FP7- Grant Agreement no. 283393 – *RadioNet3*

Project name: Advanced Radio Astronomy in Europe

Funding scheme: Combination of CP & CSA

Start date: 01 January 2012 Duration: 48 month

Deliverable D10.13

Demonstrator pipelines (code in repository) for the selected applications.

Due date of deliverable: 2015-02-28

Actual submission date: 2015-06-02

Deliverable Leading Partner: JOINT INSTITUTE FOR V.L.B.I. IN EUROPE (J.I.V.E.),
THE NETHRELANDS

An European project supported within the 7th framework programme

1 Document information

Type	Demonstrator - Report
Title	Demonstrator pipelines (code in repository) for the selected applications.
WP	10 (Hilado)
Authors	Des Small (JIVE), Mark Kettenis (JIVE), Bojan Nicolic (UCAM)

1.1 Dissemination Level

Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

1.2 Content

2 Short description

A frozen copy of the code repository associated with this deliverable can be found on the RadioNet3 Wiki at

http://www.radionet-eu.org/radionet3wiki/doku.php?id=jra:hilado

And can be downloaded directly using the link:

http://www.radionet-eu.org/radionet3wiki/lib/exe/fetch.php?media=na:hilado-d10.13.tgz

This includes a simplified version of the EVN imaging pipeline in the Swift parallel scripting language. A description of the code, and the ideas behind it, is given in the attached attached document titled *Hilado Deliverable 10.13: A framework for minimal recomputation*.

Copyright

Hilado Deliverable 10.13: A framework for minimal recomputation

Des Small[1], Mark Kettenis[1], and Bojan Nikolic[2]

[1]JIVE
[2]University of Cambridge

February 27, 2015

1 Introduction

Work Package 4 of Hilado is concerned with "bringing it to the user". As described in the Activity Plan (de Vos, 2012), we have considered the role of domain specific languages and analysis tools in the context of minimizing the amount of redundant calculation performed while developing an analysis pipeline or during exploratory data analysis. This report will describe the work carried out so far in this work group and the anticipated directions of future work.

We begin with an explanation of the kind of analysis we intend to carry out and the kind of programs that are suitable for analyzing this way. The syntax of the programs we have to work with turns out not to be a good fit for our analysis, and we discuss three related ways of improving the fit: firstly, a "toy" language which makes the dataflow more explicit and is independent of any specific astronomical back-end; secondly, an approach based on execution graphs for the Casa package (*NRAO Casa Webpage*), described in detail in Section 5, and finally a version of the ParselTongue (Kettenis et al., 2006; *The ParselTongue Wiki* 2015) package that has been adapted to use the Swift programming language (Wilde et al., 2011).

The work described here was intended to develop a proof of concept, together with a test-bed implementation. Now that the concept has been proved and a test-bed implemented, we finish by describing how we see this work developing in the future. One important thing to note is that although this report is focused on astronomical applications and Sections 5 and 6 are devoted to Casa and ParselTongue specifically, the problem we address is of general interest to the field of scientific programming. Any environment where analysis pipelines are incrementally refined, data-sets are large, and computation is expensive and time-consuming could potentially benefit from the methods and ideas outlined here, and one important task we intend to undertake in the future is to publish a paper to bring this work and its potential benefits to the attention of a wider community.

2 Pipelines, graphs, and functional programs

We consider the case where an astronomer is developing a pipeline similar in spirit to the fragment shown in Algorithm 1.

Algorithm 1: Pipeline algorithm with side effects

```
fn ← "datafile";
data ← read_data(fn, 1);
munge_data(vis=data, opcode="CAL", p=0.7);
restrain_data(vis=data, threshold=0.4);
plots ← make_plots(data, b)
```

(The syntax is fictional, as are the functions invoked, but both are modeled on the style of Casa and ParselTongue.) The astronomer then decides to try the pipeline again, with a different parameter for the ‘threshold’ parameter in the ‘restrain_data’ procedure, as shown in Algorithm 2.

