

af1ak: Advanced Framework for Learning Astrophysical Knowledge

—A visual programming approach to analyze multi-spectral astronomical data—

— ビジュアルプログラミング・アプローチを用いた
天文学における分光データ解析 —

Malik Olivier BOUSSEJRA

Graduate School of Science and Technology
Center for Information and Computer Science
Keio University

慶應義塾大学
大学院理工学研究科 開放環境科学専攻

This dissertation is submitted for the degree of
Doctor of Philosophy in Engineering

August 2019

I would like to dedicate this thesis to my loving family, especially 李若菲.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements.

Malik Olivier BOUSSEJRA
August 2019

Acknowledgements

Nothing could have been done with the tremendous help from my advisor, Professor Issei Fujishiro, yet a lot of people assisted me at all times I needed help, including many specialists in astronomy. I should first warmly thank the members of the jury, Professor Makoto Uemura, Professor Tomoharu Oka and Professor Hideo Saito for their feedback on my doctoral defense. Professor Fujishiro introduced Professor Makoto Uemura from Hiroshima University, who in turned introduced Dr. Kazuya Matsubayashi from the Department of Astronomy at Kyoto University. Dr. Matsubayashi guided us towards solutions to astronomical computational problems, especially equivalent widths. Professor Yuriko Takeshima from Tokyo University of Technology provided valuable pieces of advice thanks to her expertise in visualization, and helped in evaluating a `flak`.

When I was struggling to find a new theme for my Ph.D. thesis, while wandering in Keio University's Yagami campus, I met with Professor Tomoharu Oka from the same university, who introduced me to his then Ph.D. student Shunya Takekawa. Shunya is now a researcher at the Nobeyama Radio Observatory in Minamimaki, Nagano. He is the one who guided me into approaching the problem with a visual programming language for astronomical data analysis. I am extremely thankful for his advice, and his recommendations of various conferences and workshops to present a `flak`.

During one of such conferences, the Astronomical Data Analysis Software & System (ADASS) *XXVIII* conference, I met Dr. François Bonnardel from the Strasbourg Astronomical Data Center, to whom I would like to extend warm thanks for his thorough explanations and knowledge about various astronomical standards, especially the SIA standard. The same thanks should be directed to Dr. Hendrik Heintz from the Center for Astronomy of the University of Heidelberg for the tutorial he presented about IVOA interoperability standards and the time he offered me for answering a beginner's questions.

In my own laboratory, Mr. Rikuo Uchiki implemented most of the astrophysical primitives, while I was mostly concerned with the overall design and the computational engine. Mr. Yuhei Iwata, a Ph.D. candidate under Professor Oka, helped with some debugging on the macOS release. Thanks to him, a `flak` could run on the latest Mojave.

I should as well thank all the people who provided their help during a concurrent project on forensic medicine. A lot of people assisted me at all time I needed help, including specialists in forensic medicine from University of Yamanashi starting from Professor Noboru Adachi, Professor Hideki Shojo and Dr. Ryohei Takahashi, but also specialists in computer graphics such as Professor Xiaoyang Mao and Professor Masahiro Toyoura. Professor Fujishiro was the first person who thought about applying computer visualization to computational forensics, and who proposed to create a mark-up language to describe forensic medicine operations. Professor Noboru Adachi, Professor Hideo Shojo and Dr. Ryohei Takahashi shared with us all their forensic knowledge for us to build a semantically correct mark-up language, and their constant suggestions for improving our framework were priceless.

This work is supported in part by both JSPS KAKENHI Grant Numbers 17K00173 and 17H00737. Until 2017, the work was as well supported in part by JSPS KAKENHI under Grant-in-Aid for Scientific Research (A) No. 26240015, and by a Grant-in-Aid for the Leading Graduate School program for “Science for Development of Super Mature Society” from the Ministry of Education, Culture, Sport, Science, and Technology in Japan.

The template used to typeset this thesis is based on the Cambridge University Engineering Department Ph. D. thesis template created by Krishna Kumar.

**af1ak: Advanced Framework for Learning
Astrophysical Knowledge
—A visual programming approach
to analyze multi-spectral astronomical data—**

Abstract

This thesis describes an extendable graphical framework, `af1ak` (Advanced Framework for Learning Astrophysical Knowledge), which provides a collaborative visualization environment for the exploratory analysis of multi-spectral astronomical datasets. `af1ak` allows the astronomer to share and define analytics pipelines through a node editing interface, in which the user can compose together a set of built-in transforms (e.g. dataset import, integration, Gaussian fit) over astronomical datasets. `af1ak` supports the loading and exporting of *Flexible Image Transport System* (FITS) files, the *de facto* data interchange format used in astronomy, allowing interoperability with existing astronomy software.

The contribution of this thesis lies in that we leverage visual programming techniques to conduct fine-grained astronomical transformations, filtering and visual analyses on multi-spectral datasets, with the possibility for astronomers to interactively fine-tune all the interacting parameters. As the visual program is gradually designed, the computed results can be visualized in real time, thus `af1ak` puts the astronomer in the loop.

Moreover, `af1ak` can query and load datasets from public astronomical data repositories, by implementing standard query protocols used in astronomy, such as Simple Image Access (SIA). `af1ak` extends the FITS standard in a manner such that the full provenance of the output data created by `af1ak` be preserved and reviewable, and that the exported file be usable by other common astronomical analysis software. By embedding `af1ak`'s provenance data into FITS files, we both achieve interoperability with existing software and full end-to-end reproducibility of the process by which astronomers make discoveries.

Not only is `af1ak` fast and responsive, but the macro it supports can be conveniently exported, imported and shared among researchers using a purposefully built data interchange format and protocol. A user can implement some common analytical procedure either by combining several nodes and creating a macro with `af1ak`'s user interface, or by programmatically implementing new nodes in the Rust programming language.

During the development of `af1ak`, we worked together with astronomers to provide a universal tool that suits their analytic needs as close as possible. `af1ak` joins ease of use, responsiveness, shareability and reproducibility of the analytical process by which astronomers make discoveries.

Keywords

Astronomy, Astrophysics, Provenance, Visual Programming, Scientific Visualization.

aflak: Advanced Framework for Learning Astrophysical Knowledge

—ビジュアルプログラミング手法を使用した天文学における
分光データ解析—

論文要旨

本学位請求論文では、協働的な可視化環境を提供することにより、天文学分光データの探究的分析を可能とする拡張可能なグラフィカルフレームワーク **aflak** (Advanced Framework for Learning Astronomical Knowledge) を提案する。そのノードエディタ・インタフェースを通じて、天文学者はシステムに予め組み込まれた変換関数（天文データの取込み、積分、ガウス関数フィッティング等）から分析パイプラインを組み上げられるほか、そのパイプラインを他の研究者と共有することもできる。**aflak** は、天文学における事実上の標準データ交換フォーマットである Flexible Image Transport System (FITS) のファイル読み込み/書き出しに対応しているため、既存の天文学ソフトウェアとの相互運用も可能である。

本研究の主要な貢献は、ビジュアルプログラミングの技術を活用して、天文学で多用されている、粒度の細かい変換やフィルタリング、視覚分析に関連するパラメタ群を容易に微調整させられることである。ビジュアルプログラムの設計途中でも、その算出結果をリアルタイムで可視化できるので、まさに「天文学者・イン・ザ・ループ」を構築している。

さらに **aflak** は、Simple Image Access (SIA) 等の天文学の標準的な照会プロトコルに対応しており、公開されている天文データレポジトリのデータセットを照会したり読み込んだりすることができる。しかも **aflak** は、FITS 様式を拡張して、書き出したデータの完全な出自を保持できるので、外部での検閲を許すだけでなく、その FITS ファイルを他の天文ソフトウェアにも読み込ませることや、天文学者の発見につながったデータ処理プロセスを完全に再現することも可能である。

aflak の特長は速い応答性だけではない。そのユーザインタフェースを利用して、ユーザは複数のノードを組み合わせて一般的な分析手順を提供するマクロを作成することもできる。また、マクロ定義に利用される新たなノードを、プログラミング言語 Rust で実装することもできる。作成されたマクロは、意図的に構

築されたデータ交換フォーマットとプロトコルを利用して読み込み/書き出しができ、研究者間で便利な共有も可能である。

aflak は、天文学者のニーズに最大限に応えられるような汎用ツールを提供する目的で、天文学者と協同で開発されてきた。aflak は、使いやすさや応答性、共有可能性、発見に資する分析プロセスの再現性を兼備している。

キーワード

天文学，宇宙物理学，出自，ビジュアルプログラミング，科学技術データ可視化。

Table of contents

List of figures	xiii
List of tables	xvii
1 Introduction: How Do Astronomers Analyze Datasets?	1
1.1 A collaborative endeavor	1
1.2 Data used by astronomers	2
1.3 Data retrieval and analysis for the astronomer	3
1.4 An aside about forensics	4
1.5 af1ak's introduction	5
2 Related Works	7
2.1 Viewers and analyzers for astronomical use	7
2.1.1 Viewing astronomical datasets	7
2.1.2 Inspiration from non-astronomy viewers	8
2.1.3 Tools for data analytics	8
2.1.4 Moving to Python	11
2.2 Visual programming language paradigm	12
2.3 A visual programming approach for viewing and analyzing astronomical datasets	13
2.4 Visualization and provenance in forensic science	15
3 af1ak: Advanced Framework for Learning Astronomical Knowledge	20
3.1 Background and motivation	20
3.2 Design goals	22
3.2.1 Ease of use and responsiveness	22
3.2.2 Re-usability and extendability	22
3.2.3 Collaborative development	22
3.3 System overview	22

3.3.1	Description	22
3.3.2	Detailed description of components	23
3.3.3	Value nodes, type checking and error handling	32
4	Implementation Details	35
4.1	Description of algorithms and implementation	35
4.1.1	Language and library choices	35
4.1.2	Multi-crate structure	36
4.1.3	cake: Computation mAKe	36
4.1.4	<i>MetaTransform</i> data structure	38
4.1.5	Computing output with cache	40
4.2	Macro support for cake	41
4.2.1	Design decisions	41
4.2.2	Data structures for macro support	42
4.2.3	Some changes in computation logic	43
4.2.4	Macro user interface	43
4.3	SIA integration for provenance management	45
4.3.1	Overview of the SIA specification	45
4.3.2	Integration with SIA	45
4.3.3	Provenance management with aflak	46
4.4	User interface: An event-based architecture	48
4.5	Implementing astronomical libraries in Rust	49
4.5.1	FITS libraries	49
4.5.2	Convenience in opening FITS files	50
4.5.3	Virtual Observatory standards	51
4.6	Defining your own nodes with Rust	51
4.7	DevOps	53
4.7.1	Portability: Challenges in supporting Linux (Debian and Ubuntu), macOS and Windows	54
4.7.2	Development workflow	54
4.7.3	Release mechanism	54
5	Evaluation	56
5.1	Checking compliance to standards	56
5.2	First use case: Equivalent width	58
5.2.1	Introduction to <i>human-in-the-loop</i> concept	58
5.2.2	Use case	59

5.3	Second use case: Velocity field map	59
5.4	Comparison with current tools	59
5.4.1	In-depth comparison	61
5.4.2	Equivalent width with a macro	63
5.5	Advantages of provenance management in a visual context not limited to astronomy	63
5.6	Distribution and recognition	66
6	Future Works	69
6.1	Stronger interoperability with VO standards	69
6.2	Application to other astronomical problems	70
6.2.1	Arbitrary non-linear slicing	70
6.2.2	Interferometry	70
6.3	More room for improvement	73
7	Conclusion	74
	References	76

List of figures

1.1	General workflow for a user of the proposed framework, a f l a k. Datasets can be loaded either from the file system or by issuing requests to open data repositories. The visual program can be interactively developed while the user can see a visualization of the program's output in real time.	6
1.2	General workflow of the <i>Legal Medicine Mark-up Language</i> framework, with all the involved users.	6
2.1	A screen capture of SAOImage DS9. Though DS9 is mainly used as a viewer, we can still see some sub-menus related to data analysis.	9
2.2	A screen capture of PyQtGraph in use. Output windows in a f l a k's interface to visualize datacubes were inspired from PyQtGraph's interface (screen capture from [Cam]).	10
2.3	AVS/Express screenshot, from Bungartz <i>et al</i> [BFM98].	13
2.4	KNIME's node editor interface.	14
2.5	Example of a semi-complex use of a f l a k to extract the equivalent width of a three-dimensional dataset (refer to Carroll's book for precise definition of <i>equivalent width</i> [CO07]). The <code>open_fits</code> node opens a FITS file, then several transformations are applied to the file to extract the equivalent width in the right-most output node (Output #3). The result of each output node is visualized in a corresponding output window. Continuum emission is computed by node #4 on the left side, and by node #6 on the right side. The average of the emission line is computed by node #5. Intermediary results are visualized via output nodes #1 and #2.	16
2.6	LMML Browser: Data input interface (head external examination). Top: Fixed forms for data relevant to head examination. Bottom: Preview of the paragraph dealing with the head included in the final forensic report. . . .	18
2.7	LMML Browser: A visualization to explore the result of the autopsy. . . .	19

3.1	An example of transformation node, which computes the linear composition of two images. It has four inputs (on the left): two images u , v and two scalar parameters a and b . The transformation node has a single output slot (on the right) from which the image $au + bv$ comes out. Parameters a and b can either be directly input by the user inside the node, or can be taken from the value of an output of another node.	23
3.2	General UI of <code>af1ak</code> . We can see all of <code>af1ak</code> 's components. (a) Current node list (b) Documentation of last selected node (<i>active</i> node) (c) Node editor, with output nodes on the left (d) Output windows for visualization	25
3.3	Left pane of <code>af1ak</code> 's interface, containing the list of nodes in the current editor's window.	26
3.4	The figure shows the pop-up dialog that allows to select a new node to add to the node editor's graph. The pop-up dialog can be opened by right-clicking on any empty space in the node editor.	27
3.5	Interfaces of an output window showing a 2D image with several color maps and axes.	29
3.6	Examples of interaction handles representing a bound value in a visualization of a two-dimensional image.	30
3.7	Visualization of the waveform of a datacube. This is the interface of a one-dimensional image when displayed in an output window. The vertical line is an interactive handle that represents an x -axis value selected by the user.	31
3.8	Many types of value nodes as they appear in <code>af1ak</code>	33
3.9	<code>af1ak</code> will prevent the user from wasting their CPU resources. No cyclic dependencies can be created. The same kind of errors will be detected, prevented and a message will be displayed if a user tries to nest recursive macros.	33
3.10	An output window shows a runtime error. The <code>fits_to_image</code> node cannot find a FITS HDU with index 20. So the error bubbles up to Output #2, and its stack-trace is displayed in the output window for easy debugging.	34
4.1	<code>af1ak</code> 's modular structure.	37

- 4.2 Example of a macro that opens an image from the MaNGA dataset [B⁺15]. This simple macro takes two inputs: a FITS file and an integer representing a frame number. The macro then outputs the waveform and the image data at the provided frame. This is the result that can be seen in output windows #1 and #2. The way the FITS file is open and the choice of extension (FLUX) is quite specific to the MaNGA dataset, hence the merit of defining a macro for such task. 44
- 4.3 `af1ak` querying and displaying an object from the GAVO (*German Astrophysical Virtual Observatory*) repository. The user can select the sky coordinates of the object and a data repository that will be queried. The record is then downloaded, cached and displayed on the screen. Some intermediary results, like the direct URL from which the image is downloaded or FITS metadata are displayed as well. The displayed object has the following ID: `ivo://org.gavo.dc/~?rosat/image_data/rda_1/us900176p-1_n1_p1_r2_f2_p1/rp900176a01_mex.fits.gz` 47
- 4.4 FITS file structure. Image courtesy of *Introduction to the HST Data Handbooks*, section 2.2 on “Multi-Extension FITS File Format” [S⁺11]. 50
- 5.1 Image with a size in the gigabyte range queried from GAVO SIAP 1.0 repository, as shown within `af1ak`. SIAP 1.0 dates from around 2002 and does not support any sort of “ID” for datasets. GAVO SIAP 1.0: `http://dc.zah.uni-heidelberg.de/hppunion/q/im/siap.xml` 57
- 5.2 The visualization discovery process as presented by Johnson *et al.* [JMM⁺05]. This figure represents the core visualization concept of *human-in-the-loop*. 58
- 5.3 A node graph for computing the velocity field map using the effect of Doppler shift on an emission line. It generates two field maps (one in output #1 and one in output #2) with two different computation methods. Average value of the image around the emission line is shown in output #3. We see that there is a fast moving object on the lower right-hand side. . . 60
- 5.4 Example of using an `af1ak` macro to compute equivalent width. The macro encapsulates all the logic implemented as shown in the node editor in Figure 2.5, only exposing the relevant constants that the astronomers are expected to gradually adjust until they get a satisfactory outcome. . . 64
- 5.5 An example graph of provenance discovery. Starting with a released dataset (left), the involved activities (blue boxes), progenitor entities (yellow rounded boxes) and responsible agents (orange pentagons) are discovered. [SRB⁺19] 66

5.6	Provenance in forensics: Reconstitution of crime according to different hypotheses.	66
5.7	af1ak's repository has 12 stargazers. We never attempted to do any kind of pro-active campaign to earn stars.	67
6.1	Representation of af1ak's planned extension to support arbitrary slicing.	71
6.2	Block diagram of AIPS from a user point of view. Various communications paths are shown among the main interactive program, AIPS, the batch program AIPSB, and the collection of separate tasks (Figure 1 in [Gre03]). . .	72

List of tables

4.1	Dynamic Syntax Tree data structure	38
4.2	MetaTransform data structure	39
4.3	Transform data structure	39
4.4	Data structure of Algorithm enumeration	39
4.5	Macro data structure	42

List of listings

2.1	Example of use of IRAF's command-line interface	11
4.1	Event enumeration used for node editor's user interface	48
4.2	How to define your own custom node in Rust	52
4.3	Load a FITS file	53
5.1	Extracting equivalent width with Bash/IRAF	62
5.2	Installing a <code>flak</code> : A one-liner.	68

Chapter 1

Introduction: How Do Astronomers Analyze Datasets?

“Astronomy is similar to forensic science, in that it relies entirely on the detection and analysis of the leftovers of past events—mostly consisting of radiation—to reconstruct a plausible explanation for what is being observed.”

Author’s re-rendering of a speech by Kirk Borne [Bor18]

1.1 A collaborative endeavor

As all scientific disciplines, astrophysics requires deep collaborative support among researchers in order to make breakthrough results. The cliché of the lonely researcher hardly exists anymore in real life. Astrophysicists are expected to provide to their peers reproducible research and accurate retelling of the analytic process by which a result was achieved. This cooperation is clearly shown by the ever increasing average number of co-authors in papers, including for example the 2019 highly mediatized technological development that allowed the M87* black hole to be directly imaged. Imaging such a far away object requires a telescope roughly the size of the Earth in order to mitigate light diffraction on collecting the photons from an object with an angular size as small as M87*. Such feat can only be done by creating a virtual telescope through the cooperation of many telescopes scattered over the whole Earth—the paper by Akiyama *et al.* that revealed the black hole image boasts 143 different affiliations [AAA⁺19].

Interestingly, astronomy is sometimes compared to forensic science in that astronomers are only able to gather the traces of what is left of far away physical phenomena, and from

there draw conclusions [Bor18]. Indeed, all that is left to analyze and understand these phenomena is mostly the radiation that was emitted from them eons ago¹.

In astrophysics, datasets are already publicly shared via open repositories, with each data sample being assigned a unique identifier, but not the same can be said about the complete raw analysis process, which involves everything from the original data to the output that allowed an astronomer to draw specific conclusions. This of course includes the analytical program devised by the astronomer, but as well all the steps by which this program was refined. Hence, we can foresee the need for better sharing practices of analytical processes and provenance data to improve reproducibility and potential for collaboration.

1.2 Data used by astronomers

Astrophysical data typically consists of multi-spectral images. Depending on the specific field of study an astrophysicist may dive in, one may encounter datasets with three to five dimensions. In addition to the two common spatial axes (usually right ascension and declination, but other kinds of domain-specific spacial coordinates, such as galactic coordinates may as well be used), one may found:

Wavelength: The spacial period of a given ray received by a pixel's sensor. By taking into account the Doppler effect, radial speed towards the Earth may be extracted from this value. Thus datasets only containing the resulting speed may be found (e.g. datasets used by Oka *et al.* [OTI⁺17]).

Polarization: A value characteristic of specific milieu or objects (e.g. *blazars* emit a polarized light [FSN⁺18]).

Time: Dynamic phenomenon may include time-dependent data (e.g. solar flare).

Those kinds of multi-dimensional data are represented as datacubes. The most common format used by astronomers to store and exchange such data is *Flexible Image Transport System* (FITS) [WG79]. FITS is a data format developed in the 70s and 80s for university mainframes, already from a need to facilitate data exchange between research centers, and has evolved along the decades it has been in use. FITS is a very interesting format to investigate for the interested. Through its evolution, it has remained backward-compatible with its newer extended features, but still displays peculiar characteristics for the modern observer. Indeed, FITS was developed to be conveniently recorded on IBM punch-cards,

¹As a pedantic addition, apart from electromagnetic radiation, neutrinos or gravitational waves may as well be detected.

which were 80 columns long. This is why if one takes a look at the modern FITS specification, one will see many instances of paddings to fill bits by multiples of 80 characters. All those rules may seem strange to the non-initiated, however they make sense when looked through the lens of their historical backgrounds. This fact alone is of deep intellectual interest for anyone dealing with a parser or a writer that must support this format. The author had the privilege to meet with Jessica Mink, the creator of the FITS format, who also happens to be the discoverer of the rings of Uranus.

A simple FITS file will be separated into two parts: header and data. The header will contain, as one may expect, metadata regarding the content of the file². The data will contain an array of numbers, represented using a little-endian representation, which is the contrary to the big-endian representation of numbers now commonly used by our processors. A complying modern FITS parser will thus require to do bit-wise conversions on floating point representations of real numbers.

Apart from number crunching, astronomers also deal with data tables. An extension of FITS allows to encode data tables within the data part of a FITS file. A standard derived from the representation was codified and is called ObsCore (Observation Data Model Core Components and its Implementation in the Table Access Protocol) [LTD⁺17].

1.3 Data retrieval and analysis for the astronomer

An astronomer's workflow can be broken down into the following four steps:

1. Find or query a dataset of interest.
2. Run some analytical task on the object.
3. Observe the result of the analytical tasks.
4. Draw conclusions or go back to any of the previous tasks.

Finding and querying datasets can be done using data repositories. Astronomical data repositories usually expose APIs following certain standards to allow the user to query datasets according to their properties. Among such standards, the *Simple Image Access* (SIA) protocol is widely used [OLD⁺08]. Objects for example can be queried depending on their sky coordinates or luminosity. Query along sky coordinates is usually dubbed "cone search," as even if two objects appear in the same area of the sky, they can be extremely

²This is not without reminiscing of DICOM (Digital Imaging and COmmunication in Medicine) as used in medicine.

far from each other, both lying within an extremely thin cone (with a dimension ranging around the order of a few milliseconds of arc) stretching from the Earth and passing through the objects.

The astronomer can then use analytical tools to analyze the queried dataset. Such tool will compute common image algebra on the object (Gaussian fit, summation, etc.). The array of analytics is very broad and differs widely between sub-fields of astronomy: a radio-astronomer will not work using the same analytical primitives as an astronomer specializing in the visible light. During our research, we attempted to get feedback from as many astronomers as possible, to avoid alienating any of the tools we develop to a specific subfield. Indeed, the purpose of our research is to develop ubiquitous and general tools.

Finally, after analytical tasks are done, the astronomer must visualize the result of the analysis. Many different FITS viewers exist for example. During that phase, there are several types of conclusions to be drawn. Hopefully, the researcher would be able to find something worthy enough to become an astronomical paper. However, most of the time this will not be the case and the result will not be fruitful. The astronomer may instead conclude that the data they queried does not suit their need, and they will try with another dataset. Or they may conclude that one of the parameters they used during analysis was not exactly appropriate, and they may review the analytical pipeline to start another observation. The point is, whether they can draw appropriate conclusion or not is dependent on the quality of the visualization tools they use.

1.4 An aside about forensics

Kirk Borne said that “astronomy is similar to forensic science, in that it relies entirely on the detection and analysis of the leftovers of past events—mostly consisting of radiation—to reconstruct a plausible explanation for what is being observed” (rephrased by the author).

Forensics is the systematical, scientific method of compiling, analyzing and profiling data or evidence relevant to the unearthing of past events, in order to understand what succession of happenstances led to the given consequences. Let us focus on the *computational* branch of forensic science. The word “computational” can actually be appended to a lot of disciplines, e.g. computational linguistics or computational biology. In the same vein, a body of knowledge called *computational forensics* can be defined. Computational methods provide tools that enable a medical examiner or investigator to better analyze pieces of evidence, otherwise even these experts would be overwhelmed by the sheer amount of information modern data gathering schemes provide [FS08]. The reader is most certainly

already familiar with fingerprint or DNA analysis, which are examples of practical use of computational forensics methods.

We can see that forensic science and astronomy are not devoid of common points. The author of this thesis worked on a visualization environment for forensic medicine, called *Legal Medicine Mark-up Language* (LMML) [BAS⁺16b, BAS⁺16a]. He proposed a framework for computational forensics that allows quicker and smoother input of forensic data collected during autopsy (i.e. the medical examination of a body to uncover the reasons that led to death) through a graphical user interface specially designed to be used while autopsying. From there, the framework can output a graphical visualization of the input data, so that the application be usable and understandable by medical examiners, investigators and judicial courts alike, each with their own respective point of view. Hence, the proposed integrated environment serves the specific needs of all the stakeholders involved in the handling of a forensic case, as shown in Figure 1.2.

