PALSE: Python Analysis of Large Scale (Computer) Experiments

Frédéric Cazals, Tom Dreyfus, Noël Malod-Dognin, Alix Lhéritier

To cite this version:

Frédéric Cazals, Tom Dreyfus, Noël Malod-Dognin, Alix Lhéritier. PALSE: Python Analysis of Large Scale (Computer) Experiments. [Research Report] RR-8165, INRIA. 2012, pp.16. hal-00759589

PALSE: Python Analysis of Large Scale (Computer) Experiments

F. Cazals and T. Dreyfus and A. Lhéritier and N. Malod-Dognin

RESEARCH REPORT
N° 8165
December 2012
Project-Team ABS

ISSN 0249-6399   ISRN INRIA/RR--8165--FR+ENG

PALSE: Python Analysis of Large Scale (Computer) Experiments

F. Cazals and T. Dreyfus and A. Lhéritier and N. Malod-Dognin

Project-Team ABS

Research Report n° 8165 — December 2012 — 13 pages

Abstract: A tenet of Science is the ability to reproduce the results, and a related issue is the possibility to archive and interpret the raw results of (computer) experiments. This paper presents an elementary python framework addressing this latter goal.

Consider a computing pipeline consisting of raw data generation, raw data parsing, and data analysis i.e. graphical and statistical analysis. PALSE addresses these last two steps by leveraging the hierarchical structure of XML documents.

More precisely, assume that the raw results of a program are stored in XML format, possibly generated by the serialization mechanism of the boost C++ libraries. For raw data parsing, PALSE imports the raw data as XML documents, and exploits the tree structure of the XML together with the XML Path Language to access and select specific values. For graphical and statistical analysis, PALSE gives direct access to ScientificPython, R, and gnuplot.

In a nutshell, PALSE combines standards languages (python, XML, XML Path Language) and tools (Boost serialization, ScientificPython, R, gnuplot) in such a way that once the raw data have been generated, graphical plots and statistical analysis just require a handful of lines of python code.

The framework applies to virtually any type of data, and may find a broad class of applications.

Key-words: Experiments, data analysis, statistics, python, scripting

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

PALSE: Analyse Python de résultats d'expériences à grande échelle

Résumé : Le caractère scientifique d'une étude repose sur sa reproductibilité, un problème lié étant la possibilité d'archiver et d'interpréter aisément les résultats de l'étude. Ce travail présente un canevas élémentaire traitant ces deux derniers objectifs.

Considérons une chaîne de traitement, faisant intervenir la génération de données, leur analyse syntaxique (*parsing*), et leur interprétation graphique et statistique. PALSE gère ces deux derniers aspects, en tirant profit de la structure hiérarchique de documents XML.

Plus précisément, considérons le cas de résultats générés au format XML par un programme informatique, par exemple par les mécanismes de *serialization* des librairies C++ boost. Pour l'analyse syntaxique, PALSE importe les données brutes comme documents XML, et exploite la structure arborescente d'XML combinée aux langage XML Path pour accéder à des informations spécifiques. Pour les manipulations graphiques et statistiques, PALSE fait un appel direct à ScientificPython, R, et gnuplot.

En résumé, PALSE combine des langages standards (python, XML, XML Path Language) et des outils standards (Boost serialization, ScientificPython, R, gnuplot), de telle sorte que lorsque les résultats ont été générés, leur analyse graphique et statistique ne nécessite que quelques lignes de code python.

Ce canevas s'applique à virtuellement tout type de données, de telle sorte que PALSE devrait s'avérer utile pour un large spectre d'applications.

Mots-clés : Expériences, analyse de données, statistiques, python, scripting

1 On Data Generation and Analysis in Computational Science

In computational Science, complex phenomena are modeled and simulated by involved algorithms and programs, which themselves often rely on various parameters. Such a complex investigation strategy raises three core problems. The first one is the correctness of software, namely its ability to meet specifications. This is clearly a non trivial issue, since a number of constraints may be imposed, such as logical constraints, the numerical constraints (numerical correctness is non trivial, since rounding floating point representations according to the IEEE 754 standard may result in erroneous floating point values reported), or performance constraints (requirements on the running time may be mandatory, e.g. for real time applications), etc. The second one relates to the ability to re-generate raw data, a non trivial issue given in particular the numerous parameters that may be involved, and also to the possible random nature of the phenomena investigated. This particular motivated the recent *Executable Paper Grand Challenge*, proposed by Elsevier. [1] The third one, which is our focus in this note, is concerned with the archival and the analysis of raw data. Handling raw data indeed raises several issues:

- *Raw data perenniality.* Make the raw data perennial by ensuring that anyone can make sense out of them even once decoupled from the program that generated them. This is non trivial since the executable and scripts / other programs parsing the output need to co-evolve.
- *Raw data availability.* Make the perennial raw data accessible, to allow novel analysis by scientists equipped with different methodological tools.
- *Raw data parsing and analysis.* Ease the parsing of the raw data, so as to prepare the graphical and statistical analysis.

2 The PALSE Design

PALSE is a lightweight python framework addressing the issues just discussed. Its design, which we discuss now, is summarized on Fig. 1.

2.1 Preamble: on Data Formats

PALSE assumes that a given (computer) experiments yields one file storing the results. To make these data perennial and foster their availability, PALSE uses XML, since this language is a standard one, and most importantly, provides an abstract structure amenable to high-level querying and filtering operations.

2.2 Raw Data Generation

By raw data we refer to the results of some (computer) experiment—PALSE can be used to handle data generated by any device, or even manually archived. Archives and their (de-)construction is a tenet of PALSE, and following the Boost library [2] serialization engines. Following the documentation [3], *we use the term "serialization" to mean the reversible deconstruction of an arbitrary*

[1] See `http://www.executablepapers.com/`

[2] Boost is a set of peer-reviewed C++ libraries, see `http://www.boost.org/`

[3] `http://www.boost.org/doc/libs/1_49_0/libs/serialization/doc/index.html`

set of C++ data structures to a sequence of bytes. Such a system can be used to reconstitute an equivalent structure in another program context. That is, an *archive* refers to a specific rendering of the aforementioned sequence of byte, and PALSE takes as input XML archives. For computer experiments, the generation of XML data naturally depends on the programming language used, two of them being of particular interest: C++ and python.

In C++, the boost serialization tools accommodate the native C++ types and the data structures of the Standard Template Library Fig. 1). In python, the recursive structure of dictionaries makes conversion of dictionary into an XML tree (and vice-versa) a trivial task.

2.3 Raw Data Parsing and Database Creation

Consider a set of XML archives, one per (computer) experiment. PALSE imports each such file as an XML trees. A *database* corresponds to a set of *isomorphic* trees corresponding to experiments generated with varying parameters. Therefore, several databases may be used to accommodate several sets of parameters, or to store results from different sources. The creation of the trees from the files is delegated to the lxml[4] library.

2.4 Raw Data Querying

Given the XML trees stored in database(s), the retrieval of the values of interest for graphical and statistical analysis combines features of the XML and XPath languages.

The XML hierarchical representation. XML provides a hierarchical representation of a document, which may be seen as a rooted ordered tree of nodes called *Elements* (Fig. 2, and the supplemental Fig. 2). Each Element is characterized by a tag-name, and possibly contains attributes, a text field and a number of child Elements — there is also a optional tail string which is not supported by PALSE):

- The tag-name of an Element e corresponds to the name of the start-tag $(< tag - name >)$ and the end-tag $(< /tag - name >)$ of e in its XML representation.

- An attribute of an Element e is a variable having a particular value defined in its XML representation with the start-tag of e $(< tag - name\ att =' value' >)$.

- The text field of an Element e is defined as any text in-between the start-tag and the end-tag of e in its XML representation, which is not embedded in-between another (start-tag, end-tag) pair $(< tag - name > text - field < /tag - name >)$.

- All the children of e are the Elements which are defined in its XML representation in-between the start-tag and the end-tag of e $(< tag - name >< child - tag - name >< /child - tag - name >< /tag - name >)$.

In the sequel, by *data value*, we refer either to the value of an attribute or of a text field.

An Element may be also embedded in a namespace, that is represented by a pair *(prefix, uri)*. The uri is a string of characters identifying a name or a resource, and the prefix is a simple name associated to the uri. Thus, if an Element e having a tag-name *tag* is embedded in a namespace *(prefix, uri)*, its tag-name is represented in the XML file by *prefix:tag*. However, in the XML tree, all prefixes are replaced by their corresponding uri, so that the tag name of the Element e is represented in the XML tree by *{uri}tag*. In the sequel, the term *name-value-pair* refers to the (tag-name, text-field) of a specific Element.