Algorithm 2: Revised pipeline algorithm with side effects

```
fn ← "datafile";
data ← read_data(fn, 1);
munge_data(vis=data, opcode="CAL", p=0.7);
restrain_data(vis=data, threshold=0.5);
plots ← make_plots(data, b)
```

All the steps before the call to ‘restrain_data’ are unchanged, and our goal is to develop an environment in which this is (automatically) recognized and old values are transparently cached and reused where appropriate. However, the above examples illustrate one of the pitfalls, shared by Casa and ParselTongue, that immediately confronts any attempt to do this: functions change the value of the arguments they are given.

As analysts of programs, we might hope for a notation in which inputs and outputs to functions are distinguished, and inputs are never changed. With a straightforward adaptation of the notation we could then rewrite the two programs to be compared as shown in Algorithms 3 and 4.

Algorithm 3: Original pipeline algorithm with return values

```
fn ← "datafile";
data ← read_data(fn, 1);
data ← munge_data(vis=data, opcode="CAL", p=0.7);
data ← restrain_data(vis=data, threshold=0.4);
plots ← make_plots(data, b)
```

Algorithm 4: Modified pipeline algorithm with return values

```
data $\leftarrow$ read_data(fn, 1);
data $\leftarrow$ munge_data(vis=data, opcode="CAL", p=0.7);
data $\leftarrow$ restrain_data(vis=data, threshold=0.5);
plots $\leftarrow$ make_plots(data, b)
```

We should note that Casa and ParselTongue have the conventions they have for excellent reasons: astronomical data-sets can be large, and it is much cheaper and simpler to modify them in place than it is to make a new copy with each operation applied. That is unless there is a way to make copies cheap: Section 5 will describe exactly this, by using a version of Casa on top of the ZFS file system to copy only the parts of the data set that are changed. This still leaves a trade-off to be made between the cost of recomputing data and the cost of storing it. In some case it may be more economical to recompute some intermediates; we can envisage that a cost-function could be defined which allows this trade-off to be specified.

Note that Algorithms 3 and 4 still describe *imperative* programs. The value of the variable ‘data’ still changes as we go through the program, so that we cannot identify the variable name with any fixed data set. In many so-called functional programming languages it is not possible to reassign variables, following a precedent set by the Id programming language Arvind et al., 1978; Algorithm 5 shows how Algorithm 3 might be written in this style.

Algorithm 5: Original pipeline algorithm in single-assignment form

```
fn $\leftarrow$ "datafile";
$data_0$ $\leftarrow$ read_data(fn, 1);
$data_1$ $\leftarrow$ munge_data(vis=$data_0$, opcode="CAL", p=0.7);
$data_2$ $\leftarrow$ restrain_data(vis=$data_1$, threshold=0.4);
plots $\leftarrow$ make_plots($data_2$, b)
```

This style has advantages. Every variable has a single unambiguous definiton, and so it is possible to reorder the evaluations so that computations can be performed, potentially in parallel, as soon as their inputs are available. The Swift parallel scripting language Wilde et al., 2011 is based on precisely this insight, and Section 6 will describe a version of ParselTongue built to use this language.

However, developing and altering a script written in the style of Algorithm 5 is not an inviting prospect. In practice programmers in functional languages have a variety of idioms to draw on to avoid writing programs in quite this style. The work described in Sections 3 and 4 was implemented in the purely functional language Haskell (Peyton Jones et al., 2003), and it proved to be a generally pleasant experience. But since we do not propose that astronomers should be expected to master functional programming, we assume instead a middle ground, where inputs and outputs are clearly

$$\begin{tikzcd}
\text{Syntax}_1 \arrow[r, leftrightarrow, "\text{Tree diff}"] \arrow[d, "\text{dataflow}"'] & \text{Syntax}_2 \arrow[d, "\text{dataflow}"] \\
\text{Use-def}_1 \arrow[r, Leftrightarrow, "\text{inferred}"] & \text{Use-def}_2
\end{tikzcd}$$

Figure 1: Inferring relations between variable uses from the syntax tree

identified in the syntax, but reassignment of variables is permitted.