Such a system would allow anyone, from the layman to the hardened investigator, to have a better understanding of a case. We see that the challenges that forensic investigators and astronomers face are not without common points, in that the need of tracking the provenance of the measurements and pieces of information from which conclusion are drawn is paramount to explaining the phenomena by which that information was created.

1.5 aflak's introduction

This thesis presents a free and open source software framework, aflak (Advanced Framework for Learning Astrophysical Knowledge)³, which allows interactive analysis and provenance management on multi-dimensional spectral data, from the origin of astronomical data stored into dedicated and public repositories to the final output of the analysis. aflak is developed and hosted on GitHub at <https://github.com/aflak-vis/aflak>. Binaries can be compiled from source or downloaded from there.

A free software model was chosen for aflak to promote the free exchange of programs between research. Indeed, most of current tools in astronomy are developed by astronomers for astronomers. And it is very much natural for them to share the tools they use to make discoveries. The author does not want to break from this model—not because he is not greedy, far from it! Hopefully, his knowledge as a software engineer can be help-

³Interestingly, the name aflak is a backronym: The signification of the initials was chosen by Professor Issei Fujishiro after the author came up with the name aflak. aflak means “stars” in Arabic (أفلاك *aflāk*). This is a tribute to the contribution of traditional Arabic astronomers. Even now, many English names for stars come from Arabic, such as Betelgeuse (from إبط الجوزاء *Ibt al-Jauzā*) or Aldebaran (from الدبران *ad-Dabarān*).

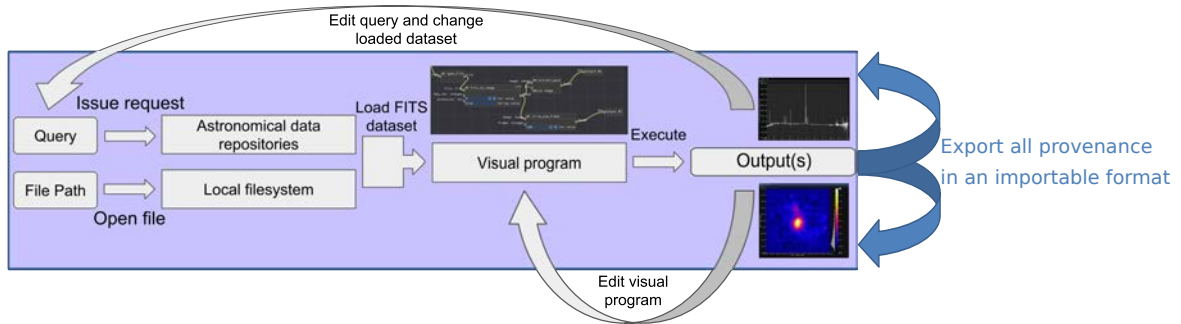


Figure 1.1 General workflow for a user of the proposed framework, a f l a k. Datasets can be loaded either from the file system or by issuing requests to open data repositories. The visual program can be interactively developed while the user can see a visualization of the program's output in real time.

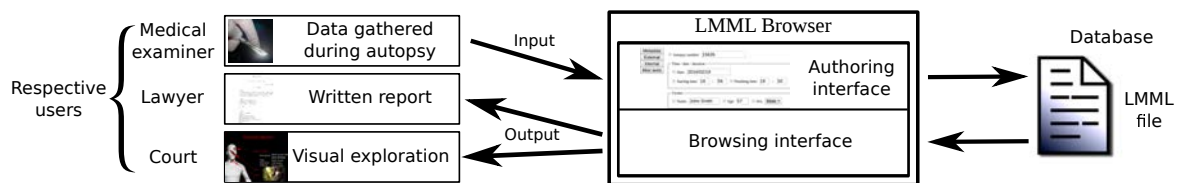


Figure 1.2 General workflow of the *Legal Medicine Mark-up Language* framework, with all the involved users.

ful for astronomers to develop better tools suited for their problems, and astronomers will in turn be able to contribute to the development of a f l a k.

a f l a k provides a visual programming environment that allows to load a dataset, to apply transformations on it and to visualize the outputs of these transformations in real time, thus providing a fast and smooth feedback loop to astronomers. a f l a k has built-in support for FITS files (Flexible Image Transport System, which, as we explained a previous paragraph, has been the *de facto* standard for astrophysical image for a long time, and includes common data transformations used by astronomers [BMT⁺19, BTU⁺19, BUT⁺19]). Visually interacting with the data not only assists the astronomer in finding particular objects, but it also helps in the design of programs by smoothly and regularly checking the output, making rapid prototyping of an analysis pipeline possible.

This thesis will first present the state of the art and the motivations behind the need for a tool such as a f l a k, before describing how the proposed framework contributes to data analytics and provenance management in astrophysics. Then opinions regarding the potential for astronomical research in continuing developing such tools will be expressed.

Chapter 2

Related Works

“[A] handful of astronomers who no longer had patience with how outdated existing tools were and wanted to move to a modern environment began to do so. [...] The reality is that no current astronomy institution could have created the equivalent to Astropy, even if funding were available [...]. This is partly due to institutional culture, partly due to management, and partly due to the extreme competition for funds. [...] The current code infrastructure (e.g. IRAF, DS9) is rapidly crumbling, was never designed or intended to have the longevity being asked of it, and is generations behind modern design and techniques.”

about *Astropy*, by Demitri Muna *et al.* [MAA⁺16]

2.1 Viewers and analyzers for astronomical use

2.1.1 Viewing astronomical datasets

Astrophysicists use many different kinds of viewers, all of them with their own idiosyncrasies and specializations within a specific sub-field. Most of these tools are free and open source software, and we know from astronomers that the apparently most famous and most used of which is SAOImage DS9 [JM03], which can open FITS files and provide simple analytic tools (a screen capture of DS9 can be found in Figure 2.1). Lately, QFitsView [Ott12] has been gaining traction. Even some commercial endeavors, such as NightLight, have been released [Mun17]. Kent *et al.*'s undertaking to re-use existing free modeling software such as Blender to image FITS files deserves notice [Ken13]. We can as well identify new

developments to visualize very large datasets that cannot be loaded into a single modern computer's running memory. As sensor technology improves, it can only be expected that such tools will become ever more important in the future [PQF⁺14, HFB11].

Interestingly, the creator of NightLight, Muna, criticizes the current stance of astrophysicists who only use free software at the expense of productivity. There are not enough contributors to free software in astronomy, with very few incentives to become a maintainer. Indeed, being a maintainer eats an astronomer's precious time, a time that cannot be used to do research or write papers. Few are willing to incur such an opportunity cost. This causes software quality to significantly drop, according to Muna [MAA⁺16]. This makes us reflect on our attempt as visualization researcher to build software for astronomers.

2.1.2 Inspiration from non-astronomy viewers

It is worthy to note that during this research, other data visualization tools that deal with datacubes were also surveyed. One such tool is PyQtGraph [Cam], a Python library maintained by Luke Campagnola, a researcher in neuroscience. PyQtGraph presents many advantages for exploratory analysis and real-time visualization compared to `matplotlib`. It relies on Qt for its rendering engine and is comparatively faster at redrawing than `matplotlib`. Its interface as shown in Figure 2.2 is very intuitive to use. Unfortunately, it is still limited in speed by the Python language it is built with. When `af1ak`'s development was first started, it was attempted to extend PyQtGraph for visualizing astronomical datacubes. However, as soon as the datasets got more than a few hundreds of megabytes, computing speed started to linger and did not satisfy `af1ak`'s requirements for real-time analysis, notwithstanding the attempts to re-write the critical parts of the code in Cython¹.

2.1.3 Tools for data analytics

Nonetheless, the tools cited in the previous sections are viewers and they do not provide many data analytics features, if any. Other tools are in charge of data analytics, the oldest of which being IRAF [Tod86]. Then, PyRAF was developed with the successful objective of providing a more user-friendly programming syntax to the IRAF environment [DLPWG01]. PyRAF, with a Python-based shell syntax, includes a built-in image algebra and many transformations routinely used in astrophysics. PyRAF can execute scripts to enable code re-use. PyRAF, rather than adding feature to IRAF, is more of a new

¹Cython is a compiler for a superset of Python (a sort of "statically typed" Python) that allows to write faster code in a Python environment: <https://cython.org/>.

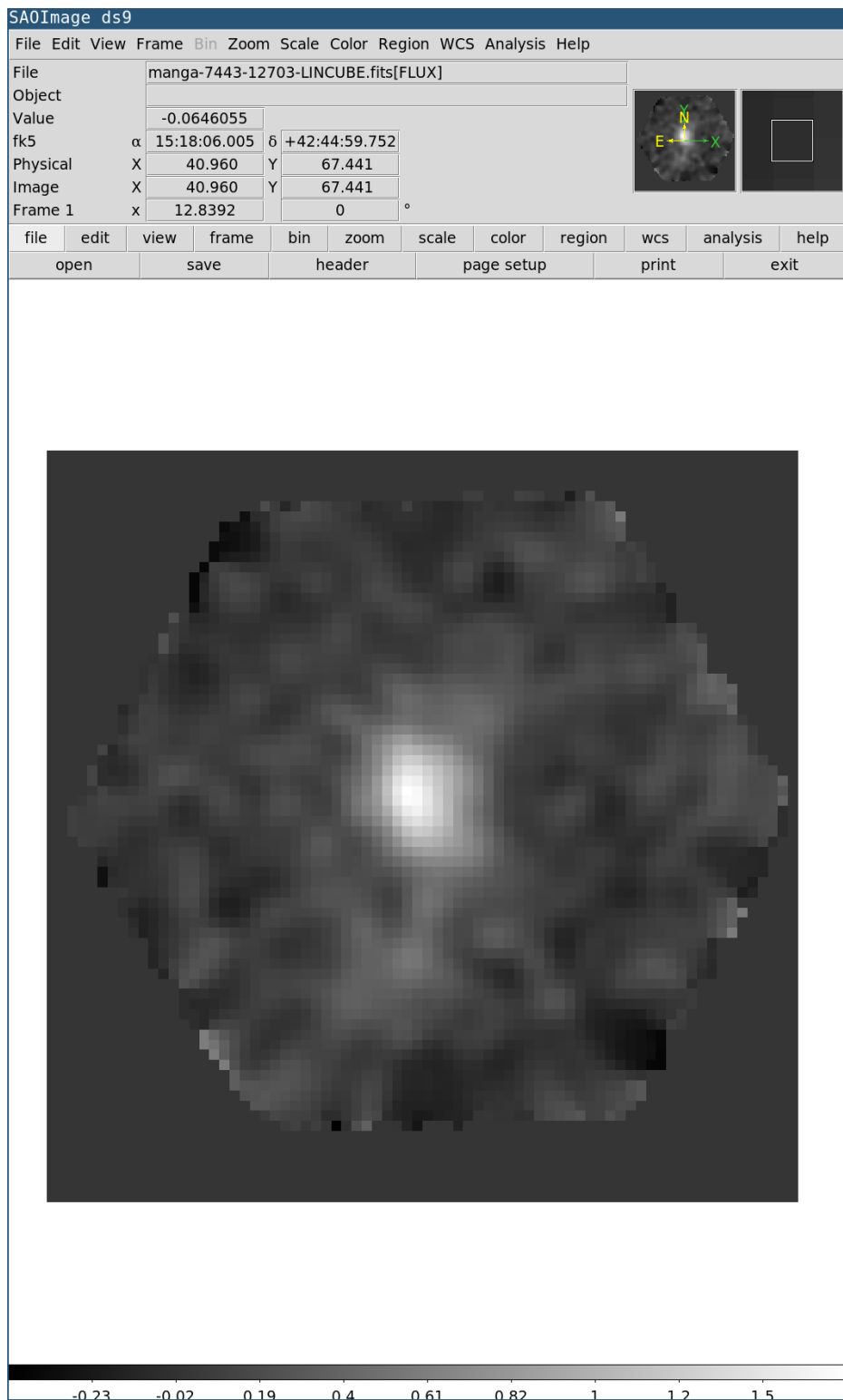


Figure 2.1 A screen capture of SAOImage DS9. Though DS9 is mainly used as a viewer, we can still see some sub-menus related to data analysis.

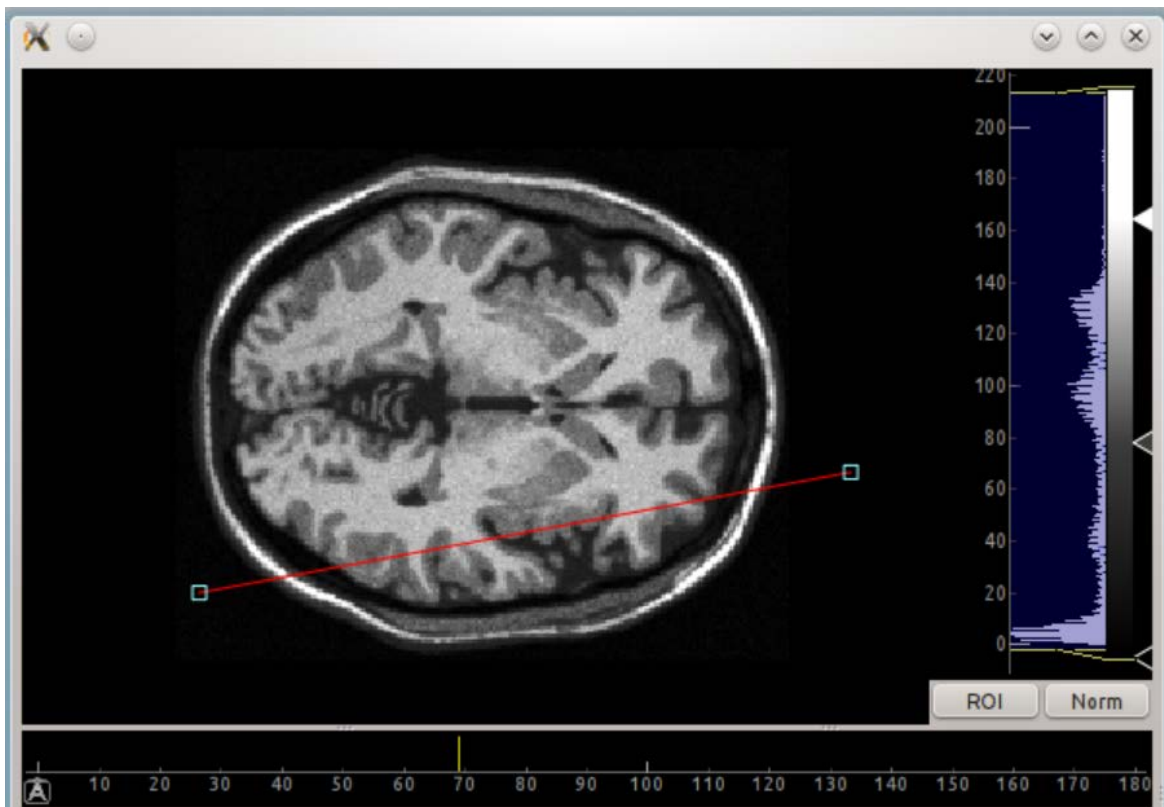


Figure 2.2 A screen capture of PyQtGraph in use. Output windows in aflak’s interface to visualize datacubes were inspired from PyQtGraph’s interface (screen capture from [Cam]).

shell around it that creates a Python-like command-line prompt. We can see an example of IRAF in use in Listing 2.1. Each IRAF command will take some FITS file(s) as input and output a FITS file. For example, the command `imcomb` (for **image combine**) in Listing 2.1, when the `combine` flag is set to `average` and given a datacube as input, will average two-dimensional hyperplanes together into a single two-dimensional image. The other used command, `imarith` in the example (for **image arithmetic**), as its name implies, will do arithmetic operations on FITS files. In this specific example, it will subtract the values in `on-average.fits` from the values in `off-average.fits`, and store the result in `off-on-average.fits`.

Listing 2.1 Example of use of IRAF’s command-line interface

```
imcomb @file-off1.list off1-average.fits combine=average
imcomb @file-on.list on-average.fits combine=average
imcomb @file-off2.list off2-average.fits combine=average
imcomb off1-average.fits , off2-average.fits off-average.fits \
    combine=average weight=@scale-off.dat
imarith off-average.fits - on-average.fits off-on-average.fits
```

2.1.4 Moving to Python

Then came Astropy, a Python library that solves most of the computing needs of astrophysicists [RTG⁺13, MAA⁺16]. With the Python scientific stack (Numpy, SciPy, etc.), it is relatively accessible to implement some custom analysis in Python. Astropy is a library that is actually a collection of smaller astronomical libraries. For example the PyFITS library [BB99] was included in the astropy library as the `astropy.io.fits` module. However, this does not solve the problem of the lack of software maintainers pointed by Muna. Moreover, the Python language is not a general graphical tool. It requires the learning of a programming language and its ecosystem, including many external libraries, the adoption of tools such as `pip`² or `virtualenv`³ to download and use such libraries, etc.

Now, we can see the unfortunate divide between visualization and analysis tools in the astronomy software ecosystem. An astrophysicist’s workflow usually consists in first manually analyzing datasets by applying and composing transformations on them; only then do they export the result—usually as a FITS file—to glance at it inside a viewer. Even for the widely acclaimed Astropy, external libraries (e.g. `matplotlib`) are necessary

²<https://pypi.org/project/pip/>

³<https://pypi.org/project/virtualenv/>

to visualize the results. `matplotlib` may be suitable to programmatically generating appealing figures for publication, but because of the slowness of figure generation when dealing with datasets more than a few hundreds of megabytes, it is not suitable for interactive and exploratory analysis.

2.2 Visual programming language paradigm

Several tools make use of a visual programming approach in order to achieve better accessibility for domain experts. AVS/Express or IBM Data Explorer are the pioneers of visual programming systems for visualization applications, which have been released as commercial software from the 1990s [Cam95]. Examples of use of *modular visualization environments* such as AVS/Express are highly detailed in a paper by Bungartz [BFM98], and a screenshot of the system is shown in Figure 2.3. The base of such modular visualization environment systems is that, as the name indicates, they should be:

modular, in that they include a user-extendable module library;

visualization, in that the program output is a visualization;

environments, in that they provide a visual programming environment.

Furthermore, Parker explains that SCIRun took this concept further by extending the dataflow visual paradigm to include numerical simulations [PJ95]. And in the same vein, MeVisLab leverages analogous concepts applied to medical data [mev]. VisTrails systematizes provenance management with visual graphs representing the pedigree of the analysis output [BCC⁺05], and the same visual approach to workflow management is used by Kepler [ABJF06]. Meyer *et al.* presents Voreen [MSRMH09], which implements an extensive graph editor to define volume visualization procedure based on ray-casting. OpenAlea is an example of visual programming environment to describe plant structural models, which allows real time visualization of the result [PDKB⁺08]. ViPER also presents a visual programming system for real-time molecular visualization [SSO02].

KNIME [BCD⁺09] is another interesting example of analytical system used for example to create machine learning models in bioscience. You can see a sample of KNIME's interface in Figure 2.4. KNIME is a model in term of easy to read visual programming interface. It can exports data in several formats, including CSV (Comma-Separated Values) or JavaScript (using the D3.js⁴ visualization library). Nevertheless, KNIME's transformations have many

⁴Data-Driven Documents: <https://d3js.org/>.

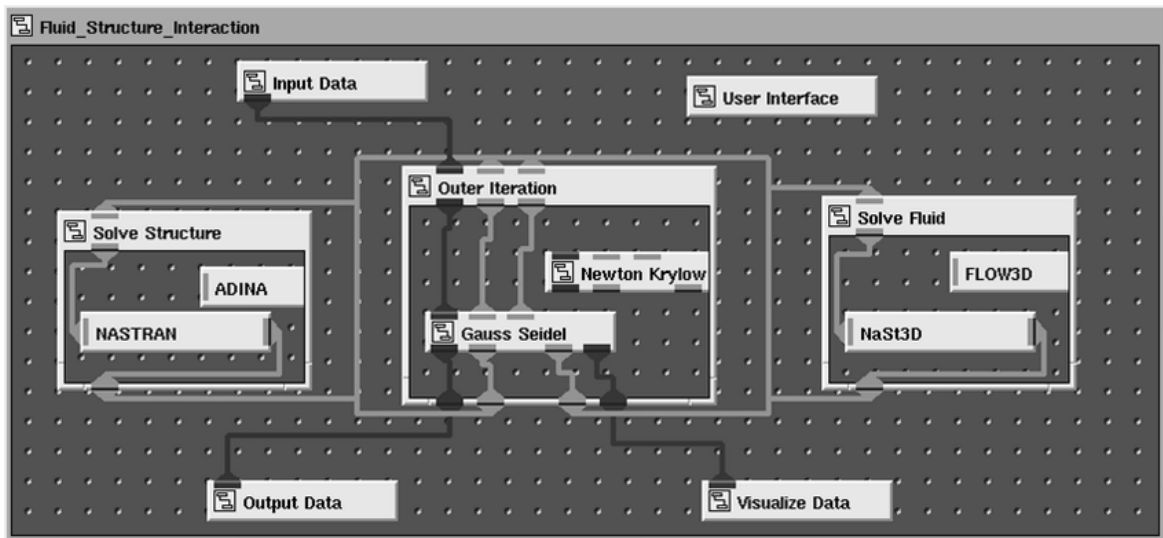


Figure 2.3 AVS/Express screenshot, from Bungartz *et al* [BFM98].

hidden parameters. Fiddling with parameters and finding the options the user wants to modify is a time-consuming process. Besides, rather than recomputing an output in real time, KNIME’s program must be executed using an execution button.

All these systems, which are designed within the scope of a specific domain and/or purpose, use the concept of *node* to represent a module through which the data flows and is transformed.

2.3 A visual programming approach for viewing and analyzing astronomical datasets

To get around the shortcomings raised in section 2.1, `af1ak`’s objective is to provide a universal collaborative and integrated environment to analyze and view astronomical data with very fast iterations [BMT⁺19, BTU⁺19, BUT⁺19]. While `matplotlib` provides publishing quality graphs, it is far from suitable for fast iterations on relatively big datasets. Moreover, there is no built-in solution for code sharing among researchers. The best one can do is to share source files, but then no provenance is supported, reducing accessibility for convenient reproducible research.

Moreover, `af1ak` follows a visual approach similar to the tools mentioned in section 2.2, by combining nodes within a node graph to allow the user to compose algebraic

⁵ Screen capture from <https://www.knime.com/knime-software/knime-analytics-platform> (accessed on August 15th 2019).

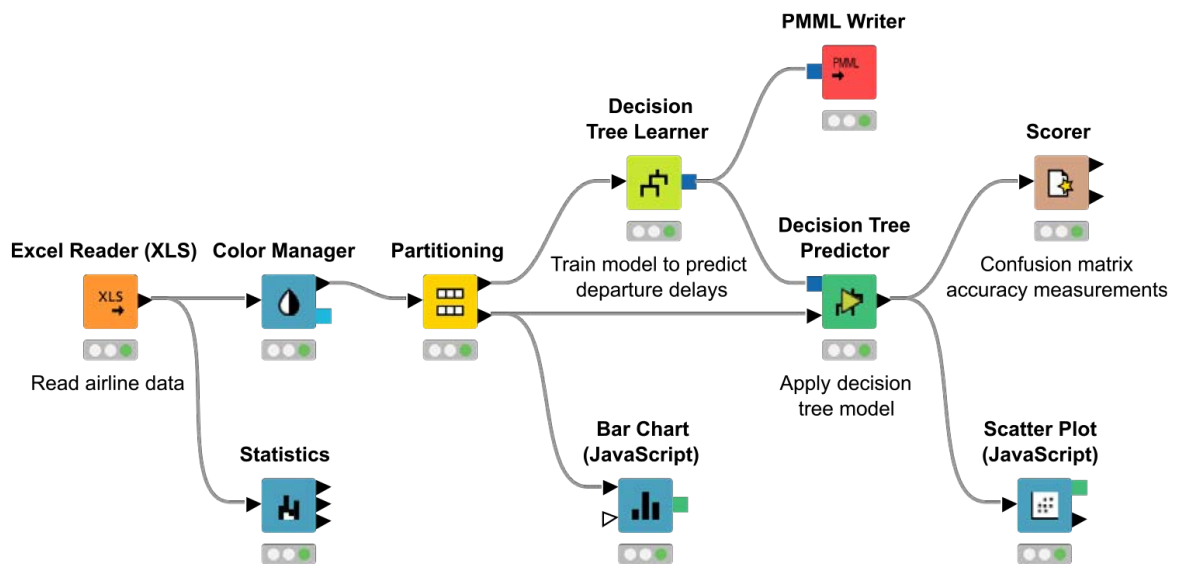


Figure 2.4 KNIME's node editor interface⁵.

transforms. Each node has input slots into which data flows, and a number of output slots from which the transformed data comes out. `af1ak` provides an n -dimensional image algebra interface [RWD90] similar to that of NumPy or IRAF, which can be used by the user to smoothly visualize the resulting computations. Other recent frameworks provide analysis tools for visualizing multi-spectral datasets. This includes BASTet [RB18], a web-based visualization environment fine-tuned for mass spectroscopy imaging. However BASTet does not provide any visual programming feature, contrary to `af1ak`. The reader may refer to Figure 2.5 for a view of `af1ak` in use. Real-time computing is taken very seriously within `af1ak`: any change in the node editor must flawlessly trigger a re-computation and a re-rendering of the visualized output. This sets `af1ak` apart from other visual programming approaches such as the one laid out with KNIME on the previous section. What's more, `af1ak` can export the current state of its node editor and embed it into the end results of the analysis to track provenance and guarantee reproducibility of the conducted study.

2.4 Visualization and provenance in forensic science

In section 1.4, we saw the common points between astronomy and forensic science and an introduction to the development of the *Legal Medicine Mark-up Language*.

