GNU Astronomy Utilities

Astronomical data manipulation and analysis programs and libraries
for version 0.9, 17 April 2019

Mohammad Akhlaghi

Gnuastro (source code, book and webpage) authors (sorted by number of commits):

Mohammad Akhlaghi (mohammad@akhlaghi.org, 1263)
Mosè Giordano (mose@gnu.org, 29)
Vladimir Markelov (vmatroskin@gmail.com, 18)
Boud Roukema (boud@cosmo.torun.pl, 7)
Andreas Stieger (astieger@suse.com, 1)
Leindert Boogaard (leindertboogaard@gmail.com, 1)
Lucas MacQuarrie (macquarrielucas@gmail.com, 1)
Thérèse Godefroy (godef.th@free.fr, 1)

This book documents version 0.9 of the GNU Astronomy Utilities (Gnuastro). Gnuastro provides various programs and libraries for astronomical data manipulation and analysis.

For myself, I am interested in science and in philosophy only because I want to learn something about the riddle of the world in which we live, and the riddle of man's knowledge of that world. And I believe that only a revival of interest in these riddles can save the sciences and philosophy from narrow specialization and from an obscurantist faith in the expert's special skill, and in his personal knowledge and authority; a faith that so well fits our 'post-rationalist' and 'post-critical' age, proudly dedicated to the destruction of the tradition of rational philosophy, and of rational thought itself.

—*Karl Popper. The logic of scientific discovery. 1959.*

Short Contents

Table of Contents

1 Introduction

GNU Astronomy Utilities (Gnuastro) is an official GNU package consisting of separate programs and libraries for the manipulation and analysis of astronomical data. All the programs share the same basic command-line user interface for the comfort of both the users and developers. Gnuastro is written to comply fully with the GNU coding standards so it integrates finely with the GNU/Linux operating system. This also enables astronomers to expect a fully familiar experience in the source code, building, installing and command-line user interaction that they have seen in all the other GNU software that they use. The official and always up to date version of this book (or manual) is freely available under Appendix C [GNU Free Doc. License], page 474, in various formats (PDF, HTML, plain text, info, and as its Texinfo source) at `http://www.gnu.org/software/gnuastro/manual/`.

For users who are new to the GNU/Linux environment, unless otherwise specified most of the topics in Chapter 3 [Installation], page 61, and Chapter 4 [Common program behavior], page 91, are common to all GNU software, for example installation, managing command-line options or getting help (also see Section 1.6 [New to GNU/Linux?], page 8). So if you are new to this empowering environment, we encourage you to go through these chapters carefully. They can be a starting point from which you can continue to learn more from each program's own manual and fully benefit from and enjoy this wonderful environment. Gnuastro also comes with a large set of libraries, so you can write your own programs using Gnuastro's building blocks, see Section 11.1 [Review of library fundamentals], page 320, for an introduction.

In Gnuastro, no change to any program or library will be committed to its history, before it has been fully documented here first. As discussed in Section 1.2 [Science and its tools], page 2, this is a founding principle of the Gnuastro.

1.1 Quick start

The latest official release tarball is always available as `gnuastro-latest.tar.gz` (`http://ftp.gnu.org/gnu/gnuastro/gnuastro-latest.tar.gz`). For better compression (faster download), and robust archival features, an Lzip (`http://www.nongnu.org/lzip/lzip.html`) compressed tarball is also available at `gnuastro-latest.tar.lz` (`http://ftp.gnu.org/gnu/gnuastro/gnuastro-latest.tar.lz`), see Section 3.2.1 [Release tarball], page 71, for more details on the tarball release[1].

Let's assume the downloaded tarball is in the `TOPGNUASTRO` directory. The first two commands below can be used to decompress the source. If you download `tar.lz` and your Tar implementation doesn't recognize Lzip (the second command fails), run the third and fourth lines[2]. Note that lines starting with `##` don't need to be typed.

[1] The Gzip library and program are commonly available on most systems. However, Gnuastro recommends Lzip as described above and the beta-releases are also only distributed in `tar.lz`. You can download and install Lzip's source (in `.tar.gz` format) from its webpage and follow the same process as below: Lzip has no dependencies, so simply decompress, then run `./configure`, `make`, `sudo make install`.

[2] In case Tar doesn't directly uncompress your `.tar.lz` tarball, you can merge the separate calls to Lzip and Tar (shown in the main body of text) into one command by directly piping the output of Lzip into Tar with a command like this: `$ lzip -cd gnuastro-0.5.tar.lz | tar -xf -`

```
## Go into the download directory.
$ cd TOPGNUASTRO

## Also works on 'tar.gz'. GNU Tar recognizes both formats.
$ tar xf gnuastro-latest.tar.lz

## Only when previous command fails.
$ lzip -d gnuastro-latest.tar.lz
$ tar xf gnuastro-latest.tar
```

Gnuastro has three mandatory dependencies and some optional dependencies for extra functionality, see Section 3.1 [Dependencies], page 61, for the full list. In Section 3.1.4 [Dependencies from package managers], page 68, we have prepared the command to easily install Gnuastro's dependencies using the package manager of some operating systems. When the mandatory dependencies are ready, you can configure, compile, check and install Gnuastro on your system with the following commands.

```
$ cd gnuastro-X.X                 # Replace X.X with version number.
$ ./configure
$ make -j8                        # Replace 8 with no. CPU threads.
$ make check
$ sudo make install
```

See Section 3.3.5 [Known issues], page 89, if you confront any complications. For each program there is an 'Invoke ProgramName' sub-section in this book which explains how the programs should be run on the command-line (for example Section 5.3.1 [Invoking Table], page 148). You can read the same section on the command-line by running `$ info astprogname` (for example `info asttable`). The 'Invoke ProgramName' sub-section starts with a few examples of each program and goes on to explain the invocation details. See Section 4.3 [Getting help], page 109, for all the options you have to get help. In Chapter 2 [Tutorials], page 16, some real life examples of how these programs might be used are given.

1.2 Science and its tools

History of science indicates that there are always inevitably unseen faults, hidden assumptions, simplifications and approximations in all our theoretical models, data acquisition and analysis techniques. It is precisely these that will ultimately allow future generations to advance the existing experimental and theoretical knowledge through their new solutions and corrections.

In the past, scientists would gather data and process them individually to achieve an analysis thus having a much more intricate knowledge of the data and analysis. The theoretical models also required little (if any) simulations to compare with the data. Today both methods are becoming increasingly more dependent on pre-written software. Scientists are dissociating themselves from the intricacies of reducing raw observational data in experimentation or from bringing the theoretical models to life in simulations. These 'intricacies' are precisely those unseen faults, hidden assumptions, simplifications and approximations that define scientific progress.

> Unfortunately, most persons who have recourse to a computer for statistical analysis of data are not much interested either in computer programming or in

> statistical method, being primarily concerned with their own proper business. Hence the common use of library programs and various statistical packages. ... It's time that was changed.
>
> —*F. J. Anscombe. The American Statistician, Vol. 27, No. 1. 1973*

Anscombe's quartet (`http://en.wikipedia.org/wiki/Anscombe%27s_quartet`) demonstrates how four data sets with widely different shapes (when plotted) give nearly identical output from standard regression techniques. Anscombe uses this (now famous) quartet, which was introduced in the paper quoted above, to argue that "*Good statistical analysis is not a purely routine matter, and generally calls for more than one pass through the computer*". Echoing Anscombe's concern after 44 years, some of the highly recognized statisticians of our time (Leek, McShane, Gelman, Colquhoun, Nuijten and Goodman), wrote in Nature that:

> We need to appreciate that data analysis is not purely computational and algorithmic — it is a human behaviour....Researchers who hunt hard enough will turn up a result that fits statistical criteria — but their discovery will probably be a false positive.
>
> —*Five ways to fix statistics, Nature, 551, Nov 2017.*

Users of statistical (scientific) methods (software) are therefore not passive (objective) agents in their result. Therefore, it is necessary to actually understand the method, not just use it as a black box. The subjective experience gained by frequently using a method/software is not sufficient to claim an understanding of how the tool/method works and how relevant it is to the data and analysis. This kind of subjective experience is prone to serious misunderstandings about the data, what the software/statistical-method really does (especially as it gets more complicated), and thus the scientific interpretation of the result. This attitude is further encouraged through non-free software[3], poorly written (or non-existent) scientific software manuals, and non-reproducible papers[4]. This approach to scientific software and methods only helps in producing dogmas and an "*obscurantist faith in the expert's special skill, and in his personal knowledge and authority*"[5].

> Program or be programmed. Choose the former, and you gain access to the control panel of civilization. Choose the latter, and it could be the last real choice you get to make.
>
> —*Douglas Rushkoff. Program or be programmed, O/R Books (2010).*

It is obviously impractical for any one human being to gain the intricate knowledge explained above for every step of an analysis. On the other hand, scientific data can be large and numerous, for example images produced by telescopes in astronomy. This requires efficient algorithms. To make things worse, natural scientists have generally not been trained in the advanced software techniques, paradigms and architecture that are

[3] `https://www.gnu.org/philosophy/free-sw.html`

[4] Where the authors omit many of the analysis/processing "details" from the paper by arguing that they would make the paper too long/unreadable. However, software engineers have been dealing with such issues for a long time. There are thus software management solutions that allow us to supplement papers with all the details necessary to exactly reproduce the result. For example see zenodo.1163746 (`https://doi.org/10.5281/zenodo.1163746`) and zenodo.1164774 (`https://doi.org/10.5281/zenodo.1164774`) and this general discussion (`http://akhlaghi.org/reproducible-science.html`).

[5] Karl Popper. The logic of scientific discovery. 1959. Larger quote is given at the start of the PDF (for print) version of this book.

taught in computer science or engineering courses and thus used in most software. The GNU Astronomy Utilities are an effort to tackle this issue.

Gnuastro is not just a software, this book is as important to the idea behind Gnuastro as the source code (software). This book has tried to learn from the success of the "Numerical Recipes" book in educating those who are not software engineers and computer scientists but still heavy users of computational algorithms, like astronomers. There are two major differences.

The first difference is that Gnuastro's code and the background information are segregated: the code is moved within the actual Gnuastro software source code and the underlying explanations are given here in this book. In the source code, every non-trivial step is heavily commented and correlated with this book, it follows the same logic of this book, and all the programs follow a similar internal data, function and file structure, see Section 12.4 [Program source], page 451. Complementing the code, this book focuses on thoroughly explaining the concepts behind those codes (history, mathematics, science, software and usage advise when necessary) along with detailed instructions on how to run the programs. At the expense of frustrating "professionals" or "experts", this book and the comments in the code also intentionally avoid jargon and abbreviations. The source code and this book are thus intimately linked, and when considered as a single entity can be thought of as a real (an actual software accompanying the algorithms) "Numerical Recipes" for astronomy.

The second major, and arguably more important, difference is that "Numerical Recipes" does not allow you to distribute any code that you have learned from it. In other words, it does not allow you to release your software's source code if you have used their codes, you can only publicly release binaries (a black box) to the community. Therefore, while it empowers the privileged individual who has access to it, it exacerbates social ignorance. Exactly at the opposite end of the spectrum, Gnuastro's source code is released under the GNU general public license (GPL) and this book is released under the GNU free documentation license. You are therefore free to distribute any software you create using parts of Gnuastro's source code or text, or figures from this book, see Section 1.3 [Your rights], page 6.

With these principles in mind, Gnuastro's developers aim to impose the minimum requirements on you (in computer science, engineering and even the mathematics behind the tools) to understand and modify any step of Gnuastro if you feel the need to do so, see Section 12.1 [Why C programming language?], page 445, and Section 12.2 [Program design philosophy], page 447.

Without prior familiarity and experience with optics, it is hard to imagine how, Galileo could have come up with the idea of modifying the Dutch military telescope optics to use in astronomy. Astronomical objects could not be seen with the Dutch military design of the telescope. In other words, it is unlikely that Galileo could have asked a random optician to make modifications (not understood by Galileo) to the Dutch design, to do something no astronomer of the time took seriously. In the paradigm of the day, what could be the purpose of enlarging geometric spheres (planets) or points (stars)? In that paradigm only the position and movement of the heavenly bodies was important, and that had already been accurately studied (recently by Tycho Brahe).

In the beginning of his "The Sidereal Messenger" (published in 1610) he cautions the readers on this issue and *before* describing his results/observations, Galileo instructs us on how to build a suitable instrument. Without a detailed description of *how* he made his tools

and done his observations, no reasonable person would believe his results. Before he actually saw the moons of Jupiter, the mountains on the Moon or the crescent of Venus, Galileo was "evasive"[6] to Kepler. Science is defined by its tools/methods, *not* its raw results[7].

The same is true today: science cannot progress with a black box, or poorly released code. The source code of a research is the new (abstractified) communication language in science, understandable by humans *and* computers. Source code (in any programming language) is a language/notation designed to express all the details that would be too tedious/long/frustrating to report in spoken languages like English, similar to mathematic notation.

Today, the quality of the source code that goes into a scientific result (and the distribution of that code) is as critical to scientific vitality and integrity, as the quality of its written language/English used in publishing/distributing its paper. A scientific paper will not even be reviewed by any respectable journal if its written in a poor language/English. A similar level of quality assessment is thus increasingly becoming necessary regarding the codes/methods used to derive the results of a scientific paper.

Bjarne Stroustrup (creator of the C++ language) says: "*Without understanding software, you are reduced to believing in magic*". Ken Thomson (the designer or the Unix operating system) says "*I abhor a system designed for the 'user' if that word is a coded pejorative meaning 'stupid and unsophisticated'.*" Certainly no scientist (user of a scientific software) would want to be considered a believer in magic, or stupid and unsophisticated.

This can happen when scientists get too distant from the raw data and methods, and are mainly discussing results. In other words, when they feel they have tamed Nature into their own high-level (abstract) models (creations), and are mainly concerned with scaling up, or industrializing those results. Roughly five years before special relativity, and about two decades before quantum mechanics fundamentally changed Physics, Lord Kelvin is quoted as saying:

> There is nothing new to be discovered in physics now. All that remains is more and more precise measurement.
>
> —*William Thomson (Lord Kelvin), 1900*

A few years earlier Albert. A. Michelson made the following statement:

> The more important fundamental laws and facts of physical science have all been discovered, and these are now so firmly established that the possibility of their ever being supplanted in consequence of new discoveries is exceedingly remote.... Our future discoveries must be looked for in the sixth place of decimals.
>
> —*Albert. A. Michelson, dedication of Ryerson Physics Lab, U. Chicago 1894*

If scientists are considered to be more than mere "puzzle" solvers[8] (simply adding to the decimals of existing values or observing a feature in 10, 100, or 100000 more galaxies

---

[6] Galileo G. (Translated by Maurice A. Finocchiaro). *The essential Galileo*. Hackett publishing company, first edition, 2008.

[7] For example, take the following two results on the age of the universe: roughly 14 billion years (suggested by the current consensus of the standard model of cosmology) and less than 10,000 years (suggested from some interpretations of the Bible). Both these numbers are *results*. What distinguishes these two results, is the tools/methods that were used to derive them. Therefore, as the term "Scientific method" also signifies, a scientific statement it defined by its *method*, not its result.

[8] Thomas S. Kuhn. *The Structure of Scientific Revolutions*, University of Chicago Press, 1962.

or stars, as Kelvin and Michelson clearly believed), they cannot just passively sit back and uncritically repeat the previous (observational or theoretical) methods/tools on new data. Today there is a wealth of raw telescope images ready (mostly for free) at the finger tips of anyone who is interested with a fast enough internet connection to download them. The only thing lacking is new ways to analyze this data and dig out the treasure that is lying hidden in them to existing methods and techniques.

> New data that we insist on analyzing in terms of old ideas (that is, old models which are not questioned) cannot lead us out of the old ideas. However many data we record and analyze, we may just keep repeating the same old errors, missing the same crucially important things that the experiment was competent to find.
>
> —*Jaynes, Probability theory, the logic of science. Cambridge U. Press (2003).*

1.3 Your rights

The paragraphs below, in this section, belong to the GNU Texinfo[9] manual and are not written by us! The name "Texinfo" is just changed to "GNU Astronomy Utilities" or "Gnuastro" because they are released under the same licenses and it is beautifully written to inform you of your rights.

GNU Astronomy Utilities is "free software"; this means that everyone is free to use it and free to redistribute it on certain conditions. Gnuastro is not in the public domain; it is copyrighted and there are restrictions on its distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of Gnuastro that they might get from you.

Specifically, we want to make sure that you have the right to give away copies of the programs that relate to Gnuastro, that you receive the source code or else can get it if you want it, that you can change these programs or use pieces of them in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, we have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the Gnuastro related programs, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for our own protection, we must make certain that everyone finds out that there is no warranty for the programs that relate to Gnuastro. If these programs are modified by someone else and passed on, we want their recipients to know that what they have is not what we distributed, so that any problems introduced by others will not reflect on our reputation.

The full text of the licenses for the Gnuastro book and software can be respectively found in Appendix D [GNU Gen. Pub. License v3], page 482[10] and Appendix C [GNU Free Doc. License], page 474[11].

---

[9] Texinfo is the GNU documentation system. It is used to create this book in all the various formats.

[10] Also available in `http://www.gnu.org/copyleft/gpl.html`

[11] Also available in `http://www.gnu.org/copyleft/fdl.html`

1.4 Naming convention

Gnuastro is a package of independent programs and a collection of libraries, here we are mainly concerned with the programs. Each program has an official name which consists of one or two words, describing what they do. The latter are printed with no space, for example NoiseChisel or Crop. On the command-line, you can run them with their executable names which start with an `ast` and might be an abbreviation of the official name, for example `astnoisechisel` or `astcrop`, see Section 3.3.1.3 [Executable names], page 83.

We will use "ProgramName" for a generic official program name and `astprogname` for a generic executable name. In this book, the programs are classified based on what they do and thoroughly explained. An alphabetical list of the programs that are installed on your system with this installation are given in Appendix A [Gnuastro programs list], page 467. That list also contains the executable names and version numbers along with a one line description.

1.5 Version numbering

Gnuastro can have two formats of version numbers, for official and unofficial releases. Official Gnuastro releases are announced on the `info-gnuastro` mailing list, they have a version control tag in Gnuastro's development history, and their version numbers are formatted like "`A.B`". `A` is a major version number, marking a significant planned achievement (for example see Section 1.5.1 [GNU Astronomy Utilities 1.0], page 8), while `B` is a minor version number, see below for more on the distinction. Note that the numbers are not decimals, so version 2.34 is much more recent than version 2.5, which is not equal to 2.50.

Gnuastro also allows a unique version number for unofficial releases. Unofficial releases can mark any point in Gnuastro's development history. This is done to allow astronomers to easily use any point in the version controlled history for their data-analysis and research publication. See Section 3.2.2 [Version controlled source], page 72, for a complete introduction. This section is not just for developers and is intended to straightforward and easy to read, so please have a look if you are interested in the cutting-edge. This unofficial version number is a meaningful and easy to read string of characters, unique to that particular point of history. With this feature, users can easily stay up to date with the most recent bug fixes and additions that are committed between official releases.

The unofficial version number is formatted like: `A.B.C-D`. `A` and `B` are the most recent official version number. `C` is the number of commits that have been made after version `A.B`. `D` is the first 4 or 5 characters of the commit hash number[12]. Therefore, the unofficial version number '`3.92.8-29c8`', corresponds to the 8th commit after the official version `3.92` and its commit hash begins with `29c8`. The unofficial version number is sort-able (unlike the raw hash) and as shown above is descriptive of the state of the unofficial release. Of course an official release is preferred for publication (since its tarballs are easily available and it has gone through more tests, making it more stable), so if an official release is announced prior to your publication's final review, please consider updating to the official release.

The major version number is set by a major goal which is defined by the developers and user community before hand, for example see Section 1.5.1 [GNU Astronomy Utilities

[12] Each point in Gnuastro's history is uniquely identified with a 40 character long hash which is created from its contents and previous history for example: `5b17501d8f29ba3cd610673261e6e2229c846d35`. So the string `D` in the version for this commit could be `5b17`, or `5b175`.

1.0], page 8. The incremental work done in minor releases are commonly small steps in achieving the major goal. Therefore, there is no limit on the number of minor releases and the difference between the (hypothetical) versions 2.927 and 3.0 can be a small (negligible to the user) improvement that finalizes the defined goals.

1.5.1 GNU Astronomy Utilities 1.0

Currently (prior to Gnuastro 1.0), the aim of Gnuastro is to have a complete system for data manipulation and analysis at least similar to IRAF[13]. So an astronomer can take all the standard data analysis steps (starting from raw data to the final reduced product and standard post-reduction tools) with the various programs in Gnuastro.

The maintainers of each camera or detector on a telescope can provide a completely transparent shell script or Makefile to the observer for data analysis. This script can set configuration files for all the required programs to work with that particular camera. The script can then run the proper programs in the proper sequence. The user/observer can easily follow the standard shell script to understand (and modify) each step and the parameters used easily. Bash (or other modern GNU/Linux shell scripts) is powerful and made for this gluing job. This will simultaneously improve performance and transparency. Shell scripting (or Makefiles) are also basic constructs that are easy to learn and readily available as part of the Unix-like operating systems. If there is no program to do a desired step, Gnuastro's libraries can be used to build specific programs.

The main factor is that all observatories or projects can freely contribute to Gnuastro and all simultaneously benefit from it (since it doesn't belong to any particular one of them), much like how for-profit organizations (for example RedHat, or Intel and many others) are major contributors to free and open source software for their shared benefit. Gnuastro's copyright has been fully awarded to GNU, so it doesn't belong to any particular astronomer or astronomical facility or project.

1.6 New to GNU/Linux?

Some astronomers initially install and use a GNU/Linux operating system because their necessary tools can only be installed in this environment. However, the transition is not necessarily easy. To encourage you in investing the patience and time to make this transition, and actually enjoy it, we will first start with a basic introduction to GNU/Linux operating systems. Afterwards, in Section 1.6.1 [Command-line interface], page 9, we'll discuss the wonderful benefits of the command-line interface, how it beautifully complements the graphic user interface, and why it is worth the (apparently steep) learning curve. Finally a complete chapter (Chapter 2 [Tutorials], page 16) is devoted to real world scenarios of using Gnuastro (on the command-line). Therefore if you don't yet feel comfortable with the command-line we strongly recommend going through that chapter after finishing this section.

You might have already noticed that we are not using the name "Linux", but "GNU/Linux". Please take the time to have a look at the following essays and FAQs for a complete understanding of this very important distinction.

- `https://www.gnu.org/gnu/gnu-users-never-heard-of-gnu.html`

[13] `http://iraf.noao.edu/`

- https://www.gnu.org/gnu/linux-and-gnu.html
- https://www.gnu.org/gnu/why-gnu-linux.html
- https://www.gnu.org/gnu/gnu-linux-faq.html

In short, the Linux kernel[14] is built using the GNU C library (glibc) and GNU compiler collection (gcc). The Linux kernel software alone is just a means for other software to access the hardware resources, it is useless alone: to say "running Linux", is like saying "driving your carburetor".

To have an operating system, you need lower-level (to build the kernel), and higher-level (to use it) software packages. The majority of such software in most Unix-like operating systems are GNU software: "the whole system is basically GNU with Linux loaded". Therefore to acknowledge GNU's instrumental role in the creation and usage of the Linux kernel and the operating systems that use it, we should call these operating systems "GNU/Linux".

1.6.1 Command-line interface

One aspect of Gnuastro that might be a little troubling to new GNU/Linux users is that (at least for the time being) it only has a command-line user interface (CLI). This might be contrary to the mostly graphical user interface (GUI) experience with proprietary operating systems. Since the various actions available aren't always on the screen, the command-line interface can be complicated, intimidating, and frustrating for a first-time user. This is understandable and also experienced by anyone who started using the computer (from childhood) in a graphical user interface (this includes most of Gnuastro's authors). Here we hope to convince you of the unique benefits of this interface which can greatly enhance your productivity while complementing your GUI experience.

Through GNOME 3[15], most GNU/Linux based operating systems now have an advanced and useful GUI. Since the GUI was created long after the command-line, some wrongly consider the command line to be obsolete. Both interfaces are useful for different tasks. For example you can't view an image, video, pdf document or web page on the command-line. On the other hand you can't reproduce your results easily in the GUI. Therefore they should not be regarded as rivals but as complementary user interfaces, here we will outline how the CLI can be useful in scientific programs.

You can think of the GUI as a veneer over the CLI to facilitate a small subset of all the possible CLI operations. Each click you do on the GUI, can be thought of as internally running a different CLI command. So asymptotically (if a good designer can design a GUI which is able to show you all the possibilities to click on) the GUI is only as powerful as the command-line. In practice, such graphical designers are very hard to find for every program, so the GUI operations are always a subset of the internal CLI commands. For programs that are only made for the GUI, this results in not including lots of potentially useful operations. It also results in 'interface design' to be a crucially important part of any GUI program. Scientists don't usually have enough resources to hire a graphical designer, also the complexity of the GUI code is far more than CLI code, which is harmful for a scientific software, see Section 1.2 [Science and its tools], page 2.

For programs that have a GUI, one action on the GUI (moving and clicking a mouse, or tapping a touchscreen) might be more efficient and easier than its CLI counterpart (typing

[14] In Unix-like operating systems, the kernel connects software and hardware worlds.

[15] http://www.gnome.org/

the program name and your desired configuration). However, if you have to repeat that same action more than once, the GUI will soon become frustrating and prone to errors. Unless the designers of a particular program decided to design such a system for a particular GUI action, there is no general way to run any possible series of actions automatically on the GUI.

On the command-line, you can run any series of of actions which can come from various CLI capable programs you have decided your self in any possible permutation with one command[16]. This allows for much more creativity and exact reproducibility that is not possible to a GUI user. For technical and scientific operations, where the same operation (using various programs) has to be done on a large set of data files, this is crucially important. It also allows exact reproducibility which is a foundation principle for scientific results. The most common CLI (which is also known as a shell) in GNU/Linux is GNU Bash, we strongly encourage you to put aside several hours and go through this beautifully explained web page: `https://flossmanuals.net/command-line/`. You don't need to read or even fully understand the whole thing, only a general knowledge of the first few chapters are enough to get you going.

Since the operations in the GUI are limited and they are visible, reading a manual is not that important in the GUI (most programs don't even have any!). However, to give you the creative power explained above, with a CLI program, it is best if you first read the manual of any program you are using. You don't need to memorize any details, only an understanding of the generalities is needed. Once you start working, there are more easier ways to remember a particular option or operation detail, see Section 4.3 [Getting help], page 109.

To experience the command-line in its full glory and not in the GUI terminal emulator, press the following keys together: `CTRL+ALT+F4`[17] to access the virtual console. To return back to your GUI, press the same keys above replacing `F4` with `F7` (or `F1`, or `F2`, depending on your GNU/Linux distribution). In the virtual console, the GUI, with all its distracting colors and information, is gone. Enabling you to focus entirely on your actual work.

For operations that use a lot of your system's resources (processing a large number of large astronomical images for example), the virtual console is the place to run them. This is because the GUI is not competing with your research work for your system's RAM and CPU. Since the virtual consoles are completely independent, you can even log out of your GUI environment to give even more of your hardware resources to the programs you are running and thus reduce the operating time.

Since it uses far less system resources, the CLI is also convenient for remote access to your computer. Using secure shell (SSH) you can log in securely to your system (similar to the virtual console) from anywhere even if the connection speeds are low. There are apps for smart phones and tablets which allow you to do this.

---

[16] By writing a shell script and running it, for example see the tutorials in Chapter 2 [Tutorials], page 16.

[17] Instead of `F4`, you can use any of the keys from `F1` to `F6` for different virtual consoles depending on your GNU/Linux distribution, try them all out. You can also run a separate GUI from within this console if you want to.

1.7 Report a bug

According to Wikipedia "a software bug is an error, flaw, failure, or fault in a computer program or system that causes it to produce an incorrect or unexpected result, or to behave in unintended ways". So when you see that a program is crashing, not reading your input correctly, giving the wrong results, or not writing your output correctly, you have found a bug. In such cases, it is best if you report the bug to the developers. The programs will also inform you if known impossible situations occur (which are caused by something unexpected) and will ask the users to report the bug issue.

Prior to actually filing a bug report, it is best to search previous reports. The issue might have already been found and even solved. The best place to check if your bug has already been discussed is the bugs tracker on Section 12.9 [Gnuastro project webpage], page 459, at `https://savannah.gnu.org/bugs/?group=gnuastro`. In the top search fields (under "Display Criteria") set the "Open/Closed" drop-down menu to "Any" and choose the respective program or general category of the bug in "Category" and click the "Apply" button. The results colored green have already been solved and the status of those colored in red is shown in the table.

Recently corrected bugs are probably not yet publicly released because they are scheduled for the next Gnuastro stable release. If the bug is solved but not yet released and it is an urgent issue for you, you can get the version controlled source and compile that, see Section 3.2.2 [Version controlled source], page 72.

To solve the issue as readily as possible, please follow the following to guidelines in your bug report. The How to Report Bugs Effectively (`http://www.chiark.greenend.org.uk/~sgtatham/bugs.html`) and How To Ask Questions The Smart Way (`http://catb.org/~esr/faqs/smart-questions.html`) essays also provide some good generic advice for all software (don't contact their authors for Gnuastro's problems). Mastering the art of giving good bug reports (like asking good questions) can greatly enhance your experience with any free and open source software. So investing the time to read through these essays will greatly reduce your frustration after you see something doesn't work the way you feel it is supposed to for a large range of software, not just Gnuastro.

Be descriptive

Please provide as many details as possible and be very descriptive. Explain what you expected and what the output was: it might be that your expectation was wrong. Also please clearly state which sections of the Gnuastro book (this book), or other references you have studied to understand the problem. This can be useful in correcting the book (adding links to likely places where users will check). But more importantly, it will be encouraging for the developers, since you are showing how serious you are about the problem and that you have actually put some thought into it. "To be able to ask a question clearly is two-thirds of the way to getting it answered." – John Ruskin (1819-1900).

Individual and independent bug reports

If you have found multiple bugs, please send them as separate (and independent) bugs (as much as possible). This will significantly help us in managing and resolving them sooner.

Reproducible bug reports

If we cannot exactly reproduce your bug, then it is very hard to resolve it. So please send us a Minimal working example[18] along with the description. For example in running a program, please send us the full command-line text and the output with the `-P` option, see Section 4.1.2.3 [Operating mode options], page 100. If it is caused only for a certain input, also send us that input file. In case the input FITS is large, please use Crop to only crop the problematic section and make it as small as possible so it can easily be uploaded and downloaded and not waste the archive's storage, see Section 6.1 [Crop], page 151.

There are generally two ways to inform us of bugs:

- Send a mail to `bug-gnuastro@gnu.org`. Any mail you send to this address will be distributed through the bug-gnuastro mailing list[19]. This is the simplest way to send us bug reports. The developers will then register the bug into the project webpage (next choice) for you.
- Use the Gnuastro project webpage at `https://savannah.gnu.org/projects/gnuastro/`: There are two ways to get to the submission page as listed below. Fill in the form as described below and submit it (see Section 12.9 [Gnuastro project webpage], page 459, for more on the project webpage).
  - Using the top horizontal menu items, immediately under the top page title. Hovering your mouse on "Support" will open a drop-down list. Select "Submit new".
  - In the main body of the page, under the "Communication tools" section, click on "Submit new item".

Once the items have been registered in the mailing list or webpage, the developers will add it to either the "Bug Tracker" or "Task Manager" trackers of the Gnuastro project webpage. These two trackers can only be edited by the Gnuastro project developers, but they can be browsed by anyone, so you can follow the progress on your bug. You are most welcome to join us in developing Gnuastro and fixing the bug you have found maybe a good starting point. Gnuastro is designed to be easy for anyone to develop (see Section 1.2 [Science and its tools], page 2) and there is a full chapter devoted to developing it: Chapter 12 [Developing], page 445.

1.8 Suggest new feature

We would always be happy to hear of suggested new features. For every program there are already lists of features that we are planning to add. You can see the current list of plans from the Gnuastro project webpage at `https://savannah.gnu.org/projects/gnuastro/` and following "Tasks"→"Browse" on the horizontal menu at the top of the page immediately under the title, see Section 12.9 [Gnuastro project webpage], page 459. If you want to request a feature to an existing program, click on the "Display Criteria" above the list and under "Category", choose that particular program. Under "Category" you can also see the existing suggestions for new programs or other cases like installation, documentation or libraries. Also be sure to set the "Open/Closed" value to "Any".

[18] `http://en.wikipedia.org/wiki/Minimal_Working_Example`

[19] `https://lists.gnu.org/mailman/listinfo/bug-gnuastro`

If the feature you want to suggest is not already listed in the task manager, then follow the steps that are fully described in Section 1.7 [Report a bug], page 11. Please have in mind that the developers are all busy with their own astronomical research, and implementing existing "task"s to add or resolving bugs. Gnuastro is a volunteer effort and none of the developers are paid for their hard work. So, although we will try our best, please don't not expect that your suggested feature be immediately included (with the next release of Gnuastro).

The best person to apply the exciting new feature you have in mind is you, since you have the motivation and need. In fact Gnuastro is designed for making it as easy as possible for you to hack into it (add new features, change existing ones and so on), see Section 1.2 [Science and its tools], page 2. Please have a look at the chapter devoted to developing (Chapter 12 [Developing], page 445) and start applying your desired feature. Once you have added it, you can use it for your own work and if you feel you want others to benefit from your work, you can request for it to become part of Gnuastro. You can then join the developers and start maintaining your own part of Gnuastro. If you choose to take this path of action please contact us before hand (Section 1.7 [Report a bug], page 11) so we can avoid possible duplicate activities and get interested people in contact.

Gnuastro is a collection of low level programs: As described in Section 12.2 [Program design philosophy], page 447, a founding principle of Gnuastro is that each library or program should be basic and low-level. High level jobs should be done by running the separate programs or using separate functions in succession through a shell script or calling the libraries by higher level functions, see the examples in Chapter 2 [Tutorials], page 16. So when making the suggestions please consider how your desired job can best be broken into separate steps and modularized.

1.9 Announcements

Gnuastro has a dedicated mailing list for making announcements (`info-gnuastro`). Anyone can subscribe to this mailing list. Anytime there is a new stable or test release, an email will be circulated there. The email contains a summary of the overall changes along with a detailed list (from the `NEWS` file). This mailing list is thus the best way to stay up to date with new releases, easily learn about the updated/new features, or dependencies (see Section 3.1 [Dependencies], page 61).

To subscribe to this list, please visit `https://lists.gnu.org/mailman/listinfo/info-gnuastro`. Traffic (number of mails per unit time) in this list is designed to be low: only a handful of mails per year. Previous announcements are available on its archive (`http://lists.gnu.org/archive/html/info-gnuastro/`).

1.10 Conventions

In this book we have the following conventions:

- All commands that are to be run on the shell (command-line) prompt as the user start with a `$`. In case they must be run as a super-user or system administrator, they will start with a single `#`. If the command is in a separate line and next line `is also in the code type face`, but doesn't have any of the `$` or `#` signs, then it is the output of

the command after it is run. As a user, you don't need to type those lines. A line that starts with `##` is just a comment for explaining the command to a human reader and must not be typed.

- If the command becomes larger than the page width a `\` is inserted in the code. If you are typing the code by hand on the command-line, you don't need to use multiple lines or add the extra space characters, so you can omit them. If you want to copy and paste these examples (highly discouraged!) then the `\` should stay.

  The `\` character is a shell escape character which is used commonly to make characters which have special meaning for the shell loose that special place (the shell will not treat them specially if there is a `\` behind them). When it is a last character in a line (the next character is a new-line character) the new-line character looses its meaning an the shell sees it as a simple white-space character, enabling you to use multiple lines to write your commands.

1.11 Acknowledgments

Gnuastro would not have been possible without scholarships and grants from several funding institutions. We thus ask that if you used Gnuastro in any of your papers/reports, please add the proper citation and acknowledge this instrumental support. For details of which papers to cite (may be different for different programs) and get the acknowledgment statement to include in your paper, please run the relevant programs with the common `--cite` option like the example commands below (for more on `--cite`, please see Section 4.1.2.3 [Operating mode options], page 100).

```
$ astnoisechisel --cite
$ astmkcatalog --cite
```

Here, we'll acknowledge all the institutions (and their grants) along with the people who helped make Gnuastro possible. The full list of Gnuastro authors is available at the start of this book and the `AUTHORS` file in the source code (both are generated automatically from the version controlled history). The plain text file `THANKS`, which is also distributed along with the source code, contains the list of people and institutions who played an indirect role in Gnuastro (not committed any code in the Gnuastro version controlled history).

The Japanese Ministry of Education, Culture, Sports, Science, and Technology (MEXT) scholarship for Mohammad Akhlaghi's Masters and PhD degree in Tohoku University Astronomical Institute had an instrumental role in the long term learning and planning that made the idea of Gnuastro possible. The very critical view points of Professor Takashi Ichikawa (Mohammad's adviser) were also instrumental in the initial ideas and creation of Gnuastro. Afterwards, the European Research Council (ERC) advanced grant 339659-MUSICOS (Principal investigator: Roland Bacon) was vital in the growth and expansion of Gnuastro. Working with Roland at the Centre de Recherche Astrophysique de Lyon (CRAL), enabled a thorough re-write of the core functionality of all libraries and programs, turning Gnuastro into the large collection of generic programs and libraries it is today. Work on improving Gnuastro and making it mature is now continuing primarily in the Instituto de Astrofisica de Canarias (IAC) and in particular in collaboration with Johan Knapen and Ignacio Trujillo.

In general, we would like to gratefully thank the following people for their useful and constructive comments and suggestions (in alphabetical order by family name): Valentina

Abril-melgarejo, Marjan Akbari, Roland Bacon, Roberto Baena Gall\'e, Karl Berry, Leindert Boogaard, Nicolas Bouché, Fernando Buitrago, Adrian Bunk, Rosa Calvi, Nushkia Chamba, Benjamin Clement, Nima Dehdilani, Antonio Diaz Diaz, Pierre-Alain Duc, Elham Eftekhari, Gaspar Galaz, Thérèse Godefroy, Madusha Gunawardhana, Stephen Hamer, Takashi Ichikawa, Raúl Infante Sainz, Brandon Invergo, Oryna Ivashtenko, Aurélien Jarno, Lee Kelvin, Brandon Kelly, Mohammad-Reza Khellat, Johan Knapen, Geoffry Krouchi, Floriane Leclercq, Alan Lefor, Guillaume Mahler, Juan Molina Tobar, Francesco Montanari, Dmitrii Oparin, Bertrand Pain, William Pence, Mamta Pommier, Bob Proulx, Teymoor Saifollahi, Yahya Sefidbakht, Alejandro Serrano Borlaff, Jenny Sorce, Lee Spitler, Richard Stallman, Michael Stein, Ole Streicher, Alfred M. Szmidt, Michel Tallon, Juan C. Tello, Éric Thiébaut, Ignacio Trujillo, David Valls-Gabaud, Aaron Watkins, Christopher Willmer, Sara Yousefi Taemeh, Johannes Zabl. The GNU French Translation Team is also managing the French version of the top Gnuastro webpage which we highly appreciate. Finally we should thank all the (sometimes anonymous) people in various online forums which patiently answered all our small (but important) technical questions.

All work on Gnuastro has been voluntary, but the authors are most grateful to the following institutions (in chronological order) for hosting us in our research. Where necessary, these institutions have disclaimed any ownership of the parts of Gnuastro that were developed there, thus insuring the freedom of Gnuastro for the future (see Section 12.11.1 [Copyright assignment], page 462). We highly appreciate their support for free software, and thus free science, and therefore a free society.

Tohoku University Astronomical Institute, Sendai, Japan.
University of Salento, Lecce, Italy.
Centre de Recherche Astrophysique de Lyon (CRAL), Lyon, France.
Instituto de Astrofisica de Canarias (IAC), Tenerife, Spain.

2 Tutorials

To help new users have a smooth and easy start with Gnuastro, in this chapter several thoroughly elaborated tutorials, or cookbooks, are provided. These tutorials demonstrate the capabilities of different Gnuastro programs and libraries, along with tips and guidelines for the best practices of using them in various realistic situations.

We strongly recommend going through these tutorials to get a good feeling of how the programs are related (built in a modular design to be used together in a pipeline), very similar to the core Unix-based programs that they were modeled on. Therefore these tutorials will greatly help in optimally using Gnuastro's programs (and generally, the Unix-like command-line environment) effectively for your research.

In Section 2.1 [Sufi simulates a detection], page 17, we'll start with a fictional[1] tutorial explaining how Abd al-rahman Sufi (903 – 986 A.D., the first recorded description of "nebulous" objects in the heavens is attributed to him) could have used some of Gnuastro's programs for a realistic simulation of his observations and see if his detection of nebulous objects was trust-able. Because all conditions are under control in a simulated/mock environment/dataset, mock datasets can be a valuable tool to inspect the limitations of your data analysis and processing. But they need to be as realistic as possible, so the first tutorial is dedicated to this important step of an analysis.

The next two tutorials (Section 2.2 [General program usage tutorial], page 24, and Section 2.3 [Detecting large extended targets], page 47) use real input datasets from some of the deep Hubble Space Telescope (HST) images and the Sloan Digital Sky Survey (SDSS) respectively. Their aim is to demonstrate some real-world problems that many astronomers often face and how they can be be solved with Gnuastro's programs.

The ultimate aim of Section 2.2 [General program usage tutorial], page 24, is to detect galaxies in a deep HST image, measure their positions and brightness and select those with the strongest colors. In the process, it takes many detours to introduce you to the useful capabilities of many of the programs. So please be patient in reading it. If you don't have much time and can only try one of the tutorials, we recommend this one.

Section 2.3 [Detecting large extended targets], page 47, deals with a major problem in astronomy: effectively detecting the faint outer wings of bright (and large) nearby galaxies to extremely low surface brightness levels (roughly 1/20th of the local noise level in the example discussed). Besides the interesting scientific questions in these low-surface brightness features, failure to properly detect them will bias the measurements of the background objects and the survey's noise estimates. This is an important issue, especially in wide

[1] The two historically motivated tutorials (Section 2.1 [Sufi simulates a detection], page 17, and Section 2.4 [Hubble visually checks and classifies his catalog], page 57) are not intended to be a historical reference (the historical facts of this fictional tutorial used Wikipedia as a reference). This form of presenting a tutorial was influenced by the PGF/TikZ and Beamer manuals. They are both packages in in TEX and LATEX, the first is a high-level vector graphic programming environment, while with the second you can make presentation slides. On a similar topic, there are also some nice words of wisdom for Unix-like systems called Rootless Root (`http://catb.org/esr/writings/unix-koans`). These also have a similar style but they use a mythical figure named Master Foo. If you already have some experience in Unix-like systems, you will definitely find these Unix Koans entertaining/educative.

surveys. Because bright/large galaxies and stars[2], cover a significant fraction of the survey area.

Finally, in Section 2.4 [Hubble visually checks and classifies his catalog], page 57, we go into the historical/fictional world again to see how Hubble could have used Gnuastro's programs to visually check and classify a sample of galaxies which ultimately lead him to the "Hubble fork" classification of galaxy morphologies.

In these tutorials, we have intentionally avoided too many cross references to make it more easy to read. For more information about a particular program, you can visit the section with the same name as the program in this book. Each program section in the subsequent chapters starts by explaining the general concepts behind what it does, for example see Section 6.3 [Convolve], page 177. If you only want practical information on running a program, for example its options/configuration, input(s) and output(s), please consult the subsection titled "Invoking ProgramName", for example see Section 7.2.2 [Invoking NoiseChisel], page 230. For an explanation of the conventions we use in the example codes through the book, please see Section 1.10 [Conventions], page 13.

2.1 Sufi simulates a detection

It is the year 953 A.D. and Sufi[3] is in Shiraz as a guest astronomer. He had come there to use the advanced 123 centimeter astrolabe for his studies on the Ecliptic. However, something was bothering him for a long time. While mapping the constellations, there were several non-stellar objects that he had detected in the sky, one of them was in the Andromeda constellation. During a trip he had to Yemen, Sufi had seen another such object in the southern skies looking over the Indian ocean. He wasn't sure if such cloud-like non-stellar objects (which he was the first to call 'Sahābi' in Arabic or 'nebulous') were real astronomical objects or if they were only the result of some bias in his observations. Could such diffuse objects actually be detected at all with his detection technique?

He still had a few hours left until nightfall (when he would continue his studies on the ecliptic) so he decided to find an answer to this question. He had thoroughly studied Claudius Ptolemy's (90 – 168 A.D) Almagest and had made lots of corrections to it, in particular in measuring the brightness. Using his same experience, he was able to measure a magnitude for the objects and wanted to simulate his observation to see if a simulated object with the same brightness and size could be detected in a simulated noise with the same detection technique. The general outline of the steps he wants to take are:

1. Make some mock profiles in an over-sampled image. The initial mock image has to be over-sampled prior to convolution or other forms of transformation in the image. Through his experiences, Sufi knew that this is because the image of heavenly bodies is actually transformed by the atmosphere or other sources outside the atmosphere (for example gravitational lenses) prior to being sampled on an image. Since that transformation occurs on a continuous grid, to best approximate it, he should do all the work on a finer pixel grid. In the end he can re-sample the result to the initially desired grid size.

[2] Stars also have similarly large and extended wings due to the point spread function, see Section 8.1.1.2 [Point spread function], page 285.

[3] Abd al-rahman Sufi (903 – 986 A.D.), also known in Latin as Azophi was an Iranian astronomer. His manuscript "Book of fixed stars" contains the first recorded observations of the Andromeda galaxy, the Large Magellanic Cloud and seven other non-stellar or 'nebulous' objects.

2. Convolve the image with a point spread function (PSF, see Section 8.1.1.2 [Point spread function], page 285) that is over-sampled to the same resolution as the mock image. Since he wants to finish in a reasonable time and the PSF kernel will be very large due to oversampling, he has to use frequency domain convolution which has the side effect of dimming the edges of the image. So in the first step above he also has to build the image to be larger by at least half the width of the PSF convolution kernel on each edge.
3. With all the transformations complete, the image should be re-sampled to the same size of the pixels in his detector.
4. He should remove those extra pixels on all edges to remove frequency domain convolution artifacts in the final product.
5. He should add noise to the (until now, noise-less) mock image. After all, all observations have noise associated with them.

Fortunately Sufi had heard of GNU Astronomy Utilities from a colleague in Isfahan (where he worked) and had installed it on his computer a year before. It had tools to do all the steps above. He had used MakeProfiles before, but wasn't sure which columns he had chosen in his user or system wide configuration files for which parameters, see Section 4.2 [Configuration files], page 106. So to start his simulation, Sufi runs MakeProfiles with the `-P` option to make sure what columns in a catalog MakeProfiles currently recognizes and the output image parameters. In particular, Sufi is interested in the recognized columns (shown below).

```
$ astmkprof -P

[[[ ... Truncated lines ... ]]]

# Output:
 type         float32     # Type of output: e.g., int16, float32, etc...
 mergedsize   1000,1000   # Number of pixels along first FITS axis.
 oversample   5           # Scale of oversampling (>0 and odd).

[[[ ... Truncated lines ... ]]]

# Columns, by info (see '--searchin'), or number (starting from 1):
 ccol         2           # Center along first FITS axis (horizontal).
 ccol         3           # Center along second FITS axis (vertical).
 fcol         4           # sersic (1), moffat (2), gaussian (3),
                          # point (4), flat (5), circumference (6).
 rcol         5           # Effective radius or FWHM in pixels.
 ncol         6           # Sersic index or Moffat beta.
 pcol         7           # Position angle.
 qcol         8           # Axis ratio.
 mcol         9           # Magnitude.
 tcol         10          # Truncation in units of radius or pixels.

[[[ ... Truncated lines ... ]]]
```

In Gnuastro, column counting starts from 1, so the columns are ordered such that the first column (number 1) can be an ID he specifies for each object (and MakeProfiles ignores), each subsequent column is used for another property of the profile. It is also possible to use column names for the values of these options and change these defaults, but Sufi preferred to stick to the defaults. Fortunately MakeProfiles has the capability to also make the PSF which is to be used on the mock image and using the `--prepforconv` option, he can also make the mock image to be larger by the correct amount and all the sources to be shifted by the correct amount.

For his initial check he decides to simulate the nebula in the Andromeda constellation. The night he was observing, the PSF had roughly a FWHM of about 5 pixels, so as the first row (profile), he defines the PSF parameters and sets the radius column (`rcol` above, fifth column) to `5.000`, he also chooses a Moffat function for its functional form. Remembering how diffuse the nebula in the Andromeda constellation was, he decides to simulate it with a mock Sérsic index 1.0 profile. He wants the output to be 500 pixels by 500 pixels, so he puts the mock profile in the center. Looking at his drawings of it, he decides a reasonable effective radius for it would be 40 pixels on this image pixel scale, he sets the axis ratio and position angle to approximately correct values too and finally he sets the total magnitude of the profile to 3.44 which he had accurately measured. Sufi also decides to truncate both the mock profile and PSF at 5 times the respective radius parameters. In the end he decides to put four stars on the four corners of the image at very low magnitudes as a visual scale.

Using all the information above, he creates the catalog of mock profiles he wants in a file named `cat.txt` (short for catalog) using his favorite text editor and stores it in a directory named `simulationtest` in his home directory. [The `cat` command prints the contents of a file, short for concatenation. So please copy-paste the lines after "`cat cat.txt`" into `cat.txt` when the editor opens in the steps above it, note that there are 7 lines, first one starting with `#`]:

```
$ mkdir ~/simulationtest
$ cd ~/simulationtest
$ pwd
/home/rahman/simulationtest
$ emacs cat.txt
$ ls
cat.txt
$ cat cat.txt
# Column 4: PROFILE_NAME [,str7] Radial profile's functional name
 1  0.0000   0.0000  moffat  5.000  4.765  0.0000  1.000  30.000  5.000
 2  250.00   250.00  sersic  40.00  1.000  -25.00  0.400  3.4400  5.000
 3  50.000   50.000  point   0.000  0.000  0.0000  0.000  9.0000  0.000
 4  450.00   50.000  point   0.000  0.000  0.0000  0.000  9.2500  0.000
 5  50.000   450.00  point   0.000  0.000  0.0000  0.000  9.5000  0.000
 6  450.00   450.00  point   0.000  0.000  0.0000  0.000  9.7500  0.000
```

The zero-point magnitude for his observation was 18. Now he has all the necessary parameters and runs MakeProfiles with the following command:

```
$ astmkprof --prepforconv --mergedsize=500,500 --zeropoint=18.0 cat.txt
MakeProfiles started on Sat Oct  6 16:26:56 953
```

```
  - 6 profiles read from cat.txt
  - Random number generator (RNG) type: mt19937
  - Using 8 threads.
  ---- row 2 complete, 5 left to go
  ---- row 3 complete, 4 left to go
  ---- row 4 complete, 3 left to go
  ---- row 5 complete, 2 left to go
  ---- ./0_cat.fits created.
  ---- row 0 complete, 1 left to go
  ---- row 1 complete, 0 left to go
  - ./cat.fits created.                                0.041651 seconds
MakeProfiles finished in 0.267234 seconds

$ls
0_cat.fits  cat.fits  cat.txt
```

The file `0_cat.fits` is the PSF Sufi had asked for and `cat.fits` is the image containing the other 5 objects. The PSF is now available to him as a separate file for the convolution step. While he was preparing the catalog, one of his students approached him and was also following the steps. When he opened the image, the student was surprised to see that all the stars are only one pixel and not in the shape of the PSF as we see when we image the sky at night. So Sufi explained to him that the stars will take the shape of the PSF after convolution and this is how they would look if we didn't have an atmosphere or an aperture when we took the image. The size of the image was also surprising for the student, instead of 500 by 500, it was 2630 by 2630 pixels. So Sufi had to explain why oversampling is important for parts of the image where the flux change is significant over a pixel. Sufi then explained to him that after convolving we will re-sample the image to get our originally desired size. To convolve the image, Sufi ran the following command:

```
$ astconvolve --kernel=0_cat.fits cat.fits
Convolve started on Mon Apr  6 16:35:32 953
  - Using 8 CPU threads.
  - Input: cat.fits (hdu: 1)
  - Kernel: 0_cat.fits (hdu: 1)
  - Input and Kernel images padded.                    0.075541 seconds
  - Images converted to frequency domain.              6.728407 seconds
  - Multiplied in the frequency domain.                0.040659 seconds
  - Converted back to the spatial domain.              3.465344 seconds
  - Padded parts removed.                              0.016767 seconds
Convolve finished in:  10.422161 seconds

$ls
0_cat.fits  cat_convolved.fits  cat.fits  cat.txt
```

When convolution finished, Sufi opened the `cat_convolved.fits` file and showed the effect of convolution to his student and explained to him how a PSF with a larger FWHM would make the points even wider. With the convolved image ready, they were prepared to re-sample it to the original pixel scale Sufi had planned [from the `$ astmkprof -P` command above, recall that MakeProfiles had over-sampled the image by 5 times]. Sufi explained

the basic concepts of warping the image to his student and ran Warp with the following command:

```
$ astwarp --scale=1/5 --centeroncorner cat_convolved.fits
Warp started on Mon Apr  6 16:51:59 953
 Using 8 CPU threads.
 Input: cat_convolved.fits (hdu: 1)
 matrix:
        0.2000   0.0000   0.4000
        0.0000   0.2000   0.4000
        0.0000   0.0000   1.0000

$ ls
0_cat.fits          cat_convolved_scaled.fits     cat.txt
cat_convolved.fits  cat.fits

$ astfits -p cat_convolved_scaled.fits | grep NAXIS
NAXIS   =                    2 / number of data axes
NAXIS1  =                  526 / length of data axis 1
NAXIS2  =                  526 / length of data axis 2
```

`cat_convolved_warped.fits` now has the correct pixel scale. However, the image is still larger than what we had wanted, it is 526 ($500 + 13 + 13$) by 526 pixels. The student is slightly confused, so Sufi also re-samples the PSF with the same scale and shows him that it is 27 ($2 \times 13 + 1$) by 27 pixels. Sufi goes on to explain how frequency space convolution will dim the edges and that is why he added the `--prepforconv` option to MakeProfiles, see Section 8.1.2 [If convolving afterwards], page 289. Now that convolution is done, Sufi can remove those extra pixels using Crop with the command below. Crop's `--section` option accepts coordinates inclusively and counting from 1 (according to the FITS standard), so the crop's first pixel has to be 14, not 13.

```
$ astcrop cat_convolved_scaled.fits --section=14:*-13,14:*-13    \
          --zeroisnotblank
Crop started on Sat Oct  6 17:03:24 953
  - Read metadata of 1 image.                          0.001304 seconds
  ---- ...nvolved_scaled_cropped.fits created: 1 input.
Crop finished in:  0.027204 seconds

$ls
0_cat.fits          cat_convolved_scaled_cropped.fits  cat.fits
cat_convolved.fits  cat_convolved_scaled.fits          cat.txt
```

Finally, `cat_convolved_scaled_cropped.fits` has the same dimensions as Sufi had desired in the beginning. All this trouble was certainly worth it because now there is no dimming on the edges of the image and the profile centers are more accurately sampled. The final step to simulate a real observation would be to add noise to the image. Sufi set the zeropoint magnitude to the same value that he set when making the mock profiles and looking again at his observation log, he had measured the background flux near the nebula had a magnitude of 7 that night. So using these values he ran MakeNoise:

```
$ astmknoise --zeropoint=18 --background=7 --output=out.fits    \
```

```
            cat_convolved_warped_crop.fits
MakeNoise started on Mon Apr  6 17:05:06 953
  - Generator type: mt19937
  - Generator seed: 1428318100
MakeNoise finished in:  0.033491 (seconds)

$ls
0_cat.fits         cat_convolved_scaled_cropped.fits cat.fits  out.fits
cat_convolved.fits cat_convolved_scaled.fits         cat.txt
```

The `out.fits` file now contains the noised image of the mock catalog Sufi had asked for. Seeing how the `--output` option allows the user to specify the name of the output file, the student was confused and wanted to know why Sufi hadn't used it before? Sufi then explained to him that for intermediate steps it is best to rely on the automatic output, see Section 4.8 [Automatic output], page 124. Doing so will give all the intermediate files the same basic name structure, so in the end you can simply remove them all with the Shell's capabilities. So Sufi decided to show this to the student by making a shell script from the commands he had used before.

The command-line shell has the capability to read all the separate input commands from a file. This is useful when you want to do the same thing multiple times, with only the names of the files or minor parameters changing between the different instances. Using the shell's history (by pressing the up keyboard key) Sufi reviewed all the commands and then he retrieved the last 5 commands with the `$ history 5` command. He selected all those lines he had input and put them in a text file named `mymock.sh`. Then he defined the `edge` and `base` shell variables for easier customization later. Finally, before every command, he added some comments (lines starting with `#`) for future readability.

```
# Basic settings:
edge=13
base=cat

# Remove any existing image to avoid confusion.
rm out.fits

# Run MakeProfiles to create an oversampled FITS image.
astmkprof --prepforconv --mergedsize=500,500 --zeropoint=18.0   \
          "$base".txt

# Convolve the created image with the kernel.
astconvolve --kernel=0_"$base".fits "$base".fits

# Scale the image back to the intended resolution.
astwarp --scale=1/5 --centeroncorner "$base"_convolved.fits

# Crop the edges out (dimmed during convolution). '--section' accepts
# inclusive coordinates, so the start of start of the section must be
# one pixel larger than its end.
st_edge=$(( edge + 1 ))
```

```
astcrop "$base"_convolved_scaled.fits --zeroisnotblank          \
        --section=$st_edge:*-$edge,$st_edge:*-$edge

# Add noise to the image.
astmknoise --zeropoint=18 --background=7 --output=out.fits      \
           "$base"_convolved_scaled_cropped.fits

# Remove all the temporary files.
rm 0*.fits cat*.fits
```

He used this chance to remind the student of the importance of comments in code or shell scripts: when writing the code, you have a good mental picture of what you are doing, so writing comments might seem superfluous and excessive. However, in one month when you want to re-use the script, you have lost that mental picture and rebuilding it is can be time-consuming and frustrating. The importance of comments is further amplified when you want to share the script with a friend/colleague. So it is good to accompany any script/code with useful comments while you are writing it (have a good mental picture of what/why you are doing something).

Sufi then explained to the eager student that you define a variable by giving it a name, followed by an `=` sign and the value you want. Then you can reference that variable from anywhere in the script by calling its name with a `$` prefix. So in the script whenever you see `$base`, the value we defined for it above is used. If you use advanced editors like GNU Emacs or even simpler ones like Gedit (part of the GNOME graphical user interface) the variables will become a different color which can really help in understanding the script. We have put all the `$base` variables in double quotation marks (`"`) so the variable name and the following text do not get mixed, the shell is going to ignore the `"` after replacing the variable value. To make the script executable, Sufi ran the following command:

```
$ chmod +x mymock.sh
```

Then finally, Sufi ran the script, simply by calling its file name:

```
$ ./mymock.sh
```

After the script finished, the only file remaining is the `out.fits` file that Sufi had wanted in the beginning. Sufi then explained to the student how he could run this script anywhere that he has a catalog if the script is in the same directory. The only thing the student had to modify in the script was the name of the catalog (the value of the `base` variable in the start of the script) and the value to the `edge` variable if he changed the PSF size. The student was also happy to hear that he won't need to make it executable again when he makes changes later, it will remain executable unless he explicitly changes the executable flag with `chmod`.

The student was really excited, since now, through simple shell scripting, he could really speed up his work and run any command in any fashion he likes allowing him to be much more creative in his works. Until now he was using the graphical user interface which doesn't have such a facility and doing repetitive things on it was really frustrating and some times he would make mistakes. So he left to go and try scripting on his own computer.

Sufi could now get back to his own work and see if the simulated nebula which resembled the one in the Andromeda constellation could be detected or not. Although it was extremely

faint[4], fortunately it passed his detection tests and he wrote it in the draft manuscript that would later become "Book of fixed stars". He still had to check the other nebula he saw from Yemen and several other such objects, but they could wait until tomorrow (thanks to the shell script, he only has to define a new catalog). It was nearly sunset and they had to begin preparing for the night's measurements on the ecliptic.

2.2 General program usage tutorial

Measuring colors of astronomical objects in broad-band or narrow-band images is one of the most basic and common steps in astronomical analysis. Here, we will use Gnuastro's programs to get a physical scale (area at certain redshifts) of the field we are studying, detect objects in a Hubble Space Telescope (HST) image, measure their colors and identify the ones with the largest colors to visual inspection and their spatial position in the image. After this tutorial, you can also try the Section 2.3 [Detecting large extended targets], page 47, tutorial which goes into a little more detail on optimally configuring NoiseChisel (Gnuastro's detection tool) in special situations.

During the tutorial, we will take many detours to explain, and practically demonstrate, the many capabilities of Gnuastro's programs. In the end you will see that the things you learned during this tutorial are much more generic than this particular problem and can be used in solving a wide variety of problems involving the analysis of data (images or tables). So please don't rush, and go through the steps patiently to optimally master Gnuastro.

In this tutorial, we'll use the HST eXtreme Deep Field (`https://archive.stsci.edu/prepds/xdf`) dataset. Like almost all astronomical surveys, this dataset is free for download and usable by the public. You will need the following tools in this tutorial: Gnuastro, SAO DS9[5], GNU Wget[6], and AWK (most common implementation is GNU AWK[7]).

This tutorial was first prepared for the "Exploring the Ultra-Low Surface Brightness Universe" workshop (November 2017) at the ISSI in Bern, Switzerland. It was further extended in the "4th Indo-French Astronomy School" (July 2018) organized by LIO, CRAL CNRS UMR5574, UCBL, and IUCAA in Lyon, France. We are very grateful to the organizers of these workshops and the attendees for the very fruitful discussions and suggestions that made this tutorial possible.

Write the example commands manually: Try to type the example commands on your terminal manually and use the history feature of your command-line (by pressing the "up" button to retrieve previous commands). Don't simply copy and paste the commands shown here. This will help simulate future situations when you are processing your own datasets.

[4] The brightness of a diffuse object is added over all its pixels to give its final magnitude, see Section 8.1.3 [Flux Brightness and magnitude], page 290. So although the magnitude 3.44 (of the mock nebula) is orders of magnitude brighter than 6 (of the stars), the central galaxy is much fainter. Put another way, the brightness is distributed over a large area in the case of a nebula.

[5] See Section B.1 [SAO ds9], page 469, available at `http://ds9.si.edu/site/Home.html`

[6] `https://www.gnu.org/software/wget`

[7] `https://www.gnu.org/software/gawk`

A handy feature of Gnuastro is that all program names start with `ast`. This will allow your command-line processor to easily list and auto-complete Gnuastro's programs for you. Try typing the following command (press `TAB` key when you see `<TAB>`) to see the list:

```
$ ast<TAB><TAB>
```

Any program that starts with `ast` (including all Gnuastro programs) will be shown. By choosing the subsequent characters of your desired program and pressing `<TAB><TAB>` again, the list will narrow down and the program name will auto-complete once your input characters are unambiguous. In short, you often don't need to type the full name of the program you want to run.

Gnuastro contains a large number of programs and it is natural to forget the details of each program's options or inputs and outputs. Therefore, before starting the analysis, let's review how you can access this book to refresh your memory any time you want. For example when working on the command-line, without having to take your hands off the keyboard. When you install Gnuastro, this book is also installed on your system along with all the programs and libraries, so you don't need an internet connection to to access/read it. Also, by accessing this book as described below, you can be sure that it corresponds to your installed version of Gnuastro.

GNU Info[8] is the program in charge of displaying the manual on the command-line (for more, see Section 4.3.4 [Info], page 111). To see this whole book on your command-line, please run the following command and press subsequent keys. Info has its own mini-environment, therefore we'll show the keys that must be pressed in the mini-environment after a `->` sign. You can also ignore anything after the `#` sign in the middle of the line, they are only for your information.

```
$ info gnuastro               # Open the top of the manual.
-> <SPACE>                    # All the book chapters.
-> <SPACE>                    # Continue down: show sections.
-> <SPACE> ...                # Keep pressing space to go down.
-> q                          # Quit Info, return to the command-line.
```

The thing that greatly simplifies navigation in Info is the links (regions with an underline). You can immediately go to the next link in the page with the `<TAB>` key and press `<ENTER>` on it to go into that part of the manual. Try the commands above again, but this time also use `<TAB>` to go to the links and press `<ENTER>` on them to go to the respective section of the book. Then follow a few more links and go deeper into the book. To return to the previous page, press `l` (small L). If you are searching for a specific phrase in the whole book (for example an option name), press `s` and type your search phrase and end it with an `<ENTER>`.

You don't need to start from the top of the manual every time. For example, to get to Section 7.2.2 [Invoking NoiseChisel], page 230, run the following command. In general, all programs have such an "Invoking ProgramName" section in this book. These sections are specifically for the description of inputs, outputs and configuration options of each program. You can access them directly for each program by giving its executable name to Info.

```
$ info astnoisechisel
```

[8] GNU Info is already available on almost all Unix-like operating systems.

The other sections don't have such shortcuts. To directly access them from the command-line, you need to tell Info to look into Gnuastro's manual, then look for the specific section (an unambiguous title is necessary). For example, if you only want to review/remember NoiseChisel's Section 7.2.2.2 [Detection options], page 234), just run the following command. Note how case is irrelevant for Info when calling a title in this manner.

```
$ info gnuastro "Detection options"
```

In general, Info is a powerful and convenient way to access this whole book with detailed information about the programs you are running. If you are not already familiar with it, please run the following command and just read along and do what it says to learn it. Don't stop until you feel sufficiently fluent in it. Please invest the half an hour's time necessary to start using Info comfortably. It will greatly improve your productivity and you will start reaping the rewards of this investment very soon.

```
$ info info
```

As a good scientist you need to feel comfortable to play with the features/options and avoid (be critical to) using default values as much as possible. On the other hand, our human memory is limited, so it is important to be able to easily access any part of this book fast and remember the option names, what they do and their acceptable values.

If you just want the option names and a short description, calling the program with the `--help` option might also be a good solution like the first example below. If you know a few characters of the option name, you can feed the output to `grep` like the second or third example commands.

```
$ astnoisechisel --help
$ astnoisechisel --help | grep quant
$ astnoisechisel --help | grep check
```

one can vim /usr/local/etc/astnoisechisel.conf to read the default setups, corresponding to the descriptions in help file.

Let's start the processing. First, to keep things clean, let's create a `gnuastro-tutorial` directory and continue all future steps in it:

```
$ mkdir gnuastro-tutorial
$ cd gnuastro-tutorial
```

We will be using the near infra-red Wide Field Camera (`http://www.stsci.edu/hst/wfc3`) dataset. If you already have them in another directory (for example `XDFDIR`), you can set the `download` directory to be a symbolic link to `XDFDIR` with a command like this:

```
$ ln -s XDFDIR download
```

If the following images aren't already present on your system, you can make a `download` directory and download them there.

```
$ mkdir download
$ cd download
$ xdfurl=http://archive.stsci.edu/pub/hlsp/xdf
$ wget $xdfurl/hlsp_xdf_hst_wfc3ir-60mas_hudf_f105w_v1_sci.fits
$ wget $xdfurl/hlsp_xdf_hst_wfc3ir-60mas_hudf_f160w_v1_sci.fits
$ cd ..
```

In this tutorial, we'll just use these two filters. Later, you will probably need to download more filters, you can use the shell's `for` loop to download them all in series (one after the

other[9]) with one command like the one below for the WFC3 filters. Put this command instead of the two `wget` commands above. Recall that all the extra spaces, back-slashes (\), and new lines can be ignored if you are typing on the lines on the terminal.

```
$ for f in f105w f125w f140w f160w; do                                  \
    wget $xdfurl/hlsp_xdf_hst_wfc3ir-60mas_hudf_"$f"_v1_sci.fits;       \
  done
```

First, let's visually inspect the dataset. Let's take F160W image as an example. Do the steps below with the other image(s) too (and later with any dataset that you want to work on). It is very important to understand your dataset visually. Note how ds9 doesn't follow the GNU style of options where "long" and "short" options are preceded by `--` and `-` respectively (for example `--width` and `-w`, see Section 4.1.1.2 [Options], page 93).

Ds9's `-zscale` option is a good scaling to highlight the low surface brightness regions, and as the name suggests, `-zoom to fit` will fit the whole dataset in the window. If the window is too small, expand it with your mouse, then press the "zoom" button on the top row of buttons above the image, then in the row below it, press "zoom fit". You can also zoom in and out by scrolling your mouse or the respective operation on your touch-pad when your cursor/pointer is over the image.

```
$ ds9 download/hlsp_xdf_hst_wfc3ir-60mas_hudf_f160w_v1_sci.fits     \
      -zscale -zoom to fit
```

The first thing you might notice is that the regions with no data have a value of zero in this image. The next thing might be that the dataset actually has two "depth"s (see Section 7.4.2 [Quantifying measurement limits], page 258). The exposure time of the deep inner region is more than 4 times of the outer parts. Fortunately the XDF survey web-page (above) contains the vertices of the deep flat WFC3-IR field. With Gnuastro's Crop program[10], you can use those vertices to cutout this deep infra-red region from the larger image. We'll make a directory called `flat-ir` and keep the flat infra-red regions in that directory (with a '`xdf-`' suffix for a shorter and easier filename).

```
$ mkdir flat-ir
$ astcrop --mode=wcs -h0 --output=flat-ir/xdf-f105w.fits             \
          --polygon="53.187414,-27.779152 : 53.159507,-27.759633 :   \
                     53.134517,-27.787144 : 53.161906,-27.807208"    \
          download/hlsp_xdf_hst_wfc3ir-60mas_hudf_f105w_v1_sci.fits
$ astcrop --mode=wcs -h0 --output=flat-ir/xdf-f160w.fits             \
          --polygon="53.187414,-27.779152 : 53.159507,-27.759633 :   \
                     53.134517,-27.787144 : 53.161906,-27.807208"    \
          download/hlsp_xdf_hst_wfc3ir-60mas_hudf_f160w_v1_sci.fits
```

The only thing varying in the two calls to Gnuastro's Crop program is the filter name. Therefore, to simplify the command, and later allow work on more filters, we can use the shell's `for` loop. Notice how the two places where the filter names (`f105w` and `f160w`) are used above have been replaced with `$f` (the shell variable that `for` is in charge of setting) below. To generalize this for more filters later, you can simply add the other filter names in the first line before the semi-colon (`;`).

[9] Note that you only have one port to the internet, so downloading in parallel will actually be slower than downloading in series.

[10] To learn more about the crop program see Section 6.1 [Crop], page 151.

```
$ for f in f105w f160w; do                                              \
    astcrop --mode=wcs -h0 --output=flat-ir/xdf-$f.fits                 \
            --polygon="53.187414,-27.779152 : 53.159507,-27.759633 :    \
                       53.134517,-27.787144 : 53.161906,-27.807208"     \
            download/hlsp_xdf_hst_wfc3ir-60mas_hudf_"$f"_v1_sci.fits; \
  done
```

Please open these images and inspect them with the same `ds9` command you used above. You will see how it is nicely flat now and doesn't have varying depths. Another important result of this crop is that regions with no data now have a NaN (Not-a-Number, or a blank value) value, not zero. Zero is a number, and thus a meaningful value, especially when you later want to NoiseChisel[11]. Generally, when you want to ignore some pixels in a dataset, and avoid higher-level ambiguities or complications, it is always best to give them blank values (not zero, or some other absurdly large or small number).

This is the deepest image we currently have of the sky. The first thing that comes to mind may be this: "How large is this field?". The FITS world coordinate system (WCS) meta data standard contains the key to answering this question: the `CDELT` keyword[12]. With the commands below, we'll use it (along with the image size) to find the answer. The lines starting with `##` are just comments for you to help in following the steps. Don't type them on the terminal. The commands are intentionally repetitive in some places to better understand each step and also to demonstrate the beauty of command-line features like variables, pipes and loops. Later, if you would like to repeat this process on another dataset, you can just use commands 3, 7, and 9.

Use shell history: Don't forget to make effective use of your shell's history. This is especially convenient when you just want to make a small change to your previous command. Press the "up" key on your keyboard (possibly multiple times) to see your previous command(s).

```
## (1)  See the general statistics of non-blank pixel values.
$ aststatistics flat-ir/xdf-f160w.fits

## (2)  We only want the number of non-blank pixels.
$ aststatistics flat-ir/xdf-f160w.fits --number
```

[11] As you will see below, unlike most other detection algorithms, NoiseChisel detects the objects from their faintest parts, it doesn't start with their high signal-to-noise ratio peaks. Since the Sky is already subtracted in many images and noise fluctuates around zero, zero is commonly higher than the initial threshold applied. Therefore not ignoring zero-valued pixels in this image, will cause them to part of the detections!

[12] In the FITS standard, the `CDELT` keywords (`CDELT1` and `CDELT2` in a 2D image) specify the scales of each coordinate. In the case of this image it is in units of degrees-per-pixel. See Section 8 of the FITS standard (`https://fits.gsfc.nasa.gov/standard40/fits_standard40aa-le.pdf`) for more. In short, with the `CDELT` convention, rotation (`PC` or `CD` keywords) and scales (`CDELT`) are separated. In the FITS standard the `CDELT` keywords are optional. When `CDELT` keywords aren't present, the `PC` matrix is assumed to contain *both* the coordinate rotation and scales. Note that not all FITS writers use the `CDELT` convention. So you might not find the `CDELT` keywords in the WCS meta data of some FITS files. However, all Gnuastro programs (which use the default FITS keyword writing format of WCSLIB), the `CDELT` convention is used, even if the input doesn't have it. So when rotation and scaling are combined and finding the pixel scale isn't trivial from the raw keyword values, you can feed the dataset to any (simple) Gnuastro program (for example Arithmetic). The output will have the `CDELT` keyword.

```
## (3)  Keep the result of the command above in the shell variable 'n'.
$ n=$(aststatistics flat-ir/xdf-f160w.fits --number)

## (4)  See what is stored the shell variable 'n'.
$ echo $n

## (5)  Show all the FITS keywords of this image.
$ astfits flat-ir/xdf-f160w.fits -h1

## (6)  The resolution (in degrees/pixel) is in the 'CDELT' keywords.
##      Only show lines that contain these characters, by feeding
##      the output of the previous command to the 'grep' program.
$ astfits flat-ir/xdf-f160w.fits -h1 | grep CDELT

## (7)  Save the resolution (same in both dimensions) in the variable
##      'r'. The last part uses AWK to print the third 'field' of its
##      input line. The first two fields were 'CDELT1' and '='.
$ r=$(astfits flat-ir/xdf-f160w.fits -h1 | grep CDELT1   \
              | awk '{print $3}')

## (8)  Print the values of 'n' and 'r'.
$ echo $n $r

## (9)  Use the number of pixels (first number passed to AWK) and
##      length of each pixel's edge (second number passed to AWK)
##      to estimate the area of the field in arc-minutes squared.
$ area=$(echo $n $r | awk '{print $1 * ($2^2) * 3600}')
```

The area of this field is 4.03817 (or 4.04) arc-minutes squared. Just for comparison, this is roughly 175 times smaller than the average moon's angular area (with a diameter of 30arc-minutes or half a degree).

AWK for table/value processing: AWK is a powerful and simple tool for text processing. Above (and further below) some simple examples are shown. GNU AWK (the most common implementation) comes with a free and wonderful book (`https://www.gnu.org/software/gawk/manual/`) in the same format as this book which will allow you to master it nicely. Just like this manual, you can also access GNU AWK's manual on the command-line whenever necessary without taking your hands off the keyboard.

This takes us to the second question that you have probably asked yourself when you saw the field for the first time: "How large is this area at different redshifts?". To get a feeling of the tangential area that this field covers at redshift 2, you can use Section 9.1 [CosmicCalculator], page 307. In particular, you need the tangential distance covered by 1 arc-second as raw output. Combined with the field's area, we can then calculate the tangential distance in Mega Parsecs squared (Mpc^2).

```
## Print general cosmological properties at redshift 2.
```

```
$ astcosmiccal -z2

## When given a "Specific calculation" option, CosmicCalculator
## will just print that particular calculation. See the options
## under this title in the output of '--help' for more.
$ astcosmiccal --help

## Only print the "Tangential dist. covered by 1arcsec at z (kpc)".
## in units of kpc/arc-seconds.
$ astcosmiccal -z2 --arcsectandist

## Convert this distance to kpc^2/arcmin^2 and save in 'k'.
$ k=$(astcosmiccal -z2 --arcsectandist | awk '{print ($1*60)^2}')

## Multiply by the area of the field (in arcmin^2) and divide by
## 10^6 to return value in Mpc^2.
$ echo $k $area | awk '{print $1 * $2 / 1e6}'
```

At redshift 2, this field therefore covers 1.07145 Mpc^2. If you would like to see how this tangential area changes with redshift, you can use a shell loop like below.

```
$ for z in 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0; do          \
    k=$(astcosmiccal -z$z --arcsectandist);                     \
    echo $z $k $area | awk '{print $1, ($2*60)^2 * $3 / 1e6}';  \
  done
```

Fortunately, the shell has a useful tool/program to print a sequence of numbers that is nicely called `seq`. You can use it instead of typing all the different redshifts in this example. For example the loop below will print the same range of redshifts (between 0.5 and 5) but with increments of 0.1.

```
$ for z in $(seq 0.5 0.1 5); do                                 \
    k=$(astcosmiccal -z$z --arcsectandist);                     \
    echo $z $k $area | awk '{print $1, ($2*60)^2 * $3 / 1e6}';  \
  done
```

This is a fast and simple way for this repeated calculation when it is only necessary once. However, if you commonly need this calculation and possibly for a larger number of redshifts, the command above can be slow. This is because the CosmicCalculator program has a lot of overhead. To be generic and easy to operate, it has to parse the command-line and all configuration files (see below) which contain human-readable characters and need a lot of processing to be ready for processing by the computer. Afterwards, CosmicCalculator has to check the sanity of its inputs and check which of its many options you have asked for. It has to do all of these for every redshift in the loop above.

To greatly speed up the processing, you can directly access the root work-horse of CosmicCalculator without all that overhead. Using Gnuastro's library, you can write your own tiny program particularly designed for this exact calculation (and nothing else!). To do that, copy and paste the following C program in a file called `myprogram.c`.

```
#include <math.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <gnuastro/cosmology.h>

int
main(void)
{
  double area=4.03817;          /* Area of field (arcmin^2). */
  double z, adist, tandist;     /* Temporary variables.      */

  /* Constants from Plank 2018 (arXiv:1807.06209, Table 2) */
  double H0=67.66, olambda=0.6889, omatter=0.3111, oradiation=0;

  /* Do the same thing for all redshifts (z) between 0.1 and 5. */
  for(z=0.1; z<5; z+=0.1)
    {
      /* Calculate the angular diameter distance. */
      adist=gal_cosmology_angular_distance(z, H0, olambda,
                                           omatter, oradiation);

      /* Calculate the tangential distance of one arcsecond. */
      tandist = adist * 1000 * M_PI / 3600 / 180;

      /* Print the redshift and area. */
      printf("%-5.2f %g\n", z, pow(tandist * 60,2) * area / 1e6);
    }

  /* Tell the system that everything finished successfully. */
  return EXIT_SUCCESS;
}
```

To greatly simplify the compilation, linking and running of simple C programs like this that use Gnuastro's library, Gnuastro has Section 11.2 [BuildProgram], page 328. This program designed to manage Gnuastro's dependencies, compile and link the program and then run the new program. To build and run the program above, simply run the following command:

```
$ astbuildprog myprogram.c
```

Did you notice how much faster this was compared to the shell loop we wrote above? You might have noticed that a new file called `myprogram` is also created in the directory. This is the compiled program that was created and run by the command above (its in binary machine code format, not human-readable any more). You can run it again to get the same results with a command like this:

```
$ ./myprogram
```

The efficiency of `myprogram` compared to CosmicCalculator is because the requested processing is faster/comparable to the overheads necessary for each processing. For other programs that take large input datasets (images for example), the overhead is usually negligible compared to the processing. In such cases, the libraries are only useful if you want a different/new processing compared to the functionalities in Gnuastro's existing programs.

Gnuastro has a large library which is heavily used by all the programs. In other words, the library is like the skeleton of Gnuastro. For the full list of available functions classified by context, please see Section 11.3 [Gnuastro library], page 332. Gnuastro's library and BuildProgram are created to make it easy for you to use these powerful features as you like. This gives you a high level of creativity, while also providing efficiency and robustness. Several other complete working examples (involving images and tables) of Gnuastro's libraries can be see in Section 11.4 [Library demo programs], page 437. Let's stop the discussion on libraries at this point in this tutorial and get back to Gnuastro's already built programs which were the main purpose of this tutorial.

None of Gnuastro's programs keep a default value internally within their code. However, when you ran CosmicCalculator with the `-z2` option above, it completed its processing and printed results. So where did the "default" cosmological parameter values (like the matter density and etc) come from? The values come from the command-line or a configuration file (see Section 4.2.2 [Configuration file precedence], page 107).

CosmicCalculator has a limited set of parameters and doesn't need any particular file inputs. Therefore, we'll use it to discuss configuration files which are an important part of all Gnuastro's programs (see Section 4.2 [Configuration files], page 106).

Once you get comfortable with configuration files, you can easily do the same for the options of all Gnuastro programs. For example, NoiseChisel has the largest number of options in the programs. Therefore configuration files will be useful for it when you use different datasets (with different noise properties or in different research contexts). The configuration of each program (besides its version) is vital for the reproducibility of your results, so it is important to manage them properly.

As we saw above, the full list of the options in all Gnuastro programs can be seen with the `--help` option. Try calling it with CosmicCalculator as shown below. Note how options are grouped by context to make it easier to find your desired option. However, in each group, options are ordered alphabetically.

```
$ astcosmiccal --help
```

The options that need a value have an `=` sign after their long version and `FLT`, `INT` or `STR` for floating point numbers, integer numbers and strings (filenames for example) respectively. All options have a long format and some have a short format (a single character), for more see Section 4.1.1.2 [Options], page 93.

When you are using a program, it is often necessary to check the value the option has just before the program starts its processing. In other words, after it has parsed the command-line options and all configuration files. You can see the values of all options that need one with the `--printparams` or `-P` option that is common to all programs (see Section 4.1.2 [Common options], page 95). In the command below, try replacing `-P` with `--printparams` to see how both do the same operation.

```
$ astcosmiccal -P
```

Let's say you want a different Hubble constant. Try running the following command to see how the Hubble constant in the output of the command above has changed. Afterwards, delete the `-P` and add a `-z2` to see the results with the new cosmology (or configuration).

```
$ astcosmiccal -P --H0=70
```

From the output of the `--help` option, note how the option for Hubble constant has both short (`-H`) and long (`--H0`) formats. One final note is that the equal (`=`) sign is not

mandatory. In the short format, the value can stick to the actual option (the short option name is just one character after-all and thus easily identifiable) and in the long format, a white-space character is also enough.

```
$ astcosmiccal -H70    -z2
$ astcosmiccal --H0 70 -z2 --arcsectandist
```

Let's assume that in one project, you want to only use rounded cosmological parameters (H0 of 70km/s/Mpc and matter density of 0.3). You should therefore run CosmicCalculator like this:

```
$ astcosmiccal --H0=70 --olambda=0.7 --omatter=0.3 -z2
```

But having to type these extra options every time you run CosmicCalculator will be prone to errors (typos in particular) and also will be frustrating and slow. Therefore in Gnuastro, you can put all the options and their values in a "Configuration file" and tell the programs to read the option values from there.

Let's create a configuration file. In your favorite text editor, make a file named `my-cosmology.conf` (or `my-cosmology.txt`, the suffix doesn't matter) which contains the following lines. One space between the option value and name is enough, the values are just under each other to help in readability. Also note that you can only use long option names in configuration files.

```
H0       70
olambda  0.7
omatter  0.3
```

You can now tell CosmicCalculator to read this file for option values immediately using the `--config` option as shown below. Do you see how the output of the following command corresponds to the option values in `my-cosmology.conf` (previous command)?

```
$ astcosmiccal --config=my-cosmology.conf -z2
```

If you need this cosmology every time you are working in a specific directory, you can benefit from Gnuastro's default configuration files to avoid having to call the `--config` option. Let's assume that you want any CosmicCalculator call you make in the `my-cosmology` directory to use these parameters. You just have to copy the above configuration file into a special directory and file:

```
$ mkdir my-cosmology
$ mkdir my-cosmology/.gnuastro
$ mv my-cosmology.conf my-cosmology/.gnuastro/astcosmiccal.conf
```

Once you run CosmicCalculator within `my-cosmology` as shown below, you will see how your cosmology has been implemented without having to type anything extra on the command-line.

```
$ cd my-cosmology
$ astcosmiccal -z2
$ cd ..
```

To further simplify the process, you can use the `--setdirconf` option. If you are already in your desired directory, calling this option with the others will automatically write the final values (along with descriptions) in `.gnuastro/astcosmiccal.conf`. For example the commands below will make the same configuration file automatically (with one extra call to CosmicCalculator).

```
$ mkdir my-cosmology2
$ cd my-cosmology2
$ astcosmiccal --H0 70 --olambda=0.7 --omatter=0.3 --setdirconf
$ astcosmiccal -z2
$ cd ..
```

Gnuastro's programs also have default configuration files for a specific user (when run in any directory). This allows you to set a special behavior every time a program is run by a specific user. Only the directory and filename differ from the above, the rest of the process is similar to before. Finally, there are also system-wide configuration files that can be used to define the option values for all users on a system. See Section 4.2.2 [Configuration file precedence], page 107, for a more detailed discussion.

We are now ready to start processing the downloaded images. Since these datasets are already aligned, you don't need to align them to make sure the pixel grid covers the same region in all inputs. Gnuastro's Warp program has features for such pixel-grid warping (see Section 6.4 [Warp], page 199). Therefore, just for a demonstration, let's assume one image needs to be rotated by 20 degrees to correspond to the other. To do that, you can run the following command:

```
$ astwarp flat-ir/xdf-f160w.fits --rotate=20
```

Open the output and see it. If your final image is already aligned with RA and Dec, you can simply use the `--align` option and let Warp calculate the necessary rotation and apply it.

Warp can generally be used for any kind of pixel grid manipulation (warping). For example the outputs of the commands below will respectively have larger pixels (new resolution being one quarter the original resolution), get shifted by 2.8 (by sub-pixel), get a shear of 2, and be tilted (projected). After running each, please open the output file and see the effect.

```
$ astwarp flat-ir/xdf-f160w.fits --scale=0.25
$ astwarp flat-ir/xdf-f160w.fits --translate=2.8
$ astwarp flat-ir/xdf-f160w.fits --shear=2
$ astwarp flat-ir/xdf-f160w.fits --project=0.001,0.0005
```

If you need to do multiple warps, you can combine them in one call to Warp. For example to first rotate the image, then scale it, run this command:

```
$ astwarp flat-ir/xdf-f160w.fits --rotate=20 --scale=0.25
```

If you have multiple warps, do them all in one command. Don't warp them in separate commands because the correlated noise will become too strong. As you see in the matrix that is printed when you run Warp, it merges all the warps into a single warping matrix (see Section 6.4.1 [Warping basics], page 200, and Section 6.4.2 [Merging multiple warpings], page 202) and simply applies that just once. Recall that since this is done through matrix multiplication, order matters in the separate operations. In fact through Warp's `--matrix` option, you can directly request your desired final warp and don't have to break it up into different warps like above (see Section 6.4.4 [Invoking Warp], page 204).

Fortunately these datasets are already aligned to the same pixel grid, so you don't actually need the files that were just generated. You can safely delete them all with the following command. Here, you see why we put the processed outputs that we need later

into a separate directory. In this way, the top directory can be used for temporary files for testing that you can simply delete with a generic command like below.

```
$ rm *.fits
```

To detect the signal in the image (separate interesting pixels from noise), we'll run NoiseChisel (Section 7.2 [NoiseChisel], page 225):

```
$ astnoisechisel flat-ir/xdf-f160w.fits
```

NoiseChisel's output is a single FITS file containing multiple extensions. In the FITS format, each extension contains a separate dataset (image in this case). You can get basic information about the extensions in a FITS file with Gnuastro's Fits program (see Section 5.1 [Fits], page 128):

```
$ astfits xdf-f160w_detected.fits
```

From the output list, we see that NoiseChisel's output contains 5 extensions and the first (counting from zero, with name `NOISECHISEL-CONFIG`) is empty: it has value of `0` in the last column (which shows its size). The first extension in all the outputs of Gnuastro's programs only contains meta-data: data about/describing the datasets within (all) the output's extension(s). This allows the first extension to keep meta-data about all the extensions and is recommended by the FITS standard, see Section 5.1 [Fits], page 128, for more. This generic meta-data (for the whole file) is very important for being able to reproduce this same result later.