In the following sections (Sections 3 and 4) we describe some tools we have developed to analyze a language in that spirit, drawing particularly on the literature around compiler development (see, for example, Appel, 1997; Appel, 1998a). The language itself is a toy language invented for this work, since have to supply a full parser and syntax tree tools for any language we process, and since we are only really concerned with basic conditionals, loops, assignments and function calls it is simpler to define a language that only supports these. Despite its limitations this language is adequate to describe typical pipeline scripts, and after processing to identify reusable data products the resulting abridged script can be exported to either of the back-ends we consider later.

3 Syntax-based approaches

Following pioneering work by Chawathe et al. (1996), a series of papers has sought to track tree and graph based differences in database schemas and ontologies (Eder and Wiggisser, 2006; Eder and Wiggisser, 2007) as well as the text of computer programs (Fluri et al., 2007).

However, differences in the text between two programs is not sufficient to know which values are the same in both and what needs to be recomputed. To calculate this, we need to also track the uses of each variable definition, bearing in mind that variables (in non-functional languages) can be redefined. The technique of computing *use-def chains* is well-known in the compiler literature (Aho et al., 1986; Harrold et al., 1993) and addresses precisely this problem.

To build a recalculation calculator based on this approach, we take two programs, which we assume to be variants of each other, we compute the syntax tree for each. We then calculate the difference between the syntax trees, using the method of Fluri et al. (2007). For each syntax tree we use standard dataflow algorithms to calculate the usage of each definition, and then we can combine the tree differences with this to calculate an inferred change in the resulting dataflow, as depicted graphically in Figure 1.

A program was successfully developed that performed this analysis on programs in our toy language, but this approach was ultimately rejected in favour of the graph partition idea described in the next section.

4 Graph-partitioning approaches

As remarked above, the ability to reassign variables in a conventional 'imperative' programming language complicates the analysis of programs written in them – the dataflow analysis performed in Section 3 above is largely concerned with disambiguating variable assignments.

By contrast, programs written in functional programming languages, where each variable can only be assigned to once, are considerably more tractable to analysis. The disadvantage of functional programming languages is that they require programmers to get used to a quite different way of organizing their code, and we do not want to put any unnecessary stumbling blocks in the way of programmers that could discourage take up of our tools.

Fortunately, we can achieve many of the benefits of functional programming in an imperative setting by using an intermediate language (internal to the analysis stage and not exposed to the user) with single static assignment (SSA). In SSA form, as the name suggests, each variable is assigned to once in the program text; each variable (say, d) in the original program is replaced by distinct variables (d_1, d_2,. . . , d_n) where necessary to preserve this single-assignment property. An important point to note is that when the program is in SSA form, we can think of it as a graph with definitions as the nodes, and function calls arguments, if they are variables, as edges to linking to the nodes where they are defined. Once this graph has been constructed, the variable names become annotations – all the information in the program is contained in the graph.

Following Cytron et al. (1991)'s work on an efficient algorithm to compute SSA form for arbitrary programs it became a popular form of analysis in compiler development generally (Appel, 1998a; Appel, 1998b). For our purposes the key paper is Alpern et al. (1988) (which actually predates Cytron et al.'s work) that shows how to use SSA form of a program to infer equality of variables within a program. Their method involves constructing a further graph, the 'value graph', from the SSA form of the program. It is then straightforward to partition the value graph to identify equivalent nodes, which correspond to equivalent variables.

Calculating equivalent variables within a single program is valuable for a conventional compiler. What we want, however, is to calculate equality of variables across *different* programs, but this turns out to be a fairly straightforward extension of Alpern et al.'s idea: we can simply develop a global value graph for all the programs being considered (keeping track of which nodes correspond to which programs) and apply the same analysis, as summarized in Algorithm 6.