Visualization and computer graphics technologies are being more and more utilized in many jurisdictions of late. They are especially used to show injuries or body positions to better understand or demonstrate the succession of events that led to such traumas [Per10]. Though they show convincing results, these are *ad hoc* methods that require someone competent in computer graphics, because of the heavy manual post-processing that is needed. Such methods inevitably call for large-scale cooperation between medical staff and computer engineers [UBS⁺12, BGB13].

Therefore, it is desired for the LMML framework to automatize that cooperation by providing the missing link: a piece of software translating raw autopsy data as input by the medical examiner to computer graphics, which can then be appreciated by untrained personnel, including investigators, prosecutors, lawyers and members of the jury. To that end, these premises require to create a data model to store, describe and arrange forensic data, thus creating an *ontology* for computational forensics. Ontology is what “defines a set of representational primitives with which to model a domain of knowledge or discourse” [Gru09].

Hitherto, visualization technologies have mainly been used to help for the investigation. However, their potential to help in bringing about a judgment during trial has been

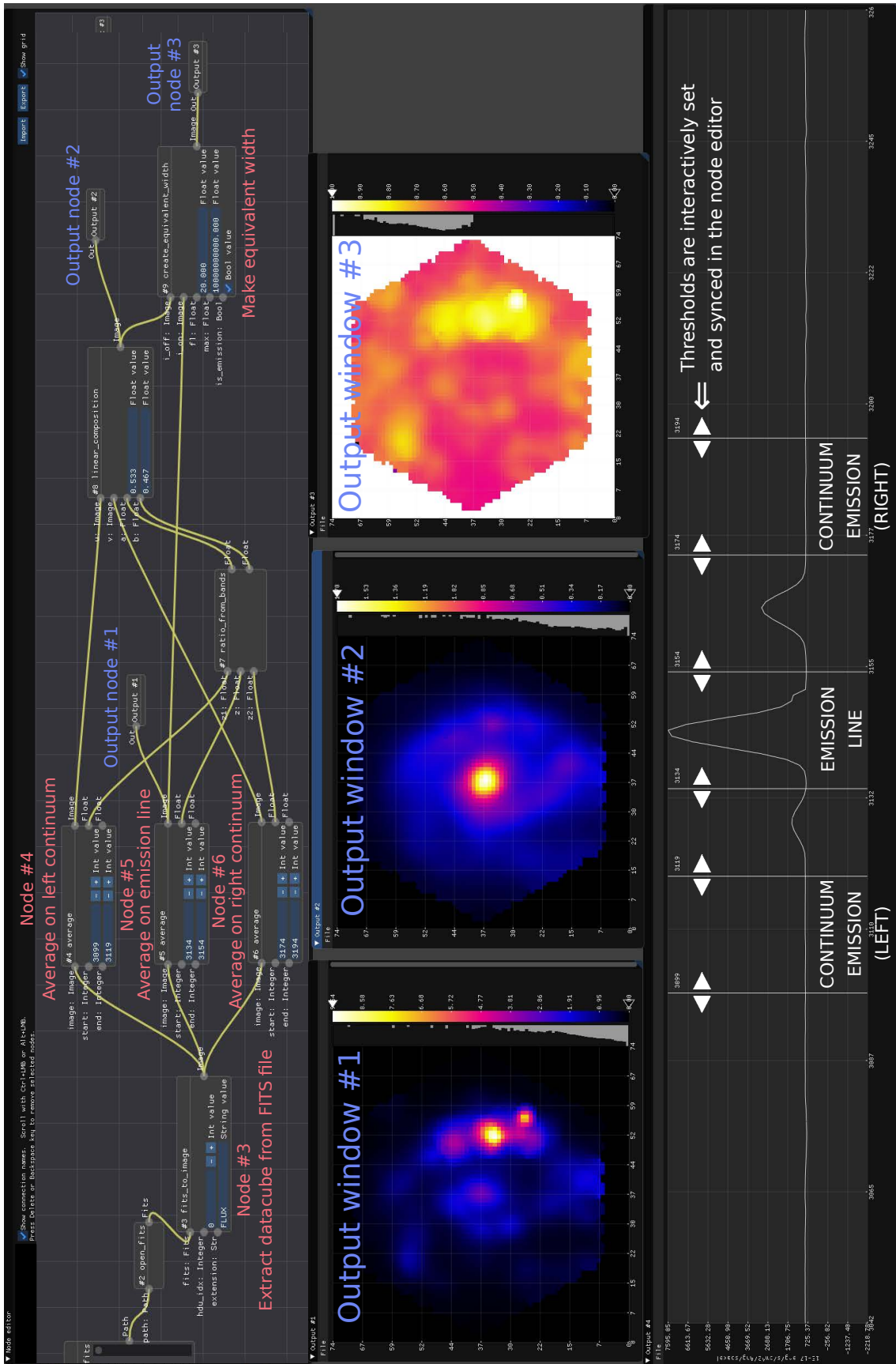


Figure 2.5 Example of a semi-complex use of aFak to extract the equivalent width of a three-dimensional dataset (refer to Carroll's book for precise definition of *equivalent width* [CO07]). The open_fits node opens a FITS file, then several transformations are applied to the file to extract the equivalent width in the right-most output node (Output #3). The result of each output node is visualized in a corresponding output window. Continuum emission is computed by node #4 on the left side, and by node #6 on the right side. The average of the emission line is computed by node #5. Intermediary results are visualized via output nodes #1 and #2.

neglected until now. Thanks to visualization technologies, a trial's stakeholders can more easily understand the case at hand. We believe that such mean would allow to hold members of the jury to be fairer in their judgment—through a well-informed choice—and their choosing of the proportioned sentence.

Moreover, the LMML system includes an interface to input forensic data as the autopsy is being conducted. This is crucial in that, some areas, e.g. Japan, are lacking competent medical examiners, leading medical staff to be overly solicited [OSSK13]. Examination must thus be performed quicker to increase efficiency. In short, the contribution of LMML is three-fold: the design of a semantic language that describes forensic issues, the conception of an interface to create, edit, analyze or query files written in that language and the creation of visualizations usable for all the investigation's stakeholders. Figure 2.6 shows an example of the authoring interface to define a forensic case, while Figure 2.7 shows a resulting output visualization generated from the input content.

Regarding LMML's stake on provenance, provenance management allows to keep track of whatever was done to solve a case. All the trial and error of the investigators that led them to the answer would be recorded as well. Thanks to this, LMML can make possible the reproducibility of an investigation. If a case were to be re-opened, it is not infrequent that a new investigation must be conducted. We could thus check exactly how the past investigation was done and improve the new inspection. Beyond any doubt, provenance management shows the same benefits in the context of astronomy.

Autopsy records (External examination - Head)

Hair

Color: Dyed color:

Length (top) in cm: Length (edges) in cm:

Left external auditory meatus

Haemorrhage:

Right external auditory meatus

Haemorrhage:

2 . Head
Hair is black and dyed brown.
Top hair length is 3 cm, while edge length is 5 cm.

No haemorrhage spotted in left external auditory meatus. No injury in auricle.
No haemorrhage spotted in right external auditory meatus. No injury in auricle.

2-1 Injury
From the left temple, 1 cm in the frontal direction, a pre-mortem slash wound was spotted. Slash wounds with heavy bleeding.




Figure 2.6 LMML Browser: Data input interface (head external examination). Top: Fixed forms for data relevant to head examination. Bottom: Preview of the paragraph dealing with the head included in the final forensic report.

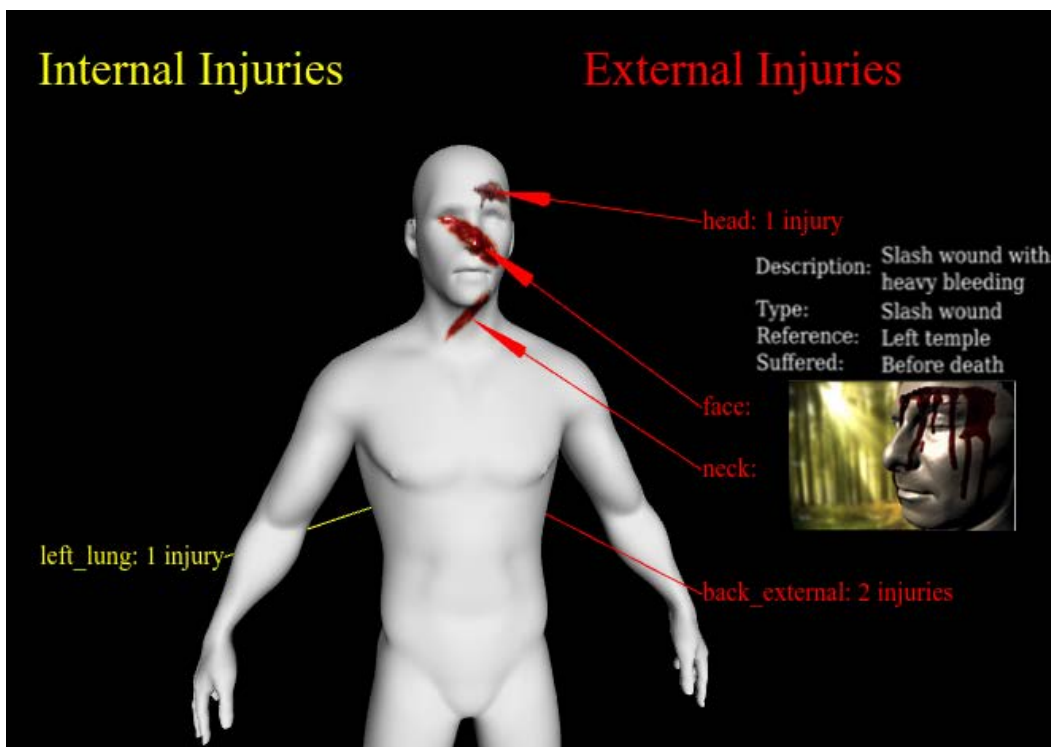


Figure 2.7 LMML Browser: A visualization to explore the result of the autopsy.

Chapter 3

aflak: Advanced Framework for Learning Astronomical Knowledge

“Flow-Based Programming is best understood as a *coordination* language, rather than as a *programming* language. Coordination and modularity are two sides of the same coin, and several years ago Nate Edwards of IBM coined the term “configurable modularity” to denote an ability to re-use independent components just by changing their interconnections [...]. One of the important characteristics of systems exhibiting configurable modularity is that you can build them out of “black box” re-usable modules, much like the chips which are used to build logic in hardware.”

Paul J. Morrison, IBM [Mor94]

IBM pioneered visual programming systems for business applications since the 1970s and 1980s, then, visual programming systems for visualization applications have been implemented from the 1990s, with IBM-developed commercial systems (IBM Data Explorer), as we saw in section 2.2.

3.1 Background and motivation

Data-driven science gathers information from several sources and conducts non-trivial analysis on it. Keeping track of the provenance of the original information and of the process behind a successful or failed analysis is necessary to validate a hypothesis. Astro-

physics is such a domain. Though not domain-specific, general frameworks for provenance management have been developed for scientific workflow [ABJF06, SZP15].

While it was not always the case, lately astronomy journals have been requiring authors to provide the unique identifiers of the original datasets from which they conducted their analyses. All modern sky surveys and data publishers categorize and assign unique IDs to astronomical objects. This ID can for example be cited in academic papers to give credits to the survey that provided the source. Moreover, most astronomical data publishers implement methods to issue a query on objects using various standards. In fact, astronomers maintain a body of standards so that all data publishers agree on the means of transmission, storage and querying of datasets (and other resources). This body of standards is known as the *Virtual Observatory (VO)* and is maintained by the *International Virtual Observatory Alliance (IVOA)* [ivo]. To promote interoperability with existing software, it is both a necessity and an asset for a flask to be compliant with the published VO recommendations.

Going back to IDs, industrious astronomers will keep track of the identifier of the objects they study. Where to find such identifier is defined in the ObsCore protocol in use since 2011 [LTD⁺17] (we will come back to ObsCore later). Then they will do some analysis on the object, and publish their results. During the analysis process, various operations such as transforms or data extractions are run. From then, a final output data is created. The astronomer will then publish the final data along with a description of their analysis. However, how can other scientists be sure that the analysis is reproducible and that the description is accurate? There are limits to peer-review, and there is currently no defined standard in astrophysics to manage provenance. Yet, many astronomers believe defining guidelines for provenance management is a necessity, and some work has started being conducted to create standardized methods to describe provenance with a proposed recommendation for a *Provenance Data Model* [SRB⁺18, GRS⁺18]). Unfortunately, there is no common agreement as of today and the project is yet to be completed.

As a solution to the above problem, the author proposes an flask extension to the FITS standard to record provenance within FITS data files, along with an integration of flask's analysis components with astronomical standards for querying images. Among the various VO standards for querying, the *Simple Image Access Protocol 2.0 (SIAP)* [OLD⁺08] was selected to be supported first, for its relative simplicity and integrability with a node editor interface.

3.2 Design goals

af1ak is designed to meet the following requirements.

3.2.1 Ease of use and responsiveness

af1ak is designed with ease of use in mind. Though using af1ak requires astronomy-related domain-specific knowledge, it is designed to be intuitive. Dataflow is clearly indicated by connections between box-shaped nodes. The interface is responsive: the output of a visual program is refreshed in real time as the program is being updated, with very minor delay, so that the user not be confused about the provenance of the data being shown. Moreover, errors are shown using a stack trace to show at which exact node an error occurred.

3.2.2 Re-usability and extendability

af1ak can export and import the state of its interface. Moreover, af1ak allows for the import and export of shareable composite nodes, referred to as “macro” hereafter. Macros can be shared among other users via a cloud platform or file exchange. What’s more, nodes may be implemented as side-loaded shared libraries, allowing to create user-defined nodes in a language that supports C/C++ calling conventions.

3.2.3 Collaborative development

No research is done alone anymore. af1ak aims at making collaborative development and code-sharing as easy as possible.

3.3 System overview

3.3.1 Description

af1ak’s programming interface is composed of a node editor, where nodes can be freely composed by linking a node’s output slot to another node’s input slot with the mouse cursor [BMT⁺19, BTU⁺19]. The node editor’s left pane contains the node list and the documentation of the currently selected node. By selecting a node in the node list, the view will directly jump and center onto that specific node in the node graph. The node graph basically reads from left (input) to right (output), in accordance with the left-to-right convention of most writing systems. The node editor itself is a representation of dataflow.

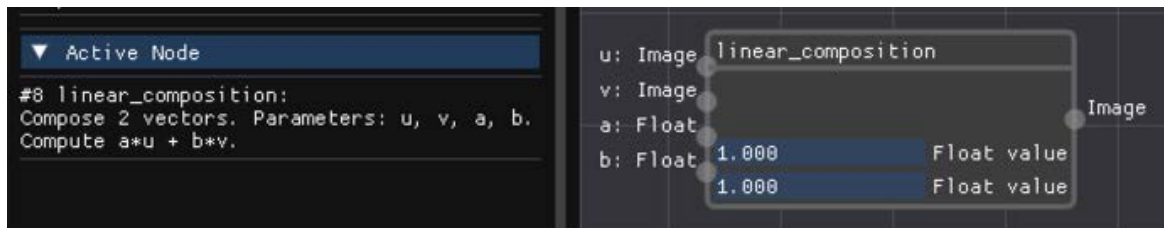


Figure 3.1 An example of transformation node, which computes the linear composition of two images. It has four inputs (on the left): two images u , v and two scalar parameters a and b . The transformation node has a single output slot (on the right) from which the image $au + bv$ comes out. Parameters a and b can either be directly input by the user inside the node, or can be taken from the value of an output of another node.

There are three different types of nodes:

Transformation nodes: Composable transformation module. Contain a specific amount of input slots and output slots, each with a specific expected data type. For an example of a transformation node, refer to Figure 3.1.

Value nodes: Contain a parameter of a certain data type that can be input by the user, or that can be externally set by another tool. Some may consider that a value node is a special case of a transformation node without any input slot. However, in aFlak’s context, they are semantically different in that they are not modular “black boxes,” as the value in a value node can be externally set or edited using the user interface. A list of supported types for value nodes can be seen in Figure 3.8.


Output nodes: Final output of the flowing data. Any data type can be redirected to an output node. When a new output node is created, a new *output window* containing a visualization of the data arriving to this node is opened. The output window and the output node share the same number. Such output windows can be seen in Figure 2.5.

A node may as well be the result of the composition of several nodes. This feature is referred to as *macro*. This will be explained further in details in section 4.2.1.






3.3.2 Detailed description of components

Figure 3.2 shows aFlak’s interface. The interface has several sub-windows, all of which can be freely moved, re-sized or minimized (by double-clicking on the window’s header). On the first run, aFlak will infer a workable window layout from the current screen resolution. On subsequent run, the last used layout—which is user-defined—is re-used. Indeed, the layout of sub-windows is saved every time aFlak is closed.

Node editor window

The main sub-window is the node editor, on top of Figure 3.2. The left side of the node editor contains a resizable and minimizable left pane, which holds two dropdowns: a dropdown containing the list of nodes in the node editor and a dropdown containing miscellaneous information about the *active* node. The *active* node is defined as the latest selected node (several nodes can be selected at the same time by holding the “Shift” button ). In the node list, each transformation and value node is represented by the string “#<N> <node name>,” where N is the ID that identifies the node in the node editor. All nodes are assigned a unique ID. Each output node will be represented as “Output #<N>.” Besides, on clicking on an item in the node list, the node editor will jump and focus on the clicked node corresponding to the clicked item. This is convenient to find nodes that are outside of the node editor view. An enlarged view of the left pane can be seen in Figure 3.3.

The “Active node” dropdown contains information about the current active node. First, we can find the node ID, then the node name. Then a short explanation about what the transformation the node performs. After this follows a detailed explanation about the expected values as input(s) and output(s) for the node. Finally, some metadata is displayed, including the author of the node, the date of creation and its version.

The node editor itself consists of a visual directed acyclic node graph. The user can pan the node editor by holding the left control key  while dragging the mouse on the background of the node editor. Adding a new node—from value node to output node, including transformation node—can be done from the “Add node” pop-up dialog as shown on Figure 3.4, which can be opened by right-clicking anywhere on the background. Nodes can be wired together by dragging the mouse from an output slot (represented by a circle on the left of a node) to an input slot (represented by a circle on the right of a node), and vice-versa. Nodes can be deleted by selecting them with the mouse and pressing  key ( +  on macOS), or by  right-clicking and choosing “Delete” in the sub-menu. In addition, a node is added on the nearest available empty space around the point the cursor of the mouse is located.

Apart from the node graph, the node editor contains both export and import buttons, which allows to save or load the current state of the editor. It contains as well a few ergonomic options. For example showing connection names can be disabled to reduce clutter when numerous nodes are displayed at the same time, allowing the user to only concentrate on visualizing the dataflow.

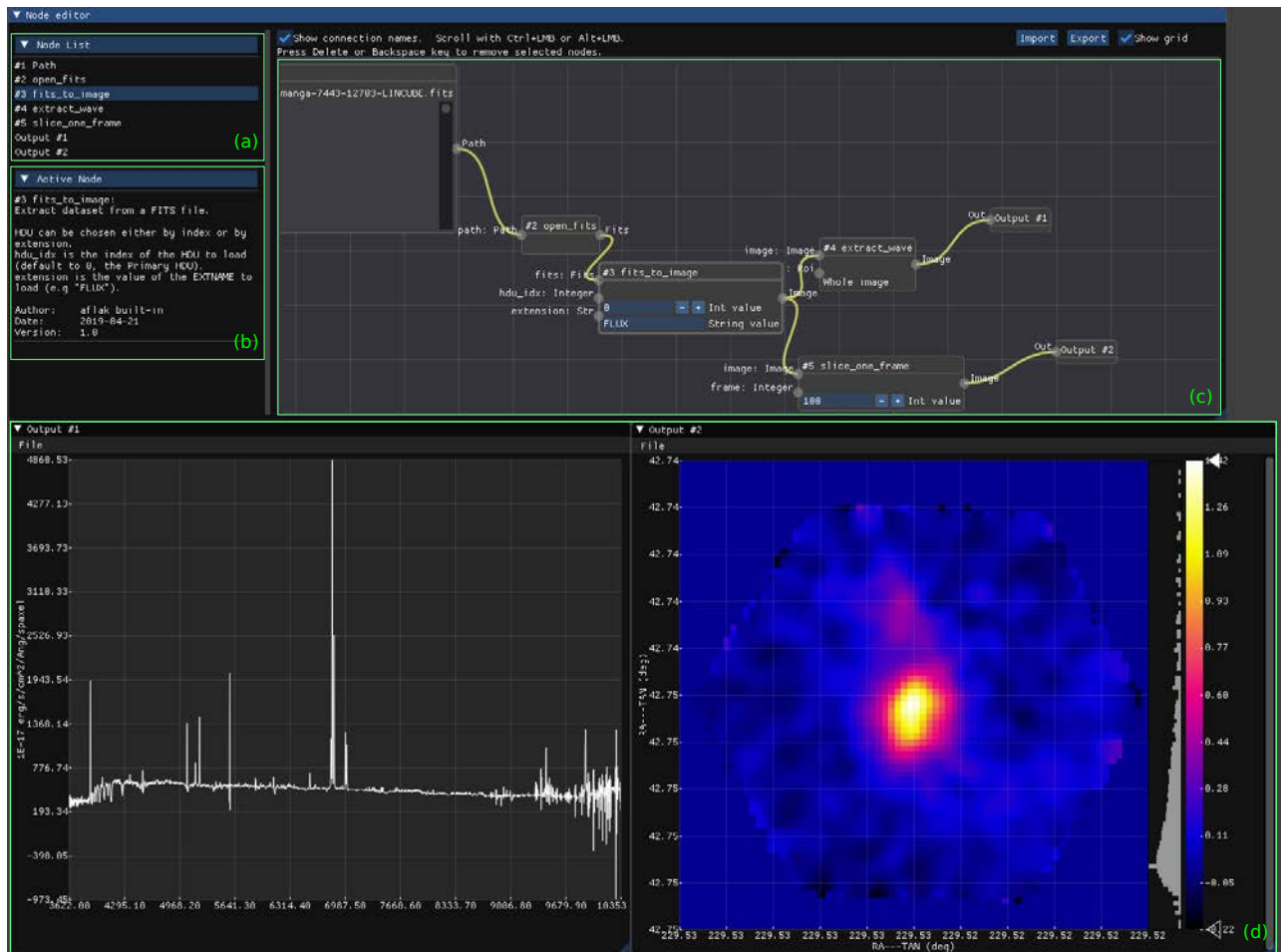


Figure 3.2 General UI of af1ak. We can see all of af1ak's components.

- (a) Current node list
- (b) Documentation of last selected node (*active node*)
- (c) Node editor, with output nodes on the left
- (d) Output windows for visualization

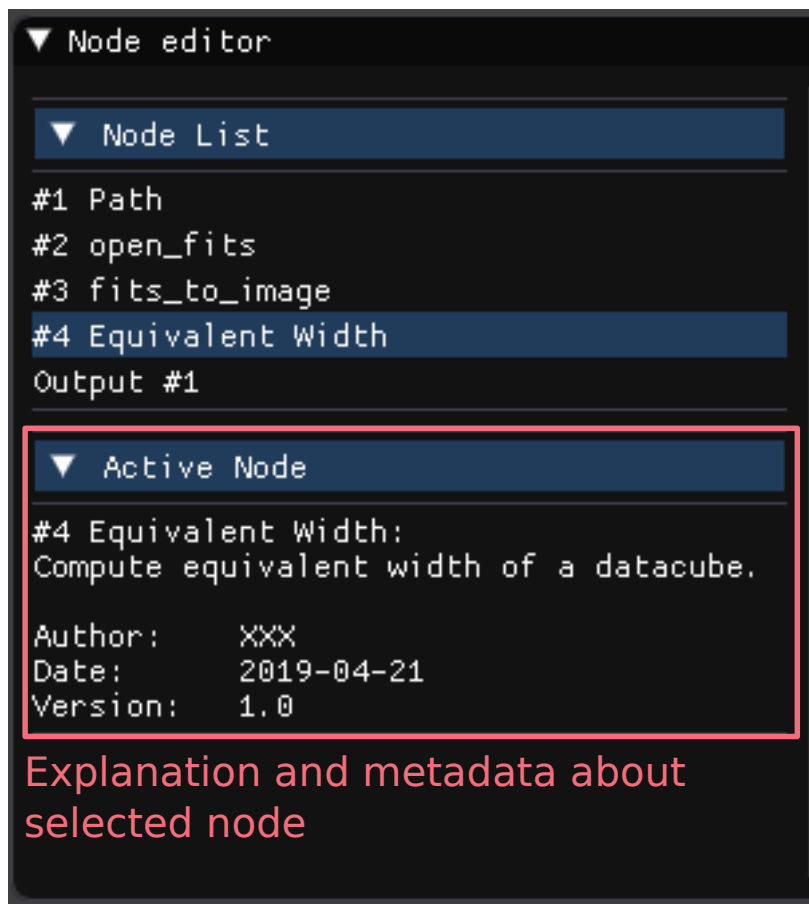


Figure 3.3 Left pane of aflak's interface, containing the list of nodes in the current editor's window.