The second extension of NoiseChisel's output (numbered 1 and named `INPUT-NO-SKY`) is the Sky-subtracted input that you provided. The third (`DETECTIONS`) is NoiseChisel's main output which is a binary image with only two possible values for all pixels: 0 for noise and 1 for signal. Since it only has two values, to avoid taking too much space on your computer, its numeric datatype an unsigned 8-bit integer (or `uint8`)[13]. The fourth and fifth (`SKY` and `SKY_STD`) extensions, have the Sky and its standard deviation values for the input on a tile grid and were calculated over the undetected regions (for more on the importance of the Sky value, see Section 7.1.3 [Sky value], page 211).

Reproducing your results later (or checking the configuration of the program that produced the dataset at a later time during your higher-level analysis) is very important in any research. Therefore, Let's first take a closer look at the `NOISECHISEL-CONFIG` extension. But first, we'll run NoiseChisel with `-P` to see the option values in a format we are already familiar with (to help in the comparison).

```
$ astnoisechisel -P
$ astfits xdf-f160w_detected.fits -h0
```

The first group of FITS header keywords are standard keywords (containing the `SIMPLE` and `BITPIX` keywords the first empty line). They are required by the FITS standard and must be present in any FITS extension. The second group contain the input file and all the options with their values in that run of NoiseChisel. Finally, the last group contain the date and version information of Gnuastro and its dependencies. The "versions and date" group of keywords are present in all Gnuastro's FITS extension outputs, for more see Section 4.9 [Output FITS files], page 125.

Note that if a keyword name is larger than 8 characters, it is preceded by a `HIERARCH` keyword and that all keyword names are in capital letters. Therefore, if you want to see

[13] To learn more about numeric data types see Section 4.5 [Numeric data types], page 115.

only one keyword's value by feeding the output to Grep, you should ask Grep to ignore case with its `-i` option (short name for `--ignore-case`). For example, below we'll check the value to the `--snminarea` option, note how we don't need Grep's `-i` option when it is fed with `astnoisechisel -P` since it is already in small-caps there. The extra white spaces in the first command are only to help in readability, you can ignore them when typing.

```
$ astnoisechisel -P                    | grep    snminarea
$ astfits xdf-f160w_detected.fits -h0 | grep -i snminarea
```

Let's continue with the extensions in NoiseChisel's output that contain a dataset by visually inspecting them (here, we'll use SAO DS9). Since the file contains multiple related extensions, the easiest way to view all of them in DS9 is to open the file as a "Multi-extension data cube" with the `-mecube` option as shown below[14].

```
$ ds9 -mecube xdf-f160w_detected.fits -zscale -zoom to fit
```

A "cube" window opens along with DS9's main window. The buttons and horizontal scroll bar in this small new window can be used to navigate between the extensions. In this mode, all DS9's settings (for example zoom or color-bar) will be identical between the extensions. Try zooming into to one part and flipping through the extensions to see how the galaxies were detected along with the Sky and Sky standard deviation values for that region. Just have in mind that NoiseChisel's job is *only* detection (separating signal from noise), We'll do segmentation on this result later to find the individual galaxies/peaks over the detected pixels.

One good way to see if you have missed any signal (small galaxies, or the wings of brighter galaxies) is to mask all the detected pixels and inspect the noise pixels. For this, you can use Gnuastro's Arithmetic program (in particular its `where` operator, see Section 6.2.2 [Arithmetic operators], page 162). With the command below, all detected pixels (in the `DETECTIONS` extension) will be set to NaN in the output (`nc-masked.fits`). To make the command easier to read/write, let's just put the file name in a shell variable (`img`) first. A shell variable's value can be retrieved by adding a `$` before its name.

```
$ img=xdf-f160w_detected.fits
$ astarithmetic $img $img nan where -hINPUT-NO-SKY -hDETECTIONS      \
                --output=mask-det.fits
```

To invert the result (only keep the values of detected pixels), you can flip the detected pixel values (from 0 to 1 and vice-versa) by adding a `not` after the second `$img`:

```
$ astarithmetic $img $img not nan where -hINPUT-NO-SKY -hDETECTIONS  \
                --output=mask-sky.fits
```

Looking again at the detected pixels, we see that there are thin connections between many of the smaller objects or extending from larger objects. This shows that we have dug in too deep, and that we are following correlated noise.

Correlated noise is created when we warp datasets from individual exposures (that are each slightly offset compared to each other) into the same pixel grid, then add them to form the final result. Because it mixes nearby pixel values, correlated noise is a form of

[14] You can configure your graphic user interface to open DS9 in multi-extension cube mode by default when using the GUI (double clicking on the file). If your graphic user interface is GNOME (another GNU software, it is most common in GNU/Linux operating systems), a full description is given in Section B.1.1 [Viewing multiextension FITS images], page 469

convolution and it smooths the image. In terms of the number of exposures (and thus correlated noise), the XDF dataset is by no means an ordinary dataset. It is the result of warping and adding roughly 80 separate exposures which can create strong correlated noise/smoothing. In common surveys the number of exposures is usually 10 or less.

Let's tweak NoiseChisel's configuration a little to get a better result on this dataset. Don't forget that "*Good statistical analysis is not a purely routine matter, and generally calls for more than one pass through the computer*" (Anscombe 1973, see Section 1.2 [Science and its tools], page 2). A good scientist must have a good understanding of her tools to make a meaningful analysis. So don't hesitate in playing with the default configuration and reviewing the manual when you have a new dataset in front of you. Robust data analysis is an art, therefore a good scientist must first be a good artist.

NoiseChisel can produce "Check images" to help you visualize and inspect how each step is done. You can see all the check images it can produce with this command.

```
$ astnoisechisel --help | grep check
```

Let's check the overall detection process to get a better feeling of what NoiseChisel is doing with the following command. To learn the details of NoiseChisel in more detail, please see Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`).

```
$ astnoisechisel flat-ir/xdf-f160w.fits --checkdetection
```

The check images/tables are also multi-extension FITS files. As you saw from the command above, when check datasets are requested, NoiseChisel won't go to the end. It will abort as soon as all the extensions of the check image are ready. Try listing the extensions of the output with `astfits` and then opening them with `ds9` as we done above. In order to understand the parameters and their biases (especially as you are starting to use Gnuastro, or running it a new dataset), it is *strongly* encouraged to play with the different parameters and use the respective check images to see which step is affected by your changes and how, for example see Section 2.3 [Detecting large extended targets], page 47.

The `OPENED_AND_LABELED` extension shows the initial detection step of NoiseChisel. We see these thin connections between smaller points are already present here (a relatively early stage in the processing). Such connections at the lowest surface brightness limits usually occur when the dataset is too smoothed. Because of correlated noise, the dataset is already artificially smoothed, therefore further smoothing it with the default kernel may be the problem. Therefore, one solution is to use a sharper kernel (NoiseChisel's first step in its processing).

By default NoiseChisel uses a Gaussian with full-width-half-maximum (FWHM) of 2 pixels. We can use Gnuastro's MakeProfiles to build a kernel with FWHM of 1.5 pixel (truncated at 5 times the FWHM, like the default) using the following command. MakeProfiles is a powerful tool to build any number of mock profiles on one image or independently, to learn more of its features and capabilities, see Section 8.1 [MakeProfiles], page 284.

```
$ astmkprof --kernel=gaussian,1.5,5 --oversample=1
```

Please open the output `kernel.fits` and have a look (it is very small and sharp). We can now tell NoiseChisel to use this instead of the default kernel with the following command (we'll keep checking the detection steps)

```
$ astnoisechisel flat-ir/xdf-f160w.fits --kernel=kernel.fits  \
                 --checkdetection
```

Looking at the `OPENED_AND_LABELED` extension, we see that the thin connections between smaller peaks has now significantly decreased. Going two extensions/steps ahead (in the first `HOLES-FILLED`), you can see that during the process of finding false pseudo-detections, too many holes have been filled: see how the many of the brighter galaxies are connected?

Try looking two extensions ahead (in the first `PSEUDOS-FOR-SN`), you can see that there aren't too many pseudo-detections because of all those extended filled holes. If you look closely, you can see the number of pseudo-detections in the result NoiseChisel prints (around 4000). This is another side-effect of correlated noise. To address it, we should slightly increase the pseudo-detection threshold (`--dthresh`, run with `-P` to see its default value):

```
$ astnoisechisel flat-ir/xdf-f160w.fits --kernel=kernel.fits  \
                 --dthresh=0.2 --checkdetection
```

Before visually inspecting the check image, you can see the effect of this change in NoiseChisel's command-line output: notice how the number of pseudos has increased to roughly 5500. Open the check image now and have a look, you can see how the pseudo-detections are distributed much more evenly in the image. The signal-to-noise ratio of pseudo-detections define NoiseChisel's reference for removing false detections, so they are very important to get right. Let's have a look at their signal-to-noise distribution with `--checksn`.

```
$ astnoisechisel flat-ir/xdf-f160w.fits --kernel=kernel.fits  \
                 --dthresh=0.2 --checkdetection --checksn
```

The output `xdf-f160w_detsn.fits` file contains two extensions for the pseudo-detections over the undetected (sky) regions and those over detections. The first column is the pseudo-detection label which you can see in the respective[15] `PSEUDOS-FOR-SN` extension of `xdf-f160w_detcheck.fits`. You can see the table columns with the first command below and get a feeling for its distribution with the second command. We'll discuss the two Table and Statistics programs later.

```
$ asttable xdf-f160w_detsn.fits
$ aststatistics xdf-f160w_detsn.fits -c2
```

The correlated noise is again visible in this pseudo-detection signal-to-noise distribution: it is highly skewed. A small change in the quantile will translate into a big change in the S/N value. For example see the difference between the three 0.99, 0.95 and 0.90 quantiles with this command:

```
$ aststatistics xdf-f160w_detsn.fits -c2                        \
                --quantile=0.99 --quantile=0.95 --quantile=0.90
```

If you run NoiseChisel with `-P`, you'll see the default signal-to-noise quantile `--snquant` is 0.99. In effect with this option you specify the purity level you want (contamination by false detections). With the `aststatistics` command above, you see that a small number of extra false detections (impurity) in the final result causes a big change in completeness (you can detect more lower signal-to-noise true detections). So let's loosen-up our desired purity level and then mask the detected pixels like before to see if we have missed anything.

```
$ astnoisechisel flat-ir/xdf-f160w.fits --kernel=kernel.fits  \
                 --dthresh=0.2 --snquant=0.95
```

[15] The first `PSEUDOS-FOR-SN` in `xdf-f160w_detsn.fits` is for the pseudo-detections over the undetected regions and the second is for those over detected regions.

```
$ img=xdf-f160w_detected.fits
$ astarithmetic $img $img nan where -h1 -h2 --output=mask-det.fits
```

Overall it seems good, but if you play a little with the color-bar and look closer in the noise, you'll see a few very sharp, but faint, objects that have not been detected. This only happens for under-sampled datasets like HST (where the pixel size is larger than the point spread function FWHM). So this won't happen on ground-based images. Because of this, sharp and faint objects will be very small and eroded too easily during NoiseChisel's erosion step.

To address this problem of sharp objects, we can use NoiseChisel's `--noerodequant` option. All pixels above this quantile will not be eroded, thus allowing us to preserve faint and sharp objects. Check its default value, then run NoiseChisel like below and make the mask again. You will see many of those sharp objects are now detected.

```
$ astnoisechisel flat-ir/xdf-f160w.fits --kernel=kernel.fits     \
                 --noerodequant=0.95 --dthresh=0.2 --snquant=0.95
```

This seems to be fine and we can continue with our analysis. Before finally running NoiseChisel, let's just see how you can have all the raw outputs of NoiseChisel (Detection map and Sky and Sky Standard deviation) in a highly compressed format for archivability. For example the Sky-subtracted input is a redundant dataset: you can always generate it by subtracting the Sky from the input image. With the commands below you can turn the default NoiseChisel output that is larger than 100 megabytes in this case into about 200 kilobytes by removing all the redundant information in it, then compressing it:

```
$ astnoisechisel flat-ir/xdf-f160w.fits --oneelempertile --rawoutput
$ gzip --best xdf-f160w_detected.fits
```

You can open `xdf-f160w_detected.fits.gz` directly in SAO DS9 or feed it to any of Gnuastro's programs without having to uncompress it. Higher-level programs that take NoiseChisel's output as input can also deal with this compressed image where the Sky and its Standard deviation are one pixel-per-tile.

To avoid having to write these options on every call to NoiseChisel, we'll just make a configuration file in a visible `config` directory. Then we'll define the hidden `.gnuastro` directory (that all Gnuastro's programs will look into for configuration files) as a symbolic link to the `config` directory. Finally, we'll write the finalized values of the options into NoiseChisel's standard configuration file within that directory. We'll also put the kernel in a separate directory to keep the top directory clean of any files we later need.

```
$ mkdir kernel config
$ ln -s config/ .gnuastro
$ mv kernel.fits det-kernel.fits
$ echo "kernel kernel/det-kernel.fits" > config/astnoisechisel.conf
$ echo "noerodequant 0.95"            >> config/astnoisechisel.conf
$ echo "dthresh      0.2"             >> config/astnoisechisel.conf
$ echo "snquant      0.95"            >> config/astnoisechisel.conf
```

We are now ready to finally run NoiseChisel on the two filters and keep the output in a dedicated directory (`nc`).

```
$ rm *.fits
$ mkdir nc
$ astnoisechisel flat-ir/xdf-f160w.fits --output=nc/xdf-f160w.fits
```

```
$ astnoisechisel flat-ir/xdf-f105w.fits --output=nc/xdf-f105w.fits
```

Before continuing with the higher-level processing of this dataset, let's pause to use NoiseChisel's multi-extension output as a demonstration for working with FITS extensions using Gnuastro's Fits program (see Section 5.1 [Fits], page 128.

Let's say you need to copy a HDU/extension (image or table) from one FITS file to another. After the command below, `objects.fits` file will contain only one extension: a copy of NoiseChisel's binary detection map. There are similar options to conveniently cut (`--cut`, copy, then remove from the input) or delete (`--remove`) HDUs from a FITS file also.

```
$ astfits nc/xdf-f160w.fits --copy=DETECTIONS -odetections.fits
```

NoiseChisel puts some general information on its outputs in the FITS header of the respective extension. To see the full list of keywords in an extension, you can again use the Fits program like above. But instead of HDU manipulation options, give it the HDU you are interested in with `-h`. You can also give the HDU number (as listed in the output above), for example `-h2` instead of `-hDETECTIONS`.

```
$ astfits nc/xdf-f160w.fits -hDETECTIONS
```

The `DETSN` keyword in NoiseChisel's `DETECTIONS` extension contains the true pseudo-detection signal-to-noise ratio that was found by NoiseChisel on the dataset. It is not easy to find it in the middle of all the other keywords printed by the command above (especially in files that have many more keywords). To fix the problem, you can pipe the output of the command above into `grep` (a program for matching lines which is available on almost all Unix-like operating systems).

```
$ astfits nc/xdf-f160w.fits -hDETECTIONS | grep DETSN
```

If you just want the value of the keyword and not the full FITS keyword line, you can use AWK. In the example below, AWK will print the third word (separated by white space characters) in any line that has a first column value of `DETSN`.

```
$ astfits nc/xdf-f160w.fits -h2 | awk '$1=="DETSN" {print $3}'
```

The main output of NoiseChisel is the binary detection map (`DETECTIONS` extension), which only has two values of 1 or 0. This is useful when studying the noise, but hardly of any use when you actually want to study the targets/galaxies in the image, especially in such a deep field where the detection map of almost everything is connected. To find the galaxies over the detections, we'll use Gnuastro's Section 7.3 [Segment], page 243, program:

```
$ rm *.fits
$ mkdir seg
$ astsegment nc/xdf-f160w.fits -oseg/xdf-f160w.fits
```

Segment's operation is very much like NoiseChisel (in fact, prior to version 0.6, it was part of NoiseChisel), for example the output is a multi-extension FITS file, it has check images and uses the undetected regions as a reference. Please have a look at Segment's multi-extension output with `ds9` to get a good feeling of what it has done. Like NoiseChisel, the first extension is the input. The `CLUMPS` extension shows the true "clumps" with values that are ≥ 1, and the diffuse regions labeled as -1. In the `OBJECTS` extension, we see that the large detections of NoiseChisel (that may have contained many galaxies) are now broken up into separate labels. see Section 7.3 [Segment], page 243, for more.

Having localized the regions of interest in the dataset, we are ready to do measurements on them with Section 7.4 [MakeCatalog], page 255. Besides the IDs, we want to measure (in this order) the Right Ascension (with `--ra`), Declination (`--dec`), magnitude (`--magnitude`), and signal-to-noise ratio (`--sn`) of the objects and clumps. The following command will make these measurements on Segment's F160W output:

```
$ mkdir cat
$ astmkcatalog seg/xdf-f160w.fits --ids --ra --dec --magnitude --sn \
               --zeropoint=25.94                                    \
               --clumpscat --output=cat/xdf-f160w.fits
```

From the printed statements on the command-line, you see that MakeCatalog read all the extensions in Segment's output for the various measurements it needed.

The output of the MakeCatalog command above is a FITS table. The two clump and object catalogs are available in the two extensions of the single FITS file[16]. Let's inspect the separate extensions with the Fits program like before (as shown below). Later, we'll inspect the table in each extension with Gnuastro's Table program (see Section 5.3 [Table], page 147). Note that we could have used `-hOBJECTS` and `-hCLUMPS` instead of `-h1` and `-h2` respectively.

```
$ astfits  cat/xdf-f160w.fits              # Extension information
$ asttable cat/xdf-f160w.fits -h1 --info   # Objects catalog info.
$ asttable cat/xdf-f160w.fits -h1          # Objects catalog columns.
$ asttable cat/xdf-f160w.fits -h2 -i       # Clumps catalog info.
$ asttable cat/xdf-f160w.fits -h2          # Clumps catalog columns.
```

As you see above, when given a specific table (file name and extension), Table will print the full contents of all the columns. To see basic information about each column (for example name, units and comments), simply append a `--info` (or `-i`).

To print the contents of special column(s), just specify the column number(s) (counting from `1`) or the column name(s) (if they have one). For example, if you just want the magnitude and signal-to-noise ratio of the clumps (in `-h2`), you can get it with any of the following commands

```
$ asttable cat/xdf-f160w.fits -h2 -c5,6
$ asttable cat/xdf-f160w.fits -h2 -c5,SN
$ asttable cat/xdf-f160w.fits -h2 -c5         -c6
$ asttable cat/xdf-f160w.fits -h2 -cMAGNITUDE -cSN
```

In the example above, the clumps catalog has two ID columns (one for the over-all clump ID and one for the ID of the clump in its host object), while the objects catalog only has one ID column. Therefore, the location of the magnitude column differs between the object and clumps catalog. So if you want to specify the columns by number, you will need to change the numbers when viewing the clump and objects catalogs. This is a useful advantage of having/using column names[17].

```
$ asttable catalog/xdf-f160w.fits -h1 -c4 -c5
```

[16] MakeCatalog can also output plain text tables. However, in the plain text format you can only have one table per file. Therefore, if you also request measurements on clumps, two plain text tables will be created (suffixed with `_o.txt` and `_c.txt`).

[17] Column meta-data (including a name) can also be specified in plain text tables, see Section 4.6.2 [Gnuastro text table format], page 119.

```
$ asttable catalog/xdf-f160w.fits -h2 -c5 -c6
```

Finally, the comments in MakeCatalog's output (`COMMENT` keywords in the FITS headers, or lines starting with `#` in plain text) contain some important information about the input dataset that can be useful (for example pixel area or per-pixel surface brightness limit). For example have a look at the output of this command:

```
$ astfits cat/xdf-f160w.fits -h1 | grep COMMENT
```

To calculate colors, we also need magnitude measurements on the F105W filter. However, the galaxy properties might differ between the filters (which is the whole purpose behind measuring colors). Also, the noise properties and depth of the datasets differ. Therefore, if we simply follow the same Segment and MakeCatalog calls above for the F105W filter, we are going to get a different number of objects and clumps. Matching the two catalogs is possible (for example with Section 7.5 [Match], page 279), but the fact that the measurements will be done on different pixels, can bias the result. Since the Point spread function (PSF) of both images is very similar, an accurate color calculation can only be done when magnitudes are measured from the same pixels on both images.

The F160W image is deeper, thus providing better detection/segmentation, and redder, thus observing smaller/older stars and representing more of the mass in the galaxies. We will thus use the pixel labels generated on the F160W filter, but do the measurements on the F105W filter (using the `--valuesfile` option) in the command below. Notice how the only difference between this call to MakeCatalog and the previous one is `--valuesfile`, the value given to `--zeropoint` and the output name.

```
$ astmkcatalog seg/xdf-f160w.fits --ids --ra --dec --magnitude --sn \
               --valuesfile=nc/xdf-f105w.fits --zeropoint=26.27     \
               --clumpscat --output=cat/xdf-f105w.fits
```

Look into what MakeCatalog printed on the command-line. You can see that (as requested) the object and clump labels were taken from the respective extensions in `seg/xdf-f160w.fits`, while the values and Sky standard deviation were done on `nc/xdf-f105w.fits`.

Since we used the same labeled image on both filters, the number of rows in both catalogs are the same. The clumps are not affected by the hard-to-deblend and low signal-to-noise diffuse regions, they are more robust for calculating the colors (compared to objects). Therefore from this step onward, we'll continue with clumps.

We can finally calculate the colors of the objects from these two datasets. If you inspect the contents of the two catalogs, you'll notice that because they were both derived from the same segmentation maps, the rows are ordered identically (they correspond to the same object/clump in both filters). But to be generic (usable even when the rows aren't ordered similarly) and display another useful program in Gnuastro, we'll use Section 7.5 [Match], page 279.

As the name suggests, Gnuastro's Match program will match rows based on distance (or aperture in 2D) in one (or two) columns. In the command below, the options relating to each catalog are placed under it for easy understanding. You give Match two catalogs (from the two different filters we derived above) as argument, and the HDUs containing them (if they are FITS files) with the `--hdu` and `--hdu2` options. The `--ccol1` and `--ccol2` options specify which columns should be matched with which in the two catalogs. With `--aperture`

you specify the acceptable error (radius in 2D), in the same units as the columns (see below for why we have requested an aperture of 0.35 arcseconds, or less than 6 HST pixels).

The `--outcols` is a very convenient feature in Match: you can use it to specify which columns from the two catalogs you want in the output (merge two input catalogs into one). If the first character is an '`a`', the respective matched column (number or name, similar to Table above) in the first catalog will be written in the output table. When the first character is a '`b`', the respective column from the second catalog will be written in the output. Also, if the first character is followed by `_all`, then all the columns from the respective catalog will be put in the output.

```
$ astmatch cat/xdf-f160w.fits           cat/xdf-f105w.fits         \
           --hdu=CLUMPS                 --hdu2=CLUMPS              \
           --ccol1=RA,DEC               --ccol2=RA,DEC             \
           --aperture=0.35/3600 --log                              \
           --outcols=a_all,bMAGNITUDE,bSN                          \
           --output=cat/xdf-f160w-f105w.fits
```

By default (when `--quiet` isn't called), the Match program will just print the number of matched rows in the standard output. If you have a look at your input catalogs, this should be the same as the number of rows in them. Let's have a look at the columns in the matched catalog:

```
$ asttable cat/xdf-f160w-f105w.fits -i
```

Indeed, its exactly the columns we wanted. There is just one confusion however: there are two `MAGNITUDE` and `SN` columns. Right now, you know that the first one was from the F160W filter, and the second was for F105W. But in one hour, you'll start doubting your self: going through your command history, trying to answer this question: "which magnitude corresponds to which filter?". You should never torture your future-self (or colleagues) like this! So, let's rename these confusing columns in the matched catalog. The FITS standard for tables stores the column names in the `TTYPE` header keywords, so let's have a look:

```
$ astfits cat/xdf-f160w-f105w.fits -h1 | grep TTYPE
```

Changing/updating the column names is as easy as updating the values to these options with the first command below, and with the second, confirm this change:

```
$ astfits cat/xdf-f160w-f105w.fits -h1                          \
          --update=TTYPE5,MAG_F160W   --update=TTYPE6,SN_F160W  \
          --update=TTYPE7,MAG_F105W   --update=TTYPE8,SN_F105W
$ asttable cat/xdf-f160w-f105w.fits -i
```

If you noticed, when running Match, the previous command, we also asked for `--log`. Many Gnuastro programs have this option to provide some detailed information on their operation in case you are curious. Here, we are using it to justify the value we gave to `--aperture`. Even though you asked for the output to be written in the `cat` directory, a listing of the contents of your current directory will show you an extra `astmatch.fits` file. Let's have a look at what columns it contains.

```
$ ls
$ asttable astmatch.log -i
```

The `MATCH_DIST` column contains the distance of the matched rows, let's have a look at the distribution of values in this column. You might be asking yourself "why should the

positions of the two filters differ when I gave MakeCatalog the same segmentation map?" The reason is that the central positions are *flux-weighted*. Therefore the `--valuesfile` dataset you give to MakeCatalog will also affect the center measurements[18]. Recall that the Spectral Energy Distribution (SED) of galaxies is not flat and they have substructure, therefore, they can have different shapes/morphologies in different filters.

Gnuastro has a simple program for basic statistical analysis. The command below will print some basic information about the distribution (minimum, maximum, median and etc), along with a cute little ASCII histogram to visually help you understand the distribution on the command-line without the need for a graphic user interface (see Section 7.1.4 [Invoking Statistics], page 215). This ASCII histogram can be useful when you just want some coarse and general information on the input dataset. It is also useful when working on a server (where you may not have graphic user interface), and finally, its fast.

```
$ aststatistics astmatch.fits -cMATCH_DIST
```

The units of this column are the same as the columns you gave to Match: in degrees. You see that while almost all the objects matched very nicely, the maximum distance is roughly 0.31 arcseconds. This is why we asked for an aperture of 0.35 arcseconds when doing the match.

We can now use AWK to find the colors. We'll ask AWK to only use rows that don't have a NaN magnitude in either filter[19]. We will also ignore columns which don't have reliable F105W measurement (with a S/N less than 7[20]).

```
$ asttable cat/xdf-f160w-f105w.fits -cMAG_F160W,MAG_F105W,SN_F105W  \
           | awk '$1!="nan" && $2!="nan" && $3>7 {print $2-$1}'     \
           > f105w-f160w.txt
```

You can inspect the distribution of colors with the Statistics program again:

```
$ aststatistics f105w-f160w.txt -c1
```

You can later use Gnuastro's Statistics program with the `--histogram` option to build a much more fine-grained histogram as a table to feed into your favorite plotting program for a much more accurate/appealing plot (for example with PGFPlots in LaTeX). If you just want a specific measure, for example the mean, median and standard deviation, you can ask for them specifically with this command:

```
$ aststatistics f105w-f160w.txt -c1 --mean --median --std
```

Some researchers prefer to have colors in a fixed aperture for all the objects. The colors we calculated above used a different segmentation map for each object. This might not satisfy some science cases. So, let's make a fixed aperture catalog. To make an catalog from fixed apertures, we should make a labeled image which has a fixed label for each aperture. That labeled image can be given to MakeCatalog instead of Segment's labeled detection image.

---

[18] To only measure the center based on the labeled pixels (and ignore the pixel values), you can ask for the columns that contain `geo` (for geometric) in them. For example `--geow1` or `--geow2` for the RA and Declination (first and second world-coordinates).

[19] This can happen even on the reference image. It is because of the current way clumps are defined in Segment when they are placed on strong gradients. It is because of high "river" values on such gradients. See Section 7.3.1 [Segment changes after publication], page 245. To avoid this problem, you can currently ask for the `--brighntessnoriver` output column.

[20] The value of 7 is taken from the clump S/N threshold in F160W (where the clumps were defined).

To generate the apertures catalog, we'll first read the positions from F160W catalog and set the other parameters of each profile to be a fixed circle of radius 5 pixels (we want all apertures to be identical in this scenario).

```
$ rm *.fits *.txt
$ asttable cat/xdf-f160w.fits -hCLUMPS -cRA,DEC                    \
           | awk '!/^#/{print NR, $1, $2, 5, 5, 0, 0, 1, NR, 1}' \
           > apertures.txt
```

We can now feed this catalog into MakeProfiles to build the apertures for us. See Section 8.1.5 [Invoking MakeProfiles], page 291, for a description of the options. The most important for this particular job is `--mforflatpix`, it tells MakeProfiles that the values in the magnitude column should be used for each pixel of a flat profile. Without it, MakeProfiles would build the profiles such that the *sum* of the pixels of each profile would have a *magnitude* (in log-scale) of the value given in that column (what you would expect when simulating a galaxy for example).

```
$ astmkprof apertures.txt --background=flat-ir/xdf-f160w.fits     \
            --clearcanvas --replace --type=int16 --mforflatpix    \
            --mode=wcs
```

The first thing you might notice in the printed information is that the profiles are not built in order. This is because MakeProfiles works in parallel, and parallel CPU operations are asynchronous. You can try running MakeProfiles with one thread (using `--numthreads=1`) to see how order is respected in that case.

Open the output `apertures.fits` file and see the result. Where the apertures overlap, you will notice that one label has replaced the other (because of the `--replace` option). In the future, MakeCatalog will be able to work with overlapping labels, but currently it doesn't. If you are interested, please join us in completing Gnuastro with added improvements like this (see task 14750[21]).

We can now feed the `apertures.fits` labeled image into MakeCatalog instead of Segment's output as shown below. In comparison with the previous MakeCatalog call, you will notice that there is no more `--clumpscat` option, since each aperture is treated as a separate "object" here.

```
$ astmkcatalog apertures.fits -h1 --zeropoint=26.27        \
               --valuesfile=nc/xdf-f105w.fits              \
               --ids --ra --dec --magnitude --sn           \
               --output=cat/xdf-f105w-aper.fits
```

This catalog has the same number of rows as the catalog produced from clumps, therefore similar to how we found colors, you can compare the aperture and clump magnitudes for example. You can also change the filter name and zeropoint magnitudes and run this command again to have the fixed aperture magnitude in the F160W filter and measure colors on apertures.

As a final step, let's go back to the original clumps-based catalogs we generated before. We'll find the objects with the strongest color and make a cutout to inspect them visually and finally, we'll see how they are located on the image.

[21] `https://savannah.gnu.org/task/index.php?14750`

First, let's see what the objects with a color more than two magnitudes look like. As you see, this is very much like the command above for selecting the colors, only instead of printing the color, we'll print the RA and Dec. With the command below, the positions of all lines with a color more than 1.5 will be put in `reddest.txt`

```
$ asttable cat/xdf-f160w-f105w.fits                                  \
           -cMAG_F160W,MAG_F105W,SN_F105W,RA,DEC                     \
           | awk '$1!="nan" && $2!="nan" && $2-$1>1.5 && $3>7        \
                 {print $4,$5}' > reddest.txt
```

We can now feed `reddest.txt` into Gnuastro's crop to see what these objects look like. To keep things clean, we'll make a directory called `crop-red` and ask Crop to save the crops in this directory. We'll also add a `-f160w.fits` suffix to the crops (to remind us which image they came from). The width of the crops will be 15 arcseconds.

```
$ mkdir crop-red
$ astcrop --mode=wcs --coordcol=3 --coordcol=4 flat-ir/xdf-f160w.fits \
          --catalog=reddest.txt --width=15/3600,15/3600               \
          --suffix=-f160w.fits --output=crop-red
```

Like the MakeProfiles command above, you might notice that the crops aren't made in order. This is because each crop is independent of the rest, therefore crops are done in parallel, and parallel operations are asynchronous. In the command above, you can change `f160w` to `f105w` to make the crops in both filters.

To view the crops more easily (not having to open ds9 for each image), you can convert the FITS crops into the JPEG format with a shell loop like below.

```
$ cd crop-red
$ for f in *.fits; do                                                 \
    astconvertt $f --fluxlow=-0.001 --fluxhigh=0.005 --invert -ojpg;  \
  done
$ cd ..
```

You can now use your general graphic user interface image viewer to flip through the images more easily. On GNOME, you can use the "Eye of GNOME" image viewer (with executable name of `eog`). Run the command below to open the first one (if you aren't using GNOME, use the command of your image viewer instead of `eog`):

```
$ eog 1-f160w.jpg
```

In Eye of GNOME, you can flip through the images and compare them visually more easily by pressing the `<SPACE>` key. Of course, the flux ranges have been chosen generically here for seeing the fainter parts. Therefore, brighter objects will be fully black.

The `for` loop above to convert the images will do the job in series: each file is converted only after the previous one is complete. If you have GNU Parallel (`https://www.gnu.org/s/parallel`), you can greatly speed up this conversion. GNU Parallel will run the separate commands simultaneously on different CPU threads in parallel. For more information on efficiently using your threads, see Section 4.4 [Multi-threaded operations], page 112. Here is a replacement for the shell `for` loop above using GNU Parallel.

```
$ cd crop-red
$ parallel astconvertt --fluxlow=-0.001 --fluxhigh=0.005 --invert   \
           -ojpg ::: *.fits
```

```
$ cd ..
```

As the final action, let's see how these objects are positioned over the dataset. DS9 has the "Region"s concept for this purpose. You just have to convert your catalog into a "region file" to feed into DS9. To do that, you can use AWK again as shown below.

```
$ awk 'BEGIN{print "# Region file format: DS9 version 4.1";     \
             print "global color=green width=2";               \
             print "fk5";}                                     \
       {printf "circle(%s,%s,1\")\n", $1, $2;}' reddest.txt    \
       > reddest.reg
```

This region file can be loaded into DS9 with its `-regions` option to display over any image (that has world coordinate system). In the example below, we'll open Segment's output and load the regions over all the extensions (to see the image and the respective clump):

```
$ ds9 -mecube seg/xdf-f160w.fits -zscale -zoom to fit    \
      -regions load all reddest.reg
```

In conclusion, we hope this extended tutorial has been a good starting point to help in your exciting research. If this book or any of the programs in Gnuastro have been useful for your research, please cite the respective papers and share your thoughts and suggestions with us (it can be very encouraging). All Gnuastro programs have a `--cite` option to help you cite the authors' work more easily. Just note that it may be necessary to cite additional papers for different programs, so please try it out for any program you used.

```
$ astmkcatalog --cite
$ astnoisechisel --cite
```

2.3 Detecting large extended targets

The outer wings of large and extended objects can sink into the noise very gradually and can have a large variety of shapes (for example due to tidal interactions). Therefore separating the outer boundaries of the galaxies from the noise can be particularly tricky. Besides causing an under-estimation in the total estimated brightness of the target, failure to detect such faint wings will also cause a bias in the noise measurements, thereby hampering the accuracy of any measurement on the dataset. Therefore even if they don't constitute a significant fraction of the target's light, or aren't your primary target, these regions must not be ignored. In this tutorial, we'll walk you through the strategy of detecting such targets using Section 7.2 [NoiseChisel], page 225.

Don't start with this tutorial: If you haven't already completed Section 2.2 [General program usage tutorial], page 24, we strongly recommend going through that tutorial before starting this one. Basic features like access to this book on the command-line, the configuration files of Gnuastro's programs, benefiting from the modular nature of the programs, viewing multi-extension FITS files, or using NoiseChisel's outputs are discussed in more detail there.

We'll try to detect the faint tidal wings of the beautiful M51 group[22] in this tutorial. We'll use a dataset/image from the public Sloan Digital Sky Survey (`http://www.sdss.org/`), or SDSS. Due to its more peculiar low surface brightness structure/features, we'll focus on the dwarf companion galaxy of the group (or NGC 5195). To get the image, you can use SDSS's Simple field search (`https://dr12.sdss.org/fields`) tool. As long as it is covered by the SDSS, you can find an image containing your desired target either by providing a standard name (if it has one), or its coordinates. To access the dataset we will use here, write `NGC5195` in the "Object Name" field and press "Submit" button.

Type the example commands: Try to type the example commands on your terminal and use the history feature of your command-line (by pressing the "up" button to retrieve previous commands). Don't simply copy and paste the commands shown here. This will help simulate future situations when you are processing your own datasets.

You can see the list of available filters under the color image. For this demonstration, we'll use the r-band filter image. By clicking on the "r-band FITS" link, you can download the image. Alternatively, you can just run the following command to download it with GNU Wget[23]. To keep things clean, let's also put it in a directory called `ngc5195`. With the `-O` option, we are asking Wget to save the downloaded file with a more manageable name: `r.fits.bz2` (this is an r-band image of NGC 5195, which was the directory name).

```
$ mkdir ngc5195
$ cd ngc5195
$ topurl=https://dr12.sdss.org/sas/dr12/boss/photoObj/frames
$ wget $topurl/301/3716/6/frame-r-003716-6-0117.fits.bz2 -Or.fits.bz2
```

This server keeps the files in a Bzip2 compressed file format. So we'll first decompress it with the following command. By convention, compression programs delete the original file (compressed when uncompressing, or uncompressed when compressing). To keep the original file, you can use the `--keep` or `-k` option which is available in most compression programs for this job. Here, we don't need the compressed file any more, so we'll just let `bunzip` delete it for us and keep the directory clean.

```
$ bunzip2 r.fits.bz2
```

Let's see how NoiseChisel operates on it with its default parameters:

```
$ astnoisechisel r.fits -h0
```

As described in Section 7.2.2.3 [NoiseChisel output], page 241, NoiseChisel's default output is a multi-extension FITS file. A method to view them effectively and easily is discussed in Section B.1.1 [Viewing multiextension FITS images], page 469. For more on tweaking NoiseChisel and optimizing its output for archiving or sending to colleagues, see the NoiseChisel part of the previous tutorial in Section 2.2 [General program usage tutorial], page 24.

[22] `https://en.wikipedia.org/wiki/M51_Group`

[23] To make the command easier to view on screen or in a page, we have defined the top URL of the image as the `topurl` shell variable. You can just replace the value of this variable with `$topurl` in the `wget` command.

Open the output `r_detected.fits` file and have a look at the extensions, the first extension is only meta-data and contains NoiseChisel's configuration parameters. The rest are the Sky-subtracted input, the detection map, Sky values and Sky standard deviation.

Flipping through the extensions in a FITS viewer (for example SAO DS9), you will see that the Sky-subtracted image looks reasonable (there are no major artifacts due to bad Sky subtraction compared to the input). The second extension also seems reasonable with a large detection map that covers the whole of NGC5195, but also extends beyond towards the bottom of the image. Try going back and forth between the `DETECTIONS` and `SKY` extensions, you will notice that there is still significant signal beyond the detected pixels. You can tell that this signal belongs to the galaxy because the far-right side of the image is dark and the brighter tiles (that weren't interpolated) are surrounding the detected pixels.

The fact that signal from the galaxy remains in the Sky dataset shows that you haven't done a good detection. Generally, any time your target is much larger than the tile size and the signal is almost flat (like this case), this *will* happen. Therefore, when there are large objects in the dataset, **the best place** to check the accuracy of your detection is the estimated Sky image.

When dominated by the background, noise has a symmetric distribution. However, signal is not symmetric (we don't have negative signal). Therefore when non-constant signal is present in a noisy dataset, the distribution will be positively skewed. This skewness is a good measure of how much signal we have in the distribution. The skewness can be accurately measured by the difference in the mean and median: assuming no strong outliers, the more distant they are, the more skewed the dataset is. For more see Section 7.1.3.3 [Quantifying signal in a tile], page 213.

However, skewness is only a proxy for signal when the signal has structure (varies per pixel). Therefore, when it is approximately constant over a whole tile, or sub-set of the image, the signal's effect is just to shift the symmetric center of the noise distribution to the positive and there won't be any skewness (major difference between the mean and median): this positive[24] shift that preserves the symmetric distribution is the Sky value. When there is a gradient over the dataset, different tiles will have different constant shifts/Sky-values, for example see Figure 11 of Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`).

To get less scatter in measuring the mean and median (and thus better estimate the skewness), you will need a larger tile. So let's play with the tessellation a little to see how it affects the result. In Gnuastro, you can see the option values (`--tilesize` in this case) by adding the `-P` option to your last command. Try running NoiseChisel with `-P` to see its default tile size.

You can clearly see that the default tile size is indeed much smaller than this (huge) galaxy and its tidal features. As a result, NoiseChisel was unable to identify the skewness within the tiles under the outer parts of M51 and NGC 5159 and the threshold has been over-estimated on those tiles. To see which tiles were used for estimating the quantile threshold (no skewness was measured), you can use NoiseChisel's `--checkqthresh` option:

```
$ astnoisechisel r.fits -h0 --checkqthresh
```

Notice how this option doesn't allow NoiseChisel to finish. NoiseChisel aborted after finding the quantile thresholds. When you call any of NoiseChisel's `--check*` options, by

[24] In processed images, where the Sky value can be over-estimated, this constant shift can be negative.

default, it will abort as soon as all the check steps have been written in the check file (a multi-extension FITS file). This allows you to focus on the problem you wanted to check as soon as possible (you can disable this feature with the `--continueaftercheck` option).

To optimize the threshold-related settings for this image, let's playing with this quantile threshold check image a little. Don't forget that "*Good statistical analysis is not a purely routine matter, and generally calls for more than one pass through the computer*" (Anscombe 1973, see Section 1.2 [Science and its tools], page 2). A good scientist must have a good understanding of her tools to make a meaningful analysis. So don't hesitate in playing with the default configuration and reviewing the manual when you have a new dataset in front of you. Robust data analysis is an art, therefore a good scientist must first be a good artist.

The first extension of `r_qthresh.fits` (`CONVOLVED`) is the convolved input image where the threshold(s) is defined and applied. For more on the effect of convolution and thresholding, see Sections 3.1.1 and 3.1.2 of Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`). The second extension (`QTHRESH_ERODE`) has a blank value for all the pixels of any tile that was identified as having significant signal. The next two extensions (`QTHRESH_NOERODE` and `QTHRESH_EXPAND`) are the other two quantile thresholds that are necessary in NoiseChisel's later steps. Every step in this file is repeated on the three thresholds.

Play a little with the color bar of the `QTHRESH_ERODE` extension, you clearly see how the non-blank tiles around NGC 5195 have a gradient. As one line of attack against discarding too much signal below the threshold, NoiseChisel rejects outlier tiles. Go forward by three extensions to `VALUE1_NO_OUTLIER` and you will see that many of the tiles over the galaxy have been removed in this step. For more on the outlier rejection algorithm, see the latter half of Section 7.1.3.3 [Quantifying signal in a tile], page 213.

However, the default outlier rejection parameters weren't enough, and when you play with the color-bar, you see that the faintest parts of the galaxy outskirts still remain. Therefore have two strategies for approaching this problem: 1) Increase the tile size to get more accurate measurements of skewness. 2) Strengthen the outlier rejection parameters to discard more of the tiles with signal. Fortunately in this image we have a sufficiently large region on the right of the image that the galaxy doesn't extend to. So we can use the more robust first solution. In situations where this doesn't happen (for example if the field of view in this image was shifted to have more of M51 and less sky) you are limited to a combination of the two solutions or just to the second solution.

Skipping convolution for faster tests: The slowest step of NoiseChisel is the convolution of the input dataset. Therefore when your dataset is large (unlike the one in this test), and you are not changing the input dataset or kernel in multiple runs (as in the tests of this tutorial), it is faster to do the convolution separately once (using Section 6.3 [Convolve], page 177) and use NoiseChisel's `--convolved` option to directly feed the convolved image and avoid convolution. For more on `--convolved`, see Section 7.2.2.1 [NoiseChisel input], page 231.

To identify the skewness caused by the flat NGC 5195 and M51 tidal features on the tiles under it, we thus have to choose a tile size that is larger than the gradient of the signal. Let's try a 100 by 100 tile size:

```
$ astnoisechisel r.fits -h0 --tilesize=100,100 --checkqthresh
```

You can clearly see the effect of this increased tile size: the tiles are much larger ($\times 4$ in area) and when you look into `VALUE1_NO_OUTLIER`, you see that almost all the tiles under the galaxy have been discarded and we are only left with tiles in the right-most part of the image. The next group of extensions (those ending with `_INTERP`), give a value to all blank tiles based on the nearest tiles with a measurement. The following group of extensions (ending with `_SMOOTH`) have smoothed the interpolated image to avoid sharp cuts on tile edges.

Inspecting `THRESH1_SMOOTH`, you can see that there is no longer any significant gradient and no major signature of NGC 5195 exists. But before finishing the quantile threshold, let's have a closer look at the final extension (`QTHRESH-APPLIED`) which is thresholded image. Slide the dynamic range in your FITS viewer so 0 valued pixels are black and all non-zero pixels are white. You will see that the black holes are not evenly distributed. Those that follow the tail of NGC 5195 are systematically smaller than those in the far-right of the image. This suggests that we can decrease the quantile threshold (`--qthresh`) even further: there is still signal down there!

```
$ rm r_qthresh.fits
$ astnoisechisel r.fits -h0 --tilesize=100,100 --qthresh=0.2
```

Since the quantile threshold of the previous command was satisfactory, we finally removed `--checkqthresh` to let NoiseChisel proceed until completion. Looking at the `DETECTIONS` extension of NoiseChisel's output, we see the right-ward edges in particular have many holes that are fully surrounded by signal and the signal stretches out in the noise very thinly. This suggests that there is still signal that can be detected. You can confirm this guess by looking at the `SKY` extension to see that indeed, we still have traces of the galaxy outskirts there. Therefore, we should dig deeper into the noise.

Let's decrease the growth quantile (for larger/deeper growth into the noise, with `--detgrowquant`) and increase the size of holes that can be filled (if they are fully surrounded by signal, with `--detgrowmaxholesize`).

```
$ astnoisechisel r.fits -h0 --tilesize=100,100 --qthresh=0.2          \
                 --detgrowquant=0.65 --detgrowmaxholesize=10000
```

Looking into the output, we now clearly see that the tidal features of M51 and NGC 5195 are detected nicely in the same direction as expected (towards the bottom right side of the image). However, as discussed above, the best measure of good detection is the noise, not the detections themselves. So let's look at the Sky and its Standard deviation. The Sky still has a very faint shadow of the galaxy outskirts (the values on the left are very slightly larger than those on the right).

Let's calculate this gradient as a function of noise. First we'll collapse the image along the second (vertical) FITS dimension to have a 1D array. Then we'll calculate the top and bottom values of this array and define that as the gradient. Then we'll estimate the mean standard deviation over the image and divide it by the first value. The first two commands are just for demonstration of the collapsed dataset:

```
$ astarithmetic r_detected.fits 2 collapse-mean -hSKY -ocollapsed.fits
$ asttable collapsed.fits
$ skydiff=$(astarithmetic r_detected.fits 2 collapse-mean set-i   \
                          i maxvalue i minvalue - -hSKY -q)
$ echo $skydiff
```

```
$ std=$(aststatistics r_detected.fits -hSKY_STD --mean)
$ echo $std
$ echo "$std $skydiff" | awk '{print $1/$2}'
```

This gradient in the Sky (output of first command below) is much less (by more than 20 times) than the standard deviation (final extension). So we can stop configuring NoiseChisel at this point in the tutorial. We leave further configuration for a more accurate detection to you as an exercise.

In this shallow image, this extent may seem too far deep into the noise for visual confirmation. Therefore, if the statistical argument above, to justify the reality of this extended structure, hasn't convinced you, see the deep images of this system in Watkins et al. [2015] (`https://arxiv.org/abs/1501.04599`), or a 12 hour deep image of this system (with a 12-inch telescope): `https://i.redd.it/jfqgpqg0hfk11.jpg`[25].

Now that we know this detection is real, let's measure how deep we carved the signal out of noise. For this measurement, we'll need to estimate the average flux on the outer edges of the detection. Fortunately all this can be done with a few simple commands (and no higher-level language mini-environments) using Section 6.2 [Arithmetic], page 161, and Section 7.4 [MakeCatalog], page 255. First, let's give a separate label to all the connected pixels of NoiseChisel's detection map:

```
$ astarithmetic r_detected.fits 2 connected-components -hDETECTIONS \
                -olabeled.fits
```

Of course, you can find the the label of the main galaxy visually, but to have a little more fun, lets do this automatically. The M51 group detection is by far the largest detection in this image. We can thus easily find the ID/label that corresponds to it. We'll first run MakeCatalog to find the area of all the detections, then we'll use AWK to find the ID of the largest object and keep it as a shell variable (`id`):

```
$ astmkcatalog labeled.fits --ids --geoarea -h1 -ocat.txt
$ id=$(awk '!/^#/{if($2>max) {id=$1; max=$2}} END{print id}' cat.txt)
$ echo $id
```

To separate the outer edges of the detections, we'll need to "erode" the detections. We'll erode two times (one time would be too thin for such a huge object), using a maximum connectivity of 2 (8-connected neighbors). We'll then save the output in `eroded.fits`.

```
$ astarithmetic r_detected.fits 0 gt 2 erode -hDETECTIONS -oeroded.fits
```

We should now just keep the pixels that have the ID of the M51 group, but a value of 0 in `erode.fits`. We'll keep the output in `boundary.fits`.

```
$ astarithmetic labeled.fits $id eq eroded.fits 0 eq and          \
                -g1 -oboundary.fits
```

Open the image and have a look. You'll see that the detected edge of the M51 group is now clearly visible. You can use `boundary.fits` to mark (set to blank) this boundary on the input image and get a visual feeling of how far it extends:

```
$ astarithmetic r.fits boundary.fits nan where -ob-masked.fits -h0
```

[25] The image is taken from this Reddit discussion: `https://www.reddit.com/r/Astronomy/comments/9d6x0q/12_hours_of_exposure_on_the_whirlpool_galaxy/`

To quantify how deep we have detected the low-surface brightness regions, we'll use the command below. In short it just divides all the non-zero pixels of `boundary.fits` in the Sky subtracted input (first extension of NoiseChisel's output) by the pixel standard deviation of the same pixel. This will give us a signal-to-noise ratio image. The mean value of this image shows the level of surface brightness that we have achieved.

You can also break the command below into multiple calls to Arithmetic and create temporary files to understand it better. However, if you have a look at Section 6.2.1 [Reverse polish notation], page 162, and Section 6.2.2 [Arithmetic operators], page 162, you should be able to easily understand what your computer does when you run this command[26].

```
$ astarithmetic r_detected.fits boundary.fits not nan where \
                r_detected.fits /                           \
                meanvalue                                   \
                -hINPUT-NO-SKY -h1 -hSKY_STD --quiet
```

The outer wings where therefore non-parametrically detected until S/N ≈ 0.05!

This is very good! But the signal-to-noise ratio is a relative measurement. Let's also measure the depth of our detection in absolute surface brightness units; or magnitudes per square arcseconds. To find out, we'll first need to calculate how many pixels of this image are in one arcsecond-squared. Fortunately the world coordinate system (or WCS) meta data of Gnuastro's output FITS files (in particular the `CDELT` keywords) give us this information.

```
$ n=$(astfits r_detected.fits -h1                                  \
             | awk '/CDELT1/ {p=1/($3*3600); print p*p}')
$ echo $n
```

Now, let's calculate the average sky-subtracted flux in the border region per pixel.

```
$ f=$(astarithmetic r_detected.fits boundary.fits not nan where set-i \
                    i sumvalue i numvalue / -q -hINPUT-NO-SKY)
$ echo $f
```

We can just multiply the two to get the average flux on this border in one arcsecond squared. We also have the r-band SDSS zeropoint magnitude[27] to be 24.80. Therefore we can get the surface brightness of the outer edge (in magnitudes per arcsecond squared) using the following command. Just note that `log` in AWK is in base-2 (not 10), and that AWK doesn't have a `log10` operator. So we'll do an extra division by `log(10)` to correct for this.

```
$ z=24.80
$ echo "$n $f $z" | awk '{print -2.5*log($1*$2)/log(10)+$3}'
--> 30.0646
```

This shows that on a single-exposure SDSS image, we have reached a surface brightness limit of roughly 30 magnitudes per arcseconds squared!

In interpreting this value, you should just have in mind that NoiseChisel works based on the contiguity of signal in the pixels. Therefore the larger the object, the deeper NoiseChisel

[26] `boundary.fits` (extension `1`) is a binary (0 or 1 valued) image. Applying the `not` operator on it, just flips all its pixels. Through the `where` operator, we are setting all the newly 1-valued pixels in `r_detected.fits` (extension `INPUT-NO-SKY`) to NaN/blank. In the second line, we are dividing all the non-blank values by `r_detected.fits` (extension `SKY_STD`). This gives the signal-to-noise ratio for each of the pixels on the boundary. Finally, with the `meanvalue` operator, we are taking the mean value of all the non-blank pixels and reporting that as a single number.

[27] From `http://classic.sdss.org/dr7/algorithms/fluxcal.html`

can carve it out of the noise. In other words, this reported depth, is only for this particular object and dataset, processed with this particular NoiseChisel configuration: if the M51 group in this image was larger/smaller than this, or if the image was larger/smaller, or if we had used a different configuration, we would go deeper/shallower.

The NoiseChisel configuration found here is NOT GENERIC for any large object: As you saw above, the reason we chose this particular configuration for NoiseChisel to detect the wings of the M51 group was strongly influenced by this particular object in this particular image. When low surface brightness signal takes over such a large fraction of your dataset (and you want to accurately detect/account for it), to make sure that it is successfully detected, you will need some manual checking, intervention, or customization. In other words, to make sure that your noise measurements are least affected by the signal[28].

To avoid typing all these options every time you run NoiseChisel on this image, you can use Gnuastro's configuration files, see Section 4.2 [Configuration files], page 106. For an applied example of setting/using them, see Section 2.2 [General program usage tutorial], page 24.

To continue your analysis of such datasets with extended emission, you can use Section 7.3 [Segment], page 243, to identify all the "clumps" over the diffuse regions: background galaxies and foreground stars.

```
$ astsegment r_detected.fits
```

Open the output `r_detected_segmented.fits` as a multi-extension data cube like before and flip through the first and second extensions to see the detected clumps (all pixels with a value larger than 1). To optimize the parameters and make sure you have detected what you wanted, its highly recommended to visually inspect the detected clumps on the input image.

For visual inspection, you can make a simple shell script like below. It will first call MakeCatalog to estimate the positions of the clumps, then make an SAO ds9 region file and open ds9 with the image and region file. Recall that in a shell script, the numeric variables (like `$1`, `$2`, and `$3` in the example below) represent the arguments given to the script. But when used in the AWK arguments, they refer to column numbers.

To create the shell script, using your favorite text editor, put the contents below into a file called `check-clumps.sh`. Recall that everything after a `#` is just comments to help you understand the command (so read them!). Also note that if you are copying from the PDF version of this book, fix the single quotes in the AWK command.

```
#! /bin/bash
set -e    # Stop execution when there is an error.
set -u    # Stop execution when a variable is not initialized.

# Run MakeCatalog to write the coordinates into a FITS table.
```

[28] In the future, we may add capabilities to optionally automate some of the choices made here, please join us in doing this if you are interested. However, given the many problems in existing "smart" solutions, such automatic changing of the configuration may cause more problems than they solve. So even when they are implemented, we would strongly recommend manual checks and intervention for a robust analysis.

```
# Default output is '$1_cat.fits'.
astmkcatalog $1.fits --clumpscat --ids --ra --dec

# Use Gnuastro's Table program to read the RA and Dec columns of the
# clumps catalog (in the 'CLUMPS' extension). Then pipe the columns
# to AWK for saving as a DS9 region file.
asttable $1"_cat.fits" -hCLUMPS -cRA,DEC                                \
         | awk 'BEGIN { print "# Region file format: DS9 version 4.1"; \
                        print "global color=green width=1";            \
                        print "fk5" }                                   \
                { printf "circle(%s,%s,1\")\n", $1, $2 }' > $1.reg

# Show the image (with the requested color scale) and the region file.
ds9 -geometry 1800x3000 -mecube $1.fits -zoom to fit                    \
    -scale limits $2 $3 -regions load all $1.reg

# Clean up (delete intermediate files).
rm $1"_cat.fits" $1.reg
```

Finally, you just have to activate the script's executable flag with the command below. This will enable you to directly/easily call the script as a command.

```
$ chmod +x check-clumps.sh
```

This script doesn't expect the `.fits` suffix of the input's filename as the first argument. Because the script produces intermediate files (a catalog and DS9 region file, which are later deleted). However, we don't want multiple instances of the script (on different files in the same directory) to collide (read/write to the same intermediate files). Therefore, we have used suffixes added to the input's name to identify the intermediate files. Note how all the `$1` instances in the commands (not within the AWK command[29]) are followed by a suffix. If you want to keep the intermediate files, put a `#` at the start of the last line.

The few, but high-valued, bright pixels in the central parts of the galaxies can hinder easy visual inspection of the fainter parts of the image. With the second and third arguments to this script, you can set the numerical values of the color map (first is minimum/black, second is maximum/white). You can call this script with any[30] output of Segment (when `--rawoutput` is *not* used) with a command like this:

```
$ ./check-clumps.sh r_detected_segmented -0.1 2
```

Go ahead and run this command. You will see the intermediate processing being done and finally it opens SAO DS9 for you with the regions superimposed on all the extensions of Segment's output. The script will only finish (and give you control of the command-line) when you close DS9. If you need your access to the command-line before closing DS9, add a `&` after the end of the command above.

While DS9 is open, slide the dynamic range (values for black and white, or minimum/maximum values in different color schemes) and zoom into various regions of the

[29] In AWK, `$1` refers to the first column, while in the shell script, it refers to the first argument.

[30] Some modifications are necessary based on the input dataset: depending on the dynamic range, you have to adjust the second and third arguments. But more importantly, depending on the dataset's world coordinate system, you have to change the region `width`, in the AWK command. Otherwise the circle regions can be too small/large.

M51 group to see if you are satisfied with the detected clumps. Don't forget that through the "Cube" window that is opened along with DS9, you can flip through the extensions and see the actual clumps also. The questions you should be asking your self are these: 1) Which real clumps (as you visually *feel*) have been missed? In other words, is the *completeness* good? 2) Are there any clumps which you *feel* are false? In other words, is the *purity* good?

Note that completeness and purity are not independent of each other, they are anti-correlated: the higher your purity, the lower your completeness and vice-versa. You can see this by playing with the purity level using the `--snquant` option. Run Segment as shown above again with `-P` and see its default value. Then increase/decrease it for higher/lower purity and check the result as before. You will see that if you want the best purity, you have to sacrifice completeness and vice versa.

One interesting region to inspect in this image is the many bright peaks around the central parts of M51. Zoom into that region and inspect how many of them have actually been detected as true clumps. Do you have a good balance between completeness and purity? Also look out far into the wings of the group and inspect the completeness and purity there.

An easier way to inspect completeness (and only completeness) is to mask all the pixels detected as clumps and visually inspecting the rest of the pixels. You can do this using Arithmetic in a command like below. For easy reading of the command, we'll define the shell variable `i` for the image name and save the output in `masked.fits`.

```
$ i=r_detected_segmented.fits
$ astarithmetic $i $i 0 gt nan where -hINPUT -hCLUMPS -omasked.fits
```

Inspecting `masked.fits`, you can see some very diffuse peaks that have been missed, especially as you go farther away from the group center and into the diffuse wings. This is due to the fact that with this configuration, we have focused more on the sharper clumps. To put the focus more on diffuse clumps, you can use a wider convolution kernel. Using a larger kernel can also help in detecting the existing clumps to fainter levels (thus better separating them from the surrounding diffuse signal).

You can make any kernel easily using the `--kernel` option in Section 8.1 [MakeProfiles], page 284. But note that a larger kernel is also going to wash-out many of the sharp/small clumps close to the center of M51 and also some smaller peaks on the wings. Please continue playing with Segment's configuration to obtain a more complete result (while keeping reasonable purity). We'll finish the discussion on finding true clumps at this point.

The properties of the background objects can then easily be measured using Section 7.4 [MakeCatalog], page 255. To measure the properties of the background objects (detected as clumps over the diffuse region), you shouldn't mask the diffuse region. When measuring clump properties with Section 7.4 [MakeCatalog], page 255, the ambient flux (from the diffuse region) is calculated and subtracted. If the diffuse region is masked, its effect on the clump brightness cannot be calculated and subtracted.

To keep this tutorial short, we'll stop here. See Section 2.2 [General program usage tutorial], page 24, and Section 7.3 [Segment], page 243, for more on using Segment, producing catalogs with MakeCatalog and using those catalogs.

Finally, if this book or any of the programs in Gnuastro have been useful for your research, please cite the respective papers and share your thoughts and suggestions with us (it can be very encouraging). All Gnuastro programs have a `--cite` option to help you cite

the authors' work more easily. Just note that it may be necessary to cite additional papers for different programs, so please use `--cite` with any program that has been useful in your work.

```
$ astmkcatalog --cite
$ astnoisechisel --cite
```

2.4 Hubble visually checks and classifies his catalog

In 1924 Hubble[31] announced his discovery that some of the known nebulous objects are too distant to be within the the Milky Way (or Galaxy) and that they were probably distant Galaxies[32] in their own right. He had also used them to show that the redshift of the nebulae increases with their distance. So now he wants to study them more accurately to see what they actually are. Since they are nebulous or amorphous, they can't be modeled (like stars that are always a point) easily. So there is no better way to distinguish them than to visually inspect them and see if it is possible to classify these nebulae or not.

Hubble has stored all the FITS images of the objects he wants to visually inspect in his `/mnt/data/images` directory. He has also stored his catalog of extra-galactic nebulae in `/mnt/data/catalogs/extragalactic.txt`. Any normal user on his GNU/Linux system (including himself) only has read access to the contents of the `/mnt/data` directory. He has done this by running this command as root:

```
# chmod -R 755 /mnt/data
```

Hubble has done this intentionally to avoid mistakenly deleting or modifying the valuable images he has taken at Mount Wilson while he is working as an ordinary user. Retaking all those images and data is simply not an option. In fact they are also in another hard disk (`/dev/sdb1`). So if the hard disk which stores his GNU/Linux distribution suddenly malfunctions due to work load, his data is not in harms way. That hard disk is only mounted to this directory when he wants to use it with the command:

```
# mount /dev/sdb1 /mnt/data
```

In short, Hubble wants to keep his data safe and fortunately by default Gnuastro allows for this. Hubble creates a temporary `visualcheck` directory in his home directory for this check. He runs the following commands to make the directory and change to it[33]:

```
$ mkdir ~/visualcheck
$ cd ~/visualcheck
$ pwd
/home/edwin/visualcheck
$ ls
```

[31] Edwin Powell Hubble (1889 – 1953 A.D.) was an American astronomer who can be considered as the father of extra-galactic astronomy, by proving that some nebulae are too distant to be within the Galaxy. He then went on to show that the universe appears to expand and also done a visual classification of the galaxies that is known as the Hubble fork.

[32] Note that at that time, "Galaxy" was a proper noun used to refer to the Milky way. The concept of a galaxy as we define it today had not yet become common. Hubble played a major role in creating today's concept of a galaxy.

[33] The `pwd` command is short for "Print Working Directory" and `ls` is short for "list" which shows the contents of a directory.

Hubble has multiple images in `/mnt/data/images`, some of his targets might be on the edges of an image and so several images need to be stitched to give a good view of them. Also his extra-galactic targets belong to various pointings in the sky, so they are not in one large image. Gnuastro's Crop is just the program he wants. The catalog in `extragalactic.txt` is a plain text file which stores the basic information of all his known 200 extra-galactic nebulae. If you don't have any particular catalog and accompanying image, you can use one the Hubble Space Telescope F160W catalog that we produced in Section 2.2 [General program usage tutorial], page 24, along with the accompanying image (specify the exact image name, not `/mnt/data/images/*.fits`). You can select the brightest galaxies for an easier classification.

In its second column, the catalog has each object's Right Ascension (the first column is a label he has given to each object), and in the third, the object's declination (which he specifies with the `--coordcol` option). Also, since the coordinates are in the world coordinate system (WCS, not pixel positions) units, he adds `--mode=wcs`.

```
$ astcrop --coordcol=2 --coordcol=3 /mnt/data/images/*.fits     \
          --mode=wcs /mnt/data/catalogs/extragalactic.txt
Crop started on Tue Jun  14 10:18:11 1932
  ---- ./4_crop.fits                  1 1
  ---- ./2_crop.fits                  1 1
  ---- ./1_crop.fits                  1 1
[[[ Truncated middle of list ]]]
  ---- ./198_crop.fits                1 1
  ---- ./195_crop.fits                1 1
  - 200 images created.
  - 200 were filled in the center.
  - 0 used more than one input.
Crop finished in:  2.429401 (seconds)
```

Hubble already knows that thread allocation to the the CPU cores is asynchronous. Hence each time you run it, the order of which job gets done first differs. When using Crop the order of outputs is irrelevant since each crop is independent of the rest. This is why the crops are not necessarily created in the same input order. He is satisfied with the default width of the outputs (which he inspected by running `$ astcrop -P`). If he wanted a different width for the cropped images, he could do that with the `--wwidth` option which accepts a value in arc-seconds. When he lists the contents of the directory again he finds his 200 objects as separate FITS images.

```
$ ls
1_crop.fits 2_crop.fits ... 200_crop.fits
```

The FITS image format was not designed for efficient/fast viewing, but mainly for accurate storing of the data. So he chooses to convert the cropped images to a more common image format to view them more quickly and easily through standard image viewers (which load much faster than FITS image viewer). JPEG is one of the most recognized image formats that is supported by most image viewers. Fortunately Gnuastro has just such a tool to convert various types of file types to and from each other: ConvertType. Hubble has already heard of GNU Parallel from one of his colleagues at Mount Wilson Observatory. It allows multiple instances of a command to be run simultaneously on the system, so he uses it in conjunction with ConvertType to convert all the images to JPEG.

```
$ parallel astconvertt -ojpg ::: *_crop.fits
```

For his graphical user interface Hubble is using GNOME which is the default in most distributions in GNU/Linux. The basic image viewer in GNOME is the Eye of GNOME, which has the executable file name `eog`[34]. Since he has used it before, he knows that once it opens an image, he can use the `ENTER` or `SPACE` keys on the keyboard to go to the next image in the directory or the `Backspace` key to go the previous image. So he opens the image of the first object with the command below and with his cup of coffee in his other hand, he flips through his targets very fast to get a good initial impression of the morphologies of these extra-galactic nebulae.

```
$ eog 1_crop.jpg
```

Hubble's cup of coffee is now finished and he also got a nice general impression of the shapes of the nebulae. He tentatively/mentally classified the objects into three classes while doing the visual inspection. One group of the nebulae have a very simple elliptical shape and seem to have no internal special structure, so he gives them code 1. Another clearly different class are those which have spiral arms which he associates with code 2 and finally there seems to be a class of nebulae in between which appear to have a disk but no spiral arms, he gives them code 3.

Now he wants to know how many of the nebulae in his extra-galactic sample are within each class. Repeating the same process above and writing the results on paper is very time consuming and prone to errors. Fortunately Hubble knows the basics of GNU Bash shell programming, so he writes the following short script with a loop to help him with the job. After all, computers are made for us to operate and knowing basic shell programming gives Hubble this ability to creatively operate the computer as he wants. So using GNU Emacs[35] (his favorite text editor) he puts the following text in a file named `classify.sh`.

```
for name in *.jpg
do
    eog $name &
    processid=$!
    echo -n "$name belongs to class: "
    read class
    echo $name $class >> classified.txt
    kill $processid
done
```

Fortunately GNU Emacs or even simpler editors like Gedit (part of the GNOME graphical user interface) will display the variables and shell constructs in different colors which can really help in understanding the script. Put simply, the `for` loop gets the name of each JPEG file in the directory this script is run in and puts it in `name`. In the shell, the value of a variable is used by putting a `$` sign before the variable name. Then Eye of GNOME is run on the image in the background to show him that image and its process ID is saved internally (this is necessary to close Eye of GNOME later). The shell then prompts the user to specify a class and after saving it in `class`, it prints the file name and the given

[34] Eye of GNOME is only available for users of the GNOME graphical desktop environment which is the default in most GNU/Linux distributions. If you use another graphical desktop environment, replace `eog` with any other image viewer.

[35] This can be done with any text editor

class in the next line of a file named `classified.txt`. To make the script executable (so he can run it later any time he wants) he runs:

```
$ chmod +x classify.sh
```

Now he is ready to do the classification, so he runs the script:

```
$ ./classify.sh
```

In the end he can delete all the JPEG and FITS files along with Crop's log file with the following short command. The only files remaining are the script and the result of the classification.

```
$ rm *.jpg *.fits astcrop.txt
$ ls
classified.txt   classify.sh
```

He can now use `classified.txt` as input to a plotting program to plot the histogram of the classes and start making interpretations about what these nebulous objects that are outside of the Galaxy are.

3 Installation

The latest released version of Gnuastro source code is always available at the following URL:

`http://ftpmirror.gnu.org/gnuastro/gnuastro-latest.tar.gz`

Section 1.1 [Quick start], page 1, describes the commands necessary to configure, build, and install Gnuastro on your system. This chapter will be useful in cases where the simple procedure above is not sufficient, for example your system lacks a mandatory/optional dependency (in other words, you can't pass the `$ ./configure` step), or you want greater customization, or you want to build and install Gnuastro from other random points in its history, or you want a higher level of control on the installation. Thus if you were happy with downloading the tarball and following Section 1.1 [Quick start], page 1, then you can safely ignore this chapter and come back to it in the future if you need more customization.

Section 3.1 [Dependencies], page 61, describes the mandatory, optional and bootstrapping dependencies of Gnuastro. Only the first group are required/mandatory when you are building Gnuastro using a tarball (see Section 3.2.1 [Release tarball], page 71), they are very basic and low-level tools used in most astronomical software, so you might already have them installed, if not they are very easy to install as described for each. Section 3.2 [Downloading the source], page 71, discusses the two methods you can obtain the source code: as a tarball (a significant snapshot in Gnuastro's history), or the full history[1]. The latter allows you to build Gnuastro at any random point in its history (for example to get bug fixes or new features that are not released as a tarball yet).

The building and installation of Gnuastro is heavily customizable, to learn more about them, see Section 3.3 [Build and install], page 76. This section is essentially a thorough explanation of the steps in Section 1.1 [Quick start], page 1. It discusses ways you can influence the building and installation. If you encounter any problems in the installation process, it is probably already explained in Section 3.3.5 [Known issues], page 89. In Appendix B [Other useful software], page 469, the installation and usage of some other free software that are not directly required by Gnuastro but might be useful in conjunction with it is discussed.

3.1 Dependencies

A minimal set of dependencies are mandatory for building Gnuastro from the standard tarball release. If they are not present you cannot pass Gnuastro's configuration step. The mandatory dependencies are therefore very basic (low-level) tools which are easy to obtain, build and install, see Section 3.1.1 [Mandatory dependencies], page 62, for a full discussion.

If you have the packages of Section 3.1.2 [Optional dependencies], page 64, Gnuastro will have additional functionality (for example converting FITS images to JPEG or PDF). If you are installing from a tarball as explained in Section 1.1 [Quick start], page 1, you can stop reading after this section. If you are cloning the version controlled source (see Section 3.2.2 [Version controlled source], page 72), an additional bootstrapping step is required before configuration and its dependencies are explained in Section 3.1.3 [Bootstrapping dependencies], page 66.

[1] Section 3.1.3 [Bootstrapping dependencies], page 66, are required if you clone the full history.

Your operating system's package manager is an easy and convenient way to download and install the dependencies that are already pre-built for your operating system. In Section 3.1.4 [Dependencies from package managers], page 68, we'll list some common operating system package manager commands to install the optional and mandatory dependencies.

3.1.1 Mandatory dependencies

The mandatory Gnuastro dependencies are very basic and low-level tools. They all follow the same basic GNU based build system (like that shown in Section 1.1 [Quick start], page 1), so even if you don't have them, installing them should be pretty straightforward. In this section we explain each program and any specific note that might be necessary in the installation.

3.1.1.1 GNU Scientific library

The GNU Scientific Library (`http://www.gnu.org/software/gsl/`), or GSL, is a large collection of functions that are very useful in scientific applications, for example integration, random number generation, and Fast Fourier Transform among many others. To install GSL from source, you can run the following commands after you have downloaded `gsl-latest.tar.gz` (`http://ftpmirror.gnu.org/gsl/gsl-latest.tar.gz`):

```
$ tar xf gsl-latest.tar.gz
$ cd gsl-X.X                    # Replace X.X with version number.
$ ./configure
$ make -j8                      # Replace 8 with no. CPU threads.
$ make check
$ sudo make install
```

3.1.1.2 CFITSIO

CFITSIO (`http://heasarc.gsfc.nasa.gov/fitsio/`) is the closest you can get to the pixels in a FITS image while remaining faithful to the FITS standard (`http://fits.gsfc.nasa.gov/fits_standard.html`). It is written by William Pence, the principal author of the FITS standard[2], and is regularly updated. Setting the definitions for all other software packages using FITS images.

Some GNU/Linux distributions have CFITSIO in their package managers, if it is available and updated, you can use it. One problem that might occur is that CFITSIO might not be configured with the `--enable-reentrant` option by the distribution. This option allows CFITSIO to open a file in multiple threads, it can thus provide great speed improvements. If CFITSIO was not configured with this option, any program which needs this capability will warn you and abort when you ask for multiple threads (see Section 4.4 [Multi-threaded operations], page 112).

To install CFITSIO from source, we strongly recommend that you have a look through Chapter 2 (Creating the CFITSIO library) of the CFITSIO manual and understand the options you can pass to `$ ./configure` (they aren't too much). This is a very basic package

[2] Pence, W.D. et al. Definition of the Flexible Image Transport System (FITS), version 3.0. (2010) Astronomy and Astrophysics, Volume 524, id.A42, 40 pp.

for most astronomical software and it is best that you configure it nicely with your system. Once you download the source and unpack it, the following configure script should be enough for most purposes. Don't forget to read chapter two of the manual though, for example the second option is only for 64bit systems. The manual also explains how to check if it has been installed correctly.

CFITSIO comes with two executable files called `fpack` and `funpack`. From their manual: they "are standalone programs for compressing and uncompressing images and tables that are stored in the FITS (Flexible Image Transport System) data format. They are analogous to the gzip and gunzip compression programs except that they are optimized for the types of astronomical images that are often stored in FITS format". The commands below will compile and install them on your system along with CFITSIO. They are not essential for Gnuastro, since they are just wrappers for functions within CFITSIO, but they can come in handy. The `make utils` command is only available for versions above 3.39, it will build these executable files along with several other executable test files which are deleted in the following commands before the installation (otherwise the test files will also be installed).

The commands necessary to decompress, build and install CFITSIO from source are described below. Let's assume you have downloaded `cfitsio_latest.tar.gz` (`http://heasarc.gsfc.nasa.gov/FTP/software/fitsio/c/cfitsio_latest.tar.gz`) and are in the same directory:

```
$ tar xf cfitsio_latest.tar.gz
$ cd cfitsio
$ ./configure --prefix=/usr/local --enable-sse2 --enable-reentrant
$ make
$ make utils
$ ./testprog > testprog.lis
$ diff testprog.lis testprog.out    # Should have no output
$ cmp testprog.fit testprog.std     # Should have no output
$ rm cookbook fitscopy imcopy smem speed testprog
$ sudo make install
```

3.1.1.3 WCSLIB

WCSLIB (`http://www.atnf.csiro.au/people/mcalabre/WCS/`) is written and maintained by one of the authors of the World Coordinate System (WCS) definition in the FITS standard (`http://fits.gsfc.nasa.gov/fits_standard.html`)[3], Mark Calabretta. It might be already built and ready in your distribution's package management system. However, here the installation from source is explained, for the advantages of installation from source please see Section 3.1.1 [Mandatory dependencies], page 62. To install WCSLIB you will need to have CFITSIO already installed, see Section 3.1.1.2 [CFITSIO], page 62.

WCSLIB also has plotting capabilities which use PGPLOT (a plotting library for C). If you wan to use those capabilities in WCSLIB, Section B.2 [PGPLOT], page 472, provides the PGPLOT installation instructions. However PGPLOT is old[4], so its installation is not

[3] Greisen E.W., Calabretta M.R. (2002) Representation of world coordinates in FITS. Astronomy and Astrophysics, 395, 1061-1075.

[4] As of early June 2016, its most recent version was uploaded in February 2001.

easy, there are also many great modern WCS plotting tools (mostly in written in Python). Hence, if you will not be using those plotting functions in WCSLIB, you can configure it with the `--without-pgplot` option as shown below.

If you have the cURL library[5] on your system and you installed CFITSIO version 3.42 or later, you will need to also link with the cURL library at configure time (through the `-lcurl` option as shown below). CFITSIO uses the cURL library for its HTTPS (or HTTP Secure[6]) support and if it is present on your system, CFITSIO will depend on it. Therefore, if `./configure` command below fails (you don't have the cURL library), then remove this option and rerun it.

Let's assume you have downloaded `wcslib.tar.bz2` (`ftp://ftp.atnf.csiro.au/pub/software/wcslib/wcslib.tar.bz2`) and are in the same directory, to configure, build, check and install WCSLIB follow the steps below.

```
$ tar xf wcslib.tar.bz2

## In the 'cd' command, replace 'X.X' with version number.
$ cd wcslib-X.X

## If './configure' fails, remove '-lcurl' and run again.
$ ./configure LIBS="-pthread -lcurl -lm" --without-pgplot     \
              --disable-fortran
$ make
$ make check
$ sudo make install
```

3.1.2 Optional dependencies

The libraries listed here are only used for very specific applications, therefore if you don't want these operations, Gnuastro will be built and installed without them and you don't have to have the dependencies.

If the `./configure` script can't find these requirements, it will warn you in the end that they are not present and notify you of the operation(s) you can't do due to not having them. If the output you request from a program requires a missing library, that program is going to warn you and abort. In the case of program dependencies (like GPL GhostScript), if you install them at a later time, the program will run. This is because if required libraries are not present at build time, the executables cannot be built, but an executable is called by the built program at run time so if it becomes available, it will be used. If you do install an optional library later, you will have to rebuild Gnuastro and reinstall it for it to take effect.

GNU Libtool

> Libtool is a program to simplify managing of the libraries to build an executable (a program). GNU Libtool has some added functionality compared to other implementations. If GNU Libtool isn't present on your system at configuration time, a warning will be printed and Section 11.2 [BuildProgram], page 328, won't be built or installed. The configure script will look into your search path

[5] `https://curl.haxx.se`

[6] `https://en.wikipedia.org/wiki/HTTPS`

(`PATH`) for GNU Libtool through the following executable names: `libtool` (acceptable only if it is the GNU implementation) or `glibtool`. See Section 3.3.1.2 [Installation directory], page 79, for more on `PATH`.

GNU Libtool (the binary/executable file) is a low-level program that is probably already present on your system, and if not, is available in your operating system package manager[7]. If you want to install GNU Libtool's latest version from source, please visit its webpage (`https://www.gnu.org/software/libtool/`).

Gnuastro's tarball is shipped with an internal implementation of GNU Libtool. Even if you have GNU Libtool, Gnuastro's internal implementation is used for the building and installation of Gnuastro. As a result, you can still build, install and use Gnuastro even if you don't have GNU Libtool installed on your system. However, this internal Libtool does not get installed. Therefore, after Gnuastro's installation, if you want to use Section 11.2 [BuildProgram], page 328, to compile and link your own C source code which uses the Section 11.3 [Gnuastro library], page 332, you need to have GNU Libtool available on your system (independent of Gnuastro). See Section 11.1 [Review of library fundamentals], page 320, to learn more about libraries.

libgit2 Git is one of the most common version control systems (see Section 3.2.2 [Version controlled source], page 72). When `libgit2` is present, and Gnuastro's programs are run within a version controlled directory, outputs will contain the version number of the working directory's repository for future reproducibility. See the `COMMIT` keyword header in Section 4.9 [Output FITS files], page 125, for a discussion.

libjpeg libjpeg is only used by ConvertType to read from and write to JPEG images, see Section 5.2.1 [Recognized file formats], page 138. libjpeg (`http://www.ijg.org/`) is a very basic library that provides tools to read and write JPEG images, most Unix-like graphic programs and libraries use it. Therefore you most probably already have it installed. libjpeg-turbo (`http://libjpeg-turbo.virtualgl.org/`) is an alternative to libjpeg. It uses Single instruction, multiple data (SIMD) instructions for ARM based systems that significantly decreases the processing time of JPEG compression and decompression algorithms.

libtiff libtiff is used by ConvertType and the libraries to read TIFF images, see Section 5.2.1 [Recognized file formats], page 138. libtiff (`http://www.simplesystems.org/libtiff/`) is a very basic library that provides tools to read and write TIFF images, most Unix-like operating system graphic programs and libraries use it. Therefore even if you don't have it installed, it must be easily available in your package manager.

GPL Ghostscript

GPL Ghostscript's executable (`gs`) is called by ConvertType to compile a PDF file from a source PostScript file, see Section 5.2 [ConvertType], page 138.

[7] Note that we want the binary/executable Libtool program which can be run on the command-line. In Debian-based operating systems which separate various parts of a package, you want want `libtool-bin`, the `libtool` package won't contain the executable program.

Therefore its headers (and libraries) are not needed. With a very high probability you already have it in your GNU/Linux distribution. Unfortunately it does not follow the standard GNU build style so installing it is very hard. It is best to rely on your distribution's package managers for this.

3.1.3 Bootstrapping dependencies

Bootstrapping is only necessary if you have decided to obtain the full version controlled history of Gnuastro, see Section 3.2.2 [Version controlled source], page 72, and Section 3.2.2.1 [Bootstrapping], page 73. Using the version controlled source enables you to always be up to date with the most recent development work of Gnuastro (bug fixes, new functionalities, improved algorithms and etc). If you have downloaded a tarball (see Section 3.2 [Downloading the source], page 71), then you can ignore this subsection.

To successfully run the bootstrapping process, there are some additional dependencies to those discussed in the previous subsections. These are low level tools that are used by a large collection of Unix-like operating systems programs, therefore they are most probably already available in your system. If they are not already installed, you should be able to easily find them in any GNU/Linux distribution package management system (`apt-get`, `yum`, `pacman` and etc). The short names in parenthesis in `typewriter` font after the package name can be used to search for them in your package manager. For the GNU Portability Library, GNU Autoconf Archive and TEX Live, it is recommended to use the instructions here, not your operating system's package manager.

GNU Portability Library (Gnulib)

To ensure portability for a wider range of operating systems (those that don't include GNU C library, namely glibc), Gnuastro depends on the GNU portability library, or Gnulib. Gnulib keeps a copy of all the functions in glibc, implemented (as much as possible) to be portable to other operating systems. The `bootstrap` script can automatically clone Gnulib (as a `gnulib/` directory inside Gnuastro), however, as described in Section 3.2.2.1 [Bootstrapping], page 73, this is not recommended.

The recommended way to bootstrap Gnuastro is to first clone Gnulib and the Autoconf archives (see below) into a local directory outside of Gnuastro. Let's call it `DEVDIR`[8] (which you can set to any directory). Currently in Gnuastro, both Gnulib and Autoconf archives have to be cloned in the same top directory[9] like the case here[10]:

```
$ DEVDIR=/home/yourname/Development
$ cd $DEVDIR
```

[8] If you are not a developer in Gnulib or Autoconf archives, `DEVDIR` can be a directory that you don't backup. In this way the large number of files in these projects won't slow down your backup process or take bandwidth (if you backup to a remote server).

[9] If you already have the Autoconf archives in a separate directory, or can't clone it in the same directory as Gnulib, or you have it with another directory name (not `autoconf-archive/`), you can follow this short step. Set `AUTOCONFARCHIVES` to your desired address. Then define a symbolic link in `DEVDIR` with the following command so Gnuastro's bootstrap script can find it:
`$ ln -s $AUTOCONFARCHIVES $DEVDIR/autoconf-archive`.

[10] If your internet connection is active, but Git complains about the network, it might be due to your network setup not recognizing the git protocol. In that case use the following URL for the HTTP protocol instead (for Autoconf archives, replace the name): `http://git.sv.gnu.org/r/gnulib.git`

```
$ git clone git://git.sv.gnu.org/gnulib.git
$ git clone git://git.sv.gnu.org/autoconf-archive.git
```

You now have the full version controlled source of these two repositories in separate directories. Both these packages are regularly updated, so every once in a while, you can run `$ git pull` within them to get any possible updates.

GNU Automake (`automake`)

GNU Automake will build the `Makefile.in` files in each sub-directory using the (hand-written) `Makefile.am` files. The `Makefile.in`s are subsequently used to generate the `Makefile`s when the user runs `./configure` before building.

GNU Autoconf (`autoconf`)

GNU Autoconf will build the `configure` script using the configurations we have defined (hand-written) in `configure.ac`.

GNU Autoconf Archive

These are a large collection of tests that can be called to run at `./configure` time. See the explanation under GNU Portability Library above for instructions on obtaining it and keeping it up to date.

GNU Libtool (`libtool`)

GNU Libtool is in charge of building all the libraries in Gnuastro. The libraries contain functions that are used by more than one program and are installed for use in other programs. They are thus put in a separate directory (`lib/`).

GNU help2man (`help2man`)

GNU help2man is used to convert the output of the `--help` option (Section 4.3.2 [`--help`], page 110) to the traditional Man page (Section 4.3.3 [Man pages], page 111).

LaTeX and some TeX packages

Some of the figures in this book are built by LaTeX (using the PGF/TikZ package). The LaTeX source for those figures is version controlled for easy maintenance not the actual figures. So the `./boostrap` script will run LaTeX to build the figures. The best way to install LaTeX and all the necessary packages is through TeX live (`https://www.tug.org/texlive/`
) which is a package manager for TeX related tools that is independent of any operating system. It is thus preferred to the TeX Live versions distributed by your operating system.

To install TeX Live, go to the webpage and download the appropriate installer by following the "download" link. Note that by default the full package repository will be downloaded and installed (around 4 Giga Bytes) which can take *very* long to download and to update later. However, most packages are not needed by everyone, it is easier, faster and better to install only the "Basic scheme" (consisting of only the most basic TeX and LaTeX packages, which is less than 200 Mega bytes)[11].

After the installation, be sure to set the environment variables as suggested in the end of the outputs. Any time you confront (need) a package you don't have,

[11] You can also download the DVD iso file at a later time to keep as a backup for when you don't have internet connection if you need a package.

simply install it with a command like below (similar to how you install software from your operating system's package manager)[12]. To install all the necessary TEX packages for a successful Gnuastro bootstrap, run this command:

```
$ su
# tlmgr install epsf jknapltx caption biblatex biber iftex \
                etoolbox logreq xstring xkeyval pgf ms     \
                xcolor pgfplots times rsfs pstools epspdf
```

ImageMagick (`imagemagick`)
: ImageMagick is a wonderful and robust program for image manipulation on the command-line. `bootsrap` uses it to convert the book images into the formats necessary for the various book formats.

3.1.4 Dependencies from package managers

The most basic way to install a package on your system is to build the packages from source yourself. Alternatively, you can use your operating system's package manager to download pre-compiled files and install them. The latter choice is easier and faster. However, we recommend that you build the Section 3.1.1 [Mandatory dependencies], page 62, yourself from source (all necessary commands and links are given in the respective section). Here are some basic reasons behind this recommendation.

1. Your distribution's pre-built package might not be the most recent release.
2. For each package, Gnuastro might preform better (or require) certain configuration options that your distribution's package managers didn't add for you. If present, these configuration options are explained during the installation of each in the sections below (for example in Section 3.1.1.2 [CFITSIO], page 62). When the proper configuration has not been set, the programs should complain and inform you.
3. For the libraries, they might separate the binary file from the header files which can cause confusion, see Section 3.3.5 [Known issues], page 89.
4. Like any other tool, the science you derive from Gnuastro's tools highly depend on these lower level dependencies, so generally it is much better to have a close connection with them. By reading their manuals, installing them and staying up to date with changes/bugs in them, your scientific results and understanding (of what is going on, and thus how you interpret your scientific results) will also correspondingly improve.

Based on your package manager, you can use any of the following commands to install the mandatory and optional dependencies. If your package manager isn't included in the list below, please send us the respective command, so we add it. Gnuastro itself if also already packaged in some package managers (for example Debian or Homebrew).

As discussed above, we recommend installing the *mandatory* dependencies manually from source (see Section 3.1.1 [Mandatory dependencies], page 62). Therefore, in each command below, first the optional dependencies are given. The mandatory dependencies are included after an empty line. If you would also like to install the mandatory dependencies with your package manager, just ignore the empty line.