Algorithm 6: Identifying equal variables

```
value_graph = $\emptyset$;
for $f \in$ program_files do
    ast $\leftarrow$ generate_abstract_syntax(f);
    cfg $\leftarrow$ generate_control_flow_graph(ast);
    domF $\leftarrow$ calculate_dominance_frontiers(cfg);
    ssa $\leftarrow$ calculate_SSA_form(cfg, domF);
    valG $\leftarrow$ calculate_value_graph(valG);
    value_graph $\leftarrow$ value_graph $\cup$ valG;
end
partition global value graph;
filter out non-variables from partitions
```

This global analysis has been successfully implement with the toy language, and the (Haskell) code is in the repository that accompanies this document. Having an algorithm is an important part of our project, but it also needs to be applied in the context of real astronomical software, and the following two sections discuss that.

5 Combining with Casa

As we remarked in Section 2, the first thing we need to begin analyzing programs is to be clear about the inputs and outputs to functions. As we also noted, this is often not encoded in the syntax of scientific programming tools, and Casa (*NRAO Casa Webpage*) is no exception. Casa also suffers from another common phenomenon which is problematic for our purposes, in that its datasets are stored as 'measurement sets', which are effectively large binary objects that are opaque at the operating system level. (Technically a measurement set is a directory in the Unix environment, but its contents are opaque binary formatted files.)

Internally the measurement set is made up of a number of tables, and it is often the case that a procedure will modify one table, or a few tables, while leaving the others constant. This means that changes to the measurement set are localized on disk. This insight means that while it would be impractical to copy the whole measurement set for every operation, it is possible to make copies if the commonalities between the copies can be identified and the commonality exploited.

Exactly this is possible with the Zettabyte File System (ZFS), originally developed for Solaris but now well-supported under Linux (Bonwick and Moore, 2008), which supports a (block-based) copy-on-write mechanism for this purpose. With this in hand, we can consider writing scripts in which the role of the 'data' variable in the scripts of Section 2 is played by the

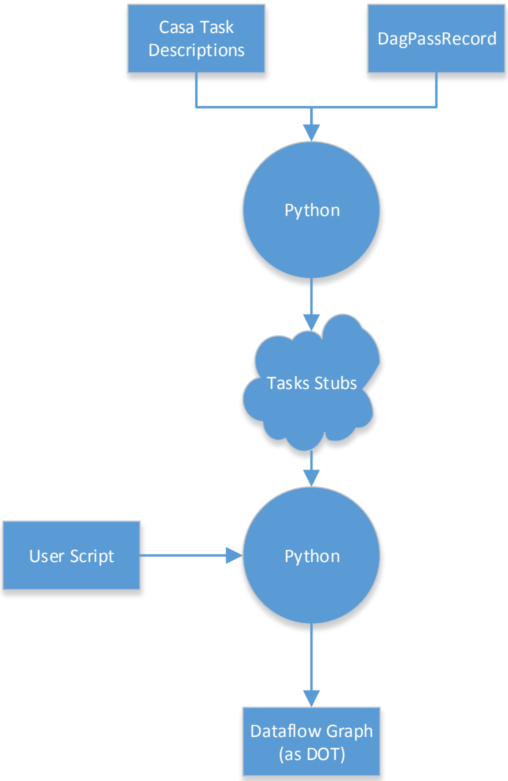

Figure 2: Turning Casa scripts into execution graphs.

measurement set itself. Doing so also means that the problem of storing the various intermediate values of the variable is trivially solved.

That leaves the problem that the base language for Casa is Python (van Rossum, 1995), a fully-fledged programming language, and that the procedures in Casa do not distinguish inputs and outputs in their syntax. An example of a Python Casa script is shown in Appendix A.