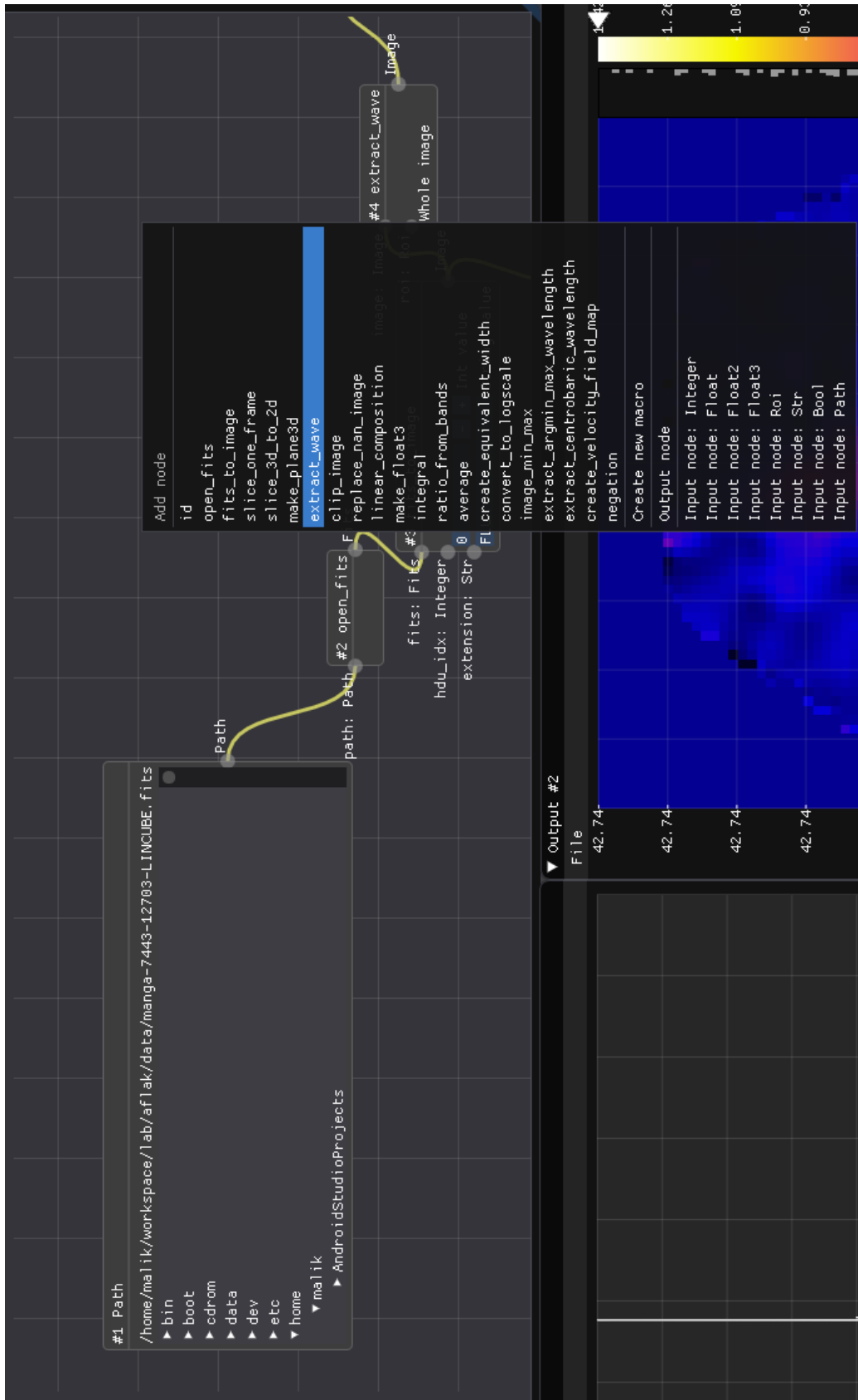


Figure 3.4 The figure shows the pop-up dialog that allows to select a new node to add to the node editor's graph. The pop-up dialog can be opened by right-clicking on any empty space in the node editor.

Output windows

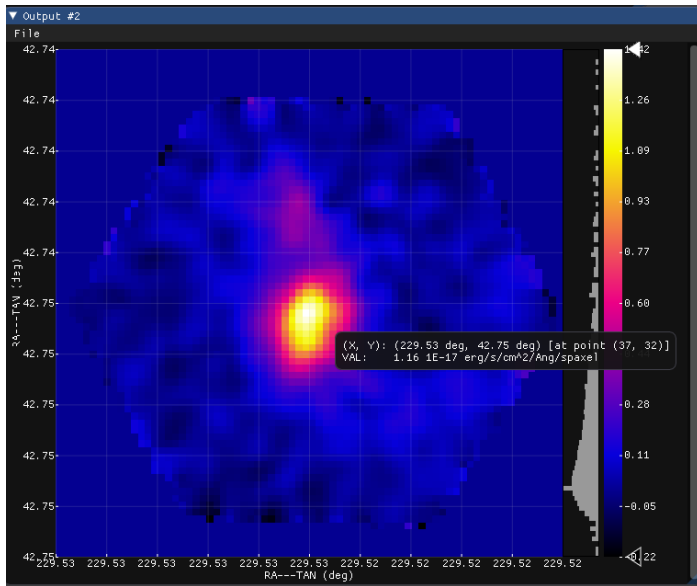
Any type can be redirected to an output node in the node editor. Each output node is assigned an ID (starting from 1). For each output node a corresponding output window is opened to visualize the data flowing into the output node. Depending on the data type that goes into the output node, a different interface will be used for the output window. Figure 3.5 shows examples of output windows for visualizing a two-dimensional image. In any case, each output window has a save feature, inside the menu “File,” then “Save.” Using the save feature, the content of the output window will be saved to a FITS or text file. Only multi-dimensional images are saved into FITS files. Other types will use text files.

The design of the output window is partly inspired from PyQtGraph, as was revealed in section 2.1.2. On the right, a color bar shows the mapping from value to colors, along with a histogram showing the distribution of values. The histogram used logarithmic coordinates, as in astronomy, there is usually a huge difference (several orders of magnitude) of luminosity (photon count) between empty areas, a planet or galactic periphery, and a bright star or galaxy core. Several color maps are available. They can be selected by right-clicking on the color bar. The default color map is the “Flame” color map shown in Figures 3.5a and 3.5b. Figure 3.5c shows an example of the “GreyClip” color map, which is very convenient to singularize points above or below a specific threshold. The user may as well drag the top arrows and bottoms on the right of the color bar, to change the contrast and selectively decide what are the more appropriate objects that should be visualized. An example of change in contrast is shown in Figure 3.5d.

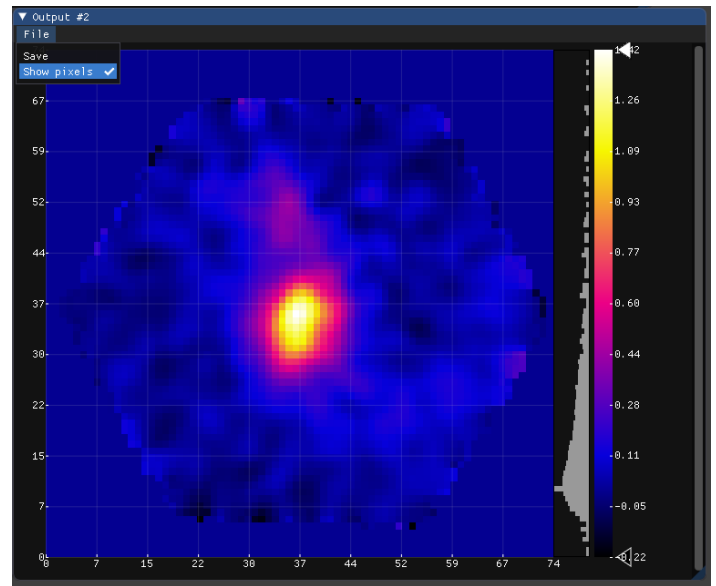
Besides, the “two-dimensional image” view of an output window contains a feature to select a region of interest, as shown in Figure 3.6b, and a feature to select specific values on the x - or y -axis (in Figure 3.6a).

Finally, Figure 3.7 shows the view of an output window that renders a one-dimensional image (a waveform) as a curve. The user can zoom in and out with the wheel of the mouse, and select a value on the x -axis.

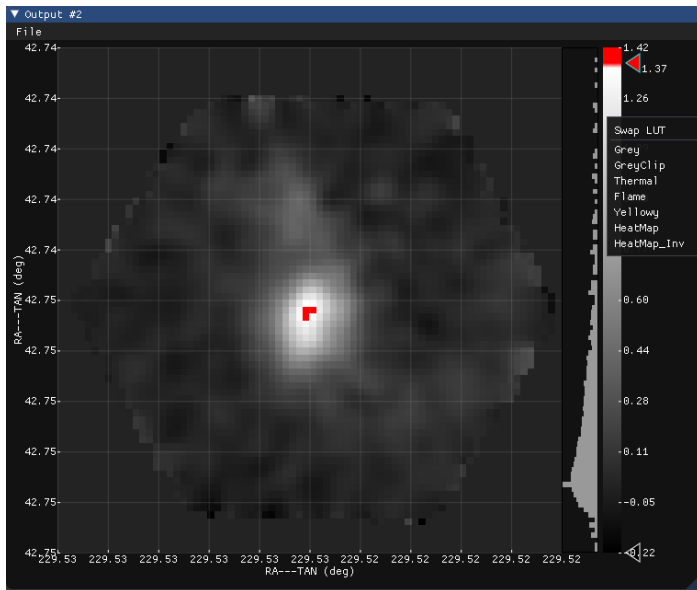
It should be as well noted that apart from one- and two-dimensional images, output windows in current version of `af1ak` can visualize simple types such as boolean values, integers, floating point numbers, strings of characters, but also more complex types such as astronomical tables or FITS files (the content of the FITS header would then be shown).



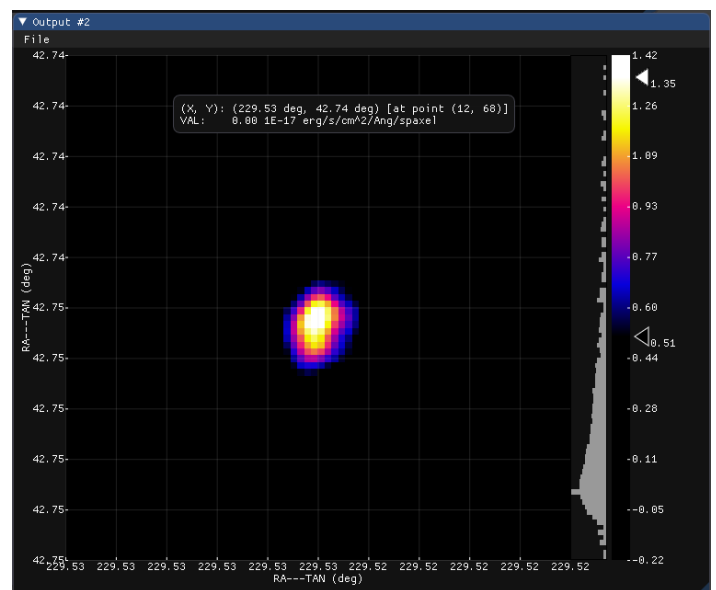
(a) A two-dimensional image visualized with sky coordinates used as axis.



(b) A two-dimensional image visualized with pixel coordinates used as axes. Pixel coordinates can be selected by toggling the menu “File,” then “Show pixels.”

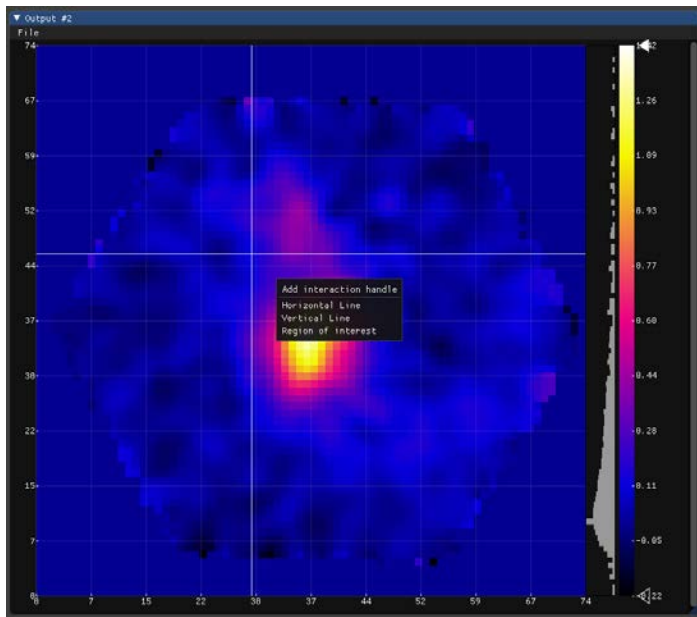


(c) The color Look-Up Table (LUT) can be changed from the “Swap LUT” menu.

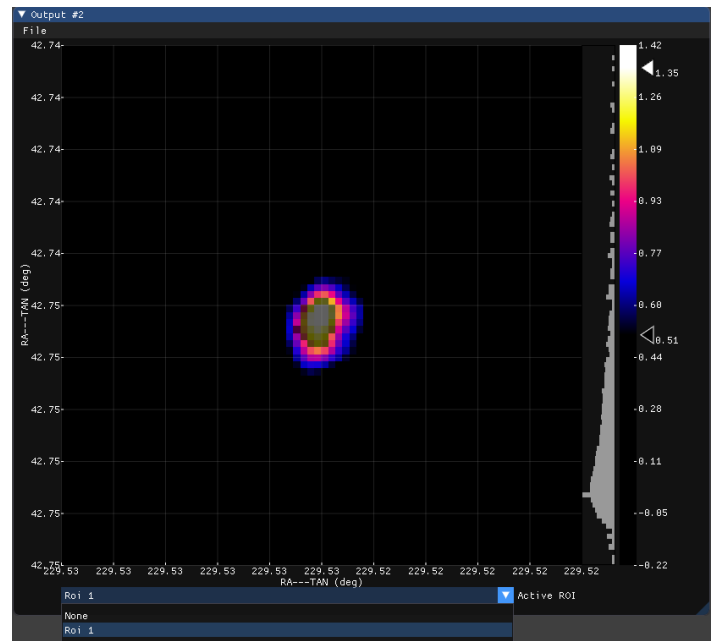


(d) The arrows on the right of the color map can be dragged to change contrast. On this image, the background is made darker to discern more clearly the galactic core.

Figure 3.5 Interfaces of an output window showing a 2D image with several color maps and axes.



(a) This sub-figure demonstrates the feature to select an axis (horizontal or vertical) value of interest on a two-dimensional image.



(b) This sub-figure demonstrates the feature to select region of interest on a two-dimensional image. Several regions of interest can be drawn on the same image. The “active” region of interest is highlighted on the image and is the one which is actually being drawn when the mouse is clicked. The user can change the active region of interest at any time via the dropdown menu on the bottom.

Figure 3.6 Examples of interaction handles representing a bound value in a visualization of a two-dimensional image.

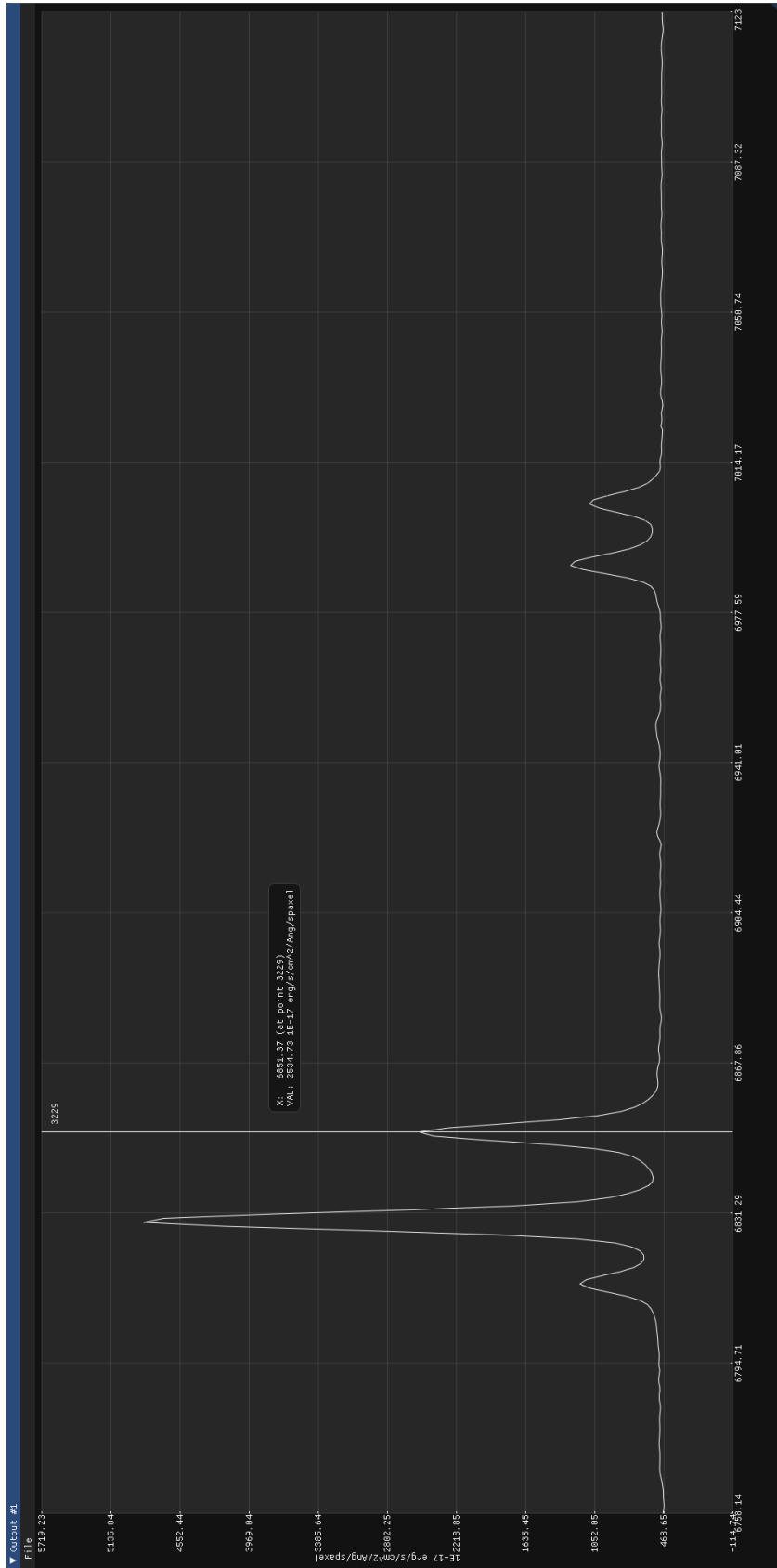


Figure 3.7 Visualization of the waveform of a datacube. This is the interface of a one-dimensional image when displayed in an output window. The vertical line is an interactive handle that represents an x -axis value selected by the user.

3.3.3 Value nodes, type checking and error handling

Value nodes

A value node could be considered and is in fact the same as a transformation node without input. However, there is a non-negligible conceptual difference. Value nodes are not black boxes. Their values can be set from the interface. Please refer to Figure 3.8 to see how the value nodes look like on `afLak`'s interface. For example, the value node of type `Path` contains a local path referring to a file on the user's file system. It is natural that the value node of a variable of type `Path` allows the user to select a file by exposing a file-explorer-like interface.

A value node of type `Integer` allows the user to input an integer, which will become the value encased in the node. It is moreover suited for an integer input interface to provide decrement (-) and increment (+) buttons.

While a `Float` value node simply contains a float that can be input by the user. The same can be said for `Str` (for "string") and `Bool` (for "boolean").

It should be noted that all values that can be selected and manipulated inside output windows as shown in Figures 3.7 and 3.6b can be redirected and bound to a value node and be re-used within the node editor. The binding is *bi-directional*: editing the value of the node from the node interface will update the representation of the value in the output window as well. This allows the user have fine-grained visual control on several parameters while designing an analytics pipeline.

Type checking

`afLak` is type-checked when the node editor is built. An output slot can only be attached to an input slot with the same type. If the user wrongfully attempts to join two incompatible slots together, the action will be aborted and an explanatory message error similar to the one shown in Figure 3.9 will pop up on the screen. However, for convenience, some restricted type conversions are also possible. For example, integer can be converted to float (trivially) and float can be converted back to integer by rounding. This improves the user experience as they may wire an output slot of type `Integer` to an input slot of type `Float`.

Error handling

There are two types of errors: graph errors and runtime errors. Graph errors are handled and detected before they actually cause issues. These may be type errors or doing something that creates an uncomputable graphs, as shown in Figure 3.9.

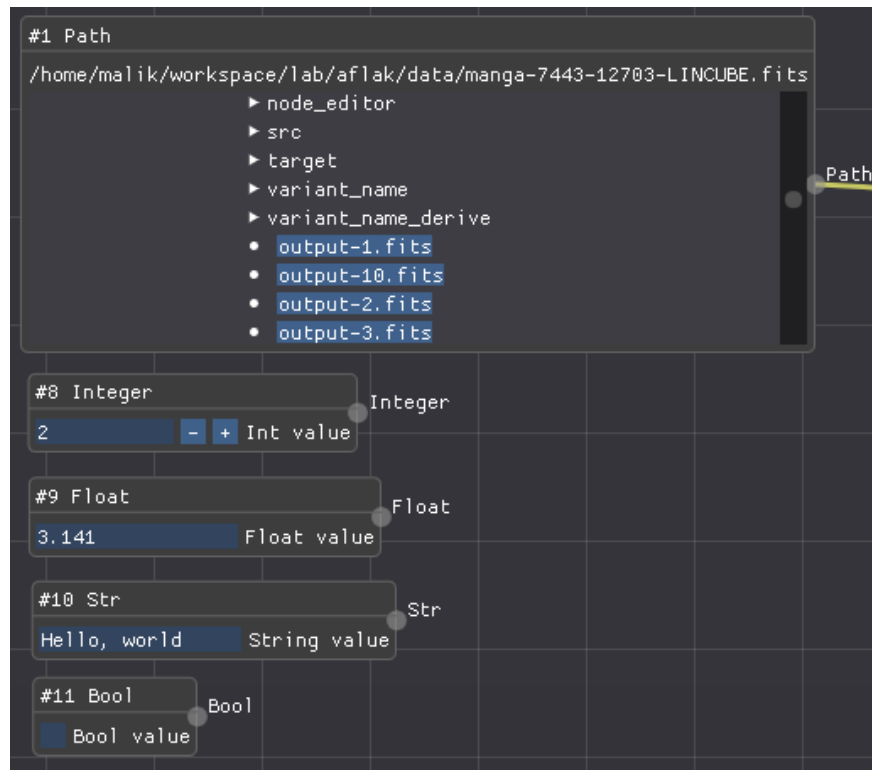


Figure 3.8 Many types of value nodes as they appear in aflak.

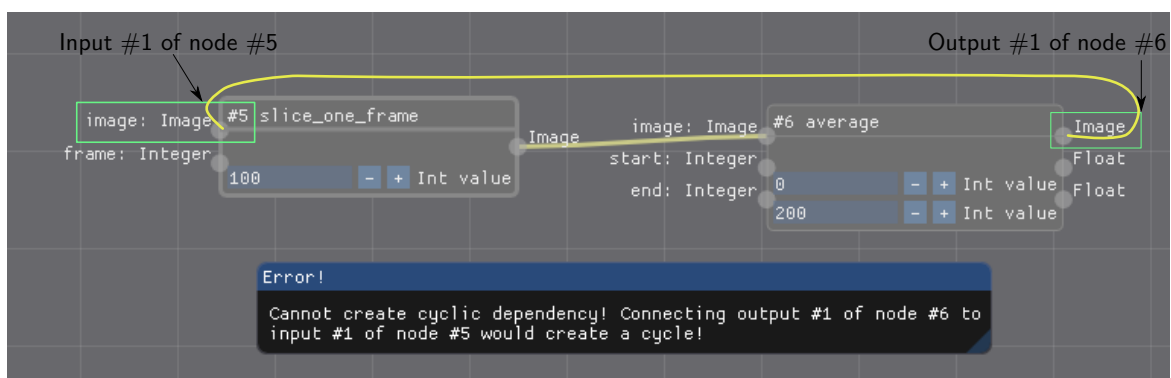


Figure 3.9 aflak will prevent the user from wasting their CPU resources. No cyclic dependencies can be created. The same kind of errors will be detected, prevented and a message will be displayed if a user tries to nest recursive macros.

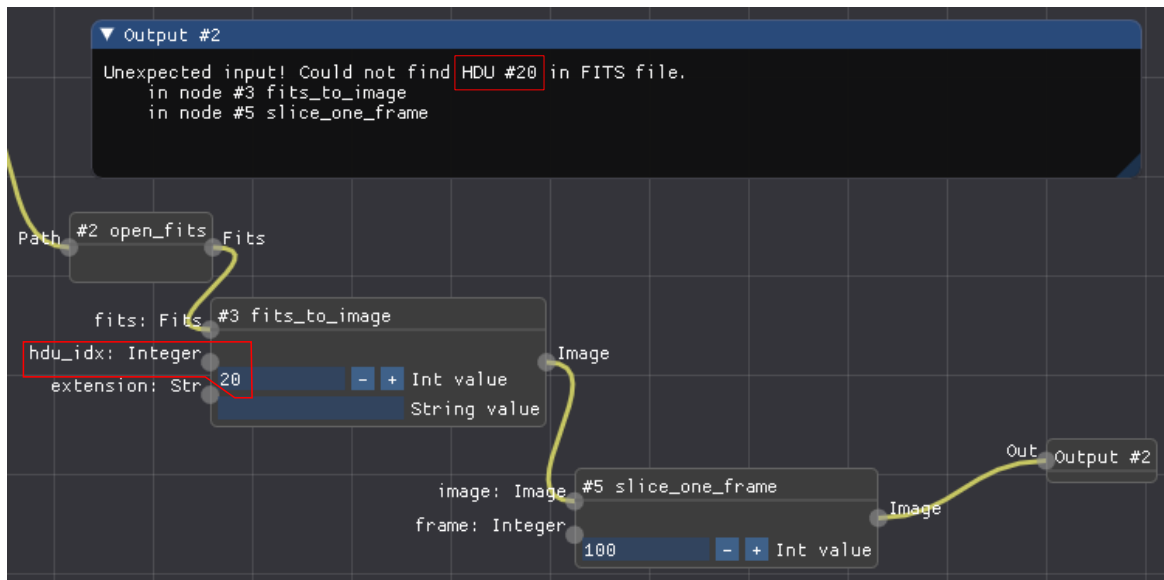


Figure 3.10 An output window shows a runtime error. The `fits_to_image` node cannot find a FITS HDU with index 20. So the error bubbles up to Output #2, and its stack-trace is displayed in the output window for easy debugging.