[12] After running TEX, or LATEX, you might get a warning complaining about a `missingfile`. Run '`tlmgr info missingfile`' to see the package(s) containing that file which you can install.

For better archivability and compression ratios, Gnuastro's recommended tarball compression format is with the Lzip (`http://lzip.nongnu.org/lzip.html`) program, see Section 3.2.1 [Release tarball], page 71. Therefore, the package manager commands below also contain Lzip.

`apt-get` (Debian-based OSs: Debian, Ubuntu, Linux Mint, and etc)

Debian (`https://en.wikipedia.org/wiki/Debian`) is one of the oldest GNU/Linux distributions[13]. It thus has a very extended user community and a robust internal structure and standards. All of it is free software and based on the work of volunteers around the world. Many distributions are thus derived from it, for example Ubuntu and Linux Mint. This arguably makes Debian-based OSs the largest, and most used, class of GNU/Linux distributions. All of them use Debian's Advanced Packaging Tool (APT, for example `apt-get`) for managing packages.

```
$ sudo apt-get install ghostscript libtool-bin libjpeg-dev  \
                       libtiff-dev libgit2-dev lzip         \
                                                            \
                       libgsl0-dev libcfitsio-dev wcslib-dev
```

Gnuastro is packaged (`https://tracker.debian.org/pkg/gnuastro`) in Debian (and thus some of its derivate operating systems). Just make sure it is the most recent version.

`dnf`
`yum` (Red Hat-based OSs: Red Hat, Fedora, CentOS, Scientific Linux, and etc)

Red Hat Enterprise Linux (`https://en.wikipedia.org/wiki/Red_Hat`) (RHEL) is released by Red Hat Inc. RHEL requires paid subscriptions for use of its binaries and support. But since it is free software, many other teams use its code to spin-off their own distributions based on RHEL. Red Hat-based GNU/Linux distributions initially used the "Yellowdog Updated, Modifier" (YUM) package manager, which has been replaced by "Dandified yum" (DNF). If the latter isn't available on your system, you can use `yum` instead of `dnf` in the command below.

```
$ sudo dnf install ghostscript libtool libjpeg-devel        \
                   libtiff-devel libgit2-devel lzip         \
                                                            \
                   gsl-devel cfitsio-devel wcslib-devel
```

`brew` (macOS)

macOS (`https://en.wikipedia.org/wiki/MacOS`) is the operating system used on Apple devices. macOS does not come with a package manager pre-installed, but several widely used, third-party package managers exist, such as Homebrew or MacPorts. Both are free software. Currently we have only tested Gnuastro's installation with Homebrew as described below.

If not already installed, first obtain Homebrew by following the instructions at `https://`

[13] `https://en.wikipedia.org/wiki/List_of_Linux_distributions#Debian-based`

`brew.sh`. Homebrew manages packages in different 'taps'. To install WCSLIB (discussed in Section 3.1.1 [Mandatory dependencies], page 62) via Homebrew you will need to `tap` into `brewsci/science` first (the tap may change in the future, but can be found by calling `brew search wcslib`).

```
$ brew install ghostscript libtool libjpeg libtiff          \
               libgit2 lzip                                 \
                                                            \
               gsl cfitsio
$ brew tap brewsci/science
$ brew install wcslib
```

`pacman` (Arch Linux)

Arch Linux (`https://en.wikipedia.org/wiki/Arch_Linux`) is a smaller GNU/Linux distribution, which follows the KISS principle ("keep it simple, stupid") as a general guideline. It "focuses on elegance, code correctness, minimalism and simplicity, and expects the user to be willing to make some effort to understand the system's operation". Arch Linux uses "Package manager" (Pacman) to manage its packages/components.

```
$ sudo pacman -S ghostscript libtool libjpeg libtiff        \
                 libgit2 lzip                               \
                                                            \
                 gsl cfitsio wcslib
```

`zypper` (openSUSE and SUSE Linux Enterprise Server)

openSUSE (`https://www.opensuse.org`) is a community project supported by SUSE (`https://www.suse.com`) with both stable and rolling releases. SUSE Linux Enterprise Server[14] (SLES) is the commercial offering which shares code and tools. Many additional packages are offered in the Build Service[15]. openSUSE and SLES use `zypper` (cli) and YaST (GUI) for managing repositories and packages.

```
$ sudo zypper install ghostscript_any libtool pkgconfig     \
                cfitsio-devel gsl-devel libcurl-devel       \
                libgit2-devel libjpeg62-devel libtiff-devel \
                wcslib-devel
```

When building Gnuastro, run the configure script with the following `CPPFLAGS` environment variable:

```
$ ./configure CPPFLAGS="-I/usr/include/cfitsio"
```

Usually, when libraries are installed by operating system package managers, there should be no problems when configuring and building other programs from source (that depend on the libraries: Gnuastro in this case). However, in some special conditions, problems may pop-up during the configuration, building, or checking/running any of Gnuastro's programs. The most common of such problems and their solution are discussed below.

[14] `https://www.suse.com/products/server`

[15] `https://build.opensuse.org`

Not finding library during configuration: If a library is installed, but during Gnuastro's `configure` step the library isn't found, then configure Gnuastro like the command below (correcting `/path/to/lib`). For more, see Section 3.3.5 [Known issues], page 89, and Section 3.3.1.2 [Installation directory], page 79.

```
$ ./configure LDFLAGS="-I/path/to/lib"
```

Not finding header (.h) files while building: If a library is installed, but during Gnuastro's `make` step, the library's header (file with a `.h` suffix) isn't found, then configure Gnuastro like the command below (correcting `/path/to/include`). For more, see Section 3.3.5 [Known issues], page 89, and Section 3.3.1.2 [Installation directory], page 79.

```
$ ./configure CPPFLAGS="-I/path/to/include"
```

Gnuastro's programs don't run during check or after install: If a library is installed, but the programs don't run due to linking problems, set the `LD_LIBRARY_PATH` variable like below (assuming Gnuastro is installed in `/path/to/installed`). For more, see Section 3.3.5 [Known issues], page 89, and Section 3.3.1.2 [Installation directory], page 79.

```
$ export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/path/to/installed/lib"
```

3.2 Downloading the source

Gnuastro's source code can be downloaded in two ways. As a tarball, ready to be configured and installed on your system (as described in Section 1.1 [Quick start], page 1), see Section 3.2.1 [Release tarball], page 71. If you want official releases of stable versions this is the best, easiest and most common option. Alternatively, you can clone the version controlled history of Gnuastro, run one extra bootstrapping step and then follow the same steps as the tarball. This will give you access to all the most recent work that will be included in the next release along with the full project history. The process is thoroughly introduced in Section 3.2.2 [Version controlled source], page 72.

3.2.1 Release tarball

A release tarball (commonly compressed) is the most common way of obtaining free and open source software. A tarball is a snapshot of one particular moment in the Gnuastro development history along with all the necessary files to configure, build, and install Gnuastro easily (see Section 1.1 [Quick start], page 1). It is very straightforward and needs the least set of dependencies (see Section 3.1.1 [Mandatory dependencies], page 62). Gnuastro has tarballs for official stable releases and pre-releases for testing. See Section 1.5 [Version numbering], page 7, for more on the two types of releases and the formats of the version numbers. The URLs for each type of release are given below.

Official stable releases (`http://ftp.gnu.org/gnu/gnuastro`):

> This URL hosts the official stable releases of Gnuastro. Always use the most recent version (see Section 1.5 [Version numbering], page 7). By clicking on the "Last modified" title of the second column, the files will be sorted by their date which you can also use to find the latest version. It is recommended to use a

mirror to download these tarballs, please visit `http://ftpmirror.gnu.org/gnuastro/` and see below.

Pre-release tar-balls (`http://alpha.gnu.org/gnu/gnuastro`):
This URL contains unofficial pre-release versions of Gnuastro. The pre-release versions of Gnuastro here are for enthusiasts to try out before an official release. If there are problems, or bugs then the testers will inform the developers to fix before the next official release. See Section 1.5 [Version numbering], page 7, to understand how the version numbers here are formatted. If you want to remain even more up-to-date with the developing activities, please clone the version controlled source as described in Section 3.2.2 [Version controlled source], page 72.

Gnuastro's official/stable tarball is released with two formats: Gzip (with suffix `.tar.gz`) and Lzip (with suffix `.tar.lz`). The pre-release tarballs (after version 0.3) are released only as an Lzip tarball. Gzip is a very well-known and widely used compression program created by GNU and available in most systems. However, Lzip provides a better compression ratio and more robust archival capacity. For example Gnuastro 0.3's tarball was 2.9MB and 4.3MB with Lzip and Gzip respectively, see the Lzip webpage (`http://www.nongnu.org/lzip/lzip.html`) for more. Lzip might not be pre-installed in your operating system, if so, installing it from your operating system's package manager or from source is very easy and fast (it is a very small program).

The GNU FTP server is mirrored (has backups) in various locations on the globe (`http://www.gnu.org/order/ftp.html`). You can use the closest mirror to your location for a more faster download. Note that only some mirrors keep track of the pre-release (alpha) tarballs. Also note that if you want to download immediately after and announcement (see Section 1.9 [Announcements], page 13), the mirrors might need some time to synchronize with the main GNU FTP server.

3.2.2 Version controlled source

The publicly distributed Gnuastro tar-ball (for example `gnuastro-X.X.tar.gz`) does not contain the revision history, it is only a snapshot of the source code at one significant instant of Gnuastro's history (specified by the version number, see Section 1.5 [Version numbering], page 7), ready to be configured and built. To be able to develop successfully, the revision history of the code can be very useful to track when something was added or changed, also some updates that are not yet officially released might be in it.

We use Git for the version control of Gnuastro. For those who are not familiar with it, we recommend the ProGit book (`https://git-scm.com/book/en`). The whole book is publicly available for online reading and downloading and does a wonderful job at explaining the concepts and best practices.

Let's assume you want to keep Gnuastro in the `TOPGNUASTRO` directory (can be any directory, change the value below). The full version controlled history of Gnuastro can be cloned in `TOPGNUASTRO/gnuastro` by running the following commands[16]:

```
$ TOPGNUASTRO=/home/yourname/Research/projects/
```

[16] If your internet connection is active, but Git complains about the network, it might be due to your network setup not recognizing the Git protocol. In that case use the following URL which uses the HTTP protocol instead: `http://git.sv.gnu.org/r/gnuastro.git`

```
$ cd $TOPGNUASTRO
$ git clone git://git.sv.gnu.org/gnuastro.git
```

The `$TOPGNUASTRO/gnuastro` directory will contain hand-written (version controlled) source code for Gnuastro's programs, libraries, this book and the tests. All are divided into sub-directories with standard and very descriptive names. The version controlled files in the top cloned directory are either mainly in capital letters (for example `THANKS` and `README`) or mainly written in small-caps (for example `configure.ac` and `Makefile.am`). The former are non-programming, standard writing for human readers containing high-level information about the whole package. The latter are instructions to customize the GNU build system for Gnuastro. For more on Gnuastro's source code structure, please see Chapter 12 [Developing], page 445. We won't go any deeper here.

The cloned Gnuastro source cannot immediately be configured, compiled, or installed since it only contains hand-written files, not automatically generated or imported files which do all the hard work of the build process. See Section 3.2.2.1 [Bootstrapping], page 73, for the process of generating and importing those files (its not too hard!). Once you have bootstrapped Gnuastro, you can run the standard procedures (in Section 1.1 [Quick start], page 1). Very soon after you have cloned it, Gnuastro's main `master` branch will be updated on the main repository (since the developers are actively working on Gnuastro), for the best practices in keeping your local history in sync with the main repository see Section 3.2.2.2 [Synchronizing], page 75.

3.2.2.1 Bootstrapping

The version controlled source code lacks the source files that we have not written or are automatically built. These automatically generated files are included in the distributed tar ball for each distribution (for example `gnuastro-X.X.tar.gz`, see Section 1.5 [Version numbering], page 7) and make it easy to immediately configure, build, and install Gnuastro. However from the perspective of version control, they are just bloatware and sources of confusion (since they are not changed by Gnuastro developers).

The process of automatically building and importing necessary files into the cloned directory is known as *bootstrapping*. All the instructions for an automatic bootstrapping are available in `bootstrap` and configured using `bootstrap.conf`. `bootstrap` and `COPYING` (which contains the software copyright notice) are the only files not written by Gnuastro developers but under version control to enable simple bootstrapping and legal information on usage immediately after cloning. `bootstrap.conf` is maintained by the GNU Portability Library (Gnulib) and this file is an identical copy, so do not make any changes in this file since it will be replaced when Gnulib releases an update. Make all your changes in `bootstrap.conf`.

The bootstrapping process has its own separate set of dependencies, the full list is given in Section 3.1.3 [Bootstrapping dependencies], page 66. They are generally very low-level and used by a very large set of commonly used programs, so they are probably already installed on your system. The simplest way to bootstrap Gnuastro is to simply run the bootstrap script within your cloned Gnuastro directory as shown below. However, please read the next paragraph before doing so (see Section 3.2.2 [Version controlled source], page 72, for `TOPGNUASTRO`).

```
$ cd TOPGNUASTRO/gnuastro
$ ./bootstrap                       # Requires internet connection
```

Without any options, `bootstrap` will clone Gnulib within your cloned Gnuastro directory (`TOPGNUASTRO/gnuastro/gnulib`) and download the necessary Autoconf archives macros. So if you run bootstrap like this, you will need an internet connection every time you decide to bootstrap. Also, Gnulib is a large package and cloning it can be slow. It will also keep the full Gnulib repository within your Gnuastro repository, so if another one of your projects also needs Gnulib, and you insist on running bootstrap like this, you will have two copies. In case you regularly backup your important files, Gnulib will also slow down the backup process. Therefore while the simple invocation above can be used with no problem, it is not recommended. To do better, see the next paragraph.

The recommended way to get these two packages is thoroughly discussed in Section 3.1.3 [Bootstrapping dependencies], page 66, (in short: clone them in the separate `DEVDIR/` directory). The following commands will take you into the cloned Gnuastro directory and run the `bootstrap` script, while telling it to copy some files (instead of making symbolic links, with the `--copy` option, this is not mandatory[17]) and where to look for Gnulib (with the `--gnulib-srcdir` option). Please note that the address given to `--gnulib-srcdir` has to be an absolute address (so don't use `~` or `../` for example).

```
$ cd $TOPGNUASTRO/gnuastro
$ ./bootstrap --copy --gnulib-srcdir=$DEVDIR/gnulib
```

Since Gnulib and Autoconf archives are now available in your local directories, you don't need an internet connection every time you decide to remove all untracked files and redo the bootstrap (see box below). You can also use the same command on any other project that uses Gnulib. All the necessary GNU C library functions, Autoconf macros and Automake inputs are now available along with the book figures. The standard GNU build system (Section 1.1 [Quick start], page 1) will do the rest of the job.

Undoing the bootstrap: During the development, it might happen that you want to remove all the automatically generated and imported files. In other words, you might want to reverse the bootstrap process. Fortunately Git has a good program for this job: `git clean`. Run the following command and every file that is not version controlled will be removed.

```
git clean -fxd
```

It is best to commit any recent change before running this command. You might have created new files since the last commit and if they haven't been committed, they will all be gone forever (using `rm`). To get a list of the non-version controlled files instead of deleting them, add the `n` option to `git clean`, so it becomes `-fxdn`.

Besides the `bootstrap` and `bootstrap.conf`, the `bootstrapped/` directory and `README-hacking` file are also related to the bootstrapping process. The former hosts all the imported (bootstrapped) directories. Thus, in the version controlled source, it only contains a `REAME` file, but in the distributed tar-ball it also contains sub-directories filled with all bootstrapped files. `README-hacking` contains a summary of the bootstrapping process discussed in this section. It is a necessary reference when you haven't built this book yet. It is thus not distributed in the Gnuastro tarball.

[17] The `--copy` option is recommended because some backup systems might do strange things with symbolic links.

3.2.2.2 Synchronizing

The bootstrapping script (see Section 3.2.2.1 [Bootstrapping], page 73) is not regularly needed: you mainly need it after you have cloned Gnuastro (once) and whenever you want to re-import the files from Gnulib, or Autoconf archives[18] (not too common). However, Gnuastro developers are constantly working on Gnuastro and are pushing their changes to the official repository. Therefore, your local Gnuastro clone will soon be out-dated. Gnuastro has two mailing lists dedicated to its developing activities (see Section 12.10 [Developing mailing lists], page 460). Subscribing to them can help you decide when to synchronize with the official repository.

To pull all the most recent work in Gnuastro, run the following command from the top Gnuastro directory. If you don't already have a built system, ignore `make distclean`. The separate steps are described in detail afterwards.

```
$ make distclean && git pull && autoreconf -f
```

You can also run the commands separately:

```
$ make distclean
$ git pull
$ autoreconf -f
```

If Gnuastro was already built in this directory, you don't want some outputs from the previous version being mixed with outputs from the newly pulled work. Therefore, the first step is to clean/delete all the built files with `make distclean`. Fortunately the GNU build system allows the separation of source and built files (in separate directories). This is a great feature to keep your source directory clean and you can use it to avoid the cleaning step. Gnuastro comes with a script with some useful options for this job. It is useful if you regularly pull recent changes, see Section 3.3.2 [Separate build and source directories], page 85.

After the pull, we must re-configure Gnuastro with `autoreconf -f` (part of GNU Autoconf). It will update the `./configure` script and all the `Makefile.in`[19] files based on the hand-written configurations (in `configure.ac` and the `Makefile.am` files). After running `autoreconf -f`, a warning about `TEXI2DVI` might show up, you can ignore that.

The most important reason for re-building Gnuastro's build system is to generate/update the version number for your updated Gnuastro snapshot. This generated version number will include the commit information (see Section 1.5 [Version numbering], page 7). The version number is included in nearly all outputs of Gnuastro's programs, therefore it is vital for reproducing an old result.

As a summary, be sure to run '`autoreconf -f`' after every change in the Git history. This includes synchronization with the main server or even a commit you have made yourself.

If you would like to see what has changed since you last synchronized your local clone, you can take the following steps instead of the simple command above (don't type anything after `#`):

```
$ git checkout master             # Confirm if you are on master.
```

[18] `https://savannah.gnu.org/task/index.php?13993` is defined for you to check if significant (for Gnuastro) updates are made in these repositories, since the last time you pulled from them.

[19] In the GNU build system, `./configure` will use the `Makefile.in` files to create the necessary `Makefile` files that are later read by `make` to build the package.

```
$ git fetch origin                # Fetch all new commits from server.
$ git log master..origin/master   # See all the new commit messages.
$ git merge origin/master         # Update your master branch.
$ autoreconf -f                   # Update the build system.
```

By default `git log` prints the most recent commit first, add the `--reverse` option to see the changes chronologically. To see exactly what has been changed in the source code along with the commit message, add a `-p` option to the `git log`.

If you want to make changes in the code, have a look at Chapter 12 [Developing], page 445, to get started easily. Be sure to commit your changes in a separate branch (keep your `master` branch to follow the official repository) and re-run `autoreconf -f` after the commit. If you intend to send your work to us, you can safely use your commit since it will be ultimately recorded in Gnuastro's official history. If not, please upload your separate branch to a public hosting service, for example GitLab (`https://gitlab.com`), and link to it in your report/paper. Alternatively, run `make distcheck` and upload the output `gnuastro-X.X.X.XXXX.tar.gz` to a publicly accessible webpage so your results can be considered scientific (reproducible) later.

3.3 Build and install

This section is basically a longer explanation to the sequence of commands given in Section 1.1 [Quick start], page 1. If you didn't have any problems during the Section 1.1 [Quick start], page 1, steps, you want to have all the programs of Gnuastro installed in your system, you don't want to change the executable names during or after installation, you have root access to install the programs in the default system wide directory, the Letter paper size of the print book is fine for you or as a summary you don't feel like going into the details when everything is working, you can safely skip this section.

If you have any of the above problems or you want to understand the details for a better control over your build and install, read along. The dependencies which you will need prior to configuring, building and installing Gnuastro are explained in Section 3.1 [Dependencies], page 61. The first three steps in Section 1.1 [Quick start], page 1, need no extra explanation, so we will skip them and start with an explanation of Gnuastro specific configuration options and a discussion on the installation directory in Section 3.3.1 [Configuring], page 76, followed by some smaller subsections: Section 3.3.3 [Tests], page 88, Section 3.3.4 [A4 print book], page 88, and Section 3.3.5 [Known issues], page 89, which explains the solutions to known problems you might encounter in the installation steps and ways you can solve them.

3.3.1 Configuring

The `$ ./configure` step is the most important step in the build and install process. All the required packages, libraries, headers and environment variables are checked in this step. The behaviors of make and make install can also be set through command line options to this command.

The configure script accepts various arguments and options which enable the final user to highly customize whatever she is building. The options to configure are generally very similar to normal program options explained in Section 4.1.1 [Arguments and options], page 92. Similar to all GNU programs, you can get a full list of the options along with a short explanation by running

```
$ ./configure --help
```

A complete explanation is also included in the `INSTALL` file. Note that this file was written by the authors of GNU Autoconf (which builds the `configure` script), therefore it is common for all programs which use the `$ ./configure` script for building and installing, not just Gnuastro. Here we only discuss cases where you don't have super-user access to the system and if you want to change the executable names. But before that, a review of the options to configure that are particular to Gnuastro are discussed.

3.3.1.1 Gnuastro configure options

Most of the options to configure (which are to do with building) are similar for every program which uses this script. Here the options that are particular to Gnuastro are discussed. The next topics explain the usage of other configure options which can be applied to any program using the GNU build system (through the configure script).

`--enable-debug`

Compile/build Gnuastro with debugging information, no optimization and without shared libraries.

In order to allow more efficient programs when using Gnuastro (after the installation), by default Gnuastro is built with a 3rd level (a very high level) optimization and no debugging information. By default, libraries are also built for static *and* shared linking (see Section 11.1.2 [Linking], page 324). However, when there are crashes or unexpected behavior, these three features can hinder the process of localizing the problem. This configuration option is identical to manually calling the configuration script with `CFLAGS="-g -O0" --disable-shared`.

In the (rare) situations where you need to do your debugging on the shared libraries, don't use this option. Instead run the configure script by explicitly setting `CFLAGS` like this:

```
$ ./configure CFLAGS="-g -O0"
```

`--enable-check-with-valgrind`

Do the `make check` tests through Valgrind. Therefore, if any crashes or memory-related issues (segmentation faults in particular) occur in the tests, the output of Valgrind will also be put in the `tests/test-suite.log` file without having to manually modify the check scripts. This option will also activate Gnuastro's debug mode (see the `--enable-debug` configure-time option described above).

Valgrind is free software. It is a program for easy checking of memory-related issues in programs. It runs a program within its own controlled environment and can thus identify the exact line-number in the program's source where a memory-related issue occurs. However, it can significantly slow-down the tests. So this option is only useful when a segmentation fault is found during `make check`.

`--enable-progname`

Only build and install `progname` along with any other program that is enabled in this fashion. `progname` is the name of the executable without the `ast`, for example `crop` for Crop (with the executable name of `astcrop`).

Note that by default all the programs will be installed. This option (and the `--disable-progname` options) are only relevant when you don't want to install all the programs. Therefore, if this option is called for any of the programs in Gnuastro, any program which is not explicitly enabled will not be built or installed.

`--disable-progname`
`--enable-progname=no`
Do not build or install the program named `progname`. This is very similar to the `--enable-progname`, but will build and install all the other programs except this one.

`--enable-gnulibcheck`
Enable checks on the GNU Portability Library (Gnulib). Gnulib is used by Gnuastro to enable users of non-GNU based operating systems (that don't use GNU C library or glibc) to compile and use the advanced features that this library provides. We make extensive use of such functions. If you give this option to `$ ./configure`, when you run `$ make check`, first the functions in Gnulib will be tested, then the Gnuastro executables. If your operating system does not support glibc or has an older version of it and you have problems in the build process (`$ make`), you can give this flag to configure to see if the problem is caused by Gnulib not supporting your operating system or Gnuastro, see Section 3.3.5 [Known issues], page 89.

`--disable-guide-message`
`--enable-guide-message=no`
Do not print a guiding message during the GNU Build process of Section 1.1 [Quick start], page 1. By default, after each step, a message is printed guiding the user what the next command should be. Therefore, after `./configure`, it will suggest running `make`. After `make`, it will suggest running `make check` and so on. If Gnuastro is configured with this option, for example

```
$ ./configure --disable-guide-message
```

Then these messages will not be printed after any step (like most programs). For people who are not yet fully accustomed to this build system, these guidelines can be very useful and encouraging. However, if you find those messages annoying, use this option.

Note: If some programs are enabled and some are disabled, it is equivalent to simply enabling those that were enabled. Listing the disabled programs is redundant.

The tests of some programs might depend on the outputs of the tests of other programs. For example MakeProfiles is one the first programs to be tested when you run `$ make check`. MakeProfiles' test outputs (FITS images) are inputs to many other programs (which in turn provide inputs for other programs). Therefore, if you don't install MakeProfiles for example, the tests for many the other programs will be skipped. To avoid this, in one run, you can install all the programs and run the tests but not install. If everything is working correctly, you can run configure again with only the programs you want. However, don't run the tests and directly install after building.

3.3.1.2 Installation directory

One of the most commonly used options to `./configure` is `--prefix`, it is used to define the directory that will host all the installed files (or the "prefix" in their final absolute file name). For example, when you are using a server and you don't have administrator or root access. In this example scenario, if you don't use the `--prefix` option, you won't be able to install the built files and thus access them from anywhere without having to worry about where they are installed. However, once you prepare your startup file to look into the proper place (as discussed thoroughly below), you will be able to easily use this option and benefit from any software you want to install without having to ask the system administrators or install and use a different version of a software that is already installed on the server.

The most basic way to run an executable is to explicitly write its full file name (including all the directory information) and run it. One example is running the configuration script with the `$ ./configure` command (see Section 1.1 [Quick start], page 1). By giving a specific directory (the current directory or `./`), we are explicitly telling the shell to look in the current directory for an executable file named '`configure`'. Directly specifying the directory is thus useful for executables in the current (or nearby) directories. However, when the program (an executable file) is to be used a lot, specifying all those directories will become a significant burden. For example, the `ls` executable lists the contents in a given directory and it is (usually) installed in the `/usr/bin/` directory by the operating system maintainers. Therefore, if using the full address was the only way to access an executable, each time you wanted a listing of a directory, you would have to run the following command (which is very inconvenient, both in writing and in remembering the various directories).

```
$ /usr/bin/ls
```

To address this problem, we have the `PATH` environment variable. To understand it better, we will start with a short introduction to the shell variables. Shell variable values are basically treated as strings of characters. For example, it doesn't matter if the value is a name (string of *alphabetic* characters), or a number (string of *numeric* characters), or both. You can define a variable and a value for it by running

```
$ myvariable1=a_test_value
$ myvariable2="a test value"
```

As you see above, if the value contains white space characters, you have to put the whole value (including white space characters) in double quotes (`"`). You can see the value it represents by running

```
$ echo $myvariable1
$ echo $myvariable2
```

If a variable has no value or it wasn't defined, the last command will only print an empty line. A variable defined like this will be known as long as this shell or terminal is running. Other terminals will have no idea it existed. The main advantage of shell variables is that if they are exported[20], subsequent programs that are run within that shell can access their value. So by changing their value, you can change the "environment" of a program which uses them. The shell variables which are accessed by programs are therefore known as

[20] By running `$ export myvariable=a_test_value` instead of the simpler case in the text

"environment variables"[21]. You can see the full list of exported variables that your shell recognizes by running:

```
$ printenv
```

`HOME` is one commonly used environment variable, it is any user's (the one that is logged in) top directory. Try finding it in the command above. It is used so often that the shell has a special expansion (alternative) for it: '`~`'. Whenever you see file names starting with the tilde sign, it actually represents the value to the `HOME` environment variable, so `~/doc` is the same as `$HOME/doc`.

Another one of the most commonly used environment variables is `PATH`, it is a list of directories to search for executable names. Its value is a list of directories (separated by a colon, or '`:`'). When the address of the executable is not explicitly given (like `./configure` above), the system will look for the executable in the directories specified by `PATH`. If you have a computer nearby, try running the following command to see which directories your system will look into when it is searching for executable (binary) files, one example is printed here (notice how `/usr/bin`, in the `ls` example above, is one of the directories in `PATH`):

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/bin
```

By default `PATH` usually contains system-wide directories, which are readable (but not writable) by all users, like the above example. Therefore if you don't have root (or administrator) access, you need to add another directory to `PATH` which you actually have write access to. The standard directory where you can keep installed files (not just executables) for your own user is the `~/.local/` directory. The names of hidden files start with a '`.`' (dot), so it will not show up in your common command-line listings, or on the graphical user interface. You can use any other directory, but this is the most recognized.

The top installation directory will be used to keep all the package's components: programs (executables), libraries, include (header) files, shared data (like manuals), or configuration files (see Section 11.1 [Review of library fundamentals], page 320, for a thorough introduction to headers and linking). So it commonly has some of the following sub-directories for each class of installed components respectively: `bin/`, `lib/`, `include/` `man/`, `share/`, `etc/`. Since the `PATH` variable is only used for executables, you can add the `~/.local/bin` directory (which keeps the executables/programs or more generally, "binary" files) to `PATH` with the following command. As defined below, first the existing value of `PATH` is used, then your given directory is added to its end and the combined value is put back in `PATH` (run '`$ echo $PATH`' afterwards to check if it was added).

```
$ PATH=$PATH:~/.local/bin
```

Any executable that you installed in `~/.local/bin` will now be usable without having to remember and write its full address. However, as soon as you leave/close your current terminal session, this modified `PATH` variable will be forgotten. Adding the directories which contain executables to the `PATH` environment variable each time you start a terminal is also very inconvenient and prone to errors. Fortunately, there are standard 'startup files' defined by your shell precisely for this (and other) purposes. There is a special startup file for every significant starting step:

[21] You can use shell variables for other actions too, for example to temporarily keep some names or run loops on some files.

`/etc/profile` and everything in `/etc/profile.d/`
: These startup scripts are called when your whole system starts (for example after you turn on your computer). Therefore you need administrator or root privileges to access or modify them.

`~/.bash_profile`
: If you are using (GNU) Bash as your shell, the commands in this file are run, when you log in to your account *through Bash*. Most commonly when you login through the virtual console (where there is no graphic user interface).

`~/.bashrc`
: If you are using (GNU) Bash as your shell, the commands here will be run each time you start a terminal and are already logged in. For example, when you open your terminal emulator in the graphic user interface.

For security reasons, it is highly recommended to directly type in your `HOME` directory value by hand in startup files instead of using variables. So in the following, let's assume your user name is '`name`' (so `~` may be replaced with `/home/name`). To add `~/.local/bin` to your `PATH` automatically on any startup file, you have to "export" the new value of `PATH` in the startup file that is most relevant to you by adding this line:

```
export PATH=$PATH:/home/name/.local/bin
```

Now that you know your system will look into `~/.local/bin` for executables, you can tell Gnuastro's configure script to install everything in the top `~/.local` directory using the `--prefix` option. When you subsequently run `$ make install`, all the install-able files will be put in their respective directory under `~/.local/` (the executables in `~/.local/bin`, the compiled library files in `~/.local/lib`, the library header files in `~/.local/include` and so on, to learn more about these different files, please see Section 11.1 [Review of library fundamentals], page 320). Note that tilde ('`~`') expansion will not happen if you put a '`=`' between `--prefix` and `~/.local`[22], so we have avoided the `=` character here which is optional in GNU-style options, see Section 4.1.1.2 [Options], page 93.

```
$ ./configure --prefix ~/.local
```

You can install everything (including libraries like GSL, CFITSIO, or WCSLIB which are Gnuastro's mandatory dependencies, see Section 3.1.1 [Mandatory dependencies], page 62) locally by configuring them as above. However, recall that `PATH` is only for executable files, not libraries and that libraries can also depend on other libraries. For example WCSLIB depends on CFITSIO and Gnuastro needs both. Therefore, when you installed a library in a non-recognized directory, you have to guide the program that depends on them to look into the necessary library and header file directories. To do that, you have to define the `LDFLAGS` and `CPPFLAGS` environment variables respectively. This can be done while calling `./configure` as shown below:

```
$ ./configure LDFLAGS=-L/home/name/.local/lib           \
              CPPFLAGS=-I/home/name/.local/include      \
              --prefix ~/.local
```

It can be annoying/buggy to do this when configuring every software that depends on such libraries. Hence, you can define these two variables in the most relevant startup

[22] If you insist on using '`=`', you can use `--prefix=$HOME/.local`.

file (discussed above). The convention on using these variables doesn't include a colon to separate values (as `PATH`-like variables do), they use white space characters and each value is prefixed with a compiler option[23]: note the `-L` and `-I` above (see Section 4.1.1.2 [Options], page 93), for `-I` see Section 11.1.1 [Headers], page 321, and for `-L`, see Section 11.1.2 [Linking], page 324. Therefore we have to keep the value in double quotation signs to keep the white space characters and adding the following two lines to the startup file of choice:

```
export LDFLAGS="$LDFLAGS -L/home/name/.local/lib"
export CPPFLAGS="$CPPFLAGS -I/home/name/.local/include"
```

Dynamic libraries are linked to the executable every time you run a program that depends on them (see Section 11.1.2 [Linking], page 324, to fully understand this important concept). Hence dynamic libraries also require a special path variable called `LD_LIBRARY_PATH` (same formatting as `PATH`). To use programs that depend on these libraries, you need to add `~/.local/lib` to your `LD_LIBRARY_PATH` environment variable by adding the following line to the relevant start-up file:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/home/name/.local/lib
```

If you also want to access the Info (see Section 4.3.4 [Info], page 111) and man pages (see Section 4.3.3 [Man pages], page 111) documentations add `~/.local/share/info` and `~/.local/share/man` to your `INFOPATH`[24] and `MANPATH` environment variables respectively.

A final note is that order matters in the directories that are searched for all the variables discussed above. In the examples above, the new directory was added after the system specified directories. So if the program, library or manuals are found in the system wide directories, the user directory is no longer searched. If you want to search your local installation first, put the new directory before the already existing list, like the example below.

```
export LD_LIBRARY_PATH=/home/name/.local/lib:$LD_LIBRARY_PATH
```

This is good when a library, for example CFITSIO, is already present on the system, but the system-wide install wasn't configured with the correct configuration flags (see Section 3.1.1.2 [CFITSIO], page 62), or you want to use a newer version and you don't have administrator or root access to update it on the whole system/server. If you update `LD_LIBRARY_PATH` by placing `~/.local/lib` first (like above), the linker will first find the CFITSIO you installed for yourself and link with it. It thus will never reach the system-wide installation.

There are important security problems with using local installations first: all important system-wide executables and libraries (important executables like `ls` and `cp`, or libraries like the C library) can be replaced by non-secure versions with the same file names and put in the customized directory (`~/.local` in this example). So if you choose to search in your customized directory first, please *be sure* to keep it clean from executables or libraries with the same names as important system programs or libraries.

---

[23] These variables are ultimately used as options while building the programs, so every value has be an option name followed be a value as discussed in Section 4.1.1.2 [Options], page 93.

[24] Info has the following convention: "If the value of `INFOPATH` ends with a colon [or it isn't defined] ..., the initial list of directories is constructed by appending the build-time default to the value of `INFOPATH`." So when installing in a non-standard directory and if `INFOPATH` was not initially defined, add a colon to the end of `INFOPATH` as shown below, otherwise Info will not be able to find system-wide installed documentation:
`echo 'export INFOPATH=$INFOPATH:/home/name/.local/share/info:' >> ~/.bashrc`
Note that this is only an internal convention of Info, do not use it for other `*PATH` variables.

Summary: When you are using a server which doesn't give you administrator/root access AND you would like to give priority to your own built programs and libraries, not the version that is (possibly already) present on the server, add these lines to your startup file. See above for which startup file is best for your case and for a detailed explanation on each. Don't forget to replace '`/YOUR-HOME-DIR`' with your home directory (for example '`/home/your-id`'):

```
export PATH="/YOUR-HOME-DIR/.local/bin:$PATH"
export LDFLAGS="-L/YOUR-HOME-DIR/.local/lib $LDFLAGS"
export MANPATH="/YOUR-HOME-DIR/.local/share/man/:$MANPATH"
export CPPFLAGS="-I/YOUR-HOME-DIR/.local/include $CPPFLAGS"
export INFOPATH="/YOUR-HOME-DIR/.local/share/info/:$INFOPATH"
export LD_LIBRARY_PATH="/YOUR-HOME-DIR/.local/lib:$LD_LIBRARY_PATH"
```

Afterwards, you just need to add an extra `--prefix=/YOUR-HOME-DIR/.local` to the `./configure` command of the software that you intend to install. Everything else will be the same as a standard build and install, see Section 1.1 [Quick start], page 1.

3.3.1.3 Executable names

At first sight, the names of the executables for each program might seem to be uncommonly long, for example `astnoisechisel` or `astcrop`. We could have chosen terse (and cryptic) names like most programs do. We chose this complete naming convention (something like the commands in TEX) so you don't have to spend too much time remembering what the name of a specific program was. Such complete names also enable you to easily search for the programs.

To facilitate typing the names in, we suggest using the shell auto-complete. With this facility you can find the executable you want very easily. It is very similar to file name completion in the shell. For example, simply by typing the letters below (where `[TAB]` stands for the Tab key on your keyboard)

```
$ ast[TAB][TAB]
```

you will get the list of all the available executables that start with `ast` in your `PATH` environment variable directories. So, all the Gnuastro executables installed on your system will be listed. Typing the next letter for the specific program you want along with a Tab, will limit this list until you get to your desired program.

In case all of this does not convince you and you still want to type short names, some suggestions are given below. You should have in mind though, that if you are writing a shell script that you might want to pass on to others, it is best to use the standard name because other users might not have adopted the same customization. The long names also serve as a form of documentation in such scripts. A similar reasoning can be given for option names in scripts: it is good practice to always use the long formats of the options in shell scripts, see Section 4.1.1.2 [Options], page 93.

The simplest solution is making a symbolic link to the actual executable. For example let's assume you want to type `ic` to run Crop instead of `astcrop`. Assuming you installed Gnuastro executables in `/usr/local/bin` (default) you can do this simply by running the following command as root:

```
# ln -s /usr/local/bin/astcrop /usr/local/bin/ic
```

In case you update Gnuastro and a new version of Crop is installed, the default executable name is the same, so your custom symbolic link still works.

The installed executable names can also be set using options to `$ ./configure`, see Section 3.3.1 [Configuring], page 76. GNU Autoconf (which configures Gnuastro for your particular system), allows the builder to change the name of programs with the three options `--program-prefix`, `--program-suffix` and `--program-transform-name`. The first two are for adding a fixed prefix or suffix to all the programs that will be installed. This will actually make all the names longer! You can use it to add versions of program names to the programs in order to simultaneously have two executable versions of a program.

The third configure option allows you to set the executable name at install time using the SED program. SED is a very useful 'stream editor'. There are various resources on the internet to use it effectively. However, we should caution that using configure options will change the actual executable name of the installed program and on every re-install (an update for example), you have to also add this option to keep the old executable name updated. Also note that the documentation or configuration files do not change from their standard names either.

For example, let's assume that typing `ast` on every invocation of every program is really annoying you! You can remove this prefix from all the executables at configure time by adding this option:

```
$ ./configure --program-transform-name='s/ast/ /'
```

3.3.1.4 Configure and build in RAM

Gnuastro's configure and build process (the GNU build system) involves the creation, reading, and modification of a large number of files (input/output, or I/O). Therefore file I/O issues can directly affect the work of developers who need to configure and build Gnuastro numerous times. Some of these issues are listed below:

- I/O will cause wear and tear on both the HDDs (mechanical failures) and SSDs (decreasing the lifetime).
- Having the built files mixed with the source files can greatly affect backing up (synchronization) of source files (since it involves the management of a large number of small files that are regularly changed. Backup software can of course be configured to ignore the built files and directories. However, since the built files are mixed with the source files and can have a large variety, this will require a high level of customization.

One solution to address both these problems is to use the tmpfs file system (`https://en.wikipedia.org/wiki/Tmpfs`). Any file in tmpfs is actually stored in the RAM (and possibly SAWP), not on HDDs or SSDs. The RAM is built for extensive and fast I/O. Therefore the large number of file I/Os associated with configuring and building will not harm the HDDs or SSDs. Due to the volatile nature of RAM, files in the tmpfs file-system will be permanently lost after a power-off. Since all configured and built files are derivative files (not files that have been directly written by hand) there is no problem in this and this feature can be considered as an automatic cleanup.

The modern GNU C library (and thus the Linux kernel) defines the `/dev/shm` directory for this purpose in the RAM (POSIX shared memory). To build in it, you can use the GNU build system's ability to build in a separate directory (not necessarily in the source

directory) as shown below. Just set `SRCDIR` as the address of Gnuastro's top source directory (for example, the unpacked tarball).

```
$ mkdir /dev/shm/tmp-gnuastro-build
$ cd /dev/shm/tmp-gnuastro-build
$ SRCDIR/configure --srcdir=SRCDIR
$ make
```

Gnuastro comes with a script to simplify this process of configuring and building in a different directory (a "clean" build), for more see Section 3.3.2 [Separate build and source directories], page 85.

3.3.2 Separate build and source directories

The simple steps of Section 1.1 [Quick start], page 1, will mix the source and built files. This can cause inconvenience for developers or enthusiasts following the the most recent work (see Section 3.2.2 [Version controlled source], page 72). The current section is mainly focused on this later group of Gnuastro users. If you just install Gnuastro on major releases (following Section 1.9 [Announcements], page 13), you can safely ignore this section.

When it is necessary to keep the source (which is under version control), but not the derivative (built) files (after checking or installing), the best solution is to keep the source and the built files in separate directories. One application of this is already discussed in Section 3.3.1.4 [Configure and build in RAM], page 84.

To facilitate this process of configuring and building in a separate directory, Gnuastro comes with the `developer-build` script. It is available in the top source directory and is *not* installed. It will make a directory under a given top-level directory (given to `--top-build-dir`) and build Gnuastro in there directory. It thus keeps the source completely separated from the built files. For easy access to the built files, it also makes a symbolic link to the built directory in the top source files called `build`.

When run without any options, default values will be used for its configuration. As with Gnuastro's programs, you can inspect the default values with `-P` (or `--printparams`, the output just looks a little different here). The default top-level build directory is `/dev/shm`: the shared memory directory in RAM on GNU/Linux systems as described in Section 3.3.1.4 [Configure and build in RAM], page 84.

Besides these, it also has some features to facilitate the job of developers or bleeding edge users like the `--debug` option to do a fast build, with debug information, no optimization, and no shared libraries. Here is the full list of options you can feed to this script to configure its operations.

Not all Gnuastro's common program behavior usable here: `developer-build` is just a non-installed script with a very limited scope as described above. It thus doesn't have all the common option behaviors or configuration files for example.

White space between option and value: `developer-build` doesn't accept an `=` sign between the options and their values. It also needs at least one character between the option and its value. Therefore `-n 4` or `--numthreads 4` are acceptable, while `-n4`, `-n=4`, or `--numthreads=4` aren't. Finally multiple short option names cannot be merged: for example you can say `-c -n 4`, but unlike Gnuastro's programs, `-cn4` is not acceptable.

Reusable for other packages: This script can be used in any software which is configured and built using the GNU Build System. Just copy it in the top source directory of that software and run it from there.

`-b STR`
`--top-build-dir STR`
: The top build directory to make a directory for the build. If this option isn't called, the top build directory is `/dev/shm` (only available in GNU/Linux operating systems, see Section 3.3.1.4 [Configure and build in RAM], page 84).

`-V`
`--version`
: Print the version string of Gnuastro that will be used in the build. This string will be appended to the directory name containing the built files.

`-a`
`--autoreconf`
: Run `autoreconf -f` before building the package. In Gnuastro, this is necessary when a new commit has been made to the project history. In Gnuastro's build system, the Git description will be used as the version, see Section 1.5 [Version numbering], page 7, and Section 3.2.2.2 [Synchronizing], page 75.

`-c`
`--clean`
: Delete the contents of the build directory (clean it) before starting the configuration and building of this run.

  This is useful when you have recently pulled changes from the main Git repository, or committed a change your self and ran `autoreconf -f`, see Section 3.2.2.2 [Synchronizing], page 75. After running GNU Autoconf, the version will be updated and you need to do a clean build.

`-d`
`--debug`
: Build with debugging flags (for example to use in GNU Debugger, also known as GDB, or Valgrind), disable optimization and also the building of shared libraries. Similar to running the configure script of below

  ```
  $ ./configure --enable-debug
  ```

  Besides all the debugging advantages of building with this option, it will also be significantly speed up the build (at the cost of slower built programs). So when you are testing something small or working on the build system itself, it will be much faster to test your work with this option.

`-v`
`--valgrind`
: Build all `make check` tests within Valgrind. For more, see the description of `--enable-check-with-valgrind` in Section 3.3.1.1 [Gnuastro configure options], page 77.

`-j INT`
`--jobs INT`
: The maximum number of threads/jobs for Make to build at any moment. As the name suggests (Make has an identical option), the number given to this option is directly passed on to any call of Make with its `-j` option.

`-C`
`--check`
: After finishing the build, also run `make check`. By default, `make check` isn't run because the developer usually has their own checks to work on (for example defined in `tests/during-dev.sh`).

`-i`
`--install`
: After finishing the build, also run `make install`.

`-D`
`--dist`
: Run `make dist-lzip pdf` to build a distribution tarball (in `.tar.lz` format) and a PDF manual. This can be useful for archiving, or sending to colleagues who don't use Git for an easy build and manual.

`-u STR`

`--upload STR`
: Activate the `--dist` (`-D`) option, then use secure copy (`scp`, part of the SSH tools) to copy the tarball and PDF to the `src` and `pdf` sub-directories of the specified server and its directory (value to this option). For example `--upload my-server:dir`, will copy the tarball in the `dir/src`, and the PDF manual in `dir/pdf` of `my-server` server. It will then make a symbolic link in the top server directory to the tarball that is called `gnuastro-latest.tar.lz`.

`-p`
`--publish`
: Short for `--autoreconf --clean --debug --check --upload STR`. `--debug` is added because it will greatly speed up the build. It will have no effect on the produced tarball. This is good when you have made a commit and are ready to publish it on your server (if nothing crashes). Recall that if any of the previous steps fail the script aborts.

`-I`

`--install-archive`
: Short for `--autoreconf --clean --check --install --dist`. This is useful when you actually want to install the commit you just made (if the build and checks succeed). It will also produce a distribution tarball and PDF manual for easy access to the installed tarball on your system at a later time.

Ideally, Gnuastro's Git version history makes it easy for a prepared system to revert back to a different point in history. But Gnuastro also needs to bootstrap files and also your collaborators might (usually do!) find it too much of a burden to do the bootstrapping themselves. So it is convenient to have a tarball and PDF manual of the version you have installed (and are using in your research) handily available.

`-h`
`--help`
`-P`
`--printparams`
Print a description of this script along with all the options and their current values.

3.3.3 Tests

After successfully building (compiling) the programs with the `$ make` command you can check the installation before installing. To run the tests, run

```
$ make check
```

For every program some tests are designed to check some possible operations. Running the command above will run those tests and give you a final report. If everything is OK and you have built all the programs, all the tests should pass. In case any of the tests fail, please have a look at Section 3.3.5 [Known issues], page 89, and if that still doesn't fix your problem, look that the `./tests/test-suite.log` file to see if the source of the error is something particular to your system or more general. If you feel it is general, please contact us because it might be a bug. Note that the tests of some programs depend on the outputs of other program's tests, so if you have not installed them they might be skipped or fail. Prior to releasing every distribution all these tests are checked. If you have a reasonably modern terminal, the outputs of the successful tests will be colored green and the failed ones will be colored red.

These scripts can also act as a good set of examples for you to see how the programs are run. All the tests are in the `tests/` directory. The tests for each program are shell scripts (ending with `.sh`) in a sub-directory of this directory with the same name as the program. See Section 12.7 [Test scripts], page 458, for more detailed information about these scripts in case you want to inspect them.

3.3.4 A4 print book

The default print version of this book is provided in the letter paper size. If you would like to have the print version of this book on paper and you are living in a country which uses A4, then you can rebuild the book. The great thing about the GNU build system is that the book source code which is in Texinfo is also distributed with the program source code, enabling you to do such customization (hacking).

In order to change the paper size, you will need to have GNU Texinfo installed. Open `doc/gnuastro.texi` with any text editor. This is the source file that created this book. In the first few lines you will see this line:

```
@c@afourpaper
```

In Texinfo, a line is commented with `@c`. Therefore, un-comment this line by deleting the first two characters such that it changes to:

```
@afourpaper
```

Save the file and close it. You can now run

```
$ make pdf
```

and the new PDF book will be available in `SRCdir/doc/gnuastro.pdf`. By changing the `pdf` in `$ make pdf` to `ps` or `dvi` you can have the book in those formats. Note that you can do this for any book that is in Texinfo format, they might not have `@afourpaper` line, so you can add it close to the top of the Texinfo source file.

3.3.5 Known issues

Depending on your operating system and the version of the compiler you are using, you might confront some known problems during the configuration (`$ ./configure`), compilation (`$ make`) and tests (`$ make check`). Here, their solutions are discussed.

- `$ ./configure`: *Configure complains about not finding a library even though you have installed it.* The possible solution is based on how you installed the package:
  - From your distribution's package manager. Most probably this is because your distribution has separated the header files of a library from the library parts. Please also install the 'development' packages for those libraries too. Just add a `-dev` or `-devel` to the end of the package name and re-run the package manager. This will not happen if you install the libraries from source. When installed from source, the headers are also installed.
  - From source. Then your linker is not looking where you installed the library. If you followed the instructions in this chapter, all the libraries will be installed in `/usr/local/lib`. So you have to tell your linker to look in this directory. To do so, configure Gnuastro like this:

    ```
    $ ./configure LDFLAGS="-L/usr/local/lib"
    ```

    If you want to use the libraries for your other programming projects, then export this environment variable in a start-up script similar to the case for `LD_LIBRARY_PATH` explained below, also see Section 3.3.1.2 [Installation directory], page 79.
- `$ make`: *Complains about an unknown function on a non-GNU based operating system.* In this case, please run `$ ./configure` with the `--enable-gnulibcheck` option to see if the problem is from the GNU Portability Library (Gnulib) not supporting your system or if there is a problem in Gnuastro, see Section 3.3.1.1 [Gnuastro configure options], page 77. If the problem is not in Gnulib and after all its tests you get the same complaint from `make`, then please contact us at `bug-gnuastro@gnu.org`. The cause is probably that a function that we have used is not supported by your operating system and we didn't included it along with the source tar ball. If the function is available in Gnulib, it can be fixed immediately.
- `$ make`: *Can't find the headers (.h files) of installed libraries.* Your C pre-processor (CPP) isn't looking in the right place. To fix this, configure Gnuastro with an additional `CPPFLAGS` like below (assuming the library is installed in `/usr/local/include`:

  ```
  $ ./configure CPPFLAGS="-I/usr/local/include"
  ```

If you want to use the libraries for your other programming projects, then export this environment variable in a start-up script similar to the case for `LD_LIBRARY_PATH` explained below, also see Section 3.3.1.2 [Installation directory], page 79.

- `$ make check`: *Only the first couple of tests pass, all the rest fail or get skipped.* It is highly likely that when searching for shared libraries, your system doesn't look into the `/usr/local/lib` directory (or wherever you installed Gnuastro or its dependencies). To make sure it is added to the list of directories, add the following line to your `~/.bashrc` file and restart your terminal. Don't forget to change `/usr/local/lib` if the libraries are installed in other (non-standard) directories.

  ```
  export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
  ```

  You can also add more directories by using a colon '`:`' to separate them. See Section 3.3.1.2 [Installation directory], page 79, and Section 11.1.2 [Linking], page 324, to learn more on the `PATH` variables and dynamic linking respectively.
- `$ make check`: *The tests relying on external programs (for example* `fitstopdf.sh` *fail.)* This is probably due to the fact that the version number of the external programs is too old for the tests we have preformed. Please update the program to a more recent version. For example to create a PDF image, you will need GPL Ghostscript, but older versions do not work, we have successfully tested it on version 9.15. Older versions might cause a failure in the test result.
- `$ make pdf`: *The PDF book cannot be made.* To make a PDF book, you need to have the GNU Texinfo program (like any program, the more recent the better). A working TEX program is also necessary, which you can get from Tex Live[25].
- After `make check`: do not copy the programs' executables to another (for example, the installation) directory manually (using `cp`, or `mv` for example). In the default configuration[26], the program binaries need to link with Gnuastro's shared library which is also built and installed with the programs. Therefore, to run successfully before and after installation, linking modifications need to be made by GNU Libtool at installation time. `make install` does this internally, but a simple copy might give linking errors when you run it. If you need to copy the executables, you can do so after installation.

If your problem was not listed above, please file a bug report (Section 1.7 [Report a bug], page 11).

[25] `https://www.tug.org/texlive/`

[26] If you configure Gnuastro with the `--disable-shared` option, then the libraries will be statically linked to the programs and this problem won't exist, see Section 11.1.2 [Linking], page 324.

4 Common program behavior

All the programs in Gnuastro share a set of common behavior mainly to do with user interaction to facilitate their usage and development. This includes how to feed input datasets into the programs, how to configure them, specifying the outputs, numerical data types, treating columns of information in tables and etc. This chapter is devoted to describing this common behavior in all programs. Because the behaviors discussed here are common to several programs, they are not repeated in each program's description.

In Section 4.1 [Command-line], page 91, a very general description of running the programs on the command-line is discussed, like difference between arguments and options, as well as options that are common/shared between all programs. None of Gnuastro's programs keep any internal configuration value (values for their different operational steps), they read their configuration primarily from the command-line, then from specific files in directory, user, or system-wide settings. Using these configuration files can greatly help reproducible and robust usage of Gnuastro, see Section 4.2 [Configuration files], page 106, for more.

It is not possible to always have the different options and configurations of each program on the top of your head. It is very natural to forget the options of a program, their current default values, or how it should be run and what it did. Gnuastro's programs have multiple ways to help you refresh your memory in multiple levels (just an option name, a short description, or fast access to the relevant section of the manual. See Section 4.3 [Getting help], page 109, for more for more on benefiting from this very convenient feature.

Many of the programs use the multi-threaded character of modern CPUs, in Section 4.4 [Multi-threaded operations], page 112, we'll discuss how you can configure this behavior, along with some tips on making best use of them. In Section 4.5 [Numeric data types], page 115, we'll review the various types to store numbers in your datasets: setting the proper type for the usage context[1] can greatly improve the file size and also speed of reading, writing or processing them.

We'll then look into the recognized table formats in Section 4.6 [Tables], page 117, and how large datasets are broken into tiles, or mesh grid in Section 4.7 [Tessellation], page 123. Finally, we'll take a look at the behavior regarding output files: Section 4.8 [Automatic output], page 124, describes how the programs set a default name for their output when you don't give one explicitly (using `--output`). When the output is a FITS file, all the programs also store some very useful information in the header that is discussed in Section 4.9 [Output FITS files], page 125.

4.1 Command-line

Gnuastro's programs are customized through the standard Unix-like command-line environment and GNU style command-line options. Both are very common in many Unix-like operating system programs. In Section 4.1.1 [Arguments and options], page 92, we'll start with the difference between arguments and options and elaborate on the GNU style of op-

[1] For example if the values in your dataset can only be integers between 0 or 65000, store them in a unsigned 16-bit type, not 64-bit floating point type (which is the default in most systems). It takes four times less space and is much faster to process.

tions. Afterwards, in Section 4.1.2 [Common options], page 95, we'll go into the detailed list of all the options that are common to all the programs in Gnuastro.

4.1.1 Arguments and options

When you type a command on the command-line, it is passed onto the shell (a generic name for the program that manages the command-line) as a string of characters. As an example, see the "Invoking ProgramName" sections in this manual for some examples of commands with each program, like Section 5.3.1 [Invoking Table], page 148, Section 5.1.1 [Invoking Fits], page 130, or Section 7.1.4 [Invoking Statistics], page 215.

The shell then brakes up your string into separate *tokens* or *words* using any *metacharacters* (like white-space, tab, `|`, `>` or `;`) that are in the string. On the command-line, the first thing you usually enter is the name of the program you want to run. After that, you can specify two types of tokens: *arguments* and *options*. In the GNU-style, arguments are those tokens that are not preceded by any hyphens (`-`, see Section 4.1.1.1 [Arguments], page 93). Here is one example:

```
$ astcrop --center=53.162551,-27.789676 -w10/3600 --mode=wcs udf.fits
```

In the example above, we are running Section 6.1 [Crop], page 151, to crop a region of width 10 arc-seconds centered at the given RA and Dec from the input Hubble Ultra-Deep Field (UDF) FITS image. Here, the argument is `udf.fits`. Arguments are most commonly the input file names containing your data. Options start with one or two hyphens, followed by an identifier for the option (the option's name, for example, `--center`, `-w`, `--mode` in the example above) and its value (anything after the option name, or the optional `=` character). Through options you can configure how the program runs (interprets the data you provided).

Arguments can be mandatory and optional and unlike options, they don't have any identifiers. Hence, when there multiple arguments, their order might also matter (for example in `cp` which is used for copying one file to another location). The outputs of `--usage` and `--help` shows which arguments are optional and which are mandatory, see Section 4.3.1 [`--usage`], page 109.

As their name suggests, *options* can be considered to be optional and most of the time, you don't have to worry about what order you specify them in. When the order does matter, or the option can be invoked multiple times, it is explicitly mentioned in the "Invoking ProgramName" section of each program (this is a very important aspect of an option).

In case your arguments or option values contain any of the shell's meta-characters, you have to quote them. If there is only one such character, you can use a backslash (`\`) before it. If there are multiple, it might be easier to simply put your whole argument or option value inside of double quotes (`"`). In such cases, everything inside the double quotes will be seen as one token or word.

For example, let's say you want to specify the header data unit (HDU) of your FITS file using a complex expression like '`3; images(exposure > 100)`'. If you simply add these after the `--hdu` (`-h`) option, the programs in Gnuastro will read the value to the HDU option as '`3`' and run. Then, the shell will attempt to run a separate command '`images(exposure > 100)`' and complain about a syntax error. This is because the semicolon (`;`) is an 'end of command' character in the shell. To solve this problem you can simply put double quotes around the whole string you want to pass to `--hdu` as seen below:

```
$ astcrop --hdu="3; images(exposure > 100)" image.fits
```

4.1.1.1 Arguments

In Gnuastro, arguments are almost exclusively used as the input data file names. Please consult the first few paragraph of the "Invoking ProgramName" section for each program for a description of what it expects as input, how many arguments, or input data, it accepts, or in what order. Everything particular about how a program treats arguments, is explained under the "Invoking ProgramName" section for that program.

Generally, if there is a standard file name extension for a particular format, that filename extension is used to separate the kinds of arguments. The list below shows the data formats that are recognized in Gnuastro's programs based on their file name endings. Any argument that doesn't end with the specified extensions below is considered to be a text file (usually catalogs, see Section 4.6 [Tables], page 117). In some cases, a program can accept specific formats, for example Section 5.2 [ConvertType], page 138, also accepts `.jpg` images.

- `.fits`: The standard file name ending of a FITS image.
- `.fit`: Alternative (3 character) FITS suffix.
- `.fits.Z`: A FITS image compressed with `compress`.
- `.fits.gz`: A FITS image compressed with GNU zip (gzip).
- `.fits.fz`: A FITS image compressed with `fpack`.
- `.imh`: IRAF format image file.

Through out this book and in the command-line outputs, whenever we want to generalize all such astronomical data formats in a text place holder, we will use `ASTRdata`, we will assume that the extension is also part of this name. Any file ending with these names is directly passed on to CFITSIO to read. Therefore you don't necessarily have to have these files on your computer, they can also be located on an FTP or HTTP server too, see the CFITSIO manual for more information.

CFITSIO has its own error reporting techniques, if your input file(s) cannot be opened, or read, those errors will be printed prior to the final error by Gnuastro.

4.1.1.2 Options

Command-line options allow configuring the behavior of a program in all GNU/Linux applications for each particular execution on a particular input data. A single option can be called in two ways: *long* or *short*. All options in Gnuastro accept the long format which has two hyphens an can have many characters (for example `--hdu`). Short options only have one hyphen (`-`) followed by one character (for example `-h`). You can see some examples in the list of options in Section 4.1.2 [Common options], page 95, or those for each program's "Invoking ProgramName" section. Both formats are shown for those which support both. First the short is shown then the long.

Usually, the short options are for when you are writing on the command-line and want to save keystrokes and time. The long options are good for shell scripts, where you aren't usually rushing. Long options provide a level of documentation, since they are more descriptive and less cryptic. Usually after a few months of not running a program, the short options will be forgotten and reading your previously written script will not be easy.

Some options need to be given a value if they are called and some don't. You can think of the latter type of options as on/off options. These two types of options can be distinguished using the output of the `--help` and `--usage` options, which are common to all GNU software, see Section 4.3 [Getting help], page 109. In Gnuastro we use the following strings to specify when the option needs a value and what format that value should be in. More specific tests will be done in the program and if the values are out of range (for example negative when the program only wants a positive value), an error will be reported.

`INT` The value is read as an integer.

`FLT` The value is read as a float. There are generally two types, depending on the context. If they are for fractions, they will have to be less than or equal to unity.

`STR` The value is read as a string of characters (for example a file name) or other particular settings like a HDU name, see below.

To specify a value in the short format, simply put the value after the option. Note that since the short options are only one character long, you don't have to type anything between the option and its value. For the long option you either need white space or an `=` sign, for example `-h2`, `-h 2`, `--hdu 2` or `--hdu=2` are all equivalent.

The short format of on/off options (those that don't need values) can be concatenated for example these two hypothetical sequences of options are equivalent: `-a -b -c4` and `-abc4`. As an example, consider the following command to run Crop:

```
$ astcrop -Dr3 --wwidth 3 catalog.txt --deccol=4 ASTRdata
```

The `$` is the shell prompt, `astcrop` is the program name. There are two arguments (`catalog.txt` and `ASTRdata`) and four options, two of them given in short format (`-D`, `-r`) and two in long format (`--width` and `--deccol`). Three of them require a value and one (`-D`) is an on/off option.

If an abbreviation is unique between all the options of a program, the long option names can be abbreviated. For example, instead of typing `--printparams`, typing `--print` or maybe even `--pri` will be enough, if there are conflicts, the program will warn you and show you the alternatives. Finally, if you want the argument parser to stop parsing arguments beyond a certain point, you can use two dashes: `--`. No text on the command-line beyond these two dashes will be parsed.

Gnuastro has two types of options with values, those that only take a single value are the most common type. If these options are repeated or called more than once on the command-line, the value of the last time it was called will be assigned to it. This is very useful when you are testing/experimenting. Let's say you want to make a small modification to one option value. You can simply type the option with a new value in the end of the command and see how the script works. If you are satisfied with the change, you can remove the original option for human readability. If the change wasn't satisfactory, you can remove the one you just added and not worry about forgetting the original value. Without this capability, you would have to memorize or save the original value somewhere else, run the command and then change the value again which is not at all convenient and is potentially cause lots of bugs.

On the other hand, some options can be called multiple times in one run of a program and can thus take multiple values (for example see the `--column` option in Section 5.3.1

[Invoking Table], page 148. In these cases, the order of stored values is the same order that you specified on the command-line.

Gnuastro's programs don't keep any internal default values, so some options are mandatory and if they don't have a value, the program will complain and abort. Most programs have many such options and typing them by hand on every call is impractical. To facilitate the user experience, after parsing the command-line, Gnuastro's programs read special configuration files to get the necessary values for the options you haven't identified on the command-line. These configuration files are fully described in Section 4.2 [Configuration files], page 106.

CAUTION: In specifying a file address, if you want to use the shell's tilde expansion (`~`) to specify your home directory, leave at least one space between the option name and your value. For example use `-o ~/test`, `--output ~/test` or `--output= ~/test`. Calling them with `-o~/test` or `--output=~/test` will disable shell expansion.

CAUTION: If you forget to specify a value for an option which requires one, and that option is the last one, Gnuastro will warn you. But if it is in the middle of the command, it will take the text of the next option or argument as the value which can cause undefined behavior.

NOTE: In some contexts Gnuastro's counting starts from 0 and in others 1. You can assume by default that counting starts from 1, if it starts from 0 for a special option, it will be explicitly mentioned.

4.1.2 Common options

To facilitate the job of the users and developers, all the programs in Gnuastro share some basic command-line options for the options that are common to many of the programs. The full list is classified as Section 4.1.2.1 [Input/Output options], page 95, Section 4.1.2.2 [Processing options], page 98, and Section 4.1.2.3 [Operating mode options], page 100. In some programs, some of the options are irrelevant, but still recognized (you won't get an unrecognized option error, but the value isn't used). Unless otherwise mentioned, these options are identical between all programs.

4.1.2.1 Input/Output options

These options are to do with the input and outputs of the various programs.

`--stdintimeout`
: Number of micro-seconds to wait for writing/typing in the *first line* of standard input from the command-line (see Section 4.1.3 [Standard input], page 104). This is only relevant for programs that also accept input from the standard input, *and* you want to manually write/type the contents on the terminal. When the standard input is already connected to a pipe (output of another program), there won't be any waiting (hence no timeout, thus making this option redundant).

If the first line-break (for example with the `ENTER` key) is not provided before the timeout, the program will abort with an error that no input was given. Note that this time interval is *only* for the first line that you type. Once the first line is given, the program will assume that more data will come and accept rest of your inputs without any time limit. You need to specify the ending of the standard input, for example by pressing `CTRL-D` after a new line.

Note that any input you write/type into a program on the command-line with Standard input will be discarded (lost) once the program is finished. It is only recoverable manually from your command-line (where you actually typed) as long as the terminal is open. So only use this feature when you are sure that you don't need the dataset (or have a copy of it somewhere else).

`-h STR/INT`
`--hdu=STR/INT`

The name or number of the desired Header Data Unit, or HDU, in the FITS image. A FITS file can store multiple HDUs or extensions, each with either an image or a table or nothing at all (only a header). Note that counting of the extensions starts from 0(zero), not 1(one). Counting from 0 is forced on us by CFITSIO which directly reads the value you give with this option (see Section 3.1.1.2 [CFITSIO], page 62). When specifying the name, case is not important so `IMAGE`, `image` or `ImAgE` are equivalent.

CFITSIO has many capabilities to help you find the extension you want, far beyond the simple extension number and name. See CFITSIO manual's "HDU Location Specification" section for a very complete explanation with several examples. A `#` is appended to the string you specify for the HDU[2] and the result is put in square brackets and appended to the FITS file name before calling CFITSIO to read the contents of the HDU for all the programs in Gnuastro.

`-s STR`
`--searchin=STR`

Where to match/search for columns when the column identifier wasn't a number, see Section 4.6.3 [Selecting table columns], page 121. The acceptable values are `name`, `unit`, or `comment`. This option is only relevant for programs that take table columns as input.

`-I`
`--ignorecase`

Ignore case while matching/searching column meta-data (in the field specified by the `--searchin`). The FITS standard suggests to treat the column names as case insensitive, which is strongly recommended here also but is not enforced. This option is only relevant for programs that take table columns as input.

This option is not relevant to Section 11.2 [BuildProgram], page 328, hence in that program the short option `-I` is used for include directories, not to ignore case.

[2] With the `#` character, CFITSIO will only read the desired HDU into your memory, not all the existing HDUs in the fits file.

`-o STR`
`--output=STR`
The name of the output file or directory. With this option the automatic output names explained in Section 4.8 [Automatic output], page 124, are ignored.

`-T STR`
`--type=STR`
The data type of the output depending on the program context. This option isn't applicable to some programs like Section 5.1 [Fits], page 128, and will be ignored by them. The different acceptable values to this option are fully described in Section 4.5 [Numeric data types], page 115.

`-D`
`--dontdelete`
By default, if the output file already exists, Gnuastro's programs will silently delete it and put their own outputs in its place. When this option is activated, if the output file already exists, the programs will not delete it, will warn you, and will abort.

`-K`
`--keepinputdir`
In automatic output names, don't remove the directory information of the input file names. As explained in Section 4.8 [Automatic output], page 124, if no output name is specified (with `--output`), then the output name will be made in the existing directory based on your input's file name (ignoring the directory of the input). If you call this option, the directory information of the input will be kept and the automatically generated output name will be in the same directory as the input (usually with a suffix added). Note that his is only relevant if you are running the program in a different directory than the input data.

`-t STR`
`--tableformat=STR`
The output table's type. This option is only relevant when the output is a table and its format cannot be deduced from its filename. For example, if a name ending in `.fits` was given to `--output`, then the program knows you want a FITS table. But there are two types of FITS tables: FITS ASCII, and FITS binary. Thus, with this option, the program is able to identify which type you want. The currently recognized values to this option are:

`txt` A plain text table with white-space characters between the columns (see Section 4.6.2 [Gnuastro text table format], page 119).

`fits-ascii`
A FITS ASCII table (see Section 4.6.1 [Recognized table formats], page 117).

`fits-binary`
A FITS binary table (see Section 4.6.1 [Recognized table formats], page 117).

4.1.2.2 Processing options

Some processing steps are common to several programs, so they are defined as common options to all programs. Note that this class of common options is thus necessarily less common between all the programs than those described in Section 4.1.2.1 [Input/Output options], page 95, or Section 4.1.2.3 [Operating mode options], page 100, options. Also, if they are irrelevant for a program, these options will not display in the `--help` output of the program.

`--minmapsize=INT`
: The minimum size (in bytes) to store the contents of each main processing array of a program as a file (on the non-volatile HDD/SSD), not in RAM. This can be very useful for large datasets which can be very memory intensive such that your RAM will not be sufficient to keep/process them. A random filename is assigned to the array which will keep the contents of the array as long as it is necessary.

  If the `.gnuastro` directory exists and is writable, then the random file will be placed there. Otherwise, the `.gnuastro_mmap` directory will be checked. If `.gnuastro_mmap` does not exist, or is not writable also, the random file will be directly written in the current directory with the `.gnuastro_mmap_` prefix (followed by some random characters).

  When this option has a value of `0` (zero), all arrays that use this option in a program will actually be in a file (not in RAM). When the value is `-1` (largest possible number in the unsigned integer types) these arrays will be definitely allocated in RAM. However, for complex programs like Section 7.2 [NoiseChisel], page 225, it is recommended to not set it to `0`, but a value like `10000` so the many small arrays necessary during processing are stored in RAM and only larger ones are saved as a file.

  Please note that using a non-volatile file (in the HDD/SDD) instead of RAM can significantly increase the program's running time, especially on HDDs. So it is best to give this option large values by default. You can then decrease it for a specific program's invocation on a large input after you see memory issues arise (for example an error, or the program not aborting and fully consuming your memory).

  The random file will be deleted once it is no longer needed by the program. The `.gnuastro` directory will also be deleted if it has no other contents (you may also have configuration files in this directory, see Section 4.2 [Configuration files], page 106). If you see randomly named files remaining in this directory when the program finishes normally, please send us a bug report so we address the problem, see Section 1.7 [Report a bug], page 11.

`-Z INT[,INT[,...]]`
`--tilesize=[,INT[,...]]`
: The size of regular tiles for tessellation, see Section 4.7 [Tessellation], page 123. For each dimension an integer length (in units of data-elements or pixels) is necessary. If the number of input dimensions is different from the number of values given to this option, the program will stop with an error. Values must be

separated by commas (`,`) and can also be fractions (for example `4/2`). If they are fractions, the result must be an integer, otherwise an error will be printed.

`-M INT[,INT[,...]]`
`--numchannels=INT[,INT[,...]]`
The number of channels for larger input tessellation, see Section 4.7 [Tessellation], page 123. The number and types of acceptable values are similar to `--tilesize`. The only difference is that instead of length, the integers values given to this option represent the *number* of channels, not their size.

`-F FLT`
`--remainderfrac=FLT`
The fraction of remainder size along all dimensions to add to the first tile. See Section 4.7 [Tessellation], page 123, for a complete description. This option is only relevant if `--tilesize` is not exactly divisible by the input dataset's size in a dimension. If the remainder size is larger than this fraction (compared to `--tilesize`), then the remainder size will be added with one regular tile size and divided between two tiles at the start and end of the given dimension.

`--workoverch`
Ignore the channel borders for the high-level job of the given application. As a result, while the channel borders are respected in defining the small tiles (such that no tile will cross a channel border), the higher-level program operation will ignore them, see Section 4.7 [Tessellation], page 123.

`--checktiles`
Make a FITS file with the same dimensions as the input but each pixel is replaced with the ID of the tile that it is associated with. Note that the tile IDs start from 0. See Section 4.7 [Tessellation], page 123, for more on Tiling an image in Gnuastro.

`--oneelempertile`
When showing the tile values (for example with `--checktiles`, or when the program's output is tessellated) only use one element for each tile. This can be useful when only the relative values given to each tile compared to the rest are important or need to be checked. Since the tiles usually have a large number of pixels within them the output will be much smaller, and so easier to read, write, store, or send.

Note that when the full input size in any dimension is not exactly divisible by the given `--tilesize` in that dimension, the edge tile(s) will have different sizes (in units of the input's size), see `--remainderfrac`. But with this option, all displayed values are going to have the (same) size of one data-element. Hence, in such cases, the image proportions are going to be slightly different with this option.

If your input image is not exactly divisible by the tile size and you want one value per tile for some higher-level processing, all is not lost though. You can see how many pixels were within each tile (for example to weight the values or discard some for later processing) with Gnuastro's Statistics (see Section 7.1 [Statistics], page 208) as shown below. The output FITS file is going to have

two extensions, one with the median calculated on each tile and one with the number of elements that each tile covers. You can then use the `where` operator in Section 6.2 [Arithmetic], page 161, to set the values of all tiles that don't have the regular area to a blank value.

```
$ aststatistics --median --number --ontile input.fits    \
                --oneelempertile --output=o.fits
$ REGULAR_AREA=1600    # Check second extension of 'o.fits'.
$ astarithmetic o.fits o.fits $REGULAR_AREA ne nan where \
                -h1 -h2
```

Note that if `input.fits` also has blank values, then the median on tiles with blank values will also be ignored with the command above (which is desirable).

`--inteponlyblank`
When values are to be interpolated, only change the values of the blank elements, keep the non-blank elements untouched.

`--interpmetric=STR`
The metric to use for finding nearest neighbors. Currently it only accepts the Manhattan (or taxicab) metric with `manhattan`, or the radial metric with `radial`.

The Manhattan distance between two points is defined with $|\Delta x| + |\Delta y|$. Thus the Manhattan metric has the advantage of being fast, but at the expense of being less accurate. The radial distance is the standard definition of distance in a Euclidean space: $\sqrt{\Delta x^2 + \Delta y^2}$. It is accurate, but the multiplication and square root can slow down the processing.

`--interpnumngb=INT`
The number of nearby non-blank neighbors to use for interpolation.

4.1.2.3 Operating mode options

Another group of options that are common to all the programs in Gnuastro are those to do with the general operation of the programs. The explanation for those that are not only limited to Gnuastro but are common to all GNU programs start with (GNU option).

`--` (GNU option) Stop parsing the command-line. This option can be useful in scripts or when using the shell history. Suppose you have a long list of options, and want to see if removing some of them (to read from configuration files, see Section 4.2 [Configuration files], page 106) can give a better result. If the ones you want to remove are the last ones on the command-line, you don't have to delete them, you can just add `--` before them and if you don't get what you want, you can remove the `--` and get the same initial result.

`--usage` (GNU option) Only print the options and arguments and abort. This is very useful for when you know the what the options do, and have just forgot their long/short identifiers, see Section 4.3.1 [`--usage`], page 109.

`-?`
`--help` (GNU option) Print all options with an explanation and abort. Adding this option will print all the options in their short and long formats, also displaying

which ones need a value if they are called (with an `=` after the long format followed by a string specifying the format, see Section 4.1.1.2 [Options], page 93). A short explanation is also given for what the option is for. The program will quit immediately after the message is printed and will not do any form of processing, see Section 4.3.2 [`--help`], page 110.

`-V`
`--version`
(GNU option) Print a short message, showing the full name, version, copyright information and program authors and abort. On the first line, it will print the official name (not executable name) and version number of the program. Following this is a blank line and a copyright information. The program will not run.

`-q`
`--quiet` Don't report steps. All the programs in Gnuastro that have multiple major steps will report their steps for you to follow while they are operating. If you do not want to see these reports, you can call this option and only error/warning messages will be printed. If the steps are done very fast (depending on the properties of your input) disabling these reports will also decrease running time.

`--cite` Print all necessary information to cite and acknowledge Gnuastro in your published papers. With this option, the programs will print the BibTEX entry to include in your paper for Gnuastro in general, and the particular program's paper (if that program comes with a separate paper). It will also print the necessary acknowledgment statement to add in the respective section of your paper and it will abort. For a more complete explanation, please see Section 1.11 [Acknowledgments], page 14.

Citations and acknowledgments are vital for the continued work on Gnuastro. Gnuastro started, and is continued, based on separate research projects. So if you find any of the tools offered in Gnuastro to be useful in your research, please use the output of this command to cite and acknowledge the program (and Gnuastro) in your research paper. Thank you.

Gnuastro is still new, there is no separate paper only devoted to Gnuastro yet. Therefore currently the paper to cite for Gnuastro is the paper for NoiseChisel which is the first published paper introducing Gnuastro to the astronomical community. Upon reaching a certain point, a paper completely devoted to describing Gnuastro's many functionalities will be published, see Section 1.5.1 [GNU Astronomy Utilities 1.0], page 8.

`-P`
`--printparams`
With this option, Gnuastro's programs will read your command-line options and all the configuration files. If there is no problem (like a missing parameter or a value in the wrong format or range) and immediately before actually running, the programs will print the full list of option names, values and descriptions, sorted and grouped by context and abort. They will also report the version number, the date they were configured on your system and the time they were reported.

As an example, you can give your full command-line options and even the input and output file names and finally just add `-P` to check if all the parameters are finely set. If everything is OK, you can just run the same command (easily retrieved from the shell history, with the top arrow key) and simply remove the last two characters that showed this option.

Since no program will actually start its processing when this option is called, the otherwise mandatory arguments for each program (for example input image or catalog files) are no longer required when you call this option.

`--config=STR`

Parse `STR` as a configuration file immediately when this option is confronted (see Section 4.2 [Configuration files], page 106). The `--config` option can be called multiple times in one run of any Gnuastro program on the command-line or in the configuration files. In any case, it will be immediately read (before parsing the rest of the options on the command-line, or lines in a configuration file).

Note that by definition, options on the command-line still take precedence over those in any configuration file, including the file(s) given to this option if they are called before it. Also see `--lastconfig` and `--onlyversion` on how this option can be used for reproducible results. You can use `--checkconfig` (below) to check/confirm the parsing of configuration files.

`--checkconfig`

Print options and their values, within the command-line or configuration files, as they are parsed (see Section 4.2.2 [Configuration file precedence], page 107). If an option has already been set, or is ignored by the program, this option will also inform you with special values like `--ALREADY-SET--`. Only options that are parsed after this option are printed, so to see the parsing of all input options, it is recommended to put this option immediately after the program name before any other options.

This is a very good option to confirm where the value of each option is has been defined in scenarios where there are multiple configuration files (for debugging).

`-S`
`--setdirconf`

Update the current directory configuration file for the Gnuastro program and quit. The full set of command-line and configuration file options will be parsed and options with a value will be written in the current directory configuration file for this program (see Section 4.2 [Configuration files], page 106). If the configuration file or its directory doesn't exist, it will be created. If a configuration file exists it will be replaced (after it, and all other configuration files have been read). In any case, the program will not run.

This is the recommended method[3] to edit/set the configuration file for all future calls to Gnuastro's programs. It will internally check if your values are in the correct range and type and save them according to the configuration file format, see Section 4.2.1 [Configuration file format], page 106. So if there are

[3] Alternatively, you can use your favorite text editor.

unreasonable values to some options, the program will notify you and abort before writing the final configuration file.

When this option is called, the otherwise mandatory arguments, for example input image or catalog file(s), are no longer mandatory (since the program will not run).

`-U`
`--setusrconf`
Update the user configuration file and quit (see Section 4.2 [Configuration files], page 106). See explanation under `--setdirconf` for more details.

`--lastconfig`
This is the last configuration file that must be read. When this option is confronted in any stage of reading the options (on the command-line or in a configuration file), no other configuration file will be parsed, see Section 4.2.2 [Configuration file precedence], page 107, and Section 4.2.3 [Current directory and User wide], page 108. Like all on/off options, on the command-line, this option doesn't take any values. But in a configuration file, it takes the values of `0` or `1`, see Section 4.2.1 [Configuration file format], page 106. If it is present in a configuration file with a value of `0`, then all later occurrences of this option will be ignored.

`--onlyversion=STR`
Only run the program if Gnuastro's version is exactly equal to `STR` (see Section 1.5 [Version numbering], page 7). Note that it is not compared as a number, but as a string of characters, so `0`, or `0.0` and `0.00` are different. If the running Gnuastro version is different, then this option will report an error and abort as soon as it is confronted on the command-line or in a configuration file. If the running Gnuastro version is the same as `STR`, then the program will run as if this option was not called.

This is useful if you want your results to be exactly reproducible and not mistakenly run with an updated/newer or older version of the program. Besides internal algorithmic/behavior changes in programs, the existence of options or their names might change between versions (especially in these earlier versions of Gnuastro).

Hence, when using this option (probably in a script or in a configuration file), be sure to call it before other options. The benefit is that, when the version differs, the other options won't be parsed and you, or your collaborators/users, won't get errors saying an option in your configuration doesn't exist in the running version of the program.

Here is one example of how this option can be used in conjunction with the `--lastconfig` option. Let's assume that you were satisfied with the results of this command: `astnoisechisel image.fits --snquant=0.95` (along with various options set in various configuration files). You can save the state of NoiseChisel and reproduce that exact result on `image.fits` later by following these steps (the the extra spaces, and `\`, are only for easy readability, if you want to try it out, only one space between each token is enough).

```
$ echo "onlyversion X.XX"             > reproducible.conf
```

```
$ echo "lastconfig 1"                 >> reproducible.conf
$ astnoisechisel image.fits --snquant=0.95 -P            \
                                       >> reproducible.conf
```

`--onlyversion` was available from Gnuastro 0.0, so putting it immediately at the start of a configuration file will ensure that later, you (or others using different version) won't get a non-recognized option error in case an option was added/removed. `--lastconfig` will inform the installed NoiseChisel to not parse any other configuration files. This is done because we don't want the user's user-wide or system wide option values affecting our results. Finally, with the third command, which has a `-P` (short for `--printparams`), NoiseChisel will print all the option values visible to it (in all the configuration files) and the shell will append them to `reproduce.conf`. Hence, you don't have to worry about remembering the (possibly) different options in the different configuration files.

Afterwards, if you run NoiseChisel as shown below (telling it to read this configuration file with the `--config` option). You can be sure that there will either be an error (for version mismatch) or it will produce exactly the same result that you got before.

```
$ astnoisechisel --config=reproducible.conf
```

`--log` Some programs can generate extra information about their outputs in a log file. When this option is called in those programs, the log file will also be printed. If the program doesn't generate a log file, this option is ignored.

> `--log` **isn't thread-safe**: The log file usually has a fixed name. Therefore if two simultaneous calls (with `--log`) of a program are made in the same directory, the program will try to write to the same file. This will cause problems like unreasonable log file, undefined behavior, or a crash.

`-N INT`
`--numthreads=INT`
Use `INT` CPU threads when running a Gnuastro program (see Section 4.4 [Multi-threaded operations], page 112). If the value is zero (`0`), or this option is not given on the command-line or any configuration file, the value will be determined at run-time: the maximum number of threads available to the system when you run a Gnuastro program.

Note that multi-threaded programming is only relevant to some programs. In others, this option will be ignored.

4.1.3 Standard input

The most common way to feed the primary/first input dataset into a program is to give its filename as an argument (discussed in Section 4.1.1.1 [Arguments], page 93). When you want to run a series of programs in sequence, this means that each will have to keep the output of each program in a separate file and re-type that file's name in the next command. This can be very slow and frustrating (mis-typing a file's name).

To solve the problem, the founders of Unix defined pipes to directly feed the output of one program (its "Standard output" stream) into the "standard input" of a next program.

This removes the need to make temporary files between separate processes and became one of the best demonstrations of the Unix-way, or Unix philosophy.

Every program has three streams identifying where it reads/writes non-file inputs/outputs: *Standard input*, *Standard output*, and *Standard error*. When a program is called alone, all three are directed to the terminal that you are using. If it needs an input, it will prompt you for one and you can type it in. Or, it prints its results in the terminal for you to see.

For example, say you have a FITS table/catalog containing the B and V band magnitudes (`MAG_B` and `MAG_V` columns) of a selection of galaxies along with many other columns. If you want to see only these two columns in your terminal, can use Gnuastro's Section 5.3 [Table], page 147, program like below:

```
$ asttable cat.fits -cMAG_B,MAG_V
```

Through the Unix pipe mechanism, when the shell confronts the pipe character (`|`), it connects the standard output of the program before the pipe, to the standard input of the program after it. So it is literally a "pipe": everything that you would see printed by the first program on the command (without any pipe), is now passed to the second program (and not seen by you).

To continue the previous example, let's say you want to see the B-V color. To do this, you can pipe Table's output to AWK (a wonderful tool for processing things like plain text tables):

```
$ asttable cat.fits -cMAG_B,MAG_V | awk '{print $1-$2}'
```

But understanding the distribution by visually seeing all the numbers under each other is not too useful! You can therefore feed this single column information into Section 7.1 [Statistics], page 208, to give you a general feeling of the distribution with the same command:

```
$ asttable cat.fits -cMAG_B,MAG_V | awk '{print $1-$2}' | aststatistics
```

Gnuastro's programs that accept input from standard input, only look into the Standard input stream if there is no first argument. In other words, arguments take precedence over Standard input. When no argument is provided, the programs check if the standard input stream is already full or not (output from another program is waiting to be used). If data is present in the standard input stream, it is used.

When the standard input is empty, the program will wait `--stdintimeout` micro-seconds for you to manually enter the first line (ending with a new-line character, or the `ENTER` key, see Section 4.1.2.1 [Input/Output options], page 95). If it detects the first line in this time, there is no more time limit, and you can manually write/type all the lines for as long as it takes. To inform the program that Standard input has finished, press `CTRL-D` after a new line. If the program doesn't catch the first line before the time-out finishes, it will abort with an error saying that no input was provided.

Manual input in Standard input is discarded: Be careful that when you manually fill the Standard input, the data will be discarded once the program finishes and reproducing the result will be impossible. Therefore this form of providing input is only good for temporary tests.

Standard input currently only for plain text: Currently Standard input only works for plain text inputs like the example above. We will later allow FITS files into the programs through standard input also.

4.2 Configuration files

Each program needs a certain number of parameters to run. Supplying all the necessary parameters each time you run the program is very frustrating and prone to errors. Therefore all the programs read the values for the necessary options you have not given in the command line from one of several plain text files (which you can view and edit with any text editor). These files are known as configuration files and are usually kept in a directory named `etc/` according to the file system hierarchy standard[4].

The thing to have in mind is that none of the programs in Gnuastro keep any internal default value. All the values must either be stored in one of the configuration files or explicitly called in the command-line. In case the necessary parameters are not given through any of these methods, the program will print a missing option error and abort. The only exception to this is `--numthreads`, whose default value is determined at run-time using the number of threads available to your system, see Section 4.4 [Multi-threaded operations], page 112. Of course, you can still provide a default value for the number of threads at any of the levels below, but if you don't, the program will not abort. Also note that through automatic output name generation, the value to the `--output` option is also not mandatory on the command-line or in the configuration files for all programs which don't rely on that value as an input[5], see Section 4.8 [Automatic output], page 124.

4.2.1 Configuration file format

The configuration files for each program have the standard program executable name with a '`.conf`' suffix. When you download the source code, you can find them in the same directory as the source code of each program, see Section 12.4 [Program source], page 451.

Any line in the configuration file whose first non-white character is a `#` is considered to be a comment and is ignored. An empty line is also similarly ignored. The long name of the option should be used as an identifier. The parameter name and parameter value have to be separated by any number of 'white-space' characters: space, tab or vertical tab. By default several space characters are used. If the value of an option has space characters (most commonly for the `hdu` option), then the full value can be enclosed in double quotation signs (`"`, similar to the example in Section 4.1.1 [Arguments and options], page 92). If it is an option without a value in the `--help` output (on/off option, see Section 4.1.1.2 [Options], page 93), then the value should be `1` if it is to be 'on' and `0` otherwise.