This problem has been addressed by the ‘recipe’ program included in the repository accompanying this program. It uses Python’s reflectional abilities, and some additional information about the nature of the parameters to specific procedures, to intercept a program and instead of simply running it, to generate a SSA-style graph of how its execution, including renaming all the intermediate data products with unique names, as shown in Figure 2. These graphs are exported in the ‘dot’ file format of the GraphViz package (Gansner and North, 2000) for further analysis. The latest release includes a mechanism to execute such a graph, to look for commonalities in scripts by searching graphs for isomorphisms, as shown in Figure 3 or to export the

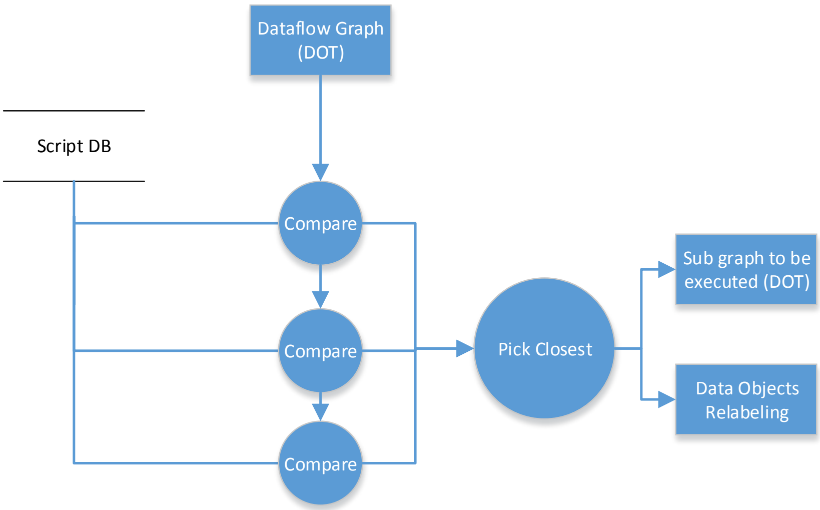

Figure 3: Comparing execution graphs from Casa scripts.

graph as a script. The expectation is that this code will be integrated with the Haskell code implementing the graph partition algorithm of Section 4 before the end of the Hilado project, at which point we will have a tool that is suitable for scientists to try out. The ultimate goal, as summarized in Figure 4, is for this process to run transparently, with the user submitting a script and getting a set of data products as output.

6 Combining with ParselTongue

ParselTongue (Kettenis et al., 2006) is a Python interface to the Astronomical Image Processing System (AIPS) (Wells, 1985). It also keeps data in tables embedded in files that are outwardly monolithic and opaque, but unlike Casa it has an internal versioning scheme, whereby each table has a current version number and old versions of tables are still accessible if explicitly requested. This means that we do not have to provide a mechanism to store reusable data products, but it also means we have to be able to do the bookkeeping to identify them.

The approach we have used here is to extend the Swift scripting language (Wilde et al., 2011) to support ParselTongue procedures (which are in turn wrappers around AIPS's own concept of 'tasks'). Swift is a single-assignment language, so using it on top of ParselTongue is formally equivalent to the execution graph in the previous section and the SSA-graph representation of the program in Section 4. A working example of a pipeline

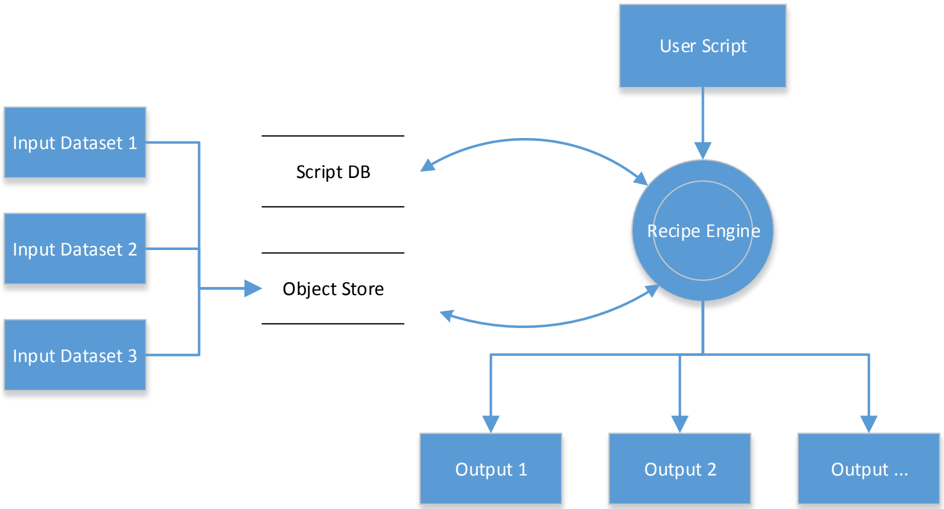

Figure 4: Transparent execution of Casa scripts with object cache.