Runtime errors, on the other hand, are not detected when the graphical program is created. The output value of a node returning an error will bubble up, following the dataflow, until an output node. Then the linked output window will print a stack-trace explaining what kind of error occurred. Such a runtime error is shown in Figure 3.10.

Chapter 4

Implementation Details

“The devil is in the detail.”

Common English idiom

4.1 Description of algorithms and implementation

4.1.1 Language and library choices

In order to achieve high speed and responsiveness, care was taken to design efficient algorithms and data structures. The author first started developing a prototype of `af1ak` in Python, based on `PyQtGraph`. However, he quickly faced the limitation of Python. As `PyQtGraph` is still a bunch of classes around `PyQt`, itself a wrapper of the C++ library `Qt`, writing Python code on top of `PyQt` was required. The result was slow and did not meet our standards of responsiveness. At first, some study was done to go closer to the metal by directly writing C++ code in `Qt`. The author could have proceeded into that direction. If not for the idea to borrow technologies from the gaming industry. Indeed, few software can boast to be as responsive as games. Games are engineered from the ground up to be responsive. This is when he decided not to use `Qt`, a graphical library suited for graphical desktop applications (such as text editors, calculators, etc.), but to use an immediate-mode graphical user interface. Our choice went with the C++ library “Dear ImGui”¹. Then regarding the language, it was decided to use Rust.

¹Dear ImGui is an open-source library to make portable user interface using a drawing back-end such as OpenGL. It is free, open-source and available at <https://github.com/ocornut/imgui>. The author of this thesis is a maintainer of the Rust bindings for Dear ImGui, a crate called `imgui-rs` hosted at <https://github.com/Gekkio/imgui-rs>.

`af1ak` is indeed written in Rust, which allows C/C++-like fine-grained control on memory layout to maximize performance, while providing a higher-level syntax and a memory-safe paradigm for writing highly concurrent programs, boosting the productivity of the implementer. Given the highly parallel code structure of `af1ak` (the node editor’s program is evaluated using several threads on several CPU cores, in addition to the single-threaded UI loop), Rust was a natural choice.

4.1.2 Multi-crate structure

`af1ak` boasts a modular structure separated into several crates². Its architecture is presented in Figure 4.1. It is completely modular. All components are theoretically independent and can be re-used by another independent piece of software.

The upper layer consists of an node editor (whose crate is called `node_editor`) engine and a plotting library (`af1ak_plot`) to visualize the output data. The plotting library was written from scratch tailored for our needs, using simple OpenGL calls for rendering. All the content of the output windows showing image data are basically rendered with the plotting library.

The node editor engine has a compute back-end, which was named `cake`, that manages pending computational tasks in a multi-threaded manner. `cake` itself relies on a defined set of elementary primitives—you could call that a “standard library” for our visualization domain—used to define all the transforms usable in the node graph. Primitives and data types are completely interchangeable. However, for the purpose of this thesis, only data types and primitives adapted to astrophysical data were developed—for example, the primitive module to open FITS files and load its content from the file system is only relevant in the astronomy community.

All of computational tasks run by `af1ak` are managed in an independent sub-crate called `af1ak_cake` (a.k.a. `cake`, standing for Computational `mAKE`). First the basic structure of `cake`, excluding macro support, will be presented. Then macro implementation will be explained in detail, while highlighting the choice of data structures.

4.1.3 `cake`: Computation `mAKE`

`cake`’s main data structure is called “*DST*” (for Dynamic Syntax Tree). It is defined as shown in Table 4.1. What must be kept in mind is that a *DST* represents a directed acyclic graph of nodes. To represent such graph, we must keep a collection of transforma-

²A crate refers to a importable module, in Rust’s terms. This is what Ruby calls a gem, Python calls a package, etc.

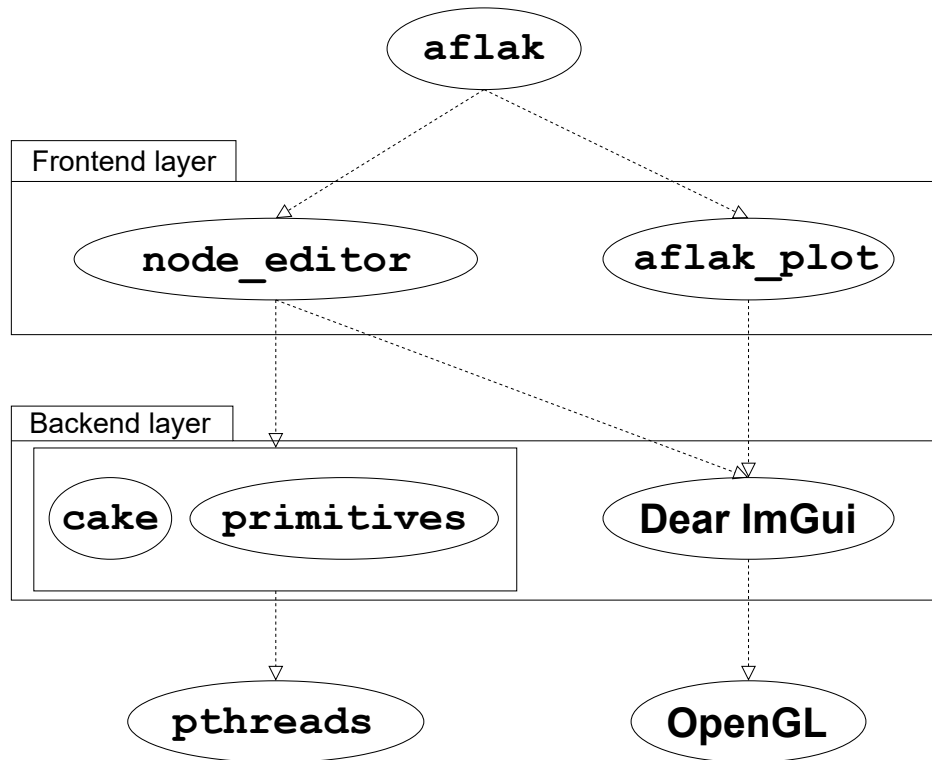


Figure 4.1 aflak’s modular structure.

tion nodes (field `transforms` in Table 4.1). To do that, we assign a unique identifier to all nodes in the node editor. When a new node is added, an unused identifier—for a transformation node, an instance of `TransformId` is used as identifier—is assigned to the new node. The actual node whose inner data is contained in a `MetaTransform` data structure is then inserted into the sorted map containing all transforms.

But keeping the collection of transformation nodes is not enough: edges in the directed acyclic graph must as well be kept in memory. This is done in the `edges` field in Table 4.1. For aflak’s use case, the data structure `Output` represents the n -th output slot of the node indexed by `TransformId` as the tuple $(TransformId, n)$. Similarly, the data structure `Input` represents the n -th input slot of the node indexed by `TransformId` as the tuple $(TransformId, n)$. The collection of edges of a graph can then be defined as a map from an output to a list of inputs, as shown in Table 4.1. Indeed, a single output slot can be attached to an indefinite number of input slots (one-to-many relationship). The other way around (single input slot attached to many output slot) does not make any sense as data can only flow from an output slot to (an)other node(s)’s input slot(s).

Finally, the collection of final outputs, which are defined as the actual values that are computed out of the graph, is stored in as a map in the field `outputs` in Table 4.1. Each

Table 4.1 Dynamic Syntax Tree data structure

<i>DST</i>	
transforms	<i>Map</i> ⟨ <i>TransformId</i> , <i>MetaTransform</i> ⟩
edges	<i>Map</i> ⟨ <i>Output</i> , <i>List</i> ⟨ <i>Input</i> ⟩⟩
outputs	<i>Map</i> ⟨ <i>OutputId</i> , <i>Option</i> ⟨ <i>Output</i> ⟩⟩
<i>Input</i>	
transform_id	<i>TransformId</i>
input_index	Unsigned Integer
<i>Output</i>	
transform_id	<i>TransformId</i>
output_index	Unsigned Integer

output is assigned an identifier (an *OutputId*). Each *OutputId* is associated through a map with nothing—the final output is an orphan and not attached to any node’s output slot—, or with an output slot—represented with the data structure *Output* defined above. The concept “an output slot or nothing” is represented as the data structure *Option*⟨*Output*⟩.

When a *DST* instance is constructed and every time a new node is added, the graph is checked for consistency. For example, no circular path can ever be created, `cake` will not allow that and gracefully abort the addition of a new node. `af1ak` will then print the error message to the screen to let the user know what error was successfully avoided.

4.1.4 *MetaTransform* data structure

As referenced in Table 4.1, the content of a node is stored as a *MetaTransform* data structure. A *MetaTransform* is a *Transform* to which metadata is added. The *Transform* contains the actual data describing how the computation for this node should be processed. *MetaTransform*’s metadata contains the `input_defaults` field, which is a list of optional values that represents the user-editable default input values for the node. For example, for the node in Figure 3.1, `input_defaults`’s value would be:

[*None*, *None*, *Some*(1.0), *Some*(1.0)]

(first and second input slots have no default values, third and fourth input slots have 1.0 as default value).

Whenever the node is updated (e.g. by updating a default input value), the `updated_on` field is updated to contain the instant³ on which the node was updated. As we will see later in section 4.1.5 on cache, storing the instant at which the node is updated is crucial so that cache be functional.

³ In `af1ak`’s current implementation, an instant is not really a timestamp, but a measurement of a monotonically non-decreasing clock.

Table 4.2 MetaTransform data structure

<i>MetaTransform</i>	
<i>t</i>	<i>Transform</i>
<i>input_defaults</i>	<i>List<Option<T>></i> (where <i>T</i> is the type of the values passed around when a graph is computed)
<i>updated_on</i>	<i>Instant</i>

Table 4.3 Transform data structure

<i>Transform</i>	
<i>algorithm</i>	<i>Algorithm</i>
<i>updated_on</i>	<i>Instant</i>

Transform's data structure tabulated in Table 4.3 is just a tuple of an *Algorithm* and the instant on which the *Algorithm* data structure was updated. *Algorithm*, as shown in Table 4.4, is an enumeration that can be one of the three following variants:

Function: A pointer to a pure function that takes an array of data of type *T* and returns an array of data of type either *T* or *E*, where *T* is the data type on which computations are done (an enumeration over multi-spectral data, images, FITS files, etc. in `af1ak`'s context) and *E* is an error type, that can be represented as a human-readable string for ease of debugging.

Constant: A constant of type *T*. This actually represents a constant node. Such a node has no input slot and has a single output slot that always returns *T*'s value.

Macro: Contains a handle to a macro *MacroHandle* (see section 4.2.1).

All in all, we now have explained all the data structures used to store a node graph. We will then see how the computing is actually running.

Table 4.4 Data structure of Algorithm enumeration

<i>Algorithm</i> 's variant list	
Function	Pure function pointer with some meta-data (name, description, version number, default values, input and output types etc.)
Constant	<i>T</i>
Macro	<i>MacroHandle</i> (see section 4.2.1)

4.1.5 Computing output with cache

Computation model

Given a *DST* representing a node graph, we iterate over each final output identified by an *OutputId*. For each final output attached to a node's output slot, the value at this output slot is computed and then returned.

A value at an output slot is computed as follows. Let N a node with n input slots and m output slots. For i and j two non-zero integers, I_i^N is defined as N 's i -th input slot, and O_j^N is defined N 's j -th output slot, The value at the i -th output slot of node N is noted as o_i^N , while value at the j -th output slot of node N is noted as o_j^N .

Let f the function so that $f(i_1^N, \dots, i_n^N) = (o_1^N, \dots, o_m^N)$, i.e. f is equivalent to the function that is evaluated when data (i_1^N, \dots, i_n^N) enters node N .

Let us say we want to compute one of the o_j^N for $1 \leq j \leq m$. For each i such as $1 \leq i \leq n$, we have either one of the three following cases:

- I_i^N is not attached to any output slot and has no default value. In that case, o_j^N is uncomputable and an error indicating that a dependency is missing is raised and propagated.
- I_i^N has an associated default value x and is not attached to any output slot. Then $i_i^N \leftarrow x$.
- I_i^N is attached to an output slot, say $O_{j'}^D$ (designating j' -th output of node D , where D is a dependency of N attached to I_i^N). Then $i_i^N \leftarrow o_{j'}^D$.

With the above premises, we can then assuredly compute $f(i_1^N, \dots, i_n^N)$, and thus we have o_j^N . The values of dependencies such that $o_{j'}^D$ can be recursively computed. The recursion will terminate, as the graph has no cycle and is finite, so there exists a starting node N_{start} such as no of N_{start} 's input slots are attached to another node's output slot.

Adding cache to the model

Recursively recomputing the value at an output slot from scratch every time the value must be displayed would take too much time and be extremely inefficient. For a flask to be responsive, it was necessary to implement caching. The approach used is to keep track of the instants when values at output slots are first computable.

Let O_j^N the j -th output slot of node N . flask appends the smallest instant in time t_{ON} when the output slot's is theoretically computable to the value o_j^N in the current node graph's state.

We define an `updated_on` function that computes t_{0N} for each node. It is recursively defined as:

$$\text{updated_on}(N) = \max \begin{pmatrix} N.\text{updated_on} \\ N.t.\text{updated_on} \\ \text{updated_on}(D_1) \\ \vdots \\ \text{updated_on}(D_i) \\ \vdots \\ \text{updated_on}(D_n) \end{pmatrix} \quad (4.1)$$

where N is a node represented as an instance of the datatype *MetaTransform* (see Table 4.2) and D_i for $1 \leq i \leq n$ the n dependencies of N (n may be 0 if N has no dependency, in that case the recursion ends).

By definition, for a node N , `updated_on(N)` is the smallest instant in time when the outputs of N are computable in the current state of the full node graph. We then compute all the o_j^N and append `updated_on(N)` to the value o_j^N , storing them both into memory. Considering that we are now at time t_0 , let $t_{00N} = \text{updated_on}(N)$. Next time o_j^N needs to be computed again say, at time t_1 , if $t_{10N} = t_{00N}$ then there was no change in the graph state that would cause o_j^N to change, so we just retrieve the cached value. If on the contrary $t_{10N} > t_{00N}$, then o_j^N will be recomputed and the new value added to the cache alongside the new t_{10N} . The case $t_{10N} < t_{00N}$ cannot occur.

In addition, the cache data structure needs to support multi-threading, as all `af1ak`'s computing is done on several threads as a background process. This is `af1ak` leverages a concurrent hash map implementation based on bucket-level multi-reader locks⁴.

4.2 Macro support for cake

4.2.1 Design decisions

The representation of a macro is a chunk of memory that has to be shared between several threads. `af1ak`'s runtime is composed of a UI thread that renders the user interface and handles the user's inputs, and several computation threads that compute the node editor's outputs. As a result, the UI thread requires read access to display the macro on the node screen, but requires as well write access to update the macro on user input. The compu-

⁴ Implementation of the concurrent hash map used for cache lies in the `chashmap` crate: <https://docs.rs/chashmap/2.2.2/chashmap/>

Table 4.5 Macro data structure

<i>Macro</i>	
id	<i>Uuid</i>
name	<i>String</i>
dst	<i>DST</i>
inputs	<i>List</i> \langle <i>MacroInput</i> \rangle
updated_on	<i>Instant</i>

<i>MacroInput</i>	
name	<i>String</i>
slot	<i>Input</i>
type_id	<i>TypeId</i> (where <i>TypeId</i> is a data type whose instance identifies the expected type of the variable that flows into the input slot)
default	<i>Option</i> \langle <i>T</i> \rangle

tation thread requires read access to retrieve the macro's DST and run computation. As a result, it is clear that a macro must be shared between threads, behind a read-write lock. We call *MacroHandle* (which showed up in Table 4.4), an atomically reference-counted shared pointer to a read-write lock to the actual chunk of memory of a *Macro*, whose data structure we will define in the next section.

4.2.2 Data structures for macro support

A macro is a data structure defined as shown in Table 4.5. It includes a UUID version 4 (randomly generated Universally Unique Identifier)[LMS05], which is unique identifier that identifies a macro more specifically than its name. The UUID is generated on creation of a new macro according to the specification to guarantee its uniqueness, even after the macro is *shared* among users. Next, a macro contains a human-readable name (used for display) and an embedded DST that describes the behavior of the macro. In addition, while the outputs of a macro are determined by the output nodes of the embedded DST, it is necessary to keep the list of inputs for the macro. This list of inputs is represented by the `inputs` field of type *List* \langle *MacroInput* \rangle . In the current implementation, the list of inputs is inferred and recomputed every time the macro's inner DST is updated: the inner DST's unattached input slots are considered to be the whole macro's input slots. Finally, as for all previously defined data structures, an `updated_on` *Instant* is appended.

4.2.3 Some changes in computation logic

The algorithm used for the whole graph is the same as the one explained in section 4.1.5, only the approach to computing the macro node is different. First, for a macro node N the `updated_on` function is defined as:

$$\text{updated_on}(N) = \max \left(\begin{array}{c} N.\text{updated_on} \\ N.t.\text{updated_on} \\ \text{macro's updated_on} \\ \text{updated_on}(D_1) \\ \vdots \\ \text{updated_on}(D_i) \\ \vdots \\ \text{updated_on}(D_n) \end{array} \right) \quad (4.2)$$

The only difference with equation (4.1) is that the macro's inner `updated_on` value is taken into account (in red in equation (4.2)).

When a macro node is evaluated, the macro is first deep-copied and sent to the computation thread to ensure that the macro's state does not change during computation. Each of the current values that flow into the macro's input slots is copied and rewired to the corresponding input slots in the macro's DST. Then the macro's DST is evaluated and the macro's output values are calculated.

4.2.4 Macro user interface

The macro editing user interface is integrated into a `flak`'s UI. Figure 4.2 shows a screen capture of a `flak`'s macro editor. A new macro may be created by a right click on the graph editor and then selecting the "Create new macro" option. On clicking, an empty macro will pop up. Once a macro has been created, it can be re-used and added as many times as one wishes into the node editor. By double-clicking on a macro, this macro's editor is opened (if it was not) and focused. The macro editor is mostly similar to a `flak`'s main editor. It supports import and export of macros using a tailor-made data format.

In addition, nested macros are supported. a `flak` implements some basic sanity checks to prevent endless loops from occurring. For example, if a macro is added within itself (i.e. recursion), a `flak` will show an error and prevent the user from proceeding and shoot themselves in the foot.

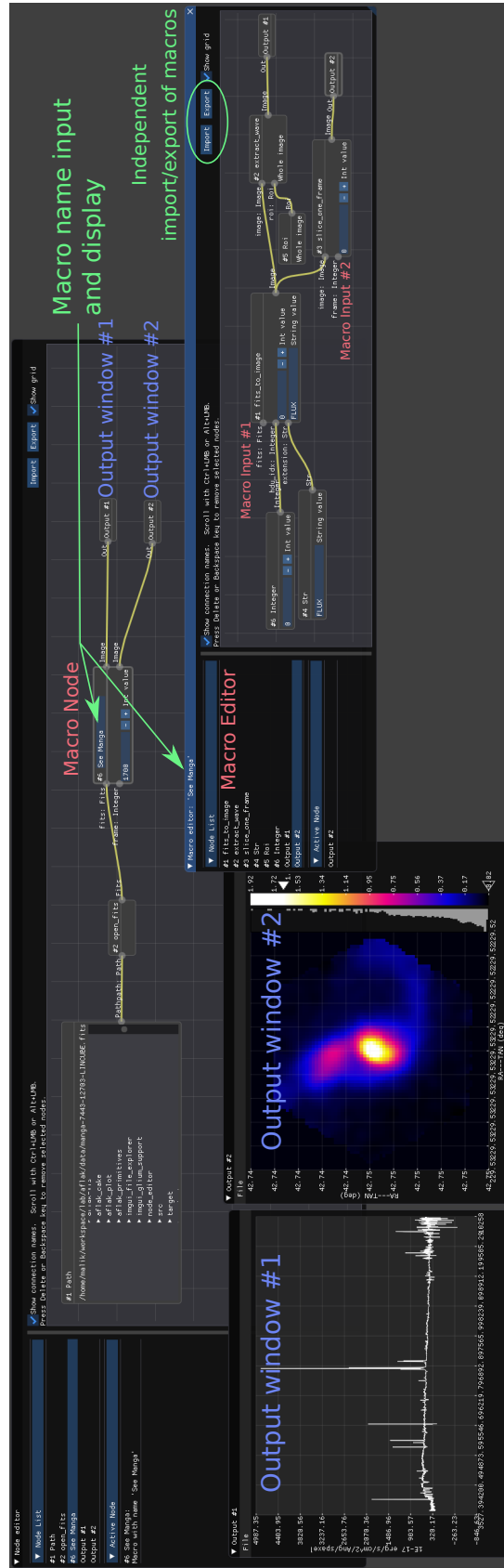


Figure 4.2 Example of a macro that opens an image from the MaNGA dataset [B⁺15]. This simple macro takes two inputs: a FITS file and an integer representing a frame number. The macro then outputs the waveform and the image data at the provided frame. This is the result that can be seen in output windows #1 and #2. The way the FITS file is open and the choice of extension (FLUX) is quite specific to the MaNGA dataset, hence the merit of defining a macro for such task.

4.3 SIA integration for provenance management

4.3.1 Overview of the SIA specification

As stated in the previous section, SIA (*Simple Image Access*) allows to make query on datasets from a data repository. SIA consists in a server and a client. The server contains a queryable database of astronomical objects and their corresponding multi-dimensional data files.

SIA relies on HTTP to communicate with the server in a simple client-server relationship. Several parameters can be appended to an HTTP GET request within the query string. Such parameters include POS to query an object by sky coordinates, BAND to query an object by energy intervals, and so on. Describing all the possible query parameters would be beyond the scope of this paper.

The response to an SIA query is a VOTable, which is basically a table containing specific astronomical metadata in an XML schema, whose specification is defined in the *VOTable Format Definition* [OR⁺13]. The result returned by an SIA query is a specific subset of VOTable, called ObsCore table, whose structure is described in the specification of the same name [LTD⁺17]. An ObsCore table records much information per row and not all of it is of interest for a `flak`'s use cases. The information that mostly interests us for integration with `flak` includes the ID of the object (which is referred to as `obj_publisher_id` in the specification, for simplicity, we will hereafter just refer to this string as "ID"), the direct download link to the dataset (called `access_url`) and the MIME type of the data (called `access_format`). As the SIA standard requires that the link in `access_url` points to a FITS file if the `access_format` is `application/fits`, we will restrict our queries to the `application/fits` format.

As `flak` is implemented in Rust [rus], all the standards mentioned above had to be implemented from scratch in order to integrate them with `flak`.⁵ Implementations in Python (in the Astropy and PyVO packages [pyv]) exist and were used as references.

4.3.2 Integration with SIA

To integrate *Simple Image Access 2.0* with `flak`, we created a node called `sia_query` that takes the URL of a data repository and some query parameters as input (see Figure 4.3). The user can use a drop-down to select from a list of common data repositories, or use a custom URL. The node runs the query, and parses the resulting response into an ObsCore

⁵Those Rust implementations of the VO standards are available as standalone libraries separate from `flak` in <https://github.com/aflak-vis/vo>.

table. The resulting ObsCore table can be previewed in a separate window if necessary by connecting the output result into an output node (as a reminder, all output nodes in `af1ak` connects to an output window, inside which the data flowing into the output node can be visualized).

We then created another node, `access_url`, to extract the direct hyperlink to the dataset from a row of the ObsCore table via the row index. Finally we prepared a `download_fits` node that downloads FITS files from a URL, decompresses them (if they are compressed), and caches them to the disk. By connecting all those four nodes together, we can query metadata, select a row from the result and download the data to the referred object. The full workflow can be seen in Figure 4.3.

4.3.3 Provenance management with `af1ak`

The complete current state of the node editor can at any time be exported and serialized as a `.ron` file. RON (for *Rusty Object Notation*)⁶ is a data format similar to JSON, but more adapted to Rust data-structure, allowing for smoother integration with `af1ak`. `af1ak` can then load a RON file, deserialize it and reproduce a previous node editor state, either by clicking the “Import” button in the graphical interface or by using the `--load` argument from the command line interface.

Moreover, each output result can be separately exported. Depending on the type of output result, it will be exported as a simple text file (for example for string of characters or numbers), or as a FITS file. Multi-dimensional data (including 1-dimension spectrum) is all exported as FITS files to which is attached metadata describing how this FITS file was generated inside `af1ak`'s node editor. Thus, the state of the node editor is embedded in all exported FITS files. Those FITS files are standard compliant and can be opened with any FITS viewer. But more importantly, when they are re-opened with `af1ak`, the visual program that was used to create them is automatically deserialized from the FITS metadata and displayed in the node editor.