In each non-commented and non-blank line, any text after the first two words (option identifier and value) is ignored. If an option identifier is not recognized in the configuration file, the name of the file, the line number of the unrecognized option, and the unrecognized identifier name will be reported and the program will abort. If a parameter is repeated more more than once in the configuration files, accepts only one value, and is not set on the command-line, then only the first value will be used, the rest will be ignored.

[4] `http://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard`

[5] One example of a program which uses the value given to `--output` as an input is ConvertType, this value specifies the type of the output through the value to `--output`, see Section 5.2.3 [Invoking ConvertType], page 142.

You can build or edit any of the directories and the configuration files yourself using any text editor. However, it is recommended to use the `--setdirconf` and `--setusrconf` options to set default values for the current directory or this user, see Section 4.1.2.3 [Operating mode options], page 100. With these options, the values you give will be checked before writing in the configuration file. They will also print a set of commented lines guiding the reader and will also classify the options based on their context and write them in their logical order to be more understandable.

4.2.2 Configuration file precedence

The option values in all the programs of Gnuastro will be filled in the following order. If an option only takes one value which is given in an earlier step, any value for that option in a later step will be ignored. Note that if the `lastconfig` option is specified in any step below, no other configuration files will be parsed (see Section 4.1.2.3 [Operating mode options], page 100).

1. Command-line options, for a particular run of ProgramName.
2. `.gnuastro/astprogname.conf` is parsed by ProgramName in the current directory.
3. `.gnuastro/gnuastro.conf` is parsed by all Gnuastro programs in the current directory.
4. `$HOME/.local/etc/astprogname.conf` is parsed by ProgramName in the user's home directory (see Section 4.2.3 [Current directory and User wide], page 108).
5. `$HOME/.local/etc/gnuastro.conf` is parsed by all Gnuastro programs in the user's home directory (see Section 4.2.3 [Current directory and User wide], page 108).
6. `prefix/etc/astprogname.conf` is parsed by ProgramName in the system-wide installation directory (see Section 4.2.4 [System wide], page 108, for `prefix`).
7. `prefix/etc/gnuastro.conf` is parsed by all Gnuastro programs in the system-wide installation directory (see Section 4.2.4 [System wide], page 108, for `prefix`).

The basic idea behind setting this progressive state of checking for parameter values is that separate users of a computer or separate folders in a user's file system might need different values for some parameters.

Checking the order: You can confirm/check the order of parsing configuration files using the `--checkconfig` option with any Gnuastro program, see Section 4.1.2.3 [Operating mode options], page 100. Just be sure to place this option immediately after the program name, before any other option.

As you see above, there can also be a configuration file containing the common options in all the programs: `gnuastro.conf` (see Section 4.1.2 [Common options], page 95). If options specific to one program are specified in this file, there will be unrecognized option errors, or unexpected behavior if the option has different behavior in another program. On the other hand, there is no problem with `astprogname.conf` containing common options[6].

[6] As an example, the `--setdirconf` and `--setusrconf` options will also write the common options they have read in their produced `astprogname.conf`.

Manipulating the order: You can manipulate this order or add new files with the following two options which are fully described in Section 4.1.2.3 [Operating mode options], page 100:

`--config` Allows you to define any file to be parsed as a configuration file on the command-line or within the any other configuration file. Recall that the file given to `--config` is parsed immediately when this option is confronted (on the command-line or in a configuration file).

`--lastconfig`
Allows you to stop the parsing of subsequent configuration files. Note that if this option is given in a configuration file, it will be fully read, so its position in the configuration doesn't matter (unlike `--config`).

One example of benefiting from these configuration files can be this: raw telescope images usually have their main image extension in the second FITS extension, while processed FITS images usually only have one extension. If your system-wide default input extension is 0 (the first), then when you want to work with the former group of data you have to explicitly mention it to the programs every time. With this progressive state of default values to check, you can set different default values for the different directories that you would like to run Gnuastro in for your different purposes, so you won't have to worry about this issue any more.

The same can be said about the `gnuastro.conf` files: by specifying a behavior in this single file, all Gnuastro programs in the respective directory, user, or system-wide steps will behave similarly. For example to keep the input's directory when no specific output is given (see Section 4.8 [Automatic output], page 124), or to not delete an existing file if it has the same name as a given output (see Section 4.1.2.1 [Input/Output options], page 95).

4.2.3 Current directory and User wide

For the current (local) and user-wide directories, the configuration files are stored in the hidden sub-directories named `.gnuastro/` and `$HOME/.local/etc/` respectively. Unless you have changed it, the `$HOME` environment variable should point to your home directory. You can check it by running `$ echo $HOME`. Each time you run any of the programs in Gnuastro, this environment variable is read and placed in the above address. So if you suddenly see that your home configuration files are not being read, probably you (or some other program) has changed the value of this environment variable.

Although it might cause confusions like above, this dependence on the `HOME` environment variable enables you to temporarily use a different directory as your home directory. This can come in handy in complicated situations. To set the user or current directory configuration files based on your command-line input, you can use the `--setdirconf` or `--setusrconf`, see Section 4.1.2.3 [Operating mode options], page 100.

4.2.4 System wide

When Gnuastro is installed, the configuration files that are shipped with the distribution are copied into the (possibly system wide) `prefix/etc/` directory. For more details on `prefix`, see Section 3.3.1.2 [Installation directory], page 79, (by default it is: `/usr/local`). This directory is the final place (with the lowest priority) that the programs in Gnuastro will check to retrieve parameter values.

If you remove an option and its value from the system wide configuration files, you either have to specify it in more immediate configuration files or set it each time in the command-line. Recall that none of the programs in Gnuastro keep any internal default values and will abort if they don't find a value for the necessary parameters (except the number of threads and output file name). So even though you might never expect to use an optional option, it safe to have it available in this system-wide configuration file even if you don't intend to use it frequently.

Note that in case you install Gnuastro from your distribution's repositories, `prefix` will either be set to `/` (the root directory) or `/usr`, so you can find the system wide configuration variables in `/etc/` or `/usr/etc/`. The prefix of `/usr/local/` is conventionally used for programs you install from source by your self as in Section 1.1 [Quick start], page 1.

4.3 Getting help

Probably the first time you read this book, it is either in the PDF or HTML formats. These two formats are very convenient for when you are not actually working, but when you are only reading. Later on, when you start to use the programs and you are deep in the middle of your work, some of the details will inevitably be forgotten. Going to find the PDF file (printed or digital) or the HTML webpage is a major distraction.

GNU software have a very unique set of tools for aiding your memory on the command-line, where you are working, depending how much of it you need to remember. In the past, such command-line help was known as "online" help, because they were literally provided to you 'on' the command 'line'. However, nowadays the word "online" refers to something on the internet, so that term will not be used. With this type of help, you can resume your exciting research without taking your hands off the keyboard.

Another major advantage of such command-line based help routines is that they are installed with the software in your computer, therefore they are always in sync with the executable you are actually running. Three of them are actually part of the executable. You don't have to worry about the version of the book or program. If you rely on external help (a PDF in your personal print or digital archive or HTML from the official webpage) you have to check to see if their versions fit with your installed program.

If you only need to remember the short or long names of the options, `--usage` is advised. If it is what the options do, then `--help` is a great tool. Man pages are also provided for those who are use to this older system of documentation. This full book is also available to you on the command-line in Info format. If none of these seems to resolve the problems, there is a mailing list which enables you to get in touch with experienced Gnuastro users. In the subsections below each of these methods are reviewed.

4.3.1 `--usage`

If you give this option, the program will not run. It will only print a very concise message showing the options and arguments. Everything within square brackets (`[]`) is optional. For example here are the first and last two lines of Crop's `--usage` is shown:

```
$ astcrop --usage
Usage: astcrop [-Do?IPqSVW] [-d INT] [-h INT] [-r INT] [-w INT]
            [-x INT] [-y INT] [-c INT] [-p STR] [-N INT] [--deccol=INT]
            ....
```

```
[--setusrconf] [--usage] [--version] [--wcsmode]
[ASCIIcatalog] FITSimage(s).fits
```

There are no explanations on the options, just their short and long names shown separately. After the program name, the short format of all the options that don't require a value (on/off options) is displayed. Those that do require a value then follow in separate brackets, each displaying the format of the input they want, see Section 4.1.1.2 [Options], page 93. Since all options are optional, they are shown in square brackets, but arguments can also be optional. For example in this example, a catalog name is optional and is only required in some modes. This is a standard method of displaying optional arguments for all GNU software.

4.3.2 `--help`

If the command-line includes this option, the program will not be run. It will print a complete list of all available options along with a short explanation. The options are also grouped by their context. Within each context, the options are sorted alphabetically. Since the options are shown in detail afterwards, the first line of the `--help` output shows the arguments and if they are optional or not, similar to Section 4.3.1 [`--usage`], page 109.

In the `--help` output of all programs in Gnuastro, the options for each program are classified based on context. The first two contexts are always options to do with the input and output respectively. For example input image extensions or supplementary input files for the inputs. The last class of options is also fixed in all of Gnuastro, it shows operating mode options. Most of these options are already explained in Section 4.1.2.3 [Operating mode options], page 100.

The help message will sometimes be longer than the vertical size of your terminal. If you are using a graphical user interface terminal emulator, you can scroll the terminal with your mouse, but we promised no mice distractions! So here are some suggestions:

- `Shift + PageUP` to scroll up and `Shift + PageDown` to scroll down. For most help output this should be enough. The problem is that it is limited by the number of lines that your terminal keeps in memory and that you can't scroll by lines, only by whole screens.
- Pipe to `less`. A pipe is a form of shell re-direction. The `less` tool in Unix-like systems was made exactly for such outputs of any length. You can pipe (`|`) the output of any program that is longer than the screen to it and then you can scroll through (up and down) with its many tools. For example:

  ```
  $ astnoisechisel --help | less
  ```

  Once you have gone through the text, you can quit `less` by pressing the `q` key.
- Redirect to a file. This is a less convenient way, because you will then have to open the file in a text editor! You can do this with the shell redirection tool (`>`):

  ```
  $ astnoisechisel --help > filename.txt
  ```

In case you have a special keyword you are looking for in the help, you don't have to go through the full list. GNU Grep is made for this job. For example if you only want the list of options whose `--help` output contains the word "axis" in Crop, you can run the following command:

```
$ astcrop --help | grep axis
```

If the output of this option does not fit nicely within the confines of your terminal, GNU does enable you to customize its output through the environment variable `ARGP_HELP_FMT`, you can set various parameters which specify the formatting of the help messages. For example if your terminals are wider than 70 spaces (say 100) and you feel there is too much empty space between the long options and the short explanation, you can change these formats by giving values to this environment variable before running the program with the `--help` output. You can define this environment variable in this manner:

```
$ export ARGP_HELP_FMT=rmargin=100,opt-doc-col=20
```

This will affect all GNU programs using GNU C library's `argp.h` facilities as long as the environment variable is in memory. You can see the full list of these formatting parameters in the "Argp User Customization" part of the GNU C library manual. If you are more comfortable to read the `--help` outputs of all GNU software in your customized format, you can add your customization (similar to the line above, without the `$` sign) to your `~/.bashrc` file. This is a standard option for all GNU software.

4.3.3 Man pages

Man pages were the Unix method of providing command-line documentation to a program. With GNU Info, see Section 4.3.4 [Info], page 111, the usage of this method of documentation is highly discouraged. This is because Info provides a much more easier to navigate and read environment.

However, some operating systems require a man page for packages that are installed and some people are still used to this method of command line help. So the programs in Gnuastro also have Man pages which are automatically generated from the outputs of `--version` and `--help` using the GNU help2man program. So if you run

```
$ man programname
```

You will be provided with a man page listing the options in the standard manner.

4.3.4 Info

Info is the standard documentation format for all GNU software. It is a very useful command-line document viewing format, fully equipped with links between the various pages and menus and search capabilities. As explained before, the best thing about it is that it is available for you the moment you need to refresh your memory on any command-line tool in the middle of your work without having to take your hands off the keyboard. This complete book is available in Info format and can be accessed from anywhere on the command-line.

To open the Info format of any installed programs or library on your system which has an Info format book, you can simply run the command below (change `executablename` to the executable name of the program or library):

```
$ info executablename
```

In case you are not already familiar with it, run `$ info info`. It does a fantastic job in explaining all its capabilities its self. It is very short and you will become sufficiently fluent in about half an hour. Since all GNU software documentation is also provided in Info, your whole GNU/Linux life will significantly improve.

Once you've become an efficient navigator in Info, you can go to any part of this book or any other GNU software or library manual, no matter how long it is, in a matter of seconds.

It also blends nicely with GNU Emacs (a text editor) and you can search manuals while you are writing your document or programs without taking your hands off the keyboard, this is most useful for libraries like the GNU C library. To be able to access all the Info manuals installed in your GNU/Linux within Emacs, type `Ctrl-H + i`.

To see this whole book from the beginning in Info, you can run

```
$ info gnuastro
```

If you run Info with the particular program executable name, for example `astcrop` or `astnoisechisel`:

```
$ info astprogramname
```

you will be taken to the section titled "Invoking ProgramName" which explains the inputs and outputs along with the command-line options for that program. Finally, if you run Info with the official program name, for example Crop or NoiseChisel:

```
$ info ProgramName
```

you will be taken to the top section which introduces the program. Note that in all cases, Info is not case sensitive.

4.3.5 help-gnuastro mailing list

Gnuastro maintains the help-gnuastro mailing list for users to ask any questions related to Gnuastro. The experienced Gnuastro users and some of its developers are subscribed to this mailing list and your email will be sent to them immediately. However, when contacting this mailing list please have in mind that they are possibly very busy and might not be able to answer immediately.

To ask a question from this mailing list, send a mail to `help-gnuastro@gnu.org`. Anyone can view the mailing list archives at `http://lists.gnu.org/archive/html/help-gnuastro/`. It is best that before sending a mail, you search the archives to see if anyone has asked a question similar to yours. If you want to make a suggestion or report a bug, please don't send a mail to this mailing list. We have other mailing lists and tools for those purposes, see Section 1.7 [Report a bug], page 11, or Section 1.8 [Suggest new feature], page 12.

4.4 Multi-threaded operations

Some of the programs benefit significantly when you use all the threads your computer's CPU has to offer to your operating system. The number of threads available can be larger than the number of physical (hardware) cores in the CPU (also known as Simultaneous multithreading). For example, in Intel's CPUs (those that implement its Hyper-threading technology) the number of threads is usually double the number of physical cores in your CPU. On a GNU/Linux system, the number of threads available can be found with the command `$ nproc` command (part of GNU Coreutils).

Gnuastro's programs can find the number of threads available to your system internally at run-time (when you execute the program). However, if a value is given to the `--numthreads` option, the given number will be used, see Section 4.1.2.3 [Operating mode options], page 100, and Section 4.2 [Configuration files], page 106, for ways to use this option. Thus `--numthreads` is the only common option in Gnuastro's programs with a value that doesn't have to be specified anywhere on the command-line or in the configuration files.

4.4.1 A note on threads

Spinning off threads is not necessarily the most efficient way to run an application. Creating a new thread isn't a cheap operation for the operating system. It is most useful when the input data are fixed and you want the same operation to be done on parts of it. For example one input image to Crop and multiple crops from various parts of it. In this fashion, the image is loaded into memory once, all the crops are divided between the number of threads internally and each thread cuts out those parts which are assigned to it from the same image. On the other hand, if you have multiple images and you want to crop the same region(s) out of all of them, it is much more efficient to set `--numthreads=1` (so no threads spin off) and run Crop multiple times simultaneously, see Section 4.4.2 [How to run simultaneous operations], page 113.

You can check the boost in speed by first running a program on one of the data sets with the maximum number of threads and another time (with everything else the same) and only using one thread. You will notice that the wall-clock time (reported by most programs at their end) in the former is longer than the latter divided by number of physical CPU cores (not threads) available to your operating system. Asymptotically these two times can be equal (most of the time they aren't). So limiting the programs to use only one thread and running them independently on the number of available threads will be more efficient.

Note that the operating system keeps a cache of recently processed data, so usually, the second time you process an identical data set (independent of the number of threads used), you will get faster results. In order to make an unbiased comparison, you have to first clean the system's cache with the following command between the two runs.

```
$ sync; echo 3 | sudo tee /proc/sys/vm/drop_caches
```

SUMMARY: Should I use multiple threads? Depends:

- If you only have **one** data set (image in most cases!), then yes, the more threads you use (with a maximum of the number of threads available to your OS) the faster you will get your results.
- If you want to run the same operation on **multiple** data sets, it is best to set the number of threads to 1 and use Make, or GNU Parallel, as explained in Section 4.4.2 [How to run simultaneous operations], page 113.

4.4.2 How to run simultaneous operations

There are two[7] approaches to simultaneously execute a program: using GNU Parallel or Make (GNU Make is the most common implementation). The first is very useful when you only want to do one job multiple times and want to get back to your work without actually keeping the command you ran. The second is usually for more important operations, with lots of dependencies between the different products (for example a full scientific research).

[7] A third way would be to open multiple terminal emulator windows in your GUI, type the commands separately on each and press `Enter` once on each terminal, but this is far too frustrating, tedious and prone to errors. It's therefore not a realistic solution when tens, hundreds or thousands of operations (your research targets, multiplied by the operations you do on each) are to be done.

GNU Parallel

When you only want to run multiple instances of a command on different threads and get on with the rest of your work, the best method is to use GNU parallel. Surprisingly GNU Parallel is one of the few GNU packages that has no Info documentation but only a Man page, see Section 4.3.4 [Info], page 111. So to see the documentation after installing it please run

```
$ man parallel
```

As an example, let's assume we want to crop a region fixed on the pixels (500, 600) with the default width from all the FITS images in the `./data` directory ending with `sci.fits` to the current directory. To do this, you can run:

```
$ parallel astcrop --numthreads=1 --xc=500 --yc=600 ::: \
  ./data/*sci.fits
```

GNU Parallel can help in many more conditions, this is one of the simplest, see the man page for lots of other examples. For absolute beginners: the backslash (\) is only a line breaker to fit nicely in the page. If you type the whole command in one line, you should remove it.

Make

Make is a program for building "targets" (e.g., files) using "recipes" (a set of operations) when their known "prerequisites" (other files) have been updated. It elegantly allows you to define dependency structures for building your final output and updating it efficiently when the inputs change. It is the most common infra-structure to build software today.

Scientific research methodology is very similar to software development: you start by testing a hypothesis on a small sample of objects/targets with a simple set of steps. As you are able to get promising results, you improve the method and use it on a larger, more general, sample. In the process, you will confront many issues that have to be corrected (bugs in software development jargon). Make a wonderful tool to manage this style of development. It has been used to make reproducible papers, for example see the reproduction pipeline (`https://gitlab.com/makhlaghi/NoiseChisel-paper`) of the paper introducing Section 7.2 [NoiseChisel], page 225, (one of Gnuastro's programs).

GNU Make[8] is the most common implementation which (similar to nearly all GNU programs, comes with a wonderful manual[9]). Make is very basic and simple, and thus the manual is short (the most important parts are in the first roughly 100 pages) and easy to read/understand.

Make comes with a `--jobs` (`-j`) option which allows you to specify the maximum number of jobs that can be done simultaneously. For example if you have 8 threads available to your operating system. You can run:

```
$ make -j8
```

With this command, Make will process your `Makefile` and create all the targets (can be thousands of FITS images for example) simultaneously on 8 threads, while fully respecting their dependencies (only building a file/target when its prerequisites are successfully built). Make is thus strongly recommended for

[8] `https://www.gnu.org/software/make/`

[9] `https://www.gnu.org/software/make/manual/`

managing scientific research where robustness, archiving, reproducibility and speed[10] are important.

4.5 Numeric data types

At the lowest level, the computer stores everything in terms of `1` or `0`. For example, each program in Gnuastro, or each astronomical image you take with the telescope is actually a string of millions of these zeros and ones. The space required to keep a zero or one is the smallest unit of storage, and is known as a *bit*. However, understanding and manipulating this string of bits is extremely hard for most people. Therefore, we define packages of these bits along with a standard on how to interpret the bits in each package as a *type*.

The most basic standard for reading the bits is integer numbers ($..., -2, -1, 0, 1, 2, ...$, more bits will give larger limits). The common integer types are 8, 16, 32, and 64 bits wide. For each width, there are two standards for reading the bits: signed and unsigned integers. In the former, negative numbers are allowed and in the latter, they aren't. The `unsigned` types thus have larger positive limits (one extra bit), but no negative value. When the context of your work doesn't involve negative numbers (for example counting, where negative is not defined), it is best to use the `unsigned` types. For full numerical range of all integer types, see below.

Another standard of converting a given number of bits to numbers is the floating point standard, this standard can approximately store any real number with a given precision. There are two common floating point types: 32-bit and 64-bit, for single and double precision floating point numbers respectively. The former is sufficient for data with less than 8 significant decimal digits (most astronomical data), while the latter is good for less than 16 significant decimal digits. The representation of real numbers as bits is much more complex than integers. If you are interested, you can start with the Wikipedia article (`https://en.wikipedia.org/wiki/Floating_point`).

With the conversion operators in Gnuastro's Arithmetic, you can change the types of data to each other, which is necessary in some contexts. For example the program/library, that you intend to feed the data into, only accepts floating point values, but you have an integer image. Another situation that conversion can be helpful is when you know that your data only has values that fit within `int8` or `uint16`. However it is currently formatted in the `float64` type. Operations involving floating point or larger integer types are significantly slower than integer or smaller-width types respectively. In the latter case, it also requires much more (by 8 or 4 times in the example above) storage space. So when you confront such situations and want to store/archive/transfter the data, it is best convert them to the most efficient type.

The short and long names for the recognized numeric data types in Gnuastro are listed below. Both short and long names can be used when you want to specify a type. For example, as a value to the common option `--type` (see Section 4.1.2.1 [Input/Output options], page 95), or in the information comment lines of Section 4.6.2 [Gnuastro text table format], page 119. The ranges listed below are inclusive.

[10] Besides its multi-threaded capabilities, Make will only re-build those targets that depend on a change you have made, not the whole work. For example, if you have set the prerequisites properly, you can easily test the changing of a parameter on your paper's results without having to re-do everything (which is much faster). This allows you to be much more productive in easily checking various ideas/assumptions of the different stages of your research and thus produce a more robust result for your exciting science.

`u8`
`uint8` 8-bit unsigned integers, range:
[0 to $2^8 - 1$] or [0 to 255].

`i8`
`int8` 8-bit signed integers, range:
[-2^7 to $2^7 - 1$] or [−128 to 127].

`u16`
`uint16` 16-bit unsigned integers, range:
[0 to $2^{16} - 1$] or [0 to 65535].

`i16`
`int16` 16-bit signed integers, range:
[-2^{15} to $2^{15} - 1$] or [−32768 to 32767].

`u32`
`uint32` 32-bit unsigned integers, range:
[0 to $2^{32} - 1$] or [0 to 4294967295].

`i32`
`int32` 32-bit signed integers, range:
[-2^{31} to $2^{31} - 1$] or [−2147483648 to 2147483647].

`u64`
`uint64` 64-bit unsigned integers, range
[0 to $2^{64} - 1$] or [0 to 18446744073709551615].

`i64`
`int64` 64-bit signed integers, range:
[-2^{63} to $2^{63} - 1$] or [−9223372036854775808 to 9223372036854775807].

`f32`
`float32` 32-bit (single-precision) floating point types. The maximum (minimum is its negative) possible value is 3.402823×10^{38}. Single-precision floating points can accurately represent a floating point number up to ~ 7.2 significant decimals. Given the heavy noise in astronomical data, this is usually more than sufficient for storing results.

`f64`
`float64` 64-bit (double-precision) floating point types. The maximum (minimum is its negative) possible value is $\sim 10^{308}$. Double-precision floating points can accurately represent a floating point number ~ 15.9 significant decimals. This is usually good for processing (mixing) the data internally, for example a sum of single precision data (and later storing the result as `float32`).

Some file formats don't recognize all types. Some file formats don't recognize all the types, for example the FITS standard (see Section 5.1 [Fits], page 128) does not define `uint64` in binary tables or images. When a type is not acceptable for output into a given file format, the respective Gnuastro program or library will let you know and abort. On the command-line, you can use the Section 6.2 [Arithmetic], page 161, program to convert the numerical type of a dataset, in the libraries, you can call `gal_data_copy_to_new_type`.

4.6 Tables

"A table is a collection of related data held in a structured format within a database. It consists of columns, and rows." (from Wikipedia). Each column in the table contains the values of one property and each row is a collection of properties (columns) for one target object. For example, let's assume you have just ran MakeCatalog (see Section 7.4 [MakeCatalog], page 255) on an image to measure some properties for the labeled regions (which might be detected galaxies for example) in the image. For each labeled region (detected galaxy), there will be a *row* which groups its measured properties as *columns*, one column for each property. One such property can be the object's magnitude, which is the sum of pixels with that label, or its center can be defined as the light-weighted average value of those pixels. Many such properties can be derived from the raw pixel values and their position, see Section 7.4.5 [Invoking MakeCatalog], page 266, for a long list.

As a summary, for each labeled region (or, galaxy) we have one *row* and for each measured property we have one *column*. This high-level structure is usually the first step for higher-level analysis, for example finding the stellar mass or photometric redshift from magnitudes in multiple colors. Thus, tables are not just outputs of programs, in fact it is much more common for tables to be inputs of programs. For example, to make a mock galaxy image, you need to feed in the properties of each galaxy into Section 8.1 [MakeProfiles], page 284, for it do the inverse of the process above and make a simulated image from a catalog, see Section 2.1 [Sufi simulates a detection], page 17. In other cases, you can feed a table into Section 6.1 [Crop], page 151, and it will crop out regions centered on the positions within the table, see Section 2.4 [Hubble visually checks and classifies his catalog], page 57. So to end this relatively long introduction, tables play a very important role in astronomy, or generally all branches of data analysis.

In Section 4.6.1 [Recognized table formats], page 117, the currently recognized table formats in Gnuastro are discussed. You can use any of these tables as input or ask for them to be built as output. The most common type of table format is a simple plain text file with each row on one line and columns separated by white space characters, this format is easy to read/write by eye/hand. To give it the full functionality of more specific table types like the FITS tables, Gnuastro has a special convention which you can use to give each column a name, type, unit, and comments, while still being readable by other plain text table readers. This convention is described in Section 4.6.2 [Gnuastro text table format], page 119.

When tables are input to a program, the program reading it needs to know which column(s) it should use for its desired purposes. Gnuastro's programs all follow a similar convention, on the way you can select columns in a table. They are thoroughly discussed in Section 4.6.3 [Selecting table columns], page 121.

4.6.1 Recognized table formats

The list of table formats that Gnuastro can currently read from and write to are described below. Each has their own advantage and disadvantages, so a short review of the format is also provided to help you make the best choice based on how you want to define your input tables or later use your output tables.

Plain text table

This is the most basic and simplest way to create, view, or edit the table by hand on a text editor. The other formats described below are less eye-friendly and have a more formal structure (for easier computer readability). It is fully described in Section 4.6.2 [Gnuastro text table format], page 119.

FITS ASCII tables

The FITS ASCII table extension is fully in ASCII encoding and thus easily readable on any text editor (assuming it is the only extension in the FITS file). If the FITS file also contains binary extensions (for example an image or binary table extensions), then there will be many hard to print characters. The FITS ASCII format doesn't have new line characters to separate rows. In the FITS ASCII table standard, each row is defined as a fixed number of characters (value to the `NAXIS1` keyword), so to visually inspect it properly, you would have to adjust your text editor's width to this value. All columns start at given character positions and have a fixed width (number of characters).

Numbers in a FITS ASCII table are printed into ASCII format, they are not in binary (that the CPU uses). Hence, they can take a larger space in memory, loose their precision, and take longer to read into memory. If you are dealing with integer type columns (see Section 4.5 [Numeric data types], page 115), another issue with FITS ASCII tables is that the type information for the column will be lost (there is only one integer type in FITS ASCII tables). One problem with the binary format on the other hand is that it isn't portable (different CPUs/compilers) have different standards for translating the zeros and ones. But since ASCII characters are defined on a byte and are well recognized, they are better for portability on those various systems. Gnuastro's plain text table format described below is much more portable and easier to read/write/interpret by humans manually.

Generally, as the name implies, this format is useful for when your table mainly contains ASCII columns (for example file names, or descriptions). They can be useful when you need to include columns with structured ASCII information along with other extensions in one FITS file. In such cases, you can also consider header keywords (see Section 5.1 [Fits], page 128).

FITS binary tables

The FITS binary table is the FITS standard's solution to the issues discussed with keeping numbers in ASCII format as described under the FITS ASCII table title above. Only columns defined as a string type (a string of ASCII characters) are readable in a text editor. The portability problem with binary formats discussed above is mostly solved thanks to the portability of CFITSIO (see Section 3.1.1.2 [CFITSIO], page 62) and the very long history of the FITS format which has been widely used since the 1970s.

In the case of most numbers, storing them in binary format is more memory efficient than ASCII format. For example, to store `-25.72034` in ASCII format, you need 9 bytes/characters. But if you keep this same number (to the approximate precision possible) as a 4-byte (32-bit) floating point number, you can keep/transmit it with less than half the amount of memory. When catalogs

contain thousands/millions of rows in tens/hundreds of columns, this can lead to significant improvements in memory/band-width usage. Moreover, since the CPU does its operations in the binary formats, reading the table in and writing it out is also much faster than an ASCII table.

When you are dealing with integer numbers, the compression ratio can be even better, for example if you know all of the values in a column are positive and less than `255`, you can use the `unsigned char` type which only takes one byte! If they are between `-128` and `127`, then you can use the (signed) `char` type. So if you are thoughtful about the limits of your integer columns, you can greatly reduce the size of your file and also the speed at which it is read/written. This can be very useful when sharing your results with collaborators or publishing them. To decrease the file size even more you can name your output as ending in `.fits.gz` so it is also compressed after creation. Just note that compression/decompressing is CPU intensive and can slow down the writing/reading of the file.

Fortunately the FITS Binary table format also accepts ASCII strings as column types (along with the various numerical types). So your dataset can also contain non-numerical columns.

4.6.2 Gnuastro text table format

Plain text files are the most generic, portable, and easiest way to (manually) create, (visually) inspect, or (manually) edit a table. In this format, the ending of a row is defined by the new-line character (a line on a text editor). So when you view it on a text editor, every row will occupy one line. The delimiters (or characters separating the columns) are white space characters (space, horizontal tab, vertical tab) and a comma (`,`). The only further requirement is that all rows/lines must have the same number of columns.

The columns don't have to be exactly under each other and the rows can be arbitrarily long with different lengths. For example the following contents in a file would be interpreted as a table with 4 columns and 2 rows, with each element interpreted as a `double` type (see Section 4.5 [Numeric data types], page 115).

```
1     2.234948   128   39.8923e8
2 , 4.454        792     72.98348e7
```

However, the example above has no other information about the columns (it is just raw data, with no meta-data). To use this table, you have to remember what the numbers in each column represent. Also, when you want to select columns, you have to count their position within the table. This can become frustrating and prone to bad errors (getting the columns wrong) especially as the number of columns increase. It is also bad for sending to a colleague, because they will find it hard to remember/use the columns properly.

To solve these problems in Gnuastro's programs/libraries you aren't limited to using the column's number, see Section 4.6.3 [Selecting table columns], page 121. If the columns have names, units, or comments you can also select your columns based on searches/matches in these fields, for example see Section 5.3 [Table], page 147. Also, in this manner, you can't guide the program reading the table on how to read the numbers. As an example, the first and third columns above can be read as integer types: the first column might be an ID and the third can be the number of pixels an object occupies in an image. So there is no need to read these to columns as a `double` type (which takes more memory, and is slower).

In the bare-minimum example above, you also can't use strings of characters, for example the names of filters, or some other identifier that includes non-numerical characters. In the absence of any information, only numbers can be read robustly. Assuming we read columns with non-numerical characters as string, there would still be the problem that the strings might contain space (or any delimiter) character for some rows. So, each 'word' in the string will be interpreted as a column and the program will abort with an error that the rows don't have the same number of columns.

To correct for these limitations, Gnuastro defines the following convention for storing the table meta-data along with the raw data in one plain text file. The format is primarily designed for ease of reading/writing by eye/fingers, but is also structured enough to be read by a program.

When the first non-white character in a line is `#`, or there are no non-white characters in it, then the line will not be considered as a row of data in the table (this is a pretty standard convention in many programs, and higher level languages). In the former case, the line is interpreted as a *comment*. If the comment line starts with '`# Column N:`', then it is assumed to contain information about column `N` (a number, counting from 1). Comment lines that don't start with this pattern are ignored and you can use them to include any further information you want to store with the table in the text file. A column information comment is assumed to have the following format:

```
# Column N: NAME [UNIT, TYPE, BLANK] COMMENT
```

Any sequence of characters between '`:`' and '`[`' will be interpreted as the column name (so it can contain anything except the '`[`' character). Anything between the '`]`' and the end of the line is defined as a comment. Within the brackets, anything before the first '`,`' is the units (physical units, for example km/s, or erg/s), anything before the second '`,`' is the short type identifier (see below, and Section 4.5 [Numeric data types], page 115). Finally (still within the brackets), any non-white characters after the second '`,`' are interpreted as the blank value for that column (see Section 6.1.3 [Blank pixels], page 154). Note that blank values will be stored in the same type as the column, not as a string[11].

When a formatting problem occurs (for example you have specified the wrong type code, see below), or the the column was already given meta-data in a previous comment, or the column number is larger than the actual number of columns in the table (the non-commented or empty lines), then the comment information line will be ignored.

When a comment information line can be used, the leading and trailing white space characters will be stripped from all of the elements. For example in this line:

```
# Column 5:  column name   [km/s,    f32,-99] Redshift as speed
```

The `NAME` field will be '`column name`' and the `TYPE` field will be '`f32`'. Note how all the white space characters before and after strings are not used, but those in the middle remained. Also, white space characters aren't mandatory. Hence, in the example above, the `BLANK` field will be given the value of '`-99`'.

Except for the column number (`N`), the rest of the fields are optional. Also, the column information comments don't have to be in order. In other words, the information for column $N + m$ ($m > 0$) can be given in a line before column N. Also, you don't have to

[11] For floating point types, the `nan`, or `inf` strings (both not case-sensitive) refer to IEEE NaN (not a number) and infinity values respectively and will be stored as a floating point, so they are acceptable.

specify information for all columns. Those columns that don't have this information will be interpreted with the default settings (like the case above: values are double precision floating point, and the column has no name, unit, or comment). So these lines are all acceptable for any table (the first one, with nothing but the column number is redundant):

```
# Column 5:
# Column 1: ID [,i] The Clump ID.
# Column 3: mag_f160w [AB mag, f] Magnitude from the F160W filter
```

The data type of the column should be specified with one of the following values:

- For a numeric column, you can use any of the numeric types (and their recognized identifiers) described in Section 4.5 [Numeric data types], page 115.
- '`strN`': for strings. The `N` value identifies the length of the string (how many characters it has). The start of the string on each row is the first non-delimiter character of the column that has the string type. The next `N` characters will be interpreted as a string and all leading and trailing white space will be removed.

  If the next column's characters, are closer than `N` characters to the start of the string column in that line/row, they will be considered part of the string column. If there is a new-line character before the ending of the space given to the string column (in other words, the string column is the last column), then reading of the string will stop, even if the `N` characters are not complete yet. See `tests/table/table.txt` for one example. Therefore, the only time you have to pay attention to the positioning and spaces given to the string column is when it is not the last column in the table.

  The only limitation in this format is that trailing and leading white space characters will be removed from the columns that are read. In most cases, this is the desired behavior, but if trailing and leading white-spaces are critically important to your analysis, define your own starting and ending characters and remove them after the table has been read. For example in the sample table below, the two '`|`' characters (which are arbitrary) will remain in the value of the second column and you can remove them manually later. If only one of the leading or trailing white spaces is important for your work, you can only use one of the '`|`'s.

  ```
  # Column 1: ID [label, uc]
  # Column 2: Notes [no unit, str50]
  1    leading and trailing white space is ignored here    2.3442e10
  2   |         but they will be preserved here        |   8.2964e11
  ```

Note that the FITS binary table standard does not define the `unsigned int` and `unsigned long` types, so if you want to convert your tables to FITS binary tables, use other types. Also, note that in the FITS ASCII table, there is only one integer type (`long`). So if you convert a Gnuastro plain text table to a FITS ASCII table with the Section 5.3 [Table], page 147, program, the type information for integers will be lost. Conversely if integer types are important for you, you have to manually set them when reading a FITS ASCII table (for example with the Table program when reading/converting into a file, or with the `gnuastro/table.h` library functions when reading into memory).

4.6.3 Selecting table columns

At the lowest level, the only defining aspect of a column in a table is its number, or position. But selecting columns purely by number is not very convenient and, especially when the

tables are large it can be very frustrating and prone to errors. Hence, table file formats (for example see Section 4.6.1 [Recognized table formats], page 117) have ways to store additional information about the columns (meta-data). Some of the most common pieces of information about each column are its *name*, the *units* of data in the it, and a *comment* for longer/informal description of the column's data.

To facilitate research with Gnuastro, you can select columns by matching, or searching in these three fields, besides the low-level column number. To view the full list of information on the columns in the table, you can use the Table program (see Section 5.3 [Table], page 147) with the command below (replace `table-file` with the filename of your table, if its FITS, you might also need to specify the HDU/extension which contains the table):

```
$ asttable --information table-file
```

Gnuastro's programs need the columns for different purposes, for example in Crop, you specify the columns containing the central coordinates of the crop centers with the `--coordcol` option (see Section 6.1.4.1 [Crop options], page 156). On the other hand, in MakeProfiles, to specify the column containing the profile position angles, you must use the `--pcol` option (see Section 8.1.5.1 [MakeProfiles catalog], page 292). Thus, there can be no unified common option name to select columns for all programs (different columns have different purposes). However, when the program expects a column for a specific context, the option names end in the `col` suffix like the examples above. These options accept values in integer (column number), or string (metadata match/search) format.

If the value can be parsed as a positive integer, it will be seen as the low-level column number. Note that column counting starts from 1, so if you ask for column 0, the respective program will abort with an error. When the value can't be interpreted as an a integer number, it will be seen as a string of characters which will be used to match/search in the table's meta-data. The meta-data field which the value will be compared with can be selected through the `--searchin` option, see Section 4.1.2.1 [Input/Output options], page 95. `--searchin` can take three values: `name`, `unit`, `comment`. The matching will be done following this convention:

- If the value is enclosed in two slashes (for example `-x/RA_/`, or `--coordcol=/RA_/`, see Section 6.1.4.1 [Crop options], page 156), then it is assumed to be a regular expression with the same convention as GNU AWK. GNU AWK has a very well written chapter (`https://www.gnu.org/software/gawk/manual/html_node/Regexp.html`) describing regular expressions, so we we will not continue discussing them here. Regular expressions are a very powerful tool in matching text and useful in many contexts. We thus strongly encourage reviewing this chapter for greatly improving the quality of your work in many cases, not just for searching column meta-data in Gnuastro.
- When the string isn't enclosed between '`/`'s, any column that exactly matches the given value in the given field will be selected.

Note that in both cases, you can ignore the case of alphabetic characters with the `--ignorecase` option, see Section 4.1.2.1 [Input/Output options], page 95. Also, in both cases, multiple columns may be selected with one call to this function. In this case, the order of the selected columns (with one call) will be the same order as they appear in the table.

4.7 Tessellation

It is sometimes necessary to classify the elements in a dataset (for example pixels in an image) into a grid of individual, non-overlapping tiles. For example when background sky gradients are present in an image, you can define a tile grid over the image. When the tile sizes are set properly, the background's variation over each tile will be negligible, allowing you to measure (and subtract) it. In other cases (for example spatial domain convolution in Gnuastro, see Section 6.3 [Convolve], page 177), it might simply be for speed of processing: each tile can be processed independently on a separate CPU thread. In the arts and mathematics, this process is formally known as tessellation (`https://en.wikipedia.org/wiki/Tessellation`).

The size of the regular tiles (in units of data-elements, or pixels in an image) can be defined with the `--tilesize` option. It takes multiple numbers (separated by a comma) which will be the length along the respective dimension (in FORTRAN/FITS dimension order). Divisions are also acceptable, but must result in an integer. For example `--tilesize=30,40` can be used for an image (a 2D dataset). The regular tile size along the first FITS axis (horizontal when viewed in SAO ds9) will be 30 pixels and along the second it will be 40 pixels. Ideally, `--tilesize` should be selected such that all tiles in the image have exactly the same size. In other words, that the dataset length in each dimension is divisible by the tile size in that dimension.

However, this is not always possible: the dataset can be any size and every pixel in it is valuable. In such cases, Gnuastro will look at the significance of the remainder length, if it is not significant (for example one or two pixels), then it will just increase the size of the first tile in the respective dimension and allow the rest of the tiles to have the required size. When the remainder is significant (for example one pixel less than the size along that dimension), the remainder will be added to one regular tile's size and the large tile will be cut in half and put in the two ends of the grid/tessellation. In this way, all the tiles in the central regions of the dataset will have the regular tile sizes and the tiles on the edge will be slightly larger/smaller depending on the remainder significance. The fraction which defines the remainder significance along all dimensions can be set through `--remainderfrac`.

The best tile size is directly related to the spatial properties of the property you want to study (for example, gradient on the image). In practice we assume that the gradient is not present over each tile. So if there is a strong gradient (for example in long wavelength ground based images) or the image is of a crowded area where there isn't too much blank area, you have to choose a smaller tile size. A larger mesh will give more pixels and and so the scatter in the results will be less (better statistics).

For raw image processing, a single tessellation/grid is not sufficient. Raw images are the unprocessed outputs of the camera detectors. Modern detectors usually have multiple readout channels each with its own amplifier. For example the Hubble Space Telescope Advanced Camera for Surveys (ACS) has four amplifiers over its full detector area dividing the square field of view to four smaller squares. Ground based image detectors are not exempt, for example each CCD of Subaru Telescope's Hyper Suprime-Cam camera (which has 104 CCDs) has four amplifiers, but they have the same height of the CCD and divide the width by four parts.

The bias current on each amplifier is different, and initial bias subtraction is not perfect. So even after subtracting the measured bias current, you can usually still identify the

boundaries of different amplifiers by eye. See Figure 11(a) in Akhlaghi and Ichikawa (2015) for an example. This results in the final reduced data to have non-uniform amplifier-shaped regions with higher or lower background flux values. Such systematic biases will then propagate to all subsequent measurements we do on the data (for example photometry and subsequent stellar mass and star formation rate measurements in the case of galaxies).

Therefore an accurate analysis requires a two layer tessellation: the top layer contains larger tiles, each covering one amplifier channel. For clarity we'll call these larger tiles "channels". The number of channels along each dimension is defined through the `--numchannels`. Each channel is then covered by its own individual smaller tessellation (with tile sizes determined by the `--tilesize` option). This will allow independent analysis of two adjacent pixels from different channels if necessary. If the image is processed or the detector only has one amplifier, you can set the number of channels in both dimension to 1.

The final tessellation can be inspected on the image with the `--checktiles` option that is available to all programs which use tessellation for localized operations. When this option is called, a FITS file with a `_tiled.fits` suffix will be created along with the outputs, see Section 4.8 [Automatic output], page 124. Each pixel in this image has the number of the tile that covers it. If the number of channels in any dimension are larger than unity, you will notice that the tile IDs are defined such that the first channels is covered first, then the second and so on. For the full list of processing-related common options (including tessellation options), please see Section 4.1.2.2 [Processing options], page 98.

4.8 Automatic output

All the programs in Gnuastro are designed such that specifying an output file or directory (based on the program context) is optional. When no output name is explicitly given (with `--output`, see Section 4.1.2.1 [Input/Output options], page 95), the programs will automatically set an output name based on the input name(s) and what the program does. For example when you are using ConvertType to save FITS image named `dataset.fits` to a JPEG image and don't specify a name for it, the JPEG output file will be name `dataset.jpg`. When the input is from the standard input (for example a pipe, see Section 4.1.3 [Standard input], page 104), and `--output` isn't given, the output name will be the program's name (for example `converttype.jpg`).

Another very important part of the automatic output generation is that all the directory information of the input file name is stripped off of it. This feature can be disabled with the `--keepinputdir` option, see Section 4.1.2.1 [Input/Output options], page 95. It is the default because astronomical data are usually very large and organized specially with special file names. In some cases, the user might not have write permissions in those directories[12].

Let's assume that we are working on a report and want to process the FITS images from two projects (ABC and DEF), which are stored in the sub-directories named `ABCproject/` and `DEFproject/` of our top data directory (`/mnt/data`). The following shell commands show how one image from the former is first converted to a JPEG image through ConvertType and then the objects from an image in the latter project are detected using NoiseChisel. The text after the `#` sign are comments (not typed!).

```
$ pwd                                                  # Current location
```

[12] In fact, even if the data is stored on your own computer, it is advised to only grant write permissions to the super user or root. This way, you won't accidentally delete or modify your valuable data!

```
/home/usrname/research/report
$ ls                                          # List directory contents
ABC01.jpg
$ ls /mnt/data/ABCproject                                  # Archive 1
ABC01.fits ABC02.fits ABC03.fits
$ ls /mnt/data/DEFproject                                  # Archive 2
DEF01.fits DEF02.fits DEF03.fits
$ astconvertt /mnt/data/ABCproject/ABC02.fits --output=jpg    # Prog 1
$ ls
ABC01.jpg ABC02.jpg
$ astnoisechisel /mnt/data/DEFproject/DEF01.fits              # Prog 2
$ ls
ABC01.jpg ABC02.jpg DEF01_detected.fits
```

4.9 Output FITS files

The output of many of Gnuastro's programs are (or can be) FITS files. The FITS format has many useful features for storing scientific datasets (cubes, images and tables) along with a robust features for archivability. For more on this standard, please see Section 5.1 [Fits], page 128.

As a community convention described in Section 5.1 [Fits], page 128, the first extension of all FITS files produced by Gnuastro's programs only contains the meta-data that is intended for the file's extension(s). For a Gnuastro program, this generic meta-data (that is stored as FITS keyword records) is its configuration when it produced this dataset: file name(s) of input(s) and option names, values and comments. Note that when the configuration is too trivial (only input filename, for example the program Section 5.3 [Table], page 147) no meta-data is written in this extension.

FITS keywords have the following limitations in regards to generic option names and values which are described below:

- If a keyword (option name) is longer than 8 characters, the first word in the record (80 character line) is `HIERARCH` which is followed by the keyword name.
- Values can be at most 75 characters, but for strings, this changes to 73 (because of the two extra `'` characters that are necessary). However, if the value is a file name, containing slash (`/`) characters to separate directories, Gnuastro will break the value into multiple keywords.
- Keyword names ignore case, therefore they are all in capital letters. Therefore, if you want to use Grep to inspect these keywords, use the `-i` option, like the example below.

```
$ astfits image_detected.fits -h0 | grep -i snquant
```

The keywords above are classified (separated by an empty line and title) as a group titled "ProgramName configuration". This meta-data extension, as well as all the other extensions (which contain data), also contain have final group of keywords to keep the basic date and version information of Gnuastro, its dependencies and the pipeline that is using Gnuastro (if its under version control).

`DATE` The creation time of the FITS file. This date is written directly by CFITSIO and is in UT format.

`COMMIT` Git's commit description from the running directory of Gnuastro's programs. If the running directory is not version controlled or `libgit2` isn't installed (see Section 3.1.2 [Optional dependencies], page 64) then this keyword will not be present. The printed value is equivalent to the output of the following command:

```
git describe --dirty --always
```

If the running directory contains non-committed work, then the stored value will have a '`-dirty`' suffix. This can be very helpful to let you know that the data is not ready to be shared with collaborators or submitted to a journal. You should only share results that are produced after all your work is committed (safely stored in the version controlled history and thus reproducible).

At first sight, version control appears to be mainly a tool for software developers. However progress in a scientific research is almost identical to progress in software development: first you have a rough idea that starts with handful of easy steps. But as the first results appear to be promising, you will have to extend, or generalize, it to make it more robust and work in all the situations your research covers, not just your first test samples. Slowly you will find wrong assumptions or bad implementations that need to be fixed ('bugs' in software development parlance). Finally, when you submit the research to your collaborators or a journal, many comments and suggestions will come in, and you have to address them.

Software developers have created version control systems precisely for this kind of activity. Each significant moment in the project's history is called a "commit", see Section 3.2.2 [Version controlled source], page 72. A snapshot of the project in each "commit" is safely stored away, so you can revert back to it at a later time, or check changes/progress. This way, you can be sure that your work is reproducible and track the progress and history. With version control, experimentation in the project's analysis is greatly facilitated, since you can easily revert back if a brainstorm test procedure fails.

One important feature of version control is that the research result (FITS image, table, report or paper) can be stamped with the unique commit information that produced it. This information will enable you to exactly reproduce that same result later, even if you have made changes/progress. For one example of a research paper's reproduction pipeline, please see the reproduction pipeline (`https://gitlab.com/makhlaghi/NoiseChisel-paper`) of the paper (`https://arxiv.org/abs/1505.01664`) describing Section 7.2 [NoiseChisel], page 225.

`CFITSIO` The version of CFITSIO used (see Section 3.1.1.2 [CFITSIO], page 62).

`WCSLIB` The version of WCSLIB used (see Section 3.1.1.3 [WCSLIB], page 63). Note that older versions of WCSLIB do not report the version internally. So this is only available if you are using more recent WCSLIB versions.

`GSL` The version of GNU Scientific Library that was used, see Section 3.1.1.1 [GNU Scientific library], page 62.

`GNUASTRO` The version of Gnuastro used (see Section 1.5 [Version numbering], page 7).

Here is one example of the last few lines of an example output.

```
                / Versions and date
DATE    = '...'                 / file creation date
COMMIT  = 'v0-8-g547f6eb'       / Commit description in running dir.
CFITSIO = '3.45    '            / CFITSIO version.
WCSLIB  = '5.19    '            / WCSLIB version.
GSL     = '2.5     '            / GNU Scientific Library version.
GNUASTRO= '0.7     '            / GNU Astronomy Utilities version.
END
```

5 Data containers

The most low-level and basic property of a dataset is how it is stored. To process, archive and transmit the data, you need a container to store it first. From the start of the computer age, different formats have been defined to store data, optimized for particular applications. One format/container can never be useful for all applications: the storage defines the application and vice-versa. In astronomy, the Flexible Image Transport System (FITS) standard has become the most common format of data storage and transmission. It has many useful features, for example multiple sub-containers (also known as extensions or header data units, HDUs) within one file, or support for tables as well as images. Each HDU can store an independent dataset and its corresponding meta-data. Therefore, Gnuastro has one program (see Section 5.1 [Fits], page 128) specifically designed to manipulate FITS HDUs and the meta-data (header keywords) in each HDU.

Your astronomical research does not just involve data analysis (where the FITS format is very useful). For example you want to demonstrate your raw and processed FITS images or spectra as figures within slides, reports, or papers. The FITS format is not defined for such applications. Thus, Gnuastro also comes with the ConvertType program (see Section 5.2 [ConvertType], page 138) which can be used to convert a FITS image to and from (where possible) other formats like plain text and JPEG (which allow two way conversion), along with EPS and PDF (which can only be created from FITS, not the other way round).

Finally, the FITS format is not just for images, it can also store tables. Binary tables in particular can be very efficient in storing catalogs that have more than a few tens of columns and rows. However, unlike images (where all elements/pixels have one data type), tables contain multiple columns and each column can have different properties: independent data types (see Section 4.5 [Numeric data types], page 115) and meta-data. In practice, each column can be viewed as a separate container that is grouped with others in the table. The only shared property of the columns in a table is thus the number of elements they contain. To allow easy inspection/manipulation of table columns, Gnuastro has the Table program (see Section 5.3 [Table], page 147). It can be used to select certain table columns in a FITS table and see them as a human readable output on the command-line, or to save them into another plain text or FITS table.

5.1 Fits

The "Flexible Image Transport System", or FITS, is by far the most common data container format in astronomy and in constant use since the 1970s. Archiving (future usage, simplicity) has been one of the primary design principles of this format. In the last few decades it has proved so useful and robust that the Vatican Library has also chosen FITS for its "long-term digital preservation" project[1].

Although the full name of the standard invokes the idea that it is only for images, it also contains complete and robust features for tables. It started off in the 1970s and was formally published as a standard in 1981, it was adopted by the International Astronomical Union (IAU) in 1982 and an IAU working group to maintain its future was defined in 1988. The FITS 2.0 and 3.0 standards were approved in 2000 and 2008 respectively, and the 4.0 draft has also been released recently, please see the FITS standard document webpage (`https://`

[1] `https://www.vaticanlibrary.va/home.php?pag=progettodigit`

`fits.gsfc.nasa.gov/fits_standard.html`) for the full text of all versions. Also see the FITS 3.0 standard paper (`https://doi.org/10.1051/0004-6361/201015362`) for a nice introduction and history along with the full standard.

Many common image formats, for example a JPEG, only have one image/dataset per file, however one great advantage of the FITS standard is that it allows you to keep multiple datasets (images or tables along with their separate meta-data) in one file. In the FITS standard, each data + metadata is known as an extension, or more formally a header data unit or HDU. The HDUs in a file can be completely independent: you can have multiple images of different dimensions/sizes or tables as separate extensions in one file. However, while the standard doesn't impose any constraints on the relation between the datasets, it is strongly encouraged to group data that are contextually related with each other in one file. For example an image and the table/catalog of objects and their measured properties in that image. Other examples can be images of one patch of sky in different colors (filters), or one raw telescope image along with its calibration data (tables or images).

As discussed above, the extensions in a FITS file can be completely independent. To keep some information (meta-data) about the group of extensions in the FITS file, the community has adopted the following convention: put no data in the first extension, so it is just meta-data. This extension can thus be used to store Meta-data regarding the whole file (grouping of extensions). Subsequent extensions may contain data along with their own separate meta-data. All of Gnuastro's programs also follow this convention: the main output dataset(s) are placed in the second (or later) extension(s). The first extension contains no data the program's configuration (input file name, along with all its option values) are stored as its meta-data, see Section 4.9 [Output FITS files], page 125.

The meta-data contain information about the data, for example which region of the sky an image corresponds to, the units of the data, what telescope, camera, and filter the data were taken with, it observation date, or the software that produced it and its configuration. Without the meta-data, the raw dataset is practically just a collection of numbers and really hard to understand, or connect with the real world (other datasets). It is thus strongly encouraged to supplement your data (at any level of processing) with as much meta-data about your processing/science as possible.

The meta-data of a FITS file is in ASCII format, which can be easily viewed or edited with a text editor or on the command-line. Each meta-data element (known as a keyword generally) is composed of a name, value, units and comments (the last two are optional). For example below you can see three FITS meta-data keywords for specifying the world coordinate system (WCS, or its location in the sky) of a dataset:

```
LATPOLE =           -27.805089 / [deg] Native latitude of celestial pole
RADESYS = 'FK5'                / Equatorial coordinate system
EQUINOX =               2000.0 / [yr] Equinox of equatorial coordinates
```

However, there are some limitations which discourage viewing/editing the keywords with text editors. For example there is a fixed length of 80 characters for each keyword (its name, value, units and comments) and there are no new-line characters, so on a text editor all the keywords are seen in one line. Also, the meta-data keywords are immediately followed by the data which are commonly in binary format and will show up as strange looking characters on a text editor, and significantly slowing down the processor.

Gnuastro's Fits program was designed to allow easy manipulation of FITS extensions and meta-data keywords on the command-line while conforming fully with the FITS standard. For example you can copy or cut (copy and remove) HDUs/extensions from one FITS file to another, or completely delete them. It also has features to delete, add, or edit meta-data keywords within one HDU.

5.1.1 Invoking Fits

Fits can print or manipulate the FITS file HDUs (extensions), meta-data keywords in a given HDU. The executable name is `astfits` with the following general template

```
$ astfits [OPTION...] ASTRdata
```

One line examples:

```
## View general information about every extension:
$ astfits image.fits

## Print the header keywords in the second HDU (counting from 0):
$ astfits image.fits -h1

## Only print header keywords that contain 'NAXIS':
$ astfits image.fits -h1 | grep NAXIS

## Only print the WCS standard PC matrix elements
$ astfits image.fits -h1 | grep 'PC._.'

## Copy a HDU from input.fits to out.fits:
$ astfits input.fits --copy=hdu-name --output=out.fits

## Update the OLDKEY keyword value to 153.034:
$ astfits --update=OLDKEY,153.034,"Old keyword comment"

## Delete one COMMENT keyword and add a new one:
$ astfits --delete=COMMENT --comment="Anything you like ;-)."

## Write two new keywords with different values and comments:
$ astfits --write=MYKEY1,20.00,"An example keyword" --write=MYKEY2,fd
```

When no action is requested (and only a file name is given), Fits will print a list of information about the extension(s) in the file. This information includes the HDU number, HDU name (`EXTNAME` keyword), type of data (see Section 4.5 [Numeric data types], page 115, and the number of data elements it contains (size along each dimension for images and table rows and columns). You can use this to get a general idea of the contents of the FITS file and what HDU to use for further processing, either with the Fits program or any other Gnuastro program.

Here is one example of information about a FITS file with four extensions: the first extension has no data, it is a purely meta-data HDU (commonly used to keep meta-data about the whole file, or grouping of extensions, see Section 5.1 [Fits], page 128). The second extension is an image with name `IMAGE` and single precision floating point type (`float32`, see Section 4.5 [Numeric data types], page 115), it has 4287 pixels along its first (horizontal)

axis and 4286 pixels along its second (vertical) axis. The third extension is also an image with name `MASK`. It is in 2-byte integer format (`int16`) which is commonly used to keep information about pixels (for example to identify which ones were saturated, or which ones had cosmic rays and so on), note how it has the same size as the `IMAGE` extension. The third extension is a binary table called `CATALOG` which has 12371 rows and 5 columns (it probably contains information about the sources in the image).

```
GNU Astronomy Utilities X.X
Run on Day Month DD HH:MM:SS YYYY
-----
HDU (extension) information: 'image.fits'.
 Column 1: Index (counting from 0).
 Column 2: Name ('EXTNAME' in FITS standard).
 Column 3: Image data type or 'table' format (ASCII or binary).
 Column 4: Size of data in HDU.
-----
0      n/a             uint8           0
1      IMAGE           float32         4287x4286
2      MASK            int16           4287x4286
3      CATALOG         table_binary    12371x5
```

If a specific HDU is identified on the command-line with the `--hdu` (or `-h` option) and no operation requested, then the full list of header keywords in that HDU will be printed (as if the `--printallkeys` was called, see below). It is important to remember that this only occurs when `--hdu` is given on the command-line. The `--hdu` value given in a configuration file will only be used when a specific operation on keywords requested. Therefore as described in the paragraphs above, when no explicit call to the `--hdu` option is made on the command-line and no operation is requested (on the command-line or configuration files), the basic information of each HDU/extension is printed.

The operating mode and input/output options to Fits are similar to the other programs and fully described in Section 4.1.2 [Common options], page 95. The options particular to Fits can be divided into two groups: 1) those related to modifying HDUs or extensions (see Section 5.1.1.1 [HDU manipulation], page 131), and 2) those related to viewing/modifying meta-data keywords (see Section 5.1.1.2 [Keyword manipulation], page 132). These two classes of options cannot be called together in one run: you can either work on the extensions or meta-data keywords in any instance of Fits.

5.1.1.1 HDU manipulation

Each header data unit, or HDU (also known as an extension), in a FITS file is an independent dataset (data + meta-data). Multiple HDUs can be stored in one FITS file, see Section 5.1 [Fits], page 128. The HDU modifying options to the Fits program are listed below.

These options may be called multiple times in one run. If so, the extensions will be copied from the input FITS file to the output FITS file in the given order (on the command-line and also in configuration files, see Section 4.2.2 [Configuration file precedence], page 107). If the separate classes are called together in one run of Fits, then first `--copy` is run (on all specified HDUs), followed by `--cut` (again on all specified HDUs), and then `--remove` (on all specified HDUs).

The `--copy` and `--cut` options need an output FITS file (specified with the `--output` option). If the output file exists, then the specified HDU will be copied following the last extension of the output file (the existing HDUs in it will be untouched). Thus, after Fits finishes, the copied HDU will be the last HDU of the output file. If no output file name is given, then automatic output will be used to store the HDUs given to this option (see Section 4.8 [Automatic output], page 124).

`-n`
`--numhdus`
: Print the number of extensions/HDUs in the given file. Note that this option must be called alone and will only print a single number. It is thus useful in scripts, for example when you need to do check the number of extensions in a FITS file.

  For a complete list of basic meta-data on the extensions in a FITS file, don't use any of the options in this section or in Section 5.1.1.2 [Keyword manipulation], page 132. For more, see Section 5.1.1 [Invoking Fits], page 130.

`-C STR`
`--copy=STR`
: Copy the specified extension into the output file, see explanations above.

`-k STR`
`--cut=STR`
: Cut (copy to output, remove from input) the specified extension into the output file, see explanations above.

`-R STR`
`--remove=STR`
: Remove the specified HDU from the input file. From CFITSIO: "In the case of deleting the primary array (the first HDU in the file) then [it] will be replaced by a null primary array containing the minimum set of required keywords and no data.". So in practice, any existing data (array) and meta-data in the first extension will be removed, but the number of extensions in the file won't change. This is because of the unique position the first FITS extension has in the FITS standard (for example it cannot be used to store tables).

5.1.1.2 Keyword manipulation

The meta-data in each header data unit, or HDU (also known as extension, see Section 5.1 [Fits], page 128) is stored as "keyword"s. Each keyword consists of a name, value, unit, and comments. The Fits program (see Section 5.1 [Fits], page 128) options related to viewing and manipulating keywords in a FITS HDU are described below.

To see the full list of keywords in a FITS HDU, you can use the `--printallkeys` option. If any of the keywords are to be modified, the headers of the input file will be changed. If you want to keep the original FITS file or HDU, it is easiest to create a copy first and then run Fits on that. In the FITS standard, keywords are always uppercase. So case does not matter in the input or output keyword names you specify.

Most of the options can accept multiple instances in one command. For example you can add multiple keywords to delete by calling `--delete` multiple times, since repeated

keywords are allowed, you can even delete the same keyword multiple times. The action of such options will start from the top most keyword.

The precedence of operations are described below. Note that while the order within each class of actions is preserved, the order of individual actions is not. So irrespective of what order you called `--delete` and `--update`. First, all the delete operations are going to take effect then the update operations.

1. `--delete`
2. `--rename`
3. `--update`
4. `--write`
5. `--asis`
6. `--history`
7. `--comment`
8. `--date`
9. `--printallkeys`
10. `--verify`
11. `--copykeys`

All possible syntax errors will be reported before the keywords are actually written. FITS errors during any of these actions will be reported, but Fits won't stop until all the operations are complete. If `--quitonerror` is called, then Fits will immediately stop upon the first error.

If you want to inspect only a certain set of header keywords, it is easiest to pipe the output of the Fits program to GNU Grep. Grep is a very powerful and advanced tool to search strings which is precisely made for such situations. For example if you only want to check the size of an image FITS HDU, you can run:

```
$ astfits input.fits | grep NAXIS
```

FITS STANDARD KEYWORDS: Some header keywords are necessary for later operations on a FITS file, for example BITPIX or NAXIS, see the FITS standard for their full list. If you modify (for example remove or rename) such keywords, the FITS file extension might not be usable any more. Also be careful for the world coordinate system keywords, if you modify or change their values, any future world coordinate system (like RA and Dec) measurements on the image will also change.

The keyword related options to the Fits program are fully described below.

`-a STR`
`--asis=STR`

Write `STR` exactly into the FITS file header with no modifications. If it does not conform to the FITS standards, then it might cause trouble, so please be very careful with this option. If you want to define the keyword from scratch, it is best to use the `--write` option (see below) and let CFITSIO worry about the standards. The best way to use this option is when you want to add a keyword from one FITS file to another unchanged and untouched. Below is an example

of such a case that can be very useful sometimes (on the command-line or in scripts):

```
$ key=$(astfits firstimage.fits | grep KEYWORD)
$ astfits --asis="$key" secondimage.fits
```

In particular note the double quotation signs (`"`) around the reference to the `key` shell variable (`$key`), since FITS keywords usually have lots of space characters, if this variable is not quoted, the shell will only give the first word in the full keyword to this option, which will definitely be a non-standard FITS keyword and will make it hard to work on the file afterwords. See the "Quoting" section of the GNU Bash manual for more information if your keyword has the special characters `$`, `‘`, or `\`.

`-d STR`
`--delete=STR`

Delete one instance of the `STR` keyword from the FITS header. Multiple instances of `--delete` can be given (possibly even for the same keyword, when its repeated in the meta-data). All keywords given will be removed from the headers in the same given order. If the keyword doesn't exist, Fits will give a warning and return with a non-zero value, but will not stop. To stop as soon as an error occurs, run with `--quitonerror`.

`-r STR`
`--rename=STR`

Rename a keyword to a new value. `STR` contains both the existing and new names, which should be separated by either a comma (`,`) or a space character. Note that if you use a space character, you have to put the value to this option within double quotation marks (`"`) so the space character is not interpreted as an option separator. Multiple instances of `--rename` can be given in one command. The keywords will be renamed in the specified order. If the keyword doesn't exist, Fits will give a warning and return with a non-zero value, but will not stop. To stop as soon as an error occurs, run with `--quitonerror`.

`-u STR`
`--update=STR`

Update a keyword, its value, its comments and its units in the format described below. If there are multiple instances of the keyword in the header, they will be changed from top to bottom (with multiple `--update` options).

The format of the values to this option can best be specified with an example:

```
--update=KEYWORD,value,"comments for this keyword",unit
```

If there is a writing error, Fits will give a warning and return with a non-zero value, but will not stop. To stop as soon as an error occurs, run with `--quitonerror`.

The value can be any numerical or string value[2]. Other than the `KEYWORD`, all the other values are optional. To leave a given token empty, follow the preceding

[2] Some tricky situations arise with values like '`87095e5`', if this was intended to be a number it will be kept in the header as `8709500000` and there is no problem. But this can also be a shortened Git commit hash. In the latter case, it should be treated as a string and stored as it is written. Commit hashes are very important in keeping the history of a file during your research and such values might arise without you

comma (`,`) immediately with the next. If any space character is present around the commas, it will be considered part of the respective token. So if more than one token has space characters within it, the safest method to specify a value to this option is to put double quotation marks around each individual token that needs it. Note that without double quotation marks, space characters will be seen as option separators and can lead to undefined behavior.

`-w STR`
`--write=STR`

Write a keyword to the header. For the possible value input formats, comments and units for the keyword, see the `--update` option above. The special names (first string) below will cause a special behavior:

`/`

Write a "title" to the list of keywords. A title consists of one blank line and another which is blank for several spaces and starts with a slash (`/`). The second string given to this option is the "title" or string printed after the slash. For example with the command below you can add a "title" of 'My keywords' after the existing keywords and add the subsequent `K1` and `K2` keywords under it (note that keyword names are not case sensitive).

```
$ astfits test.fits -h1 --write=/,"My keywords" \
          --write=k1,1.23,"My first keyword"    \
          --write=k2,4.56,"My second keyword"
$ astfits test.fits -h1
[[[ ... truncated ... ]]]

                        / My keywords
K1      =                  1.23 / My first keyword
K2      =                  4.56 / My second keyword
END
```

Adding a "title" before each contextually separate group of header keywords greatly helps in readability and visual inspection of the keywords. So generally, when you want to add new FITS keywords, its good practice to also add a title before them.

The reason you need to use `/` as the keyword name for setting a title is that `/` is the first non-white character.

The title(s) is(are) written into the FITS with the same order that `--write` is called. Therefore in one run of the Fits program, you can specify many different titles (with their own keywords under them). For example the command below that builds on the previous example and adds another group of keywords named `A1` and `A2`.

```
$ astfits test.fits -h1 --write=/,"My keywords"   \
          --write=k1,1.23,"My first keyword"      \
```

noticing them in your reproduction pipeline. One solution is to use `git describe` instead of the short hash alone. A less recommended solution is to add a space after the commit hash and Fits will write the value as '`87095e5` ' in the header. If you later compare the strings on the shell, the space character will be ignored by the shell in the latter solution and there will be no problem.

```
--write=k2,4.56,"My second keyword"     \
--write=/,"My second group of keywords" \
--write=a1,7.89,"First keyword"         \
--write=a2,0.12,"Second keyword"
```

`checksum` When nothing is given afterwards, the header integrity keywords[3] `DATASUM` and `CHECKSUM` will be written/updated. They are calculated and written by CFITSIO. They thus comply with the FITS standard 4.0 that defines these keywords.

If a value is given (for example `--write=checksum,my-own-checksum,"my checksum"`), then CFITSIO won't be called to calculate these two keywords and the value (as well as possible comment and unit) will be written just like any other keyword. This is generally not recommended, but necessary in special circumstances (where the checksum needs to be manually updated for example).

`DATASUM` only depends on the data section of the HDU/extension, so it is not changed when you update the keywords. But `CHECKSUM` also depends on the header and will not be valid if you make any further changes to the header. This includes any further keyword modification options in the same call to the Fits program. Therefore it is recommended to write these keywords as the last keywords that are written/modified in the extension. You can use the `--verify` option (described below) to verify the values of these two keywords.

`datasum` Similar to `checksum`, but only write the `DATASUM` keyword (that doesn't depend on the header keywords, only the data).

`-H STR`
`--history STR`
Add a `HISTORY` keyword to the header with the given value. A new `HISTORY` keyword will be created for every instance of this option. If the string given to this option is longer than 70 characters, it will be separated into multiple keyword cards. If there is an error, Fits will give a warning and return with a non-zero value, but will not stop. To stop as soon as an error occurs, run with `--quitonerror`.

`-c STR`
`--comment STR`
Add a `COMMENT` keyword to the header with the given value. Similar to the explanation for `--history` above.

`-t`
`--date` Put the current date and time in the header. If the `DATE` keyword already exists in the header, it will be updated. If there is a writing error, Fits will give a warning and return with a non-zero value, but will not stop. To stop as soon as an error occurs, run with `--quitonerror`.

[3] Section 4.4.2.7 (page 15) of `https://fits.gsfc.nasa.gov/standard40/fits_standard40aa-le.pdf`

`-p`
`--printallkeys`
: Print all the keywords in the specified FITS extension (HDU) on the command-line. If this option is called along with any of the other keyword editing commands, as described above, all other editing commands take precedence to this. Therefore, it will print the final keywords after all the editing has been done.

`-v`
`--verify` Verify the `DATASUM` and `CHECKSUM` data integrity keywords of the FITS standard. See the description under the `checksum` (under `--write`, above) for more on these keywords.

This option will print `Verified` for both keywords if they can be verified. Otherwise, if they don't exist in the given HDU/extension, it will print `NOT-PRESENT`, and if they cannot be verified it will print `INCORRECT`. In the latter case (when the keyword values exist but can't be verified), the Fits program will also return with a failure.

By default this function will also print a short description of the `DATASUM` AND `CHECKSUM` keywords. You can suppress this extra information with `--quiet` option.

`--copykeys=INT:INT`
: Copy the input's keyword records in the given range (inclusive) to the output HDU (specified with the `--output` and `--outhdu` options, for the filename and HDU/extension respectively).

The given string to this option must be two integers separated by a colon (`:`). The first integer must be positive (counting of the keyword records starts from 1). The second integer may be negative (zero is not acceptable) or an integer larger than the first.

A negative second integer means counting from the end. So `-1` is the last copy-able keyword (not including the `END` keyword).

To see the header keywords of the input with a number before them, you can pipe the output of the FITS program (when it prints all the keywords in an extension) into the `cat` program like below:

```
$ astfits input.fits -h1 | cat -n
```

`--outhdu` The HDU/extension to write the output keywords of `--copykeys`.

`-Q`
`--quitonerror`
: Quit if any of the operations above are not successful. By default if an error occurs, Fits will warn the user of the faulty keyword and continue with the rest of actions.

`-s STR`
`--datetosec STR`
: Interpret the value of the given keyword in the FITS date format (most generally: `YYYY-MM-DDThh:mm:ss.ddd...`) and return the corresponding Unix epoch time (number of seconds that have passed since 00:00:00 Thursday, January 1st,

1970). The `Thh:mm:ss.ddd...` section (specifying the time of day), and also the `.ddd...` (specifying the fraction of a second) are optional. The value to this option must be the FITS keyword name that contains the requested date, for example `--datetosec=DATE-OBS`.

This option can also interpret the older FITS date format (`DD/MM/YYThh:mm:ss.ddd...`) where only two characters are given to the year. In this case (following the GNU C Library), this option will make the following assumption: values 68 to 99 correspond to the years 1969 to 1999, and values 0 to 68 as the years 2000 to 2068.

This is a very useful option for operations on the FITS date values, for example sorting FITS files by their dates, or finding the time difference between two FITS files. The advantage of working with the Unix epoch time is that you don't have to worry about calendar details (for example the number of days in different months, or leap years, and etc).

5.2 ConvertType

The FITS format used in astronomy was defined mainly for archiving, transmission, and processing. In other situations, the data might be useful in other formats. For example, when you are writing a paper or report, or if you are making slides for a talk, you can't use a FITS image. Other image formats should be used. In other cases you might want your pixel values in a table format as plain text for input to other programs that don't recognize FITS. ConvertType is created for such situations. The various types will increase with future updates and based on need.

The conversion is not only one way (from FITS to other formats), but two ways (except the EPS and PDF formats[4]). So you can also convert a JPEG image or text file into a FITS image. Basically, other than EPS/PDF, you can use any of the recognized formats as different color channel inputs to get any of the recognized outputs. So before explaining the options and arguments (in Section 5.2.3 [Invoking ConvertType], page 142), we'll start with a short description of the recognized files types in Section 5.2.1 [Recognized file formats], page 138, followed a short introduction to digital color in Section 5.2.2 [Color], page 141.

5.2.1 Recognized file formats

The various standards and the file name extensions recognized by ConvertType are listed below. Currently Gnuastro uses the file name's suffix to identify the format.

FITS or IMH

Astronomical data are commonly stored in the FITS format (or the older data IRAF `.imh` format), a list of file name suffixes which indicate that the file is in this format is given in Section 4.1.1.1 [Arguments], page 93.

Each image extension of a FITS file only has one value per pixel/element. Therefore, when used as input, each input FITS image contributes as one color channel. If you want multiple extensions in one FITS file for different color channels, you have to repeat the file name multiple times and use the `--hdu`, `--hdu2`, `--hdu3` or `--hdu4` options to specify the different extensions.

[4] Because EPS and PDF are vector, not raster/pixelated formats

JPEG The JPEG standard was created by the Joint photographic experts group. It is currently one of the most commonly used image formats. Its major advantage is the compression algorithm that is defined by the standard. Like the FITS standard, this is a raster graphics format, which means that it is pixelated.

A JPEG file can have 1 (for gray-scale), 3 (for RGB) and 4 (for CMYK) color channels. If you only want to convert one JPEG image into other formats, there is no problem, however, if you want to use it in combination with other input files, make sure that the final number of color channels does not exceed four. If it does, then ConvertType will abort and notify you.

The file name endings that are recognized as a JPEG file for input are: `.jpg`, `.JPG`, `.jpeg`, `.JPEG`, `.jpe`, `.jif`, `.jfif` and `.jfi`.

TIFF TIFF (or Tagged Image File Format) was originally designed as a common format for scanners in the early 90s and since then it has grown to become very general. In many aspects, the TIFF standard is similar to the FITS image standard: it can allow data of many types (see Section 4.5 [Numeric data types], page 115), and also allows multiple images to be stored in a single file (each image in the file is called a 'directory' in the TIFF standard). However, unlike FITS, it can only store images, it has no constructs for tables. Another (inconvenient) difference with the FITS standard is that keyword names are stored as numbers, not human-readable text.

However, outside of astronomy, because of its support of different numeric data types, many fields use TIFF images for accurate (for example 16-bit integer or floating point for example) imaging data.

Currently ConvertType can only read TIFF images, if you are interested in writing TIFF images, please get in touch with us.

EPS The Encapsulated PostScript (EPS) format is essentially a one page PostScript file which has a specified size. PostScript also includes non-image data, for example lines and texts. It is a fully functional programming language to describe a document. Therefore in ConvertType, EPS is only an output format and cannot be used as input. Contrary to the FITS or JPEG formats, PostScript is not a raster format, but is categorized as vector graphics.

The Portable Document Format (PDF) is currently the most common format for documents. Some believe that PDF has replaced PostScript and that PostScript is now obsolete. This view is wrong, a PostScript file is an actual plain text file that can be edited like any program source with any text editor. To be able to display its programmed content or print, it needs to pass through a processor or compiler. A PDF file can be thought of as the processed output of the compiler on an input PostScript file. PostScript, EPS and PDF were created and are registered by Adobe Systems.

With these features in mind, you can see that when you are compiling a document with TEX or LATEX, using an EPS file is much more low level than a JPEG and thus you have much greater control and therefore quality. Since it also includes vector graphic lines we also use such lines to make a thin border around the image to make its appearance in the document much better. No matter the resolution of the display or printer, these lines will always be clear

and not pixelated. In the future, addition of text might be included (for example labels or object IDs) on the EPS output. However, this can be done better with tools within TEX or LATEX such as PGF/Tikz[5].

If the final input image (possibly after all operations on the flux explained below) is a binary image or only has two colors of black and white (in segmentation maps for example), then PostScript has another great advantage compared to other formats. It allows for 1 bit pixels (pixels with a value of 0 or 1), this can decrease the output file size by 8 times. So if a gray-scale image is binary, ConvertType will exploit this property in the EPS and PDF (see below) outputs.

The standard formats for an EPS file are `.eps`, `.EPS`, `.epsf` and `.epsi`. The EPS outputs of ConvertType have the `.eps` suffix.

PDF As explained above, a PDF document is a static document description format, viewing its result is therefore much faster and more efficient than PostScript. To create a PDF output, ConvertType will make a PostScript page description and convert that to PDF using GPL Ghostscript. The suffixes recognized for a PDF file are: `.pdf`, `.PDF`. If GPL Ghostscript cannot be run on the PostScript file, it will remain and a warning will be printed.

`blank` This is not actually a file type! But can be used to fill one color channel with a blank value. If this argument is given for any color channel, that channel will not be used in the output.

Plain text Plain text files have the advantage that they can be viewed with any text editor or on the command-line. Most programs also support input as plain text files. As input, each plain text file is considered to contain one color channel.

In ConvertType, the recognized extensions for plain text files are `.txt` and `.dat`. As described in Section 5.2.3 [Invoking ConvertType], page 142, if you just give these extensions, (and not a full filename) as output, then automatic output will be preformed to determine the final output name (see Section 4.8 [Automatic output], page 124). Besides these, when the format of a file cannot be recognized from its name, ConvertType will fall back to plain text mode. So you can use any name (even without an extension) for a plain text input or output. Just note that when the suffix is not recognized, automatic output will not be preformed.

The basic input/output on plain text images is very similar to how tables are read/written as described in Section 4.6.2 [Gnuastro text table format], page 119. Simply put, the restrictions are very loose, and there is a convention to define a name, units, data type (see Section 4.5 [Numeric data types], page 115), and comments for the data in a commented line. The only difference is that as a table, a text file can contain many datasets (columns), but as a 2D image, it can only contain one dataset. As a result, only one information comment line is necessary for a 2D image, and instead of the starting '`# Column N`' (`N` is the column number), the information line for a 2D image must start

[5] `http://sourceforge.net/projects/pgf/`

with '`# Image 1`'. When ConvertType is asked to output to plain text file, this information comment line is written before the image pixel values.

When converting an image to plain text, consider the fact that if the image is large, the number of columns in each line will become very large, possibly making it very hard to open in some text editors.

Standard output (command-line)

This is very similar to the plain text output, but instead of creating a file to keep the printed values, they are printed on the command line. This can be very useful when you want to redirect the results directly to another program in one command with no intermediate file. The only difference is that only the pixel values are printed (with no information comment line). To print to the standard output, set the output name to '`stdout`'.

5.2.2 Color

Color is defined by mixing various measurements/filters. In digital monitors or common digital cameras, colors are displayed/stored by mixing the three basic colors of red, green and blue (RGB) with various proportions. When printing on paper, standard printers use the cyan, magenta, yellow and key (CMYK, key=black) color space. In other words, for each displayed/printed pixel of a color image, the dataset/image has three or four values.

To store/show the three values for each pixel, cameras and monitors allocate a certain fraction of each pixel's area to red, green and blue filters. These three filters are thus built into the hardware at the pixel level. However, because measurement accuracy is very important in scientific instruments, and we want to do measurements (take images) with various/custom filters (without having to order a new expensive detector!), scientific detectors use the full area of the pixel to store one value for it in a single/mono channel dataset. To make measurements in different filters, we just place a filter in the light path before the detector. Therefore, the FITS format that is used to store astronomical datasets is inherently a mono-channel format (see Section 5.2.1 [Recognized file formats], page 138, or Section 5.1 [Fits], page 128).

When a subject has been imaged in multiple filters, you can feed each different filter into the red, green and blue channels and obtain a colored visualization. In ConvertType, you can do this by giving each separate single-channel dataset (for example in the FITS image format) as an argument (in the proper order), then asking for the output in a format that supports multi-channel datasets (for example JPEG or PDF, see the examples in Section 5.2.3 [Invoking ConvertType], page 142).

As discussed above, color is not defined when a dataset/image contains a single value for each pixel. However, we interact with scientific datasets through monitors or printers (which allow multiple values per pixel and produce color with them). As a result, there is a lot of freedom in visualizing a single-channel dataset. The most basic is to use shades of black (because of its strong contrast with white). This scheme is called grayscale. To help in visualization, more complex mappings can be defined. For example, the values can be scaled to a range of 0 to 360 and used as the "Hue" term of the Hue-Saturation-Value (`https://en.wikipedia.org/wiki/HSL_and_HSV`) (HSV) color space (while fixing the "Saturation" and "Value" terms). In ConvertType, you can use the `--colormap` option to choose between different mappings of mono-channel inputs, see Section 5.2.3 [Invoking ConvertType], page 142.

Since grayscale is a commonly used mapping of single-valued datasets, we'll continue with a closer look at how it is stored. One way to represent a gray-scale image in different color spaces is to use the same proportions of the primary colors in each pixel. This is the common way most FITS image viewers work: for each pixel, they fill all the channels with the single value. While this is necessary for displaying a dataset, there are downsides when storing/saving this type of grayscale visualization (for example in a paper).

- Three (for RGB) or four (for CMYK) values have to be stored for every pixel, this makes the output file very heavy (in terms of bytes).
- If printing, the printing errors of each color channel can make the printed image slightly more blurred than it actually is.

To solve both these problems when storing grayscale visualization, the best way is to save a single-channel dataset into the black channel of the CMYK color space. The JPEG standard is the only common standard that accepts CMYK color space.

The JPEG and EPS standards set two sizes for the number of bits in each channel: 8-bit and 12-bit. The former is by far the most common and is what is used in ConvertType. Therefore, each channel should have values between 0 to $2^8 - 1 = 255$. From this we see how each pixel in a gray-scale image is one byte (8 bits) long, in an RGB image, it is 3 bytes long and in CMYK it is 4 bytes long. But thanks to the JPEG compression algorithms, when all the pixels of one channel have the same value, that channel is compressed to one pixel. Therefore a Grayscale image and a CMYK image that has only the K-channel filled are approximately the same file size.

5.2.3 Invoking ConvertType

ConvertType will convert any recognized input file type to any specified output type. The executable name is `astconvertt` with the following general template

```
$ astconvertt [OPTION...] InputFile [InputFile2] ... [InputFile4]
```

One line examples:

```
## Convert an image in FITS to PDF:
$ astconvertt image.fits --output=pdf

## Similar to before, but use the SLS color map:
$ astconvertt image.fits --colormap=sls --output=pdf

## Convert an image in JPEG to FITS (with multiple extensions
## if its color):
$ astconvertt image.jpg -oimage.fits

## Use three plain text 2D arrays to create an RGB JPEG output:
$ astconvertt f1.txt f2.txt f3.fits -o.jpg

## Use two images and one blank for an RGB EPS output:
$ astconvertt M31_r.fits M31_g.fits blank -oeps

## Directly pass input from output of another program through Standard
## input (not a file).
```

```
$ cat 2darray.txt | astconvertt -oimg.fits
```

The output's file format will be interpreted from the value given to the `--output` option. It can either be given on the command-line or in any of the configuration files (see Section 4.2 [Configuration files], page 106). Note that if the output suffix is not recognized, it will default to plain text format, see Section 5.2.1 [Recognized file formats], page 138.

At most four input files (one for each color channel for formats that allow it) are allowed in ConvertType. The first input dataset can either be a file or come from Standard input (see Section 4.1.3 [Standard input], page 104). The order of multiple input files is important. After reading the input file(s) the number of color channels in all the inputs will be used to define which color space to use for the outputs and how each color channel is interpreted.

Some formats can allow more than one color channel (for example in the JPEG format, see Section 5.2.1 [Recognized file formats], page 138). If there is one input dataset (color channel) the output will be gray-scale, if three input datasets (color channels) are given, they are respectively considered to be the red, green and blue color channels. Finally, if there are four color channels they will be be cyan, magenta, yellow and black (CMYK colors).

The value to `--output` (or `-o`) can be either a full file name or just the suffix of the desired output format. In the former case, it will used for the output. In the latter case, the name of the output file will be set based on the automatic output guidelines, see Section 4.8 [Automatic output], page 124. Note that the suffix name can optionally start a `.` (dot), so for example `--output=.jpg` and `--output=jpg` are equivalent. See Section 5.2.1 [Recognized file formats], page 138,

Besides the common set of options explained in Section 4.1.2 [Common options], page 95, the options to ConvertType can be classified into input, output and flux related options. The majority of the options are to do with the flux range. Astronomical data usually have a very large dynamic range (difference between maximum and minimum value) and different subjects might be better demonstrated with a limited flux range.

Input:

`-h STR/INT`
`--hdu=STR/INT`

In ConvertType, it is possible to call the HDU option multiple times for the different input FITS or TIFF files in the same order that they are called on the command-line. Note that in the TIFF standard, one 'directory' (similar to a FITS HDU) may contain multiple color channels (for example when the image is in RGB).

Except for the fact that multiple calls are possible, this option is identical to the common `--hdu` in Section 4.1.2.1 [Input/Output options], page 95. The number of calls to this option cannot be less than the number of input FITS or TIFF files, but if there are more, the extra HDUs will be ignored, note that they will be read in the order described in Section 4.2.2 [Configuration file precedence], page 107.

Unlike CFITSIO, libtiff (which is used to read TIFF files) only recognizes numbers (counting from zero, similar to CFITSIO) for 'directory' identification. Hence the concept of names is not defined for the directories and the values to this option for TIFF files must be numbers.

Output:

`-w FLT`
`--widthincm=FLT`
The width of the output in centimeters. This is only relevant for those formats that accept such a width (not plain text for example). For most digital purposes, the number of pixels is far more important than the value to this parameter because you can adjust the absolute width (in inches or centimeters) in your document preparation program.

`-b INT`
`--borderwidth=INT`
The width of the border to be put around the EPS and PDF outputs in units of PostScript points. There are 72 or 28.35 PostScript points in an inch or centimeter respectively. In other words, there are roughly 3 PostScript points in every millimeter. If you are planning on adding a border, its significance is highly correlated with the value you give to the `--widthincm` parameter.

Unfortunately in the document structuring convention of the PostScript language, the "bounding box" has to be in units of PostScript points with no fractions allowed. So the border values only have to be specified in integers. To have a final border that is thinner than one PostScript point in your document, you can ask for a larger width in ConvertType and then scale down the output EPS or PDF file in your document preparation program. For example by setting `width` in your `includegraphics` command in TEX or LATEX. Since it is vector graphics, the changes of size have no effect on the quality of your output quality (pixels don't get different values).

`-x`
`--hex` Use Hexadecimal encoding in creating EPS output. By default the ASCII85 encoding is used which provides a much better compression ratio. When converted to PDF (or included in TEX or LATEX which is finally saved as a PDF file), an efficient binary encoding is used which is far more efficient than both of them. The choice of EPS encoding will thus have no effect on the final PDF.