[4] lxml is an easy-to-use library for handling XML and HTML in Python, see http://lxml.de/

The XPath query language. Given an XML document, the XPath language has been developed to address specific parts of this document, see `http://www.w3.org/TR/xpath/`. We note that an XPath query always specify a path to one or more Element(s). An XPath may be specified in three different ways [5]:

- (i) a sequence of tag-names only (e.g tag_1/tag_2), denominating all the Elements having as tag tag_2 and that are children of Elements with tag tag_1.
- (ii) or a sequence of tag-names, each possibly enriched by an attribute name (e.g $tag\_1/tag\_2[@att]$, or $tag\_1[@att]/tag\_2$), denominating the subset of the Elements previously described, but having the attribute *att*.
- (iii) or a sequence of tag-names, each possibly enriched by an attribute name and a value of this attribute (e.g $tag\_1/tag\_2[@att =' val']$, or $tag\_1[@att =' val']/tag\_2$), denominating the subset of the Elements previously described, but the attribute *att* having as value *val*.

Functions offered in `PALSE`. The three XPath modes just discussed are used by `PALSE` functions to query the database, in order to retrieve selected Elements or their data values (either from text fields or attributes). Before describing these functions, we note that each of them operates on the whole database, so that each function returns a list of lists (one per XML tree).

▷ Function `get_all_elements_from_database(xpath_query)`: targets all the matching Elements.
▷ Function `get_all_data_values_from_database(xpath_query)`: targets the list of data values of elements:

- if the XPath query is of type (i), the data value is the value of the text field;
- if the XPath query is of type (ii) or (iii), and is ended by an attribute (whose value is given or not), the data value is the value of the attribute *att*.

To target the first Element (or data value) only in an XML tree that matches the specified XPath:
▷ Function `get_leftmost_elements_from_database(xpath_query)`
▷ Function `get_leftmost_data_values_from_database(xpath_query)`

Finally, given a list of Elements L, a XPath query X, a data value v and a comparator *comp*, it is possible to select Elements e of L such that the specified Xpath X from e has a data value v_e positively compared with v using *comp*, a functionality also provided for basic comparators (e.g. on integers):
▷ Function `filter_elements_by_data_values_compare_to(`L`, `X`, `v`, `*comp*`)`
▷ Function comparators of integers
`filter_elements_by_data_values_lower_than_integer(`L`, `X`, `v`)`

We also note that upon calling a function returning Elements, the selection can be converted into strings thanks to the function `get_data_values_from_elements(`L`, `X`)`, with L and X the arguments described above.

[5] `http://docs.python.org/2/library/xml.etree.elementtree.html`

2.5 Data Manipulation

The data collected by the previous step typically need to undergo processing before being amenable to analysis. Since the previous step supplies python lists of strings, `PALSE` provides mechanisms to select, sort, and convert these lists into lists of elementary types. Furthermore, `PALSE` provides tools for combining lists into dictionaries, for further filtering using regular expressions (for strings) or lists of allowed keys or values.

2.6 Data Analysis

The lists just produced are ready for graphical and statistical analysis. `PALSE` encapsulates functionalities from SciPython and R for computing Pearson, Spearman or Kendall's tau correlations between the collected values, as well as Mann-Whitney rank-sum test. More generally, any package interface with python can be used, e.g. gmpy if unlimited-precision integers / rationals / floats must be used to ensure numerical correctness, gnuplot-py to generate eye-candy plots and charts, etc.

Figure 1 The `PALSE` workflow. The five steps: (1) the native data types of the application(s) are first serialized into XML archives, stored into XML files. It is assumed that one file is generated for each (computer) experiment (2) `PALSE` loads these XML files into databases of XML trees (3) Raw data are extracted from the databases using XPath queries (4) these data are prepared (sorted, filtered, ...) (5) the manipulated data are used to perform various data analyses (statistics, plots, . . .).