script written using the Swiftified version of ParselTongue is included in the repository.

To combine this approach with the analysis described in Section 4 we would either need to adjust the tools to use the Swift language, or do the analysis on our toy language (possibly somewhat extended) and export the resulting graph as a piece of Swift syntax. While a Swift parse in Haskell has already been developed as part of this project, the second option is likely to be a quicker route to a fully functional prototype for scientific users.

7 Future directions

The approaches we have described in this report are all unified around the concept of the SSA form of programs and the corresponding execution graph. We have running code for a Swiftified ParselTongue, as well as for an execution engine (and commonality detector) for Casa and a complete equivalence detection engine for our toy language in Haskell.

What we need, and what we propose to implement in the immediate future, is to link the two execution frameworks to the Haskell analysis to have a complete system which can be made available to end-users for feedback and testing which would then inform further development.

Beyond that, we intend to write up the ideas sketched here in a more systematic form for publication in a refereed journal – the problem we have been addressing is one that is common to many fields of scientific computing, it is becoming more rather than less urgent as datasets continue to grow, and our techniques, if not necessarily our code, could be applied quite generally.

Appendices

A Casa code

Below we show a listing of an example Casa pipeline, that can be processed using the tools described in Section 5.

Listing 1: Example Casa pipeline language

```
# This is the casa tutorial at:
# http://casaguides.nrao.edu/index.php?\
# title=Calibrating_a_VLA_5_GHz_continuum_survey
import os

xfl = [ "AG733_A061209.xp1",
        "AG733_B061213.xp1"]

ampcallist = ['0137+331']
phasecallist = ['2250+143', '0119+321', '0237+288',
                '0239-025', '0323+055', '0339-017',
                '0423-013']
sourcelist = ['NGC7469', 'MRK0993', 'MRK1040',
              'NGC1056', 'NGC1068', 'NGC1194',
              'NGC1241', 'NGC1320', 'F04385-0828',
              'NGC1667']
allcals = ampcallist + phasecallist
calDict = {'NGC7469':'2250+143',
           'MRK0993':'0119+321',
           'MRK1040':'0237+288',
           'NGC1056':'0237+288',
           'NGC1068':'0239-025',
           'NGC1194':'0323+055',
           'NGC1241':'0323+055',
           'NGC1320':'0339-017',
           'F04385-0828':'0423-013',
           'NGC1667':'0423-013'}

importvla(archivefiles=xfl,
          vis='ag733.ms')

flagcmd(vis='ag733.ms', inpmode='list',
       inpfile=[
           "antenna='VA15' timerange
               ='2006/12/09/04:45:10.0~04:45:20.0'",
           "antenna='VA15' timerange
               ='2006/12/09/06:12:30.0~06:14:10.0'",
           "antenna='VA15' timerange
               ='2006/12/13/04:32:40.0~04:35:30.0'",
           "antenna='VA15' timerange
               ='2006/12/09/06:12:30.0~06:12:40.0'",
           "antenna='VA15' timerange
               ='2006/12/09/06:13:00.0~06:13:40.0'",
            "antenna='VA15' timerange
                ='2006/12/09/06:13:50.0~06:14:00.0'",
            "antenna='VA15' timerange
                ='2006/12/09/06:14:00.0~06:14:10.0'"],
        action='apply')

flagdata(vis='ag733.ms',
          timerange='2006/12/09/06:24:50.0~06:25:00.0')

setjy(vis='ag733.ms', field = '0137+331',
      modimage = '/home/bnikolic/p/VLA/CalModels/3C48_C.im')