So if you want to transfer your EPS files (for example if you want to submit your paper to arXiv or journals in PostScript), their storage might become important if you have large images or lots of small ones. By default ASCII85 encoding is used which offers a much better compression ratio (nearly 40 percent) compared to Hexadecimal encoding.

`-u INT`
`--quality=INT`
The quality (compression) of the output JPEG file with values from 0 to 100 (inclusive). For other formats the value to this option is ignored. Note that only in gray-scale (when one input color channel is given) will this actually be the exact quality (each pixel will correspond to one input value). If it is in color mode, some degradation will occur. While the JPEG standard does support loss-less graphics, it is not commonly supported.

`--colormap=STR[,FLT,...]`
: The color map to visualize a single channel. The first value given to this option is the name of the color map, which is shown below. Some color maps can be configured. In this case, the configuration parameters are optionally given as numbers following the name of the color map for example see `hsv`. The table below contains the usable names of the color maps that are currently supported:

  `gray`
  `grey`
  : Grayscale color map. This color map doesn't have any parameters. The full dataset range will be scaled to 0 and $2^8 - 1 = 255$ to be stored in the requested format.

  `hsv`
  : Hue, Saturation, Value[6] color map. If no values are given after the name (`--colormap=hsv`), the dataset will be scaled to 0 and 360 for hue covering the full spectrum of colors. However, you can limit the range of hue (to show only a special color range) by explicitly requesting them after the name (for example `--colormap=hsv,20,240`).

    The mapping of a single-channel dataset to HSV is done through the Hue and Value elements: Lower dataset elements have lower "value" *and* lower "hue". This creates darker colors for fainter parts, while also respecting the range of colors.

  `sls`
  : The SLS color range, taken from the commonly used SAO DS9 (`http://ds9.si.edu`). The advantage of this color range is that it ranges from black to dark blue, and finishes with red and white. So unlike the HSV color range, it includes black and white and brighter colors (like yellow, red and white) show the larger values.

`--rgbtohsv`
: When there are three input channels and the output is in the FITS format, interpret the three input channels as red, green and blue channels (RGB) and convert them to the hue, saturation, value (HSV) color space.

  The currently supported output formats of ConvertType don't have native support for HSV. Therefore this option is only supported when the output is in FITS format and each of the hue, saturation and value arrays can be saved as one FITS extension in the output for further analysis (for example to select a certain color).

Flux range:

`-c STR`
`--change=STR`
: (`=STR`) Change pixel values with the following format `"from1:to1, from2:to2,..."`. This option is very useful in displaying labeled pixels (not actual data images which have noise) like segmentation maps. In labeled images, usually a group of pixels have a fixed integer value. With this option,

[6] `https://en.wikipedia.org/wiki/HSL_and_HSV`

you can manipulate the labels before the image is displayed to get a better output for print or to emphasize on a particular set of labels and ignore the rest. The labels in the images will be changed in the same order given. By default first the pixel values will be converted then the pixel values will be truncated (see `--fluxlow` and `--fluxhigh`).

You can use any number for the values irrespective of your final output, your given values are stored and used in the double precision floating point format. So for example if your input image has labels from 1 to 20000 and you only want to display those with labels 957 and 11342 then you can run ConvertType with these options:

```
$ astconvertt --change=957:50000,11342:50001 --fluxlow=5e4 \
   --fluxhigh=1e5 segmentationmap.fits --output=jpg
```

While the output JPEG format is only 8 bit, this operation is done in an intermediate step which is stored in double precision floating point. The pixel values are converted to 8-bit after all operations on the input fluxes have been complete. By placing the value in double quotes you can use as many spaces as you like for better readability.

`-C`
`--changeaftertrunc`
: Change pixel values (with `--change`) after truncation of the flux values, by default it is the opposite.

`-L FLT`
`--fluxlow=FLT`
: The minimum flux (pixel value) to display in the output image, any pixel value below this value will be set to this value in the output. If the value to this option is the same as `--fluxhigh`, then no flux truncation will be applied. Note that when multiple channels are given, this value is used for all the color channels.

`-H FLT`
`--fluxhigh=FLT`
: The maximum flux (pixel value) to display in the output image, see `--fluxlow`.

`-m INT`
`--maxbyte=INT`
: This is only used for the JPEG and EPS output formats which have an 8-bit space for each channel of each pixel. The maximum value in each pixel can therefore be $2^8 - 1 = 255$. With this option you can change (decrease) the maximum value. By doing so you will decrease the dynamic range. It can be useful if you plan to use those values for other purposes.

`-A INT`
`--forcemin=INT`
: Enforce the value of `--fluxlow` (when its given), even if its smaller than the minimum of the dataset and the output is format supporting color. This is particularly useful when you are converting a number of images to a common image format like JPEG or PDF with a single command and want them all to have the same range of colors, independent of the contents of the dataset.

Note that if the minimum value is smaller than `--fluxlow`, then this option is redundant.

By default, when the dataset only has two values, *and* the output format is PDF or EPS, ConvertType will use the PostScript optimization that allows setting the pixel values per bit, not byte (Section 5.2.1 [Recognized file formats], page 138). This can greatly help reduce the file size. However, when `--fluxlow` or `--fluxhigh` are called, this optimization is disabeled: even though there are only two values (is binary), the difference between them does not correspond to the full contrast of black and white.

`-B INT`
`--forcemax=INT`
Similar to `--forcemin`, but for the maximum.

`-i`
`--invert` For 8-bit output types (JPEG, EPS, and PDF for example) the final value that is stored is inverted so white becomes black and vice versa. The reason for this is that astronomical images usually have a very large area of blank sky in them. The result will be that a large are of the image will be black. Note that this behavior is ideal for gray-scale images, if you want a color image, the colors are going to be mixed up.

5.3 Table

Tables are the products of processing astronomical images and spectra. For example in Gnuastro, MakeCatalog will process the defined pixels over an object and produce a catalog (see Section 7.4 [MakeCatalog], page 255). For each identified object, MakeCatalog can print its position on the image or sky, its total brightness and many other information that is deducible from the given image. Each one of these properties is a column in its output catalog (or table) and for each input object, we have a row.

When there are only a small number of objects (rows) and not too many properties (columns), then a simple plain text file is mainly enough to store, transfer, or even use the produced data. However, to be more efficient in all these aspects, astronomers have defined the FITS binary table standard to store data in a binary (0 and 1) format, not plain text. This can offer major advantages in all those aspects: the file size will be greatly reduced and the reading and writing will be faster (because the RAM and CPU also work in binary).

The FITS standard also defines a standard for ASCII tables, where the data are stored in the human readable ASCII format, but within the FITS file structure. These are mainly useful for keeping ASCII data along with images and possibly binary data as multiple (conceptually related) extensions within a FITS file. The acceptable table formats are fully described in Section 4.6 [Tables], page 117.

Binary tables are not easily readable by human eyes. There is no fixed/unified standard on how the zero and ones should be interpreted. The Unix-like operating systems have flourished because of a simple fact: communication between the various tools is based on human readable characters[7]. So while the FITS table standards are very beneficial for the

[7] In "The art of Unix programming", Eric Raymond makes this suggestion to programmers: "When you feel the urge to design a complex binary file format, or a complex binary application protocol, it is

tools that recognize them, they are hard to use in the vast majority of available software. This creates limitations for their generic use.

'Table' is Gnuastro's solution to this problem. With Table, FITS tables (ASCII or binary) are directly accessible to the Unix-like operating systems power-users (those working the command-line or shell, see Section 1.6.1 [Command-line interface], page 9). With Table, a FITS table (in binary or ASCII formats) is only one command away from AWK (or any other tool you want to use). Just like a plain text file that you read with the `cat` command. You can pipe the output of Table into any other tool for higher-level processing, see the examples in Section 5.3.1 [Invoking Table], page 148, for some simple examples.

5.3.1 Invoking Table

Table will read/write, select, convert, or show the information of the columns in FITS ASCII table, FITS binary table and plain text table files, see Section 4.6 [Tables], page 117. Output columns can also be determined by number or regular expression matching of column names, units, or comments. The executable name is `asttable` with the following general template

```
$ asttable [OPTION...] InputFile
```

One line examples:

```
## Get the table column information (name, data type, or units):
$ asttable bintab.fits --information

## Print columns named RA and DEC, followed by all the columns where
## the name starts with "MAG_":
$ asttable bintab.fits --column=RA --column=DEC --column=/^MAG_/

## Similar to the above, but with one call to '--column' (or '-c'),
## also sort the rows by the input's photometric redshift ('Z_PHOT')
## column. To confirm the sort, you can add 'Z_PHOT' to the columns
## to print.
$ asttable bintab.fits -cRA,DEC,/^MAG_/ --sort=Z_PHOT

## Similar to the above, but only print rows that have a photometric
## redshift between 2 and 3.
$ asttable bintab.fits -cRA,DEC,/^MAG_/ --range=Z_PHOT,2:3

## Similar to above, but writes the columns to a file (with metadata)
## instead of the command-line.
$ asttable bintab.fits -cRA,DEC,/^MAG_/ --output=out.txt

## Only print the 2nd column, and the third column multiplied by 5:
$ asttable bintab.fits -c2,5 | awk '{print $1, 5*$2}'

## Only print rows with a value in the 10th column above 100000:
$ asttable bintab.fits | awk '$10>10e5 {print}'
```

generally wise to lie down until the feeling passes.". This is a great book and strongly recommended, give it a look if you want to truly enjoy your work/life in this environment.

```
## Sort the output columns by the third column, save output:
$ asttable bintab.fits | 'sort -nk3 > output.txt

## Subtract the first column from the second in 'cat.txt' and keep the
## third and fourth columns. Feed the columns to Table to write as a
## FITS table.
$ awk '!/^#/{print $2-$1, $3, $4}' cat.txt | asttable -ocat.fits

## Convert a plain text table to a binary FITS table:
$ asttable plaintext.txt --output=table.fits --tabletype=fits-binary
```

Table's input dataset can be given either as a file or from Standard input (see Section 4.1.3 [Standard input], page 104). In the absence of selected columns, all the input's columns and rows will be written to the output. If any output file is explicitly requested (with `--output`) the output table will be written in it. When no output file is explicitly requested the output table will be written to the standard output.

If the specified output is a FITS file, the type of FITS table (binary or ASCII) will be determined from the `--tabletype` option. If the output is not a FITS file, it will be printed as a plain text table (with space characters between the columns). When the columns are accompanied by meta-data (like column name, units, or comments), this information will also printed in the plain text file before the table, as described in Section 4.6.2 [Gnuastro text table format], page 119.

For the full list of options common to all Gnuastro programs please see Section 4.1.2 [Common options], page 95. Options can also be stored in directory, user or system-wide configuration files to avoid repeating on the command-line, see Section 4.2 [Configuration files], page 106. Table does not follow Automatic output that is common in most Gnuastro programs, see Section 4.8 [Automatic output], page 124. Thus, in the absence of an output file, the selected columns will be printed on the command-line with no column information, ready for redirecting to other tools like AWK or sort, similar to the examples above.

`-i`
`--information`

Only print the column information in the specified table on the command-line and exit. Each column's information (number, name, units, data type, and comments) will be printed as a row on the command-line. Note that the FITS standard only requires the data type (see Section 4.5 [Numeric data types], page 115), and in plain text tables, no meta-data/information is mandatory. Gnuastro has its own convention in the comments of a plain text table to store and transfer this information as described in Section 4.6.2 [Gnuastro text table format], page 119.

This option will take precedence over the `--column` option, so when it is called along with requested columns, the latter will be ignored. This can be useful if you forget the identifier of a column after you have already typed some on the command-line. You can simply add a `-i` and run Table to see the whole list and remember. Then you can use the shell history (with the up arrow key on the keyboard), and retrieve the last command with all the previously typed columns present, delete `-i` and add the identifier you had forgot.

`-c STR/INT`
`--column=STR/INT`
: Specify the columns to read, see Section 4.6.3 [Selecting table columns], page 121, for a thorough explanation on how the value to this option is interpreted. There are two ways to specify multiple columns: 1) multiple calls to this option, 2) using a comma between each column specifier in one call to this option. These different solutions may be mixed in one call to Table: for example, `-cRA,DEC -cMAG`, or `-cRA -cDEC -cMAG` are both equivalent to `-cRA -cDEC -cMAG`. The order of the output columns will be the same order given to the option or in the configuration files (see Section 4.2.2 [Configuration file precedence], page 107).

  This option is not mandatory, if no specific columns are requested, all the input table columns are output. When this option is called multiple times, it is possible to output one column more than once.

`-O`
`--colinfoinstdout`
: Add column metadata when the output is printed in the standard output. Usually the standard output is used for a fast visual check or to pipe into other program for further processing. So by default meta-data aren't included.

`-r STR,FLT:FLT`
`--range=STR,FLT:FLT`
: Only print the output rows that have a value within the given range in the `STR` column (can be a name or counter). For example with `--range=sn,5:20` the output's columns will only contain rows that have a value between 5 and 20 in the `sn` column (not case-sensitive).

  This option can be called multiple times (different ranges for different columns) in one run of the Table program. This is very useful for selecting the final rows from multiple criteria/columns.

  The chosen column doesn't have to be in the output columns. This is good when you just want to select using one column's values, but don't need that column anymore afterwards.

`-s STR`

`--sort=STR`
: Sort the output rows based on the values in the `STR` column (can be a column name or number). By default the sort is done in ascending/increasing order, to sort in a descending order, use `--descending`.

  The chosen column doesn't have to be in the output columns. This is good when you just want to sort using one column's values, but don't need that column anymore afterwards.

`-d`
`--descending`
: When called with `--sort`, rows will be sorted in descending order.

6 Data manipulation

Images are one of the major formats of data that is used in astronomy. The functions in this chapter explain the GNU Astronomy Utilities which are provided for their manipulation. For example cropping out a part of a larger image or convolving the image with a given kernel or applying a transformation to it.

6.1 Crop

Astronomical images are often very large, filled with thousands of galaxies. It often happens that you only want a section of the image, or you have a catalog of sources and you want to visually analyze them in small postage stamps. Crop is made to do all these things. When more than one crop is required, Crop will divide the crops between multiple threads to significantly reduce the run time.

Astronomical surveys are usually extremely large. So large in fact, that the whole survey will not fit into a reasonably sized file. Because of this, surveys usually cut the final image into separate tiles and store each tile in a file. For example the COSMOS survey's Hubble space telescope, ACS F814W image consists of 81 separate FITS images, with each one having a volume of 1.7 Giga bytes.

Even though the tile sizes are chosen to be large enough that too many galaxies/targets don't fall on the edges of the tiles, inevitably some do. So when you simply crop the image of such targets from one tile, you will miss a large area of the surrounding sky (which is essential in estimating the noise). Therefore in its WCS mode, Crop will stitch parts of the tiles that are relevant for a target (with the given width) from all the input images that cover that region into the output. Of course, the tiles have to be present in the list of input files.

Besides cropping postage stamps around certain coordinates, Crop can also crop arbitrary polygons from an image (or a set of tiles by stitching the relevant parts of different tiles within the polygon), see `--polygon` in Section 6.1.4 [Invoking Crop], page 155. Alternatively, it can crop out rectangular regions through the `--section` option from one image, see Section 6.1.2 [Crop section syntax], page 154.

6.1.1 Crop modes

In order to be comprehensive, intuitive, and easy to use, there are two ways to define the crop:

1. From its center and side length. For example if you already know the coordinates of an object and want to inspect it in an image or to generate postage stamps of a catalog containing many such coordinates.
2. The vertices of the crop region, this can be useful for larger crops over many targets, for example to crop out a uniformly deep, or contiguous, region of a large survey.

Irrespective of how the crop region is defined, the coordinates to define the crop can be in Image (pixel) or World Coordinate System (WCS) standards. All coordinates are read as floating point numbers (not integers, except for the `--section` option, see below). By setting the *mode* in Crop, you define the standard that the given coordinates must be interpreted. Here, the different ways to specify the crop region are discussed within each standard. For the full list options, please see Section 6.1.4 [Invoking Crop], page 155.

When the crop is defined by its center, the respective (integer) central pixel position will be found internally according to the FITS standard. To have this pixel positioned in the center of the cropped region, the final cropped region will have an add number of pixels (even if you give an even number to `--width` in image mode).

Furthermore, when the crop is defined as by its center, Crop allows you to only keep crops what don't have any blank pixels in the vicinity of their center (your primary target). This can be very convenient when your input catalog/coordinates originated from another survey/filter which is not fully covered by your input image, to learn more about this feature, please see the description of the `--checkcenter` option in Section 6.1.4 [Invoking Crop], page 155.

Image coordinates

In image mode (`--mode=img`), Crop interprets the pixel coordinates and widths in units of the input data-elements (for example pixels in an image, not world coordinates). In image mode, only one image may be input. The output crop(s) can be defined in multiple ways as listed below.

Center of multiple crops (in a catalog)

The center of (possibly multiple) crops are read from a text file. In this mode, the columns identified with the `--coordcol` option are interpreted as the center of a crop with a width of `--width` pixels along each dimension. The columns can contain any floating point value. The value to `--output` option is seen as a directory which will host (the possibly multiple) separate crop files, see Section 6.1.4.2 [Crop output], page 160, for more. For a tutorial using this feature, please see Section 2.4 [Hubble visually checks and classifies his catalog], page 57.

Center of a single crop (on the command-line)

The center of the crop is given on the command-line with the `--center` option. The crop width is specified by the `--width` option along each dimension. The given coordinates and width can be any floating point number.

Vertices of a single crop

In Image mode there are two options to define the vertices of a region to crop: `--section` and `--polygon`. The former is lower-level (doesn't accept floating point vertices, and only a rectangular region can be defined), it is also only available in Image mode. Please see Section 6.1.2 [Crop section syntax], page 154, for a full description of this method.

The latter option (`--polygon`) is a higher-level method to define any convex polygon (with any number of vertices) with floating point values. Please see the description of this option in Section 6.1.4 [Invoking Crop], page 155, for its syntax.

WCS coordinates

In WCS mode (`--mode=wcs`), the coordinates and widths are interpreted using the World Coordinate System (WCS, that must accompany the dataset),

not pixel coordinates. In WCS mode, Crop accepts multiple datasets as input. When the cropped region (defined by its center or vertices) overlaps with multiple of the input images/tiles, the overlapping regions will be taken from the respective input (they will be stitched when necessary for each output crop).

In this mode, the input images do not necessarily have to be the same size, they just need to have the same orientation and pixel resolution. Currently only orientation along the celestial coordinates is accepted, if your input has a different orientation you can use Warp's `--align` option to align the image before cropping it (see Section 6.4 [Warp], page 199).

Each individual input image/tile can even be smaller than the final crop. In any case, any part of any of the input images which overlaps with the desired region will be used in the crop. Note that if there is an overlap in the input images/tiles, the pixels from the last input image read are going to be used for the overlap. Crop will not change pixel values, so it assumes your overlapping tiles were cutout from the same original image. There are multiple ways to define your cropped region as listed below.

Center of multiple crops (in a catalog)
: Similar to catalog inputs in Image mode (above), except that the values along each dimension are assumed to have the same units as the dataset's WCS information. For example, the central RA and Dec value for each crop will be read from the first and second calls to the `--coordcol` option. The width of the cropped box (in units of the WCS, or degrees in RA and Dec mode) must be specified with the `--width` option.

Center of a single crop (on the command-line)
: You can specify the center of only one crop box with the `--center` option. If it exists in the input images, it will be cropped similar to the catalog mode, see above also for `--width`.

Vertices of a single crop
: The `--polygon` option is a high-level method to define any convex polygon (with any number of vertices). Please see the description of this option in Section 6.1.4 [Invoking Crop], page 155, for its syntax.

CAUTION: In WCS mode, the image has to be aligned with the celestial coordinates, such that the first FITS axis is parallel (opposite direction) to the Right Ascension (RA) and the second FITS axis is parallel to the declination. If these conditions aren't met for an image, Crop will warn you and abort. You can use Warp's `--align` option to align the input image with these coordinates, see Section 6.4 [Warp], page 199.

As a summary, if you don't specify a catalog, you have to define the cropped region manually on the command-line. In any case the mode is mandatory for Crop to be able to interpret the values given as coordinates or widths.

6.1.2 Crop section syntax

When in image mode, one of the methods to crop only one rectangular section from the input image is to use the `--section` option. Crop has a powerful syntax to read the box parameters from a string of characters. If you leave certain parts of the string to be empty, Crop can fill them for you based on the input image sizes.

To define a box, you need the coordinates of two points: the first (`X1`, `Y1`) and the last pixel (`X2`, `Y2`) pixel positions in the image, or four integer numbers in total. The four coordinates can be specified with one string in this format: '`X1:X2,Y1:Y2`'. This string is given to the `--section` option. Therefore, the pixels along the first axis that are $\geq$`X1` and $\leq$`X2` will be included in the cropped image. The same goes for the second axis. Note that each different term will be read as an integer, not a float. This is a low-level option, for a higher-level way to specify region (any polygon, not just a box), please see the `--polygon` option in Section 6.1.4.1 [Crop options], page 156. Also note that in the FITS standard, pixel indexes along each axis start from unity(1) not zero(0).

You can omit any of the values and they will be filled automatically. The left hand side of the colon (`:`) will be filled with `1`, and the right side with the image size. So, `2:,:` will include the full range of pixels along the second axis and only those with a first axis index larger than `2` in the first axis. If the colon is omitted for a dimension, then the full range is automatically used. So the same string is also equal to `2:,` or `2:` or even `2`. If you want such a case for the second axis, you should set it to: `,2`.

If you specify a negative value, it will be seen as before the indexes of the image which are outside the image along the bottom or left sides when viewed in SAO ds9. In case you want to count from the top or right sides of the image, you can use an asterisk (`*`). When confronted with a `*`, Crop will replace it with the maximum length of the image in that dimension. So `*-10:*+10,*-20:*+20` will mean that the crop box will be 20×40 pixels in size and only include the top corner of the input image with 3/4 of the image being covered by blank pixels, see Section 6.1.3 [Blank pixels], page 154.

If you feel more comfortable with space characters between the values, you can use as many space characters as you wish, just be careful to put your value in double quotes, for example `--section="5:200, 123:854"`. If you forget the quotes, anything after the first space will not be seen by `--section` and you will most probably get an error because the rest of your string will be read as a filename (which most probably doesn't exist). See Section 4.1 [Command-line], page 91, for a description of how the command-line works.

6.1.3 Blank pixels

The cropped box can potentially include pixels that are beyond the image range. For example when a target in the input catalog was very near the edge of the input image. The parts of the cropped image that were not in the input image will be filled with the following two values depending on the data type of the image. In both cases, SAO ds9 will not color code those pixels.

- If the data type of the image is a floating point type (float or double), IEEE NaN (Not a number) will be used.
- For integer types, pixels out of the image will be filled with the value of the `BLANK` keyword in the cropped image header. The value assigned to it is the lowest value possible for that type, so you will probably never need it any way. Only for the

unsigned character type (`BITPIX=8` in the FITS header), the maximum value is used because it is unsigned, the smallest value is zero which is often meaningful.

You can ask for such blank regions to not be included in the output crop image using the `--noblank` option. In such cases, there is no guarantee that the image size of your outputs are what you asked for.

In some survey images, unfortunately they do not use the `BLANK` FITS keyword. Instead they just give all pixels outside of the survey area a value of zero. So by default, when dealing with float or double image types, any values that are 0.0 are also regarded as blank regions. This can be turned off with the `--zeroisnotblank` option.

6.1.4 Invoking Crop

Crop will crop a region from an image. If in WCS mode, it will also stitch parts from separate images in the input files. The executable name is `astcrop` with the following general template

```
$ astcrop [OPTION...] [ASCIIcatalog] ASTRdata ...
```

One line examples:

```
## Crop all objects in cat.txt from image.fits:
$ astcrop --catalog=cat.txt image.fits

## Crop all options in catalog (with RA,DEC) from all the files
## ending in '_drz.fits' in '/mnt/data/COSMOS/':
$ astcrop --mode=wcs --catalog=cat.txt /mnt/data/COSMOS/*_drz.fits

## Crop the outer 10 border pixels of the input image:
$ astcrop --section=10:*-10,10:*-10 --hdu=2 image.fits

## Crop region around RA and Dec of (189.16704, 62.218203):
$ astcrop --mode=wcs --center=189.16704,62.218203 goodsnorth.fits

## Crop region around pixel coordinate (568.342, 2091.719):
$ astcrop --mode=img --center=568.342,2091.719 --width=201 image.fits
```

Crop has one mandatory argument which is the input image name(s), shown above with `ASTRdata ...`. You can use shell expansions, for example `*` for this if you have lots of images in WCS mode. If the crop box centers are in a catalog, you can use the `--catalog` option. In other cases, you have to provide the single cropped output parameters must be given with command-line options. See Section 6.1.4.2 [Crop output], page 160, for how the output file name(s) can be specified. For the full list of general options to all Gnuastro programs (including Crop), please see Section 4.1.2 [Common options], page 95.

Floating point numbers can be used to specify the crop region (except the `--section` option, see Section 6.1.2 [Crop section syntax], page 154). In such cases, the floating point values will be used to find the desired integer pixel indices based on the FITS standard. Hence, Crop ultimately doesn't do any sub-pixel cropping (in other words, it doesn't change pixel values). If you need such crops, you can use Section 6.4 [Warp], page 199, to first warp the image to the a new pixel grid, then crop from that. For example, let's assume you want a crop from pixels 12.982 to 80.982 along the first dimension. You should first

translate the image by -0.482 (note that the edge of a pixel is at integer multiples of 0.5). So you should run Warp with `--translate=-0.482,0` and then crop the warped image with `--section=13:81`.

There are two ways to define the cropped region: with its center or its vertices. See Section 6.1.1 [Crop modes], page 151, for a full description. In the former case, Crop can check if the central region of the cropped image is indeed filled with data or is blank (see Section 6.1.3 [Blank pixels], page 154), and not produce any output when the center is blank, see the description under `--checkcenter` for more.

When in catalog mode, Crop will run in parallel unless you set `--numthreads=1`, see Section 4.4 [Multi-threaded operations], page 112. Note that when multiple outputs are created with threads, the outputs will not be created in the same order. This is because the threads are asynchronous and thus not started in order. This has no effect on each output, see Section 2.4 [Hubble visually checks and classifies his catalog], page 57, for a tutorial on effectively using this feature.

6.1.4.1 Crop options

The options can be classified into the following contexts: Input, Output and operating mode options. Options that are common to all Gnuastro program are listed in Section 4.1.2 [Common options], page 95, and will not be repeated here.

When you are specifying the crop vertices your self (through `--section`, or `--polygon`) on relatively small regions (depending on the resolution of your images) the outputs from image and WCS mode can be approximately equivalent. However, as the crop sizes get large, the curved nature of the WCS coordinates have to be considered. For example, when using `--section`, the right ascension of the bottom left and top left corners will not be equal. If you only want regions within a given right ascension, use `--polygon` in WCS mode.

Input image parameters:

`--hstartwcs=INT`

Specify the first keyword card (line number) to start finding the input image world coordinate system information. Distortions were only recently included in WCSLIB (from version 5). Therefore until now, different telescope would apply their own specific set of WCS keywords and put them into the image header along with those that WCSLIB does recognize. So now that WCSLIB recognizes most of the standard distortion parameters, they will get confused with the old ones and give completely wrong results. For example in the CANDELS-GOODS South images[1].

The two `--hstartwcs` and `--hendwcs` are thus provided so when using older datasets, you can specify what region in the FITS headers you want to use to read the WCS keywords. Note that this is only relevant for reading the WCS information, basic data information like the image size are read separately. These two options will only be considered when the value to `--hendwcs` is larger than that of `--hstartwcs`. So if they are equal or `--hstartwcs` is larger than `--hendwcs`, then all the input keywords will be parsed to get the WCS information of the image.

[1] `https://archive.stsci.edu/pub/hlsp/candels/goods-s/gs-tot/v1.0/`

`--hendwcs=INT`
: Specify the last keyword card to read for specifying the image world coordinate system on the input images. See `--hstartwcs`

Crop box parameters:

`-c FLT[,FLT[,...]]`
`--center=FLT[,FLT[,...]]`
: The central position of the crop in the input image. The positions along each dimension must be separated by a comma (`,`) and fractions are also acceptable. The number of values given to this option must be the same as the dimensions of the input dataset. The width of the crop should be set with `--width`. The units of the coordinates are read based on the value to the `--mode` option, see below.

`-w FLT[,FLT[,...]]`
`--width=FLT[,FLT[,...]]`
: Width of the cropped region about its center. `--width` may take either a single value (to be used for all dimensions) or multiple values (a specific value for each dimension). If in WCS mode, value(s) given to this option will be read in the same units as the dataset's WCS information along this dimension. The final output will have an odd number of pixels to allow easy identification of the pixel which keeps your requested coordinate (from `--center` or `--catalog`).

  The `--width` option also accepts fractions. For example if you want the width of your crop to be 3 by 5 arcseconds along RA and Dec respectively, you can call it with: `--width=3/3600,5/3600`.

  If you want an even sided crop, you can run Crop afterwards with `--section=":*-1,:*-1"` or `--section=2:,2:` (depending on which side you don't need), see Section 6.1.2 [Crop section syntax], page 154.

`-l STR`
`--polygon=STR`
: String of crop polygon vertices. Note that currently only convex polygons should be used. In the future we will make it work for all kinds of polygons. Convex polygons are polygons that do not have an internal angle more than 180 degrees. This option can be used both in the image and WCS modes, see Section 6.1.1 [Crop modes], page 151. The cropped image will be the size of the rectangular region that completely encompasses the polygon. By default all the pixels that are outside of the polygon will be set as blank values (see Section 6.1.3 [Blank pixels], page 154). However, if `--outpolygon` is called all pixels internal to the vertices will be set to blank.

  The syntax for the polygon vertices is similar to, and simpler than, that for `--section`. In short, the dimensions of each coordinate are separated by a comma (`,`) and each vertex is separated by a colon (`:`). You can define as many vertices as you like. If you would like to use space characters between the dimensions and vertices to make them more human-readable, then you have to put the value to this option in double quotation marks.

  For example, let's assume you want to work on the deepest part of the WFC3/IR images of Hubble Space Telescope eXtreme Deep Field (HST-XDF). According

to the webpage (`https://archive.stsci.edu/prepds/xdf/`)[2] the deepest part is contained within the coordinates:

```
[ (53.187414,-27.779152), (53.159507,-27.759633),
  (53.134517,-27.787144), (53.161906,-27.807208) ]
```

They have provided mask images with only these pixels in the WFC3/IR images, but what if you also need to work on the same region in the full resolution ACS images? Also what if you want to use the CANDELS data for the shallow region? Running Crop with `--polygon` will easily pull out this region of the image for you irrespective of the resolution. If you have set the operating mode to WCS mode in your nearest configuration file (see Section 4.2 [Configuration files], page 106), there is no need to call `--mode=wcs` on the command line. You may also provide many FITS images/tiles and Crop will stitch them to produce this cropped region:

```
$ astcrop --mode=wcs desired-filter-image(s).fits           \
   --polygon="53.187414,-27.779152 : 53.159507,-27.759633 : \
              53.134517,-27.787144 : 53.161906,-27.807208"
```

In other cases, you have an image and want to define the polygon yourself (it isn't already published like the example above). As the number of vertices increases, checking the vertex coordinates on a FITS viewer (for example SAO ds9) and typing them in one by one can be very tedious and prone to typo errors.

You can take the following steps to avoid the frustration and possible typos: Open the image with ds9 and activate its "region" mode with Edit→Region. Then define the region as a polygon with Region→Shape→Polygon. Click on the approximate center of the region you want and a small square will appear. By clicking on the vertices of the square you can shrink or expand it, clicking and dragging anywhere on the edges will enable you to define a new vertex. After the region has been nicely defined, save it as a file with Region→Save Regions. You can then select the name and address of the output file, keep the format as `REG` and press "OK". In the next window, keep format as "ds9" and "Coordinate System" as "fk5". A plain text file (let's call it `ds9.reg`) is now created.

You can now convert this plain text file to Crop's polygon format with this command (when typing on the command-line, ignore the "\" at the end of the first and second lines along with the extra spaces, these are only for nice printing):

```
$ v=$(awk 'NR==4' ds9.reg | sed -e's/polygon(//'        \
          -e's/\([^,]*,[^,]*\),/\1:/g' -e's/)//' )
$ astcrop --mode=wcs image.fits --polygon=$v
```

`--outpolygon`

Keep all the regions outside the polygon and mask the inner ones with blank pixels (see Section 6.1.3 [Blank pixels], page 154). This is practically the inverse of the default mode of treating polygons. Note that this option only works

[2] `https://archive.stsci.edu/prepds/xdf/`

when you have only provided one input image. If multiple images are given (in WCS mode), then the full area covered by all the images has to be shown and the polygon excluded. This can lead to a very large area if large surveys like COSMOS are used. So Crop will abort and notify you. In such cases, it is best to crop out the larger region you want, then mask the smaller region with this option.

`-s STR`
`--section=STR`
Section of the input image which you want to be cropped. See Section 6.1.2 [Crop section syntax], page 154, for a complete explanation on the syntax required for this input.

`-x STR/INT`
`--coordcol=STR/INT`
The column in a catalog to read as a coordinate. The value can be either the column number (starting from 1), or a match/search in the table meta-data, see Section 4.6.3 [Selecting table columns], page 121. This option must be called multiple times, depending on the number of dimensions in the input dataset. If it is called more than necessary, the extra columns (later calls to this option on the command-line or configuration files) will be ignored, see Section 4.2.2 [Configuration file precedence], page 107.

`-n STR/INT`
`--namecol=STR/INT`
Column selection of crop file name. The value can be either the column number (starting from 1), or a match/search in the table meta-data, see Section 4.6.3 [Selecting table columns], page 121. This option can be used both in Image and WCS modes, and not a mandatory. When a column is given to this option, the final crop base file name will be taken from the contents of this column. The directory will be determined by the `--output` option (current directory if not given) and the value to `--suffix` will be appended. When this column isn't given, the row number will be used instead.

Output options:

`-c FLT/INT`
`--checkcenter=FLT/INT`
Square box width of region in the center of the image to check for blank values. If any of the pixels in this central region of a crop (defined by its center) are blank, then it will not be stored in an output file. If the value to this option is zero, no checking is done. This check is only applied when the cropped region(s) are defined by their center (not by the vertices, see Section 6.1.1 [Crop modes], page 151).

The units of the value are interpreted based on the `--mode` value (in WCS or pixel units). The ultimate checked region size (in pixels) will be an odd integer around the center (converted from WCS, or when an even number of pixels are given to this option). In WCS mode, the value can be given as fractions, for example if the WCS units are in degrees, `0.1/3600` will correspond to a check size of 0.1 arcseconds.

Because survey regions don't often have a clean square or rectangle shape, some of the pixels on the sides of the survey FITS image don't commonly have any data and are blank (see Section 6.1.3 [Blank pixels], page 154). So when the catalog was not generated from the input image, it often happens that the image does not have data over some of the points.

When the given center of a crop falls in such regions or outside the dataset, and this option has a non-zero value, no crop will be created. Therefore with this option, you can specify a width of a small box (3 pixels is often good enough) around the central pixel of the cropped image. You can check which crops were created and which weren't from the command-line (if `--quiet` was not called, see Section 4.1.2.3 [Operating mode options], page 100), or in Crop's log file (see Section 6.1.4.2 [Crop output], page 160).

`-p STR`
`--suffix=STR`
The suffix (or post-fix) of the output files for when you want all the cropped images to have a special ending. One case where this might be helpful is when besides the science images, you want the weight images (or exposure maps, which are also distributed with survey images) of the cropped regions too. So in one run, you can set the input images to the science images and `--suffix=_s.fits`. In the next run you can set the weight images as input and `--suffix=_w.fits`.

`-b`
`--noblank`
Pixels outside of the input image that are in the crop box will not be used. By default they are filled with blank values (depending on type), see Section 6.1.3 [Blank pixels], page 154. This option only applies only in Image mode, see Section 6.1.1 [Crop modes], page 151.

`-z`
`--zeroisnotblank`
In float or double images, it is common to give the value of zero to blank pixels. If the input image type is one of these two types, such pixels will also be considered as blank. You can disable this behavior with this option, see Section 6.1.3 [Blank pixels], page 154.

Operating mode options:

`-O STR`
`--mode=STR`
Operate in Image mode or WCS mode when the input coordinates can be both image or WCS. The value must either be `img` or `wcs`, see Section 6.1.1 [Crop modes], page 151, for a full description.

6.1.4.2 Crop output

The string given to `--output` option will be interpreted depending on how many crops were requested, see Section 6.1.1 [Crop modes], page 151:

- When a catalog is given, the value of the `--output` (see Section 4.1.2 [Common options], page 95) will be read as the directory to store the output cropped images. Hence if

it doesn't already exist, Crop will abort with an error of a "No such file or directory" error.

The crop file names will consist of two parts: a variable part (the row number of each target starting from 1) along with a fixed string which you can set with the `--suffix` option. Optionally, you may also use the `--namecol` option to define a column in the input catalog to use as the file name instead of numbers.

- When only one crop is desired, the value to `--output` will be read as a file name. If no output is specified or if it is a directory, the output file name will follow the automatic output names of Gnuastro, see Section 4.8 [Automatic output], page 124: The string given to `--suffix` will be replaced with the `.fits` suffix of the input.

The header of each output cropped image will contain the names of the input image(s) it was cut from. If a name is longer than the 70 character space that the FITS standard allows for header keyword values, the name will be cut into several keywords from the nearest slash (`/`). The keywords have the following format: `ICFn_m` (for Crop File). Where `n` is the number of the image used in this crop and `m` is the part of the name (it can be broken into multiple keywords). Following the name is another keyword named `ICFnPIX` which shows the pixel range from that input image in the same syntax as Section 6.1.2 [Crop section syntax], page 154. So this string can be directly given to the `--section` option later.

Once done, a log file can be created in the current directory with the `--log` option. This file will have three columns and the same number of rows as the number of cropped images. There are also comments on the top of the log file explaining basic information about the run and descriptions for the columns. A short description of the columns is also given below:

1. The cropped image file name for that row.
2. The number of input images that were used to create that image.
3. A `0` if the central few pixels (value to the `--checkcenter` option) are blank and `1` if they aren't. When the crop was not defined by its center (see Section 6.1.1 [Crop modes], page 151), or `--checkcenter` was given a value of 0 (see Section 6.1.4 [Invoking Crop], page 155), the center will not be checked and this column will be given a value of `-1`.

6.2 Arithmetic

It is commonly necessary to do operations on some or all of the elements of a dataset independently (pixels in an image). For example, in the reduction of raw data it is necessary to subtract the Sky value (Section 7.1.3 [Sky value], page 211) from each image image. Later (once the images as warped into a single grid using Warp for example, see Section 6.4 [Warp], page 199), the images are co-added (the output pixel grid is the average of the pixels of the individual input images). Arithmetic is Gnuastro's program for such operations on your datasets directly from the command-line. It currently uses the reverse polish or post-fix notation, see Section 6.2.1 [Reverse polish notation], page 162, and will work on the native data types of the input images/data to reduce CPU and RAM resources, see Section 4.5 [Numeric data types], page 115. For more information on how to run Arithmetic, please see Section 6.2.3 [Invoking Arithmetic], page 173.

6.2.1 Reverse polish notation

The most common notation for arithmetic operations is the infix notation (`https://en.wikipedia.org/wiki/Infix_notation`) where the operator goes between the two operands, for example $4 + 5$. While the infix notation is the preferred way in most programming languages, currently Arithmetic does not use it since it will require parenthesis which can complicate the implementation of the code. In the near future we do plan to adopt this notation[3], but for the time being (due to time constraints on the developers), Arithmetic uses the post-fix or reverse polish notation (`https://en.wikipedia.org/wiki/Reverse_Polish_notation`). The Wikipedia article provides some excellent explanation on this notation but here we will give a short summary here for self-sufficiency.

In the post-fix notation, the operator is placed after the operands, as we will see below this removes the need to define parenthesis for most ordinary operators. For example, instead of writing `5+6`, we write `5 6 +`. To easily understand how this notation works, you can think of each operand as a node in a first-in-first-out stack. Every time an operator is confronted, it pops the number of operands it needs from the top of the stack (so they don't exist in the stack any more), does its operation and pushes the result back on top of the stack. So if you want the average of 5 and 6, you would write: `5 6 + 2 /`. The operations that are done are:

1. `5` is an operand, so it is pushed to the top of the stack.
2. `6` is an operand, so it is pushed to the top of the stack.
3. `+` is a binary operator, so pull the top two elements of the stack and perform addition on them (the order is $5 + 6$ in the example above). The result is `11`, push it on top of the stack.
4. `2` is an operand so push it onto the top of the stack.
5. `/` is a binary operator, so pull out the top two elements of the stack (top-most is `2`, then `11`) and divide the second one by the first.

In the Arithmetic program, the operands can be FITS images or numbers. As you can see, very complicated procedures can be created without the need for parenthesis or worrying about precedence. Even functions which take an arbitrary number of arguments can be defined in this notation. This is a very powerful notation and is used in languages like Postscript[4] (the programming language in Postscript and compiled into PDF files) uses this notation.

6.2.2 Arithmetic operators

The recognized operators in Arithmetic are listed below. See Section 6.2.1 [Reverse polish notation], page 162, for more on how the operators and operands should be ordered on the command-line. The operands to all operators can be a data array (for example a FITS image) or a number, the output will be an array or number according to the inputs. For example a number multiplied by an array will produce an array. The conditional operators

[3] `https://savannah.gnu.org/task/index.php?13867`

[4] See the EPS and PDF part of Section 5.2.1 [Recognized file formats], page 138, for a little more on the Postscript language.

will return pixel, or numerical values of 0 (false) or 1 (true) and stored in an `unsigned char` data type (see Section 4.5 [Numeric data types], page 115).

`+` Addition, so "`4 5 +`" is equivalent to $4 + 5$.

`-` Subtraction, so "`4 5 -`" is equivalent to $4 - 5$.

`x` Multiplication, so "`4 5 x`" is equivalent to 4×5.

`/` Division, so "`4 5 /`" is equivalent to $4/5$.

`%` Modulo (remainder), so "`3 2 %`" is equivalent to 1. Note that the modulo operator only works on integer types.

`abs` Absolute value of first operand, so "`4 abs`" is equivalent to $|4|$.

`pow` First operand to the power of the second, so "`4.3 5f pow`" is equivalent to 4.3^5. Currently `pow` will only work on single or double precision floating point numbers or images. To be sure that a number is read as a floating point (even if it doesn't have any non-zero decimals) put an `f` after it.

`sqrt` The square root of the first operand, so "`5 sqrt`" is equivalent to $\sqrt{5}$. The output will have a floating point type, but its precision is determined from the input: if the input is a 64-bit floating point, the output will also be 64-bit. Otherwise, the output will be 32-bit floating point (see Section 4.5 [Numeric data types], page 115, for the respective precision). Therefore if you require 64-bit precision in estimating the square root, convert the input to 64-bit floating point first, for example with `5 float64 sqrt`.

`log` Natural logarithm of first operand, so "`4 log`" is equivalent to $ln(4)$. The output type is determined from the input, see the explanation under `sqrt` for more.

`log10` Base-10 logarithm of first operand, so "`4 log10`" is equivalent to $\log(4)$. The output type is determined from the input, see the explanation under `sqrt` for more.

`minvalue` Minimum (non-blank) value in the top operand on the stack, so "`a.fits minvalue`" will push the the minimum pixel value in this image onto the stack. Therefore this operator is mainly intended for data (for example images), if the top operand is a number, this operator just returns it without any change. So note that when this operator acts on a single image, the output will no longer be an image, but a number. The output of this operand is in the same type as the input.

`maxvalue` Maximum (non-blank) value of first operand in the same type, similar to `minvalue`.

`numbervalue`
Number of non-blank elements in first operand in the `uint64` type, similar to `minvalue`.

`sumvalue` Sum of non-blank elements in first operand in the `float32` type, similar to `minvalue`.

`meanvalue`
: Mean value of non-blank elements in first operand in the `float32` type, similar to `minvalue`.

`stdvalue`
: Standard deviation of non-blank elements in first operand in the `float32` type, similar to `minvalue`.

`medianvalue`
: Median of non-blank elements in first operand with the same type, similar to `minvalue`.

`min`
: The first popped operand to this operator must be a positive integer number which specifies how many further operands should be popped from the stack. All the subsequently popped operands must have the same type and size. This operator (and all the variable-operand operators similar to it that are discussed below) will work in multi-threaded mode unless Arithmetic is called with the `--numthreads=1` option, see Section 4.4 [Multi-threaded operations], page 112.

  Each pixel of the output of the `min` operator will be given the minimum value of the same pixel from all the popped operands/images. For example the following command will produce an image with the same size and type as the three inputs, but each output pixel value will be the minimum of the same pixel's values in all three input images.

```
$ astarithmetic a.fits b.fits c.fits 3 min
```

  Important notes:

  - NaN/blank pixels will be ignored, see Section 6.1.3 [Blank pixels], page 154.
  - The output will have the same type as the inputs. This is natural for the `min` and `max` operators, but for other similar operators (for example `sum`, or `average`) the per-pixel operations will be done in double precision floating point and then stored back in the input type. Therefore, if the input was an integer, C's internal type conversion will be used.

`max`
: Similar to `min`, but the pixels of the output will contain the maximum of the respective pixels in all operands in the stack.

`number`
: Similar to `min`, but the pixels of the output will contain the number of the respective non-blank pixels in all input operands.

`sum`
: Similar to `min`, but the pixels of the output will contain the sum of the respective pixels in all input operands.

`mean`
: Similar to `min`, but the pixels of the output will contain the mean (average) of the respective pixels in all operands in the stack.

`std`
: Similar to `min`, but the pixels of the output will contain the standard deviation of the respective pixels in all operands in the stack.

`median`
: Similar to `min`, but the pixels of the output will contain the median of the respective pixels in all operands in the stack.

`sigclip-number`
: Combine the specified number of inputs into a single output that contains the number of remaining elements after σ-clipping on each element/pixel (for more

on σ-clipping, see Section 7.1.2 [Sigma clipping], page 209). This operator is very similar to `min`, with the exception that it expects two operands (parameters for sigma-clipping) before the total number of inputs. The first popped operand is the termination criteria and the second is the multiple of σ.

For example in the command below, the first popped operand (`0.2`) is the sigma clipping termination criteria. If the termination criteria is larger than 1 it is interpreted as the number of clips to do. But if it is between 0 and 1, then it is the tolerance level on the standard deviation (see Section 7.1.2 [Sigma clipping], page 209). The second popped operand (`5`) is the multiple of sigma to use in sigma-clipping. The third popped operand (`10`) is number of datasets that will be used (similar to the first popped operand to `min`).

```
astarithmetic a.fits b.fits c.fits 3 5 0.2 sigclip-number
```

`sigclip-median`
: Combine the specified number of inputs into a single output that contains the *median* after σ-clipping on each element/pixel. For more see `sigclip-number`.

`sigclip-mean`
: Combine the specified number of inputs into a single output that contains the *mean* after σ-clipping on each element/pixel. For more see `sigclip-number`.

`sigclip-std`
: Combine the specified number of inputs into a single output that contains the *standard deviation* after σ-clipping on each element/pixel. For more see `sigclip-number`.

`filter-mean`
: Apply mean filtering (or moving average (`https://en.wikipedia.org/wiki/Moving_average`)) on the input dataset. During mean filtering, each pixel (data element) is replaced by the mean value of all its surrounding pixels (excluding blank values). The number of surrounding pixels in each dimension (to calculate the mean) is determined through the earlier operands that have been pushed onto the stack prior to the input dataset. The number of necessary operands is determined by the dimensions of the input dataset (first popped operand). The order of the dimensions on the command-line is the order in FITS format. Here is one example:

  ```
  $ astarithmetic 5 4 image.fits filter-mean
  ```

  In this example, each pixel is replaced by the mean of a 5 by 4 box around it. The box is 5 pixels along the first FITS dimension (horizontal when viewed in ds9) and 4 pixels along the second FITS dimension (vertical).

  Each pixel will be placed in the center of the box that the mean is calculated on. If the given width along a dimension is even, then the center is assumed to be between the pixels (not in the center of a pixel). When the pixel is close to the edge, the pixels of the box that fall outside the image are ignored. Therefore, on the edge, less points will be used in calculating the mean.

  The final effect of mean filtering is to smooth the input image, it is essentially a convolution with a kernel that has identical values for all its pixels (is flat), see Section 6.3.1.1 [Convolution process], page 178.

Note that blank pixels will also be affected by this operator: if there are any non-blank elements in the box surrounding a blank pixel, in the filtered image, it will have the mean of the non-blank elements, therefore it won't be blank any more. If blank elements are important for your analysis, you can use the `isblank` with the `where` operator to set them back to blank after filtering.

`filter-median`

Apply median filtering (`https://en.wikipedia.org/wiki/Median_filter`) on the input dataset. This is very similar to `filter-mean`, except that instead of the mean value of the box pixels, the median value is used to replace a pixel value. For more on how to use this operator, please see `filter-mean`.

The median is less susceptible to outliers compared to the mean. As a result, after median filtering, the pixel values will be more discontinuous than mean filtering.

`filter-sigclip-mean`

Apply a σ-clipped mean filtering onto the input dataset. This is very similar to `filter-mean`, except that all outliers (identified by the σ-clipping algorithm) have been removed, see Section 7.1.2 [Sigma clipping], page 209, for more on the basics of this algorithm. As described there, two extra input parameters are necessary for σ-clipping: the multiple of σ and the termination criteria. `filter-sigclip-mean` therefore needs to pop two other operands from the stack after the dimensions of the box.

For example the line below uses the same box size as the example of `filter-mean`. However, all elements in the box that are iteratively beyond 3σ of the distribution's median are removed from the final calculation of the mean until the change in σ is less than 0.2.

```
$ astarithmetic 3 0.2 5 4 image.fits filter-sigclip-mean
```

The median (which needs a sorted dataset) is necessary for σ-clipping, therefore `filter-sigclip-mean` can be significantly slower than `filter-mean`. However, if there are strong outliers in the dataset that you want to ignore (for example emission lines on a spectrum when finding the continuum), this is a much better solution.

`filter-sigclip-median`

Apply a σ-clipped median filtering onto the input dataset. This operator and its necessary operands are almost identical to `filter-sigclip-mean`, except that after σ-clipping, the median value (which is less affected by outliers than the mean) is added back to the stack.

`interpolate-medianngb`

Interpolate all the blank elements of the second popped operand with the median of its nearest non-blank neighbors. The number of the nearest non-blank neighbors used to calculate the median is given by the first popped operand. Note that the distance of the nearest non-blank neighbors is irrelevant in this interpolation.

`collapse-sum`

Collapse the given dataset (second popped operand), by summing all elements along the first popped operand (a dimension in FITS standard: counting from one, from fastest dimension). The returned dataset has one dimension less compared to the input.

The output will have a double-precision floating point type irrespective of the input dataset's type. Doing the operation in double-precision (64-bit) floating point will help the collapse (summation) be affected less by floating point errors. But afterwards, single-precision floating points are usually enough in real (noisy) datasets. So depending on the type of the input and its nature, it is recommended to use one of the type conversion operators on the returned dataset.

If any WCS is present, the returned dataset will also lack the respective dimension in its WCS matrix. Therefore, when the WCS is important for later processing, be sure that the input is aligned with the respective axises: all non-diagonal elements in the WCS matrix are zero.

One common application of this operator is the creation of pseudo broad-band or narrow-band 2D images from 3D data cubes. For example integral field unit (IFU) data products that have two spatial dimensions (first two FITS dimensions) and one spectral dimension (third FITS dimension). The command below will collapse the whole third dimension into a 2D array the size of the first two dimensions, and then convert the output to single-precision floating point (as discussed above).

```
$ astarithmetic cube.fits 3 collapse-sum float32
```

`collapse-mean`

Similar to `collapse-sum`, but the returned dataset will be the mean value along the collapsed dimension, not the sum.

`collapse-number`

Similar to `collapse-sum`, but the returned dataset will be the number of non-blank values along the collapsed dimension. The output will have a 32-bit signed integer type. If the input dataset doesn't have blank values, all the elements in the returned dataset will have a single value (the length of the collapsed dimension). Therefore this is mostly relevant when there are blank values in the dataset.

`collapse-min`

Similar to `collapse-sum`, but the returned dataset will have the same numeric type as the input and will contain the minimum value for each pixel along the collapsed dimension.

`collapse-max`

Similar to `collapse-sum`, but the returned dataset will have the same numeric type as the input and will contain the maximum value for each pixel along the collapsed dimension.

`erode` Erode the foreground pixels (with value `1`) of the input dataset (second popped operand). The first popped operand is the connectivity (see description in

`connected-components`). Erosion is simply a flipping of all foreground pixels (to background; with value `0`) that are "touching" background pixels. "Touching" is defined by the connectivity. In effect, this carves off the outer borders of the foreground, making them thinner. This operator assumes a binary dataset (all pixels are `0` and `1`).

`dilate` Dilate the foreground pixels (with value `1`) of the input dataset (second popped operand). The first popped operand is the connectivity (see description in `connected-components`). Erosion is simply a flipping of all background pixels (with value `0`) to foreground that are "touching" foreground pixels. "Touching" is defined by the connectivity. In effect, this expands the outer borders of the foreground. This operator assumes a binary dataset (all pixels are `0` and `1`).

`connected-components`
Find the connected components in the input dataset (second popped operand). The first popped is the connectivity used in the connected components algorithm. The second popped operand is the dataset where connected components are to be found. It is assumed to be a binary image (with values of 0 or 1). It must have an 8-bit unsigned integer type which is the format produced by conditional operators. This operator will return a labeled dataset where the non-zero pixels in the input will be labeled with a counter (starting from 1).

The connectivity is a number between 1 and the number of dimensions in the dataset (inclusive). 1 corresponds to the weakest (symmetric) connectivity between elements and the number of dimensions the strongest. For example on a 2D image, a connectivity of 1 corresponds to 4-connected neighbors and 2 corresponds to 8-connected neighbors.

One example usage of this operator can be the identification of regions above a certain threshold, as in the command below. With this command, Arithmetic will first separate all pixels greater than 100 into a binary image (where pixels with a value of 1 are above that value). Afterwards, it will label all those that are connected.

```
$ astarithmetic in.fits 100 gt 2 connected-components
```

If your input dataset doesn't have a binary type, but you know all its values are 0 or 1, you can use the `uint8` operator (below) to convert it to binary.

`fill-holes`
Flip background (0) pixels surrounded by foreground (1) in a binary dataset. This operator takes two operands (similar to `connected-components`): the first popped operand is the connectivity (to define a hole) and the second is the binary (0 or 1 valued) dataset to fill holes in.

`invert` Invert an unsigned integer dataset. This is the only operator that ignores blank values (which are set to be the maximum values in the unsigned integer types).

This is useful in cases where the target(s) has(have) been imaged in absorption as raw formats (which are unsigned integer types). With this option, the maximum value for the given type will be subtracted from each pixel value, thus "inverting" the image, so the target(s) can be treated as emission. This can be useful when the higher-level analysis methods/tools only work on emission (positive skew in the noise, not negative).

`lt` Less than: If the second popped (or left operand in infix notation, see Section 6.2.1 [Reverse polish notation], page 162) value is smaller than the first popped operand, then this function will return a value of 1, otherwise it will return a value of 0. If both operands are images, then all the pixels will be compared with their counterparts in the other image. If only one operand is an image, then all the pixels will be compared with the the single value (number) of the other operand. Finally if both are numbers, then the output is also just one number (0 or 1). When the output is not a single number, it will be stored as an `unsigned char` type.

`le` Less or equal: similar to `lt` ('less than' operator), but returning 1 when the second popped operand is smaller or equal to the first.

`gt` Greater than: similar to `lt` ('less than' operator), but returning 1 when the second popped operand is greater than the first.

`ge` Greater or equal: similar to `lt` ('less than' operator), but returning 1 when the second popped operand is larger or equal to the first.

`eq` Equality: similar to `lt` ('less than' operator), but returning 1 when the two popped operands are equal (to double precision floating point accuracy).

`ne` Non-Equality: similar to `lt` ('less than' operator), but returning 1 when the two popped operands are *not* equal (to double precision floating point accuracy).

`and` Logical AND: returns 1 if both operands have a non-zero value and 0 if both are zero. Both operands have to be the same kind: either both images or both numbers.

`or` Logical OR: returns 1 if either one of the operands is non-zero and 0 only when both operators are zero. Both operands have to be the same kind: either both images or both numbers.

`not` Logical NOT: returns 1 when the operand is zero and 0 when the operand is non-zero. The operand can be an image or number, for an image, it is applied to each pixel separately.

`isblank` Test for a blank value (see Section 6.1.3 [Blank pixels], page 154). In essence, this is very similar to the conditional operators: the output is either 1 or 0 (see the 'less than' operator above). The difference is that it only needs one operand. Because of the definition of a blank pixel, a blank value is not even equal to itself, so you cannot use the equal operator above to select blank pixels. See the "Blank pixels" box below for more on Blank pixels in Arithmetic.

`where` Change the input (pixel) value *where*/if a certain condition holds. The conditional operators above can be used to define the condition. Three operands are required for `where`. The input format is demonstrated in this simplified example:

```
$ astarithmetic modify.fits binary.fits if-true.fits where
```

The value of any pixel in `modify.fits` that corresponds to a non-zero *and* non-blank pixel of `binary.fits` will be changed to the value of the same pixel in `if-true.fits` (this may also be a number). The 3rd and 2nd popped operands

(`modify.fits` and `binary.fits` respectively, see Section 6.2.1 [Reverse polish notation], page 162) have to have the same dimensions/size. `if-true.fits` can be either a number, or have the same dimension/size as the other two.

The 2nd popped operand (`binary.fits`) has to have `uint8` (or `unsigned char` in standard C) type (see Section 4.5 [Numeric data types], page 115). It is treated as a binary dataset (with only two values: zero and non-zero, hence the name `binary.fits` in this example). However, commonly you won't be dealing with an actual FITS file of a condition/binary image. You will probably define the condition in the same run based on some other reference image and use the conditional and logical operators above to make a true/false (or one/zero) image for you internally. For example the case below:

```
$ astarithmetic in.fits reference.fits 100 gt new.fits where
```

In the example above, any of the `in.fits` pixels that has a value in `reference.fits` greater than `100`, will be replaced with the corresponding pixel in `new.fits`. Effectively the `reference.fits 100 gt` part created the condition/binary image which was added to the stack (in memory) and later used by `where`. The command above is thus equivalent to these two commands:

```
$ astarithmetic reference.fits 100 gt --output=binary.fits
$ astarithmetic in.fits binary.fits new.fits where
```

Finally, the input operands are read and used independently, so you can use the same file more than once as any of the operands.

When the 1st popped operand to `where` (`if-true.fits`) is a single number, it may be a NaN value (or any blank value, depending on its type) like the example below (see Section 6.1.3 [Blank pixels], page 154). When the number is blank, it will be converted to the blank value of the type of the 3rd popped operand (`in.fits`). Hence, in the example below, all the pixels in `reference.fits` that have a value greater than 100, will become blank in the natural data type of `in.fits` (even though NaN values are only defined for floating point types).

```
$ astarithmetic in.fits reference.fits 100 gt nan where
```

`bitand` Bitwise AND operator: only bits with values of 1 in both popped operands will get the value of 1, the rest will be set to 0. For example (assuming numbers can be written as bit strings on the command-line): `00101000 00100010 bitand` will give `00100000`. Note that the bitwise operators only work on integer type datasets.

`bitor` Bitwise inclusive OR operator: The bits where at least one of the two popped operands has a 1 value get a value of 1, the others 0. For example (assuming numbers can be written as bit strings on the command-line): `00101000 00100010 bitand` will give `00101010`. Note that the bitwise operators only work on integer type datasets.

`bitxor` Bitwise exclusive OR operator: A bit will be 1 if it differs between the two popped operands. For example (assuming numbers can be written as bit strings on the command-line): `00101000 00100010 bitand` will give `00001010`. Note that the bitwise operators only work on integer type datasets.

`lshift` Bitwise left shift operator: shift all the bits of the first operand to the left by a number of times given by the second operand. For example (assuming numbers can be written as bit strings on the command-line): `00101000 2 lshift` will give `10100000`. This is equivalent to multiplication by 4. Note that the bitwise operators only work on integer type datasets.

`rshift` Bitwise right shift operator: shift all the bits of the first operand to the right by a number of times given by the second operand. For example (assuming numbers can be written as bit strings on the command-line): `00101000 2 rshift` will give `00001010`. Note that the bitwise operators only work on integer type datasets.

`bitnot` Bitwise not (more formally known as one's complement) operator: flip all the bits of the popped operand (note that this is the only unary, or single operand, bitwise operator). In other words, any bit with a value of `0` is changed to `1` and vice-versa. For example (assuming numbers can be written as bit strings on the command-line): `00101000 bitnot` will give `11010111`. Note that the bitwise operators only work on integer type datasets/numbers.

`uint8` Convert the type of the popped operand to 8-bit unsigned integer type (see Section 4.5 [Numeric data types], page 115). The internal conversion of C will be used.

`int8` Convert the type of the popped operand to 8-bit signed integer type (see Section 4.5 [Numeric data types], page 115). The internal conversion of C will be used.

`uint16` Convert the type of the popped operand to 16-bit unsigned integer type (see Section 4.5 [Numeric data types], page 115). The internal conversion of C will be used.

`int16` Convert the type of the popped operand to 16-bit signed integer (see Section 4.5 [Numeric data types], page 115). The internal conversion of C will be used.

`uint32` Convert the type of the popped operand to 32-bit unsigned integer type (see Section 4.5 [Numeric data types], page 115). The internal conversion of C will be used.

`int32` Convert the type of the popped operand to 32-bit signed integer type (see Section 4.5 [Numeric data types], page 115). The internal conversion of C will be used.

`uint64` Convert the type of the popped operand to 64-bit unsigned integer (see Section 4.5 [Numeric data types], page 115). The internal conversion of C will be used.

`float32` Convert the type of the popped operand to 32-bit (single precision) floating point (see Section 4.5 [Numeric data types], page 115). The internal conversion of C will be used.

`float64` Convert the type of the popped operand to 64-bit (double precision) floating point (see Section 4.5 [Numeric data types], page 115). The internal conversion of C will be used.

`set-AAA` Set the characters after the dash (`AAA` in the case shown here) as a name for the first popped operand on the stack. The named dataset will be freed from memory as soon as it is no longer needed, or if the name is reset to refer to another dataset later in the command. This operator thus enables re-usability of a dataset without having to re-read it from a file every time it is necessary during a process. When a dataset is necessary more than once, this operator can thus help simplify reading/writing on the command-line (thus avoiding potential bugs), while also speeding up the processing.

Like all operators, this operator pops the top operand off of the main processing stack, but unlike other operands, it won't add anything back to the stack immediately. It will keep the popped dataset in memory through a separate list of named datasets (not on the main stack). That list will be used to add/copy any requested dataset to the main processing stack when the name is called.

The name to give the popped dataset is part of the operator's name. For example the `set-a` operator of the command below, gives the name "`a`" to the contents of `image.fits`. This name is then used instead of the actual filename to multiply the dataset by two.

```
$ astarithmetic image.fits set-a a 2 x
```

The name can be any string, but avoid strings ending with standard filename suffixes (for example `.fits`)[5].

One example of the usefulness of this operator is in the `where` operator. For example, let's assume you want to mask all pixels larger than `5` in `image.fits` (extension number 1) with a NaN value. Without setting a name for the dataset, you have to read the file two times from memory in a command like this:

```
$ astarithmetic image.fits image.fits 5 gt nan where -g1
```

But with this operator you can simply give `image.fits` the name `i` and simplify the command above to the more readable one below (which greatly helps when the filename is long):

```
$ astarithmetic image.fits set-i   i i 5 gt nan where
```

`tofile-AAA`

Write the top operand on the operands stack into a file called `AAA` (can be any FITS file name) without changing the operands stack. If you don't need the dataset any more and would like to free it, see the `tofilefree` operator below.

By default, any file that is given to this operator is deleted before Arithmetic actually starts working on the input datasets. The deletion can be deactivated with the `--dontdelete` option (as in all Gnuastro programs, see Section 4.1.2.1 [Input/Output options], page 95). If the same FITS file is given to this operator multiple times, it will contain multiple extensions (in the same order that it was called.

For example the operator `tofile-check.fits` will write the top operand to `check.fits`. Since it doesn't modify the operands stack, this operator is very

[5] A dataset name like `a.fits` (which can be set with `set-a.fits`) will cause confusion in the the initial parser of Arithmetic. It will assume this name is a FITS file, and if it is used multiple times, Arithmetic will abort, complaining that you haven't provided enough HDUs.

convenient when you want to debug, or understanding, a string of operators and operands given to Arithmetic: simply put `tofile-AAA` anywhere in the process to see what is happening behind the scenes without modifying the overall process.

`tofilefree-AAA`
: Similar to the `tofile` operator, with the only difference that the dataset that is written to a file is popped from the operand stack and freed from memory (cannot be used any more).

Blank pixels in Arithmetic: Blank pixels in the image (see Section 6.1.3 [Blank pixels], page 154) will be stored based on the data type. When the input is floating point type, blank values are NaN. One aspect of NaN values is that by definition they will fail on *any* comparison. Hence both equal and not-equal operators will fail when both their operands are NaN! Therefore, the only way to guarantee selection of blank pixels is through the `isblank` operator explained above.

One way you can exploit this property of the NaN value to your advantage is when you want a fully zero-valued image (even over the blank pixels) based on an already existing image (with same size and world coordinate system settings). The following command will produce this for you:

```
$ astarithmetic input.fits nan eq --output=all-zeros.fits
```

Note that on the command-line you can write NaN in any case (for example `NaN`, or `NAN` are also acceptable). Reading NaN as a floating point number in Gnuastro isn't case-sensitive.

6.2.3 Invoking Arithmetic

Arithmetic will do pixel to pixel arithmetic operations on the individual pixels of input data and/or numbers. For the full list of operators with explanations, please see Section 6.2.2 [Arithmetic operators], page 162. Any operand that only has a single element (number, or single pixel FITS image) will be read as a number, the rest of the inputs must have the same dimensions. The general template is:

```
$ astarithmetic [OPTION...] ASTRdata1 [ASTRdata2] OPERATOR ...
```

One line examples:

```
## Calculate (10.32-3.84)^2.7 quietly (will just print 155.329):
$ astarithmetic -q 10.32 3.84 - 2.7 pow

## Inverse the input image (1/pixel):
$ astarithmetic 1 image.fits / --out=inverse.fits

## Multiply each pixel in image by -1:
$ astarithmetic image.fits -1 x --out=negative.fits

## Subtract extension 4 from extension 1 (counting from zero):
$ astarithmetic image.fits image.fits - --out=skysub.fits          \
                --hdu=1 --hdu=4
```

```
## Add two images, then divide them by 2 (2 is read as floating point):
$ astarithmetic image1.fits image2.fits + 2f / --out=average.fits

## Use Arithmetic's average operator:
$ astarithmetic image1.fits image2.fits average --out=average.fits

## Calculate the median of three images in three separate extensions:
$ astarithmetic img1.fits img2.fits img3.fits median               \
                -h0 -h1 -h2 --out=median.fits
```

Arithmetic's notation for giving operands to operators is fully described in Section 6.2.1 [Reverse polish notation], page 162. The output dataset is last remaining operand on the stack. When the output dataset a single number, it will be printed on the command-line. When the output is an array, it will be stored as a file.

The name of the final file can be specified with the `--output` option, but if its not given, Arithmetic will use "automatic output" on the name of the first FITS image encountered to generate an output file name, see Section 4.8 [Automatic output], page 124. By default, if the output file already exists, it will be deleted before Arithmetic starts operation. However, this can be disabled with the `--dontdelete` option (see below). At any point during Arithmetic's operation, you can also write the top operand on the stack to a file, using the `tofile` or `tofilefree` operators, see Section 6.2.2 [Arithmetic operators], page 162.

By default, the world coordinate system (WCS) information of the output dataset will be taken from the first input image (that contains a WCS) on the command-line. This can be modified with the `--wcsfile` and `--wcshdu` options described below. When the `--quiet` option isn't given, the name and extension of the dataset used for the output's WCS is printed on the command-line.

Through operators like those starting with `collapse-`, the dimensionality of the inputs may not be the same as the outputs. By default, when the output is 1D, Arithmetic will write it as a table, not an image/array. The format of the output table (plain text or FITS ASCII or binary) can be set with the `--tableformat` option, see Section 4.1.2.1 [Input/Output options], page 95). You can disable this feature (write 1D arrays as FITS images/arrays) with the `--onedasimage` option.

See Section 4.1.2 [Common options], page 95, for a review of the options in all Gnuastro programs. Arithmetic just redefines the `--hdu` and `--dontdelete` options as explained below.

`-h INT/STR`
`--hdu INT/STR`
: The header data unit of the input FITS images, see Section 4.1.2.1 [Input/Output options], page 95. Unlike most options in Gnuastro (which will ultimately only have one value for this option), Arithmetic allows `--hdu` to be called multiple times and the value of each invocation will be stored separately (for the unlimited number of input images you would like to use). Recall that for other programs this (common) option only takes a single value. So in other programs, if you specify it multiple times on the command-line, only the last value will be used and in the configuration files, it will be ignored if it already has a value.

The order of the values to `--hdu` has to be in the same order as input FITS images. Options are first read from the command-line (from left to right), then top-down in each configuration file, see Section 4.2.2 [Configuration file precedence], page 107.

If the number of HDUs is less than the number of input images, Arithmetic will abort and notify you. However, if there are more HDUs than FITS images, there is no problem: they will be used in the given order (every time a FITS image comes up on the stack) and the extra HDUs will be ignored in the end. So there is no problem with having extra HDUs in the configuration files and by default several HDUs with a value of `0` are kept in the system-wide configuration file when you install Gnuastro.

`-g INT/STR`
`--globalhdu INT/STR`
Use the value to this option as the HDU of all input FITS files. This option is very convenient when you have many input files and the dataset of interest is in the same HDU of all the files. When this option is called, any values given to the `--hdu` option (explained above) are ignored and will not be used.

`-w STR`
`--wcsfile STR`
FITS Filename containing the WCS structure that must be written to the output. The HDU/extension should be specified with `--wcshdu`.

When this option is used, the respective WCS will be read before any processing is done on the command-line and directly used in the final output. If the given file doesn't have any WCS, then the default WCS (first file on the command-line with WCS) will be used in the output.

This option will mostly be used when the default file (first of the set of inputs) is not the one containing your desired WCS. But with this option, you can also use Arithmetic to rewrite/change the WCS of an existing FITS dataset from another file:

```
$ astarithmetic data.fits --wcsfile=other.fits -ofinal.fits
```

`-W STR`
`--wcshdu STR`
HDU/extension to read the WCS within the file given to `--wcsfile`. For more, see the description of `--wcsfile`.

`-O`
`--onedasimage`
When final dataset to write as output only has one dimension, write it as a FITS image/array. By default, if the output is 1D, it will be written as a table, see above.

`-D`
`--dontdelete`
Don't delete the output file, or files given to the `tofile` or `tofilefree` operators, if they already exist. Instead append the desired datasets to the extensions

that already exist in the respective file. Note it doesn't matter if the final output file name is given with the the `--output` option, or determined automatically.

Arithmetic treats this option differently from its default operation in other Gnuastro programs (see Section 4.1.2.1 [Input/Output options], page 95). If the output file exists, when other Gnuastro programs are called with `--dontdelete`, they simply complain and abort. But when Arithmetic is called with `--dontdelete`, it will appended the dataset(s) to the existing extension(s) in the file.

Arithmetic accepts two kinds of input: images and numbers. Images are considered to be any of the inputs that is a file name of a recognized type (see Section 4.1.1.1 [Arguments], page 93) and has more than one element/pixel. Numbers on the command-line will be read into the smallest type (see Section 4.5 [Numeric data types], page 115) that can store them, so `-2` will be read as a `char` type (which is signed on most systems and can thus keep negative values), `2500` will be read as an `unsigned short` (all positive numbers will be read as unsigned), while `3.1415926535897` will be read as a `double` and `3.14` will be read as a `float`. To force a number to be read as float, add a `f` after it, so `5f` will be added to the stack as `float` (see Section 6.2.1 [Reverse polish notation], page 162).

Unless otherwise stated (in Section 6.2.2 [Arithmetic operators], page 162), the operators can deal with numeric multiple data types (see Section 4.5 [Numeric data types], page 115). For example in "`a.fits b.fits +`", the image types can be `long` and `float`. In such cases, C's internal type conversion will be used. The output type will be set to the higher-ranking type of the two inputs. Unsigned integer types have smaller ranking than their signed counterparts and floating point types have higher ranking than the integer types. So the internal C type conversions done in the example above are equivalent to this piece of C:

```
size_t i;
long a[100];
float b[100], out[100];
for(i=0;i<100;++i) out[i]=a[i]+b[i];
```

Relying on the default C type conversion significantly speeds up the processing and also requires less RAM (when using very large images).

Some operators can only work on integer types (of any length, for example bitwise operators) while others only work on floating point types, (currently only the `pow` operator). In such cases, if the operand type(s) are different, an error will be printed. Arithmetic also comes with internal type conversion operators which you can use to convert the data into the appropriate type, see Section 6.2.2 [Arithmetic operators], page 162.

The hyphen (`-`) can be used both to specify options (see Section 4.1.1.2 [Options], page 93) and also to specify a negative number which might be necessary in your arithmetic. In order to enable you to do this, Arithmetic will first parse all the input strings and if the first character after a hyphen is a digit, then that hyphen is temporarily replaced by the vertical tab character which is not commonly used. The arguments are then parsed and these strings will not be specified as an option. Then the given arguments are parsed and any vertical tabs are replaced back with a hyphen so they can be read as negative numbers. Therefore, as long as the names of the files you want to work on, don't start with a vertical tab followed by a digit, there is no problem. An important consequence of this implemen-

tation is that you should not write negative fractions like this: `-.3`, instead write them as `-0.3`.

Without any images, Arithmetic will act like a simple calculator and print the resulting output number on the standard output like the first example above. If you really want such calculator operations on the command-line, AWK (GNU AWK is the most common implementation) is much faster, easier and much more powerful. For example, the numerical one-line example above can be done with the following command. In general AWK is a fantastic tool and GNU AWK has a wonderful manual (`https://www.gnu.org/software/gawk/manual/`). So if you often confront situations like this, or have to work with large text tables/catalogs, be sure to checkout AWK and simplify your life.

```
$ echo "" | awk '{print (10.32-3.84)^2.7}'
155.329
```

6.3 Convolve

On an image, convolution can be thought of as a process to blur or remove the contrast in an image. If you are already familiar with the concept and just want to run Convolve, you can jump to Section 6.3.4 [Convolution kernel], page 195, and Section 6.3.5 [Invoking Convolve], page 196, and skip the lengthy introduction on the basic definitions and concepts of convolution.

There are generally two methods to convolve an image. The first and more intuitive one is in the "spatial domain" or using the actual image pixel values, see Section 6.3.1 [Spatial domain convolution], page 178. The second method is when we manipulate the "frequency domain", or work on the magnitudes of the different frequencies that constitute the image, see Section 6.3.2 [Frequency domain and Fourier operations], page 179. Understanding convolution in the spatial domain is more intuitive and thus recommended if you are just starting to learn about convolution. However, getting a good grasp of the frequency domain is a little more involved and needs some concentration and some mathematical proofs. However, its reward is a faster operation and more importantly a very fundamental understanding of this very important operation.

Convolution of an image will generally result in blurring the image because it mixes pixel values. In other words, if the image has sharp differences in neighboring pixel values[6], those sharp differences will become smoother. This has very good consequences in detection of signal in noise for example. In an actual observed image, the variation in neighboring pixel values due to noise can be very high. But after convolution, those variations will decrease and we have a better hope in detecting the possible underlying signal. Another case where convolution is extensively used is in mock images and modeling in general, convolution can be used to simulate the effect of the atmosphere or the optical system on the mock profiles that we create, see Section 8.1.1.2 [Point spread function], page 285. Convolution is a very interesting and important topic in any form of signal analysis (including astronomical observations). So we have thoroughly[7] explained the concepts behind it in the following sub-sections.

[6] In astronomy, the only major time we confront such sharp borders in signal are cosmic rays. All other sources of signal in an image are already blurred by the atmosphere or the optics of the instrument.

[7] A mathematician will certainly consider this explanation is incomplete and inaccurate. However this text is written for an understanding on the operations that are done on a real (not complex, discrete and noisy) astronomical image, not any general form of abstract function

6.3.1 Spatial domain convolution

The pixels in an input image represent different "spatial" positions, therefore when convolution is done only using the actual input pixel values, we name the process as being done in the "Spatial domain". In particular this is in contrast to the "frequency domain" that we will discuss later in Section 6.3.2 [Frequency domain and Fourier operations], page 179. In the spatial domain (and in realistic situations where the image and the convolution kernel don't extend to infinity), convolution is the process of changing the value of one pixel to the *weighted* average of all the pixels in its *neighborhood*.

The 'neighborhood' of each pixel (how many pixels in which direction) and the 'weight' function (how much each neighboring pixel should contribute depending on its position) are given through a second image which is known as a "kernel"[8].

6.3.1.1 Convolution process

In convolution, the kernel specifies the weight and positions of the neighbors of each pixel. To find the convolved value of a pixel, the central pixel of the kernel is placed on that pixel. The values of each overlapping pixel in the kernel and image are multiplied by each other and summed for all the kernel pixels. To have one pixel in the center, the sides of the convolution kernel have to be an odd number. This process effectively mixes the pixel values of each pixel with its neighbors, resulting in a blurred image compared to the sharper input image.

Formally, convolution is one kind of linear 'spatial filtering' in image processing texts. If we assume that the kernel has $2a+1$ and $2b+1$ pixels on each side, the convolved value of a pixel placed at x and y ($C_{x,y}$) can be calculated from the neighboring pixel values in the input image (I) and the kernel (K) from

$$C_{x,y} = \sum_{s=-a}^{a} \sum_{t=-b}^{b} K_{s,t} \times I_{x+s,y+t}.$$

Any pixel coordinate that is outside of the image in the equation above will be considered to be zero. When the kernel is symmetric about its center the blurred image has the same orientation as the original image. However, if the kernel is not symmetric, the image will be affected in the opposite manner, this is a natural consequence of the definition of spatial filtering. In order to avoid this we can rotate the kernel about its center by 180 degrees so the convolved output can have the same original orientation. Technically speaking, only if the kernel is flipped the process is known *Convolution*. If it isn't it is known as *Correlation*.

To be a weighted average, the sum of the weights (the pixels in the kernel) have to be unity. This will have the consequence that the convolved image of an object and unconvolved object will have the same brightness (see Section 8.1.3 [Flux Brightness and magnitude], page 290), which is natural, because convolution should not eat up the object photons, it only disperses them.

6.3.1.2 Edges in the spatial domain

In purely 'linear' spatial filtering (convolution), there are problems on the edges of the input image. Here we will explain the problem in the spatial domain. For a discussion of this

[8] Also known as filter, here we will use 'kernel'.

problem from the frequency domain perspective, see Section 6.3.2.10 [Edges in the frequency domain], page 194. The problem originates from the fact that on the edges, in practice[9], the sum of the weights we use on the actual image pixels is not unity. For example, as discussed above, a profile in the center of an image will have the same brightness before and after convolution. However, for partially imaged profile on the edge of the image, the brightness (sum of its pixel fluxes within the image, see Section 8.1.3 [Flux Brightness and magnitude], page 290) will not be equal, some of the flux is going to be 'eaten' by the edges.

If you ran `$ make check` on the source files of Gnuastro, you can see the this effect by comparing the `convolve_frequency.fits` with `convolve_spatial.fits` in the `./tests/` directory. In the spatial domain, by default, no assumption will be made about pixels outside of the image or any blank pixels in the image. The problem explained above will also occur on the sides of blank regions (see Section 6.1.3 [Blank pixels], page 154). The solution to this edge effect problem is only possible in the spatial domain. For pixels near the edge, we have to abandon the assumption that the sum of the kernel pixels is unity during the convolution process[10]. So taking W as the sum of the kernel pixels that overlapped with non-blank and in-image pixels, the equation in Section 6.3.1.1 [Convolution process], page 178, will become:

$$C_{x,y} = \frac{\sum_{s=-a}^{a}\sum_{t=-b}^{b} K_{s,t} \times I_{x+s,y+t}}{W}.$$

In this manner, objects which are near the edges of the image or blank pixels will also have the same brightness (within the image) before and after convolution. This correction is applied by default in Convolve when convolving in the spatial domain. To disable it, you can use the `--noedgecorrection` option. In the frequency domain, there is no way to avoid this loss of flux near the edges of the image, see Section 6.3.2.10 [Edges in the frequency domain], page 194, for an interpretation from the frequency domain perspective.

Note that the edge effect discussed here is different from the one in Section 8.1.2 [If convolving afterwards], page 289. In making mock images we want to simulate a real observation. In a real observation the images of the galaxies on the sides of the CCD are first blurred by the atmosphere and instrument, then imaged. So light from the parts of a galaxy which are immediately outside the CCD will affect the parts of the galaxy which are covered by the CCD. Therefore in modeling the observation, we have to convolve an image that is larger than the input image by exactly half of the convolution kernel. We can hence conclude that this correction for the edges is only useful when working on actual observed images (where we don't have any more data on the edges) and not in modeling.

6.3.2 Frequency domain and Fourier operations

Getting a good grip on the frequency domain is usually not an easy job! So we have decided to give the issue a complete review here. Convolution in the frequency domain (see Section 6.3.2.6 [Convolution theorem], page 187) heavily relies on the concepts of Fourier transform (Section 6.3.2.4 [Fourier transform], page 185) and Fourier series (Section 6.3.2.3 [Fourier series], page 183) so we will be investigating these important operations first. It has

[9] Because we assumed the overlapping pixels outside the input image have a value of zero.

[10] ofcourse the sum of the kernel pixels still have to be unity in general.

become something of a cliché for people to say that the Fourier series "is a way to represent a (wave-like) function as the sum of simple sine waves" (from Wikipedia). However, sines themselves are abstract functions, so this statement really adds no extra layer of physical insight.

Before jumping head-first into the equations and proofs, we will begin with a historical background to see how the importance of frequencies actually roots in our ancient desire to see everything in terms of circles. A short review of how the complex plane should be interpreted is then given. Having paved the way with these two basics, we define the Fourier series and subsequently the Fourier transform. The final aim is to explain discrete Fourier transform, however some very important concepts need to be solidified first: The Dirac comb, convolution theorem and sampling theorem. So each of these topics are explained in their own separate sub-sub-section before going on to the discrete Fourier transform. Finally we revisit (after Section 6.3.1.2 [Edges in the spatial domain], page 178) the problem of convolution on the edges, but this time in the frequency domain. Understanding the sampling theorem and the discrete Fourier transform is very important in order to be able to pull out valuable science from the discrete image pixels. Therefore we have included the mathematical proofs and figures so you can have a clear understanding of these very important concepts.

6.3.2.1 Fourier series historical background

Ever since the ancient times, the circle has been (and still is) the simplest shape for abstract comprehension. All you need is a center point and a radius and you are done. All the points on a circle are at a fixed distance from the center. However, the moment you try to connect this elegantly simple and beautiful abstract construct (the circle) with the real world (for example compute its area or its circumference), things become really hard (ideally, impossible) because the irrational number π gets involved.

The key to understanding the Fourier series (thus the Fourier transform and finally the Discrete Fourier Transform) is our ancient desire to express everything in terms of circles or the most exceptionally simple and elegant abstract human construct. Most people prefer to say the same thing in a more ahistorical manner: to break a function into sines and cosines. As the term "ancient" in the previous sentence implies, Jean-Baptiste Joseph Fourier (1768 – 1830 A.D.) was not the first person to do this. The main reason we know this process by his name today is that he came up with an ingenious method to find the necessary coefficients (radius of) and frequencies ("speed" of rotation on) the circles for any generic (integrable) function.

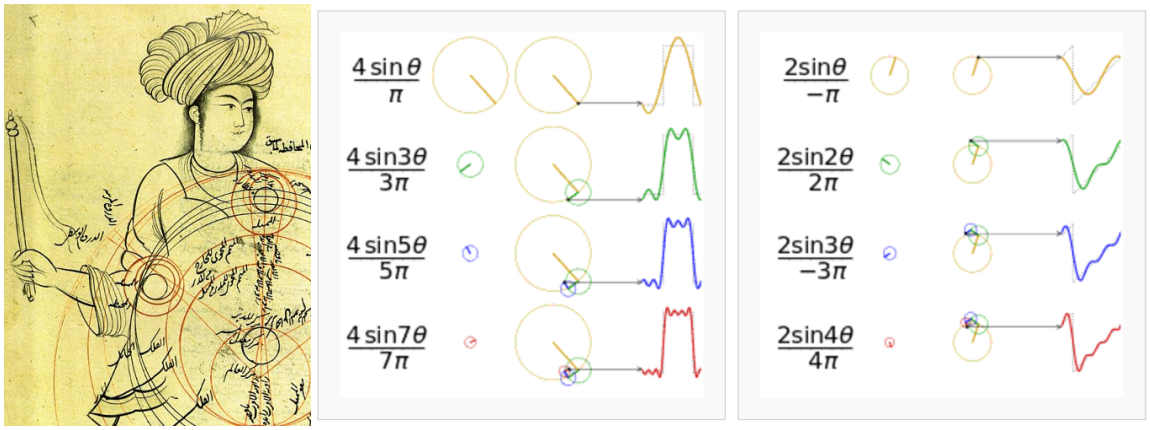


Figure 6.1: Epicycles and the Fourier series. Left: A demonstration of Mercury's epicycles relative to the "center of the world" by Qutb al-Din al-Shirazi (1236 – 1311 A.D.) retrieved from Wikipedia (`https://commons.wikimedia.org/wiki/File:Ghotb2.jpg`). Middle (`https://commons.wikimedia.org/wiki/File:Fourier_series_square_wave_circles_animation.gif`) and Right: How adding more epicycles (or terms in the Fourier series) will approximate functions. The right (`https://commons.wikimedia.org/wiki/File:Fourier_series_sawtooth_wave_circles_animation.gif`) animation is also available.

Like most aspects of mathematics, this process of interpreting everything in terms of circles, began for astronomical purposes. When astronomers noticed that the orbit of Mars and other outer planets, did not appear to be a simple circle (as everything should have been in the heavens). At some point during their orbit, the revolution of these planets would become slower, stop, go back a little (in what is known as the retrograde motion) and then continue going forward again.

The correction proposed by Ptolemy (90 – 168 A.D.) was the most agreed upon. He put the planets on Epicycles or circles whose center itself rotates on a circle whose center is the earth. Eventually, as observations became more and more precise, it was necessary to add more and more epicycles in order to explain the complex motions of the planets[11]. Figure 6.1(Left) shows an example depiction of the epicycles of Mercury in the late 13th century.

Of course we now know that if they had abdicated the Earth from its throne in the center of the heavens and allowed the Sun to take its place, everything would become much simpler and true. But there wasn't enough observational evidence for changing the "professional consensus" of the time to this radical view suggested by a small minority[12]. So the pre-

[11] See the Wikipedia page on "Deferent and epicycle" for a more complete historical review.

[12] Aristarchus of Samos (310 – 230 B.C.) appears to be one of the first people to suggest the Sun being in the center of the universe. This approach to science (that the standard model is defined by consensus) and the fact that this consensus might be completely wrong still applies equally well to our models of particle physics and cosmology today.

Galilean astronomers chose to keep Earth in the center and find a correction to the models (while keeping the heavens a purely "circular" order).

The main reason we are giving this historical background which might appear off topic is to give historical evidence that while such "approximations" do work and are very useful for pragmatic reasons (like measuring the calendar from the movement of astronomical bodies). They offer no physical insight. The astronomers who were involved with the Ptolemaic world view had to add a huge number of epicycles during the centuries after Ptolemy in order to explain more accurate observations. Finally the death knell of this world-view was Galileo's observations with his new instrument (the telescope). So the physical insight, which is what Astronomers and Physicists are interested in (as opposed to Mathematicians and Engineers who just like proving and optimizing or calculating!) comes from being creative and not limiting our selves to such approximations. Even when they work.