Now, let us explain how we embedded the node editor's state into a FITS file. A FITS file is composed of a collections of consecutive *Header Data Units* (HDUs). Each HDU is composed of two parts: the required header and an optional data array or table. When `af1ak` exports a dataset into a FITS file, a FITS file with a single HDU will be created, with a data array containing the exported dataset encoded as big-endian IEEE 32-bit floating point numbers. Now, the exported FITS file's header contains a specific key with the name `AFLAPROV` (keys in FITS headers are limited to eight ASCII characters), with the

⁶<https://github.com/ron-rs/ron>

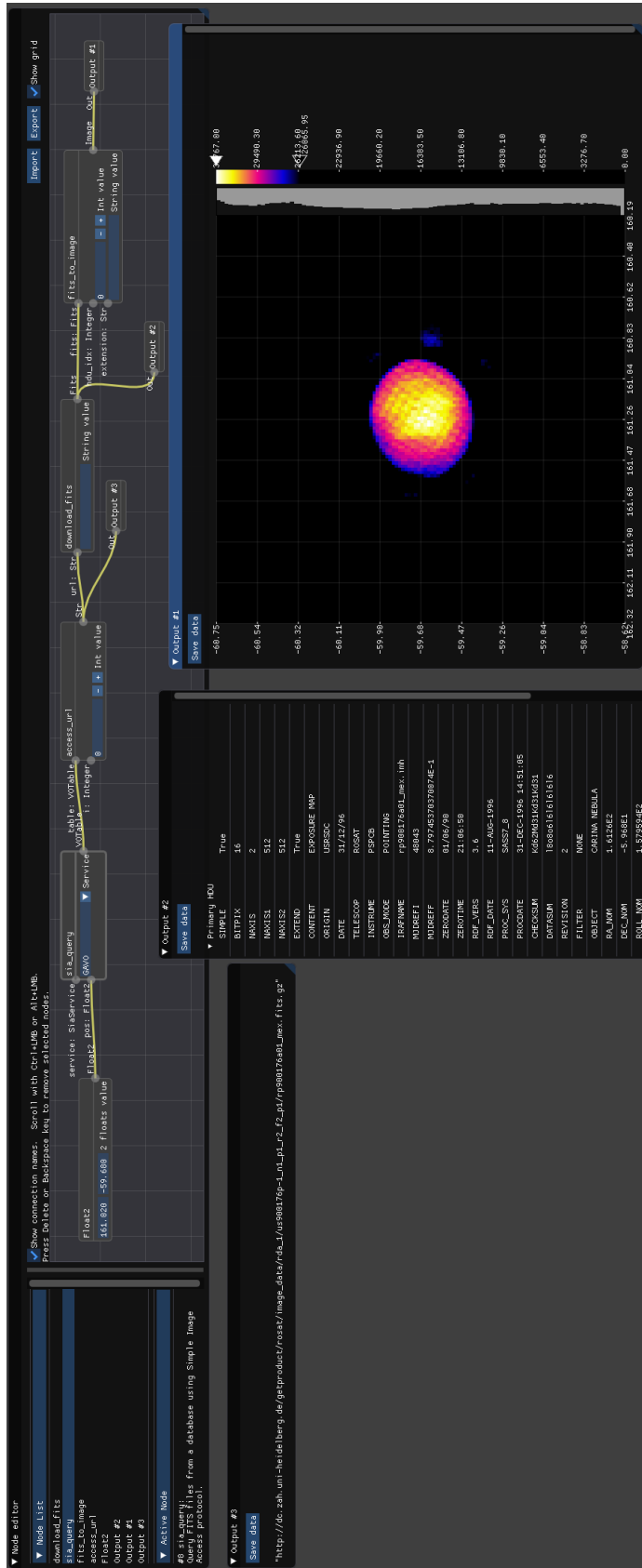


Figure 4.3 aflak querying and displaying an object from the GAVO (*German Astrophysical Virtual Observatory*) repository. The user can select the sky coordinates of the object and a data repository that will be queried. The record is then downloaded, cached and displayed on the screen. Some intermediary results, like the direct URL from which the image is downloaded or FITS metadata are displayed as well. The displayed object has the following ID: `ivo://org.gavo.dc/~?rosat/image_data/rda_1/us900176p-1_n1_p1_r2_f2_p1/rp900176a01_mex.fits.gz`

serialized state of the node editor as associated value. The FITS 4.0 standard indeed allows for strings of arbitrary length by using the CONTINUE syntax in the header [Gro]. `af1ak` uses this syntax to store the full state of the node editor and thus there is no limit to the length of the deserialized editor state.

Besides, as definitions of transforms may change with time, it is as well necessary to version each individual transform defined in `af1ak`. To that aim, `af1ak` uses semantic versioning [RvDV17]. Currently the code of the transform is not included in the serialized metadata, only an identifier to the transform, which means that an experienced `af1ak` user who creates their own nodes (beyond the nodes built in `af1ak`) must as well provide the code of the transforms he/she created to fully guarantee reproducibility of a result.

4.4 User interface: An event-based architecture

The single-threaded user interface loop that `af1ak` runs follows the following behavior.

1. Frame starts.
2. Pings compute back-end to check if there is any finished task or any task that should be started.
3. Records all user events (mouse, keyboard, window resize, etc.).
4. Re-draws the interface using OpenGL. The interface is drawn from the current immutable state of the node editor.
5. Converts each received user events into an action (e.g. “create a new node”, or “connect one slot to another”, etc.).
6. Mutates the state of the program given the list of actions to perform.
7. Frame ends.

Each time the user initiate an action, an instance of an event as defined in Listing 4.1 is fired and processed.

Listing 4.1 Event enumeration used for node editor’s user interface

```
use cake :: {  
    macros,  
    InputSlot, NodeId, Output, Transform, TransformIdx  
};
```

```

pub enum RenderEvent<T: 'static, E: 'static> {
    Connect(Output, InputSlot),
    AddTransform(&'static Transform<'static, T, E>),
    CreateOutput,
    AddConstant(&'static str),
    SetConstant(TransformIdx, Box<T>),
    WriteDefaultInput {
        t_idx: TransformIdx,
        input_index: usize,
        val: Box<T>,
    },
    RemoveNode(NodeId),
    Import,
    Export,
    AddNewMacro,
    AddMacro(macros::MacroHandle<'static, T, E>),
    EditNode(NodeId),
}

```

The name of the events should mostly be self-explanatory. Such architecture allows us to manage complexity by keeping as many part of the software immutable, while enabling code re-use. The same event architecture is indeed used for that macro editor and the main editor. Indeed, only the implementation of the event handler changes depending on whether the current editor is a macro editor or the main editor.

4.5 Implementing astronomical libraries in Rust

4.5.1 FITS libraries

`fits` is a FITS library that was created for `aflak` in Rust. It is published on <https://crates.io/crates/fits>. At the time of writing, it has 4,236 downloads.

As shown in Figure 4.4, a FITS file is basically a consecutive lists of headers and data (the tuple (header, data) is called an HDU for Header/Data Unit in FITS's jargon). Only the primary header is required. All subsequent headers and data are optional and thus called "extensions," however they are frequently used, especially to compact several kinds of data inside a single file. When building a FITS library, it is important not to read or load the

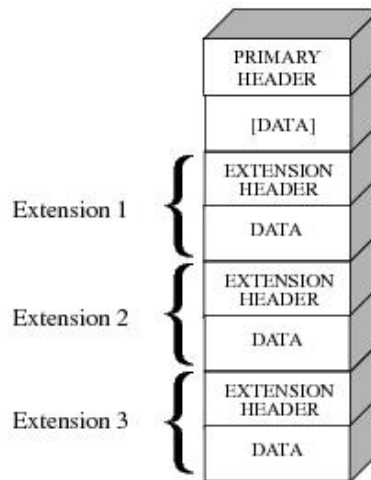


Figure 4.4 FITS file structure. Image courtesy of *Introduction to the HST Data Handbooks*, section 2.2 on “Multi-Extension FITS File Format” [S⁺11].

whole file into memory, but only to jump and access the important data regions for a the process one wants to achieve. Keeping this in mind is key to writing a fast FITS parser.

4.5.2 Convenience in opening FITS files

Every FITS file is provided in some sort of “sub-dialect.” The previous subsection explains the general structure of FITS files, and this structure is common to all FITS files. What differs between sub-dialects is the position or the ordering of the data. Then the question that comes is, “which extension contains the flux data (for example) I need for my research?” This depends on the origin of the dataset. Figure 4.2 shows a macro that deals with opening and displaying a FITS file from the MaNGA dataset.

Then we may wonder which sub-space of a multi-dimensional data array contains the data we are interested in. For example, a FITS files that contains polarization may contain a 4-dimensional datacube (right ascension, declination, wavelength and polarization), however if an astronomer is not interested in the polarization value, they may want to slice it out right from the beginning. There are even instances of FITS files with a single unique polarization value (the fourth dimension only holds a single “slice”). In that case, rather than keeping a 4-dimensional image with a dimension of $(x, y, z, 1)$, we can normalize it to a 3-dimensional image with a dimension of (x, y, z) . `af1ak` implements some heuristics to pre-generate a node program that allows to visualize the data in the FITS file. The astronomer can then get started on his analytical work from there.

4.5.3 Virtual Observatory standards

The author reserved the `vo` namespace on `crates.io`: <https://crates.io/crates/vo>. From then, more virtual observatory standards can be implemented, in addition to the implementation `vo_sia` and `vo_table`, which are not published to `crates.io` yet. The source for those implementations is available on GitHub at <https://github.com/aflak-vis/vo>

4.6 Defining your own nodes with Rust

Disclaimer

Please refer for the latest documentation built by `cake` with `rustdoc` when you attempt to define your own nodes. This document was written in August 2019 and may not be up-to-date with the latest advancements!

For this example, we will implement a simplified FITS loader node (a similar loader already exists, but you may want to create a custom loader for a specific file type that you use and that `aflak` does not support).

First, as we are building a black box with some outputs and inputs, we must first determine what are the outputs and inputs that we want. As we saw in section 4.5.1, a FITS file is composed of several independent HDUs (at least one). We thus need a path and the index of the HDU we wish to load as input. These are the two inputs we will be using to building our node. Then what do we need as output? We will assume that the FITS file to load contains an n -dimensional image, so we will use the `Image` type as output. In listing 4.2, we build a transform with the inputs and output and mentioned above.

Listing 4.2 How to define your own custom node in Rust

```

#[macro_use]
extern crate aflak_cake as cake;
// IOValue is the internal type for a value that flows in
// the node graph.
// IOErr represents a runtime computational error in the
// dataflow.
use aflak_primitives::{IOValue, IOErr};
// Defines a transform object.
// This object, once loaded by cake, will represent a node.
let transform = cake_transform!(
    // This string is similar to a Python docstring.
    // It contains the documentation of the node we are
    // creating.
    "Extract_dataset_from_a_FITS_file_at_the_given_Path.

FITS_HDU_is_chosen_by_the_index_given_by_'hdu_idx'
(defaults_to_0,i.e._the_Primary_HDU).

Author: Maliki Olivier Boussejra
Date: 2019-07-01",
    // Versioning of our node,
    // abiding by semantic versioning (SemVer)
    1, 0, 0,
    // Name of our node. Here "fits_path_to_image".
    fits_path_to_image<IOValue, IOErr>(
        // First input of our node: a file path
        path: Path,
        // Second input of our node:
        // an integer to select the HDU to open
        hdu_idx: Integer = 0,
    ) -> Image {
        let image = /* Generate image from path */;
        vec![IOValue::Image(image)]
    }
);

```

Then Listing 4.3 implements the actual code that was commented out in Listing 4.2, leaving out all the metadata:

Listing 4.3 Load a FITS file

```
extern crate fits;
// WcsArray is an floating point data array to which
// WorldCoordinates metadata is appended
use aflak_primitives::{IOValue, IOErr, WcsArray};

fits_path_to_image<IOValue, IOErr>(
    path: Path,
    hdu_idx: Integer = 0,
) -> Image {
    let path = path.as_ref();
    let result = fits::Fits::open(path)
        .map_err(|err| IOErr::IoError(
            err, format!("Could not open file '{:?}'", path)
        ))
        .and_then(|fits| {
            // Ignore error handling and validation for hdu_idx
            fits.get(hdu_idx as usize).ok_or(
                IOErr::UnexpectedInput(format!(
                    "Could not find HDU '{}' 'in' '{:?}'!",
                    hdu_idx, path,
                ))
            )
        })
        .and_then(|hdu| WcsArray::from_hdu(&hdu))
        .map(IOValue::Image)
    });
    vec![result]
}
```

4.7 DevOps

DevOps is a new word that was created during the last decade in the context of software development and release [BWZ15]. It refers to a set of software development practices that combine software development (Dev) and information technology operations (Ops).

The objective is to shorten the systems development life cycle while delivering features, fixes, and updates. In this section we will study the processes behind how `af1ak` was developed.

4.7.1 Portability: Challenges in supporting Linux (Debian and Ubuntu), macOS and Windows

To increase our user-base, we had to build a portable piece of software that runs on all major operating systems. Astronomers have a culture rooted in UNIX systems. Most research-oriented astronomers, which are `af1ak`'s target as a userbase, use Linux (usually Debian or Ubuntu) or macOS. This can be shown with the Debian astro packages: Almost all astronomy tools are packaged for Debian.

`af1ak` relies on portable technology. It uses a `glfw`-like Rust library that abstracts away the interaction of the program with the window and OpenGL management layer of the operating system. The library is called `glium`⁷. As `af1ak` is mainly developed on Linux (Debian and Ubuntu), we can assure that it is functional on such platforms. For some time it used to crash on Windows, but the latest version is functional on the Microsoft operating system. As for macOS, OpenGL and user event support kept breaking on macOS on every new OS version. It was needed to continually test against new macOS versions. Because the development team does not have any easy access to macOS machines, some of our macOS users were extremely helpful in helping us debugging `af1ak`.

4.7.2 Development workflow

`af1ak`'s development workflow makes heavy use of GitHub and Travis CI (Continue Integration). This allows us to waste less time on trivial time-consuming tasks such as testing and code-merging.

4.7.3 Release mechanism

`af1ak` distributes a binary through a relatively straightforward build system (Linux and macOS only, no Windows binary for now). The latest nightly binary can be downloaded from `af1ak`'s main GitHub page.

Besides, all the astro Debian packages which we mentioned in section 4.7.1 are maintained by Ole Streicher, from the Leibniz Institute for Astrophysics, whom the author met

⁷<https://github.com/glium/glium>

at ADASS XXVIII. Ole is open to packaging new useful astro-software to Debian. This is a path we will consider when packaging `af1ak`.

On the other hand, `af1ak` can be very easily downloaded and installed directly from its git repository using Rust's build system, *cargo*, provided that a user has a working Rust build environment installed on their machine. This ease of install and the still lack of maturity of `af1ak` in regards to Debian's standard of stability is a conclusive reason why we are still packaging the application ourselves, rather than relying on Debian's package managers.

Chapter 5

Evaluation

“Why Evaluate Software Quality?”

1. The software product may be hard to understand and difficult to modify.
2. The software product may be difficult to use, or easy to misuse.
3. The software product may be unnecessarily machine-dependent, or hard to integrate with other programs.”

Barry W. Boehm [BBL76]

`af1ak` was evaluated on a middle-end light laptop running Debian 9 with an Intel® Core™ i7-7560U CPU @ 2.40GHz processor and an integrated GPU chipset. `af1ak` is supported on all mainstream operating systems (Linux, macOS and Windows 10).

5.1 Checking compliance to standards

`af1ak` connected to the GAVO SIAP 2.0 repository and issued a query to retrieve a dataset at sky coordinates (161.0, −59.7)¹. For the same query, the same result as with PyVO is gotten. A one-gigabyte dataset from another SIAP 1.0 repository at GAVO was as well queried, and apart from the initial download duration the first time `af1ak` attempts to download the dataset, all individual transforms run in less than a second. With the hardware stated above, all animations run perfectly smoothly (see Figure 5.1).

FITS files exported with `af1ak` were tested with FITS viewers such as SAOImage DS9 and checked for strict compliance with the FITS standard using the Astropy checking tools. The exported FITS files can of course be re-loaded back into `af1ak`.

¹First component is right-ascension, second component is declination.

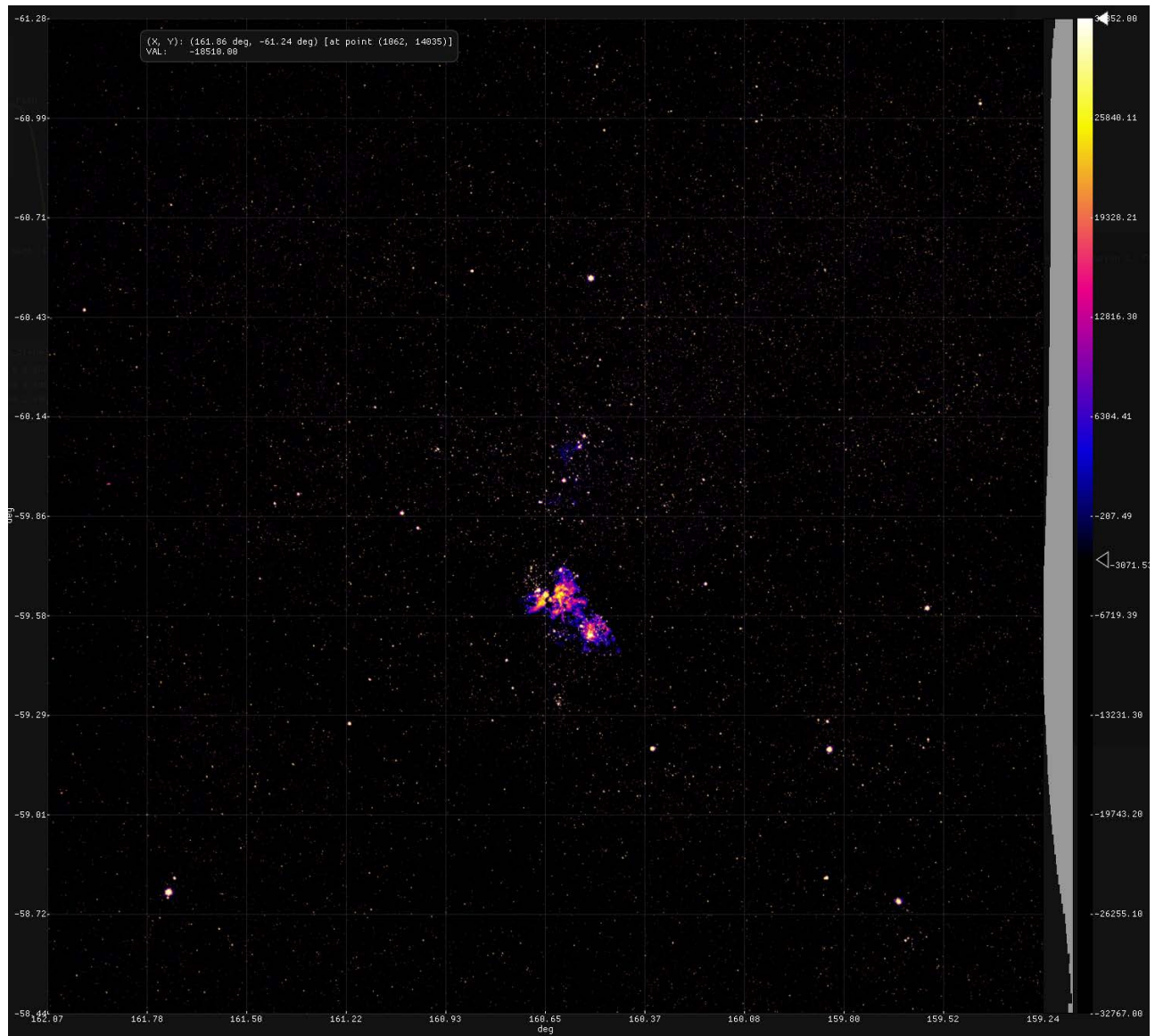


Figure 5.1 Image with a size in the gigabyte range queried from GAVO SIAP 1.0 repository, as shown within a flask. SIAP 1.0 dates from around 2002 and does not support any sort of “ID” for datasets. GAVO SIAP 1.0: <http://dc.zah.uni-heidelberg.de/hppunion/q/im/siap.xml>

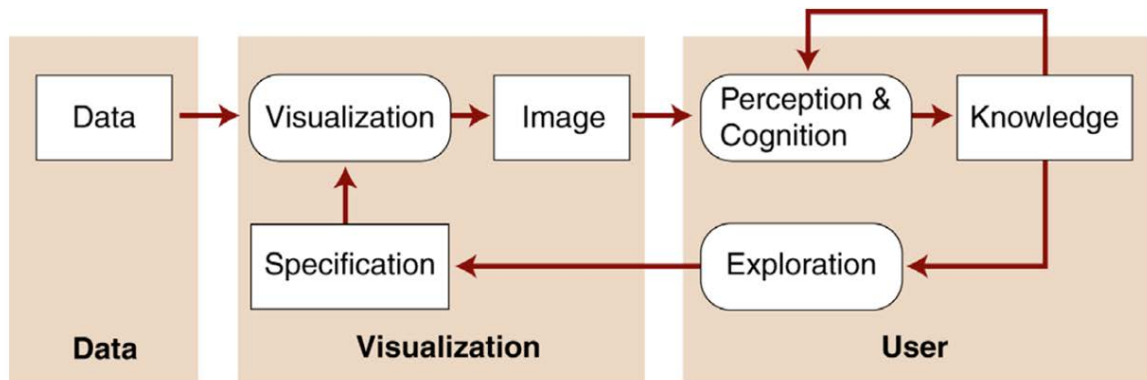


Figure 5.2 The visualization discovery process as presented by Johnson *et al.* [JMM⁺05]. This figure represents the core visualization concept of *human-in-the-loop*.

aflak’s code is free and available on the following link: <https://github.com/aflak-vis/aflak>. The implementation of the VO standards used by aflak can be found hereafter: <https://github.com/aflak-vis/vo>

5.2 First use case: Equivalent width

5.2.1 Introduction to *human-in-the-loop* concept

Figure 5.2 shows the visualization discovery process that aflak strives to achieve. aflak is designed “not to replace the human but to keep the human in the loop by extending human capabilities.” By allowing fast iteration (and prototyping) with varying input datasets and algorithms, aflak makes *astronomer-in-the-loop* a reality. The arrows in Figure 1.1, aflak’s teaser figure, explicitly shows this feedback loop in that the astronomer can at any time, by the sight of a visualized output, either go back and choose to re-query another input datasets, update the analytics pipeline in the visual program, or simply fine-tune parameters in the same visual program. While any of the above-mentioned action is done, the output values of the current visual program and thus the content of the output window is updated in real time. The next sections will give a specific example, that of the computing of equivalent widths, to prove the effectiveness of the feedback loop provided to the astronomer by aflak. Equivalent widths are an interesting subject in that they include several parameters that require manual and gradual tuning before a satisfactory result can be obtained.

5.2.2 Use case

Figure 2.5 shows an example of workflow extracting the equivalent width using the same computational method as Matsubayashi et al. [MYG⁺11]. Equivalent width can be defined as follows: “The equivalent width of a spectral line is a measure of the area of the line on a plot of intensity versus wavelength. It is found by forming a rectangle with a height equal to that of continuum emission, and finding the width such that the area of the rectangle is equal to the area in the spectral line. It is a measure of the strength of spectral features that is primarily used in astronomy” [CO07]. In Figure 2.5, the dataset is loaded with the “open_fits” node #2 and the “fits_to_image” node #3. The continuum emission is computed on both sides of the emission line by the “average” nodes #5 (left side) and #6 (right side). Then the average value of the spectral line is computed by the “average” node #4. From then, equivalent width is computed by node #8. The computed equivalent width is shown in output #3.

Computing the equivalent width requires a lot of iterations to choose the “just right” threshold wavelengths on computing continuum emission and the area of the spectral line. One interviewed astronomer reported that `af1ak` allows to quickly find the appropriate thresholds thank to the quick feedback loop. If we exclude negligible floating-point rounding errors, results obtained with PyRAF and `af1ak` are perfectly consistent.

5.3 Second use case: Velocity field map

Figure 5.3 shows a very simple program demonstrating how an astronomer can select a band on which there is an emission line (here from frame index 3135 to 3155). Similar to the first use case, the astronomer can select the appropriate thresholds for the band on which a velocity field map will be created and quickly try different computation methods.

5.4 Comparison with current tools

Sections 5.2.2 and 5.3 reviewed two analytic use cases for using `af1ak`. But such analytic features would be irrelevant without the ingrained support for reproducibility, implemented by embedding provenance data in exported FITS files whose compliance is checked in section 5.1. Bringing about *complete* reproducibility of *non-linear* analysis from a standard-compliant FITS file while maintaining interoperability with existing software is a first. This method has the advantage of simplicity in that it relies on widely supported standards and does not multiply the number of files a researcher needs to handle.