gaincal(vis='ag733.ms',caltable='cal.G',
        field=','.join(allcals), solint='inf',
        refant='VA05', append=False)

flagdata(vis='ag733.ms',antenna='EA26',
          timerange='2006/12/13/0:0:0~24:0:0')
gaincal(vis='ag733.ms',caltable='cal.G',
        field=','.join(allcals), solint='inf',
        refant='VA05', append=False)

fluxscale(vis='ag733.ms', caltable='cal.G',
          reference=','.join(ampcallist),
          transfer=','.join(phasecallist),
          fluxtable='cal.Gflx', append=False)

for source, calibrator in calDict.iteritems():
    applycal(vis='ag733.ms', field=source,
             gaintable='cal.Gflx', gainfield=calibrator)

blcal(vis = 'ag733.ms', caltable = 'cal.BL', field =
    '0137+331',
      solint = 'inf', gaintable = 'cal.Gflx',
      gainfield = '0137+331', interp = 'nearest',
      timerange = '2006/12/09/0:0:0~24:0:0')

for source, calibrator in calDict.iteritems():
    applycal(vis='ag733.ms', field=source,
             gaintable=['cal.Gflx','cal.BL'],
             gainfield=calibrator)

for source in sourcelist:
    splitfile = source + '.split.ms'
    split(vis='ag733.ms',outputvis=splitfile,
          datacolumn='corrected', field=source)


for source in sourcelist:
    splitfile = source + '.split.ms'
    clean(vis=splitfile,
          imagename = source + '_img',
          mode      =      'mfs',
          niter              =       2000,
gain                =        0.1,
threshold           = '4.5e-5Jy',
psfmode             =    'clark',
imagermode          =  'csclean',
cyclefactor     =          3,
cyclespeedup    =         -1,
multiscale          =         [],
interactive         =      False,
mask                =         [],
imsize              = [1024, 1024],
cell                = ['0.75arcsec', '0.75arcsec'],
stokes              =        'I',
weighting           =  'natural',
uvtaper             =      False,
pbcor               =      False,
minpb               =        0.1,
usescratch          =      False,
async               =      False      )
```

References

Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman (1986). *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0-201-10088-6.

Alpern, B., M. N. Wegman, and F. K. Zadeck (1988). "Detecting Equality of Variables in Programs". In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '88. San Diego, California, USA: ACM, pp. 1–11. ISBN: 0-89791-252-7. DOI: `10.1145/73560.73561`. URL: `http://doi.acm.org/10.1145/73560.73561`.

Appel, Andrew W. (1997). *Modern Compiler Implementation in C: Basic Techniques*. New York, NY, USA: Cambridge University Press. ISBN: 0521583896.

— (1998a). *Modern compiler implementation in ML*. Cambridge: Cambridge University Press. ISBN: 0-521-58274-1.

— (1998b). "SSA is Functional Programming". In: *ACM SIGPLAN NOTICES* 33.4, pp. 17–20.

Arvind, K.P. Gostelow, and W.E. Plouffe (1978). *An Asynchronous Programming Language and Computing Machine*. Tech. rep. TR114a. Irvine.

Bonwick, Jeff and B. Moore (2008). *ZFS: The Last Word in File Systems*. Paper presented at SNIA Software Developers' Conference. URL: `http://www.cs.utexas.edu/users/dahlin/Classes/GradOS/papers/zfs_lc_preso.pdf`.

Chawathe, Sudarshan S., Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom (1996). "Change Detection in Hierarchically Structured Information". In: *Proceedings of the 1996 ACM SIGMOD International*

Conference on Management of Data. SIGMOD '96. Montreal, Quebec, Canada: ACM, pp. 493–504. ISBN: 0-89791-794-4. DOI: `10.1145/233269.233366`. URL: `http://doi.acm.org/10.1145/233269.233366`.