6.3.2.2 Circles and the complex plane

Before going onto the derivation, it is also useful to review how the complex numbers and their plane relate to the circles we talked about above. The two schematics in the middle and right of Figure 6.1 show how a 1D function of time can be made using the 2D real and imaginary surface. Seeing the animation in Wikipedia will really help in understanding this important concept. At each point in time, we take the vertical coordinate of the point and use it to find the value of the function at that point in time. Figure 6.2 shows this relation with the axes marked.

Leonhard Euler[13] (1707 – 1783 A.D.) showed that the complex exponential (e^{iv} where v is real) is periodic and can be written as: $e^{iv} = \cos v + isinv$. Therefore $e^{iv+2\pi} = e^{iv}$. Later, Caspar Wessel (mathematician and cartographer 1745 – 1818 A.D.) showed how complex numbers can be displayed as vectors on a plane. Euler's identity might seem counter intuitive at first, so we will try to explain it geometrically (for deeper physical insight). On the real-imaginary 2D plane (like the left hand plot in each box of Figure 6.2), multiplying a number by i can be interpreted as rotating the point by 90 degrees (for example the value 3 on the real axis becomes $3i$ on the imaginary axis). On the other hand, $e \equiv \lim_{n\to\infty}(1+\frac{1}{n})^n$, therefore, defining $m \equiv nu$, we get:

$$e^u = \lim_{n\to\infty}\left(1+\frac{1}{n}\right)^{nu} = \lim_{n\to\infty}\left(1+\frac{u}{nu}\right)^{nu} = \lim_{m\to\infty}\left(1+\frac{u}{m}\right)^{m}$$

Taking $u \equiv iv$ the result can be written as a generic complex number (a function of v):

$$e^{iv} = \lim_{m\to\infty}\left(1+i\frac{v}{m}\right)^{m} = a(v) + ib(v)$$

For $v = \pi$, a nice geometric animation of going to the limit can be seen on Wikipedia (`https://commons.wikimedia.org/wiki/File:ExpIPi.gif`). We see that $\lim_{m\to\infty} a(\pi) = -1$, while $\lim_{m\to\infty} b(\pi) = 0$, which

[13] Other forms of this equation were known before Euler. For example in 1707 A.D. (the year of Euler's birth) Abraham de Moivre (1667 – 1754 A.D.) showed that $(\cos x + i\sin x)^n = \cos(nx) + i\sin(nx)$. In 1714 A.D., Roger Cotes (1682 – 1716 A.D. a colleague of Newton who proofread the second edition of Principia) showed that: $ix = \ln(\cos x + i\sin x)$.

gives the famous $e^{i\pi} = -1$ equation. The final value is the real number -1, however the distance of the polygon points traversed as $m \to \infty$ is half the circumference of a circle or π, showing how v in the equation above can be interpreted as an angle in units of radians and therefore how $a(v) = cos(v)$ and $b(v) = sin(v)$.

Since e^{iv} is periodic (let's assume with a period of T), it is more clear to write it as $v \equiv \frac{2\pi n}{T}t$ (where n is an integer), so $e^{iv} = e^{i\frac{2\pi n}{T}t}$. The advantage of this notation is that the period (T) is clearly visible and the frequency ($\frac{2\pi n}{T}$, in units of 1/cycle) is defined through the integer n. In this notation, t is in units of "cycle"s.

As we see from the examples in Figure 6.1 and Figure 6.2, for each constituting frequency, we need a respective 'magnitude' or the radius of the circle in order to accurately approximate the desired 1D function. The concepts of "period" and "frequency" are relatively easy to grasp when using temporal units like time because this is how we define them in every-day life. However, in an image (astronomical data), we are dealing with spatial units like distance. Therefore, by one "period" we mean the *distance* at which the signal is identical and frequency is defined as the inverse of that spatial "period". The complex circle of Figure 6.2 can be thought of the Moon rotating about Earth which is rotating around the Sun; so the "Real (signal)" axis shows the Moon's position as seen by a distant observer on the Sun as time goes by. Because of the scalar (not having any direction or vector) nature of time, Figure 6.2 is easier to understand in units of time. When thinking about spatial units, mentally replace the "Time (sec)" axis with "Distance (meters)". Because length has direction and is a vector, visualizing the rotation of the imaginary circle and the advance along the "Distance (meters)" axis is not as simple as temporal units like time.

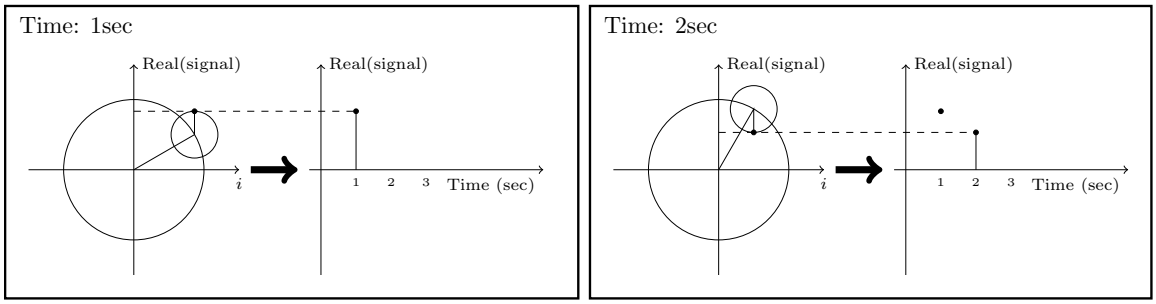


Figure 6.2: Relation between the real (signal), imaginary ($i \equiv \sqrt{-1}$) and time axes at two snapshots of time.

6.3.2.3 Fourier series

In astronomical images, our variable (brightness, or number of photo-electrons, or signal to be more generic) is recorded over the 2D spatial surface of a camera pixel. However to make things easier to understand, here we will assume that the signal is recorded in 1D (assume one row of the 2D image pixels). Also for this section and the next (Section 6.3.2.4 [Fourier transform], page 185) we will be talking about the signal before it is digitized or pixelated. Let's assume that we have the continuous function $f(l)$ which is integrable in the interval $[l_0, l_0 + L]$ (always true in practical cases like images). Take l_0 as the position of the first pixel in the assumed row of the image and L as the width of the image along that row. The

units of l_0 and L can be in any spatial units (for example meters) or an angular unit (like radians) multiplied by a fixed distance which is more common.

To approximate $f(l)$ over this interval, we need to find a set of frequencies and their corresponding 'magnitude's (see Section 6.3.2.2 [Circles and the complex plane], page 182). Therefore our aim is to show $f(l)$ as the following sum of periodic functions:

$$f(l) = \sum_{n=-\infty}^{\infty} c_n e^{i\frac{2\pi n}{L}l}$$

Note that the different frequencies ($2\pi n/L$, in units of cycles per meters for example) are not arbitrary. They are all integer multiples of the fundamental frequency of $\omega_0 = 2\pi/L$. Recall that L was the length of the signal we want to model. Therefore, we see that the smallest possible frequency (or the frequency resolution) in the end, depends on the length we observed the signal or L. In the case of each dimension on an image, this is the size of the image in the respective dimension. The frequencies have been defined in this "harmonic" fashion to insure that the final sum is periodic outside of the $[l_0, l_0 + L]$ interval too. At this point, you might be thinking that the sky is not periodic with the same period as my camera's view angle. You are absolutely right! The important thing is that since your camera's observed region is the only region we are "observing" and will be using, the rest of the sky is irrelevant; so we can safely assume the sky is periodic outside of it. However, this working assumption will haunt us later in Section 6.3.2.10 [Edges in the frequency domain], page 194.

The frequencies are thus determined by definition. So all we need to do is to find the coefficients (c_n), or magnitudes, or radii of the circles for each frequency which is identified with the integer n. Fourier's approach was to multiply both sides with a fixed term:

$$f(l)e^{-i\frac{2\pi m}{L}l} = \sum_{n=-\infty}^{\infty} c_n e^{i\frac{2\pi(n-m)}{L}l}$$

where $m > 0$[14]. We can then integrate both sides over the observation period:

$$\int_{l_0}^{l_0+L} f(l)e^{-i\frac{2\pi m}{L}l}dl = \int_{l_0}^{l_0+L} \sum_{n=-\infty}^{\infty} c_n e^{i\frac{2\pi(n-m)}{L}l}dl = \sum_{n=-\infty}^{\infty} c_n \int_{l_0}^{l_0+L} e^{i\frac{2\pi(n-m)}{L}l}dl$$

Both n and m are positive integers. Also, we know that a complex exponential is periodic so after one period (L) it comes back to its starting point. Therefore $\int_{l_0}^{l_0+L} e^{2\pi k/L}dl = 0$ for any $k > 0$. However, when $k = 0$, this integral becomes: $\int_{l_0}^{l_0+T} e^0 dt = \int_{l_0}^{l_0+T} dt = T$. Hence since the integral will be zero for all $n{\neq}m$, we get:

$$\sum_{n=-\infty}^{\infty} c_n \int_{l_0}^{l_0+T} e^{i\frac{2\pi(n-m)}{L}l}dl = Lc_m$$

[14] We could have assumed $m < 0$ and set the exponential to positive, but this is more clear.

The origin of the axis is fundamentally an arbitrary position. So let's set it to the start of the image such that $l_0 = 0$. So we can find the "magnitude" of the frequency $2\pi m/L$ within $f(l)$ through the relation:

$$c_m = \frac{1}{L}\int_0^L f(l)e^{-i\frac{2\pi m}{L}l}dl$$

6.3.2.4 Fourier transform

In Section 6.3.2.3 [Fourier series], page 183, we had to assume that the function is periodic outside of the desired interval with a period of L. Therefore, assuming that $L \to \infty$ will allow us to work with any function. However, with this approximation, the fundamental frequency (ω_0) or the frequency resolution that we discussed in Section 6.3.2.3 [Fourier series], page 183, will tend to zero: $\omega_0 \to 0$. In the equation to find c_m, every m represented a frequency (multiple of ω_0) and the integration on l removes the dependence of the right side of the equation on l, making it only a function of m or frequency. Let's define the following two variables:

$$\omega \equiv m\omega_0 = \frac{2\pi m}{L}$$

$$F(\omega) \equiv Lc_m$$

The equation to find the coefficients of each frequency in Section 6.3.2.3 [Fourier series], page 183, thus becomes:

$$F(\omega) = \int_{-\infty}^{\infty} f(l)e^{-i\omega l}dl.$$

The function $F(\omega)$ is thus the *Fourier transform* of $f(l)$ in the frequency domain. So through this transformation, we can find (analyze) the magnitudes of the constituting frequencies or the value in the frequency space[15] of our spatial input function. The great thing is that we can also do the reverse and later synthesize the input function from its Fourier transform. Let's do it: with the approximations above, multiply the right side of the definition of the Fourier Series (Section 6.3.2.3 [Fourier series], page 183) with $1 = L/L = (\omega_0 L)/(2\pi)$:

$$f(l) = \frac{1}{2\pi}\sum_{n=-\infty}^{\infty} Lc_n e^{\frac{2\pi i n}{L}l}\omega_0 = \frac{1}{2\pi}\sum_{n=-\infty}^{\infty} F(\omega)e^{i\omega l}\Delta\omega$$

To find the right most side of this equation, we renamed ω_0 as $\Delta\omega$ because it was our resolution, $2\pi n/L$ was written as ω and finally, Lc_n was written as $F(\omega)$ as we defined above. Now, as $L \to \infty$, $\Delta\omega \to 0$ so we can write:

[15] As we discussed before, this 'magnitude' can be interpreted as the radius of the circle rotating at this frequency in the epicyclic interpretation of the Fourier series, see Figure 6.1 and Figure 6.2.

$$f(l) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)e^{i\omega l}d\omega$$

Together, these two equations provide us with a very powerful set of tools that we can use to process (analyze) and recreate (synthesize) the input signal. Through the first equation, we can break up our input function into its constituent frequencies and analyze it, hence it is also known as *analysis*. Using the second equation, we can synthesize or make the input function from the known frequencies and their magnitudes. Thus it is known as *synthesis*. Here, we symbolize the Fourier transform (analysis) and its inverse (synthesis) of a function $f(l)$ and its Fourier Transform $F(\omega)$ as $\mathcal{F}[f]$ and $\mathcal{F}^{-1}[F]$.

6.3.2.5 Dirac delta and comb

The Dirac δ (delta) function (also known as an impulse) is the way that we convert a continuous function into a discrete one. It is defined to satisfy the following integral:

$$\int_{-\infty}^{\infty} \delta(l)dl = 1$$

When integrated with another function, it gives that function's value at $l = 0$:

$$\int_{-\infty}^{\infty} f(l)\delta(l)dt = f(0)$$

An impulse positioned at another point (say l_0) is written as $\delta(l - l_0)$:

$$\int_{-\infty}^{\infty} f(l)\delta(l - l_0)dt = f(l_0)$$

The Dirac δ function also operates similarly if we use summations instead of integrals. The Fourier transform of the delta function is:

$$\mathcal{F}[\delta(l)] = \int_{-\infty}^{\infty} \delta(l)e^{-i\omega l}dl = e^{-i\omega 0} = 1$$

$$\mathcal{F}[\delta(l - l_0)] = \int_{-\infty}^{\infty} \delta(l - l_0)e^{-i\omega l}dl = e^{-i\omega l_0}$$

From the definition of the Dirac δ we can also define a Dirac comb (III_P) or an impulse train with infinite impulses separated by P:

$$\mathrm{III}_P(l) \equiv \sum_{k=-\infty}^{\infty} \delta(l - kP)$$

P is chosen to represent "pixel width" later in Section 6.3.2.7 [Sampling theorem], page 189. Therefore the Dirac comb is periodic with a period of P. We have intentionally used a different name for the period of the Dirac comb compared to the input signal's length of observation that we showed with L in Section 6.3.2.3 [Fourier series], page 183. This difference is highlighted here to avoid confusion later when these two periods are needed together in Section 6.3.2.8 [Discrete Fourier transform], page 191. The Fourier transform of the Dirac comb will be necessary in Section 6.3.2.7 [Sampling theorem], page 189, so let's derive it. By its definition, it is periodic, with a period of P, so the Fourier coefficients of its Fourier Series (Section 6.3.2.3 [Fourier series], page 183) can be calculated within one period:

$$\text{III}_P = \sum_{n=-\infty}^{\infty} c_n e^{i\frac{2\pi n}{P}l}$$

We can now find the c_n from Section 6.3.2.3 [Fourier series], page 183:

$$c_n = \frac{1}{P}\int_{-P/2}^{P/2} \delta(l) e^{-i\frac{2\pi n}{P}l} = \frac{1}{P} \qquad \rightarrow \qquad \text{III}_P = \frac{1}{P}\sum_{n=-\infty}^{\infty} e^{i\frac{2\pi n}{P}l}$$

So we can write the Fourier transform of the Dirac comb as:

$$\mathcal{F}[\text{III}_P] = \int_{-\infty}^{\infty} \text{III}_P e^{-i\omega l} dl = \frac{1}{P}\sum_{n=-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-i(\omega-\frac{2\pi n}{P})l} dl = \frac{1}{P}\sum_{n=-\infty}^{\infty} \delta\left(\omega - \frac{2\pi n}{P}\right)$$

In the last step, we used the fact that the complex exponential is a periodic function, that n is an integer and that as we defined in Section 6.3.2.4 [Fourier transform], page 185, $\omega \equiv m\omega_0$, where m was an integer. The integral will be zero for any ω that is not equal to $2\pi n/P$, a more complete explanation can be seen in Section 6.3.2.3 [Fourier series], page 183. Therefore, while in the spatial domain the impulses had spacing of P (meters for example), in the frequency space, the spacing between the different impulses are $2\pi/P$ cycles per meters.

6.3.2.6 Convolution theorem

The convolution (shown with the $*$ operator) of the two functions $f(l)$ and $h(l)$ is defined as:

$$c(l) \equiv [f*h](l) = \int_{-\infty}^{\infty} f(\tau)h(l-\tau)d\tau$$

See Section 6.3.1.1 [Convolution process], page 178, for a more detailed physical (pixel based) interpretation of this definition. The Fourier transform of convolution ($C(\omega)$) can be written as:

$$C(\omega) = \int_{-\infty}^{\infty} [f*h](l)e^{-i\omega l}dl = \int_{-\infty}^{\infty} f(\tau)\left[\int_{-\infty}^{\infty} h(l-\tau)e^{-i\omega l}dl\right]d\tau$$

To solve the inner integral, let's define $s \equiv l - \tau$, so that $ds = dl$ and $l = s + \tau$ then the inner integral becomes:

$$\int_{-\infty}^{\infty} h(l-\tau)e^{-i\omega l}dl = \int_{-\infty}^{\infty} h(s)e^{-i\omega(s+\tau)}ds = e^{-i\omega\tau}\int_{-\infty}^{\infty} h(s)e^{-i\omega s}ds = H(\omega)e^{-i\omega\tau}$$

where $H(\omega)$ is the Fourier transform of $h(l)$. Substituting this result for the inner integral above, we get:

$$C(\omega) = H(\omega)\int_{-\infty}^{\infty} f(\tau)e^{-i\omega\tau}d\tau = H(\omega)F(\omega) = F(\omega)H(\omega)$$

where $F(\omega)$ is the Fourier transform of $f(l)$. So multiplying the Fourier transform of two functions individually, we get the Fourier transform of their convolution. The convolution theorem also proves a relation between the convolutions in the frequency space. Let's define:

$$D(\omega) \equiv F(\omega) * H(\omega)$$

Applying the inverse Fourier Transform or synthesis equation (Section 6.3.2.4 [Fourier transform], page 185) to both sides and following the same steps above, we get:

$$d(l) = f(l)h(l)$$

Where $d(l)$ is the inverse Fourier transform of $D(\omega)$. We can therefore re-write the two equations above formally as the convolution theorem:

$$\mathcal{F}[f * h] = \mathcal{F}[f]\mathcal{F}[h]$$

$$\mathcal{F}[fh] = \mathcal{F}[f] * \mathcal{F}[h]$$

Besides its usefulness in blurring an image by convolving it with a given kernel, the convolution theorem also enables us to do another very useful operation in data analysis: to match the blur (or PSF) between two images taken with different telescopes/cameras or under different atmospheric conditions. This process is also known as de-convolution. Let's take $f(l)$ as the image with a narrower PSF (less blurry) and $c(l)$ as the image with a wider PSF which appears more blurred. Also let's take $h(l)$ to represent the kernel that should be convolved with the sharper image to create the more blurry image. Above, we proved the relation between these three images through the convolution theorem. But there, we assumed that $f(l)$ and $h(l)$ are known (given) and the convolved image is desired.

In de-convolution, we have $f(l)$ –the sharper image– and $f*h(l)$ –the more blurry image– and we want to find the kernel $h(l)$. The solution is a direct result of the convolution theorem:

$$\mathcal{F}[h] = \frac{\mathcal{F}[f*h]}{\mathcal{F}[f]} \qquad \text{or} \qquad h(l) = \mathcal{F}^{-1}\left[\frac{\mathcal{F}[f*h]}{\mathcal{F}[f]}\right]$$

While this works really nice, it has two problems:

- If $\mathcal{F}[f]$ has any zero values, then the inverse Fourier transform will not be a number!
- If there is significant noise in the image, then the high frequencies of the noise are going to significantly reduce the quality of the final result.

A standard solution to both these problems is the Weiner de-convolution algorithm[16].

6.3.2.7 Sampling theorem

Our mathematical functions are continuous, however, our data collecting and measuring tools are discrete. Here we want to give a mathematical formulation for digitizing the continuous mathematical functions so that later, we can retrieve the continuous function from the digitized recorded input. Assuming that we have a continuous function $f(l)$, then we can define $f_s(l)$ as the 'sampled' $f(l)$ through the Dirac comb (see Section 6.3.2.5 [Dirac delta and comb], page 186):

$$f_s(l) = f(l)\text{III}_P = \sum_{n=-\infty}^{\infty} f(l)\delta(l - nP)$$

The discrete data-element f_k (for example, a pixel in an image), where k is an integer, can thus be represented as:

$$f_k = \int_{-\infty}^{\infty} f_s(l)dl = \int_{-\infty}^{\infty} f(l)\delta(l - kP)dt = f(kP)$$

Note that in practice, our discrete data points are not found in this fashion. Each detector pixel (in an image for example) has an area and averages the signal it receives over that area, not a mathematical point as the Dirac δ function defines. However, as long as the variation in the signal over one detector pixel is not significant, this can be a good approximation. Having put this issue to the side, we can now try to find the relation between the Fourier transforms of the un-sampled $f(l)$ and the sampled $f_s(l)$. For a more clear notation, let's define:

$$F_s(\omega) \equiv \mathcal{F}[f_s]$$

$$D(\omega) \equiv \mathcal{F}[\text{III}_P]$$

[16] https://en.wikipedia.org/wiki/Wiener_deconvolution

Then using the Convolution theorem (see Section 6.3.2.6 [Convolution theorem], page 187), $F_s(\omega)$ can be written as:

$$F_s(\omega) = \mathcal{F}[f(l)\text{III}_P] = F(\omega) * D(\omega)$$

Finally, from the definition of convolution and the Fourier transform of the Dirac comb (see Section 6.3.2.5 [Dirac delta and comb], page 186), we get:

$$\begin{aligned} F_s(\omega) &= \int_{-\infty}^{\infty} F(\omega)D(\omega-\mu)d\mu \\ &= \frac{1}{P}\sum_{n=-\infty}^{\infty}\int_{-\infty}^{\infty} F(\omega)\delta\left(\omega-\mu-\frac{2\pi n}{P}\right)d\mu \\ &= \frac{1}{P}\sum_{n=-\infty}^{\infty} F\left(\omega-\frac{2\pi n}{P}\right). \end{aligned}$$

$F(\omega)$ was only a simple function, see Figure 6.3(left). However, from the sampled Fourier transform function we see that $F_s(\omega)$ is the superposition of infinite copies of $F(\omega)$ that have been shifted, see Figure 6.3(right). From the equation, it is clear that the shift in each copy is $2\pi/P$.

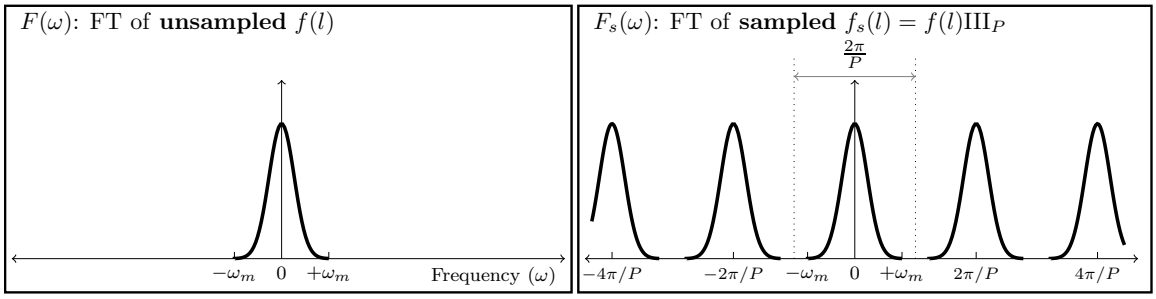


Figure 6.3: Sampling causes infinite repetition in the frequency domain. FT is an abbreviation for 'Fourier transform'. ω_m represents the maximum frequency present in the input. $F(\omega)$ is only symmetric on both sides of 0 when the input is real (not complex). In general $F(\omega)$ is complex and thus cannot be simply plotted like this. Here we have assumed a real Gaussian $f(t)$ which has produced a Gaussian $F(\omega)$.

The input $f(l)$ can have any distribution of frequencies in it. In the example of Figure 6.3(left), the input consisted of a range of frequencies equal to $\Delta\omega = 2\omega_m$. Fortunately as Figure 6.3(right) shows, the assumed pixel size (P) we used to sample this hypothetical function was such that $2\pi/P > \Delta\omega$. The consequence is that each copy of $F(\omega)$ has become completely separate from the surrounding copies. Such a digitized (sampled) data set is thus called *over-sampled*. When $2\pi/P = \Delta\omega$, P is just small enough to finely separate even the largest frequencies in the input signal and thus it is known as *critically-sampled*. Finally if $2\pi/P < \Delta\omega$ we are dealing with an *under-sampled* data

set. In an under-sampled data set, the separate copies of $F(\omega)$ are going to overlap and this will deprive us of recovering high constituent frequencies of $f(l)$. The effects of under-sampling in an image with high rates of change (for example a brick wall imaged from a distance) can clearly be visually seen and is known as *aliasing*.

When the input $f(l)$ is composed of a finite range of frequencies, $f(l)$ is known as a *band-limited* function. The example in Figure 6.3(left) was a nice demonstration of such a case: for all $\omega < -\omega_m$ or $\omega > \omega_m$, we have $F(\omega) = 0$. Therefore, when the input function is band-limited and our detector's pixels are placed such that we have critically (or over-) sampled it, then we can exactly reproduce the continuous $f(l)$ from the discrete or digitized samples. To do that, we just have to isolate one copy of $F(\omega)$ from the infinite copies and take its inverse Fourier transform.

This ability to exactly reproduce the continuous input from the sampled or digitized data leads us to the *sampling theorem* which connects the inherent property of the continuous signal (its maximum frequency) to that of the detector (the spacing between its pixels). The sampling theorem states that the full (continuous) signal can be recovered when the pixel size (P) and the maximum constituent frequency in the signal (ω_m) have the following relation[17]:

$$\frac{2\pi}{P} > 2\omega_m$$

This relation was first formulated by Harry Nyquist (1889 – 1976 A.D.) in 1928 and formally proved in 1949 by Claude E. Shannon (1916 – 2001 A.D.) in what is now known as the Nyquist-Shannon sampling theorem. In signal processing, the signal is produced (synthesized) by a transmitter and is received and de-coded (analyzed) by a receiver. Therefore producing a band-limited signal is necessary.

In astronomy, we do not produce the shapes of our targets, we are only observers. Galaxies can have any shape and size, therefore ideally, our signal is not band-limited. However, since we are always confined to observing through an aperture, the aperture will cause a point source (for which $\omega_m = \infty$) to be spread over several pixels. This spread is quantitatively known as the point spread function or PSF. This spread does blur the image which is undesirable; however, for this analysis it produces the positive outcome that there will be a finite ω_m. Though we should caution that any detector will have noise which will add lots of very high frequency (ideally infinite) changes between the pixels. However, the coefficients of those noise frequencies are usually exceedingly small.

6.3.2.8 Discrete Fourier transform

As we have stated several times so far, the input image is a digitized, pixelated or discrete array of values ($f_s(l)$, see Section 6.3.2.7 [Sampling theorem], page 189). The input is not a continuous function. Also, all our numerical calculations can only be done on a sampled, or discrete Fourier transform. Note that $F_s(\omega)$ is not discrete, it is continuous. One way would be to find the analytic $F_s(\omega)$, then sample it at any desired "freq-pixel"[18] spacing. However,

[17] This equation is also shown in some places without the 2π. Whether 2π is included or not depends on how you define the frequency

[18] We are using the made-up word "freq-pixel" so they are not confused with spatial domain "pixels".

this process would involve two steps of operations and computers in particular are not too good at analytic operations for the first step. So here, we will derive a method to directly find the 'freq-pixel'ated $F_s(\omega)$ from the pixelated $f_s(l)$. Let's start with the definition of the Fourier transform (see Section 6.3.2.4 [Fourier transform], page 185):

$$F_s(\omega) = \int_{-\infty}^{\infty} f_s(l) e^{-i\omega l} dl$$

From the definition of $f_s(\omega)$ (using x instead of n) we get:

$$\begin{aligned} F_s(\omega) &= \sum_{x=-\infty}^{\infty} \int_{-\infty}^{\infty} f(l)\delta(l - xP)e^{-i\omega l} dl \\ &= \sum_{x=-\infty}^{\infty} f_x e^{-i\omega x P} \end{aligned}$$

Where f_x is the value of $f(l)$ on the point x or the value of the xth pixel. As shown in Section 6.3.2.7 [Sampling theorem], page 189, this function is infinitely periodic with a period of $2\pi/P$. So all we need is the values within one period: $0 < \omega < 2\pi/P$, see Figure 6.3. We want X samples within this interval, so the frequency difference between each frequency sample or freq-pixel is $1/XP$. Hence we will evaluate the equation above on the points at:

$$\omega = \frac{u}{XP} \qquad u = 0, 1, 2, ..., X - 1$$

Therefore the value of the freq-pixel u in the frequency domain is:

$$F_u = \sum_{x=0}^{X-1} f_x e^{-i\frac{ux}{X}}$$

Therefore, we see that for each freq-pixel in the frequency domain, we are going to need all the pixels in the spatial domain[19]. If the input (spatial) pixel row is also X pixels wide, then we can exactly recover the xth pixel with the following summation:

$$f_x = \frac{1}{X} \sum_{u=0}^{X-1} F_u e^{i\frac{ux}{X}}$$

When the input pixel row (we are still only working on 1D data) has X pixels, then it is $L = XP$ spatial units wide. L, or the length of the input data was defined in Section 6.3.2.3 [Fourier series], page 183, and P or the space between the pixels in the input was defined in Section 6.3.2.5 [Dirac delta and comb], page 186. As we saw in Section 6.3.2.7 [Sampling theorem], page 189, the input (spatial) pixel spacing (P) specifies the range of frequencies

[19] So even if one pixel is a blank pixel (see Section 6.1.3 [Blank pixels], page 154), all the pixels in the frequency domain will also be blank.

that can be studied and in Section 6.3.2.3 [Fourier series], page 183, we saw that the length of the (spatial) input, (L) determines the resolution (or size of the freq-pixels) in our discrete Fourier transformed image. Both result from the fact that the frequency domain is the inverse of the spatial domain.

6.3.2.9 Fourier operations in two dimensions

Once all the relations in the previous sections have been clearly understood in one dimension, it is very easy to generalize them to two or even more dimensions since each dimension is by definition independent. Previously we defined l as the continuous variable in 1D and the inverse of the period in its direction to be ω. Let's show the second spatial direction with m the the inverse of the period in the second dimension with ν. The Fourier transform in 2D (see Section 6.3.2.4 [Fourier transform], page 185) can be written as:

$$F(\omega,\nu)=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}f(l,m)e^{-i(\omega l+\nu m)}dl$$

$$f(l,m)=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}F(\omega,\nu)e^{i(\omega l+\nu m)}dl$$

The 2D Dirac $\delta(l,m)$ is non-zero only when $l=m=0$. The 2D Dirac comb (or Dirac brush! See Section 6.3.2.5 [Dirac delta and comb], page 186) can be written in units of the 2D Dirac δ. For most image detectors, the sides of a pixel are equal in both dimensions. So P remains unchanged, if a specific device is used which has non-square pixels, then for each dimension a different value should be used.

$$\mathrm{III}_P(l,m)\equiv\sum_{j=-\infty}^{\infty}\sum_{k=-\infty}^{\infty}\delta(l-jP,m-kP)$$

The Two dimensional Sampling theorem (see Section 6.3.2.7 [Sampling theorem], page 189) is thus very easily derived as before since the frequencies in each dimension are independent. Let's take ν_m as the maximum frequency along the second dimension. Therefore the two dimensional sampling theorem says that a 2D band-limited function can be recovered when the following conditions hold[20]:

$$\frac{2\pi}{P}>2\omega_m \qquad \text{and} \qquad \frac{2\pi}{P}>2\nu_m$$

Finally, let's represent the pixel counter on the second dimension in the spatial and frequency domains with y and v respectively. Also let's assume that the input image has Y pixels on the second dimension. Then the two dimensional discrete Fourier transform and its inverse (see Section 6.3.2.8 [Discrete Fourier transform], page 191) can be written as:

[20] If the pixels are not a square, then each dimension has to use the respective pixel size, but since most detectors have square pixels, we assume so here too

$$F_{u,v} = \sum_{x=0}^{X-1} \sum_{y=0}^{Y-1} f_{x,y} e^{-i(\frac{ux}{X} + \frac{vy}{Y})}$$

$$f_{x,y} = \frac{1}{XY} \sum_{u=0}^{X-1} \sum_{v=0}^{Y-1} F_{u,v} e^{i(\frac{ux}{X} + \frac{vy}{Y})}$$

6.3.2.10 Edges in the frequency domain

With a good grasp of the frequency domain, we can revisit the problem of convolution on the image edges, see Section 6.3.1.2 [Edges in the spatial domain], page 178. When we apply the convolution theorem (see Section 6.3.2.6 [Convolution theorem], page 187) to convolve an image, we first take the discrete Fourier transforms (DFT, Section 6.3.2.8 [Discrete Fourier transform], page 191) of both the input image and the kernel, then we multiply them with each other and then take the inverse DFT to construct the convolved image. Of course, in order to multiply them with each other in the frequency domain, the two images have to be the same size, so let's assume that we pad the kernel (it is usually smaller than the input image) with zero valued pixels in both dimensions so it becomes the same size as the input image before the DFT.

Having multiplied the two DFTs, we now apply the inverse DFT which is where the problem is usually created. If the DFT of the kernel only had values of 1 (unrealistic condition!) then there would be no problem and the inverse DFT of the multiplication would be identical with the input. However in real situations, the kernel's DFT has a maximum of 1 (because the sum of the kernel has to be one, see Section 6.3.1.1 [Convolution process], page 178) and decreases something like the hypothetical profile of Figure 6.3. So when multiplied with the input image's DFT, the coefficients or magnitudes (see Section 6.3.2.2 [Circles and the complex plane], page 182) of the smallest frequency (or the sum of the input image pixels) remains unchanged, while the magnitudes of the higher frequencies are significantly reduced.

As we saw in Section 6.3.2.7 [Sampling theorem], page 189, the Fourier transform of a discrete input will be infinitely repeated. In the final inverse DFT step, the input is in the frequency domain (the multiplied DFT of the input image and the kernel DFT). So the result (our output convolved image) will be infinitely repeated in the spatial domain. In order to accurately reconstruct the input image, we need all the frequencies with the correct magnitudes. However, when the magnitudes of higher frequencies are decreased, longer periods (shorter frequencies) will dominate in the reconstructed pixel values. Therefore, when constructing a pixel on the edge of the image, the newly empowered longer periods will look beyond the input image edges and will find the repeated input image there. So if you convolve an image in this fashion using the convolution theorem, when a bright object exists on one edge of the image, its blurred wings will be present on the other side of the convolved image. This is often termed as circular convolution or cyclic convolution.

So, as long as we are dealing with convolution in the frequency domain, there is nothing we can do about the image edges. The least we can do is to eliminate the ghosts of the other side of the image. So, we add zero valued pixels to both the input image and the kernel in both dimensions so the image that will be convolved has a size equal to the sum of both

images in each dimension. Of course, the effect of this zero-padding is that the sides of the output convolved image will become dark. To put it another way, the edges are going to drain the flux from nearby objects. But at least it is consistent across all the edges of the image and is predictable. In Convolve, you can see the padded images when inspecting the frequency domain convolution steps with the `--viewfreqsteps` option.

6.3.3 Spatial vs. Frequency domain

With the discussions above it might not be clear when to choose the spatial domain and when to choose the frequency domain. Here we will try to list the benefits of each.

The spatial domain,

- Can correct for the edge effects of convolution, see Section 6.3.1.2 [Edges in the spatial domain], page 178.
- Can operate on blank pixels.
- Can be faster than frequency domain when the kernel is small (in terms of the number of pixels on the sides).

The frequency domain,

- Will be much faster when the image and kernel are both large.

As a general rule of thumb, when working on an image of modeled profiles use the frequency domain and when working on an image of real (observed) objects use the spatial domain (corrected for the edges). The reason is that if you apply a frequency domain convolution to a real image, you are going to loose information on the edges and generally you don't want large kernels. But when you have made the profiles in the image yourself, you can just make a larger input image and crop the central parts to completely remove the edge effect, see Section 8.1.2 [If convolving afterwards], page 289. Also due to oversampling, both the kernels and the images can become very large and the speed boost of frequency domain convolution will significantly improve the processing time, see Section 8.1.1.6 [Oversampling], page 289.

6.3.4 Convolution kernel

All the programs that need convolution will need to be given a convolution kernel file and extension. In most cases (other than Convolve, see Section 6.3 [Convolve], page 177) the kernel file name is optional. However, the extension is necessary and must be specified either on the command-line or at least one of the configuration files (see Section 4.2 [Configuration files], page 106). Within Gnuastro, there are two ways to create a kernel image:

- MakeProfiles: You can use MakeProfiles to create a parametric (based on a radial function) kernel, see Section 8.1 [MakeProfiles], page 284. By default MakeProfiles will make the Gaussian and Moffat profiles in a separate file so you can feed it into any of the programs.
- ConvertType: You can write your own desired kernel into a text file table and convert it to a FITS file with ConvertType, see Section 5.2 [ConvertType], page 138. Just be careful that the kernel has to have an odd number of pixels along its two axes, see Section 6.3.1.1 [Convolution process], page 178. All the programs that do convolution will normalize the kernel internally, so if you choose this option, you don't have to worry about normalizing the kernel. Only within Convolve, there is an option to disable normalization, see Section 6.3.5 [Invoking Convolve], page 196.

The two options to specify a kernel file name and its extension are shown below. These are common between all the programs that will do convolution.

`-k STR`
`--kernel=STR`
The convolution kernel file name. The `BITPIX` (data type) value of this file can be any standard type and it does not necessarily have to be normalized. Several operations will be done on the kernel image prior to the program's processing:

- It will be converted to floating point type.
- All blank pixels (see Section 6.1.3 [Blank pixels], page 154) will be set to zero.
- It will be normalized so the sum of its pixels equal unity.
- It will be flipped so the convolved image has the same orientation. This is only relevant if the kernel is not circular. See Section 6.3.1.1 [Convolution process], page 178.

`-U STR`
`--khdu=STR`
The convolution kernel HDU. Although the kernel file name is optional, before running any of the programs, they need to have a value for `--khdu` even if the default kernel is to be used. So be sure to keep its value in at least one of the configuration files (see Section 4.2 [Configuration files], page 106). By default, the system configuration file has a value.

6.3.5 Invoking Convolve

Convolve an input dataset (2D image or 1D spectrum for example) with a known kernel, or make the kernel necessary to match two PSFs. The general template for Convolve is:

```
$ astconvolve [OPTION...] ASTRdata
```

One line examples:

```
## Convolve mockimg.fits with psf.fits:
$ astconvolve --kernel=psf.fits mockimg.fits

## Convolve in the spatial domain:
$ astconvolve observedimg.fits --kernel=psf.fits --domain=spatial

## Find the kernel to match sharper and blurry PSF images:
$ astconvolve --kernel=sharperimage.fits --makekernel=10          \
              blurryimage.fits

## Convolve a Spectrum (column 14 in the FITS table below) with a
## custom kernel (the kernel will be normalized internally, so only
## the ratios are important). Sed is used to replace the spaces with
## new line characters so Convolve sees them as values in one column.
$ echo "1 3 10 3 1" | sed 's/ /\n/g' | astconvolve spectra.fits -c14
```

The only argument accepted by Convolve is an input image file. Some of the options are the same between Convolve and some other Gnuastro programs. Therefore, to avoid

repetition, they will not be repeated here. For the full list of options shared by all Gnuastro programs, please see Section 4.1.2 [Common options], page 95. In particular, in the spatial domain, on a multi-dimensional datasets, convolve uses Gnuastro's tessellation to speed up the run, see Section 4.7 [Tessellation], page 123. Common options related to tessellation are described in in Section 4.1.2.2 [Processing options], page 98.

1-dimensional datasets (for example spectra) are only read as columns within a table (see Section 4.6 [Tables], page 117, for more on how Gnuastro programs read tables). Note that currently 1D convolution is only implemented in the spatial domain and thus kernel-matching is also not supported.

Here we will only explain the options particular to Convolve. Run Convolve with `--help` in order to see the full list of options Convolve accepts, irrespective of where they are explained in this book.

`--kernelcolumn`
: Column containing the 1D kernel. When the input dataset is a 1-dimensional column, and the host table has more than one column, use this option to specify which column should be used.

`--nokernelflip`
: Do not flip the kernel after reading it the spatial domain convolution. This can be useful if the flipping has already been applied to the kernel.

`--nokernelnorm`
: Do not normalize the kernel after reading it, such that the sum of its pixels is unity.

`-d STR`
`--domain=STR`
: The domain to use for the convolution. The acceptable values are '`spatial`' and '`frequency`', corresponding to the respective domain.

  For large images, the frequency domain process will be more efficient than convolving in the spatial domain. However, the edges of the image will loose some flux (see Section 6.3.1.2 [Edges in the spatial domain], page 178) and the image must not contain any blank pixels, see Section 6.3.3 [Spatial vs. Frequency domain], page 195.

`--checkfreqsteps`
: With this option a file with the initial name of the output file will be created that is suffixed with `_freqsteps.fits`, all the steps done to arrive at the final convolved image are saved as extensions in this file. The extensions in order are:

  1. The padded input image. In frequency domain convolution the two images (input and convolved) have to be the same size and both should be padded by zeros.
  2. The padded kernel, similar to the above.
  3. The Fourier spectrum of the forward Fourier transform of the input image. Note that the Fourier transform is a complex operation (and not view able in one image!) So we either have to show the 'Fourier spectrum' or the

'Phase angle'. For the complex number $a + ib$, the Fourier spectrum is defined as $\sqrt{a^2 + b^2}$ while the phase angle is defined as $\arctan(b/a)$.

4. The Fourier spectrum of the forward Fourier transform of the kernel image.
5. The Fourier spectrum of the multiplied (through complex arithmetic) transformed images.
6. The inverse Fourier transform of the multiplied image. If you open it, you will see that the convolved image is now in the center, not on one side of the image as it started with (in the padded image of the first extension). If you are working on a mock image which originally had pixels of precisely 0.0, you will notice that in those parts that your convolved profile(s) did not convert, the values are now $\sim 10^{-18}$, this is due to floating-point round off errors. Therefore in the final step (when cropping the central parts of the image), we also remove any pixel with a value less than 10^{-17}.

`--noedgecorrection`
: Do not correct the edge effect in spatial domain convolution. For a full discussion, please see Section 6.3.1.2 [Edges in the spatial domain], page 178.

`-m INT`
`--makekernel=INT`
: (`=INT`) If this option is called, Convolve will do de-convolution (see Section 6.3.2.6 [Convolution theorem], page 187). The image specified by the `--kernel` option is assumed to be the sharper (less blurry) image and the input image is assumed to be the more blurry image. The value given to this option will be used as the maximum radius of the kernel. Any pixel in the final kernel that is larger than this distance from the center will be set to zero. The two images must have the same size.

  Noise has large frequencies which can make the result less reliable for the higher frequencies of the final result. So all the frequencies which have a spectrum smaller than the value given to the `minsharpspec` option in the sharper input image are set to zero and not divided. This will cause the wings of the final kernel to be flatter than they would ideally be which will make the convolved image result unreliable if it is too high. Some notes to take into account for a good result:

  - Choose a bright (unsaturated) star and use a region box (with Crop for example, see Section 6.1 [Crop], page 151) that is sufficiently above the noise.
  - Use Warp (see Section 6.4 [Warp], page 199) to warp the pixel grid so the star's center is exactly on the center of the central pixel in the cropped image. This will certainly slightly degrade the result, however, it is necessary. If there are multiple good stars, you can shift all of them, then normalize them (so the sum of each star's pixels is one) and then take their average to decrease this effect.
  - The shifting might move the center of the star by one pixel in any direction, so crop the central pixel of the warped image to have a clean image for the de-convolution.

Note that this feature is not yet supported in 1-dimensional datasets.

`-c`
`--minsharpspec`
(`=FLT`) The minimum frequency spectrum (or coefficient, or pixel value in the frequency domain image) to use in deconvolution, see the explanations under the `--makekernel` option for more information.

6.4 Warp

Image warping is the process of mapping the pixels of one image onto a new pixel grid. This process is sometimes known as transformation, however following the discussion of Heckbert 1989[21] we will not be using that term because it can be confused with only pixel value or flux transformations. Here we specifically mean the pixel grid transformation which is better conveyed with 'warp'.

Image wrapping is a very important step in astronomy, both in observational data analysis and in simulating modeled images. In modeling, warping an image is necessary when we want to apply grid transformations to the initial models, for example in simulating gravitational lensing (Radial warpings are not yet included in Warp). Observational reasons for warping an image are listed below:

- **Noise:** Most scientifically interesting targets are inherently faint (have a very low Signal to noise ratio). Therefore one short exposure is not enough to detect such objects that are drowned deeply in the noise. We need multiple exposures so we can add them together and increase the objects' signal to noise ratio. Keeping the telescope fixed on one field of the sky is practically impossible. Therefore very deep observations have to put into the same grid before adding them.
- **Resolution:** If we have multiple images of one patch of the sky (hopefully at multiple orientations) we can warp them to the same grid. The multiple orientations will allow us to 'guess' the values of pixels on an output pixel grid that has smaller pixel sizes and thus increase the resolution of the output. This process of merging multiple observations is known as Mosaicing.
- **Cosmic rays:** Cosmic rays can randomly fall on any part of an image. If they collide vertically with the camera, they are going to create a very sharp and bright spot that in most cases can be separated easily[22]. However, depending on the depth of the camera pixels, and the angle that a cosmic rays collides with it, it can cover a line-like larger area on the CCD which makes the detection using their sharp edges very hard and error prone. One of the best methods to remove cosmic rays is to compare multiple images of the same field. To do that, we need all the images to be on the same pixel grid.
- **Optical distortion:** (Not yet included in Warp) In wide field images, the optical distortion that occurs on the outer parts of the focal plane will make accurate comparison of the objects at various locations impossible. It is therefore necessary to warp the image and correct for those distortions prior to the analysis.

[21] Paul S. Heckbert. 1989. *Fundamentals of Texture mapping and Image Warping*, Master's thesis at University of California, Berkeley.

[22] All astronomical targets are blurred with the PSF, see Section 8.1.1.2 [Point spread function], page 285, however a cosmic ray is not and so it is very sharp (it suddenly stops at one pixel).

- **Detector not on focal plane:** In some cases (like the Hubble Space Telescope ACS and WFC3 cameras), the CCD might be tilted compared to the focal plane, therefore the recorded CCD pixels have to be projected onto the focal plane before further analysis.

6.4.1 Warping basics

Let's take $[\,u \quad v\,]$ as the coordinates of a point in the input image and $[\,x \quad y\,]$ as the coordinates of that same point in the output image[23]. The simplest form of coordinate transformation (or warping) is the scaling of the coordinates, let's assume we want to scale the first axis by M and the second by N, the output coordinates of that point can be calculated by

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} Mu \\ Nv \end{bmatrix} = \begin{bmatrix} M & 0 \\ 0 & N \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

Note that these are matrix multiplications. We thus see that we can represent any such grid warping as a matrix. Another thing we can do with this 2×2 matrix is to rotate the output coordinate around the common center of both coordinates. If the output is rotated anticlockwise by θ degrees from the positive (to the right) horizontal axis, then the warping matrix should become:

$$\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ucos\theta - vsin\theta \\ usin\theta + vcos\theta \end{bmatrix} = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

We can also flip the coordinates around the first axis, the second axis and the coordinate center with the following three matrices respectively:

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \qquad \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}$$

The final thing we can do with this definition of a 2×2 warping matrix is shear. If we want the output to be sheared along the first axis with A and along the second with B, then we can use the matrix:

$$\begin{bmatrix} 1 & A \\ B & 1 \end{bmatrix}$$

To have one matrix representing any combination of these steps, you use matrix multiplication, see Section 6.4.2 [Merging multiple warpings], page 202. So any combinations of these transformations can be displayed with one 2×2 matrix:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

[23] These can be any real number, we are not necessarily talking about integer pixels here.

The transformations above can cover a lot of the needs of most coordinate transformations. However they are limited to mapping the point $[\,0 \quad 0\,]$ to $[\,0 \quad 0\,]$. Therefore they are useless if you want one coordinate to be shifted compared to the other one. They are also space invariant, meaning that all the coordinates in the image will receive the same transformation. In other words, all the pixels in the output image will have the same area if placed over the input image. So transformations which require varying output pixel sizes like projections cannot be applied through this 2×2 matrix either (for example for the tilted ACS and WFC3 camera detectors on board the Hubble space telescope).

To add these further capabilities, namely translation and projection, we use the homogeneous coordinates. They were defined about 200 years ago by August Ferdinand Möbius (1790 – 1868). For simplicity, we will only discuss points on a 2D plane and avoid the complexities of higher dimensions. We cannot provide a deep mathematical introduction here, interested readers can get a more detailed explanation from Wikipedia[24] and the references therein.

By adding an extra coordinate to a point we can add the flexibility we need. The point $[\,x \quad y\,]$ can be represented as $[\,xZ \quad yZ \quad Z\,]$ in homogeneous coordinates. Therefore multiplying all the coordinates of a point in the homogeneous coordinates with a constant will give the same point. Put another way, the point $[\,x \quad y \quad Z\,]$ corresponds to the point $[\,x/Z \quad y/Z\,]$ on the constant Z plane. Setting $Z = 1$, we get the input image plane, so $[\,u \quad v \quad 1\,]$ corresponds to $[\,u \quad v\,]$. With this definition, the transformations above can be generally written as:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

We thus acquired 4 extra degrees of freedom. By giving non-zero values to the zero valued elements of the last column we can have translation (try the matrix multiplication!). In general, any coordinate transformation that is represented by the matrix below is known as an affine transformation[25]:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix}$$

We can now consider translation, but the affine transform is still spatially invariant. Giving non-zero values to the other two elements in the matrix above gives us the projective transformation or Homography[26] which is the most general type of transformation with the 3×3 matrix:

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

[24] http://en.wikipedia.org/wiki/Homogeneous_coordinates

[25] http://en.wikipedia.org/wiki/Affine_transformation

[26] http://en.wikipedia.org/wiki/Homography

So the output coordinates can be calculated from:

$$x = \frac{x'}{w} = \frac{au + bv + c}{gu + hv + 1} \qquad y = \frac{y'}{w} = \frac{du + ev + f}{gu + hv + 1}$$

Thus with Homography we can change the sizes of the output pixels on the input plane, giving a 'perspective'-like visual impression. This can be quantitatively seen in the two equations above. When $g = h = 0$, the denominator is independent of u or v and thus we have spatial invariance. Homography preserves lines at all orientations. A very useful fact about Homography is that its inverse is also a Homography. These two properties play a very important role in the implementation of this transformation. A short but instructive and illustrated review of affine, projective and also bi-linear mappings is provided in Heckbert 1989[27].

6.4.2 Merging multiple warpings

In Section 6.4.1 [Warping basics], page 200, we saw how a basic warp/transformation can be represented with a matrix. To make more complex warpings (for example to define a translation, rotation and scale as one warp) the individual matrices have to be multiplied through matrix multiplication. However matrix multiplication is not commutative, so the order of the set of matrices you use for the multiplication is going to be very important.

The first warping should be placed as the left-most matrix. The second warping to the right of that and so on. The second transformation is going to occur on the warped coordinates of the first. As an example for merging a few transforms into one matrix, the multiplication below represents the rotation of an image about a point $[\,U \quad V\,]$ anticlockwise from the horizontal axis by an angle of θ. To do this, first we take the origin to $[\,U \quad V\,]$ through translation. Then we rotate the image, then we translate it back to where it was initially. These three operations can be merged in one operation by calculating the matrix multiplication below:

$$\begin{bmatrix} 1 & 0 & U \\ 0 & 1 & V \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -U \\ 0 & 1 & -V \\ 0 & 0 & 1 \end{bmatrix}$$

6.4.3 Resampling

A digital image is composed of discrete 'picture elements' or 'pixels'. When a real image is created from a camera or detector, each pixel's area is used to store the number of photo-electrons that were created when incident photons collided with that pixel's surface area. This process is called the 'sampling' of a continuous or analog data into digital data. When we change the pixel grid of an image or warp it as we defined in Section 6.4.1 [Warping basics], page 200, we have to 'guess' the flux value of each pixel on the new grid based on the old grid, or re-sample it. Because of the 'guessing', any form of warping on the data

[27] Paul S. Heckbert. 1989. *Fundamentals of Texture mapping and Image Warping*, Master's thesis at University of California, Berkeley. Note that since points are defined as row vectors there, the matrix is the transpose of the one discussed here.

is going to degrade the image and mix the original pixel values with each other. So if an analysis can be done on an unwarped data image, it is best to leave the image untouched and pursue the analysis. However as discussed in Section 6.4 [Warp], page 199, this is not possible most of the times, so we have to accept the problem and re-sample the image.

In most applications of image processing, it is sufficient to consider each pixel to be a point and not an area. This assumption can significantly speed up the processing of an image and also the simplicity of the code. It is a fine assumption when the signal to noise ratio of the objects are very large. The question will then be one of interpolation because you have multiple points distributed over the output image and you want to find the values at the pixel centers. To increase the accuracy, you might also sample more than one point from within a pixel giving you more points for a more accurate interpolation in the output grid.

However, interpolation has several problems. The first one is that it will depend on the type of function you want to assume for the interpolation. For example you can choose a bi-linear or bi-cubic (the 'bi's are for the 2 dimensional nature of the data) interpolation method. For the latter there are various ways to set the constants[28]. Such functional interpolation functions can fail seriously on the edges of an image. They will also need normalization so that the flux of the objects before and after the warpings are comparable. The most basic problem with such techniques is that they are based on a point while a detector pixel is an area. They add a level of subjectivity to the data (make more assumptions through the functions than the data can handle). For most applications this is fine, but in scientific applications where detection of the faintest possible galaxies or fainter parts of bright galaxies is our aim, we cannot afford this loss. Because of these reasons Warp will not use such interpolation techniques.

Warp will do interpolation based on "pixel mixing"[29] or "area resampling". This is also what the Hubble Space Telescope pipeline calls "Drizzling"[30]. This technique requires no functions, it is thus non-parametric. It is also the closest we can get (make least assumptions) to what actually happens on the detector pixels. The basic idea is that you reverse-transform each output pixel to find which pixels of the input image it covers and what fraction of the area of the input pixels are covered. To find the output pixel value, you simply sum the value of each input pixel weighted by the overlap fraction (between 0 to 1) of the output pixel and that input pixel. Through this process, pixels are treated as an area not as a point (which is how detectors create the image), also the brightness (see Section 8.1.3 [Flux Brightness and magnitude], page 290) of an object will be left completely unchanged.

If there are very high spatial-frequency signals in the image (for example fringes) which vary on a scale smaller than your output image pixel size, pixel mixing can cause ailiasing[31]. So if the input image has fringes, they have to be calculated and removed separately (which would naturally be done in any astronomical application). Because of the PSF no astronomical target has a sharp change in the signal so this issue is less important for astronomical applications, see Section 8.1.1.2 [Point spread function], page 285.

[28] see `http://entropymine.com/imageworsener/bicubic/` for a nice introduction.

[29] For a graphic demonstration see `http://entropymine.com/imageworsener/pixelmixing/`.

[30] `http://en.wikipedia.org/wiki/Drizzle_(image_processing)`

[31] `http://en.wikipedia.org/wiki/Aliasing`

6.4.4 Invoking Warp

Warp an input dataset into a new grid. Any homographic warp (for example scaling, rotation, translation, projection) is acceptable, see Section 6.4.1 [Warping basics], page 200, for the definitions. The general template for invoking Warp is:

```
$ astwarp [OPTIONS...] InputImage
```

One line examples:

```
## Rotate and then scale input image:
$ astwarp --rotate=37.92 --scale=0.8 image.fits

## Scale, then translate the input image:
$ astwarp --scale 8/3 --translate 2.1 image.fits

## Align raw image with celestial coordinates:
$ astwarp --align rawimage.fits --output=aligned.fits

## Directly input a custom warping matrix (using fraction):
$ astwarp --matrix=1/5,0,4/10,0,1/5,4/10,0,0,1 image.fits

## Directly input a custom warping matrix, with final numbers:
$ astwarp --matrix="0.7071,-0.7071,  0.7071,0.7071" image.fits
```

If any processing is to be done, Warp can accept one file as input. As in all Gnuastro programs, when an output is not explicitly set with the `--output` option, the output filename will be set automatically based on the operation, see Section 4.8 [Automatic output], page 124. For the full list of general options to all Gnuastro programs (including Warp), please see Section 4.1.2 [Common options], page 95.

To be the most accurate, the input image will be read as a 64-bit double precision floating point dataset and all internal processing is done in this format (including the raw output type). You can use the common `--type` option to write the output in any type you want, see Section 4.5 [Numeric data types], page 115.

Warps must be specified as command-line options, either as (possibly multiple) modular warpings (for example `--rotate`, or `--scale`), or directly as a single raw matrix (with `--matrix`). If specified together, the latter (direct matrix) will take precedence and all the modular warpings will be ignored. Any number of modular warpings can be specified on the command-line and configuration files. If more than one modular warping is given, all will be merged to create one warping matrix. As described in Section 6.4.2 [Merging multiple warpings], page 202, matrix multiplication is not commutative, so the order of specifying the modular warpings on the command-line, and/or configuration files makes a difference (see Section 4.2.2 [Configuration file precedence], page 107). The full list of modular warpings and the other options particular to Warp are described below.

The values to the warping options (modular warpings as well as `--matrix`), are a sequence of at least one number. Each number in this sequence is separated from the next by a comma (`,`). Each number can also be written as a single fraction (with a forward-slash `/` between the numerator and denominator). Space and Tab characters are permitted between any two numbers, just don't forget to quote the whole value. Otherwise, the value will not be fully passed onto the option. See the examples above as a demonstration.

Based on the FITS standard, integer values are assigned to the center of a pixel and the coordinate [1.0, 1.0] is the center of the first pixel (bottom left of the image when viewed in SAO ds9). So the coordinate center [0.0, 0.0] is half a pixel away (in each axis) from the bottom left vertex of the first pixel. The resampling that is done in Warp (see Section 6.4.3 [Resampling], page 202) is done on the coordinate axes and thus directly depends on the coordinate center. In some situations this if fine, for example when rotating/aligning a real image, all the edge pixels will be similarly affected. But in other situations (for example when scaling an over-sampled mock image to its intended resolution, this is not desired: you want the center of the coordinates to be on the corner of the pixel. In such cases, you can use the `--centeroncorner` option which will shift the center by 0.5 before the main warp, then shift it back by -0.5 after the main warp, see below.

`-a`
`--align` Align the image and celestial (WCS) axes given in the input. After it, the vertical image direction (when viewed in SAO ds9) corresponds to the declination and the horizontal axis is the inverse of the Right Ascension (RA). The inverse of the RA is chosen so the image can correspond to what you would actually see on the sky and is common in most survey images.

Align is internally treated just like a rotation (`--rotation`), but uses the input image's WCS to find the rotation angle. Thus, if you have rotated the image before calling `--align`, you might get unexpected results (because the rotation is defined on the original WCS).

`-r FLT`
`--rotate=FLT`
Rotate the input image by the given angle in degrees: θ in Section 6.4.1 [Warping basics], page 200. Note that commonly, the WCS structure of the image is set such that the RA is the inverse of the image horizontal axis which increases towards the right in the FITS standard and as viewed by SAO ds9. So the default center for rotation is on the right of the image. If you want to rotate about other points, you have to translate the warping center first (with `--translate`) then apply your rotation and then return the center back to the original position (with another call to `--translate`, see Section 6.4.2 [Merging multiple warpings], page 202.

`-s FLT[,FLT]`
`--scale=FLT[,FLT]`
Scale the input image by the given factor(s): M and N in Section 6.4.1 [Warping basics], page 200. If only one value is given, then both image axes will be scaled with the given value. When two values are given (separated by a comma), the first will be used to scale the first axis and the second will be used for the second axis. If you only need to scale one axis, use `1` for the axis you don't need to scale. The value(s) can also be written (on the command-line or in configuration files) as a fraction.

`-f FLT[,FLT]`
`--flip=FLT[,FLT]`
Flip the input image around the given axis(s). If only one value is given, then both image axes are flipped. When two values are given (separated by a

comma), you can choose which axis to flip over. `--flip` only takes values `0` (for no flip), or `1` (for a flip). Hence, if you want to flip by the second axis only, use `--flip=0,1`.

`-e FLT[,FLT]`
`--shear=FLT[,FLT]`

Shear the input image by the given value(s): A and B in Section 6.4.1 [Warping basics], page 200. If only one value is given, then both image axes will be sheared with the given value. When two values are given (separated by a comma), the first will be used to shear the first axis and the second will be used for the second axis. If you only need to shear along one axis, use `0` for the axis that must be untouched. The value(s) can also be written (on the command-line or in configuration files) as a fraction.

`-t FLT[,FLT]`
`--translate=FLT[,FLT]`

Translate (move the center of coordinates) the input image by the given value(s): c and f in Section 6.4.1 [Warping basics], page 200. If only one value is given, then both image axes will be translated by the given value. When two values are given (separated by a comma), the first will be used to translate the first axis and the second will be used for the second axis. If you only need to translate along one axis, use `0` for the axis that must be untouched. The value(s) can also be written (on the command-line or in configuration files) as a fraction.

`-p FLT[,FLT]`
`--project=FLT[,FLT]`

Apply a projection to the input image by the given values(s): g and h in Section 6.4.1 [Warping basics], page 200. If only one value is given, then projection will apply to both axes with the given value. When two values are given (separated by a comma), the first will be used to project the first axis and the second will be used for the second axis. If you only need to project along one axis, use `0` for the axis that must be untouched. The value(s) can also be written (on the command-line or in configuration files) as a fraction.

`-m STR`
`--matrix=STR`

The warp/transformation matrix. All the elements in this matrix must be separated by comas(`,`) characters and as described above, you can also use fractions (a forward-slash between two numbers). The transformation matrix can be either a 2 by 2 (4 numbers), or a 3 by 3 (9 numbers) array. In the former case (if a 2 by 2 matrix is given), then it is put into a 3 by 3 matrix (see Section 6.4.1 [Warping basics], page 200).

The determinant of the matrix has to be non-zero and it must not contain any non-number values (for example infinities or NaNs). The elements of the matrix have to be written row by row. So for the general Homography matrix of Section 6.4.1 [Warping basics], page 200, it should be called with `--matrix=a,b,c,d,e,f,g,h,1`.

The raw matrix takes precedence over all the modular warping options listed above, so if it is called with any number of modular warps, the latter are ignored.

`-c`
`--centeroncorer`
: Put the center of coordinates on the corner of the first (bottom-left when viewed in SAO ds9) pixel. This option is applied after the final warping matrix has been finalized: either through modular warpings or the raw matrix. See the explanation above for coordinates in the FITS standard to better understand this option and when it should be used.

`--hstartwcs=INT`
: Specify the first header keyword number (line) that should be used to read the WCS information, see the full explanation in Section 6.1.4 [Invoking Crop], page 155.

`--hendwcs=INT`
: Specify the last header keyword number (line) that should be used to read the WCS information, see the full explanation in Section 6.1.4 [Invoking Crop], page 155.

`-k`
`--keepwcs`
: Do not correct the WCS information of the input image and save it untouched to the output image. By default the WCS (World Coordinate System) information of the input image is going to be corrected in the output image so the objects in the image are at the same WCS coordinates. But in some cases it might be useful to keep it unchanged (for example to correct alignments).

`-C FLT`
`--coveredfrac=FLT`
: Depending on the warp, the output pixels that cover pixels on the edge of the input image, or blank pixels in the input image, are not going to be fully covered by input data. With this option, you can specify the acceptable covered fraction of such pixels (any value between 0 and 1). If you only want output pixels that are fully covered by the input image area (and are not blank), then you can set `--coveredfrac=1`. Alternatively, a value of `0` will keep output pixels that are even infinitesimally covered by the input(so the sum of the pixels in the input and output images will be the same).

7 Data analysis

Astronomical datasets (images or tables) contain very valuable information, the tools in this section can help in analyzing, extracting, and quantifying that information. For example getting general or specific statistics of the dataset (with Section 7.1 [Statistics], page 208), detecting signal within a noisy dataset (with Section 7.2 [NoiseChisel], page 225), or creating a catalog from an input dataset (with Section 7.4 [MakeCatalog], page 255).

7.1 Statistics

The distribution of values in a dataset can provide valuable information about it. For example, in an image, if it is a positively skewed distribution, we can see that there is significant data in the image. If the distribution is roughly symmetric, we can tell that there is no significant data in the image. In a table, when we need to select a sample of objects, it is important to first get a general view of the whole sample.

On the other hand, you might need to know certain statistical parameters of the dataset. For example, if we have run a detection algorithm on an image, and we want to see how accurate it was, one method is to calculate the average of the undetected pixels and see how reasonable it is (if detection is done correctly, the average of undetected pixels should be approximately equal to the background value, see Section 7.1.3 [Sky value], page 211). In a table, you might have calculated the magnitudes of a certain class of objects and want to get some general characteristics of the distribution immediately on the command-line (very fast!), to possibly change some parameters. The Statistics program is designed for such situations.

7.1.1 Histogram and Cumulative Frequency Plot

Histograms and the cumulative frequency plots are both used to visually study the distribution of a dataset. A histogram shows the number of data points which lie within pre-defined intervals (bins). So on the horizontal axis we have the bin centers and on the vertical, the number of points that are in that bin. You can use it to get a general view of the distribution: which values have been repeated the most? how close/far are the most significant bins? Are there more values in the larger part of the range of the dataset, or in the lower part? Similarly, many very important properties about the dataset can be deduced from a visual inspection of the histogram. In the Statistics program, the histogram can be either output to a table to plot with your favorite plotting program[1], or it can be shown with ASCII characters on the command-line, which is very crude, but good enough for a fast and on-the-go analysis, see the example in Section 7.1.4 [Invoking Statistics], page 215.

The width of the bins is only necessary parameter for a histogram. In the limiting case that the bin-widths tend to zero (while assuming the number of points in the dataset tend to infinity), then the histogram will tend to the probability density function (`https://en.wikipedia.org/wiki/Probability_density_function`) of the distribution. When the absolute number of points in each bin is not relevant to the study (only the shape of the histogram is important), you

[1] We recommend PGFPlots (`http://pgfplots.sourceforge.net/`) which generates your plots directly within TEX (the same tool that generates your document).

can *normalize* a histogram so like the probability density function, the sum of all its bins will be one.

In the cumulative frequency plot of a distribution, the horizontal axis is the sorted data values and the y axis is the index of each data in the sorted distribution. Unlike a histogram, a cumulative frequency plot does not involve intervals or bins. This makes it less prone to any sort of bias or error that a given bin-width would have on the analysis. When a larger number of the data points have roughly the same value, then the cumulative frequency plot will become steep in that vicinity. This occurs because on the horizontal axis, there is little change while on the vertical axis, the indexes constantly increase. Normalizing a cumulative frequency plot means to divide each index (y axis) by the total number of data points (or the last value).

Unlike the histogram which has a limited number of bins, ideally the cumulative frequency plot should have one point for every data element. Even in small datasets (for example a 200×200 image) this will result in an unreasonably large number of points to plot (40000)! As a result, for practical reasons, it is common to only store its value on a certain number of points (intervals) in the input range rather than the whole dataset, so you should determine the number of bins you want when asking for a cumulative frequency plot. In Gnuastro (and thus the Statistics program), the number reported for each bin is the total number of data points until the larger interval value for that bin. You can see an example histogram and cumulative frequency plot of a single dataset under the `--asciihist` and `--asciicfp` options of Section 7.1.4 [Invoking Statistics], page 215.

So as a summary, both the histogram and cumulative frequency plot in Statistics will work with bins. Within each bin/interval, the lower value is considered to be within then bin (it is inclusive), but its larger value is not (it is exclusive). Formally, an interval/bin between a and b is represented by [a, b). When the over-all range of the dataset is specified (with the `--greaterequal`, `--lessthan`, or `--qrange` options), the acceptable values of the dataset are also defined with a similar inclusive-exclusive manner. But when the range is determined from the actual dataset (none of these options is called), the last element in the dataset is included in the last bin's count.

7.1.2 Sigma clipping

Let's assume that you have pure noise (centered on zero) with a clear Gaussian distribution (`https://en.wikipedia.org/wiki/Normal_distribution`), or see Section 8.2.1.1 [Photon counting noise], page 302. Now let's assume you add very bright objects (signal) on the image which have a very sharp boundary. By a sharp boundary, we mean that there is a clear cutoff (from the noise) at the pixels the objects finish. In other words, at their boundaries, the objects do not fade away into the noise. In such a case, when you plot the histogram (see Section 7.1.1 [Histogram and Cumulative Frequency Plot], page 208) of the distribution, the pixels relating to those objects will be clearly separate from pixels that belong to parts of the image that did not have any signal (were just noise). In the cumulative frequency plot, after a steady rise (due to the noise), you would observe a long flat region were for a certain range of data (horizontal axis), there is no increase in the index (vertical axis).

Outliers like the example above can significantly bias the measurement of noise statistics. σ-clipping is defined as a way to avoid the effect of such outliers. In astronomical applications, cosmic rays (when they collide at a near normal incidence angle) are a very

good example of such outliers. The tracks they leave behind in the image are perfectly immune to the blurring caused by the atmosphere and the aperture. They are also very energetic and so their borders are usually clearly separated from the surrounding noise. So σ-clipping is very useful in removing their effect on the data. See Figure 15 in Akhlaghi and Ichikawa, 2015 (`https://arxiv.org/abs/1505.01664`).

σ-clipping is defined as the very simple iteration below. In each iteration, the range of input data might decrease and so when the outliers have the conditions above, the outliers will be removed through this iteration. The exit criteria will be discussed below.

1. Calculate the standard deviation (σ) and median (m) of a distribution.
2. Remove all points that are smaller or larger than $m \pm \alpha\sigma$.
3. Go back to step 1, unless the selected exit criteria is reached.

The reason the median is used as a reference and not the mean is that the mean is too significantly affected by the presence of outliers, while the median is less affected, see Section 7.1.3.3 [Quantifying signal in a tile], page 213. As you can tell from this algorithm, besides the condition above (that the signal have clear high signal to noise boundaries) σ-clipping is only useful when the signal does not cover more than half of the full data set. If they do, then the median will lie over the outliers and σ-clipping might remove the pixels with no signal.

There are commonly two exit criteria to stop the σ-clipping iteration:

- When a certain number of iterations has taken place (second value to the `--sigclip` option is larger than 1).
- When the new measured standard deviation is within a certain tolerance level of the old one (second value to the `--sigclip` option is less than 1). The tolerance level is defined by:

$$\frac{\sigma_{old} - \sigma_{new}}{\sigma_{new}}$$

  The standard deviation is used because it is heavily influenced by the presence of outliers. Therefore the fact that it stops changing between two iterations is a sign that we have successfully removed outliers. Note that in each clipping, the dispersion in the distribution is either less or equal. So $\sigma_{old} \geq \sigma_{new}$.

When working on astronomical images, objects like galaxies and stars are blurred by the atmosphere and the telescope aperture, therefore their signal sinks into the noise very gradually. Galaxies in particular do not appear to have a clear high signal to noise cutoff at all. Therefore σ-clipping will not be useful in removing their effect on the data.

To gauge if σ-clipping will be useful for your dataset, look at the histogram (see Section 7.1.1 [Histogram and Cumulative Frequency Plot], page 208). The ASCII histogram that is printed on the command-line with `--asciihist` is good enough in most cases.

7.1.3 Sky value

One of the most important aspects of a dataset is its reference value: the value of the dataset where there is no signal. Without knowing, and thus removing the effect of, this value it is impossible to compare the derived results of many high-level analyses over the dataset with other datasets (in the attempt to associate our results with the "real" world).

In astronomy, this reference value is known as the "Sky" value: the value that noise fluctuates around: where there is no signal from detectable objects or artifacts (for example galaxies, stars, planets or comets, star spikes or internal optical ghost). Depending on the dataset, the Sky value maybe a fixed value over the whole dataset, or it may vary based on location. For an example of the latter case, see Figure 11 in Akhlaghi and Ichikawa (2015) (`https://arxiv.org/abs/1505.01664`).

Because of the significance of the Sky value in astronomical data analysis, we have devoted this subsection to it for a thorough review. We start with a thorough discussion on its definition (Section 7.1.3.1 [Sky value definition], page 211). In the astronomical literature, researchers use a variety of methods to estimate the Sky value, so in Section 7.1.3.2 [Sky value misconceptions], page 212) we review those and discuss their biases. From the definition of the Sky value, the most accurate way to estimate the Sky value is to run a detection algorithm (for example Section 7.2 [NoiseChisel], page 225) over the dataset and use the undetected pixels. However, there is also a more crude method that maybe useful when good direct detection is not initially possible (for example due to too many cosmic rays in a shallow image). A more crude (but simpler method) that is usable in such situations is discussed in Section 7.1.3.3 [Quantifying signal in a tile], page 213.

7.1.3.1 Sky value definition

This analysis is taken from Akhlaghi and Ichikawa (2015) (`https://arxiv.org/abs/1505.01664`). Let's assume that all instrument defects – bias, dark and flat – have been corrected and the brightness (see Section 8.1.3 [Flux Brightness and magnitude], page 290) of a detected object, O, is desired. The sources of flux on pixel[2] i of the image can be written as follows:

- Contribution from the target object (O_i).
- Contribution from other detected objects (D_i).
- Undetected objects or the fainter undetected regions of bright objects (U_i).
- A cosmic ray (C_i).
- The background flux, which is defined to be the count if none of the others exists on that pixel (B_i).

The total flux in this pixel (T_i) can thus be written as:

$$T_i = B_i + D_i + U_i + C_i + O_i.$$

By definition, D_i is detected and it can be assumed that it is correctly estimated (deblended) and subtracted, we can thus set $D_i = 0$. There are also methods to detect and remove

[2] For this analysis the dimension of the data (image) is irrelevant. So if the data is an image (2D) with width of w pixels, then a pixel located on column x and row y (where all counting starts from zero and (0, 0) is located on the bottom left corner of the image), would have an index: $i = x + y \times w$.

cosmic rays, for example the method described in van Dokkum (2001)[3], or by comparing multiple exposures. This allows us to set $C_i = 0$. Note that in practice, D_i and U_i are correlated, because they both directly depend on the detection algorithm and its input parameters. Also note that no detection or cosmic ray removal algorithm is perfect. With these limitations in mind, the observed Sky value for this pixel (S_i) can be defined as

$$S_i \equiv B_i + U_i.$$

Therefore, as the detection process (algorithm and input parameters) becomes more accurate, or $U_i \to 0$, the Sky value will tend to the background value or $S_i \to B_i$. Hence, we see that while B_i is an inherent property of the data (pixel in an image), S_i depends on the detection process. Over a group of pixels, for example in an image or part of an image, this equation translates to the average of undetected pixels (Sky= $\sum S_i$). With this definition of Sky, the object flux in the data can be calculated, per pixel, with

$$T_i = S_i + O_i \quad \to \quad O_i = T_i - S_i.$$

In the fainter outskirts of an object, a very small fraction of the photo-electrons in a pixel actually belongs to objects, the rest is caused by random factors (noise), see Figure 1b in Akhlaghi and Ichikawa (2015) (`https://arxiv.org/abs/1505.01664`). Therefore even a small over estimation of the Sky value will result in the loss of a very large portion of most galaxies. Besides the lost area/brightness, this will also cause an over-estimation of the Sky value and thus even more under-estimation of the object's brightness. It is thus very important to detect the diffuse flux of a target, even if they are not your primary target.

In summary, the more accurately the Sky is measured, the more accurately the brightness (sum of pixel values) of the target object can be measured (photometry). Any under/over-estimation in the Sky will directly translate to an over/under-estimation of the measured object's brightness.

The **Sky value** is only correctly found when all the detected objects (D_i and C_i) have been removed from the data.

7.1.3.2 Sky value misconceptions

As defined in Section 7.1.3 [Sky value], page 211, the sky value is only accurately defined when the detection algorithm is not significantly reliant on the sky value. In particular its detection threshold. However, most signal-based detection tools[4] use the sky value as a reference to define the detection threshold. These older techniques therefore had to rely on approximations based on other assumptions about the data. A review of those other

[3] van Dokkum, P. G. (2001). Publications of the Astronomical Society of the Pacific. 113, 1420.

[4] According to Akhlaghi and Ichikawa (2015), signal-based detection is a detection process that relies heavily on assumptions about the to-be-detected objects. This method was the most heavily used technique prior to the introduction of NoiseChisel in that paper.

techniques can be seen in Appendix A of Akhlaghi and Ichikawa (2015) (`https://arxiv.org/abs/1505.01664`).

These methods were extensively used in astronomical data analysis for several decades, therefore they have given rise to a lot of misconceptions, ambiguities and disagreements about the sky value and how to measure it. As a summary, the major methods used until now were an approximation of the mode of the image pixel distribution and σ-clipping.

- To find the mode of a distribution those methods would either have to assume (or find) a certain probability density function (PDF) or use the histogram. But astronomical datasets can have any distribution, making it almost impossible to define a generic function. Also, histogram-based results are very inaccurate (there is a large dispersion) and it depends on the histogram bin-widths. Generally, the mode of a distribution also shifts as signal is added. Therefore, even if it is accurately measured, the mode is a biased measure for the Sky value.
- Another approach was to iteratively clip the brightest pixels in the image (which is known as σ-clipping). See Section 7.1.2 [Sigma clipping], page 209, for a complete explanation. σ-clipping is useful when there are clear outliers (an object with a sharp edge in an image for example). However, real astronomical objects have diffuse and faint wings that penetrate deeply into the noise, see Figure 1 in Akhlaghi and Ichikawa (2015) (`https://arxiv.org/abs/1505.01664`).

As discussed in Section 7.1.3 [Sky value], page 211, the sky value can only be correctly defined as the average of undetected pixels. Therefore all such approaches that try to approximate the sky value prior to detection are ultimately poor approximations.

7.1.3.3 Quantifying signal in a tile

Put simply, noise can be characterized with a certain spread about the measured value. In the Gaussian distribution (most commonly used to model noise) the spread is defined by the standard deviation about the characteristic mean.

Let's start by clarifying some definitions first: *Data* is defined as the combination of signal and noise (so a noisy image is one *data*set). *Signal* is defined as the mean of the noise on each element. We'll also assume that the *background* (see Section 7.1.3.1 [Sky value definition], page 211) is subtracted and is zero.

When a data set doesn't have any signal (only noise), the mean, median and mode of the distribution are equal within statistical errors and approximately equal to the background value. Signal always has a positive value and will never become negative, see Figure 1 in Akhlaghi and Ichikawa (2015) (`https://arxiv.org/abs/1505.01664`). Therefore, as more signal is added, the mean, median and mode of the dataset shift to the positive. The mean's shift is the largest. The median shifts less, since it is defined based on an ordered distribution and so is not affected by a small number of outliers. The distribution's mode shifts the least to the positive.

Inverting the argument above gives us a robust method to quantify the significance of signal in a dataset. Namely, when the mean and median of a distribution are approximately equal, or the mean's quantile is around 0.5, we can argue that there is no significant signal.

To allow for gradients (which are commonly present in ground-based images), we can consider the image to be made of a grid of tiles (see Section 4.7 [Tessellation], page 123[5]).

[5] The options to customize the tessellation are discussed in Section 4.1.2.2 [Processing options], page 98.

Hence, from the difference of the mean and median on each tile, we can estimate the significance of signal in it. The median of a distribution is defined to be the value of the distribution's middle point after sorting (or 0.5 quantile). Thus, to estimate the presence of signal, we'll compare with the quantile of the mean with 0.5. If the absolute difference in a tile is larger than the value given to the `--meanmedqdiff` option, that tile will be ignored. You can read this option as "mean-median-quantile-difference".

The raw dataset's distribution is noisy, so using the argument above on the raw input will give a noisy result. To decrease the noise/error in estimating the mode, we will use convolution (see Section 6.3.1.1 [Convolution process], page 178). Convolution decreases the range of the dataset and enhances its skewness, See Section 3.1.1 and Figure 4 in Akhlaghi and Ichikawa (2015) (`https://arxiv.org/abs/1505.01664`). This enhanced skewness can be interpreted as an increase in the Signal to noise ratio of the objects buried in the noise. Therefore, to obtain an even better measure of the presence of signal in a tile, the mean and median discussed above are measured on the convolved image.

Through the difference of the mean and median we have actually 'detected' data in the distribution. However this "detection" was only based on the total distribution of the data in each tile (a much lower resolution). This is the main limitation of this technique. The best approach is thus to do detection over the dataset, mask all the detected pixels and use the undetected regions to estimate the sky and its standard deviation (possibly over a tessellation). This is how NoiseChisel works: it uses the argument above to find tiles that are used to find its thresholds. Several higher-level steps are done on the thresholded pixels to define the higher-level detections (see Section 7.2 [NoiseChisel], page 225).

There is one final hurdle: raw astronomical datasets are commonly peppered with Cosmic rays. Images of Cosmic rays aren't smoothed by the atmosphere or telescope aperture, so they have sharp boundaries. Also, since they don't occupy too many pixels, they don't affect the mode and median calculation. But their very high values can greatly bias the calculation of the mean (recall how the mean shifts the fastest in the presence of outliers), for example see Figure 15 in Akhlaghi and Ichikawa (2015) (`https://arxiv.org/abs/1505.01664`).

The effect of outliers like cosmic rays on the mean and standard deviation can be removed through σ-clipping, see Section 7.1.2 [Sigma clipping], page 209, for a complete explanation. Therefore, after asserting that the mode and median are approximately equal in a tile (see Section 4.7 [Tessellation], page 123), the final Sky value and its standard deviation are determined after σ-clipping with the `--sigmaclip` option.

In the end, some of the tiles will pass the mean and median quantile difference test. However, prior to interpolating over the failed tiles, another point should be considered: large and extended galaxies, or bright stars, have wings which sink into the noise very gradually. In some cases, the gradient over these wings can be on scales that is larger than the tiles. The mean-median distance test will pass on such tiles and will cause a strong peak in the interpolated tile grid, see Section 2.3 [Detecting large extended targets], page 47.

The tiles that exist over the wings of large galaxies or bright stars are outliers in the distribution of tiles that passed the mean-median quantile distance test. Therefore, the final step of "quantifying signal in a tile" is to look at this distribution and remove the outliers. σ-clipping is a good solution for removing a few outliers, but the problem with outliers of this kind is that there may be many such tiles (depending on the large/bright stars/galaxies in the image). Therefore a novel outlier rejection algorithm will be used.

To identify the first outlier, we'll use the distribution of distances between sorted elements. If there are N successful tiles, for every tile, the distance between the adjacent $N/2$ previous elements is found, giving a distribution containing $N/2-1$ points. The σ-clipped median and standard deviation of this distribution is then found (σ-clipping is configured with `--outliersclip`). Finally, if the distance between the element and its previous element is more than `--outliersigma` multiples of the σ-clipped standard deviation added with the σ-clipped median, that element is considered an outlier and all tiles larger than that value are ignored.

Formally, if we assume there are N elements. They are first sorted. Searching for the outlier starts on element $N/2$ (integer division). Let's take v_i to be the i-th element of the sorted input (with no blank values) and m and σ as the σ-clipped median and standard deviation from the distances of the previous $N/2-1$ elements (not including v_i). If the value given to `--outliersigma` is displayed with s, the i-th element is considered as an outlier when the condition below is true.

$$\frac{(v_i - v_{i-1}) - m}{\sigma} > s$$

Since i begins from the median, the outlier has to be larger than the median. You can use the check images (for example `--checksky` in the Statistics program or `--checkqthresh`, `--checkdetsky` and `--checksky` options in NoiseChisel for any of its steps that uses this outlier rejection) to inspect the steps and see which tiles have been discarded as outliers prior to interpolation.

7.1.4 Invoking Statistics

Statistics will print statistical measures of an input dataset (table column or image). The executable name is `aststatistics` with the following general template

```
$ aststatistics [OPTION ...] InputImage.fits
```

One line examples:

```
## Print some general statistics of input image:
$ aststatistics image.fits

## Print some general statistics of column named MAG_F160W:
$ aststatistics catalog.fits -h1 --column=MAG_F160W

## Make the histogram of the column named MAG_F160W:
$ aststatistics table.fits -cMAG_F160W --histogram

## Find the Sky value on image with a given kernel:
$ aststatistics image.fits --sky --kernel=kernel.fits

## Print Sigma-clipped results of records with a MAG_F160W
## column value between 26 and 27:
$ aststatistics cat.fits -cMAG_F160W -g26 -l27 --sigmaclip=3,0.2

## Print the median value of all records in column MAG_F160W that
```

```
## have a value larger than 3 in column PHOTO_Z:
$ aststatistics tab.txt -rPHOTO_Z -g3 -cMAG_F160W --median

## Calculate the median of the third column in the input table, but only
## for rows where the mean of the first and second columns is >5.
$ awk '($1+$2)/2 > 5 {print $3}' table.txt | aststatistics --median
```

Statistics can take its input dataset either from a file (image or table) or the Standard input (see Section 4.1.3 [Standard input], page 104). If any output file is to be created, the value to the `--output` option, is used as the base name for the generated files. Without `--output`, the input name will be used to generate an output name, see Section 4.8 [Automatic output], page 124. The options described below are particular to Statistics, but for general operations, it shares a large collection of options with the other Gnuastro programs, see Section 4.1.2 [Common options], page 95, for the full list. For more on reading from standard input, please see the description of `--stdintimeout` option in Section 4.1.2.1 [Input/Output options], page 95. Options can also be given in configuration files, for more, please see Section 4.2 [Configuration files], page 106.

The input dataset may have blank values (see Section 6.1.3 [Blank pixels], page 154), in this case, all blank pixels are ignored during the calculation. Initially, the full dataset will be read, but it is possible to select a specific range of data elements to use in the analysis of each run. You can either directly specify a minimum and maximum value for the range of data elements to use (with `--greaterequal` or `--lessthan`), or specify the range using quantiles (with `--qrange`). If a range is specified, all pixels outside of it are ignored before any processing.

The following set of options are for specifying the input/outputs of Statistics. There are many other input/output options that are common to all Gnuastro programs including Statistics, see Section 4.1.2.1 [Input/Output options], page 95, for those.

`-c STR/INT`
`--column=STR/INT`
: The column to use when the input file is a table with more than one column. See Section 4.6.3 [Selecting table columns], page 121, for a full description of how to use this option. For more on how tables are read in Gnuastro, please see Section 4.6 [Tables], page 117.

`-r STR/INT`
`--refcol=STR/INT`
: The reference column selector when the input file is a table. When a reference column is given, the range options below will be applied to this column and only elements in the input column that have a reference value in the correct range will be used. In practice this option allows you to select a subset of the input column based on values in another (the reference) column. All the statistical calculations will be done on the selected input column, not the reference column.

`-g FLT`
`--greaterequal=FLT`
: Limit the range of inputs into those with values greater and equal to what is given to this option. None of the values below this value will be used in any of the processing steps below.

`-l FLT`
`--lessthan=FLT`
: Limit the range of inputs into those with values less-than what is given to this option. None of the values greater or equal to this value will be used in any of the processing steps below.

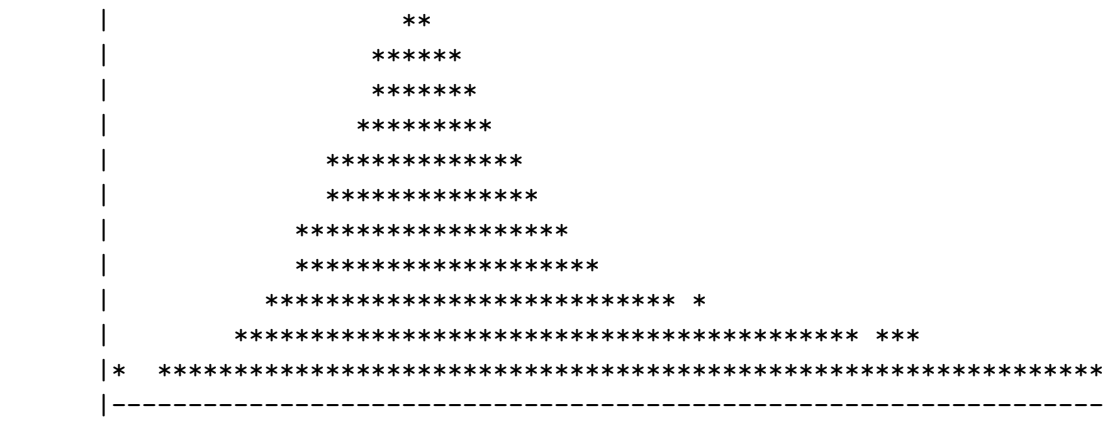
`-Q FLT[,FLT]`
`--qrange=FLT[,FLT]`
: Specify the range of usable inputs using the quantile. This option can take one or two quantiles to specify the range. When only one number is input (let's call it Q), the range will be those values in the quantile range Q to $1-Q$. So when only one value is given, it must be less than 0.5. When two values are given, the first is used as the lower quantile range and the second is used as the larger quantile range.

