from hdffile
5 recipe = pyphant.core.PyTablesPersister.loadRecipeFromHDF5File('demo. h5')
```

31 The PEAK Developers' Center: PythonEggs, http://peak.telecommunity.com/DevCenter/PythonEggs, 2007

```
 6
 7 #Configure ImageLoaderWorker
 8 inputWorker = recipe.getWorkers('Image␣Loader')[0]
 9 imageName = 'demo.png'
10 inputWorker.getParam('filename').value=imageName
11
12 #Fetch Result
13 worker = recipe.getWorkers('Apply_Mask')[0]
14 result = worker.plugCreateMaskedImage.getResult()
15
16 #Visualise Result
17 visualizer = ImageVisualizer(result)
18 visualizer.figure.savefig('result−'+imageName)
```

Listing 8: Simple Python script interfacing the Pyphant recipe depicted in Fig. 3.

6 Summary and Outlook

Pyphant is a flexible Python 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 scriptability allows for the application of carefully crafted recipes in a computing environment under the lack of graphical services or possibly the integration into completely different applications. The current stable version of Pyphant is 0.4-alpha4 which is the basis of this paper. The framework and the worker toolboxes are published under the BSD license on Sourceforge[32] and in the Cheese Shop[33].

Concerning the application of the Pyphant framework, we are planning to extend the `ImageProcessing` toolbox with more tools and provide a toolbox for solving ill-posed problems on the basis of non-linear regularisation methods[34]. Possibly the family of toolboxes can be extended even further to entirely different projects in need of a similar GUI. To circumvent the restrictions imposed by the global interpreter lock and to harness the full potential of a parallel processing environment, we plan to refactor Pyphant into a compute server architecture, which is already hinted for by the present architecture. Furthermore, we are going to incorporate the emd5 identifier of the DataContainer into a *processing history* such that the origin of each result can be backtracked to the actual realisation of the underlying algorithm and the processed original data.

Acknowledgement

The authors would like to thank the editors for the opportunity to publish an updated version of our paper presented at the Europython 2006 conference[35] in *The Python Papers* and would like to encourage everybody who finds the presented framework interesting to participate in the project. The authors would also like to thank Michael C. Röttger for creating Pyphant's logo, Nico Bruns and Josef Honerkamp for fruitful discussions on the topic. The financial support by the German BMBF (Project No.: 03C0354A) is gratefully acknowledged. Finally, Andreas W. Liehr likes to thank Yanara M. L. Kempa for napping in a baby carrier at his chest while the introduction was written.

32. Klaus Zimmermann and Andreas W. Liehr. http://sourceforge.net/projects/pyphant/, 2007.
33. Python Software Foundation. *Python cheese shop*. http://cheeseshop.python.org/pypi/, 1990-2007.
34 cf. [7]
35. Klaus Zimmermann, Lorenz Quack, and Andreas W. Liehr. *Pyphant – a python framework for modelling reusable data processing tasks*. Refereed Paper Track, CERN 2006, http://tinyurl.com/r4rdz, 2006.