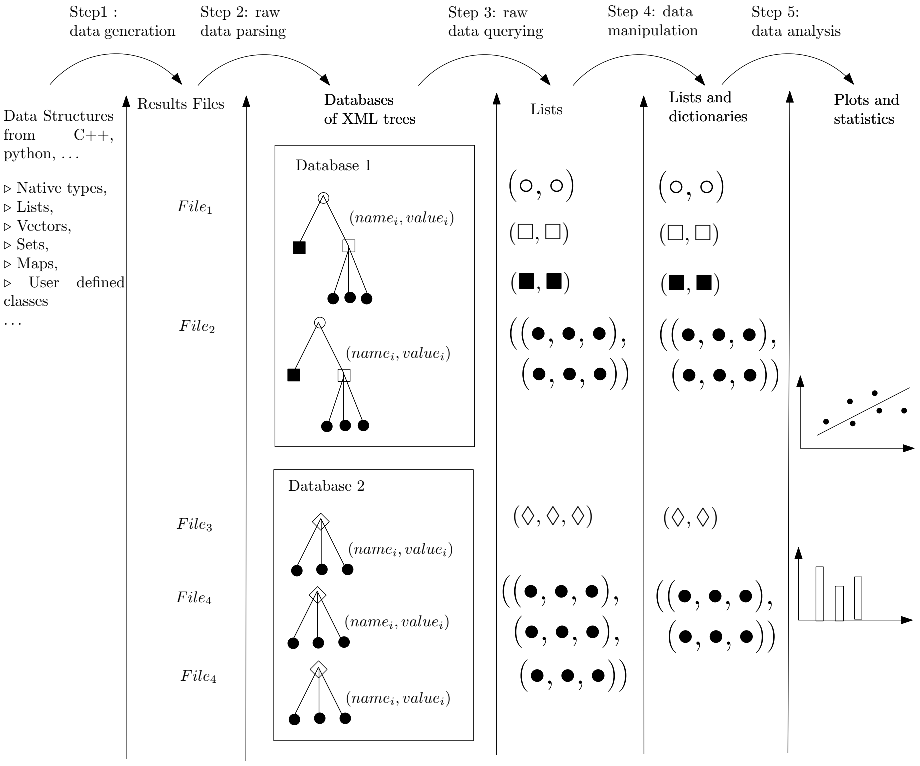
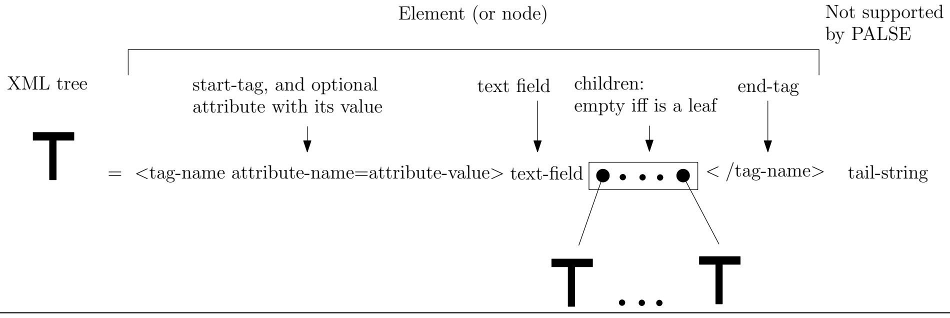


Figure 2 The XML trees handled by PALSE A tree T is recursively defined from Elements.


Element (or node)
Not supported by PALSE
XML tree
start-tag, and optional attribute with its value
text field
children: empty iff is a leaf
end-tag
T
= <tag-name attribute-name=attribute-value> text-field
< /tag-name>
tail-string
T ... T

3 Using PALSE

3.1 Using PALSE to Handle a Custom Computer Application

We illustrate the steps of the PALSE workflow (Fig. 1) on the problem of mining the relationship between the number of atoms of a monomeric or large oligomeric protein, and the number of exposed atoms [1, 2], using in-house computer application Intervor [3]. The details are provided in the supplemental section 4.1.

3.2 Using PALSE to Query Complex Experimental Data Files

As a second example, we exemplify the ability of PALSE to handle data files stemming from an involved experimental process, here files from the Protein Data Bank (http://www.rcsb.org), generated thanks to X ray crystallography or NMR experiments. Such files contain structural information of macro-molecules and macro-molecular complexes, and are notoriously challenging to handle due to the complexity of the information provided (with biological, bio-physical, genetic pieces of information inter-connected).

The code provided in the supplemental section 4.2 shows that PALSE allows dealing with PDB files elegantly. As an example, plotting the distribution of the number of atoms observed in all the (polypeptide) chains of a collection of PDB files merely requires 15 lines of code. The dataset used for this illustration consists of the 82 PDB files, all solved by X ray crystallography, contributed by J. Janin.