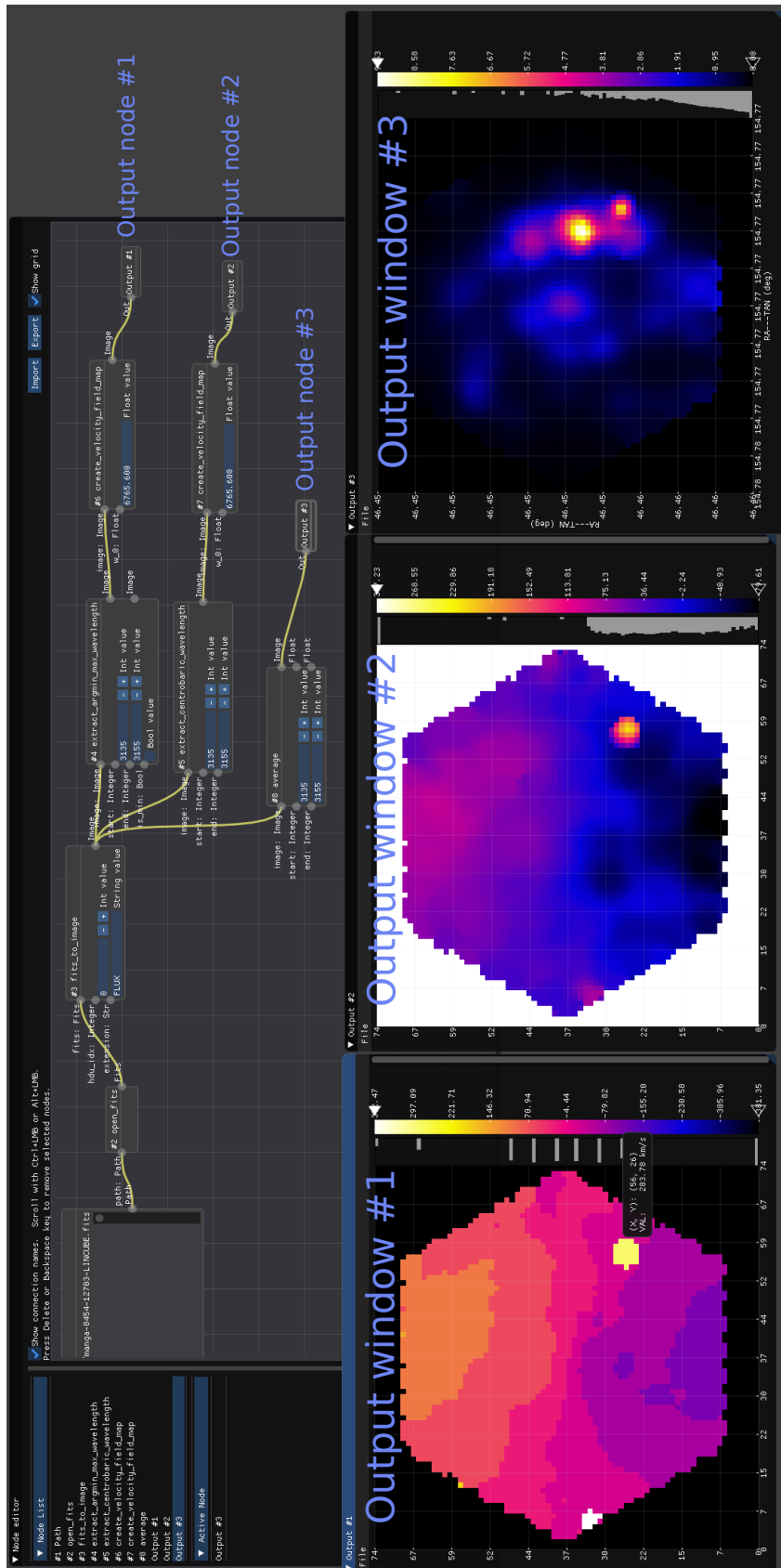


Figure 5.3 A node graph for computing the velocity field map using the effect of Doppler shift on an emission line. It generates two field maps (one in output #1 and one in output #2) with two different computation methods. Average value of the image around the emission line is shown in output #3. We see that there is a fast moving object on the lower right-hand side.

For example, we can compare with PyRAF [DLPWG01], which is a tool that provides a shell-like command prompt to analyze FITS datasets. Roughly, for each `af1ak` node an astronomer would need to manually run a PyRAF command. A PyRAF command takes at least two arguments: an input FITS file and an output FITS file. All the working files are saved and read on disk. As you may guess, the more PyRAF commands are run, the more FITS files appear in your working directory, making it very confusing to maintain the provenance of each file. Assuming that a diligent astronomer kept the history of the PyRAF commands somewhere in a text file, each command would take a few seconds to run, thus the whole process of going through a graph like that in Figure 2.5 would take about a minute. What's more, the astronomer would need to restart the process from scratch as soon as they need to change a parameter. Thanks to `af1ak`'s interactive interface, as soon as the node graph is built once, a user can seamlessly compare two different results by slightly modifying a single parameter (in a transformation, a query or input file). This can lead to new insights and discoveries.

But that is not all, thanks to the ability to download datasets from data repositories, there is no need to always keep a local copy of the data an astronomer wants to analyze (and remember where it is located). Each piece of data is uniquely identified by its `obs_publisher_id` and can be re-downloaded on demand. Query parameters can be changed with a few mouse movements and the result of the same processing on another input can be visualized after the new dataset is downloaded, or immediately if the queried dataset is cached on the disk. One interviewed domain expert reported that this is a non-negligible leap forward in terms of workflow management.

5.4.1 In-depth comparison

Listing 5.1 shows a commented Bash/IRAF script that computes the equivalent width of a datacube, using the same method as the node editor outlined in Figure 2.5.

Listing 5.1 Extracting equivalent width with Bash/IRAF

```

# Sub-datacube for each band (in Figure 2.5: nodes #3, #4 and #5)
awk 'BEGIN{for (i = 3099; i < 3119; i++) {
    printf ("manga-8454-12703-LINCUBE.fits [1][,,%d]\n", i)}
}' > file-off1.list
awk 'BEGIN{for (i = 3134; i < 3154; i++) {
    printf ("manga-8454-12703-LINCUBE.fits [1][,,%d]\n", i)}
}' > file-on.list
awk 'BEGIN{for (i = 3174; i < 3194; i++) {
    printf ("manga-8454-12703-LINCUBE.fits [1][,,%d]\n", i)}
}' > file-off2.list

# Then compute average on each band
imcomb @file-off1.list off1-average.fits combine=average
imcomb @file-on.list on-average.fits combine=average
imcomb @file-off2.list off2-average.fits combine=average

# Combine off-band images (in Figure 2.5: nodes #7 and #8)
echo "0.533333" > scale-off.dat
echo "0.466667" >> scale-off.dat
imcomb off1-average.fits , off2-average.fits off-average.fits \
    combine=average weight=@scale-off.dat

# Make equivalent-width map (in Figure 2.5: node #9)
imarith off-average.fits - on-average.fits off-on-average.fits
imarith off-on-average.fits * 20 flux.fits
imarith flux.fits / off-average.fits equivalent-width.fits

```

As the reader may infer from the above code, each operation is creating files on the file system, while subsequent operations are using the created files. The content of the intermediary files can only be seen using dedicated viewers such as DS9. With the example of equivalent widths, many constants must be precisely adjusted. Every time a constant is changed, all the next cascading operations must be re-run. Not only this process is time-consuming and error-prone—enlarging an already too long feedback loop—, but provenance is as well far from being managed. Out of the many files generated in

the file system, how does the user remember how each individual file was generated? By comparison, `af1ak` here shines by its fast feedback loop (less than a second is needed to refresh the visualizations in an output window after a change in the node editor) and automatic management of provenance. Importantly, if we exclude negligible floating-point rounding errors, results obtained with PyRAF and `af1ak` are perfectly consistent.

5.4.2 Equivalent width with a macro

The previous section compares existing tools with `af1ak`. This section gives a concrete example of macro usage for equivalent widths. Figure 5.4 shows an example of a macro implementing the computation of equivalent widths, the same processing as the one done in Figure 2.5. The advantage of using macros are clear: visual clutter on the original node interface is widely reduced, while the computing speed is not impacted for a 423 MB input dataset. Only the input datacube and the constant parameters that are relevant in computing the equivalent width are exposed by the macro. Through the sharing and the export of macros such as this one given as example, common parameterized analytics possibilities are only a few clicks away.

5.5 Advantages of provenance management in a visual context not limited to astronomy

Some astronomers highlight the importance of provenance management. A provenance data model standard is currently in development by the *International Virtual Observatory Alliance*, whose name is *ProvenanceDM* [SRB⁺19]. A representation of the data recorded through provenance is shown in Figure 5.5. `af1ak` can be used to reproduce an analysis pipeline, from the data source to the final output, following a model very similar to that of *ProvenanceDM*. As we consider the lists of use cases that *ProvenanceDM* must fulfill, we can see that `af1ak` successfully addresses each of them as well, so we can conclude that `af1ak` provides a *reference implementation* of the *ProvenanceDM* data model, as defined by the *International Virtual Observatory Alliance*.

Traceability of products: “Track the lineage of a product back to the raw material (backwards search), show the workflow or the dataflow that led to a product.” `af1ak`’s visual approach of representing data flow with a visualized directed acyclic graph meets this use case.

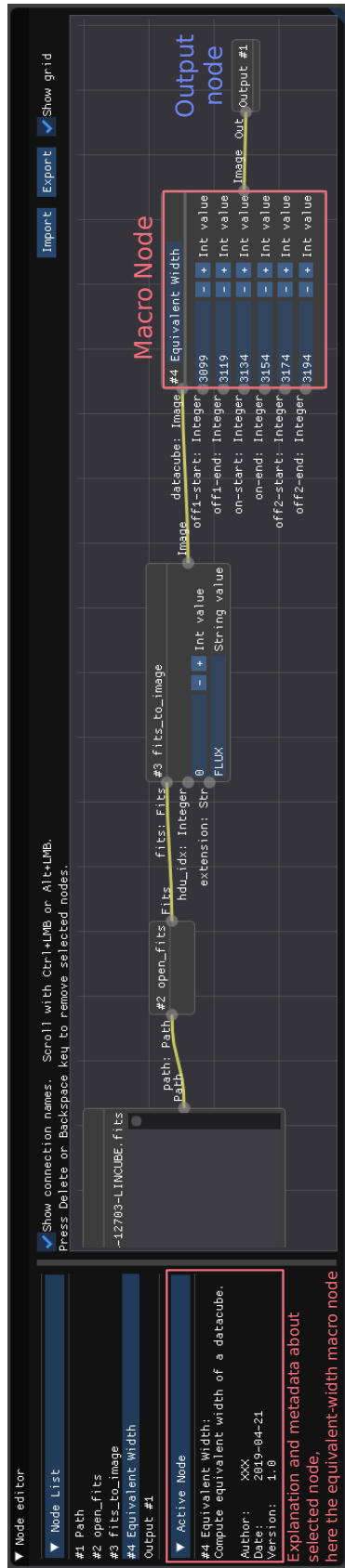


Figure 5.4 Example of using an `af1ak` macro to compute equivalent width. The macro encapsulates all the logic implemented as shown in the node editor in Figure 2.5, only exposing the relevant constants that the astronomers are expected to gradually adjust until they get a satisfactory outcome.

Acknowledgment and contact information: “Find the people involved in the production of a dataset, the people/organizations/institutes that one may want to acknowledge or can be asked for more information.” `af1ak`’s nodes contain information about their author, thus `af1ak` meets this use case.

Quality and reliability assessment: “Assess the quality and reliability of an observation, production step or dataset.” The version and the unique UUIDs of the nodes used in `af1ak` to generate any dataset are logged, which allows to check quality and reliability at any time in the future.

Identification of error location: “Find the location of possible error sources in the generation of a product.” All errors that occur during processing on `af1ak` are logged and a detailed stack trace is shown to thus user, thus fulfilling this requirement.

Search in structured provenance metadata: “Use provenance criteria to locate datasets (forward search), e.g. finding all images produced by a certain processing step or derived from data which were taken by a given facility.” It is theoretically possible to classify FITS files that were exported by `af1ak` grouped by the exact node editor that generated them, as a serialized node editor is included in all generated FITS files. Thus this requirement is met.

Besides, Wilkinson *et al.* define the FAIR principles for data sharing for *all* fields of science as “Findable, Accessible, Interoperable, Re-usable” [WDA⁺16], which is used as a design principle in the development of *ProvenanceDM*. `af1ak`’s macro fulfills the need for re-usability. Moreover, findability, accessibility and interoperability of datasets and methods are fulfilled by the implementation of Virtual Observatory standards for data retrieving and exchange (SIA, FITS format, etc.). Section 6.3 hints at the planned feature of implementing a sharing platform and repository for community-contributed nodes, which can then seamlessly be downloaded and extend `af1ak` with new analytical features. Such open repository would fulfill even more the accessibility requirements of the FAIR principles.

In any case, provenance management allows to keep track of whatever was done to come to a conclusion. All the trial and error of the researches that led them to the answer would be recorded as well. Thanks to this, we make possible the end-to-end reproducibility of any investigation. If an investigation were to be re-opened, it is not infrequent that a new investigation must be conducted. We could thus check exactly how the past investigation was done and improve the current search. The exact same needs can be found in forensic science. A framework dealing with forensic data thus has everything to gain

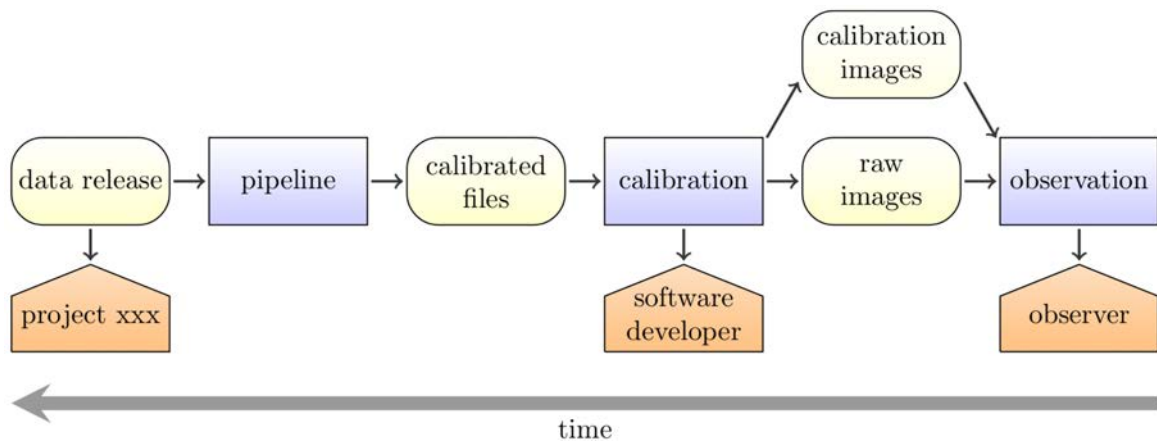


Figure 5.5 An example graph of provenance discovery. Starting with a released dataset (left), the involved activities (blue boxes), progenitor entities (yellow rounded boxes) and responsible agents (orange pentagons) are discovered. [SRB⁺19]

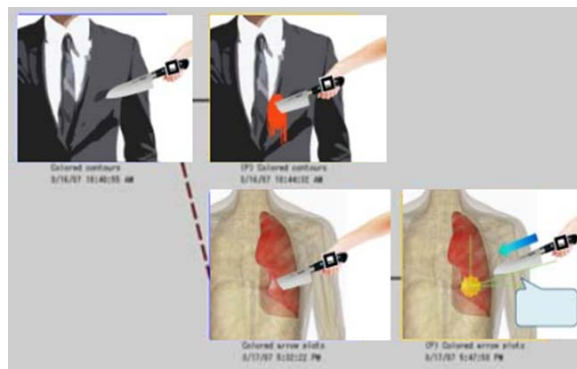


Figure 5.6 Provenance in forensics: Reconstitution of crime according to different hypotheses.

from implementing end-to-end data provenance management, and a f l a k's methodology to managing astronomical data provenance could as well be applied to forensic data provenance. Figure 5.6 represents how a graph-like provenance data model can be applied to a forensic case.

5.6 Distribution and recognition

A piece of software without users and a community is meaningless. While developing a f l a k, the author attempted to build a community around it. And even when this Ph. D. is finished, the desire to continue a f l a k's development will still be there. One could say that this Ph. D. was only an alibi to start this project.

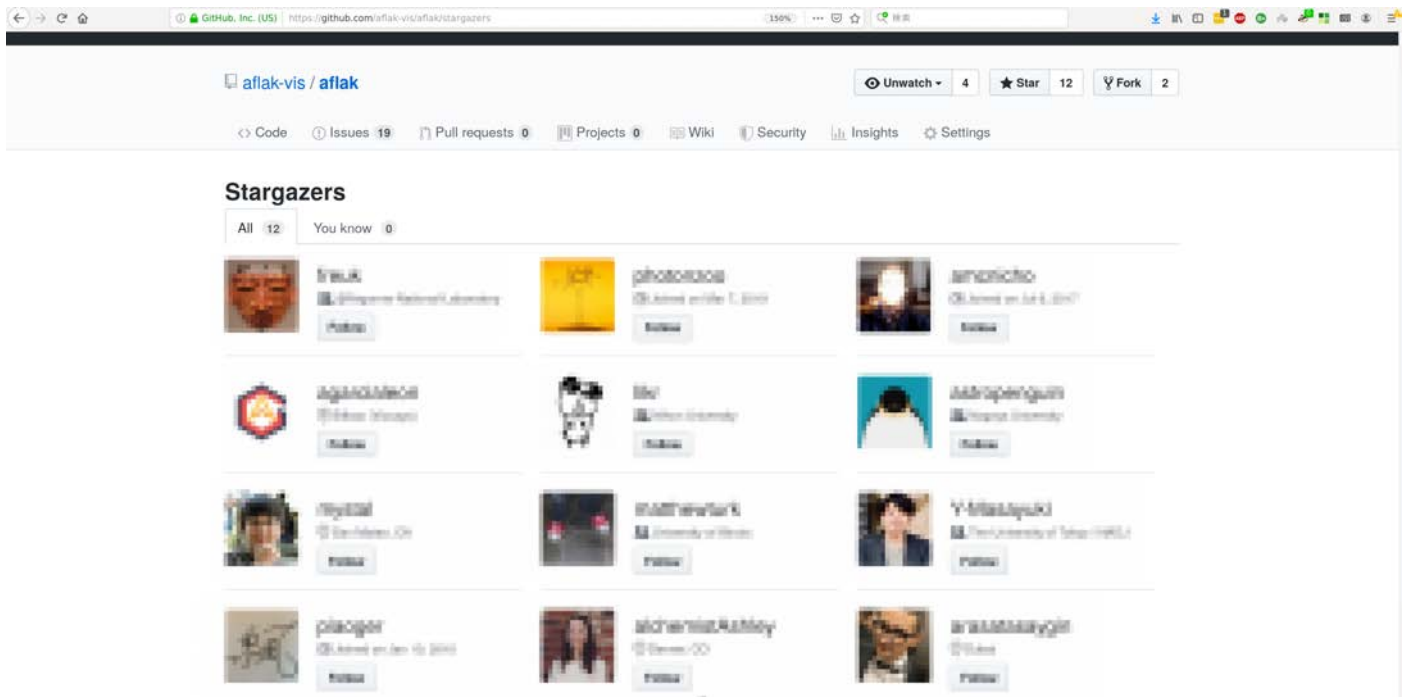


Figure 5.7 aflak’s repository has 12 stargazers. We never attempted to do any kind of pro-active campaign to earn stars.

In any case, this is not much, but aflak’s GitHub page has accumulated 12 stars on GitHub at the time of writing (see Figure 5.7 to see aflak’s stargazers²), along with 15 followers to aflak’s newsletter, run by this thesis’s author with Mailchimp³.

We as well went to many astronomical events to give oral presentation and talks. We received interested feedback from many people and even had the chance to be invited in the 2019 Integral Field Spectroscopy Study Group⁴, that will be held in November 2019.

We attempt to reduce the hurdle as low as possible for people starting to use aflak. Some astronomical software may be very cumbersome to install. As a matter of fact, the author was never able to get an IRAF installation running on his own machine. TOPCAT caused a few headaches and fiddling with the author’s Java environment before they could get it up and running. aflak boasts to be downloadable and installable from source with a single command—a one-liner—, as demonstrated in Listing 5.2.

²Screenshot taken at <https://github.com/aflak-vis/aflak/stargazers> on July 7th, 2019

³A marketing automation platform and an email marketing service at <https://mailchimp.com/>.

⁴“Integral Field Spectroscopy Study Group” is the author’s rough English translation of this conference’s original Japanese name, which is 面分光研究会 2019 –新面分光装置で花開く新しいサイエンス–.

Listing 5.2 Installing aflak: A one-liner.

```
# Install Rust (if you already have rust, skip this step)
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
# Install aflak
cargo install --git https://github.com/aflak-vis/aflak aflak

# Done!

# See CLI help
aflak --help

# Open a FITS file with aflak
aflak -f <FITS_FILE>
```

Chapter 6

Future Works

“The long term vision is not one of a fixed specific software package, but rather one of a framework which enables data centers to provide competing and co-operating data services, and which enables software providers to offer a variety of compatible analysis and visualization tools and user interfaces.”

The Virtual Observatory [Int]

6.1 Stronger interoperability with VO standards

af1ak allows to record the story of the analysis of astronomical datasets, and provides a fast and responsive environment for defining custom analytical macros and sharing them with fellow astronomers. While it currently uses its own *ad hoc* description of provenance using RON-file based deserialization, cooperation with the Provenance Data Model working group at IVOA would be a huge advantage to agree on a common format. This would allow interoperability between different software that manages analysis workflows, such as software used for cleaning and denoising raw data observed by telescopes.

Moreover, more comprehensive integration with other VO standards for querying would be a major advantage, including support for Simple Cone Search and Astronomical Data Query Language [DTB15].

6.2 Application to other astronomical problems

6.2.1 Arbitrary non-linear slicing

Some other astronomical problems may be adapted to be solved through exploratory analysis, as provided by `af1ak`. For the analysis or discovery of high-velocity clouds such as the ones sighted by Takekawa *et al.* in our home galaxy [TOI⁺17], it is very convenient to be able to make arbitrary, non-linear slicing through datacubes. Current `af1ak`'s built-in nodes and interface only supports linear slicing, it would be of great help for researchers to develop such general tool that allows to define a slicing method while visualizing it. It seems no such general tool currently exists, if we exclude custom, one-off code. We think that extending `af1ak` to include this feature is possible, and we suggest a design to implement such feature in Figure 6.1. `af1ak`'s development team plans to present this feature at the 2019 Integral Field Spectroscopy Study Group that will be held in November 2019.

6.2.2 Interferometry

What's more, interferometry analysis is a typical example of analysis with many degrees of freedom. We believe that `af1ak` would be adapted to allow an astronomer to interactively fine-tune the parameters of such analysis. Images produced by an interferometer are convolved with the Fourier transform of the data sampling, and are also affected by errors in the gain and phase calibration, often due to short-term changes in the atmosphere. AIPS, the *Astronomical Image Processing System*, was designed at about the same time as FITS was designed. Its main role has been to put interferometric data calibration, editing, imaging, and display programs in the hands of astronomers. It has provided useful functionality for data from various radio interferometers and single-dish radio telescopes [Gre03].

With the aging of AIPS, which was developed through the 1970s to the 1990s, next CASA (short for *Common Astronomy Software Applications*) took the role of main software stack for radio-astronomy analysis [MWS⁺07]. However, given the size of many datasets obtained with an interferometer, it may be difficult to maintain fast responsiveness, dear to `af1ak`. On the other hand, displaying the progress and the intermediary results of the currently running tasks would then be useful to track advancement and steer the computation, by offering an implementation of a visualization concept developed by Johnson *et al.* called "computational steering" [JPH⁺99].

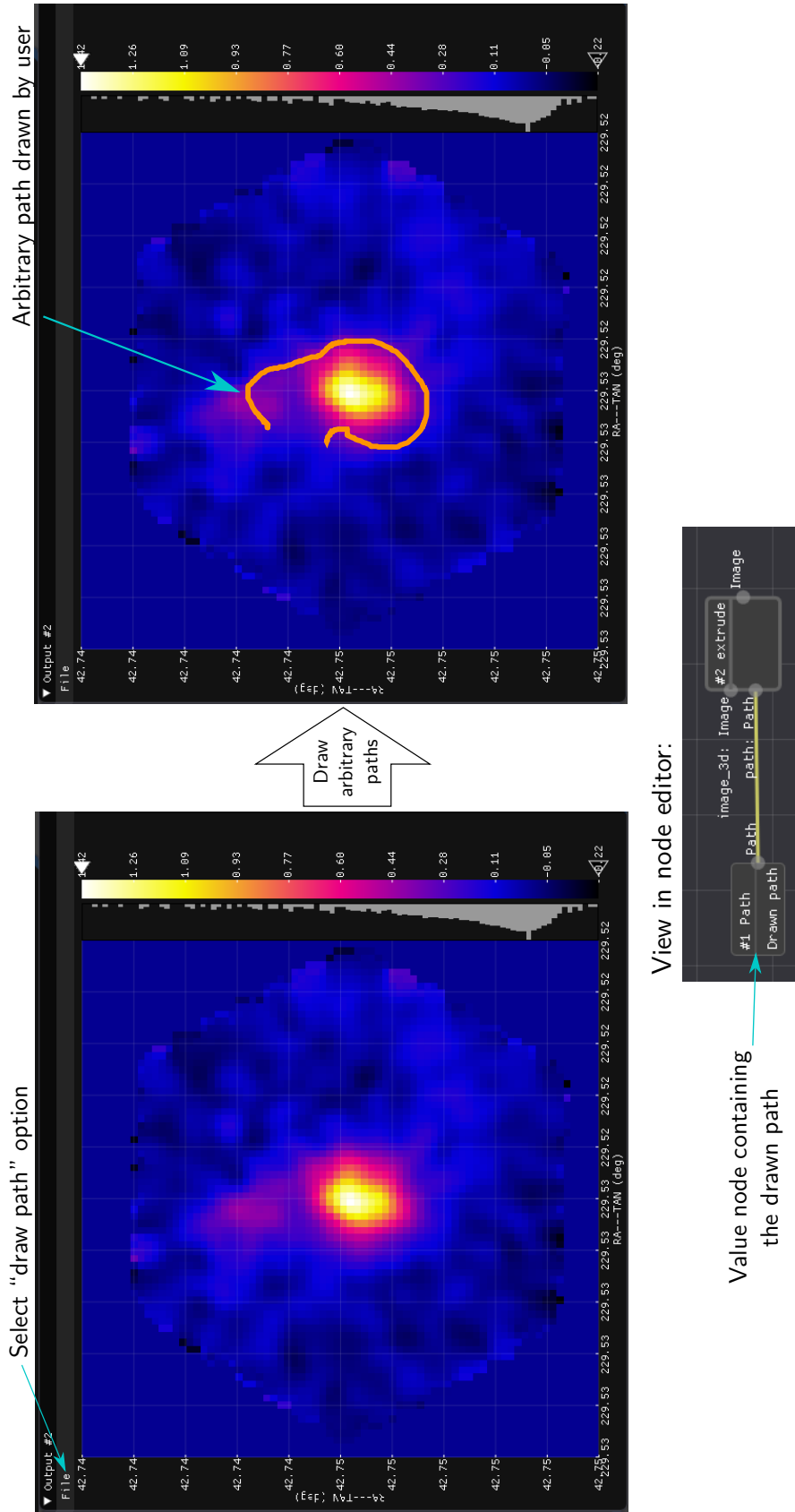


Figure 6.1 Representation of af1ak's planned extension to support arbitrary slicing.