  The quantile of a given element in a dataset is defined by the fraction of its index to the total number of values in the sorted input array. So the smallest and largest values in the dataset have a quantile of 0.0 and 1.0. The quantile is a very useful non-parametric (making no assumptions about the input) relative measure to specify a range. It can best be understood in terms of the cumulative frequency plot, see Section 7.1.1 [Histogram and Cumulative Frequency Plot], page 208. The quantile of each horizontal axis value in the cumulative frequency plot is the vertical axis value associate with it.

When no operation is requested, Statistics will print some general basic properties of the input dataset on the command-line like the example below (ran on one of the output images of `make check`[6]). This default behavior is designed to help give you a general feeling of how the data are distributed and help in narrowing down your analysis.

```
$ aststatistics convolve_spatial_scaled_noised.fits     \
                --greaterequal=9500 --lessthan=11000
Statistics (GNU Astronomy Utilities) X.X
-------
Input: convolve_spatial_scaled_noised.fits (hdu: 0)
Range: from (inclusive) 9500, upto (exclusive) 11000.
Unit: Brightness
-------
  Number of elements:                   9074
  Minimum:                              9622.35
  Maximum:                              10999.7
  Mode:                                 10055.45996
  Mode quantile:                        0.4001983908
  Median:                               10093.7
  Mean:                                 10143.98257
  Standard deviation:                   221.80834
-------
Histogram:
```

[6] You can try it by running the command in the `tests` directory, open the image with a FITS viewer and have a look at it to get a sense of how these statistics relate to the input image/dataset.



Gnuastro's Statistics is a very general purpose program, so to be able to easily understand this diversity in its operations (and how to possibly run them together), we'll divided the operations into two types: those that don't respect the position of the elements and those that do (by tessellating the input on a tile grid, see Section 4.7 [Tessellation], page 123). The former treat the whole dataset as one and can re-arrange all the elements (for example sort them), but the former do their processing on each tile independently. First, we'll review the operations that work on the whole dataset.

The group of options below can be used to get single value measurement(s) of the whole dataset. They will print only the requested value as one field in a line/row, like the `--mean`, `--median` options. These options can be called any number of times and in any order. The outputs of all such options will be printed on one line following each other (with a space character between them). This feature makes these options very useful in scripts, or to redirect into programs like GNU AWK for higher-level processing. These are some of the most basic measures, Gnuastro is still under heavy development and this list will grow. If you want another statistical parameter, please contact us and we will do out best to add it to this list, see Section 1.8 [Suggest new feature], page 12.

`-n`
`--number` Print the number of all used (non-blank and in range) elements.

`--minimum`
Print the minimum value of all used elements.

`--maximum`
Print the maximum value of all used elements.

`--sum` Print the sum of all used elements.

`-m`
`--mean` Print the mean (average) of all used elements.

`-t`
`--std` Print the standard deviation of all used elements.

`-E`
`--median` Print the median of all used elements.

`-u FLT[,FLT[,...]]`
`--quantile=FLT[,FLT[,...]]`
Print the values at the given quantiles of the input dataset. Any number of quantiles may be given and one number will be printed for each. Values can either be written as a single number or as fractions, but must be between zero and one (inclusive). Hence, in effect `--quantile=0.25` `--quantile=0.75` is equivalent to `--quantile=0.25,3/4`, or `-u1/4,3/4`.

The returned value is one of the elements from the dataset. Taking q to be your desired quantile, and N to be the total number of used (non-blank and within the given range) elements, the returned value is at the following position in the sorted array: $round(q \times N)$.

`--quantfunc=FLT[,FLT[,...]]`
Print the quantiles of the given values in the dataset. This option is the inverse of the `--quantile` and operates similarly except that the acceptable values are within the range of the dataset, not between 0 and 1. Formally it is known as the "Quantile function".

Since the dataset is not continuous this function will find the nearest element of the dataset and use its position to estimate the quantile function.

`-O`
`--mode` Print the mode of all used elements. The mode is found through the mirror distribution which is fully described in Appendix C of Akhlaghi and Ichikawa 2015 (`https://arxiv.org/abs/1505.01664`). See that section for a full description.

This mode calculation algorithm is non-parametric, so when the dataset is not large enough (larger than about 1000 elements usually), or doesn't have a clear mode it can fail. In such cases, this option will return a value of `nan` (for the floating point NaN value).

As described in that paper, the easiest way to assess the quality of this mode calculation method is to use it's symmetricity (see `--modesym` below). A better way would be to use the `--mirror` option to generate the histogram and cumulative frequency tables for any given mirror value (the mode in this case) as a table. If you generate plots like those shown in Figure 21 of that paper, then your mode is accurate.

`--modequant`
Print the quantile of the mode. You can get the actual mode value from the `--mode` described above. In many cases, the absolute value of the mode is irrelevant, but its position within the distribution is important. In such cases, this option will become handy.

`--modesym`
Print the symmetricity of the calculated mode. See the description of `--mode` for more. This mode algorithm finds the mode based on how symmetric it is, so if the symmetricity returned by this option is too low, the mode is not too accurate. See Appendix C of Akhlaghi and Ichikawa 2015 (`https://arxiv.org/abs/1505.01664`) for a full description. In practice, symmetricity values larger than 0.2 are mostly good.

`--modesymvalue`
: Print the value in the distribution where the mirror and input distributions are no longer symmetric, see `--mode` and Appendix C of Akhlaghi and Ichikawa 2015 (`https://arxiv.org/abs/1505.01664`) for more.

The list of options below are for those statistical operations that output more than one value. So while they can be called together in one run, their outputs will be distinct (each one's output will usually be printed in more than one line).

`-A`
`--asciihist`
: Print an ASCII histogram of the usable values within the input dataset along with some basic information like the example below (from the UVUDF catalog[7]). The width and height of the histogram (in units of character widths and heights on your command-line terminal) can be set with the `--numasciibins` (for the width) and `--asciiheight` options.

  For a full description of the histogram, please see Section 7.1.1 [Histogram and Cumulative Frequency Plot], page 208. An ASCII plot is certainly very crude and cannot be used in any publication, but it is very useful for getting a general feeling of the input dataset very fast and easily on the command-line without having to take your hands off the keyboard (which is a major distraction!). If you want to try it out, you can write it all in one line and ignore the `\` and extra spaces.

```
$ aststatistics uvudf_rafelski_2015.fits.gz --hdu=1         \
                --column=MAG_F160W --lessthan=40            \
                --asciihist --numasciibins=55
ASCII Histogram:
Number: 8593
Y: (linear: 0 to 660)
X: (linear: 17.7735 -- 31.4679, in 55 bins)
 |                                          ****
 |                                         *****
 |                                        ******
 |                                       ********
 |                                       *********
 |                                     ***********
 |                                   ***************
 |                                 ******************
 |                             ************************
 |                      ********************************
 |*** ***************************************************
 |-------------------------------------------------------
```

`--asciicfp`
: Print the cumulative frequency plot of the usable elements in the input dataset. Please see descriptions under `--asciihist` for more, the example below is from

[7] `https://asd.gsfc.nasa.gov/UVUDF/uvudf_rafelski_2015.fits.gz`

the same input table as that example. To better understand the cumulative frequency plot, please see Section 7.1.1 [Histogram and Cumulative Frequency Plot], page 208.

```
$ aststatistics uvudf_rafelski_2015.fits.gz --hdu=1         \
                --column=MAG_F160W --lessthan=40            \
                --asciicfp --numasciibins=55
ASCII Cumulative frequency plot:
Y: (linear: 0 to 8593)
X: (linear: 17.7735 -- 31.4679, in 55 bins)
 |                                                    *******
 |                                                  **********
 |                                                 ***********
 |                                               *************
 |                                              ***************
 |                                             ****************
 |                                           ******************
 |                                         **********************
 |                                     ************************
 |                               ******************************
 |************************************************************
 |------------------------------------------------------------
```

`-H`
`--histogram`
Save the histogram of the usable values in the input dataset into a table. The first column is the value at the center of the bin and the second is the number of points in that bin. If the `--cumulative` option is also called with this option in a run, then the table will have three columns (the third is the cumulative frequency plot). Through the `--numbins`, `--onebinstart`, or `--manualbinrange`, you can modify the first column values and with `--normalize` and `--maxbinone` you can modify the second columns. See below for the description of each.

By default (when no `--output` is specified) a plain text table will be created, see Section 4.6.2 [Gnuastro text table format], page 119. If a FITS name is specified, you can use the common option `--tableformat` to have it as a FITS ASCII or FITS binary format, see Section 4.1.2 [Common options], page 95. This table can then be fed into your favorite plotting tool and get a much more clean and nice histogram than what the raw command-line can offer you (with the `--asciihist` option).

`-C`
`--cumulative`
Save the cumulative frequency plot of the usable values in the input dataset into a table, similar to `--histogram`.

`-s`
`--sigmaclip`
Do σ-clipping on the usable pixels of the input dataset. See Section 7.1.2 [Sigma clipping], page 209, for a full description on σ-clipping and also to

better understand this option. The σ-clipping parameters can be set through the `--sclipparams` option (see below).

`--mirror=FLT`
Make a histogram and cumulative frequency plot of the mirror distribution for the given dataset when the mirror is located at the value to this option. The mirror distribution is fully described in Appendix C of Akhlaghi and Ichikawa 2015 (`https://arxiv.org/abs/1505.01664`) and currently it is only used to calculate the mode (see `--mode`).

Just note that the mirror distribution is a discrete distribution like the input, so while you may give any number as the value to this option, the actual mirror value is the closest number in the input dataset to this value. If the two numbers are different, Statistics will warn you of the actual mirror value used.

This option will make a table as output. Depending on your selected name for the output, it will be either a FITS table or a plain text table (which is the default). It contains three columns: the first is the center of the bins, the second is the histogram (with the largest value set to 1) and the third is the normalized cumulative frequency plot of the mirror distribution. The bins will be positioned such that the mode is on the starting interval of one of the bins to make it symmetric around the mirror. With this output file and the input histogram (that you can generate in another run of Statistics, using the `--onebinvalue`), it is possible to make plots like Figure 21 of Akhlaghi and Ichikawa 2015 (`https://arxiv.org/abs/1505.01664`).

The list of options below allow customization of the histogram and cumulative frequency plots (for the `--histogram`, `--cumulative`, `--asciihist`, and `--asciicfp` options).

`--numbins`
The number of bins (rows) to use in the histogram and the cumulative frequency plot tables (outputs of `--histogram` and `--cumulative`).

`--numasciibins`
The number of bins (characters) to use in the ASCII plots when printing the histogram and the cumulative frequency plot (outputs of `--asciihist` and `--asciicfp`).

`--asciiheight`
The number of lines to use when printing the ASCII histogram and cumulative frequency plot on the command-line (outputs of `--asciihist` and `--asciicfp`).

`-n`
`--normalize`
Normalize the histogram or cumulative frequency plot tables (outputs of `--histogram` and `--cumulative`). For a histogram, the sum of all bins will become one and for a cumulative frequency plot the last bin value will be one.

`--maxbinone`
Divide all the histogram values by the maximum bin value so it becomes one and the rest are similarly scaled. In some situations (for example if you want to plot the histogram and cumulative frequency plot in one plot) this can be very useful.

`--onebinstart=FLT`

Make sure that one bin starts with the value to this option. In practice, this will shift the bins used to find the histogram and cumulative frequency plot such that one bin's lower interval becomes this value.

For example when a histogram range includes negative and positive values and zero has a special significance in your analysis, then zero might fall somewhere in one bin. As a result that bin will have counts of positive and negative. By setting `--onebinstart=0`, you can make sure that one bin will only count negative values in the vicinity of zero and the next bin will only count positive ones in that vicinity.

Note that by default, the first row of the histogram and cumulative frequency plot show the central values of each bin. So in the example above you will not see the 0.000 in the first column, you will see two symmetric values.

If the value is not within the usable input range, this option will be ignored. When it is, this option is the last operation before the bins are finalized, therefore it has a higher priority than options like `--manualbinrange`.

`--manualbinrange`

Use the values given to the `--greaterequal` and `--lessthan` to define the range of all bin-based calculations like the histogram. This option itself doesn't take any value, but just tells the program to use the values of those two options instead of the minimum and maximum values of a plot. If any of the two options are not given, then the minimum or maximum will be used respectively. Therefore, if none of them are called calling this option is redundant.

The `--onebinstart` option has a higher priority than this option. In other words, `--onebinstart` takes effect after the range has been finalized and the initial bins have been defined, therefore it has the power to (possibly) shift the bins. If you want to manually set the range of the bins *and* have one bin on a special value, it is thus better to avoid `--onebinstart`.

All the options described until now were from the first class of operations discussed above: those that treat the whole dataset as one. However. It often happens that the relative position of the dataset elements over the dataset is significant. For example you don't want one median value for the whole input image, you want to know how the median changes over the image. For such operations, the input has to be tessellated (see Section 4.7 [Tessellation], page 123). Thus this class of options can't currently be called along with the options above in one run of Statistics.

`-t`
`--ontile` Do the respective single-valued calculation over one tile of the input dataset, not the whole dataset. This option must be called with at least one of the single valued options discussed above (for example `--mean` or `--quantile`). The output will be a file in the same format as the input. If the `--oneelempertile` option is called, then one element/pixel will be used for each tile (see Section 4.1.2.2 [Processing options], page 98). Otherwise, the output will have the same size as the input, but each element will have the value corresponding to that tile's value. If multiple single valued operations are called, then for each operation there will be one extension in the output FITS file.

`-y`
`--sky` Estimate the Sky value on each tile as fully described in Section 7.1.3.3 [Quantifying signal in a tile], page 213. As described in that section, several options are necessary to configure the Sky estimation which are listed below. The output file will have two extensions: the first is the Sky value and the second is the Sky standard deviation on each tile. Similar to `--ontile`, if the `--oneelempertile` option is called, then one element/pixel will be used for each tile (see Section 4.1.2.2 [Processing options], page 98).

The parameters for estimating the sky value can be set with the following options, except for the `--sclipparams` option (which is also used by the `--sigmaclip`), the rest are only used for the Sky value estimation.

`-k=STR`
`--kernel=STR`
File name of kernel to help in estimating the significance of signal in a tile, see Section 7.1.3.3 [Quantifying signal in a tile], page 213.

`--khdu=STR`
Kernel HDU to help in estimating the significance of signal in a tile, see Section 7.1.3.3 [Quantifying signal in a tile], page 213.

`--meanmedqdiff=FLT`
The maximum acceptable distance between the quantiles of the mean and median, see Section 7.1.3.3 [Quantifying signal in a tile], page 213. The initial Sky and its standard deviation estimates are measured on tiles where the quantiles of their mean and median are less distant than the value given to this option. For example `--meanmedqdiff=0.01` means that only tiles where the mean's quantile is between 0.49 and 0.51 (recall that the median's quantile is 0.5) will be used.

`--sclipparams=FLT,FLT`
The σ-clipping parameters, see Section 7.1.2 [Sigma clipping], page 209. This option takes two values which are separated by a comma (`,`). Each value can either be written as a single number or as a fraction of two numbers (for example `3,1/10`). The first value to this option is the multiple of σ that will be clipped (α in that section). The second value is the exit criteria. If it is less than 1, then it is interpreted as tolerance and if it is larger than one it is a specific number. Hence, in the latter case the value must be an integer.

`--outliersclip=FLT,FLT`
Sigma-clipping parameters for the outlier rejection of the Sky value (similar to `--sclipparams`).

Outlier rejection is useful when the dataset contains a large and diffuse (almost flat within each tile) signal. The flatness of the profile will cause it to successfully pass the mean-median quantile difference test, so we'll need to use the distribution of successful tiles for removing these false positive. For more, see the latter half of Section 7.1.3.3 [Quantifying signal in a tile], page 213.

`--outliersigma=FLT`
: Multiple of sigma to define an outlier in the Sky value estimation. If this option is given a value of zero, no outlier rejection will take place. For more see `--outliersclip` and the latter half of Section 7.1.3.3 [Quantifying signal in a tile], page 213.

`--smoothwidth=INT`
: Width of a flat kernel to convolve the interpolated tile values. Tile interpolation is done using the median of the `--interpnumngb` neighbors of each tile (see Section 4.1.2.2 [Processing options], page 98). If this option is given a value of zero or one, no smoothing will be done. Without smoothing, strong boundaries will probably be created between the values estimated for each tile. It is thus good to smooth the interpolated image so strong discontinuities do not show up in the final Sky values. The smoothing is done through convolution (see Section 6.3.1.1 [Convolution process], page 178) with a flat kernel, so the value to this option must be an odd number.

`--ignoreblankintiles`
: Don't set the input's blank pixels to blank in the tiled outputs (for example Sky and Sky standard deviation extensions of the output). This is only applicable when the tiled output has the same size as the input, in other words, when `--oneelempertile` isn't called.

  By default, blank values in the input (commonly on the edges which are outside the survey/field area) will be set to blank in the tiled outputs also. But in other scenarios this default behavior is not desired: for example if you have masked something in the input, but want the tiled output under that also.

`--checksky`
: Create a multi-extension FITS file showing the steps that were used to estimate the Sky value over the input, see Section 7.1.3.3 [Quantifying signal in a tile], page 213. The file will have two extensions for each step (one for the Sky and one for the Sky standard deviation).

7.2 NoiseChisel

Once instrumental signatures are removed from the raw data (image) in the initial reduction process (see Chapter 6 [Data manipulation], page 151). You are naturally eager to start answering the scientific questions that motivated the data collection in the first place. However, the raw dataset/image is just an array of values/pixels, that is all! These raw values cannot directly be used to answer your scientific questions: for example "how many galaxies are there in the image?".

The first high-level step in your analysis of your targets will thus be to classify, or label, the dataset elements (pixels) into two classes: 1) noise, where random effects are the major contributor to the value, and 2) signal, where non-random factors (for example light from a distant galaxy) are present. This classification of the elements in a dataset is formally known as *detection*.

In an observational/experimental dataset, signal is always buried in noise: only mock/simulated datasets are free of noise. Therefore detection, or the process of separating

signal from noise, determines the number of objects you study and the accuracy of any higher-level measurement you do on them. Detection is thus the most important step of any analysis and is not trivial. In particular, the most scientifically interesting astronomical targets are faint, can have a large variety of morphologies, along with a large distribution in brightness and size. Therefore when noise is significant, proper detection of your targets is a uniquely decisive step in your final scientific analysis/result.

NoiseChisel is Gnuastro's program for detection of targets that don't have a sharp border (almost all astronomical objects). When the targets have a sharp edges/border (for example cells in biological imaging), a simple threshold is enough to separate them from noise and each other (if they are not touching). To detect such sharp-edged targets, you can use Gnuastro's Arithmetic program in a command like below (assuming the threshold is `100`, see Section 6.2 [Arithmetic], page 161):

```
$ astarithmetic in.fits 100 gt 2 connected-components
```

Since almost no astronomical target has such sharp edges, we need a more advanced detection methodology. NoiseChisel uses a new noise-based paradigm for detection of very extended and diffuse targets that are drowned deeply in the ocean of noise. It was initially introduced in Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`). The name of NoiseChisel is derived from the first thing it does after thresholding the dataset: to erode it. In mathematical morphology, erosion on pixels can be pictured as carving-off boundary pixels. Hence, what NoiseChisel does is similar to what a wood chisel or stone chisel do. It is just not a hardware, but a software. In fact, looking at it as a chisel and your dataset as a solid cube of rock will greatly help in effectively understanding and optimally using it: with NoiseChisel you literally carve your targets out of the noise. Try running it with the `--checkdetection` option to see each step of the carving process on your input dataset.

NoiseChisel's primary output is a binary detection map with the same size as the input but only with two values: 0 and 1. Pixels that don't harbor any detected signal (noise) are given a label (or value) of zero and those with a value of 1 have been identified as hosting signal.

Segmentation is the process of classifying the signal into higher-level constructs. For example if you have two separate galaxies in one image, by default NoiseChisel will give a value of 1 to the pixels of both, but after segmentation, the pixels in each will get separate labels. NoiseChisel is only focused on detection (separating signal from noise), to *segment* the signal (into separate galaxies for example), Gnuastro has a separate specialized program Section 7.3 [Segment], page 243. NoiseChisel's output can be directly/readily fed into Segment.

For more on NoiseChisel's output format and its benefits (especially in conjunction with Section 7.3 [Segment], page 243, and later Section 7.4 [MakeCatalog], page 255), please see Akhlaghi [2016] (`https://arxiv.org/abs/1611.06387`). Just note that when that paper was published, Segment was not yet spun-off into a separate program, and NoiseChisel done both detection and segmentation.

NoiseChisel's output is designed to be generic enough to be easily used in any higher-level analysis. If your targets are not touching after running NoiseChisel and you aren't interested in their sub-structure, you don't need the Segment program at all. You can ask NoiseChisel to find the connected pixels in the output with the `--label` option. In this

case, the output won't be a binary image any more, the signal will have counters/labels starting from 1 for each connected group of pixels. You can then directly feed NoiseChisel's output into MakeCatalog for measurements over the detections and the production of a catalog (see Section 7.4 [MakeCatalog], page 255).

Thanks to the published papers mentioned above, there is no need to provide a more complete introduction to NoiseChisel in this book. However, published papers cannot be updated any more, but the software has evolved/changed. The changes since publication are documented in Section 7.2.1 [NoiseChisel changes after publication], page 227. Afterwards, in Section 7.2.2 [Invoking NoiseChisel], page 230, the details of running NoiseChisel and its options are discussed.

As discussed above, detection is one of the most important steps for your scientific result. It is therefore very important to obtain a good understanding of NoiseChisel (and afterwards Section 7.3 [Segment], page 243, and Section 7.4 [MakeCatalog], page 255). We thus strongly recommend that after reading the papers above and the respective sections of Gnuastro's book, you play a little with the settings (in the order presented in the paper and Section 7.2.2 [Invoking NoiseChisel], page 230) on a dataset you are familiar with and inspect all the check images (options starting with `--check`) to see the effect of each parameter.

We strongly recommend going over the two tutorials of Section 2.2 [General program usage tutorial], page 24, and Section 2.3 [Detecting large extended targets], page 47. They are designed to show how to most effectively use NoiseChisel for the detection of small faint objects and large extended objects. In the meantime, they will show you the modular principle behind Gnuastro's programs and how they are built to complement, and build upon, each other. Section 2.2 [General program usage tutorial], page 24, culminates in using NoiseChisel to detect galaxies and use its outputs to find the galaxy colors. Defining colors is a very common process in most science-cases. Therefore it is also recommended to (patiently) complete that tutorial for optimal usage of NoiseChisel in conjunction with all the other Gnuastro programs. Section 2.3 [Detecting large extended targets], page 47, shows you can optimize NoiseChisel's settings for very extended objects to successfully carve out to signal-to-noise ratio levels of below 1/10.

In Section 7.2.1 [NoiseChisel changes after publication], page 227, we'll review the changes in NoiseChisel since the publication of Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`). We will then review NoiseChisel's input, detection, and output options in Section 7.2.2.1 [NoiseChisel input], page 231, Section 7.2.2.2 [Detection options], page 234, and Section 7.2.2.3 [NoiseChisel output], page 241.

7.2.1 NoiseChisel changes after publication

NoiseChisel was initially introduced in Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`). It is thus strongly recommended to read this paper for a good understanding of what it does and how each parameter influences the output. To help in understanding how it works, that paper has a large number of figures showing every step on multiple mock and real examples.

However, the paper cannot be updated anymore, but NoiseChisel has evolved (and will continue to do so): better algorithms or steps have been found, thus options will be added

or removed. This book is thus the final and definitive guide to NoiseChisel. The aim of this section is to make the transition from the paper to the installed version on your system, as smooth as possible with the list below. For a more detailed list of changes in previous Gnuastro releases/versions, please see the `NEWS` file[8].

The most important change since the publication of that paper is that from Gnuastro 0.6, NoiseChisel is only in charge on detection. Segmentation of the detected signal was spun-off into a separate program: Section 7.3 [Segment], page 243. This spin-off allows much greater creativity and is in the spirit of Gnuastro's modular design (see Section 12.2 [Program design philosophy], page 447). Below you can see the major changes since that paper was published. First, the removed options/features are discussed, then we review the new features that have been added.

Removed features/options:

- `--skysubtracted`: This option was used to account for the extra noise that is added if the Sky value has already been subtracted. However, NoiseChisel estimates the Sky standard deviation based on the input data, not exposure maps or other theoretical considerations. Therefore the standard deviation of the undetected pixels also contains the errors due to any previous sky subtraction. This option is therefore no longer present in NoiseChisel.

  `--dilate`: In the paper, true detections were dilated for a final dig into the noise. However, the simple 8-connected dilation produced boxy results which were not realistic and could miss diffuse flux. The final dig into the noise is now done by "grow"ing the true detections, similar to how true clumps were grown, see the description of `--detgrowquant` below and in Section 7.2.2.2 [Detection options], page 234, for more on the new alternative.
- Segmentation has been completely moved to a new program: Section 7.3 [Segment], page 243.

Added features/options:

- `--widekernel`: NoiseChisel uses the difference between the mode and median to identify if a tile should be used for estimating the quantile thresholds (see Section 7.1.3.3 [Quantifying signal in a tile], page 213). Until now, NoiseChisel would convolve an image once and estimate the proper tiles for quantile estimations on the convolved image. The same convolved image would later be used for quantile estimation. A larger kernel does increase the skewness (and thus difference between the mode and median, therefore helps in detecting the presence signal), however, it disfigures the shapes/morphology of the objects.

  This new `--widekernel` option (and a corresponding `--wkhdu` option to specify its HDU) option are added to solve such cases. When its given, the input will be convolved with both the sharp (given through the `--kernel` option) and wide kernels. The mode and median are calculated on the dataset that is convolved with the wider kernel, then the quantiles are estimated on the image convolved with the sharper kernel.
- The quantile difference to identify tiles with no significant signal is measured between the *mean* and median. In the published paper, it was between the *mode* and median. The quantile of the mean is more sensitive to skewness (the presence of signal), so it is

[8] The `NEWS` file is present in the released Gnuastro tarball, see Section 3.2.1 [Release tarball], page 71.

preferable to the quantile of the mode. For more see Section 7.1.3.3 [Quantifying signal in a tile], page 213.

- Outlier rejection in quantile thresholds: When there are large galaxies or bright stars in the image, their gradient may be on a smaller scale than the selected tile size. In such cases, those tiles will be identified as tiles with no signal and thus preserved. An outlier identification algorithm has been added to NoiseChisel and can be configured with the following options: `--outliersigma` and `--outliersclip`. For a more complete description, see the latter half of Section 7.1.3.3 [Quantifying signal in a tile], page 213.
- `--blankasforeground`: allows blank pixels to be treated as foreground in NoiseChisel's binary operations: the initial erosion (`--erode`) and opening (`--open`) as well as the filling holes and opening step for defining pseudo-detections (`--dthresh`). In the published paper, blank pixels were treated as foreground by default. To avoid too many false positive near blank/masked regions, blank pixels are now considered to be in the background. This option will create the old behavior.
- `--skyfracnoblank`: To reduce the bias caused by undetected wings of galaxies and stars in the Sky measurements, NoiseChisel only uses tiles that have a sufficiently large fraction of undetected pixels. Until now the reference for this fraction was the whole tile size. With this option, it is now possible to ask for ignoring blank pixels when calculating the fraction. This is useful when blank/masked pixels are distributed across the image. For more, see the description of this option in Section 7.2.2.2 [Detection options], page 234.
- `dopening`: Number of openings after applying `--dthresh`. For more, see the description of this option in Section 7.2.2.2 [Detection options], page 234.
- `dopeningngb`: Number of openings after applying `--dthresh`. For more, see the description of this option in Section 7.2.2.2 [Detection options], page 234.
- `--holengb`: The connectivity (defined by the number of neighbors) to fill holes after applying `--dthresh` (above) to find pseudo-detections. For more, see the description of this option in Section 7.2.2.2 [Detection options], page 234.
- `--pseudoconcomp`: The connectivity (defined by the number of neighbors) to find individual pseudo-detections. For more, see the description of this option in Section 7.2.2.2 [Detection options], page 234.
- `--snthresh`: Manually set the S/N of true pseudo-detections and thus avoid the need to manually identify this value. For more, see the description of this option in Section 7.2.2.2 [Detection options], page 234.
- `--detgrowquant`: is used to grow the final true detections until a given quantile in the same way that clumps are grown during segmentation (compare columns 2 and 3 in Figure 10 of the paper). It replaces the old `--dilate` option in the paper and older versions of Gnuastro. Dilation is a blind growth method which causes objects to be boxy or diamond shaped when too many layers are added. However, with the growth method that is defined now, we can follow the signal into the noise with any shape. The appropriate quantile depends on your dataset's correlated noise properties and how cleanly it was Sky subtracted. The new `--detgrowmaxholesize` can also be used to specify the maximum hole size to fill as part of this growth, see the description in Section 7.2.2.2 [Detection options], page 234, for more details.

This new growth process can be much more successful in detecting diffuse flux around true detections compared to dilation and give more realistic results, but it can also increase the NoiseChisel run time (depending on the given value and input size).

- `--cleangrowndet`: A process to further clean/remove the possibility of false detections, see the descriptions under this option in Section 7.2.2.2 [Detection options], page 234.

7.2.2 Invoking NoiseChisel

NoiseChisel will detect signal in noise producing a multi-extension dataset containing a binary detection map which is the same size as the input. Its output can be readily used for input into Section 7.3 [Segment], page 243, for higher-level segmentation, or Section 7.4 [MakeCatalog], page 255, to do measurements and generate a catalog. The executable name is `astnoisechisel` with the following general template

```
$ astnoisechisel [OPTION ...] InputImage.fits
```

One line examples:

```
## Detect signal in input.fits.
$ astnoisechisel input.fits

## Inspect all the detection steps after changing a parameter.
$ astnoisechisel input.fits --qthresh=0.4 --checkdetection

## Detect signal assuming input has 4 amplifier channels along first
## dimension and 1 along the second. Also set the regular tile size
## to 100 along both dimensions:
$ astnoisechisel --numchannels=4,1 --tilesize=100,100 input.fits
```

If NoiseChisel is to do processing (for example you don't want to get help, or see the values to each input parameter), an input image should be provided with the recognized extensions (see Section 4.1.1.1 [Arguments], page 93). NoiseChisel shares a large set of common operations with other Gnuastro programs, mainly regarding input/output, general processing steps, and general operating modes. To help in a unified experience between all of Gnuastro's programs, these operations have the same command-line options, see Section 4.1.2 [Common options], page 95, for a full list/description (they are not repeated here).

As in all Gnuastro programs, options can also be given to NoiseChisel in configuration files. For a thorough description on Gnuastro's configuration file parsing, please see Section 4.2 [Configuration files], page 106. All of NoiseChisel's options with a short description are also always available on the command-line with the `--help` option, see Section 4.3 [Getting help], page 109. To inspect the option values without actually running NoiseChisel, append your command with `--printparams` (or `-P`).

NoiseChisel's input image may contain blank elements (see Section 6.1.3 [Blank pixels], page 154). Blank elements will be ignored in all steps of NoiseChisel. Hence if your dataset has bad pixels which should be masked with a mask image, please use Gnuastro's Section 6.2 [Arithmetic], page 161, program (in particular its `where` operator) to convert those pixels to blank pixels before running NoiseChisel. Gnuastro's Arithmetic program has bitwise operators helping you select specific kinds of bad-pixels when necessary.

A convolution kernel can also be optionally given. If a value (file name) is given to `--kernel` on the command-line or in a configuration file (see Section 4.2 [Configuration files], page 106), then that file will be used to convolve the image prior to thresholding. Otherwise a default kernel will be used. The default kernel is a 2D Gaussian with a FWHM of 2 pixels truncated at 5 times the FWHM. This choice of the default kernel is discussed in Section 3.1.1 of Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`). See Section 6.3.4 [Convolution kernel], page 195, for kernel related options. Passing `none` to `--kernel` will disable convolution. On the other hand, through the `--convolved` option, you may provide an already convolved image, see descriptions below for more.

NoiseChisel defines two tessellations over the input (see Section 4.7 [Tessellation], page 123). This enables it to deal with possible gradients in the input dataset and also significantly improve speed by processing each tile on different threads simultaneously. Tessellation related options are discussed in Section 4.1.2.2 [Processing options], page 98. In particular, NoiseChisel uses two tessellations (with everything between them identical except the tile sizes): a fine-grained one with smaller tiles (used in thresholding and Sky value estimations) and another with larger tiles which is used for pseudo-detections over non-detected regions of the image. The common Tessellation options described in Section 4.1.2.2 [Processing options], page 98, define all parameters of both tessellations. The large tile size for the latter tessellation is set through the `--largetilesize` option. To inspect the tessellations on your input dataset, run NoiseChisel with `--checktiles`.

Usage TIP: Frequently use the options starting with `--check`. Since the noise properties differ between different datasets, you can often play with the parameters/options for a better result than the default parameters. You can start with `--checkdetection` for the main steps. For the full list of NoiseChisel's checking options please run:

```
$ astnoisechisel --help | grep check
```

Below, we'll discuss NoiseChisel's options, classified into two general classes, to help in easy navigation. Section 7.2.2.1 [NoiseChisel input], page 231, mainly discusses the basic options relating to inputs and prior to the detection process detection. Afterwards, Section 7.2.2.2 [Detection options], page 234, fully describes every configuration parameter (option) related to detection and how they affect the final result. The order of options in this section follow the logical order within NoiseChisel. On first reading (while you are still new to NoiseChisel), it is therefore strongly recommended to read the options in the given order below. The output of `--printparams` (or `-P`) also has this order. However, the output of `--help` is sorted alphabetically. Finally, in Section 7.2.2.3 [NoiseChisel output], page 241, the format of NoiseChisel's output is discussed.

7.2.2.1 NoiseChisel input

The options here can be used to configure the inputs and output of NoiseChisel, along with some general processing options. Recall that you can always see the full list of Gnuastro's options with the `--help` (see Section 4.3 [Getting help], page 109), or `--printparams` (or `-P`) to see their values (see Section 4.1.2.3 [Operating mode options], page 100).

`-k STR`
`--kernel=STR`
File name of kernel to smooth the image before applying the threshold, see Section 6.3.4 [Convolution kernel], page 195. If no convolution is needed, give this option a value of `none`.

The first step of NoiseChisel is to convolve/smooth the image and use the convolved image in multiple steps including the finding and applying of the quantile threshold (see `--qthresh`).

The `--kernel` option is not mandatory. If not called, a 2D Gaussian profile with a FWHM of 2 pixels truncated at 5 times the FWHM is used. This choice of the default kernel is discussed in Section 3.1.1 of Akhlaghi and Ichikawa [2015]. You can use MakeProfiles to build a kernel with any of its recognized profile types and parameters. For more details, please see Section 8.1.5.3 [MakeProfiles output dataset], page 297. For example, the command below will make a Moffat kernel (with $\beta = 2.8$) with FWHM of 2 pixels truncated at 10 times the FWHM.

```
$ astmkprof --oversample=1 --kernel=moffat,2,2.8,10
```

Since convolution can be the slowest step of NoiseChisel, for large datasets, you can convolve the image once with Gnuastro's Convolve (see Section 6.3 [Convolve], page 177), and use the `--convolved` option to feed it directly to NoiseChisel. This can help getting faster results when you are playing/testing the higher-level options.

`--khdu=STR`
HDU containing the kernel in the file given to the `--kernel` option.

`--convolved=STR`
Use this file as the convolved image and don't do convolution (ignore `--kernel`). NoiseChisel will just check the size of the given dataset is the same as the input's size. If a wrong image (with the same size) is given to this option, the results (errors, bugs, and etc) are unpredictable. So please use this option with care and in a highly controlled environment, for example in the scenario discussed below.

In almost all situations, as the input gets larger, the single most CPU (and time) consuming step in NoiseChisel (and other programs that need a convolved image) is convolution. Therefore minimizing the number of convolutions can save a significant amount of time in some scenarios. One such scenario is when you want to segment NoiseChisel's detections using the same kernel (with Section 7.3 [Segment], page 243, which also supports this `--convolved` option). This scenario would require two convolutions of the same dataset: once by NoiseChisel and once by Segment. Using this option in both programs, only one convolution (prior to running NoiseChisel) is enough.

Another common scenario where this option can be convenient is when you are testing NoiseChisel (or Segment) for the best parameters. You have to run NoiseChisel multiple times and see the effect of each change. However, once you are happy with the kernel, re-convolving the input on every change of higher-level parameters will greatly hinder, or discourage, further testing. With this option, you can convolve the input image with your chosen kernel once before

running NoiseChisel, then feed it to NoiseChisel on each test run and thus save valuable time for better/more tests.

To build your desired convolution kernel, you can use Section 8.1 [MakeProfiles], page 284. To convolve the image with a given kernel you can use Section 6.3 [Convolve], page 177. Spatial domain convolution is mandatory: in the frequency domain, blank pixels (if present) will cover the whole image and gradients will appear on the edges, see Section 6.3.3 [Spatial vs. Frequency domain], page 195.

Below you can see an example of the second scenario: you want to see how variation of the growth level (through the `--detgrowquant` option) will affect the final result. Recall that you can ignore all the extra spaces, new lines, and backslash's ('\') if you are typing in the terminal. In a shell script, remove the `$` signs at the start of the lines.

```
## Make the kernel to convolve with.
$ astmkprof --oversample=1 --kernel=gaussian,2,5

## Convolve the input with the given kernel.
$ astconvolve input.fits --kernel=kernel.fits                \
              --domain=spatial --output=convolved.fits

## Run NoiseChisel with seven growth quantile values.
$ for g in 60 65 70 75 80 85 90; do                          \
    astnoisechisel input.fits --convolved=convolved.fits     \
                   --detgrowquant=0.$g --output=$g.fits;     \
  done
```

`--chdu=STR`
: The HDU/extension containing the convolved image in the file given to `--convolved`.

`-w STR`
`--widekernel=STR`
: File name of a wider kernel to use in estimating the difference of the mode and median in a tile (this difference is used to identify the significance of signal in that tile, see Section 7.1.3.3 [Quantifying signal in a tile], page 213). As displayed in Figure 4 of Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`), a wider kernel will help in identifying the skewness caused by data in noise. The image that is convolved with this kernel is *only* used for this purpose. Once the mode is found to be sufficiently close to the median, the quantile threshold is found on the image convolved with the sharper kernel (`--kernel`), see `--qthresh`).

  Since convolution will significantly slow down the processing, this feature is optional. When it isn't given, the image that is convolved with `--kernel` will be used to identify good tiles *and* apply the quantile threshold. This option is mainly useful in conditions were you have a very large, extended, diffuse signal that is still present in the usable tiles when using `--kernel`. See Section 2.3

[Detecting large extended targets], page 47, for a practical demonstration on how to inspect the tiles used in identifying the quantile threshold.

`--whdu=STR`
HDU containing the kernel file given to the `--widekernel` option.

`-L INT[,INT]`
`--largetilesize=INT[,INT]`
The size of each tile for the tessellation with the larger tile sizes. Except for the tile size, all the other parameters for this tessellation are taken from the common options described in Section 4.1.2.2 [Processing options], page 98. The format is identical to that of the `--tilesize` option that is discussed in that section.

7.2.2.2 Detection options

Detection is the process of separating the pixels in the image into two groups: 1) Signal, and 2) Noise. Through the parameters below, you can customize the detection process in NoiseChisel. Recall that you can always see the full list of NoiseChisel's options with the `--help` (see Section 4.3 [Getting help], page 109), or `--printparams` (or `-P`) to see their values (see Section 4.1.2.3 [Operating mode options], page 100).

`-Q FLT`
`--meanmedqdiff=FLT`
The maximum acceptable distance between the quantiles of the mean and median in each tile, see Section 7.1.3.3 [Quantifying signal in a tile], page 213. The quantile threshold estimates are measured on tiles where the quantiles of their mean and median are less distant than the value given to this option. For example `--meanmedqdiff=0.01` means that only tiles where the mean's quantile is between 0.49 and 0.51 (recall that the median's quantile is 0.5) will be used.

`--outliersclip=FLT,FLT`
Sigma-clipping parameters for the outlier rejection of the quantile threshold. The format of the given values is similar to `--sigmaclip` below. In NoiseChisel, outlier rejection on tiles is used when identifying the quantile thresholds (`--qthresh`, `--noerodequant`, and `detgrowquant`).

Outlier rejection is useful when the dataset contains a large and diffuse (almost flat within each tile) signal. The flatness of the profile will cause it to successfully pass the mean-median quantile difference test, so we'll need to use the distribution of successful tiles for removing these false positives. For more, see the latter half of Section 7.1.3.3 [Quantifying signal in a tile], page 213.

`--outliersigma=FLT`
Multiple of sigma to define an outlier. If this option is given a value of zero, no outlier rejection will take place. For more see `--outliersclip` and the latter half of Section 7.1.3.3 [Quantifying signal in a tile], page 213.

`-t FLT`
`--qthresh=FLT`
The quantile threshold to apply to the convolved image. The detection process begins with applying a quantile threshold to each of the tiles in the small tessellation. The quantile is only calculated for tiles that don't have any significant

signal within them, see Section 7.1.3.3 [Quantifying signal in a tile], page 213. Interpolation is then used to give a value to the unsuccessful tiles and it is finally smoothed.

The quantile value is a floating point value between 0 and 1. Assume that we have sorted the N data elements of a distribution (the pixels in each mesh on the convolved image). The quantile (q) of this distribution is the value of the element with an index of (the nearest integer to) $q \times N$ in the sorted data set. After thresholding is complete, we will have a binary (two valued) image. The pixels above the threshold are known as foreground pixels (have a value of 1) while those which lie below the threshold are known as background (have a value of 0).

`--smoothwidth=INT`
Width of flat kernel used to smooth the interpolated quantile thresholds, see `--qthresh` for more.

`--checkqthresh`
Check the quantile threshold values on the mesh grid. A file suffixed with `_qthresh.fits` will be created showing each step. With this option, NoiseChisel will abort as soon as quantile estimation has been completed, allowing you to inspect the steps leading to the final quantile threshold, this can be disabled with `--continueaftercheck`. By default the output will have the same pixel size as the input, but with the `--oneelempertile` option, only one pixel will be used for each tile (see Section 4.1.2.2 [Processing options], page 98).

`--blankasforeground`
In the erosion and opening steps below, treat blank elements as foreground (regions above the threshold). By default, blank elements in the dataset are considered to be background, so if a foreground pixel is touching it, it will be eroded. This option is irrelevant if the datasets contains no blank elements.

When there are many blank elements in the dataset, treating them as foreground will systematically erode their regions less, therefore systematically creating more false positives. So use this option (when blank values are present) with care.

`-e INT`
`--erode=INT`
The number of erosions to apply to the binary thresholded image. Erosion is simply the process of flipping (from 1 to 0) any of the foreground pixels that neighbor a background pixel. In a 2D image, there are two kinds of neighbors, 4-connected and 8-connected neighbors. You can specify which type of neighbors should be used for erosion with the `--erodengb` option, see below.

Erosion has the effect of shrinking the foreground pixels. To put it another way, it expands the holes. This is a founding principle in NoiseChisel: it exploits the fact that with very low thresholds, the holes in the very low surface brightness regions of an image will be smaller than regions that have no signal. Therefore by expanding those holes, we are able to separate the regions harboring signal.

`--erodengb=INT`
: The type of neighborhood (structuring element) used in erosion, see `--erode` for an explanation on erosion. Only two integer values are acceptable: 4 or 8. In 4-connectivity, the neighbors of a pixel are defined as the four pixels on the top, bottom, right and left of a pixel that share an edge with it. The 8-connected neighbors on the other hand include the 4-connected neighbors along with the other 4 pixels that share a corner with this pixel. See Figure 6 (a) and (b) in Akhlaghi and Ichikawa (2015) for a demonstration.

`--noerodequant`
: Pure erosion is going to carve off sharp and small objects completely out of the detected regions. This option can be used to avoid missing such sharp and small objects (which have significant pixels, but not over a large area). All pixels with a value larger than the significance level specified by this option will not be eroded during the erosion step above. However, they will undergo the erosion and dilation of the opening step below.

  Like the `--qthresh` option, the significance level is determined using the quantile (a value between 0 and 1). Just as a reminder, in the normal distribution, 1σ, 1.5σ, and 2σ are approximately on the 0.84, 0.93, and 0.98 quantiles.

`-p INT`
`--opening=INT`
: Depth of opening to be applied to the eroded binary image. Opening is a composite operation. When opening a binary image with a depth of n, n erosions (explained in `--erode`) are followed by n dilations. Simply put, dilation is the inverse of erosion. When dilating an image any background pixel is flipped (from 0 to 1) to become a foreground pixel. Dilation has the effect of fattening the foreground. Note that in NoiseChisel, the erosion which is part of opening is independent of the initial erosion that is done on the thresholded image (explained in `--erode`). The structuring element for the opening can be specified with the `--openingngb` option. Opening has the effect of removing the thin foreground connections (mostly noise) between separate foreground 'islands' (detections) thereby completely isolating them. Once opening is complete, we have *initial* detections.

`--openingngb=INT`
: The structuring element used for opening, see `--erodengb` for more information about a structuring element.

`--skyfracnoblank`
: Ignore blank pixels when estimating the fraction of undetected pixels for Sky estimation. NoiseChisel only measures the Sky over the tiles that have a sufficiently large fraction of undetected pixels (value given to `--minskyfrac`). By default this fraction is found by dividing number of undetected pixels in a tile by the tile's area. But this default behavior ignores the possibility of blank pixels. In situations that blank/masked pixels are scattered across the image and if they are large enough, all the tiles can fail the `--minskyfrac` test, thus not allowing NoiseChisel to proceed. With this option, such scenarios can be

fixed: the denominator of the fraction will be the number of non-blank elements in the tile, not the total tile area.

`-B FLT`
`--minskyfrac=FLT`
Minimum fraction (value between 0 and 1) of Sky (undetected) areas in a tile. Only tiles with a fraction of undetected pixels (Sky) larger than this value will be used to estimate the Sky value. NoiseChisel uses this option value twice to estimate the Sky value: after initial detections and in the end when false detections have been removed.

Because of the PSF and their intrinsic amorphous properties, astronomical objects (except cosmic rays) never have a clear cutoff and commonly sink into the noise very slowly. Even below the very low thresholds used by NoiseChisel. So when a large fraction of the area of one mesh is covered by detections, it is very plausible that their faint wings are present in the undetected regions (hence causing a bias in any measurement). To get an accurate measurement of the above parameters over the tessellation, tiles that harbor too many detected regions should be excluded. The used tiles are visible in the respective `--check` option of the given step.

`--checkdetsky`
Check the initial approximation of the sky value and its standard deviation in a FITS file ending with `_detsky.fits`. With this option, NoiseChisel will abort as soon as the sky value used for defining pseudo-detections is complete. This allows you to inspect the steps leading to the final quantile threshold, this behavior can be disabled with `--continueaftercheck`. By default the output will have the same pixel size as the input, but with the `--oneelempertile` option, only one pixel will be used for each tile (see Section 4.1.2.2 [Processing options], page 98).

`-s FLT,FLT`
`--sigmaclip=FLT,FLT`
The σ-clipping parameters for measuring the initial and final Sky values from the undetected pixels, see Section 7.1.2 [Sigma clipping], page 209.

This option takes two values which are separated by a comma (`,`). Each value can either be written as a single number or as a fraction of two numbers (for example `3,1/10`). The first value to this option is the multiple of σ that will be clipped (α in that section). The second value is the exit criteria. If it is less than 1, then it is interpreted as tolerance and if it is larger than one it is assumed to be the fixed number of iterations. Hence, in the latter case the value must be an integer.

`-R FLT`
`--dthresh=FLT`
The detection threshold: a multiple of the initial Sky standard deviation added with the initial Sky approximation (which you can inspect with `--checkdetsky`). This flux threshold is applied to the initially undetected regions on the unconvolved image. The background pixels that are completely engulfed in a 4-connected foreground region are converted to background

(holes are filled) and one opening (depth of 1) is applied over both the initially detected and undetected regions. The Signal to noise ratio of the resulting 'pseudo-detections' are used to identify true vs. false detections. See Section 3.1.5 and Figure 7 in Akhlaghi and Ichikawa (2015) for a very complete explanation.

`--dopening=INT`
: The number of openings to do after applying `--dthresh`.

`--dopeningngb=INT`
: The connectivity used in the opening of `--dopening`. In a 2D image this must be either 4 or 8. The stronger the connectivity, the more smaller regions will be discarded.

`--holengb=INT`
: The connectivity (defined by the number of neighbors) to fill holes after applying `--dthresh` (above) to find pseudo-detections. For example in a 2D image it must be 4 (the neighbors that are most strongly connected) or 8 (all neighbors). The stronger the connectivity, the stronger the hole will be enclosed. So setting a value of 8 in a 2D image means that the walls of the hole are 4-connected. If standard (near Sky level) values are given to `--dthresh`, setting `--holengb=4`, might fill the complete dataset and thus not create enough pseudo-detections.

`--pseudoconcomp=INT`
: The connectivity (defined by the number of neighbors) to find individual pseudo-detections. If it is a weaker connectivity (4 in a 2D image), then pseudo-detections that are connected on the corners will be treated as separate.

`-m INT`
`--snminarea=INT`
: The minimum area to calculate the Signal to noise ratio on the pseudo-detections of both the initially detected and undetected regions. When the area in a pseudo-detection is too small, the Signal to noise ratio measurements will not be accurate and their distribution will be heavily skewed to the positive. So it is best to ignore any pseudo-detection that is smaller than this area. Use `--detsnhistnbins` to check if this value is reasonable or not.

`--checksn`
: Save the S/N values of the pseudo-detections (and possibly grown detections if `--cleangrowndet` is called) into separate tables. If `--tableformat` is a FITS table, each table will be written into a separate extension of one file suffixed with `_detsn.fits`. If it is plain text, a separate file will be made for each table (ending in `_detsn_sky.txt`, `_detsn_det.txt` and `_detsn_grown.txt`). For more on `--tableformat` see Section 4.1.2.1 [Input/Output options], page 95.

  You can use these to inspect the S/N values and their distribution (in combination with the `--checkdetection` option to see where the pseudo-detections are). You can use Gnuastro's Section 7.1 [Statistics], page 208, to make a histogram of the distribution or any other analysis you would like for better understanding of the distribution (for example through a histogram).

`--minnumfalse=INT`
: The minimum number of 'pseudo-detections' over the undetected regions to identify a Signal-to-Noise ratio threshold. The Signal to noise ratio (S/N) of false pseudo-detections in each tile is found using the quantile of the S/N distribution of the pseudo-detections over the undetected pixels in each mesh. If the number of S/N measurements is not large enough, the quantile will not be accurate (can have large scatter). For example if you set `--snquant=0.99` (or the top 1 percent), then it is best to have at least 100 S/N measurements.

`-c FLT`
`--snquant=FLT`
: The quantile of the Signal to noise ratio distribution of the pseudo-detections in each mesh to use for filling the large mesh grid. Note that this is only calculated for the large mesh grids that satisfy the minimum fraction of undetected pixels (value of `--minbfrac`) and minimum number of pseudo-detections (value of `--minnumfalse`).

`--snthresh=FLT`
: Manually set the signal-to-noise ratio of true pseudo-detections. With this option, NoiseChisel will not attempt to find pseudo-detections over the noisy regions of the dataset, but will directly go onto applying the manually input value.

  This option is useful in crowded images where there is no blank sky to find the sky pseudo-detections. You can get this value on a similarly reduced dataset (from another region of the Sky with more undetected regions spaces).

`-d FLT`
`--detgrowquant=FLT`
: Quantile limit to "grow" the final detections. As discussed in the previous options, after applying the initial quantile threshold, layers of pixels are carved off the objects to identify true signal. With this step you can return those low surface brightness layers that were carved off back to the detections. To disable growth, set the value of this option to `1`.

  The process is as follows: after the true detections are found, all the non-detected pixels above this quantile will be put in a list and used to "grow" the true detections (seeds of the growth). Like all quantile thresholds, this threshold is defined and applied to the convolved dataset. Afterwards, the dataset is dilated once (with minimum connectivity) to connect very thin regions on the boundary: imagine building a dam at the point rivers spill into an open sea/ocean. Finally, all holes are filled. In the geography metaphor, holes can be seen as the closed (by the dams) rivers and lakes, so this process is like turning the water in all such rivers and lakes into soil. See `--detgrowmaxholesize` for configuring the hole filling.

`--detgrowmaxholesize=INT`
: The maximum hole size to fill during the final expansion of the true detections as described in `--detgrowquant`. This is necessary when the input contains many smaller objects and can be used to avoid marking blank sky regions as detections.

For example multiple galaxies can be positioned such that they surround an empty region of sky. If all the holes are filled, the Sky region in between them will be taken as a detection which is not desired. To avoid such cases, the integer given to this option must be smaller than the hole between such objects. However, we should caution that unless the "hole" is very large, the combined faint wings of the galaxies might actually be present in between them, so be very careful in not filling such holes.

On the other hand, if you have a very large (and extended) galaxy, the diffuse wings of the galaxy may create very large holes over the detections. In such cases, a large enough value to this option will cause all such holes to be detected as part of the large galaxy and thus help in detecting it to extremely low surface brightness limits. Therefore, especially when large and extended objects are present in the image, it is recommended to give this option (very) large values. For one real-world example, see Section 2.3 [Detecting large extended targets], page 47.

`--cleangrowndet`

After dilation, if the signal-to-noise ratio of a detection is less than the derived pseudo-detection S/N limit, that detection will be discarded. In an ideal/clean noise, a true detection's S/N should be larger than its constituent pseudo-detections because its area is larger and it also covers more signal. However, on a false detections (especially at lower `--snquant` values), the increase in size can cause a decrease in S/N below that threshold.

This will improve purity and not change completeness (a true detection will not be discarded). Because a true detection has flux in its vicinity and dilation will catch more of that flux and increase the S/N. So on a true detection, the final S/N cannot be less than pseudo-detections.

However, in many real images bad processing creates artifacts that cannot be accurately removed by the Sky subtraction. In such cases, this option will decrease the completeness (will artificially discard true detections). So this feature is not default and should to be explicitly called when you know the noise is clean.

`--checkdetection`

Every step of the detection process will be added as an extension to a file with the suffix `_det.fits`. Going through each would just be a repeat of the explanations above and also of those in Akhlaghi and Ichikawa (2015). The extension label should be sufficient to recognize which step you are observing. Viewing all the steps can be the best guide in choosing the best set of parameters. With this option, NoiseChisel will abort as soon as a snapshot of all the detection process is saved. This behavior can be disabled with `--continueaftercheck`.

`--checksky`

Check the derivation of the final sky and its standard deviation values on the mesh grid. With this option, NoiseChisel will abort as soon as the sky value is estimated over the image (on each tile). This behavior can be disabled with `--continueaftercheck`. By default the output will have the same pixel size as

the input, but with the `--oneelempertile` option, only one pixel will be used for each tile (see Section 4.1.2.2 [Processing options], page 98).

7.2.2.3 NoiseChisel output

NoiseChisel's output is a multi-extension FITS file. The main extension/dataset is a (binary) detection map. It has the same size as the input but with only two possible values for all pixels: 0 (for pixels identified as noise) and 1 (for those identified as signal/detections). The detection map is followed by a Sky and Sky standard deviation dataset (which are calculated from the binary image). By default (when `--rawoutput` isn't called), NoiseChisel will also subtract the Sky value from the input and save the sky-subtracted input as the first extension in the output with data. The zero-th extension (that contains no data), contains NoiseChisel's configuration as FITS keywords, see Section 4.9 [Output FITS files], page 125.

The name of the output file can be set by giving a value to `--output` (this is a common option between all programs and is therefore discussed in Section 4.1.2.1 [Input/Output options], page 95). If `--output` isn't used, the input name will be suffixed with `_detected.fits` and used as output, see Section 4.8 [Automatic output], page 124. If any of the options starting with `--check*` are given, NoiseChisel won't complete and will abort as soon as the respective check images are created. For more information on the different check images, see the description for the `--check*` options in Section 7.2.2.2 [Detection options], page 234, (this can be disabled with `--continueaftercheck`).

The last two extensions of the output are the Sky and its Standard deviation, see Section 7.1.3 [Sky value], page 211, for a complete explanation. They are calculated on the tile grid that you defined for NoiseChisel. By default these datasets will have the same size as the input, but with all the pixels in one tile given one value. To be more space-efficient (keep only one pixel per tile), you can use the `--oneelempertile` option, see Section 4.7 [Tessellation], page 123.

To inspect any of NoiseChisel's output files, assuming you use SAO DS9, you can configure your Graphic User Interface (GUI) to open NoiseChisel's output as a multi-extension data cube. This will allow you to flip through the different extensions and visually inspect the results. This process has been described for the GNOME GUI (most common GUI in GNU/Linux operating systems) in Section B.1.1 [Viewing multiextension FITS images], page 469.

NoiseChisel's output configuration options are described in detail below.

`--continueaftercheck`
: Continue NoiseChisel after any of the options starting with `--check` (see Section 7.2.2.2 [Detection options], page 234. NoiseChisel involves many steps and as a result, there are many checks, allowing you to inspect the status of the processing. The results of each step affect the next steps of processing. Therefore, when you want to check the status of the processing at one step, the time spent to complete NoiseChisel is just wasted/distracting time.

  To encourage easier experimentation with the option values, when you use any of the NoiseChisel options that start with `--check`, NoiseChisel will abort once its desired extensions have been written. With `--continueaftercheck` option,

you can disable this behavior and ask NoiseChisel to continue with the rest of the processing, even after the requested check files are complete.

`--ignoreblankintiles`
: Don't set the input's blank pixels to blank in the tiled outputs (for example Sky and Sky standard deviation extensions of the output). This is only applicable when the tiled output has the same size as the input, in other words, when `--oneelempertile` isn't called.

  By default, blank values in the input (commonly on the edges which are outside the survey/field area) will be set to blank in the tiled outputs also. But in other scenarios this default behavior is not desired: for example if you have masked something in the input, but want the tiled output under that also.

`-l`
`--label`
: Run a connected-components algorithm on the finally detected pixels to identify which pixels are connected to which. By default the main output is a binary dataset with only two values: 0 (for noise) and 1 (for signal/detections). See Section 7.2.2.3 [NoiseChisel output], page 241, for more.

  The purpose of NoiseChisel is to detect targets that are extended and diffuse, with outer parts that sink into the noise very gradually (galaxies and stars for example). Since NoiseChisel digs down to extremely low surface brightness values, many such targets will commonly be detected together as a single large body of connected pixels.

  To properly separate connected objects, sophisticated segmentation methods are commonly necessary on NoiseChisel's output. Gnuastro has the dedicated Section 7.3 [Segment], page 243, program for this job. Since input images are commonly large and can take a significant volume, the extra volume necessary to store the labels of the connected components in the detection map (which will be created with this `--label` option, in 32-bit signed integer type) can thus be a major waste of space. Since the default output is just a binary dataset, an 8-bit unsigned dataset is enough.

  The binary output will also encourage users to segment the result separately prior to doing higher-level analysis. As an alternative to `--label`, if you have the binary detection image, you can use the `connected-components` operator in Gnuastro's Arithmetic program to identify regions that are connected with each other. For example with this command (assuming NoiseChisel's output is called `nc.fits`):

```
$ astarithmetic nc.fits 2 connected-components -hDETECTIONS
```

`--rawoutput`
: Don't include the Sky-subtracted input image as the first extension of the output. By default, the Sky-subtracted input is put in the first extension of the output. The next extensions are NoiseChisel's main outputs described above.

  The extra Sky-subtracted input can be convenient in checking NoiseChisel's output and comparing the detection map with the input: visually see if everything you expected is detected (reasonable completeness) and that you don't have too many false detections (reasonable purity). This visual inspection is

simplified if you use SAO DS9 to view NoiseChisel's output as a multi-extension data-cube, see Section B.1.1 [Viewing multiextension FITS images], page 469.

When you are satisfied with your NoiseChisel configuration (therefore you don't need to check on every run), or you want to archive/transfer the outputs, or the datasets become large, or you are running NoiseChisel as part of a pipeline, this Sky-subtracted input image can be a significant burden (take up a large volume). The fact that the input is also noisy, makes it hard to compress it efficiently.

In such cases, this `--rawoutput` can be used to avoid the extra sky-subtracted input in the output. It is always possible to easily produce the Sky-subtracted dataset from the input (assuming it is in extension `1` of `in.fits`) and the `SKY` extension of NoiseChisel's output (let's call it `nc.fits`) with a command like below (assuming NoiseChisel wasn't run with `--oneelempertile`, see Section 4.7 [Tessellation], page 123):

```
$ astarithmetic in.fits nc.fits - -h1 -hSKY
```

Save space: with the `--rawoutput` and `--oneelempertile`, NoiseChisel's output will only be one binary detection map and two much smaller arrays with one value per tile. Since none of these have noise they can be compressed very effectively (without any loss of data) with exceptionally high compression ratios. This makes it easy to archive, or transfer, NoiseChisel's output even on huge datasets. To compress it with the most efficient method (take up less volume), run the following command:

```
$ gzip --best noisechisel_output.fits
```

The resulting `.fits.gz` file can then be fed into any of Gnuastro's programs directly, or viewed in viewers like SAO DS9, without having to decompress it separately (they will just take a little longer, because they have to internally decompress it before starting).

7.3 Segment

Once signal is separated from noise (for example with Section 7.2 [NoiseChisel], page 225), you have a binary dataset: each pixel is either signal (1) or noise (0). Signal (for example every galaxy in your image) has been "detected", but all detections have a label of 1. Therefore while we know which pixels contain signal, we still can't find out how many galaxies they contain or which detected pixels correspond to which galaxy. At the lowest (most generic) level, detection is a kind of segmentation (segmenting the the whole dataset into signal and noise, see Section 7.2 [NoiseChisel], page 225). Here, we'll define segmentation only on signal: to separate and find sub-structure within the detections.

If the targets are clearly separated, or their detected regions aren't touching, a simple connected components[9] algorithm (very basic segmentation) is enough to separate the regions that are touching/connected. This is such a basic and simple form of segmentation that Gnuastro's Arithmetic program has an operator for it: see `connected-components` in Section 6.2.2 [Arithmetic operators], page 162. Assuming the binary dataset is called `binary.fits`, you can use it with a command like this:

[9] `https://en.wikipedia.org/wiki/Connected-component_labeling`

```
$ astarithmetic binary.fits 2 connected-components
```

You can even do a very basic detection (a threshold, say at value `100`) *and* segmentation in Arithmetic with a single command like below:

```
$ astarithmetic in.fits 100 gt 2 connected-components
```

However, in most astronomical situations our targets are not nicely separated or have a sharp boundary/edge (for a threshold to suffice): they touch (for example merging galaxies), or are simply in the same line-of-sight (which is much more common). This causes their images to overlap.

In particular, when you do your detection with NoiseChisel, you will detect signal to very low surface brightness limits: deep into the faint wings of galaxies or bright stars (which can extend very far and irregularly from their center). Therefore, it often happens that several galaxies are detected as one large detection. Since they are touching, a simple connected components algorithm will not suffice. It is therefore necessary to do a more sophisticated segmentation and break up the detected pixels (even those that are touching) into multiple target objects as accurately as possible.

Segment will use a detection map and its corresponding dataset to find sub-structure over the detected areas and use them for its segmentation. Until Gnuastro version 0.6 (released in 2018), Segment was part of Section 7.2 [NoiseChisel], page 225. Therefore, similar to NoiseChisel, the best place to start reading about Segment and understanding what it does (with many illustrative figures) is Section 3.2 of Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`).

As a summary, Segment first finds true *clump*s over the detections. Clumps are associated with local maxima/minima[10] and extend over the neighboring pixels until they reach a local minimum/maximum (*river*/*watershed*). By default, Segment will use the distribution of clump signal-to-noise ratios over the undetected regions as reference to find "true" clumps over the detections. Using the undetected regions can be disabled by directly giving a signal-to-noise ratio to `--clumpsnthresh`.

The true clumps are then grown to a certain threshold over the detections. Based on the strength of the connections (rivers/watersheds) between the grown clumps, they are considered parts of one *object* or as separate *object*s. See Section 3.2 of Akhlaghi and Ichikawa [2015] (link above) for more. Segment's main output are thus two labeled datasets: 1) clumps, and 2) objects. See Section 7.3.2.3 [Segment output], page 253, for more.

To start learning about Segment, especially in relation to detection (Section 7.2 [NoiseChisel], page 225) and measurement (Section 7.4 [MakeCatalog], page 255), the recommended references are Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`) and Akhlaghi [2016] (`https://arxiv.org/abs/1611.06387`).

Those papers cannot be updated any more but the software will evolve. For example Segment became a separate program (from NoiseChisel) in 2018 (after those papers were published). Therefore this book is the definitive reference. To help in the transition from those papers to the software you are using, see Section 7.3.1 [Segment changes after publication], page 245. Finally, in Section 7.3.2 [Invoking Segment], page 246, we'll discuss Segment's inputs, outputs and configuration options.

[10] By default the maximum is used as the first clump pixel, to define clumps based on local minima, use the `--minima` option.

7.3.1 Segment changes after publication

Segment's main algorithm and working strategy were initially defined and introduced in Section 3.2 of Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`). Prior to Gnuastro version 0.6 (released 2018), one program (NoiseChisel) was in charge of detection *and* segmentation. to increase creativity and modularity, NoiseChisel's segmentation features were spun-off into a separate program (Segment). It is strongly recommended to read that paper for a good understanding of what Segment does, how it relates to detection, and how each parameter influences the output. That paper has a large number of figures showing every step on multiple mock and real examples.

However, the paper cannot be updated anymore, but Segment has evolved (and will continue to do so): better algorithms or steps have been (and will be) found. This book is thus the final and definitive guide to Segment. The aim of this section is to make the transition from the paper to your installed version, as smooth as possible through the list below. For a more detailed list of changes in previous Gnuastro releases/versions, please follow the `NEWS` file[11].

- Since the spin-off from NoiseChisel, the default kernel to smooth the input for convolution has a FWHM of 1.5 pixels (still a Gaussian). This is slightly less than NoiseChisel's default kernel (which has a FWHM of 2 pixels). This enables the better detection of sharp clumps: as the kernel gets wider, the lower signal-to-noise (but sharp/small) clumps will be washed away into the noise. You can use MakeProfiles to build your own kernel if this is too sharp/wide for your purpose. For more, see the `--kernel` option in Section 7.3.2.1 [Segment input], page 247.

  The ability to use a different convolution kernel for detection and segmentation is one example of how separating detection from segmentation into separate programs can increase creativity. In detection, you want to detect the diffuse and extended emission, but in segmentation, you want to detect sharp peaks.

- The criteria to select true from false clumps is the peak significance. It is defined to be the difference between the clump's peak value (C_c) and the highest valued river pixel around that clump (R_c). Both are calculated on the convolved image (signified by the c subscript). To avoid absolute values (differing from dataset to dataset), $C_c - R_c$ is then divided by the Sky standard deviation under the river pixel used (σ_r) as shown below:

$$\frac{C_c - R_c}{\sigma_r}$$

  When `--minima` is given, the nominator becomes $R_c - C_c$.

  The input Sky standard deviation dataset (`--std`) is assumed to be for the unconvolved image. Therefore a constant factor (related to the convolution kernel) is necessary to convert this into an absolute peak significance[12]. As far as Segment is concerned, the

[11] The `NEWS` file is present in the released Gnuastro tarball, see Section 3.2.1 [Release tarball], page 71.

[12] To get an estimate of the standard deviation correction factor between the input and convolved images, you can take the following steps: 1) Mask (set to NaN) all detections on the convolved image with the `where` operator or Section 6.2 [Arithmetic], page 161. 2) Calculate the standard deviation of the undetected (non-masked) pixels of the convolved image with the `--sky` option of Section 7.1 [Statistics],

absolute value of this correction factor is irrelevant: because it uses the ambient noise (undetected regions) to find the numerical threshold of this fraction and applies that over the detected regions.

A distribution's extremum (maximum or minimum) values, used in the new criteria, are strongly affected by scatter. On the other hand, the convolved image has much less scatter[13]. Therefore $C_c - R_c$ is a more reliable (with less scatter) measure to identify signal than $C - R$ (on the unconvolved image).

Initially, the total clump signal-to-noise ratio of each clump was used, see Section 3.2.1 of Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`). Therefore its completeness decreased dramatically when clumps were present on gradients. In tests, this measure proved to be more successful in detecting clumps on gradients and on flatter regions simultaneously.

- With the new `--minima` option, it is now possible to detect inverse clumps (for example absorption features). In such cases, the clump should be built from its smallest value.

7.3.2 Invoking Segment

Segment will identify substructure within the detected regions of an input image. Segment's output labels can be directly used for measurements (for example with Section 7.4 [MakeCatalog], page 255). The executable name is `astsegment` with the following general template

```
$ astsegment [OPTION ...] InputImage.fits
```

One line examples:

```
## Segment NoiseChisel's detected regions.
$ astsegment default-noisechisel-output.fits

## Use a hand-input S/N value for keeping true clumps
## (avoid finding the S/N using the undetected regions).
$ astsegment nc-out.fits --clumpsnthresh=10

## Inspect all the segmentation steps after changing a parameter.
$ astsegment input.fits --snquant=0.9 --checksegmentaion

## Use the fixed value of 0.01 for the input's Sky standard deviation
## (in the units of the input), and assume all the pixels are a
## detection (for example a large structure extending over the whole
## image), and only keep clumps with S/N>10 as true clumps.
$ astsegment in.fits --std=0.01 --detection=all --clumpsnthresh=10
```

If Segment is to do processing (for example you don't want to get help, or see the values of each option), at least one input dataset is necessary along with detection and error information, either as separate datasets (per-pixel) or fixed values, see Section 7.3.2.1 [Segment

page 208, (which also calculates the Sky standard deviation). Just make sure the tessellation settings of Statistics and NoiseChisel are the same (you can check with the `-P` option). 3) Divide the two standard deviation datasets to get the correction factor.

[13] For more on the effect of convolution on a distribution, see Section 3.1.1 of Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`).

input], page 247. Segment shares a large set of common operations with other Gnuastro programs, mainly regarding input/output, general processing steps, and general operating modes. To help in a unified experience between all of Gnuastro's programs, these common operations have the same names and defined in Section 4.1.2 [Common options], page 95.

As in all Gnuastro programs, options can also be given to Segment in configuration files. For a thorough description of Gnuastro's configuration file parsing, please see Section 4.2 [Configuration files], page 106. All of Segment's options with a short description are also always available on the command-line with the `--help` option, see Section 4.3 [Getting help], page 109. To inspect the option values without actually running Segment, append your command with `--printparams` (or `-P`).

To help in easy navigation between Segment's options, they are separately discussed in the three sub-sections below: Section 7.3.2.1 [Segment input], page 247, discusses how you can customize the inputs to Segment. Section 7.3.2.2 [Segmentation options], page 250, is devoted to options specific to the high-level segmentation process. Finally, in Section 7.3.2.3 [Segment output], page 253, we'll discuss options that affect Segment's output.

7.3.2.1 Segment input

Besides the input dataset (for example astronomical image), Segment also needs to know the Sky standard deviation and the regions of the dataset that it should segment. The values dataset is assumed to be Sky subtracted by default. If it isn't, you can ask Segment to subtract the Sky internally by calling `--sky`. For the rest of this discussion, we'll assume it is already sky subtracted.

The Sky and its standard deviation can be a single value (to be used for the whole dataset) or a separate dataset (for a separate value per pixel). If a dataset is used for the Sky and its standard deviation, they must either be the size of the input image, or have a single value per tile (generated with `--oneelempertile`, see Section 4.1.2.2 [Processing options], page 98, and Section 4.7 [Tessellation], page 123).

The detected regions/pixels can be specified as a detection map (for example see Section 7.2.2.3 [NoiseChisel output], page 241). If `--detection=all`, Segment won't read any detection map and assume the whole input is a single detection. For example when the dataset is fully covered by a large nearby galaxy/globular cluster.

When dataset are to be used for any of the inputs, Segment will assume they are multiple extensions of a single file by default (when `--std` or `--detection` aren't called). For example NoiseChisel's default output Section 7.2.2.3 [NoiseChisel output], page 241. When the Sky-subtracted values are in one file, and the detection and Sky standard deviation are in another, you just need to use `--detection`: in the absence of `--std`, Segment will look for both the detection labels and Sky standard deviation in the file given to `--detection`. Ultimately, if all three are in separate files, you need to call both `--detection` and `--std`.

The extensions of the three mandatory inputs can be specified with `--hdu`, `--dhdu`, and `--stdhdu`. For a full discussion on what to give to these options, see the description of `--hdu` in Section 4.1.2.1 [Input/Output options], page 95. To see their default values (along with all the other options), run Segment with the `--printparams` (or `-P`) option. Just recall that in the absence of `--detection` and `--std`, all three are assumed to be in the same file. If you only want to see Segment's default values for HDUs on your system, run this command:

```
$ astsegment -P | grep hdu
```

By default Segment will convolve the input with a kernel to improve the signal-to-noise ratio of true peaks. If you already have the convolved input dataset, you can pass it directly to Segment for faster processing (using the `--convolved` and `--chdu` options). Just don't forget that the convolved image must also be Sky-subtracted before calling Segment. If a value/file is given to `--sky`, the convolved values will also be Sky subtracted internally. Alternatively, if you prefer to give a kernel (with `--kernel` and `--khdu`), Segment can do the convolution internally. To disable convolution, use `--kernel=none`.

`--sky=STR/FLT`

The Sky value(s) to subtract from the input. This option can either be given a constant number or a file name containing a dataset (multiple values, per pixel or per tile). By default, Segment will assume the input dataset is Sky subtracted, so this option is not mandatory.

If the value can't be read as a number, it is assumed to be a file name. When the value is a file, the extension can be specified with `--skyhdu`. When its not a single number, the given dataset must either have the same size as the output or the same size as the tessellation (so there is one pixel per tile, see Section 4.7 [Tessellation], page 123).

When this option is given, its value(s) will be subtracted from the input and the (optional) convolved dataset (given to `--convolved`) prior to starting the segmentation process.

`--skyhdu=STR/INT`

The HDU/extension containing the Sky values. This is mandatory when the value given to `--sky` is not a number. Please see the description of `--hdu` in Section 4.1.2.1 [Input/Output options], page 95, for the different ways you can identify a special extension.

`--std=STR/FLT`

The Sky standard deviation value(s) corresponding to the input. The value can either be a constant number or a file name containing a dataset (multiple values, per pixel or per tile). The Sky standard deviation is mandatory for Segment to operate.

If the value can't be read as a number, it is assumed to be a file name. When the value is a file, the extension can be specified with `--skyhdu`. When its not a single number, the given dataset must either have the same size as the output or the same size as the tessellation (so there is one pixel per tile, see Section 4.7 [Tessellation], page 123).

When this option is not called, Segment will assume the standard deviation is a dataset and in a HDU/extension (`--stdhdu`) of another one of the input file(s). If a file is given to `--detection`, it will assume that file contains the standard deviation dataset, otherwise, it will look into input filename (the main argument, without any option).

`--stdhdu=INT/STR`

The HDU/extension containing the Sky standard deviation values, when the value given to `--std` is a file name. Please see the description of `--hdu` in

Section 4.1.2.1 [Input/Output options], page 95, for the different ways you can identify a special extension.

`--variance`
: The input Sky standard deviation value/dataset is actually variance. When this option is called, the square root of input Sky standard deviation (see `--std`) is used internally, not its raw value(s).

`-d STR`
`--detection=STR`
: Detection map to use for segmentation. If given a value of `all`, Segment will assume the whole dataset must be segmented, see below. If a detection map is given, the extension can be specified with `--dhdu`. If not given, Segment will assume the desired HDU/extension is in the main input argument (input file specified with no option).

  The final segmentation (clumps or objects) will only be over the non-zero pixels of this detection map. The dataset must have the same size as the input image. Only datasets with an integer type are acceptable for the labeled image, see Section 4.5 [Numeric data types], page 115. If your detection map only has integer values, but it is stored in a floating point container, you can use Gnuastro's Arithmetic program (see Section 6.2 [Arithmetic], page 161) to convert it to an integer container, like the example below:

```
$ astarithmetic float.fits int32 --output=int.fits
```

  It may happen that the whole input dataset is covered by signal, for example when working on parts of the Andromeda galaxy, or nearby globular clusters (that cover the whole field of view). In such cases, segmentation is necessary over the complete dataset, not just specific regions (detections). By default Segment will first use the undetected regions as a reference to find the proper signal-to-noise ratio of "true" clumps (give a purity level specified with `--snquant`). Therefore, in such scenarios you also need to manually give a "true" clump signal-to-noise ratio with the `--clumpsnthresh` option to disable looking into the undetected regions, see Section 7.3.2.2 [Segmentation options], page 250. In such cases, is possible to make a detection map that only has the value `1` for all pixels (for example using Section 6.2 [Arithmetic], page 161), but for convenience, you can also use `--detection=all`.

`--dhdu`
: The HDU/extension containing the detection map given to `--detection`. Please see the description of `--hdu` in Section 4.1.2.1 [Input/Output options], page 95, for the different ways you can identify a special extension.

`-k STR`
`--kernel=STR`
: The kernel used to convolve the input image. The usage of this option is identical to NoiseChisel's `--kernel` option (Section 7.2.2.1 [NoiseChisel input], page 231). Please see the descriptions there for more. To disable convolution, you can give it a value of `none`.

`--khdu`
: The HDU/extension containing the kernel used for convolution. For acceptable values, please see the description of `--hdu` in Section 4.1.2.1 [Input/Output options], page 95.

`--convolved`
: The convolved image to avoid internal convolution by Segment. The usage of this option is identical to NoiseChisel's `--convolved` option. Please see Section 7.2.2.1 [NoiseChisel input], page 231, for a thorough discussion of the usefulness and best practices of using this option.

  If you want to use the same convolution kernel for detection (with Section 7.2 [NoiseChisel], page 225) and segmentation, with this option, you can use the same convolved image (that is also available in NoiseChisel) and avoid two convolutions. However, just be careful to use the input to NoiseChisel as the input to Segment also, then use the `--sky` and `--std` to specify the Sky and its standard deviation (from NoiseChisel's output). Recall that when NoiseChisel is not called with `--rawoutput`, the first extension of NoiseChisel's output is the *Sky-subtracted* input (see Section 7.2.2.3 [NoiseChisel output], page 241). So if you use the same convolved image that you fed to NoiseChisel, but use NoiseChisel's output with Segment's `--convolved`, then the convolved image won't be Sky subtracted.

`--chdu`
: The HDU/extension containing the convolved image (given to `--convolved`). For acceptable values, please see the description of `--hdu` in Section 4.1.2.1 [Input/Output options], page 95.

`-L INT[,INT]`
`--largetilesize=INT[,INT]`
: The size of the large tiles to use for identifying the clump S/N threshold over the undetected regions. The usage of this option is identical to NoiseChisel's `--largetilesize` option (Section 7.2.2.1 [NoiseChisel input], page 231). Please see the descriptions there for more.

  The undetected regions can be a significant fraction of the dataset and finding clumps requires sorting of the desired regions, which can be slow. To speed up the processing, Segment finds clumps in the undetected regions over separate large tiles. This allows it to have to sort a much smaller set of pixels and also to treat them independently and in parallel. Both these issues greatly speed it up. Just be sure to not decrease the large tile sizes too much (less than 100 pixels in each dimension). It is important for them to be much larger than the clumps.

7.3.2.2 Segmentation options

The options below can be used to configure every step of the segmentation process in the Segment program. For a more complete explanation (with figures to demonstrate each step), please see Section 3.2 of Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`), and also Section 7.3 [Segment], page 243. By default, Segment will follow the procedure described in the paper to find the S/N threshold based on the noise properties. This can be disabled by directly giving a trustable signal-to-noise ratio to the `--clumpsnthresh` option.

Recall that you can always see the full list of Gnuastro's options with the `--help` (see Section 4.3 [Getting help], page 109), or `--printparams` (or `-P`) to see their values (see Section 4.1.2.3 [Operating mode options], page 100).

`-B FLT`
`--minskyfrac=FLT`
Minimum fraction (value between 0 and 1) of Sky (undetected) areas in a large tile. Only (large) tiles with a fraction of undetected pixels (Sky) greater than this value will be used for finding clumps. The clumps found in the undetected areas will be used to estimate a S/N threshold for true clumps. Therefore this is an important option (to decrease) in crowded fields. Operationally, this is almost identical to NoiseChisel's `--minskyfrac` option (Section 7.2.2.2 [Detection options], page 234). Please see the descriptions there for more.

`--minima` Build the clumps based on the local minima, not maxima. By default, clumps are built starting from local maxima (see Figure 8 of Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`)). Therefore, this option can be useful when you are searching for true local minima (for example absorption features).

`-m INT`
`--snminarea=INT`
The minimum area which a clump in the undetected regions should have in order to be considered in the clump Signal to noise ratio measurement. If this size is set to a small value, the Signal to noise ratio of false clumps will not be accurately found. It is recommended that this value be larger than the value to NoiseChisel's `--snminarea`. Because the clumps are found on the convolved (smoothed) image while the pseudo-detections are found on the input image. You can use `--checksn` and `--checksegmentation` to see if your chosen value is reasonable or not.

`--checksn`
Save the S/N values of the clumps over the sky and detected regions into separate tables. If `--tableformat` is a FITS format, each table will be written into a separate extension of one file suffixed with `_clumpsn.fits`. If it is plain text, a separate file will be made for each table (ending in `_clumpsn_sky.txt` and `_clumpsn_det.txt`). For more on `--tableformat` see Section 4.1.2.1 [Input/Output options], page 95.

You can use these tables to inspect the S/N values and their distribution (in combination with the `--checksegmentation` option to see where the clumps are). You can use Gnuastro's Section 7.1 [Statistics], page 208, to make a histogram of the distribution (ready for plotting in a text file, or a crude ASCII-art demonstration on the command-line).

With this option, Segment will abort as soon as the two tables are created. This allows you to inspect the steps leading to the final S/N quantile threshold, this behavior can be disabled with `--continueaftercheck`.

`--minnumfalse=INT`
The minimum number of clumps over undetected (Sky) regions to identify the requested Signal-to-Noise ratio threshold. Operationally, this is almost identical to NoiseChisel's `--minnumfalse` option (Section 7.2.2.2 [Detection options], page 234). Please see the descriptions there for more.

`-c FLT`
`--snquant=FLT`
: The quantile of the signal-to-noise ratio distribution of clumps in undetected regions, used to define true clumps. After identifying all the usable clumps in the undetected regions of the dataset, the given quantile of their signal-to-noise ratios is used to define a the signal-to-noise ratio of a "true" clump. Effectively, this can be seen as an inverse p-value measure. See Figure 9 and Section 3.2.1 of Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`) for a complete explanation. The full distribution of clump signal-to-noise ratios over the undetected areas can be saved into a table with `--checksn` option and visually inspected with `--checksegmentation`.

`-v`
`--keepmaxnearriver`
: Keep a clump whose maximum (minimum if `--minima` is called) flux is 8-connected to a river pixel. By default such clumps over detections are considered to be noise and are removed irrespective of their brightness (see Section 8.1.3 [Flux Brightness and magnitude], page 290). Over large profiles, that sink into the noise very slowly, noise can cause part of the profile (which was flat without noise) to become a very large and with a very high Signal to noise ratio. In such cases, the pixel with the maximum flux in the clump will be immediately touching a river pixel.

`-s FLT`
`--clumpsnthresh=FLT`
: The signal-to-noise threshold for true clumps. If this option is given, then the segmentation options above will be ignored and the given value will be directly used to identify true clumps over the detections. This can be useful if you have a large dataset with similar noise properties. You can find a robust signal-to-noise ratio based on a (sufficiently large) smaller portion of the dataset. Afterwards, with this option, you can speed up the processing on the whole dataset. Other scenarios where this option may be useful is when, the image might not contain enough/any Sky regions.

`-G FLT`
`--gthresh=FLT`
: Threshold (multiple of the sky standard deviation added with the sky) to stop growing true clumps. Once true clumps are found, they are set as the basis to segment the detected region. They are grown until the threshold specified by this option.

`-y INT`
`--minriverlength=INT`
: The minimum length of a river between two grown clumps for it to be considered in signal-to-noise ratio estimations. Similar to `--snminarea`, if the length of the river is too short, the signal-to-noise ratio can be noisy and unreliable. Any existing rivers shorter than this length will be considered as non-existent, independent of their Signal to noise ratio. The clumps are grown on the input image, therefore this value can be smaller than the value given to `--snminarea`.

Recall that the clumps were defined on the convolved image so `--snminarea` should be larger.

`-O FLT`
`--objbordersn=FLT`
The maximum Signal to noise ratio of the rivers between two grown clumps in order to consider them as separate 'objects'. If the Signal to noise ratio of the river between two grown clumps is larger than this value, they are defined to be part of one 'object'. Note that the physical reality of these 'objects' can never be established with one image, or even multiple images from one broad-band filter. Any method we devise to define 'object's over a detected region is ultimately subjective.

Two very distant galaxies or satellites in one halo might lie in the same line of sight and be detected as clumps on one detection. On the other hand, the connection (through a spiral arm or tidal tail for example) between two parts of one galaxy might have such a low surface brightness that they are broken up into multiple detections or objects. In fact if you have noticed, exactly for this purpose, this is the only Signal to noise ratio that the user gives into NoiseChisel. The 'true' detections and clumps can be objectively identified from the noise characteristics of the image, so you don't have to give any hand input Signal to noise ratio.

`--checksegmentation`
A file with the suffix `_seg.fits` will be created. This file keeps all the relevant steps in finding true clumps and segmenting the detections into multiple objects in various extensions. Having read the paper or the steps above. Examining this file can be an excellent guide in choosing the best set of parameters. Note that calling this function will significantly slow NoiseChisel. In verbose mode (without the `--quiet` option, see Section 4.1.2.3 [Operating mode options], page 100) the important steps (along with their extension names) will also be reported.

With this option, NoiseChisel will abort as soon as the two tables are created. This behavior can be disabled with `--continueaftercheck`.

7.3.2.3 Segment output

The main output of Segment are two label datasets (with integer types, separating the dataset's elements into different classes). They have HDU/extension names of `CLUMPS` and `OBJECTS`.

Similar to all Gnuastro's FITS outputs, the zero-th extension/HDU of the main output file only contains header keywords and image or table. It contains the Segment input files and parameters (option names and values) as FITS keywords. Note that if an option name is longer than 8 characters, the keyword name is the second word. The first word is `HIERARCH`. Also note that according to the FITS standard, the keyword names must be in capital letters, therefore, if you want to use Grep to inspect these keywords, use the `-i` option, like the example below.

```
$ astfits image_segmented.fits -h0 | grep -i snquant
```

By default, besides the `CLUMPS` and `OBJECTS` extensions, Segment's output will also contain the (technically redundant) input dataset and the sky standard deviation dataset (if it wasn't a constant number). This can help in visually inspecting the result when viewing the images as a "Multi-extension data cube" in SAO DS9 for example (see Section B.1.1 [Viewing multiextension FITS images], page 469). You can simply flip through the extensions and see the same region of the image and its corresponding clumps/object labels. It also makes it easy to feed the output (as one file) into MakeCatalog when you intend to make a catalog afterwards (see Section 7.4 [MakeCatalog], page 255. To remove these redundant extensions from the output (for example when designing a pipeline), you can use `--rawoutput`.

The `OBJECTS` and `CLUMPS` extensions can be used as input into Section 7.4 [MakeCatalog], page 255, to generate a catalog for higher-level analysis. If you want to treat each clump separately, you can give a very large value (or even a NaN, which will always fail) to the `--gthresh` option (for example `--gthresh=1e10` or `--gthresh=nan`), see Section 7.3.2.2 [Segmentation options], page 250.

For a complete definition of clumps and objects, please see Section 3.2 of Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`) and Section 7.3.2.2 [Segmentation options], page 250. The clumps are "true" local maxima (minima if `--minima` is called) and their surrounding pixels until a local minimum/maximum (caused by noise fluctuations, or another "true" clump). Therefore it may happen that some of the input detections aren't covered by clumps at all (very diffuse objects without any strong peak), while some objects may contain many clumps. Even in those that have clumps, there will be regions that are too diffuse. The diffuse regions (within the input detected regions) are given a negative label (-1) to help you separate them from the undetected regions (with a value of zero).