References

[1] C. Chothia et al. Structural invariants in protein folding. *Nature*, 254(5498):304–308, 1975.

[2] S. Miller, J. Janin, A.M. Lesk, and C. Chothia. Interior and surface of monomeric proteins. *Journal of molecular biology*, 196(3):641–656, 1987.

[3] S. Loriot and F. Cazals. Modeling macro-molecular interfaces with intervor. *Bioinformatics*, 26(7):964–965, 2010.

[4] L. Lo Conte, C. Chothia, and J. Janin. The atomic structure of protein-protein recognition sites. *Journal of Molecular Biology*, 285:2177–2198, 1999.

4 Supporting Information

4.1 Using PALSE to Handle a Custom Computer Application

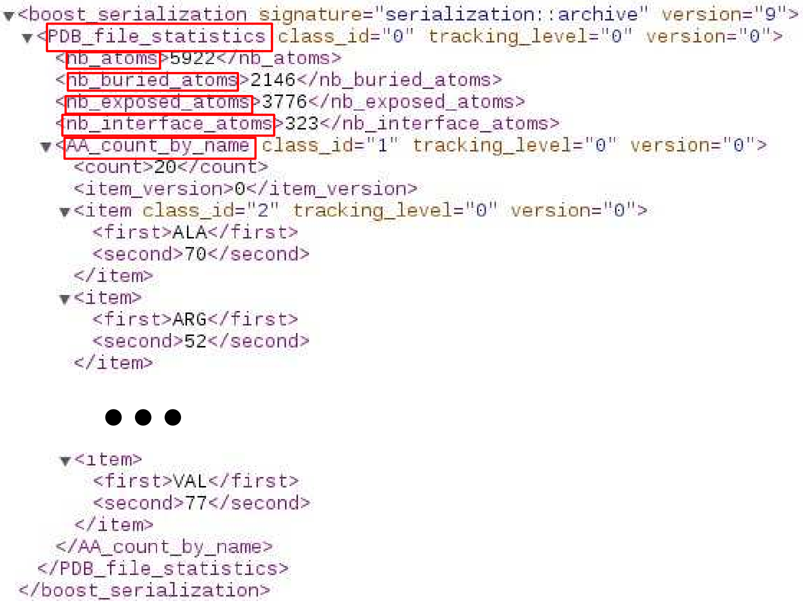
As a simple illustration, we investigate a simple property of macro-molecular complexes, processing the protein complexes from [4], using the `Intervor` software [3].

More precisely, consider a protein complex involving two partners, say A and B. The surface of a partner consists of the atoms which are exposed (or equivalently not buried) when the partner is alone. These atoms are those which may interact with the other partner. Using `Intervor` [3], we partition the atoms of a complex as exposed, buried, or interface atoms. The corresponding C++ code and resulting archive are sketched on Figs. 1 and 2. Finally, `PALSE` is used to plot the two statistics of interest (Fig. 4).

Supplemental Figure 1 Palse, step 1: generation of an XML archive. In C++, the creation of the archive consists of calling the boost serialization algorithms for native C++ types and for data structures from the Standard Template Library. Note that each call to the function `make_nvp` creates a so-called name-value pair.

```
template<class archive>
void serialize(archive& ar, const unsigned int version)
{
    using boost::serialization::make_nvp;

    // serialization for a C++ native type
    ar & make_nvp("nb_atoms", nb_atoms);
    ar & make_nvp("nb_buried_atoms", nb_buried_atoms);
    ar & make_nvp("nb_exposed_atoms", nb_exposed_atoms);
    ar & make_nvp("nb_interface_atoms", nb_interface_atoms);

    // serialization for a std::vector, which counts the number
    // of occurrences of the 20 native amino-acid types
    ar & make_nvp("AA_count_by_name", residue_count_by_name);
}
```

Supplemental Figure 2 Example XML archive generated by the code of Fig. 1. (Protein complex: PDB code 1aip.pdb). This archive is visualized using google-chrome, which allows inspecting the hierarchical structure. The text circledq in red corresponds to the names of the so-called *name-value-pairs*; concatenating these strings yields the path used to retrieve values in the tree representation of an XML archive.