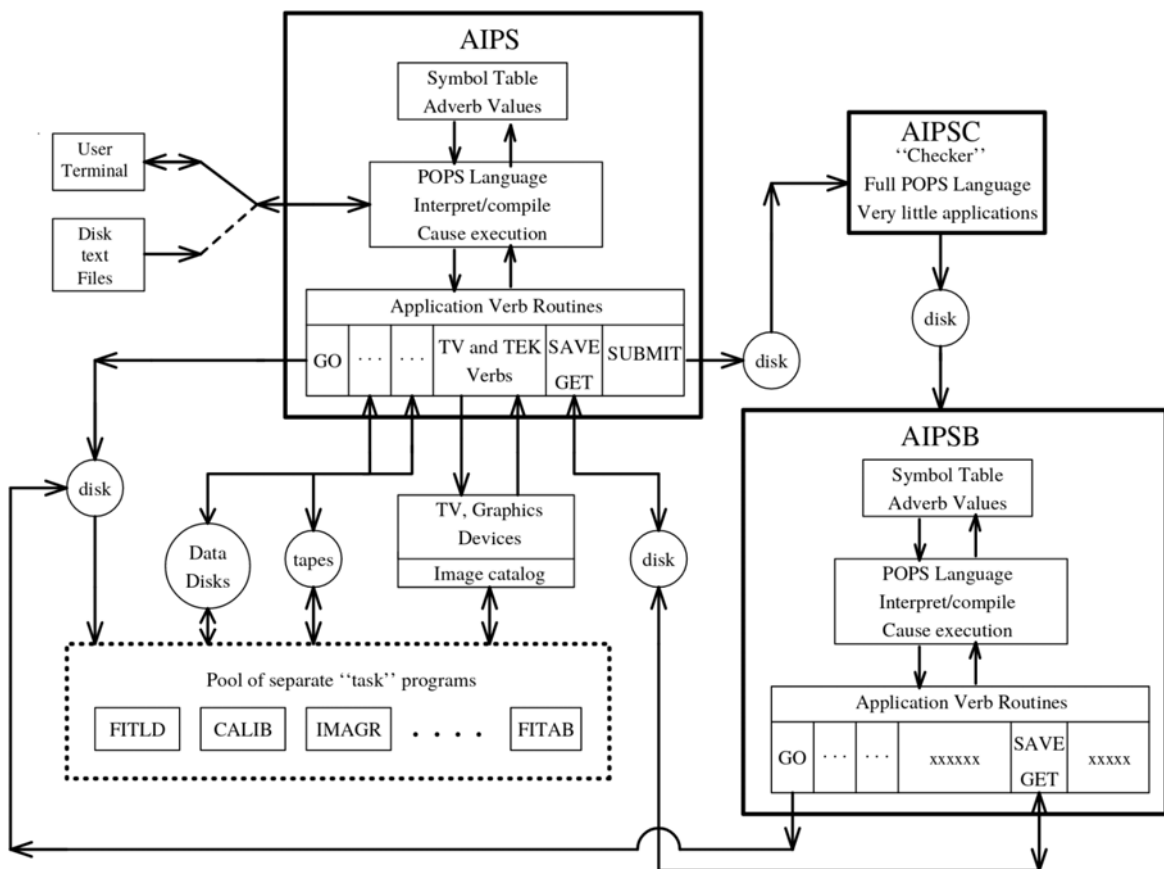


Figure 6.2 Block diagram of AIPS from a user point of view. Various communications paths are shown among the main interactive program, AIPS, the batch program AIPSB, and the collection of separate tasks (Figure 1 in [Gre03]).

6.3 More room for improvement

Other limitations include the current feature for selecting region of interest, which is limited to selecting pixels one at a time. This is sufficient for many use cases using relatively small datasets, but not in the general case. `af1ak` does not support very large datasets that do not fit in the computer's working memory. What's more, `af1ak`'s user interface was designed with a wide screen in mind—more than 2K pixels—and, although not impossible, it is impractical to use `af1ak` on smaller screens.

Furthermore, more memory-efficient use of cache is desired to reduce wasting of memory and have `af1ak` deal with bigger datasets (more than several gigabytes). Smarter probabilistic cache that garbage collects values that are very unlikely to be queried would be a huge improvement. The speed of the computing could as well be increased by introducing Just-In-Time (JIT) compilation technologies and/or clever task scheduling, beseeching us to borrow theoretical concepts from operating systems and compiler programming. Besides, on the UI side, being able to group several nodes into a single macro from the node editor screen would be extremely convenient and is a desirable feature.

We discussed about the issue of sharing the code for custom transformations made by experienced `af1ak` user in section 4.3.3. To overcome this issue, having a public `af1ak` code repository where anyone can contribute new nodes would be the solution. When loading a node editor from an exported FITS (or RON) file, the necessary code would be automatically downloaded from this public repository so that the analysis can really be 100% reproducible on any computer running `af1ak`. Having a public `af1ak` code repository where anyone can contribute new nodes or macros is a planned feature. The use of UUIDs to identify macros is a first step toward building an `af1ak` code repository.

Chapter 7

Conclusion

This thesis is a report on `af1ak`, a visual programming environment that provides fast and responsive macro support and visualization tools in the astrophysical domain. Moreover, we saw that `af1ak` successfully aims to empower astrophysicists to query and find insight in existing astrophysical datasets, while guaranteeing reproducibility through end-to-end provenance management. Ease of use, interactivity, responsiveness, collaboration, incremental improvements and interoperability with existing software in the astronomy community via the FITS and Virtual Observatory standards are taken very seriously.

Besides, by porting `af1ak` to the browser using WebAssembly, astronomical analysis and access to data could be made accessible to a broader audience. Anyone would be able to visualize *and* analyze public datasets and make reports that could be submitted to professional astronomers for review. Indeed, amateur astronomers far outnumber their professional counterparts.

Achievements

1. **M. O. Boussejra**, R. Uchiki, S. Takekawa, K. Matsubayashi, Y. Takeshima, M. Uemura, and I. Fujishiro. “aflak: Visual Programming Environment with Macro Support for Collaborative and Exploratory Astronomical Analysis,” *IEEE Transactions on Image Electronics and Visual Computing*, vol. 7, no. 2, 2019. [Accepted]
2. **M. O. Boussejra**, R. Uchiki, Y. Takeshima, K. Matsubayashi, S. Takekawa, M. Uemura, and I. Fujishiro. “aflak: Visual Programming Environment Enabling End-to-End Provenance Management for the Analysis of Astronomical Datasets,” *Visual Informatics*, vol. 3, no. 1, pp. 1–8, 2019.
3. 打木 陸雄, **M. O. Boussejra**, 松林 和也, 竹島 由里子, 植村 誠, 藤代 一成. 「AFLAK : モジュール可視化環境による等価幅マップの生成」. 日本天文学会 2019 年春季年会, R23a, 2019 年 3 月.
4. 木本 真理究, 竹川 俊也, 打木 陸雄, 松林 和也, 竹島 由里子, 植村 誠, 藤代 一成. 「アフラック : 分光データ解析用ビジュアルプログラミング環境」. 宇宙科学情報解析シンポジウム, 2019 年 2 月.
5. **M. O. Boussejra**, S. Takekawa, R. Uchiki, K. Matsubayashi, Y. Takeshima, M. Uemura, and I. Fujishiro, “aflak: Visual Programming Environment with Quick Feedback Loop, Tuned for Multi-Spectral Astrophysical Observations,” in *Proceedings of ADASS XXVIII*, 2018. [TBP]
6. **M. O. Boussejra**, K. Matsubayashi, Y. Takeshima, S. Takekawa, R. Uchiki, M. Uemura, and I. Fujishiro, “aflak: Pluggable Visual Programming Environment with Quick Feedback Loop Tuned for Multi-Spectral Astrophysical Observations,” in *Proceedings of IEEE VIS 2018*, vol. 3, 2018. [TBP]
7. **M. O. Boussejra**, N. Adachi, H. Shojo, R. Takahashi, and I. Fujishiro, “LMML: Describing Injuries for Forensic Data Visualization,” *2016 Nicograph International (NicoInt)*, p. 153, 2016. [Best Poster Award]
8. —, “LMML: Initial Developments of an Integrated Environment for Forensic Data Visualization,” in *EuroVis 2016 – Short Papers*, E. Bertini, N. Elmqvist, and T. Wischgoll, Eds. The Eurographics Association, pp. 31–35, 2016.
9. —, “LMML: Developing the Environment of the LMML Mark-up Language for Forensic Data Visualization,” *The Journal of the Institute of Image Electronics Engineers of Japan*, vol. 45, no. 1, p. 127, 2016.

References

- [AAA⁺19] K. Akiyama, A. Alberdi, W. Alef, K. Asada, R. Azulay, A.-K. Baczko, D. Ball, M. Baloković, J. Barrett, D. Bintley *et al.*, “First M87 Event Horizon Telescope results. I. The shadow of the supermassive black hole,” *The Astrophysical Journal Letters*, vol. 875, no. 1, p. L1, 2019.
- [ABJF06] I. Altintas, O. Barney, and E. Jaeger-Frank, “Provenance collection support in the Kepler scientific workflow system,” in *International Provenance and Annotation Workshop*. Springer, 2006, pp. 118–132.
- [B⁺15] K. Bundy *et al.*, “Overview of the SDSS-IV MaNGA Survey: Mapping Nearby Galaxies at Apache Point Observatory,” *The Astrophysical Journal*, vol. 798, no. 1, p. 7, Jan. 2015.
- [BAS⁺16a] M. O. Boussejra, N. Adachi, H. Shojo, R. Takahashi, and I. Fujishiro, “LMML: Describing injuries for forensic data visualization,” *2016 Nicograph International (NicoInt)*, p. 153, 2016.
- [BAS⁺16b] —, “LMML: Initial developments of an integrated environment for forensic data visualization,” in *EuroVis 2016 - Short Papers*, E. Bertini, N. Elmquist, and T. Wischgoll, Eds. The Eurographics Association, 2016, pp. 31–35.
- [BB99] P. Barrett and W. Bridgman, “PyFITS, a FITS module for Python,” in *Astronomical Data Analysis Software and Systems VIII*, vol. 172, 1999, p. 483.
- [BBL76] B. W. Boehm, J. R. Brown, and M. Lipow, “Quantitative evaluation of software quality,” in *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 1976, pp. 592–605.
- [BCC⁺05] L. Bavoil, S. P. Callahan, P. J. Crossno, J. Freire, C. E. Scheidegger, C. T. Silva, and H. T. Vo, “Vistrails: Enabling interactive multiple-view visualizations,” in *Visualization, 2005. VIS 05. IEEE*. IEEE, 2005, pp. 135–142.
- [BCD⁺09] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, K. Thiel, and B. Wiswedel, “KNIME – the Konstanz Information Miner: version 2.0 and beyond,” *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 26–31, 2009.
- [BFM98] H.-J. Bungartz, A. Frank, and F. Meier, “Design and implementation of a modular environment for coupled systems,” *Preprint SFB-438-9802*, 1998.

- [BGB13] L. Benali, S. Gromb, and C. Bou, “Post-mortem imaging in traffic fatalities: from autopsy to reconstruction of the scene using freely available software,” *International Journal of Legal Medicine*, vol. 127, no. 5, pp. 1045–1049, 2013.
- [BMT⁺19] M. O. Boussejra, K. Matsubayashi, Y. Takeshima, S. Takekawa, R. Uchiki, M. Uemura, and I. Fujishiro, “af1ak: Pluggable visual programming environment with quick feedback loop tuned for multi-spectral astrophysical observations,” *Proceedings of IEEE VIS (November 2018)*, vol. 3, 2019.
- [Bor18] K. Borne, “Massive data exploration in astronomy: What does cognitive have to do with it?” Oral presentation at Astronomical Data Analysis Software and Systems XXVIII. <http://adass2018.umd.edu/abstracts/I4-1.pdf>, 2018, accessed: 2018-12-10.
- [BTU⁺19] M. O. Boussejra, S. Takekawa, R. Uchiki, K. Matsubayashi, Y. Takeshima, M. Uemura, and I. Fujishiro, “af1ak: Visual programming environment with quick feedback loop, tuned for multi-spectral astrophysical observations,” in *Proceedings of Astronomical Data Analysis Software and Systems XXVIII*, 2019, TBP.
- [BUT⁺19] M. O. Boussejra, R. Uchiki, Y. Takeshima, K. Matsubayashi, S. Takekawa, M. Uemura, and I. Fujishiro, “af1ak: Visual programming environment enabling end-to-end provenance management for the analysis of astronomical datasets,” *Visual Informatics*, vol. 3, no. 1, pp. 1–8, 2019.
- [BWZ15] L. Bass, I. Weber, and L. Zhu, *DevOps: A software architect’s perspective*. Addison-Wesley Professional, 2015.
- [Cam] L. Campagnola, “PyQtGraph,” <http://www.pyqtgraph.org/>, accessed: 2018-06-03.
- [Cam95] G. Cameron, “Special focus: Modular Visualization Environments (MVEs),” *ACM Computer Graphics*, vol. 29, no. 2, pp. 3–60, 1995.
- [CO07] B. Carroll and D. Ostlie, *An Introduction to Modern Astrophysics*, ser. Pearson International Edition. Pearson Addison-Wesley, 2007.
- [DLPWG01] M. De La Pena, R. White, and P. Greenfield, “The PyRAF graphics system,” in *Astronomical Data Analysis Software and Systems X*, vol. 238, 2001, p. 59.
- [DTB15] P. Dowler, D. Tody, and F. Bonnarel, “IVOA Simple Image Access Version 2.0,” <http://www.ivoa.net/documents/SIA/20151223/REC-SIA-2.0-20151223.pdf>, 2015, accessed: 2018-12-09.
- [FS08] K. Franke and S. N. Srihari, “Computational forensics: An overview,” in *Computational Forensics: Second International Workshop, IWCF 2008. Washington, DC, USA, August 2008. Proceedings*, S. N. Srihari and K. Franke, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–10.
- [FSN⁺18] I. Fujishiro, N. Sawada, M. Nakayama, H.-Y. Wu, K. Watanabe, S. Takahashi, and M. Uemura, “TimeTubes: Visual exploration of observed blazar datasets,” *Journal of Physics: Conference Series*, vol. 1036, no. 1, p. 012011, 2018.

- [Gre03] E. W. Greisen, “AIPS, the VLA, and the VLBA,” in *Information Handling in Astronomy-Historical Vistas*. Springer, 2003, pp. 109–125.
- [Gro] F. W. Group, “Definition of the Flexible Image Transport System (FITS),” https://fits.gsfc.nasa.gov/standard40/fits_standard40aa-le.pdf, accessed: 2018-12-09.
- [GRS⁺18] A. Galkin, K. Riebe, O. Streicher, F. Bonnarel, M. Louys, M. Sanguillon, M. Servillat, and M. Nullmeier, “Provenance for astrophysical data,” in *International Provenance and Annotation Workshop*. Springer, 2018, pp. 252–256.
- [Gru09] T. Gruber, “Ontology,” in *Encyclopedia of database systems*. Springer, 2009, pp. 1963–1965.
- [HFB11] A. H. Hassan, C. J. Fluke, and D. G. Barnes, “Interactive visualization of the largest radioastronomy cubes,” *New Astronomy*, vol. 16, no. 2, pp. 100–109, 2011.
- [Int] International Virtual Observatory Alliance, “What is the VO?” <http://ivoa.net/about/what-is-vo.html>, accessed: 2019-08-12.
- [ivo] “International Virtual Observatory Alliance,” <http://www.ivoa.net/>, accessed: 2018-12-08.
- [JM03] W. Joye and E. Mandel, “New features of SAOImage DS9,” in *Astronomical data analysis software and systems XII*, vol. 295, 2003, p. 489.
- [JMM⁺05] C. Johnson, R. Moorhead, T. Munzner, H. Pfister, P. Rheingans, and T. S. Yoo, “NIH/NSF visualization research challenges report.” Institute of Electrical and Electronics Engineers, 2005.
- [JPH⁺99] C. Johnson, S. G. Parker, C. Hansen, G. L. Kindlmann, and Y. Livnat, “Interactive simulation and visualization,” *Computer*, vol. 32, no. 12, pp. 59–65, 1999.
- [Ken13] B. R. Kent, “Visualizing astronomical data with Blender,” *Publications of the Astronomical Society of the Pacific*, vol. 125, no. 928, p. 731, 2013.
- [LMS05] P. Leach, M. Mealling, and R. Salz, “A universally unique identifier (UUID) urn namespace,” Tech. Rep., 2005, <https://www.rfc-editor.org/rfc/pdfrfc/rfc4122.txt.pdf>.
- [LTD⁺17] M. Louys, D. Tody, P. Dowler, D. Durand, L. Michel, F. Bonnarel, A. Micol *et al.*, “Observation Data Model Core Components and its Implementation in the Table Access Protocol Version 1.1,” <http://www.ivoa.net/documents/ObsCore/20170509/REC-ObsCore-v1.1-20170509.pdf>, 2017, accessed: 2018-12-08.
- [MAA⁺16] D. Muna, M. Alexander, A. Allen, R. Ashley, D. Asmus, R. Azzollini, M. Bannister, R. Beaton, A. Benson, G. B. Berriman *et al.*, “The Astropy problem,” *arXiv preprint arXiv:1610.03159*, 2016.

- [mev] “MeVisLab: Medical image processing and visualization,” <http://www.mevislab.de/>, accessed: 2018-12-10.
- [Mor94] J. P. Morrison, “Flow-based programming,” in *Proc. 1st International Workshop on Software Engineering for Parallel and Distributed Systems*, 1994, pp. 25–29.
- [MSRMH09] J. Meyer-Spradow, T. Ropinski, J. Mensmann, and K. Hinrichs, “Voreen: A rapid-prototyping environment for ray-casting-based volume visualizations,” *IEEE Computer Graphics and Applications*, vol. 29, no. 6, pp. 6–13, 2009.
- [Mun17] D. Muna, “Introducing Nightlight: A new, modern FITS viewer,” in *Astronomical Data Analysis Software and Systems XXV*, ser. Astronomical Society of the Pacific Conference Series, N. P. F. Lorente, K. Shortridge, and R. Wayth, Eds., vol. 512, Dec. 2017, p. 621.
- [MWS⁺07] J. P. McMullin, B. Waters, D. Schiebel, W. Young, and K. Golap, “CASA architecture and applications,” in *Astronomical data analysis software and systems XVI*, vol. 376, 2007, pp. 127–130.
- [MYG⁺11] K. Matsubayashi, M. Yagi, T. Goto, A. Akita, H. Sugai, A. Kawai, A. Shimono, and T. Hattori, “Spatially resolved spectroscopic observations of a possible E+A progenitor: SDSS J160241.00+521426.9,” *The Astrophysical Journal*, vol. 729, no. 1, p. 29, 2011.
- [OLD⁺08] I. Ortiz, J. Lusted, P. Dowler, A. Szalay, Y. Shirasaki, M. A. Nieto-Santisteban, M. Ohishi, W. O’Mullane, P. Osuna *et al.*, “IVOA Astronomical Data Query Language Version 2.0,” <http://www.ivoa.net/documents/REC/ADQL/ADQL-20081030.pdf>, 2008, accessed: 2019-02-09.
- [OR⁺13] F. Ochsenbein, W. Roy *et al.*, “VOTable Format Definition Version 1.3,” <http://www.ivoa.net/documents/VOTable/20130920/REC-VOTable-1.3-20130920.pdf>, 2013, accessed: 2018-12-09.
- [OSSK13] T. Okuda, S. Shiotani, N. Sakamoto, and T. Kobayashi, “Background and current status of postmortem imaging in Japan: Short history of ‘Autopsy imaging (Ai)’,” *Forensic Science International*, vol. 225, no. 1–3, pp. 3–8, 2013, postmortem Imaging.
- [OTI⁺17] T. Oka, S. Tsujimoto, Y. Iwata, M. Nomura, and S. Takekawa, “Millimetre-wave emission from an intermediate-mass black hole candidate in the Milky Way,” *Nature Astronomy*, vol. 1, no. 10, p. 709, 2017.
- [Ott12] T. Ott, “QFitsView: FITS file viewer,” *Astrophysics Source Code Library*, October 2012.
- [PDKB⁺08] C. Pradal, S. Dufour-Kowalski, F. Boudon, C. Fournier, and C. Godin, “OpenAlea: A visual programming and component-based software platform for plant modelling,” *Functional Plant Biology*, vol. 35, no. 10, pp. 751–760, 2008.
- [Per10] A. Persson, “Postmortem visualization: The real gold standard,” in *Beautiful Visualization: Looking at Data through the Eyes of Experts*, J. Steele and N. Iliinsky, Eds. “O’Reilly Media, Inc.”, 2010, pp. 311–328.

- [PJ95] S. G. Parker and C. R. Johnson, “SCIRun: A scientific programming environment for computational steering,” in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*. ACM, 1995, p. 52.
- [PQF⁺14] S. Perkins, J. Questiaux, S. Finniss, R. Tyler, S. Blyth, and M. M. Kuttel, “Scalable desktop visualisation of very large radio astronomy data cubes,” *New Astronomy*, vol. 30, pp. 1–7, 2014.
- [pyv] “PyVO,” <https://pyvo.readthedocs.io/en/latest/>, accessed: 2018-12-09.
- [RB18] O. Rübél and B. P. Bowen, “BASTet: Shareable and reproducible analysis and visualization of mass spectrometry imaging data via OpenMSI,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 1025–1035, 2018.
- [RTG⁺13] T. P. Robitaille, E. J. Tollerud, P. Greenfield, M. Droettboom, E. Bray, T. Aldcroft, M. Davis, A. Ginsburg, A. M. Price-Whelan, W. E. Kerzendorf *et al.*, “Astropy: A community Python package for astronomy,” *Astronomy & Astrophysics*, vol. 558, p. A33, 2013.
- [rus] “The Rust Programming Language,” <http://www.rust-lang.org>, accessed: 2018-09-10.
- [RvDV17] S. Raemaekers, A. van Deursen, and J. Visser, “Semantic versioning and impact of breaking changes in the Maven repository,” *Journal of Systems and Software*, vol. 129, pp. 140–158, 2017.
- [RWD90] G. X. Ritter, J. N. Wilson, and J. L. Davidson, “Image algebra: An overview,” *Computer Vision, Graphics, and Image Processing*, vol. 49, no. 3, pp. 297–331, 1990.
- [S⁺11] E. Smith *et al.*, *Introduction to the HST Data Handbooks, Version 8.0*. Baltimore: STScI, 2011, available online at http://www.stsci.edu/itt/review/dhb_2011/Intro/intro_cover.html on 2019/07/10.
- [SRB⁺18] M. Servillat, K. Riebe, F. Bonnarel, A. Galkin, M. Louys, M. Nullmeier, M. Sanguillon, O. Streicher *et al.*, “IVOA Provenance Data Model Version 1.0,” <http://www.ivoa.net/documents/ProvenanceDM/20181015/PR-ProvenanceDM-1.0-20181015.pdf>, 2018, accessed: 2018-12-08.
- [SRB⁺19] M. Servillat, K. Riebe, C. Boisson, F. Bonnarel, A. Galkin, M. Louys, M. Nullmeier, N. Renault-Tinacci, M. Sanguillon, and O. Streicher, “IVOA Provenance Data Model Version 1.0. IVOA Proposed Recommendation 2019-07-19,” <http://www.ivoa.net/documents/ProvenanceDM/20190719/ProvenanceDM-1.0-20190719.pdf>, 2019, accessed: 2019-07-31.
- [SSO02] M. F. Sanner, D. Stoffler, and A. J. Olson, “ViPEr, a visual programming environment for Python,” in *Proceedings of the 10th International Python conference*, 2002, pp. 103–115.

- [SZP15] I. Suriarachchi, Q. Zhou, and B. Plale, “Komadu: A capture and visualization system for scientific data provenance,” *Journal of Open Research Software*, vol. 3, no. 1, 2015.
- [Tod86] D. Tody, “The IRAF data reduction and analysis system,” in *Instrumentation in astronomy VI*, vol. 627. International Society for Optics and Photonics, 1986, pp. 733–749.
- [TOI⁺17] S. Takekawa, T. Oka, Y. Iwata, S. Tokuyama, and M. Nomura, “Discovery of two small high-velocity compact clouds in the central 10 pc of our galaxy,” *The Astrophysical Journal Letters*, vol. 843, no. 1, p. L11, 2017.
- [UBS⁺12] M. Urschler, A. Bornik, E. Scheurer, K. Yen, H. Bischof, and D. Schmalstieg, “Forensic-case analysis: From 3D imaging to interactive visualization,” *IEEE Computer Graphics and Applications*, vol. 32, no. 4, pp. 79–87, Jul–Aug 2012.
- [WDA⁺16] M. D. Wilkinson, M. Dumontier, I. J. Aalbersberg, G. Appleton, M. Axton, A. Baak, N. Blomberg, J.-W. Boiten, L. B. da Silva Santos, P. E. Bourne *et al.*, “The fair guiding principles for scientific data management and stewardship,” *Scientific data*, vol. 3, 2016.
- [WG79] D. C. Wells and E. W. Greisen, “FITS: A Flexible Image Transport System,” in *Image Processing in Astronomy*, 1979, p. 445.