Cytron, Ron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck (1991). "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". In: *ACM Trans. Program. Lang. Syst.* 13.4, pp. 451–490. ISSN: 0164-0925. DOI: `10.1145/115372.115320`. URL: `http://doi.acm.org/10.1145/115372.115320`.

de Vos, Marco (2012). *Hilado Deliverable D10.1: Activity Plan, architecture and selection of benchmarks*. URL: `http://www.radionet-eu.org/radionet3wiki/lib/exe/fetch.php?media=jra:d10.1_rn3_hilado_deliverable_130502.pdf`.

Eder, Johann and Karl Wiggisser (2006). "A DAG Comparison Algorithm and Its Application to Temporal Data Warehousing". English. In: *Advances in Conceptual Modeling - Theory and Practice*. Ed. by JohnF. Roddick et al. Vol. 4231. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 217–226. ISBN: 978-3-540-47703-7. DOI: `10.1007/11908883_27`. URL: `http://dx.doi.org/10.1007/11908883_27`.

— (2007). "Detecting Changes in Ontologies via DAG Comparison". In: *Lecture Notes in Computer Science 4495*, pp. 21–35.

Fluri, Beat, Michael Wuersch, Martin PInzger, and Harald Gall (2007). "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction". In: *IEEE Trans. Softw. Eng.* 33.11, pp. 725–743. ISSN: 0098-5589. DOI: `10.1109/TSE.2007.70731`. URL: `http://dx.doi.org/10.1109/TSE.2007.70731`.

Gansner, Emden R. and Stephen C. North (2000). "An open graph visualization system and its applications to software engineering". In: *SOFTWARE - PRACTICE AND EXPERIENCE* 30.11, pp. 1203–1233. URL: `http://www.graphviz.org/`.

Harrold, Mary Jean, Brian Malloy, and Gregg Rothermel (1993). "Efficient Construction of Program Dependence Graphs". In: *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '93. Cambridge, Massachusetts, USA: ACM, pp. 160–170. ISBN: 0-89791-608-5. DOI: `10.1145/154183.154268`. URL: `http://doi.acm.org/10.1145/154183.154268`.

Kettenis, Mark, Huib Jan van Langevelde, Cormac Reynolds, and William D. Cotton (2006). "ParselTongue: AIPS Talking Python". In: *Astronomical Data Analysis Software and Systems XV*. Ed. by C. Gabriel, C. Arviset, D. Ponz, and S. Enrique. Vol. 351. Astronomical Society of the Pacific Conference Series, p. 497.

NRAO Casa Webpage. URL: `http://casa.nrao.edu/`.

Peyton Jones, Simon et al. (2003). "The Haskell 98 Language and Libraries: The Revised Report". In: *Journal of Functional Programming* 13.1. `http://www.haskell.org/definition/`, pp. 0–255.

The ParselTongue Wiki (2015). URL: `http://www.jive.nl/jivewiki/doku.php?id=parseltongue:parseltongue`.

van Rossum, Guido (1995). *Python Reference Manual*. Tech. rep. Amsterdam, The Netherlands, The Netherlands. URL: `http://www.python.org/`.

Wells, D.C. (1985). “NRAOâĂŹs Astronomical Image Processing System (AIPS)”. English. In: *Data Analysis in Astronomy*. Ed. by V.Di GesÃź, L. Scarsi, P. Crane, J.H. Friedman, and S. Levialdi. Vol. 24. Ettore Majorana International Science Series. Springer US, pp. 195–209. ISBN: 978-1-4615-9435-2. DOI: `10.1007/978-1-4615-9433-8_18`. URL: `http://dx.doi.org/10.1007/978-1-4615-9433-8_18`.

Wilde, Michael, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster (2011). “Swift: A Language for Distributed Parallel Scripting”. In: *Parallel Computing* 37.9, pp. 633–652. ISSN: 0167-8191. DOI: `10.1016/j.parco.2011.05.005`. URL: `http://dx.doi.org/10.1016/j.parco.2011.05.005`.