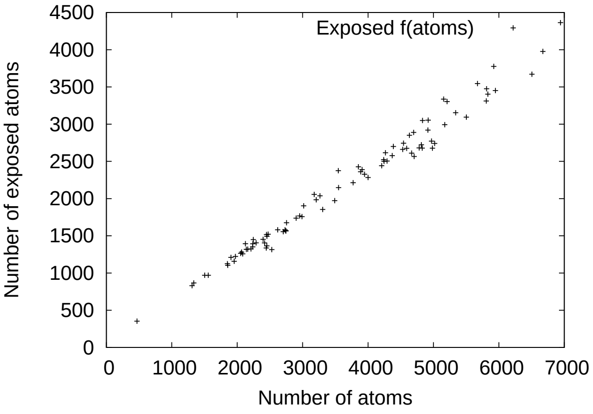


```
▼<boost_serialization signature="serialization::archive" version="9">
  ▼<PDB_file_statistics class_id="0" tracking_level="0" version="0">
     <nb_atoms>5922</nb_atoms>
     <nb_buried_atoms>2146</nb_buried_atoms>
     <nb_exposed_atoms>3776</nb_exposed_atoms>
     <nb_interface_atoms>323</nb_interface_atoms>
   ▼<AA_count_by_name class_id="1" tracking_level="0" version="0">
       <count>20</count>
       <item_version>0</item_version>
     ▼<item class_id="2" tracking_level="0" version="0">
         <first>ALA</first>
         <second>70</second>
       </item>
     ▼<item>
         <first>ARG</first>
         <second>52</second>
       </item>

          ● ● ●

     ▼<item>
         <first>VAL</first>
         <second>77</second>
       </item>
     </AA_count_by_name>
   </PDB_file_statistics>
 </boost_serialization>
```

Supplemental Figure 3 Palse, step 2 to 5. Note in particular that all the values associated with a specific path in an XML tree are retrieved using the concatenation of these paths, e.g. *PDB_file_statistics/nb_atoms* to collect the size (in atoms) of all the molecular systems processed.

```
# Step 2: Create the database of XML trees
res_dir = "%s/mols_archives/LoConte_Janin_JMB99/results/palse-example" % getenv_or_die("HOME")
Lo_Conte_etal_analysis = PALSE_xml_handle(res_dir)
Lo_Conte_etal_analysis.build_Document_Object_Models()

# Step 3: retrieve list of data from the concatenated tags of the nodes of the XML trees
# Step 4: void, since we do not perform any filtering

list_tool = PALSE_data_handle()

atoms_str = Lo_Conte_etal_analysis.get_leftmost_data_values_from_database("PDB_file_statistics/nb_atoms")
all_atoms = list_tool.convert_txtlist_to_intlist( atoms_str )

exposed_atoms_str = Lo_Conte_etal_analysis.get_leftmost_data_values_from_database("PDB_file_statistics/nb_exposed_atoms")
exposed_atoms  = list_tool.convert_txtlist_to_intlist( exposed_atoms_str )

# Step 5: perform the plots, compute the statistics

g_atoms = Gnuplot.Gnuplot()

g_atoms(’set xlabel \"Number of atoms in the complex\" ’)
g_atoms(’set ylabel \"Number of exposed atoms\" ’)
data_at_exp_at = Gnuplot.Data(all_atoms, exposed_atoms, title=’Exposed f(atoms)’,with_=’points lw 2’)
g_atoms.plot(data_at_exp_at)
g_atoms.hardcopy(’all_atoms_versus_exposed_atoms.eps’,enhanced=1,color=0,fontsize=24)

# and Pearson correlation coefficient
correlate = PALSE_statistic_handle()
correlate.pearson(all_atoms, exposed_atoms, ’atoms’, ’exposed’)
```

Supplemental Figure 4 Plot produced by the code of Fig. 3 A linear (Pearson) correlation coefficient of 0.99 is also obtained, an observation known since [1, 2].



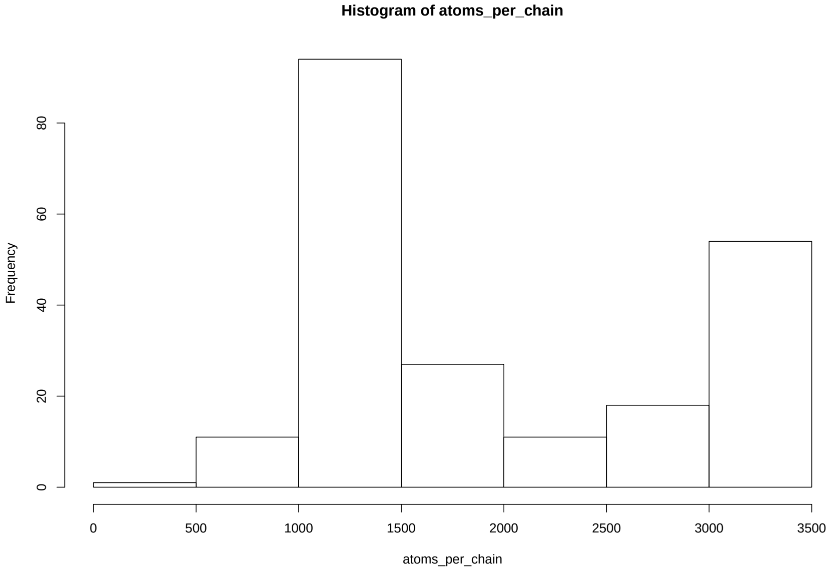
4.2 Using PALSE to Query Files from the Protein Data Bank

This second example illustrates the power of PALSE to handle complex data, to accommodate namespaces, and to perform various filtering steps. As an illustration, we plot an histrogram corresponding to the distribution of the number of atoms found in polypeptide chains of files from the PDB. As seen from Fig. 5, the corresponding python code is minimalist. An example histogram produced from this code is shown on Fig. 6.

Supplemental Figure 5 Example: atoms per chains of PDB entries. Step 1. The XML databases of input PDB entries are created using the namespaces required by the PDBML file format (XML format for the PDB). **Step 2**. Since there are possibly multiple PDB entries, chains and atoms are lists of lists: chains is a list of lists of values of an attribute representing the *id* of a chain, and atoms is a list of lists of Elements representing an *atom_site*. **Step 3**. For each PDB entry, we filter the atoms (we discard heteratoms), and then for each chain of this entry, we count the number of atoms belonging to this chain. **Step 4**. We plot in a 2D histogram the registered number of atoms per chain using R. Note that the output format is *eps*.

```
#! /usr/bin/python

from PALSE import *

# Step 1: Create the database of XML trees
pdb_handler = PALSE_xml_DB()
pdb_handler.add_namespace("PDBx", "http://pdbml.pdb.org/schema/pdbx-v40.xsd")
pdb_handler.load_from_directory("data-pdb-xml")

# Step 2: List the chains and atoms of all pdb entries
chains = pdb_handler.get_all_data_values_from_database("PDBx:struct_asymCategory/PDBx:struct_asym[@id]")
atoms = pdb_handler.get_all_elements_from_database("PDBx:atom_siteCategory/PDBx:atom_site")

# Step 3: Count the number of atoms per chain
nb_atoms_per_chain = []
for i in range(len(atoms)):
    atoms[i] = pdb_handler.filter_elements_by_data_values_equal_to_string(atoms[i], "PDBx:group_PDB", "ATOM")
    for chain in chains[i]:
        n = len(pdb_handler.filter_elements_by_data_values_equal_to_string(atoms[i], "PDBx:auth_asym_id", chain))
        if n > 0:
            nb_atoms_per_chain.append(n)

# Step 4: Plot a 2D histogram using R
PALSE_statistic_handle.Rhist2d(nb_atoms_per_chain, "hist-atoms-per-chain.eps")
```

Supplemental Figure 6 Plot produced by the code of Fig. 5, using 82 PDB entries.



4.3 Implementation Overview

In this section, we sketch the main classes of `PALSE`, and refer the interested user to the commented source code for more details.

Class `Python_dico_vs_XML_Etree`. A class performing manipulations between python dictionaries and XML Etrees. This class is meant to (de-)construct XML archives from python dictionaries. All its functions are static members.

Class `PALSE_xml_DB`. A `PALSE` database consists of a set isomorphic trees coming from XML archives, which can be queried using the functionality discussed in Sec.2.4.

Class `PALSE_DS_manipulator`. A class providing functions to manipulate python Data Structures in general, and lists of strings / native python data types in particular. All its functions are static members.

Class `PALSE_statistic_handle`. A class providing elementary statistics and plotting facilities. All its functions are static members, and resort to low level operations borrowed to gnuplot, R and scientific python.

Contents

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399