Each clump is labeled with respect to its host object. Therefore, if an object has three clumps for example, the clumps within it have labels 1, 2 and 3. As a result, if an initial detected region has multiple objects, each with a single clump, all the clumps will have a label of 1. The total number of clumps in the dataset is stored in the `NCLUMPS` keyword of the `CLUMPS` extension and printed in the verbose output of Segment (when `--quiet` is not called).

The `OBJECTS` extension of the output will give a positive counter/label to every detected pixel in the input. As described in Akhlaghi and Ichikawa [2015], the true clumps are grown until a certain threshold. If the grown clumps touch other clumps and the connection is strong enough, they are considered part of the same *object*. Once objects (grown clumps) are identified, they are grown to cover the whole detected area.

The options to configure the output of Segment are listed below:

`--continueaftercheck`
: Don't abort Segment after producing the check image(s). The usage of this option is identical to NoiseChisel's `--continueaftercheck` option (Section 7.2.2.1 [NoiseChisel input], page 231). Please see the descriptions there for more.

`--onlyclumps`
: Abort Segment after finding true clumps and don't continue with finding options. Therefore, no `OBJECTS` extension will be present in the output. Each true clump in `CLUMPS` will get a unique label, but diffuse regions will still have a negative value.

To make a catalog of the clumps, the input detection map (where all the labels are one) can be fed into Section 7.4 [MakeCatalog], page 255, along with the input detection map to Segment (that only had a value of `1` for all detected pixels) with `--clumpscat`. In this way, MakeCatalog will assume all the clumps belong to a single "object".

`--grownclumps`
In the output `CLUMPS` extension, store the grown clumps. If a detected region contains no clumps or only one clump, then it will be fully given a label of `1` (no negative valued pixels).

`--rawoutput`
Only write the `CLUMPS` and `OBJECTS` datasets in the output file. Without this option (by default), the first and last extensions of the output will the Sky-subtracted input dataset and the Sky standard deviation dataset (if it wasn't a number). When the datasets are small, these redundant extensions can make it convenient to inspect the results visually or feed the output to Section 7.4 [MakeCatalog], page 255, for measurements. Ultimately both the input and Sky standard deviation datasets are redundant (you had them before running Segment). When the inputs are large/numerous, these extra dataset can be a burden.

Save space: with the `--rawoutput`, Segment's output will only be two labeled datasets (only containing integers). Since they have no noise, such datasets can be compressed very effectively (without any loss of data) with exceptionally high compression ratios. You can use the following command to compress it with the best ratio:

```
$ gzip --best segment_output.fits
```

The resulting `.fits.gz` file can then be fed into any of Gnuastro's programs directly, without having to decompress it separately (it will just take them a little longer, because they have to decompress it internally before use).

7.4 MakeCatalog

At the lowest level, a dataset (for example an image) is just a collection of values, placed after each other in any number of dimensions (for example an image is a 2D dataset). Each data-element (pixel) just has two properties: its position (relative to the rest) and its value. In higher-level analysis, an entire dataset (an image for example) is rarely treated as a singular entity[14]. You usually want to know/measure the properties of the (separate) scientifically interesting targets that are embedded in it. For example the magnitudes, positions and elliptical properties of the galaxies that are in the image.

MakeCatalog is Gnuastro's program for localized measurements over a dataset. In other words, MakeCatalog is Gnuastro's program to convert low-level datasets (like images), to high level catalogs. The role of MakeCatalog in a scientific analysis and the benefits of its model (where detection/segmentation is separated from measurement) is discussed in

[14] You can derive the over-all properties of a complete dataset (1D table column, 2D image, or 3D data-cube) treated as a single entity with Gnuastro's Statistics program (see Section 7.1 [Statistics], page 208).

Akhlaghi [2016] (`https://arxiv.org/abs/1611.06387v1`)[15] and summarized in Section 7.4.1 [Detection and catalog production], page 257. We strongly recommend reading this short paper for a better understanding of this methodology. Understanding the effective usage of MakeCatalog, will thus also help effective use of other (lower-level) Gnuastro's programs like Section 7.2 [NoiseChisel], page 225, or Section 7.3 [Segment], page 243.

It is important to define your regions of interest for measurements *before* running MakeCatalog. MakeCatalog is specialized in doing measurements accurately and efficiently. Therefore MakeCatalog will not do detection, segmentation, or defining apertures on requested positions in your dataset. Following Gnuastro's modularity principle, there are separate and highly specialized and customizable programs in Gnuastro for these other jobs as shown below (for a usage example in a real-world analysis, see Section 2.2 [General program usage tutorial], page 24, and Section 2.3 [Detecting large extended targets], page 47).

- Section 6.2 [Arithmetic], page 161: Detection with a simple threshold.
- Section 7.2 [NoiseChisel], page 225: Advanced detection.
- Section 7.3 [Segment], page 243: Segmentation (substructure over detections).
- Section 8.1 [MakeProfiles], page 284: Aperture creation for known positions.

These programs will/can return labeled dataset(s) to be fed into MakeCatalog. A labeled dataset for measurement has the same size/dimensions as the input, but with integer valued pixels that have the label/counter for each sub-set of pixels that must be measured together. For example all the pixels covering one galaxy in an image, get the same label.

The requested measurements are then done on similarly labeled pixels. The final result is a catalog where each row corresponds to the measurements on pixels with a specific label. For example the flux weighted average position of all the pixels with a label of 42 will be written into the 42nd row of the output catalog/table's central position column[16]. Similarly, the sum of all these pixels will be the 42nd row in the brightness column and etc. Pixels with labels equal to, or smaller than, zero will be ignored by MakeCatalog. In other words, the number of rows in MakeCatalog's output is already known before running it (the maximum value of the labeled dataset).

Before getting into the details of running MakeCatalog (in Section 7.4.5 [Invoking MakeCatalog], page 266, we'll start with a discussion on the basics of its approach to separating detection from measurements in Section 7.4.1 [Detection and catalog production], page 257. A very important factor in any measurement is understanding its validity range, or limits. Therefore in Section 7.4.2 [Quantifying measurement limits], page 258, we'll discuss how to estimate the reliability of the detection and basic measurements. This section will continue with a derivation of elliptical parameters from the labeled datasets in Section 7.4.3 [Measuring elliptical parameters], page 262. For those who feel MakeCatalog's existing measurements/columns aren't enough and would like to add further measurements, in Section 7.4.4 [Adding new columns to MakeCatalog], page 264, a checklist of steps is provided for readily adding your own new measurements/columns.

[15] A published paper cannot undergo any more change, so this manual is the definitive guide.

[16] See Section 7.4.3 [Measuring elliptical parameters], page 262, for a discussion on this and the derivation of positional parameters, which includes the center.

7.4.1 Detection and catalog production

Most existing common tools in low-level astronomical data-analysis (for example SExtractor[17]) merge the two processes of detection and measurement (catalog production) in one program. However, in light of Gnuastro's modularized approach (modeled on the Unix system) detection is separated from measurements and catalog production. This modularity is therefore new to many experienced astronomers and deserves a short review here. Further discussion on the benefits of this methodology can be seen in Akhlaghi [2016] (`https://arxiv.org/abs/1611.06387v1`).

As discussed in the introduction of Section 7.4 [MakeCatalog], page 255, detection (identifying which pixels to do measurements on) can be done with different programs. Their outputs (a labeled dataset) can be directly fed into MakeCatalog to do the measurements and write the result as a catalog/table. Beyond that, Gnuastro's modular approach has many benefits that will become clear as you get more experienced in astronomical data analysis and want to be more creative in using your valuable data for the exciting scientific project you are working on. In short the reasons for this modularity can be classified as below:

- Simplicity/robustness of independent, modular tools: making a catalog is a logically separate process from labeling (detection, segmentation, or aperture production). A user might want to do certain operations on the labeled regions before creating a catalog for them. Another user might want the properties of the same pixels/objects in another image (another filter for example) to measure the colors or SED fittings.

  Here is an example of doing both: suppose you have images in various broad band filters at various resolutions and orientations. The image of one color will thus not lie exactly on another or even be in the same scale. However, it is imperative that the same pixels be used in measuring the colors of galaxies.

  To solve the problem, NoiseChisel can be run on the reference image to generate the labeled detection image. Afterwards, the labeled image can be warped into the grid of the other color (using Section 6.4 [Warp], page 199). MakeCatalog will then generate the same catalog for both colors (with the different labeled images). It is currently customary to warp the images to the same pixel grid, however, modification of the scientific dataset is very harmful for the data and creates correlated noise. It is much more accurate to do the transformations on the labeled image.

- Complexity of a monolith: Adding in a catalog functionality to the detector program will add several more steps (and many more options) to its processing that can equally well be done outside of it. This makes following what the program does harder for the users and developers, it can also potentially add many bugs.

  As an example, if the parameter you want to measure over one profile is not provided by the developers of MakeCatalog. You can simply open this tiny little program and add your desired calculation easily. This process is discussed in Section 7.4.4 [Adding new columns to MakeCatalog], page 264. However, if making a catalog was part of NoiseChisel for example, adding a new column/measurement would require a lot of energy to understand all the steps and internal structures of that huge program. It might even be so intertwined with its processing, that adding new columns might cause problems/bugs in its primary job (detection).

[17] `https://www.astromatic.net/software/sextractor`

7.4.2 Quantifying measurement limits

No measurement on a real dataset can be perfect: you can only reach a certain level/limit of accuracy. Therefore, a meaningful (scientific) analysis requires an understanding of these limits for the dataset and your analysis tools: different datasets have different noise properties and different detection methods (one method/algorith/software that is run with a different set of parameters is considered as a different detection method) will have different abilities to detect or measure certain kinds of signal (astronomical objects) and their properties in the dataset. Hence, quantifying the detection and measurement limitations with a particular dataset and analysis tool is the most crucial/critical aspect of any high-level analysis.

Here, we'll review some of the most general limits that are important in any astronomical data analysis and how MakeCatalog makes it easy to find them. Depending on the higher-level analysis, there are more tests that must be done, but these are relatively low-level and usually necessary in most cases. In astronomy, it is common to use the magnitude (a unitless scale) and physical units, see Section 8.1.3 [Flux Brightness and magnitude], page 290. Therefore the measurements discussed here are commonly used in units of magnitudes.

Surface brightness limit (of whole dataset)

As we make more observations on one region of the sky, and add the observations into one dataset, the signal and noise both increase. However, the signal increase much faster than the noise: assuming you add N datasets with equal exposure times, the signal will increases as a multiple of N, while noise increases as $\sqrt{N}$. Thus this increases the signal-to-noise ratio. Qualitatively, fainter (per pixel) parts of the objects/signal in the image will become more visible/detectable. The noise-level is known as the dataset's surface brightness limit.

You can think of the noise as muddy water that is completely covering a flat ground[18]. The signal (or astronomical objects in this analogy) will be summits/hills that start from the flat sky level (under the muddy water) and can sometimes reach outside of the muddy water. Let's assume that in your first observation the muddy water has just been stirred and you can't see anything through it. As you wait and make more observations/exposures, the mud settles down and the *depth* of the transparent water increases, making the summits visible. As the depth of clear water increases, the parts of the hills with lower heights (parts with lower surface brightness) can be seen more clearly. In this analogy, height (from the ground) is *surface brightness*[19] and the height of the muddy water is your surface brightness limit.

The outputs of NoiseChisel include the Sky standard deviation (σ) on every group of pixels (a mesh) that were calculated from the undetected pixels in each tile, see Section 4.7 [Tessellation], page 123, and Section 7.2.2.3 [NoiseChisel output], page 241. Let's take σ_m as the median σ over the successful meshes in the image (prior to interpolation or smoothing).

[18] The ground is the sky value in this analogy, see Section 7.1.3 [Sky value], page 211. Note that this analogy only holds for a flat sky value across the surface of the image or ground.

[19] Note that this muddy water analogy is not perfect, because while the water-level remains the same all over a peak, in data analysis, the Poisson noise increases with the level of data.

On different instruments, pixels have different physical sizes (for example in micro-meters, or spatial angle over the sky). Nevertheless, a pixel is our unit of data collection. In other words, while quantifying the noise, the physical or projected size of the pixels is irrelevant. We thus define the Surface brightness limit or *depth*, in units of magnitude/pixel, of a data-set, with zeropoint magnitude z, with the nth multiple of σ_m as (see Section 8.1.3 [Flux Brightness and magnitude], page 290):

$$SB_{\rm Pixel} = -2.5 \times \log_{10}(n\sigma_m) + z$$

As an example, the XDF survey covers part of the sky that the Hubble space telescope has observed the most (for 85 orbits) and is consequently very small (~ 4 arcmin^2). On the other hand, the CANDELS survey, is one of the widest multi-color surveys covering several fields (about 720 arcmin^2) but its deepest fields have only 9 orbits observation. The depth of the XDF and CANDELS-deep surveys in the near infrared WFC3/F160W filter are respectively 34.40 and 32.45 magnitudes/pixel. In a single orbit image, this same field has a depth of 31.32. Recall that a larger magnitude corresponds to less brightness.

The low-level magnitude/pixel measurement above is only useful when all the datasets you want to use belong to one instrument (telescope and camera). However, you will often find yourself using datasets from various instruments with different pixel scales (projected pixel sizes). If we know the pixel scale, we can obtain a more easily comparable surface brightness limit in units of: magnitude/arcsec^2. Let's assume that the dataset has a zeropoint value of z, and every pixel is p arcsec^2 (so A/p is the number of pixels that cover an area of A arcsec^2). If the surface brightness is desired at the nth multiple of σ_m, the following equation (in units of magnitudes per A arcsec^2) can be used:

$$SB_{\rm Projected} = -2.5 \times \log_{10}\left(n\sigma_m\sqrt{\frac{A}{p}}\right) + z$$

Note that this is just an extrapolation of the per-pixel measurement σ_m. So it should be used with extreme care: for example the dataset must have an approximately flat depth or noise properties overall. A more accurate measure for each detection over the dataset is known as the *upper-limit magnitude* which actually uses random positioning of each detection's area/footprint (see below). It doesn't extrapolate and even accounts for correlated noise patterns in relation to that detection. Therefore, the upper-limit magnitude is a much better measure of your dataset's surface brightness limit for each particular object.

MakeCatalog will calculate the input dataset's $SB_{\rm Pixel}$ and $SB_{\rm Projected}$ and write them as comments/meta-data in the output catalog(s). Just note that $SB_{\rm Projected}$ is only calculated if the input has World Coordinate System (WCS).

Completeness limit (of each detection)

As the surface brightness of the objects decreases, the ability to detect them will also decrease. An important statistic is thus the fraction of objects of

similar morphology and brightness that will be identified with our detection algorithm/parameters in the given image. This fraction is known as completeness. For brighter objects, completeness is 1: all bright objects that might exist over the image will be detected. However, as we go to objects of lower overall surface brightness, we will fail to detect some, and gradually we are not able to detect anything any more. For a given profile, the magnitude where the completeness drops below a certain level (usually above 90%) is known as the completeness limit.

Another important parameter in measuring completeness is purity: the fraction of true detections to all true detections. In effect purity is the measure of contamination by false detections: the higher the purity, the lower the contamination. Completeness and purity are anti-correlated: if we can allow a large number of false detections (that we might be able to remove by other means), we can significantly increase the completeness limit.

One traditional way to measure the completeness and purity of a given sample is by embedding mock profiles in regions of the image with no detection. However in such a study we must be really careful to choose model profiles as similar to the target of interest as possible.

Magnitude measurement error (of each detection)

Any measurement has an error and this includes the derived magnitude for an object. Note that this value is only meaningful when the object's magnitude is brighter than the upper-limit magnitude (see the next items in this list). As discussed in Section 8.1.3 [Flux Brightness and magnitude], page 290, the magnitude (M) of an object with brightness B and Zeropoint magnitude z can be written as:

$$M = -2.5\log_{10}(B) + z$$

Calculating the derivative with respect to B, we get:

$$\frac{dM}{dB} = \frac{-2.5}{B \times ln(10)}$$

From the Tailor series ($\Delta M = dM/dB \times \Delta B$), we can write:

$$\Delta M = \left|\frac{-2.5}{ln(10)}\right| \times \frac{\Delta B}{B}$$

But, $\Delta B/B$ is just the inverse of the Signal-to-noise ratio (S/N), so we can write the error in magnitude in terms of the signal-to-noise ratio:

$$\Delta M = \frac{2.5}{S/N \times ln(10)}$$

MakeCatalog uses this relation to estimate the magnitude errors. The signal-to-noise ratio is calculated in different ways for clumps and objects (see Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`)), but this single equation can be used to estimate the measured magnitude error afterwards for any type of target.

Upper limit magnitude (of each detection)

Due to the noisy nature of data, it is possible to get arbitrarily low values for a faint object's brightness (or arbitrarily high *magnitudes*). Given the scatter caused by the dataset's noise, values fainter than a certain level are meaningless: another similar depth observation will give a radically different value.

For example, while the depth of the image is 32 magnitudes/pixel, a measurement that gives a magnitude of 36 for a ~ 100 pixel object is clearly unreliable. In another similar depth image, we might measure a magnitude of 30 for it, and yet another might give 33. Furthermore, due to the noise scatter so close to the depth of the data-set, the total brightness might actually get measured as a negative value, so no magnitude can be defined (recall that a magnitude is a base-10 logarithm). This problem usually becomes relevant when the detection labels were not derived from the values being measured (for example when you are estimating colors, see Section 7.4 [MakeCatalog], page 255).

Using such unreliable measurements will directly affect our analysis, so we must not use the raw measurements. But how can we know how reliable a measurement on a given dataset is?

When we confront such unreasonably faint magnitudes, there is one thing we can deduce: that if something actually exists here (possibly buried deep under the noise), it's inherent magnitude is fainter than an *upper limit magnitude*. To find this upper limit magnitude, we place the object's footprint (segmentation map) over random parts of the image where there are no detections, so we only have pure (possibly correlated) noise, along with undetected objects. Doing this a large number of times will give us a distribution of brightness values. The standard deviation (σ) of that distribution can be used to quantify the upper limit magnitude.

Traditionally, faint/small object photometry was done using fixed circular apertures (for example with a diameter of N arc-seconds). Hence, the upper limit was like the depth discussed above: one value for the whole image. The problem with this simplified approach is that the number of pixels in the aperture directly affects the final distribution and thus magnitude. Also the image correlated noise might actually create certain patters, so the shape of the object can also affect the final result result. Fortunately, with the much more advanced hardware and software of today, we can make customized segmentation maps for each object.

When requested, MakeCatalog will randomly place each target's footprint over the dataset as described above and estimate the resulting distribution's properties (like the upper limit magnitude). The procedure is fully configurable with the options in Section 7.4.5.2 [Upper-limit settings], page 269. If one value for the whole image is required, you can either use the surface brightness limit

above or make a circular aperture and feed it into MakeCatalog to request an upper-limit magnitude for it[20].

7.4.3 Measuring elliptical parameters

The shape or morphology of a target is one of the most commonly desired parameters of a target. Here, we will review the derivation of the most basic/simple morphological parameters: the elliptical parameters for a set of labeled pixels. The elliptical parameters are: the (semi-)major axis, the (semi-)minor axis and the position angle along with the central position of the profile. The derivations below follow the SExtractor manual derivations with some added explanations for easier reading.

Let's begin with one dimension for simplicity: Assume we have a set of N values B_i (keeping the spatial distribution of brightness for example), each at position x_i. The simplest parameter we can define is the geometric center of the object (x_g) (ignoring the brightness values): $x_g = (\sum_i x_i)/N$. *Moments* are defined to incorporate both the value (brightness) and position of the data. The first moment can be written as:

$$\overline{x} = \frac{\sum_i B_i x_i}{\sum_i B_i}$$

This is essentially the weighted (by B_i) mean position. The geometric center (x_g, defined above) is a special case of this with all $B_i = 1$. The second moment is essentially the variance of the distribution:

$$\overline{x^2} \equiv \frac{\sum_i B_i(x_i - \overline{x})^2}{\sum_i B_i} = \frac{\sum_i B_i x_i^2}{\sum_i B_i} - 2\overline{x}\frac{\sum_i B_i x_i}{\sum_i B_i} + \overline{x}^2 = \frac{\sum_i B_i x_i^2}{\sum_i B_i} - \overline{x}^2$$

The last step was done from the definition of $\overline{x}$. Hence, the square root of $\overline{x^2}$ is the spatial standard deviation (along the one-dimension) of this particular brightness distribution (B_i). Crudely (or qualitatively), you can think of its square root as the distance (from $\overline{x}$) which contains a specific amount of the flux (depending on the B_i distribution). Similar to the first moment, the geometric second moment can be found by setting all $B_i = 1$. So while the first moment quantified the position of the brightness distribution, the second moment quantifies how that brightness is dispersed about the first moment. In other words, it quantifies how "sharp" the object's image is.

Before continuing to two dimensions and the derivation of the elliptical parameters, let's pause for an important implementation technicality. You can ignore this paragraph and the next two if you don't want to implement these concepts. The basic definition (first definition of $\overline{x^2}$ above) can be used without any major problem. However, using this fraction requires two runs over the data: one run to find $\overline{x}$ and another run to find $\overline{x^2}$ from $\overline{x}$, this can be

[20] If you intend to make apertures manually and not use a detection map (for example from Section 7.3 [Segment], page 243), don't forget to use the `--upmaskfile` to give NoiseChisel's output (or any a binary map, marking detected pixels, see Section 7.2.2.3 [NoiseChisel output], page 241) as a mask. Otherwise, the footprints may randomly fall over detections, giving highly skewed distributions, with wrong upper-limit distributions. See The description of `--upmaskfile` in Section 7.4.5.2 [Upper-limit settings], page 269, for more.

slow. The advantage of the last fraction above, is that we can estimate both the first and second moments in one run (since the $-\overline{x}^2$ term can easily be added later).

The logarithmic nature of floating point number digitization creates a complication however: suppose the object is located between pixels 10000 and 10020. Hence the target's pixels are only distributed over 20 pixels (with a standard deviation < 20), while the mean has a value of ~ 10000. The $\sum_i B_i^2 x_i^2$ will go to very very large values while the individual pixel differences will be orders of magnitude smaller. This will lower the accuracy of our calculation due to the limited accuracy of floating point operations. The variance only depends on the distance of each point from the mean, so we can shift all position by a constant/arbitrary K which is much closer to the mean: $\overline{x-K} = \overline{x} - K$. Hence we can calculate the second order moment using:

$$\overline{x^2} = \frac{\sum_i B_i (x_i - K)^2}{\sum_i B_i} - (\overline{x} - K)^2$$

The closer K is to $\overline{x}$, the better (the sums of squares will involve smaller numbers), as long as K is within the object limits (in the example above: $10000 \leq K \leq 10020$), the floating point error induced in our calculation will be negligible. For the most simplest implementation, MakeCatalog takes K to be the smallest position of the object in each dimension. Since K is arbitrary and an implementation/technical detail, we will ignore it for the remainder of this discussion.

In two dimensions, the mean and variances can be written as:

$$\overline{x} = \frac{\sum_i B_i x_i}{B_i}, \quad \overline{x^2} = \frac{\sum_i B_i x_i^2}{\sum_i B_i} - \overline{x}^2$$

$$\overline{y} = \frac{\sum_i B_i y_i}{B_i}, \quad \overline{y^2} = \frac{\sum_i B_i y_i^2}{\sum_i B_i} - \overline{y}^2$$

$$\overline{xy} = \frac{\sum_i B_i x_i y_i}{\sum_i B_i} - \overline{x} \times \overline{y}$$

If an elliptical profile's major axis exactly lies along the x axis, then $\overline{x^2}$ will be directly proportional with the profile's major axis, $\overline{y^2}$ with its minor axis and $\overline{xy} = 0$. However, in reality we are not that lucky and (assuming galaxies can be parameterized as an ellipse) the major axis of galaxies can be in any direction on the image (in fact this is one of the core principles behind weak-lensing by shear estimation). So the purpose of the remainder of this section is to define a strategy to measure the position angle and axis ratio of some randomly positioned ellipses in an image, using the raw second moments that we have calculated above in our image coordinates.

Let's assume we have rotated the galaxy by θ, the new second order moments are:

$$\overline{x_\theta^2} = \overline{x^2}\cos^2\theta + \overline{y^2}\sin^2\theta - 2\overline{xy}\cos\theta\sin\theta$$

$$\overline{y_\theta^2} = \overline{x^2}\sin^2\theta + \overline{y^2}\cos^2\theta + 2\overline{xy}\cos\theta\sin\theta$$

$$\overline{xy_\theta} = \overline{x^2}\cos\theta\sin\theta - \overline{y^2}\cos\theta\sin\theta + \overline{xy}(\cos^2\theta - \sin^2\theta)$$

The best θ (θ_0, where major axis lies along the x_θ axis) can be found by:

$$\left.\frac{\partial\overline{x_\theta^2}}{\partial\theta}\right|_{\theta_0} = 0$$

Taking the derivative, we get:

$$2\cos\theta_0\sin\theta_0(\overline{y^2} - \overline{x^2}) + 2(\cos^2\theta_0 - \sin^2\theta_0)\overline{xy} = 0$$

When $\overline{x^2} \neq \overline{y^2}$, we can write:

$$\tan 2\theta_0 = 2\frac{\overline{xy}}{\overline{x^2} - \overline{y^2}}.$$

MakeCatalog uses the standard C math library's `atan2` function to estimate θ_0, which we define as the position angle of the ellipse. To recall, this is the angle of the major axis of the ellipse with the x axis. By definition, when the elliptical profile is rotated by θ_0, then $\overline{xy_{\theta_0}} = 0$, $\overline{x_{\theta_0}^2}$ will be the extent of the maximum variance and $\overline{y_{\theta_0}^2}$ the extent of the minimum variance (which are perpendicular for an ellipse). Replacing θ_0 in the equations above for $\overline{x_\theta}$ and $\overline{y_\theta}$, we can get the semi-major (A) and semi-minor (B) lengths:

$$A^2 \equiv \overline{x_{\theta_0}^2} = \frac{\overline{x^2} + \overline{y^2}}{2} + \sqrt{\left(\frac{\overline{x^2} - \overline{y^2}}{2}\right)^2 + \overline{xy}^2}$$

$$B^2 \equiv \overline{y_{\theta_0}^2} = \frac{\overline{x^2} + \overline{y^2}}{2} - \sqrt{\left(\frac{\overline{x^2} - \overline{y^2}}{2}\right)^2 + \overline{xy}^2}$$

As a summary, it is important to remember that the units of A and B are in pixels (the standard deviation of a positional distribution) and that they represent the spatial light distribution of the object in both image dimensions (rotated by θ_0). When the object cannot be represented as an ellipse, this interpretation breaks down: $\overline{xy_{\theta_0}} \neq 0$ and $\overline{y_{\theta_0}^2}$ will not be the direction of minimum variance.

7.4.4 Adding new columns to MakeCatalog

MakeCatalog is designed to allow easy addition of different measurements over a labeled image (see Akhlaghi [2016] (`https://arxiv.org/abs/1611.06387v1`)). A check-list style description of necessary steps to do that is described in this section. The common development characteristics of MakeCatalog and other Gnuastro programs is explained in Chapter 12 [Developing], page 445. We strongly encourage you to have a look at that chapter to greatly simplify your navigation in the code. After adding and testing your column, you are most welcome (and encouraged) to share it with us so we can add to the next release of Gnuastro for everyone else to also benefit from your efforts.

MakeCatalog will first pass over each label's pixels two times and do necessary raw/internal calculations. Once the passes are done, it will use the raw information for filling the final catalog's columns. In the first pass it will gather mainly object information and in the second run, it will mainly focus on the clumps, or any other measurement that needs an output from the first pass. These two passes are designed to be raw summations: no extra processing. This will allow parallel processing and simplicity/clarity. So if your new calculation, needs new raw information from the pixels, then you will need to also modify the respective `mkcatalog_first_pass` and `mkcatalog_second_pass` functions (both in `bin/mkcatalog/mkcatalog.c`) and define new raw table columns in `main.h` (hopefully the comments in the code are clear enough).

In all these different places, the final columns are sorted in the same order (same order as Section 7.4.5 [Invoking MakeCatalog], page 266). This allows a particular column/option to be easily found in all steps. Therefore in adding your new option, be sure to keep it in the same relative place in the list in all the separate places (it doesn't necessarily have to be in the end), and near conceptually similar options.

`main.h` The `objectcols` and `clumpcols` enumerated variables (`enum`) define the raw/internal calculation columns. If your new column requires new raw calculations, add a row to the respective list. If your calculation requires any other settings parameters, you should add a variable to the `mkcatalogparams` structure.

`ui.c` If the new column needs raw calculations (an entry was added in `objectcols` and `clumpcols`), specify which inputs it needs in `ui_necessary_inputs`, similar to the other options. Afterwards, if your column includes any particular settings (you needed to add a variable to the `mkcatalogparams` structure in `main.h`), you should do the sanity checks and preparations for it here.

`ui.h` The `option_keys_enum` associates a unique value for each option to MakeCatalog. The options that have a short option version, the single character short comment is used for the value. Those that don't have a short option version, get a large integer automatically. You should add a variable here to identify your desired column.

`args.h` This file specifies all the parameters for the GNU C library, Argp structure that is in charge of reading the user's options. To define your new column, just copy an existing set of parameters and change the first, second and 5th values (the only ones that differ between all the columns), you should use the macro you defined in `ui.h` here.

`columns.c`
This file contains the main definition and high-level calculation of your new column through the `columns_define_alloc` and `columns_fill` functions. In the first, you specify the basic information about the column: its name, units, comments, type (see Section 4.5 [Numeric data types], page 115) and how it should be printed if the output is a text file. You should also specify the raw/internal columns that are necessary for this column here as the many existing examples show. Through the types for objects and rows, you can specify if this column is only for clumps, objects or both.

The second main function (`columns_fill`) writes the final value into the appropriate column for each object and clump. As you can see in the many existing examples, you can define your processing on the raw/internal calculations here and save them in the output.

`mkcatalog.c`
: As described before, this file contains the two main MakeCatalog work-horses: `mkcatalog_first_pass` and `mkcatalog_second_pass`, their names are descriptive enough and their internals are also clear and heavily commented.

`doc/gnuastro.texi`
: Update this manual and add a description for the new column.

7.4.5 Invoking MakeCatalog

MakeCatalog will do measurements and produce a catalog from a labeled dataset and optional values dataset(s). The executable name is `astmkcatalog` with the following general template

```
$ astmkcatalog [OPTION ...] InputImage.fits
```

One line examples:

```
## Create catalog with RA, Dec, Magnitude and Magnitude error,
## from Segment's output:
$ astmkcatalog --ra --dec --magnitude --magnitudeerr seg-out.fits

## Same catalog as above (using short options):
$ asmkcatalog -rdmG seg-out.fits

## Write the catalog to a text table:
$ astmkcatalog -mpQ seg-out.fits --output=cat.txt

## Output columns specified in 'columns.conf':
$ astmkcatalog --config=columns.conf seg-out.fits

## Use object and clump labels from a K-band image, but pixel values
## from an i-band image.
$ astmkcatalog K_segmented.fits --hdu=DETECTIONS --clumpscat     \
               --clumpsfile=K_segmented.fits --clumpshdu=CLUMPS  \
               --valuesfile=i_band.fits
```

If MakeCatalog is to do processing (not printing help or option values), an input labeled image should be provided. The options described in this section are those that are particular to MakeProfiles. For operations that MakeProfiles shares with other programs (mainly involving input/output or general processing steps), see Section 4.1.2 [Common options], page 95. Also see Chapter 4 [Common program behavior], page 91, for some general characteristics of all Gnuastro programs including MakeCatalog.

The various measurements/columns of MakeCatalog are requested as options, either on the command-line or in configuration files, see Section 4.2 [Configuration files], page 106. The full list of available columns is available in Section 7.4.5.3 [MakeCatalog measurements], page 272. Depending on the requested columns, MakeCatalog needs more than one input

dataset, for more details, please see Section 7.4.5.1 [MakeCatalog inputs and basic settings], page 267. The upper-limit measurements in particular need several configuration options which are thoroughly discussed in Section 7.4.5.2 [Upper-limit settings], page 269. Finally, in Section 7.4.5.4 [MakeCatalog output], page 278, the output file(s) created by MakeCatalog are discussed.

7.4.5.1 MakeCatalog inputs and basic settings

MakeCatalog works by using a localized/labeled dataset (see Section 7.4 [MakeCatalog], page 255). This dataset maps/labels pixels to a specific target (row number in the final catalog) and is thus the only necessary input dataset to produce a minimal catalog in any situation. Because it only has labels/counters, it must have an integer type (see Section 4.5 [Numeric data types], page 115), see below if your labels are in a floating point container. When the requested measurements only need this dataset (for example `--geox`, `--geoy`, or `--geoarea`), MakeCatalog won't read any more datasets.

Low-level measurements that only use the labeled image are rarely sufficient for any high-level science case. Therefore necessary input datasets depend on the requested columns in each run. For example, let's assume you want the brightness/magnitude and signal-to-noise ratio of your labeled regions. For these columns, you will also need to provide an extra dataset containing values for every pixel of the labeled input (to measure brightness) and another for the Sky standard deviation (to measure error). All such auxiliary input files have to have the same size (number of pixels in each dimension) as the input labeled image. Their numeric data type is irrelevant (they will be converted to 32-bit floating point internally). For the full list of available measurements, see Section 7.4.5.3 [MakeCatalog measurements], page 272.

The "values" dataset is used for measurements like brightness/magnitude, or flux-weighted positions. If it is a real image, by default it is assumed to be already Sky-subtracted prior to running MakeCatalog. If it isn't, you use the `--subtractsky` option to, so MakeCatalog reads and subtracts the Sky dataset before any processing. To obtain the Sky value, you can use the `--sky` option of Section 7.1 [Statistics], page 208, but the best recommended method is Section 7.2 [NoiseChisel], page 225, see Section 7.1.3 [Sky value], page 211.

MakeCatalog can also do measurements on sub-structures of detections. In other words, it can produce two catalogs. Following the nomenclature of Segment (see Section 7.3 [Segment], page 243), the main labeled input dataset is known as "object" labels and the (optional) sub-structure input dataset is known as "clumps". If MakeCatalog is run with the `--clumpscat` option, it will also need a labeled image containing clumps, similar to what Segment produces (see Section 7.3.2.3 [Segment output], page 253). Since clumps are defined within detected regions (they exist over signal, not noise), MakeCatalog uses their boundaries to subtract the level of signal under them.

There are separate options to explicitly request a file name and HDU/extension for each of the required input datasets as fully described below (with the `--*file` format). When each dataset is in a separate file, these options are necessary. However, one great advantage of the FITS file format, that is heavily used in astronomy, is that it allows the storage of multiple datasets in one file. So in most situations (for example if you are using the outputs of Section 7.2 [NoiseChisel], page 225, or Section 7.3 [Segment], page 243), all the necessary input datasets can be in one file.

When none of the `--*file` options are given, MakeCatalog will assume the necessary input datasets are in the file given as its argument (without any option). When the Sky or Sky standard deviation datasets are necessary and the only `--*file` option called is `--valuesfile`, MakeCatalog will search for these datasets (with the default/given HDUs) in the file given to `--valuesfile` (before looking into the the main argument file).

It may happen that your labeled objects image was created with a program that only outputs floating point files. However, you know it only has integer valued pixels that are stored in a floating point container. In such cases, you can use Gnuastro's Arithmetic program (see Section 6.2 [Arithmetic], page 161) to change the numerical data type of the image (`float.fits`) to an integer type image (`int.fits`) with a command like below:

```
$ astarithmetic float.fits int32 --output=int.fits
```

To summarize: if the input file to MakeCatalog is the default/full output of Segment (see Section 7.3.2.3 [Segment output], page 253) you don't have to worry about any of the `--*file` options below. You can just give Segment's output file to MakeCatalog as described in Section 7.4.5 [Invoking MakeCatalog], page 266. To feed NoiseChisel's output into MakeCatalog, just change the labeled dataset's header (with `--hdu=DETECTIONS`). The full list of input dataset options and general setting options are described below.

`-l STR`
`--clumpsfile=STR`
: The file containing the labeled clumps dataset when `--clumpscat` is called (see Section 7.4.5.4 [MakeCatalog output], page 278). When `--clumpscat` is called, but this option isn't, MakeCatalog will look into the main input file (given as an argument) for the required extension/HDU (value to `--clumpshdu`).

`--clumpshdu=STR`
: The HDU/extension of the clump labels dataset. Only pixels with values above zero will be considered. The clump labels dataset has to be an integer data type (see Section 4.5 [Numeric data types], page 115) and only pixels with a value larger than zero will be used. See Section 7.3.2.3 [Segment output], page 253, for a description of the expected format.

`-v STR`
`--valuesfile=STR`
: The file name of the (sky-subtracted) values dataset. When any of the columns need values to associate with the input labels (for example to measure the brightness/magnitude of a galaxy), MakeCatalog will look into a "values" for the respective pixel values. In most common processing, this is the actual astronomical image that the labels were defined, or detected, over. The HDU/extension of this dataset in the given file can be specified with `--valueshdu`. If this option is not called, MakeCatalog will look for the given extension in the main input file.

`--valueshdu=STR/INT`
: The name or number (counting from zero) of the extension containing the "values" dataset, see the descriptions above and those in `--valuesfile` for more.

`-s STR/FLT`
`--insky=STR/FLT`
: Sky value as a single number, or the file name containing a dataset (different values per pixel or tile). The Sky dataset is only necessary when `--subtractsky` is called or when a column directly related to the Sky value is requested (currently `--sky`).

  When the Sky dataset is necessary and this option is not called, MakeCatalog will assume it is a dataset first look into the `--valuesfile` (if it is given) and then the main input file (given as an argument). By default the values dataset is assumed to be already Sky subtracted, so this dataset is not necessary for many of the columns.

  This dataset may be tessellation, with one element per tile (see `--oneelempertile` of Section 4.1.2.2 [Processing options], page 98).

`--skyhdu=STR`
: HDU/extension of the Sky dataset, see `--skyfile`.

`--subtractsky`
: Subtract the sky value or dataset from the values file prior to any processing.

`-t STR/FLT`
`--instd=STR/FLT`
: Sky standard deviation value as a single number, or the file name containing a dataset (different values per pixel or tile). With the `--variance` option you can tell MakeCatalog to interpret this value/dataset as a variance image, not standard deviation.

  Important note: This must only be the SKY standard deviation or variance (not including the signal's contribution to the error). In other words, the final standard deviation of a pixel depends on how much signal there is in it. MakeCatalog will find the amount of signal within each pixel (while subtracting the Sky, if `--subtractsky` is called) and account for the extra error due to it's value (signal). Therefore if the input standard deviation (or variance) image also contains the contribution of signal to the error, then the final error measurements will be over-estimated.

`--stdhdu=STR`
: The HDU of the Sky value standard deviation image.

`--variance`
: The dataset given to `--stdfile` (and `--stdhdu` has the Sky variance of every pixel, not the Sky standard deviation.

`-z FLT`
`--zeropoint=FLT`
: The zero point magnitude for the input image, see Section 8.1.3 [Flux Brightness and magnitude], page 290.

7.4.5.2 Upper-limit settings

The upper-limit magnitude was discussed in Section 7.4.2 [Quantifying measurement limits], page 258. Unlike other measured values/columns in MakeCatalog, the upper limit

magnitude needs several extra parameters which are discussed here. All the options specific to the upper-limit measurements start with `up` for "upper-limit". The only exception is `--envseed` that is also present in other programs and is general for any job requiring random number generation in Gnuastro (see Section 8.2.1.4 [Generating random numbers], page 304).

One very important consideration in Gnuastro is reproducibility. Therefore, the values to all of these parameters along with others (like the random number generator type and seed) are also reported in the comments of the final catalog when the upper limit magnitude column is desired. The random seed that is used to define the random positions for each object or clump is unique and set based on the (optionally) given seed, the total number of objects and clumps and also the labels of the clumps and objects. So with identical inputs, an identical upper-limit magnitude will be found. However, even if the seed is identical, when the ordering of the object/clump labels differs between different runs, the result of upper-limit measurements will not be identical.

MakeCatalog will randomly place the object/clump footprint over the dataset. When the randomly placed footprint doesn't fall on any object or masked region (see `--upmaskfile`) it will be used in the final distribution. Otherwise that particular random position will be ignored and another random position will be generated. Finally, when the distribution has the desired number of successfully measured random samples (`--upnum`) the distribution's properties will be measured and placed in the catalog.

When the profile is very large or the image is significantly covered by detections, it might not be possible to find the desired number of samplings in a reasonable time. MakeProfiles will continue searching until it is unable to find a successful position (since the last successful measurement[21]), for a large multiple of `--upnum` (currently[22] this is 10). If `--upnum` successful samples cannot be found until this limit is reached, MakeCatalog will set the upper-limit magnitude for that object to NaN (blank).

MakeCatalog will also print a warning if the range of positions available for the labeled region is smaller than double the size of the region. In such cases, the limited range of random positions can artificially decrease the standard deviation of the final distribution. If your dataset can allow it (it is large enough), it is recommended to use a larger range if you see such warnings.

`--upmaskfile=STR`

File name of mask image to use for upper-limit calculation. In some cases (especially when doing matched photometry), the object labels specified in the main input and mask image might not be adequate. In other words they do not necessarily have to cover *all* detected objects: the user might have selected only a few of the objects in their labeled image. This option can be used to ignore regions in the image in these situations when estimating the upper-limit magnitude. All the non-zero pixels of the image specified by this option (in the `--upmaskhdu` extension) will be ignored in the upper-limit magnitude measurements.

[21] The counting of failed positions restarts on every successful measurement.

[22] In Gnuastro's source, this constant number is defined as the `MKCATALOG_UPPERLIMIT_MAXFAILS_MULTIP` macro in `bin/mkcatalog/main.h`, see Section 3.2 [Downloading the source], page 71.

For example, when you are using labels from another image, you can give NoiseChisel's objects image output for this image as the value to this option. In this way, you can be sure that regions with data do not harm your distribution. See Section 7.4.2 [Quantifying measurement limits], page 258, for more on the upper limit magnitude.

`--upmaskhdu=STR`
The extension in the file specified by `--upmask`.

`--upnum=INT`
The number of random samples to take for all the objects. A larger value to this option will give a more accurate result (asymptotically), but it will also slow down the process. When a randomly positioned sample overlaps with a detected/masked pixel it is not counted and another random position is found until the object completely lies over an undetected region. So you can be sure that for each object, this many samples over undetected objects are made. See the upper limit magnitude discussion in Section 7.4.2 [Quantifying measurement limits], page 258, for more.

`--uprange=INT,INT`
The range/width of the region (in pixels) to do random sampling along each dimension of the input image around each object's position. This is not a mandatory option and if not given (or given a value of zero in a dimension), the full possible range of the dataset along that dimension will be used. This is useful when the noise properties of the dataset vary gradually. In such cases, using the full range of the input dataset is going to bias the result. However, note that decreasing the the range of available positions too much will also artificially decrease the standard deviation of the final distribution (and thus bias the upper-limit measurement).

`--envseed`
Read the random number generator type and seed value from the environment (see Section 8.2.1.4 [Generating random numbers], page 304). Random numbers are used in calculating the random positions of different samples of each object.

`--upsigmaclip=FLT,FLT`
The raw distribution of random values will not be used to find the upper-limit magnitude, it will first be σ-clipped (see Section 7.1.2 [Sigma clipping], page 209) to avoid outliers in the distribution (mainly the faint undetected wings of bright/large objects in the image). This option takes two values: the first is the multiple of σ, and the second is the termination criteria. If the latter is larger than 1, it is read as an integer number and will be the number of times to clip. If it is smaller than 1, it is interpreted as the tolerance level to stop clipping. See Section 7.1.2 [Sigma clipping], page 209, for a complete explanation.

`--upnsigma=FLT`
The multiple of the final (σ-clipped) standard deviation (or σ) used to measure the upper-limit brightness or magnitude.

`--checkuplim=INT[,INT]`

Print a table of positions and measured values for all the full random distribution used for one particular object or clump. If only one integer is given to this option, it is interpreted to be an object's label. If two values are given, the first is the object label and the second is the ID of requested clump within it.

The output is a table with three columns (its type is determined with the `--tableformat` option, see Section 4.1.2.1 [Input/Output options], page 95). The first two columns are the position of the first pixel in each random sampling of this particular object/clump. The the third column is the measured flux over that region. If the region overlapped with a detection or masked pixel, then its measured value will be a NaN (not-a-number). The total number of rows is thus unknown, but you can be sure that the number of rows with non-NaN measurements is the number given to the `--upnum` option.

7.4.5.3 MakeCatalog measurements

The final group of options particular to MakeCatalog are those that specify which measurements/columns should be written into the final output table. The current measurements in MakeCatalog are those which only produce one final value for each label (for example its total brightness: a single number). All the different label's measurements can be written as one column in a final table/catalog that contains other columns for other similar single-number measurements.

Command-line options are used to identify which measurements you want in the final catalog(s) and in what order. If any of the options below is called on the command line or in any of the configuration files, it will be included as a column in the output catalog. The order of the columns is in the same order as the options were seen by MakeCatalog (see Section 4.2.2 [Configuration file precedence], page 107). Some of the columns apply to both "objects" and "clumps" and some are particular to only one of them (for the definition of "objects" and "clumps", see Section 7.3 [Segment], page 243). Columns/options that are unique to one catalog (only objects, or only clumps), are explicitly marked with [Objects] or [Clumps] to specify the catalog they will be placed in.

`--i`
`--ids` This is a unique option which can add multiple columns to the final catalog(s). Calling this option will put the object IDs (`--objid`) in the objects catalog and host-object-ID (`--hostobjid`) and ID-in-host-object (`--idinhostobj`) into the clumps catalog. Hence if only object catalogs are required, it has the same effect as `--objid`.

`--objid` [Objects] ID of this object.

`-j`
`--hostobjid`
[Clumps] The ID of the object which hosts this clump.

`--idinhostobj`
[Clumps] The ID of this clump in its host object.

`-x`
`--x` The flux weighted center of all objects and clumps along the first FITS axis (horizontal when viewed in SAO ds9), see $\overline{x}$ in Section 7.4.3 [Measuring elliptical

parameters], page 262. The weight has to have a positive value (pixel value larger than the Sky value) to be meaningful! Specially when doing matched photometry, this might not happen: no pixel value might be above the Sky value. For such detections, the geometric center will be reported in this column (see `--geox`). You can use `--weightarea` to see which was used.

`-y`
`--y` The flux weighted center of all objects and clumps along the second FITS axis (vertical when viewed in SAO ds9). See `--x`.

`--geox` The geometric center of all objects and clumps along the first FITS axis axis. The geometric center is the average pixel positions irrespective of their pixel values.

`--geoy` The geometric center of all objects and clumps along the second FITS axis axis, see `--geox`.

`--minx` The minimum position of all objects and clumps along the first FITS axis.

`--maxx` The maximum position of all objects and clumps along the first FITS axis.

`--miny` The minimum position of all objects and clumps along the second FITS axis.

`--maxy` The maximum position of all objects and clumps along the second FITS axis.

`--clumpsx`
[Objects] The flux weighted center of all the clumps in this object along the first FITS axis. See `--x`.

`--clumpsy`
[Objects] The flux weighted center of all the clumps in this object along the second FITS axis. See `--x`.

`--clumpsgeox`
[Objects] The geometric center of all the clumps in this object along the first FITS axis. See `--geox`.

`--clumpsgeoy`
[Objects] The geometric center of all the clumps in this object along the second FITS axis. See `--geox`.

`-r`
`--ra` Flux weighted right ascension of all objects or clumps, see `--x`. This is just an alias for one of the lower-level `--w1` or `--w2` options. Using the FITS WCS keywords (`CTYPE`), MakeCatalog will determine which axis corresponds to the right ascension. If no `CTYPE` keywords start with `RA`, an error will be printed when requesting this column and MakeCatalog will abort.

`-d`
`--dec` Flux weighted declination of all objects or clumps, see `--x`. This is just an alias for one of the lower-level `--w1` or `--w2` options. Using the FITS WCS keywords (`CTYPE`), MakeCatalog will determine which axis corresponds to the declination. If no `CTYPE` keywords start with `DEC`, an error will be printed when requesting this column and MakeCatalog will abort.

`--w1` Flux weighted first WCS axis of all objects or clumps, see `--x`. The first WCS axis is commonly used as right ascension in images.

`--w2` Flux weighted second WCS axis of all objects or clumps, see `--x`. The second WCS axis is commonly used as declination in images.

`--geow1` Geometric center in first WCS axis of all objects or clumps, see `--geox`. The first WCS axis is commonly used as right ascension in images.

`--geow2` Geometric center in second WCS axis of all objects or clumps, see `--geox`. The second WCS axis is commonly used as declination in images.

`--clumpsw1`
[Objects] Flux weighted center in first WCS axis of all clumps in this object, see `--x`. The first WCS axis is commonly used as right ascension in images.

`--clumpsw2`
[Objects] Flux weighted declination of all clumps in this object, see `--x`. The second WCS axis is commonly used as declination in images.

`--clumpsgeow1`
[Objects] Geometric center right ascension of all clumps in this object, see `--geox`. The first WCS axis is commonly used as right ascension in images.

`--clumpsgeow2`
[Objects] Geometric center declination of all clumps in this object, see `--geox`. The second WCS axis is commonly used as declination in images.

`-b`
`--brightness`
The brightness (sum of all pixel values), see Section 8.1.3 [Flux Brightness and magnitude], page 290. For clumps, the ambient brightness (flux of river pixels around the clump multiplied by the area of the clump) is removed, see `--riverflux`. So the sum of all the clumps brightness in the clump catalog will be smaller than the total clump brightness in the `--clumpbrightness` column of the objects catalog.

If no usable pixels (blank or below the threshold) are present over the clump or object, the stored value will be NaN (note that zero is meaningful).

`--brightnesserr`
The (1σ) error in measuring the brightness of objects or clumps.

`--clumpbrightness`
[Objects] The total brightness of the clumps within an object. This is simply the sum of the pixels associated with clumps in the object. If no usable pixels (blank or below the threshold) are present over the clump or object, the stored value will be NaN, because zero (note that zero is meaningful).

`--brightnessnoriver`
[Clumps] The Sky (not river) subtracted clump brightness. By definition, for the clumps, the average brightness of the rivers surrounding it are subtracted from it for a first order accounting for contamination by neighbors. In cases where you will be calculating the flux brightness difference later (one example

below) the contamination will be (mostly) removed at that stage, which is why this column was added.

One example might be this: you want to know the change in the clump flux as a function of threshold (see `--threshold`). So you will make two catalogs (each having this column but with different thresholds) and then subtract the lower threshold catalog (higher brightness) from the higher threshold catalog (lower brightness). The effect is most visible when the rivers have a high average signal-to-noise ratio. The removed contribution from the pixels below the threshold will be less than the river pixels. Therefore the river-subtracted brightness (`--brightness`) for the thresholded catalog for such clumps will be larger than the brightness with no threshold!

If no usable pixels (blank or below the possibly given threshold) are present over the clump or object, the stored value will be NaN (note that zero is meaningful).

`--mean` The mean sky subtracted value of pixels within the object or clump. For clumps, the average river flux is subtracted from the sky subtracted mean.

`--median` The median sky subtracted value of pixels within the object or clump. For clumps, the average river flux is subtracted from the sky subtracted median.

`-m`
`--magnitude`
The magnitude of clumps or objects, see `--brightness`.

`-e`
`--magnitudeerr`
The magnitude error of clumps or objects. The magnitude error is calculated from the signal-to-noise ratio (see `--sn` and Section 7.4.2 [Quantifying measurement limits], page 258). Note that until now this error assumes uncorrelated pixel values and also does not include the error in estimating the aperture (or error in generating the labeled image).

For now these factors have to be found by other means. Task 14124 (`https://savannah.gnu.org/task/index.php?14124`) has been defined for work on adding these sources of error too.

`--clumpsmagnitude`
[Objects] The magnitude of all clumps in this object, see `--clumpbrightness`.

`--upperlimit`
The upper limit value (in units of the input image) for this object or clump. See Section 7.4.2 [Quantifying measurement limits], page 258, and Section 7.4.5.2 [Upper-limit settings], page 269, for a complete explanation. This is very important for the fainter and smaller objects in the image where the measured magnitudes are not reliable.

`--upperlimitmag`
The upper limit magnitude for this object or clump. See Section 7.4.2 [Quantifying measurement limits], page 258, and Section 7.4.5.2 [Upper-limit settings], page 269, for a complete explanation. This is very important for the fainter and smaller objects in the image where the measured magnitudes are not reliable.

`--upperlimitonesigma`
: The 1σ upper limit value (in units of the input image) for this object or clump. See Section 7.4.2 [Quantifying measurement limits], page 258, and Section 7.4.5.2 [Upper-limit settings], page 269, for a complete explanation. When `--upnsigma=1`, this column's values will be the same as `--upperlimit`.

`--upperlimitsigma`
: The position of the total brightness measured within the distribution of randomly placed upperlimit measurements in units of the distribution's σ or standard deviation. See Section 7.4.2 [Quantifying measurement limits], page 258, and Section 7.4.5.2 [Upper-limit settings], page 269, for a complete explanation.

`--upperlimitquantile`
: The position of the total brightness measured within the distribution of randomly placed upperlimit measurements as a quantile (value between 0 or 1). See Section 7.4.2 [Quantifying measurement limits], page 258, and Section 7.4.5.2 [Upper-limit settings], page 269, for a complete explanation. If the object is brighter than the brightest randomly placed profile, a value of `inf` is returned. If it is less than the minimum, a value of `-inf` is reported.

`--upperlimitskew`
: This column contains the non-parametric skew of the sigma-clipped random distribution that was used to estimate the upper-limit magnitude. Taking μ as the mean, ν as the median and σ as the standard deviation, the traditional definition of skewness is defined as: $(\mu - \nu)/\sigma$.

  This can be a good measure to see how much you can trust the random measurements, or in other words, how accurately the regions with signal have been masked/detected. If the skewness is strong (and to the positive), then you can tell that you have a lot of undetected signal in the dataset, and therefore that the upper-limit measurement (and other measurements) are not reliable.

`--riverave`
: [Clumps] The average brightness of the river pixels around this clump. River pixels were defined in Akhlaghi and Ichikawa 2015. In short they are the pixels immediately outside of the clumps. This value is used internally to find the brightness (or magnitude) and signal to noise ratio of the clumps. It can generally also be used as a scale to gauge the base (ambient) flux surrounding the clump. In case there was no river pixels, then this column will have the value of the Sky under the clump. So note that this value is *not* sky subtracted.

`--rivernum`
: [Clumps] The number of river pixels around this clump, see `--riverflux`.

`-n`
`--sn`
: The Signal to noise ratio (S/N) of all clumps or objects. See Akhlaghi and Ichikawa (2015) for the exact equations used.

`--sky`
: The sky flux (per pixel) value under this object or clump. This is actually the mean value of all the pixels in the sky image that lie on the same position as the object or clump.

`--std` The sky value standard deviation (per pixel) for this clump or object. This is the square root of the mean variance under the object, or the root mean square.

`-C`
`--numclumps`
[Objects] The number of clumps in this object.

`-a`
`--area` The raw area (number of pixels) in any clump or object independent of what pixel it lies over (if it is NaN/blank or unused for example).

`--clumpsarea`
[Objects] The total area of all the clumps in this object.

`--weightarea`
The area (number of pixels) used in the flux weighted position calculations.

`--geoarea`
The area of all the pixels labeled with an object or clump. Note that unlike `--area`, pixel values are completely ignored in this column. For example, if a pixel value is blank, it won't be counted in `--area`, but will be counted here.

`-A`
`--semimajor`
The pixel-value weighted semi-major axis of the profile (assuming it is an ellipse) in units of pixels. See Section 7.4.3 [Measuring elliptical parameters], page 262.

`-B`
`--semiminor`
The pixel-value weighted semi-minor axis of the profile (assuming it is an ellipse) in units of pixels. See Section 7.4.3 [Measuring elliptical parameters], page 262.

`--axisratio`
The pixel-value weighted axis ratio (semi-minor/semi-major) of the object or clump.

`-p`
`--positionangle`
The pixel-value weighted angle of the semi-major axis with the first FITS axis in degrees. See Section 7.4.3 [Measuring elliptical parameters], page 262.

`--geosemimajor`
The geometric (ignoring pixel values) semi-major axis of the profile, assuming it is an ellipse.

`--geosemiminor`
The geometric (ignoring pixel values) semi-minor axis of the profile, assuming it is an ellipse.

`--geoaxisratio`
The geometric (ignoring pixel values) axis ratio of the profile, assuming it is an ellipse.

`--geopositionangle`
: The geometric (ignoring pixel values) angle of the semi-major axis with the first FITS axis in degrees.

7.4.5.4 MakeCatalog output

After it has completed all the requested measurements (see Section 7.4.5.3 [MakeCatalog measurements], page 272), MakeCatalog will store its measurements in table(s). If an output filename is given (see `--output` in Section 4.1.2.1 [Input/Output options], page 95), the format of the table will be deduced from the name. When it isn't given, the input name will be appended with a `_cat` suffix (see Section 4.8 [Automatic output], page 124) and its format will be determined from the `--tableformat` option, which is also discussed in Section 4.1.2.1 [Input/Output options], page 95. `--tableformat` is also necessary when the requested output name is a FITS table (recall that FITS can accept ASCII and binary tables, see Section 5.3 [Table], page 147).

By default only a single catalog/table will be created for "objects", however, if `--clumpscat` is called, a secondary catalog/table will also be created. For more on "objects" and "clumps", see Section 7.3 [Segment], page 243. In short, if you only have one set of labeled images, you don't have to worry about clumps (they are deactivated by default).

The full list of MakeCatalog's output options are elaborated below.

`-C`
`--clumpscat`
: Do measurements on clumps and produce a second catalog (only devoted to clumps). When this option is given, MakeCatalog will also look for a secondary labeled dataset (identifying substructure) and produce a catalog from that. For more on the definition on "clumps", see Section 7.3 [Segment], page 243.

  When the output is a FITS file, the objects and clumps catalogs/tables will be stored as multiple extensions of one FITS file. You can use Section 5.3 [Table], page 147, to inspect the column meta-data and contents in this case. However, in plain text format (see Section 4.6.2 [Gnuastro text table format], page 119), it is only possible to keep one table per file. Therefore, if the output is a text file, two output files will be created, ending in `_o.txt` (for objects) and `_c.txt` (for clumps).

`--noclumpsort`
: Don't sort the clumps catalog based on object ID (only relevant with `--clumpscat`). This option will benefit the performance[23] of MakeCatalog when it is run on multiple threads *and* the position of the rows in the clumps catalog is irrelevant (for example you just want the number-counts).

  MakeCatalog does all its measurements on each *object* independently and in parallel. As a result, while it is writing the measurements on each object's clumps, it doesn't know how many clumps there were in previous objects. Each thread will just fetch the first available row and write the information of clumps

[23] The performance boost due to `--noclumpsort` can only be felt when there are a huge number of objects. Therefore, by default the output is sorted to avoid miss-understandings or bugs in the user's scripts when the user forgets to sort the outputs.

(in order) starting from that row. After all the measurements are done, by default (when this option isn't called), MakeCatalog will reorder/permute the clumps catalog to have both the object and clump ID in an ascending order.

If you would like to order the catalog later (when its a plain text file), you can run the following command to sort the rows by object ID (and clump ID within each object), assuming they are respectively the first and second columns:

```
$ awk '!/^#/' out_c.txt | sort -g -k1,1 -k2,2
```

`--sfmagnsigma=FLT`

The median standard deviation (from a `MEDSTD` keyword in the Sky standard deviation image) will be multiplied by the value to this option and its magnitude will be reported in the comments of the output catalog. This value is a per-pixel value, not per object/clump and is not found over an area or aperture, like the common 5σ values that are commonly reported as a measure of depth or the upper-limit measurements (see Section 7.4.2 [Quantifying measurement limits], page 258).

`--sfmagarea=FLT`

Area (in arcseconds squared) to convert the per-pixel estimation of `--sfmagnsigma` in the comments section of the output tables. Note that this is just a unit conversion using the World Coordinate System (WCS) information in the input's header. It does not actually do any measurements on this area. For random measurements on any area, please use the upper-limit columns of MakeCatalog (see the discussion on upper-limit measurements in Section 7.4.2 [Quantifying measurement limits], page 258).

7.5 Match

Data can come come from different telescopes, filters, software and even different configurations for a single software. As a result, one of the primary things to do after generating catalogs from each of these sources (for example with Section 7.4 [MakeCatalog], page 255), is to find which sources in one catalog correspond to which in the other(s). In other words, to 'match' the two catalogs with each other.

Gnuastro's Match program is in charge of such operations. The nearest objects in the two catalogs, within the given aperture, will be found and given as output. The aperture can be a circle or an ellipse with any orientation.

7.5.1 Invoking Match

When given two catalogs, Match finds the rows that are nearest to each other within an input aperture. The executable name is `astmatch` with the following general template

```
$ astmatch [OPTION ...] input-1 input-2
```

One line examples:

```
## 1D wavelength match (within 5 angstroms) of the two inputs.
## The wavelengths are in the 5th and 10th columns respectively.
$ astmatch --aperture=5e-10 --ccol1=5 --ccol2=10 in1.fits in2.txt

## Match the two catalogs with a circular aperture of width 2.
```

```
## (Units same as given positional columns).
## (By default two columns are given for '--ccol1' and '--ccol2',
##  The number of values to these determines the dimensions).
$ astmatch --aperture=2 input1.txt input2.fits

## Similar to before, but the output is created by merging various
## columns from the two inputs: columns 1, RA, DEC from the first
## input, followed by all columns starting with 'MAG' and the 'BRG'
## column from second input and finally the 10th from first input.
$ astmatch --aperture=2 input1.txt input2.fits                    \
           --outcols=a1,aRA,aDEC,b/^MAG/,bBRG,a10

## Match the two catalogs within an elliptical aperture of 1 and 2
## arcseconds along RA and Dec respectively.
$ astmatch --aperture=1/3600,2/3600 in1.fits in2.txt

## Match the RA and DEC columns of the first input with the RA_D
## and DEC_D columns of the second within a 0.5 arcseconds aperture.
$ astmatch --ccol1=RA,DEC --ccol2=RA_D,DEC_D --aperture=0.5/3600  \
           in1.fits in2.fits
```

Match will find the rows that are nearest to each other in two catalogs (given some coordinate columns). Therefore two catalogs are necessary for input. However, they don't necessarily have to be files: 1) the first catalog can also come from the standard input (for example a pipe, see Section 4.1.3 [Standard input], page 104); 2) when only one point is needed, you can use the `--coord` option to avoid creating a file for the second catalog. When the inputs are files, they can be plain text tables or FITS tables, for more see Section 4.6 [Tables], page 117.

Match follows the same basic behavior of all Gnuastro programs as fully described in Chapter 4 [Common program behavior], page 91. If the first input is a FITS file, the common `--hdu` option (see Section 4.1.2.1 [Input/Output options], page 95) should be used to identify the extension. When the second input is FITS, the extension must be specified with `--hdu2`.

When `--quiet` is not called, Match will print the number of matches found in standard output (on the command-line). When matches are found, by default, the output file(s) will be the re-arranged input tables such that the rows match each other: both output tables will have the same number of rows which are matched with each other. If `--outcols` is called, the output is a single table with rows chosen from either of the two inputs in any order. If the `--logasoutput` option is called, the output will be a single table with the contents of the log file, see below. If no matches are found, the columns of the output table(s) will have zero rows (with proper meta-data).

If no output file name is given with the `--output` option, then automatic output Section 4.8 [Automatic output], page 124, will be used to determine the output name(s). Depending on `--tableformat` (see Section 4.1.2.1 [Input/Output options], page 95), the output will then be a (possibly multi-extension) FITS file or (possibly two) plain text file(s). When the output is a FITS file, the default re-arranged inputs will be two

extensions of the output FITS file. With `--outcols` and `--logasoutput`, the FITS output will be a single table (in one extension).

When the `--log` option is called (see Section 4.1.2.3 [Operating mode options], page 100), and there was a match, Match will also create a file named `astmatch.fits` (or `astmatch.txt`, depending on `--tableformat`, see Section 4.1.2.1 [Input/Output options], page 95) in the directory it is run in. This log table will have three columns. The first and second columns show the matching row/record number (counting from 1) of the first and second input catalogs respectively. The third column is the distance between the two matched positions. The units of the distance are the same as the given coordinates (given the possible ellipticity, see description of `--aperture` below). When `--logasoutput` is called, no log file (with a fixed name) will be created. In this case, the output file (possibly given by the `--output` option) will have the contents of this log file.

`--log` **isn't thread-safe**: As described above, when `--logasoutput` is not called, the Log file has a fixed name for all calls to Match. Therefore if a separate log is requested in two simultaneous calls to Match in the same directory, Match will try to write to the same file. This will cause problems like unreasonable log file, undefined behavior, or a crash.

`-H STR`
`--hdu2=STR`
: The extension/HDU of the second input if it is a FITS file. When it isn't a FITS file, this option's value is ignored. For the first input, the common option `--hdu` must be used.

`--outcols=STR`
: Columns (from both inputs) to write into a single matched table output. The value to `--outcols` must be a comma-separated list of strings. The first character of each string specifies the input catalog: `a` for the first and `b` for the second. The rest of the characters of the string will be directly used to identify the proper column(s) in the respective table. See Section 4.6.3 [Selecting table columns], page 121, for how columns can be specified in Gnuastro.

  For example the output of `--outcols=a1,bRA,bDEC` will have three columns: the first column of the first input, along with the `RA` and `DEC` columns of the second input.

  If the string after `a` or `b` is `_all`, then all the columns of the respective input file will be written in the output. For example the command below will print all the input columns from the first catalog along with the 5th column from the second:

  ```
  $ astmatch a.fits b.fits --outcols=a_all,b5
  ```

  `_all` can be used multiple times, possibly on both inputs. Tip: if an input's column is called `_all` (an unlikely name!) and you don't want all the columns from that table the output, use its column number to avoid confusion.

  Another example is given in the one-line examples above. Compared to the default case (where two tables with all their columns) are saved separately, using this option is much faster: it will only read and re-arrange the necessary columns and it will write a single output table. Combined with regular expressions in

large tables, this can be a very powerful and convenient way to merge various tables into one.

When `--coord` is given, no second catalog will be read. The second catalog will be created internally based on the values given to `--coord`. So column names aren't defined and you can only request integer column numbers that are less than the number of coordinates given to `--coord`. For example if you want to find the row matching RA of 1.2345 and Dec of 6.7890, then you should use `--coord=1.2345,6.7890`. But when using `--outcols`, you can't give `bRA`, or `b25`.

`-l`
`--logasoutput`
The output file will have the contents of the log file: indexes in the two catalogs that match with each other along with their distance. See description above. When this option is called, a log file called `astmatch.txt` will not be created. With this option, the default output behavior (two tables containing the rearranged inputs) will be

`--notmatched`
Write the non-matching rows into the outputs, not the matched ones. Note that with this option, the two output tables will not necessarily have the same number of rows. Therefore, this option cannot be called with `--outcols`. `--outcols` prints mixed columns from both inputs, so they must all have the same number of elements and must correspond to each other.

`-c INT/STR[,INT/STR]`
`--ccol1=INT/STR[,INT/STR]`
The coordinate columns of the first input. The number of dimensions for the match is determined by the number of comma-separated values given to this option. The values can be the column number (counting from 1), exact column name or a regular expression. For more, see Section 4.6.3 [Selecting table columns], page 121. See the one-line examples above for some usages of this option.

`-C INT/STR[,INT/STR]`
`--ccol2=INT/STR[,INT/STR]`
The coordinate columns of the second input. See the example in `--ccol1` for more.

`-d FLT[,FLT]`
`--coord=FLT[,FLT]`
Manually specify the coordinates to match against the given catalog. With this option, Match will not look for a second input file/table and will directly use the coordinates given to this option.

When this option is called, the output changes in the following ways: 1) when `--outcols` is specified, for the second input, it can only accept integer numbers that are less than the number of values given to this option, see description of that option for more. 2) By default (when `--outcols` isn't used), only the matching row of the first table will be output (a single file), not two separate files (one for each table).

This option is good when you have a (large) catalog and only want to match a single coordinate to it (for example to find the nearest catalog entry to your desired point). With this option, you can write the coordinates on the command-line and thus avoid the need to make a single-row file.

`-a FLT[,FLT[,FLT]]`
`--aperture=FLT[,FLT[,FLT]]`

Parameters of the aperture for matching. The values given to this option can be fractions, for example when the position columns are in units of degrees, `1/3600` can be used to ask for one arcsecond. The interpretation of the values depends on the requested dimensions (determined from `--ccol1` and `--ccol2`) and how many values are given to this option.

1D match The aperture/interval can only take one value: half of the interval around each point (maximum distance from each point).

2D match In a 2D match, the aperture can be a circle, an ellipse aligned in the axes or an ellipse with a rotated major axis. To simply the usage, you can determine the shape based on the number of free parameters for each.

1 number For example `--aperture=2`. The aperture will be a circle of the given radius. The value will be in the same units as the columns in `--ccol1` and `--ccol2`).

2 numbers For example `--aperture=3,4e-10`. The aperture will be an ellipse (if the two numbers are different) with the respective value along each dimension. The numbers are in units of the first and second axis. In the example above, the semi-axis value along the first axis will be 3 (in units of the first coordinate) and along the second axis will be 4×10^{-10} (in units of the second coordinate). Such values can happen if you are comparing catalogs of a spectra for example. If more than one object exists in the aperture, the nearest will be found along the major axis as described in Section 8.1.1.1 [Defining an ellipse], page 284.

3 numbers For example `--aperture=2,0.6,30`. The aperture will be an ellipse (if the second value is not 1). The first number is the semi-major axis, the second is the axis ratio and the third is the position angle (in degrees). If multiple matches are found within the ellipse, the distance (to find the nearest) is calculated along the major axis in the elliptical space, see Section 8.1.1.1 [Defining an ellipse], page 284.

8 Modeling and fitting

In order to fully understand observations after initial analysis on the image, it is very important to compare them with the existing models to be able to further understand both the models and the data. The tools in this chapter create model galaxies and will provide 2D fittings to be able to understand the detections.

8.1 MakeProfiles

MakeProfiles will create mock astronomical profiles from a catalog, either individually or together in one output image. In data analysis, making a mock image can act like a calibration tool, through which you can test how successfully your detection technique is able to detect a known set of objects. There are commonly two aspects to detecting: the detection of the fainter parts of bright objects (which in the case of galaxies fade into the noise very slowly) or the complete detection of an over-all faint object. Making mock galaxies is the most accurate (and idealistic) way these two aspects of a detection algorithm can be tested. You also need mock profiles in fitting known functional profiles with observations.

MakeProfiles was initially built for extra galactic studies, so currently the only astronomical objects it can produce are stars and galaxies. We welcome the simulation of any other astronomical object. The general outline of the steps that MakeProfiles takes are the following:

1. Build the full profile out to its truncation radius in a possibly over-sampled array.
2. Multiply all the elements by a fixed constant so its total magnitude equals the desired total magnitude.
3. If `--individual` is called, save the array for each profile to a FITS file.
4. If `--nomerged` is not called, add the overlapping pixels of all the created profiles to the output image and abort.

Using input values, MakeProfiles adds the World Coordinate System (WCS) headers of the FITS standard to all its outputs (except PSF images!). For a simple test on a set of mock galaxies in one image, there is no need for the third step or the WCS information.

However in complicated simulations like weak lensing simulations, where each galaxy undergoes various types of individual transformations based on their position, those transformations can be applied to the different individual images with other programs. After all the transformations are applied, using the WCS information in each individual profile image, they can be merged into one output image for convolution and adding noise.

8.1.1 Modeling basics

In the subsections below, first a review of some very basic information and concepts behind modeling a real astronomical image is given. You can skip this subsection if you are already sufficiently familiar with these concepts.

8.1.1.1 Defining an ellipse

The PSF, see Section 8.1.1.2 [Point spread function], page 285, and galaxy radial profiles are generally defined on an ellipse. Therefore, in this section we'll start defining an ellipse on a pixelated 2D surface. Labeling the major axis of an ellipse a, and its minor axis with

b, the *axis ratio* is defined as: $q \equiv b/a$. The major axis of an ellipse can be aligned in any direction, therefore the angle of the major axis with respect to the horizontal axis of the image is defined to be the *position angle* of the ellipse and in this book, we show it with θ.

Our aim is to put a radial profile of any functional form $f(r)$ over an ellipse. Hence we need to associate a radius/distance to every point in space. Let's define the radial distance r_{el} as the distance on the major axis to the center of an ellipse which is located at i_c and j_c (in other words $r_{el} \equiv a$). We want to find r_{el} of a point located at (i, j) (in the image coordinate system) from the center of the ellipse with axis ratio q and position angle θ. First the coordinate system is rotated[1] by θ to get the new rotated coordinates of that point (i_r, j_r):

$$i_r(i,j) = +(i_c - i)\cos\theta + (j_c - j)\sin\theta$$

$$j_r(i,j) = -(i_c - i)\sin\theta + (j_c - j)\cos\theta$$

Recall that an ellipse is defined by $(i_r/a)^2 + (j_r/b)^2 = 1$ and that we defined $r_{el} \equiv a$. Hence, multiplying all elements of the the ellipse definition with r_{el}^2 we get the elliptical distance at this point point located: $r_{el} = \sqrt{i_r^2 + (j_r/q)^2}$. To place the radial profiles explained below over an ellipse, $f(r_{el})$ is calculated based on the functional radial profile desired.

MakeProfiles builds the profile starting from the nearest element (pixel in an image) in the dataset to the profile center. The profile value is calculated for that central pixel using monte carlo integration, see Section 8.1.1.5 [Sampling from a function], page 288. The next pixel is the next nearest neighbor to the central pixel as defined by r_{el}. This process goes on until the profile is fully built upto the truncation radius. This is done fairly efficiently using a breadth first parsing strategy[2] which is implemented through an ordered linked list.

Using this approach, we build the profile by expanding the circumference. Not one more extra pixel has to be checked (the calculation of r_{el} from above is not cheap in CPU terms). Another consequence of this strategy is that extending MakeProfiles to three dimensions becomes very simple: only the neighbors of each pixel have to be changed. Everything else after that (when the pixel index and its radial profile have entered the linked list) is the same, no matter the number of dimensions we are dealing with.

8.1.1.2 Point spread function

Assume we have a 'point' source, or a source that is far smaller than the maximum resolution (a pixel). When we take an image of it, it will 'spread' over an area. To quantify that spread, we can define a 'function'. This is how the point spread function or the PSF of an image is defined. This 'spread' can have various causes, for example in ground based astronomy, due to the atmosphere. In practice we can never surpass the 'spread' due to the diffraction of the lens aperture. Various other effects can also be quantified through a PSF. For example, the simple fact that we are sampling in a discrete space, namely the pixels, also produces a very small 'spread' in the image.

[1] Do not confuse the signs of *sin* with the rotation matrix defined in Section 6.4.1 [Warping basics], page 200. In that equation, the point is rotated, here the coordinates are rotated and the point is fixed.

[2] `http://en.wikipedia.org/wiki/Breadth-first_search`

Convolution is the mathematical process by which we can apply a 'spread' to an image, or in other words blur the image, see Section 6.3.1.1 [Convolution process], page 178. The Brightness of an object should remain unchanged after convolution, see Section 8.1.3 [Flux Brightness and magnitude], page 290. Therefore, it is important that the sum of all the pixels of the PSF be unity. The PSF image also has to have an odd number of pixels on its sides so one pixel can be defined as the center. In MakeProfiles, the PSF can be set by the two methods explained below.

Parametric functions

A known mathematical function is used to make the PSF. In this case, only the parameters to define the functions are necessary and MakeProfiles will make a PSF based on the given parameters for each function. In both cases, the center of the profile has to be exactly in the middle of the central pixel of the PSF (which is automatically done by MakeProfiles). When talking about the PSF, usually, the full width at half maximum or FWHM is used as a scale of the width of the PSF.

Gaussian In the older papers, and to a lesser extent even today, some researchers use the 2D Gaussian function to approximate the PSF of ground based images. In its most general form, a Gaussian function can be written as:

$$f(r) = a\exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right) + d$$

Since the center of the profile is pre-defined, μ and d are constrained. a can also be found because the function has to be normalized. So the only important parameter for MakeProfiles is the σ. In the Gaussian function we have this relation between the FWHM and σ:

$$\mathrm{FWHM_g} = 2\sqrt{2\ln 2}\sigma \approx 2.35482\sigma$$

Moffat The Gaussian profile is much sharper than the images taken from stars on photographic plates or CCDs. Therefore in 1969, Moffat proposed this functional form for the image of stars:

$$f(r) = a\left[1 + \left(\frac{r}{\alpha}\right)^2\right]^{-\beta}$$

Again, a is constrained by the normalization, therefore two parameters define the shape of the Moffat function: α and β. The radial parameter is α which is related to the FWHM by

$$\mathrm{FWHM_m} = 2\alpha\sqrt{2^{1/\beta} - 1}$$

Comparing with the PSF predicted from atmospheric turbulence theory with a Moffat function, Trujillo et al.[3] claim that β should be 4.765. They also show how the Moffat PSF contains the Gaussian PSF as a limiting case when $\beta \to \infty$.

An input FITS image

An input image file can also be specified to be used as a PSF. If the sum of its pixels are not equal to 1, the pixels will be multiplied by a fraction so the sum does become 1.

While the Gaussian is only dependent on the FWHM, the Moffat function is also dependent on β. Comparing these two functions with a fixed FWHM gives the following results:

- Within the FWHM, the functions don't have significant differences.
- For a fixed FWHM, as β increases, the Moffat function becomes sharper.
- The Gaussian function is much sharper than the Moffat functions, even when β is large.

8.1.1.3 Stars

In MakeProfiles, stars are generally considered to be a point source. This is usually the case for extra galactic studies, were nearby stars are also in the field. Since a star is only a point source, we assume that it only fills one pixel prior to convolution. In fact, exactly for this reason, in astronomical images the light profiles of stars are one of the best methods to understand the shape of the PSF and a very large fraction of scientific research is preformed by assuming the shapes of stars to be the PSF of the image.

8.1.1.4 Galaxies

Today, most practitioners agree that the flux of galaxies can be modeled with one or a few generalized de Vaucouleur's (or Sérsic) profiles.

$$I(r) = I_e \exp\left(-b_n\left[\left(\frac{r}{r_e}\right)^{1/n} - 1\right]\right)$$

Gérard de Vaucouleurs (1918-1995) was first to show in 1948 that this function best fits the galaxy light profiles, with the only difference that he held n fixed to a value of 4. 20 years later in 1968, J. L. Sérsic showed that n can have a variety of values and does not necessarily need to be 4. This profile depends on the effective radius (r_e) which is defined as the radius which contains half of the profile brightness (see Section 8.1.4 [Profile magnitude], page 290). I_e is the flux at the effective radius. The Sérsic index n is used to define the concentration of the profile within r_e and b_n is a constant dependent on n. MacArthur et al.[4] show that for $n > 0.35$, b_n can be accurately approximated using this equation:

[3] Trujillo, I., J. A. L. Aguerri, J. Cepa, and C. M. Gutierrez (2001). "The effects of seeing on Sérsic profiles - II. The Moffat PSF". In: MNRAS 328, pp. 977—985.

[4] MacArthur, L. A., S. Courteau, and J. A. Holtzman (2003). "Structure of Disk-dominated Galaxies. I. Bulge/Disk Parameters, Simulations, and Secular Evolution". In: ApJ 582, pp. 689—722.

$$b_n = 2n - \frac{1}{3} + \frac{4}{405n} + \frac{46}{25515n^2} + \frac{131}{1148175n^3} - \frac{2194697}{30690717750n^4}$$

8.1.1.5 Sampling from a function

A pixel is the ultimate level of accuracy to gather data, we can't get any more accurate in one image, this is known as sampling in signal processing. However, the mathematical profiles which describe our models have infinite accuracy. Over a large fraction of the area of astrophysically interesting profiles (for example galaxies or PSFs), the variation of the profile over the area of one pixel is not too significant. In such cases, the elliptical radius (r_{el} of the center of the pixel can be assigned as the final value of the pixel, see Section 8.1.1.1 [Defining an ellipse], page 284).

As you approach their center, some galaxies become very sharp (their value significantly changes over one pixel's area). This sharpness increases with smaller effective radius and larger Sérsic values. Thus rendering the central value extremely inaccurate. The first method that comes to mind for solving this problem is integration. The functional form of the profile can be integrated over the pixel area in a 2D integration process. However, unfortunately numerical integration techniques also have their limitations and when such sharp profiles are needed they can become extremely inaccurate.

The most accurate method of sampling a continuous profile on a discrete space is by choosing a large number of random points within the boundaries of the pixel and taking their average value (or Monte Carlo integration). This is also, generally speaking, what happens in practice with the photons on the pixel. The number of random points can be set with `--numrandom`.

Unfortunately, repeating this Monte Carlo process would be extremely time and CPU consuming if it is to be applied to every pixel. In order to not loose too much accuracy, in MakeProfiles, the profile is built using both methods explained below. The building of the profile begins from its central pixel and continues (radially) outwards. Monte Carlo integration is first applied (which yields F_r), then the central pixel value (F_c) is calculated on the same pixel. If the fractional difference ($|F_r - F_c|/F_r$) is lower than a given tolerance level (specified with `--tolerance`) MakeProfiles will stop using Monte Carlo integration and only use the central pixel value.

The ordering of the pixels in this inside-out construction is based on $r = \sqrt{(i_c - i)^2 + (j_c - j)^2}$, not r_{el}, see Section 8.1.1.1 [Defining an ellipse], page 284. When the axis ratios are large (near one) this is fine. But when they are small and the object is highly elliptical, it might seem more reasonable to follow r_{el} not r. The problem is that the gradient is stronger in pixels with smaller r (and larger r_{el}) than those with smaller r_{el}. In other words, the gradient is strongest along the minor axis. So if the next pixel is chosen based on r_{el}, the tolerance level will be reached sooner and lots of pixels with large fractional differences will be missed.

Monte Carlo integration uses a random number of points. Thus, every time you run it, by default, you will get a different distribution of points to sample within the pixel. In the case of large profiles, this will result in a slight difference of the pixels which use Monte Carlo integration each time MakeProfiles is run. To have a deterministic result, you have to fix the random number generator properties which is used to build the random distribution.

This can be done by setting the `GSL_RNG_TYPE` and `GSL_RNG_SEED` environment variables and calling MakeProfiles with the `--envseed` option. To learn more about the process of generating random numbers, see Section 8.2.1.4 [Generating random numbers], page 304.

The seed values are fixed for every profile: with `--envseed`, all the profiles have the same seed and without it, each will get a different seed using the system clock (which is accurate to within one microsecond). The same seed will be used to generate a random number for all the sub-pixel positions of all the profiles. So in the former, the sub-pixel points checked for all the pixels undergoing Monte carlo integration in all profiles will be identical. In other words, the sub-pixel points in the first (closest to the center) pixel of all the profiles will be identical with each other. All the second pixels studied for all the profiles will also receive an identical (different from the first pixel) set of sub-pixel points and so on. As long as the number of random points used is large enough or the profiles are not identical, this should not cause any systematic bias.

8.1.1.6 Oversampling

The steps explained in Section 8.1.1.5 [Sampling from a function], page 288, do give an accurate representation of a profile prior to convolution. However, in an actual observation, the image is first convolved with or blurred by the atmospheric and instrument PSF in a continuous space and then it is sampled on the discrete pixels of the camera.

In order to more accurately simulate this process, the unconvolved image and the PSF are created on a finer pixel grid. In other words, the output image is a certain odd-integer multiple of the desired size, we can call this 'oversampling'. The user can specify this multiple as a command-line option. The reason this has to be an odd number is that the PSF has to be centered on the center of its image. An image with an even number of pixels on each side does not have a central pixel.

The image can then be convolved with the PSF (which should also be oversampled on the same scale). Finally, image can be sub-sampled to get to the initial desired pixel size of the output image. After this, mock noise can be added as explained in the next section. This is because unlike the PSF, the noise occurs in each output pixel, not on a continuous space like all the prior steps.

8.1.2 If convolving afterwards

In case you want to convolve the image later with a given point spread function, make sure to use a larger image size. After convolution, the profiles become larger and a profile that is normally completely outside of the image might fall within it.

On one axis, if you want your final (convolved) image to be m pixels and your PSF is $2n+1$ pixels wide, then when calling MakeProfiles, set the axis size to $m+2n$, not m. You also have to shift all the pixel positions of the profile centers on the that axis by n pixels to the positive.

After convolution, you can crop the outer n pixels with the section crop box specification of Crop: `--section=n:*-n,n:*-n` assuming your PSF is a square, see Section 6.1.2 [Crop section syntax], page 154. This will also remove all discrete Fourier transform artifacts (blurred sides) from the final image. To facilitate this shift, MakeProfiles has the options `--xshift`, `--yshift` and `--prepforconv`, see Section 8.1.5 [Invoking MakeProfiles], page 291.

8.1.3 Flux Brightness and magnitude

Astronomical data pixels are usually in units of counts[5] or electrons or either one divided by seconds. To convert from the counts to electrons, you will need to know the instrument gain. In any case, they can be directly converted to energy or energy/time using the basic hardware (telescope, camera and filter) information. We will continue the discussion assuming the pixels are in units of energy/time.

The *brightness* of an object is defined as its total detected energy per time. This is simply the sum of the pixels that are associated with that detection by our detection tool for example Section 7.2 [NoiseChisel], page 225[6]. The *flux* of an object is in units of energy/time/area and for a detected object, it is defined as its brightness divided by the area used to collect the light from the source or the telescope aperture (for example in cm^2)[7]. Knowing the flux (f) and distance to the object (r), we can calculate its *luminosity*: $L = 4\pi r^2 f$. Therefore, flux and luminosity are intrinsic properties of the object, while brightness depends on our detecting tools (hardware and software). Here we will not be discussing luminosity, but brightness. However, since luminosity is the astrophysically interesting quantity, we also defined it here to avoid possible confusion between these two terms because they both have the same units.

Images of astronomical objects span over a very large range of brightness. With the Sun (as the brightest object) being roughly $2.5^{60} = 10^{24}$ times brighter than the faintest galaxies we can currently detect. Therefore discussing brightness will be very hard, and astronomers have chosen to use a logarithmic scale to talk about the brightness of astronomical objects. But the logarithm can only be usable with a unit-less and always positive value. Fortunately brightness is always positive and to remove the units we divide the brightness of the object (B) by a reference brightness (B_r). We then define the resulting logarithmic scale as *magnitude* through the following relation[8]

$$m - m_r = -2.5 \log_{10} \left(\frac{B}{B_r} \right)$$

m is defined as the magnitude of the object and m_r is the pre-defined magnitude of the reference brightness. One particularly easy condition is when $B_r = 1$. This will allow us to summarize all the hardware specific parameters discussed above into one number as the reference magnitude which is commonly known as the Zero-point[9] magnitude.

8.1.4 Profile magnitude

To find the profile brightness or its magnitude, (see Section 8.1.3 [Flux Brightness and magnitude], page 290), it is customary to use the 2D integration of the flux to infinity.

[5] Counts are also known as analog to digital units (ADU).

[6] If further processing is done, for example the Kron or Petrosian radii are calculated, then the detected area is not sufficient and the total area that was within the respective radius must be used.

[7] For a full object that spans over several pixels, the telescope area should be used to find the flux. However, sometimes, only the brightness per pixel is desired. In such cases this book also *loosely* uses the term flux. This is only approximately accurate however, since while all the pixels have a fixed area, the pixel size can vary with camera on the telescope.

[8] The -2.5 factor in the definition of magnitudes is a legacy of the our ancient colleagues and in particular Hipparchus of Nicaea (190-120 BC).

[9] When $B = Br = 1$, the right side of the magnitude definition will be zero. Hence the name, "zero-point".

However, in MakeProfiles we do not follow this idealistic approach and apply a more realistic method to find the total brightness or magnitude: the sum of all the pixels belonging to a profile within its predefined truncation radius. Note that if the truncation radius is not large enough, this can be significantly different from the total integrated light to infinity.

An integration to infinity is not a realistic condition because no galaxy extends indefinitely (important for high Sérsic index profiles), pixelation can also cause a significant difference between the actual total pixel sum value of the profile and that of integration to infinity, especially in small and high Sérsic index profiles. To be safe, you can specify a large enough truncation radius for such compact high Sérsic index profiles.

If oversampling is used then the brightness is calculated using the over-sampled image, see Section 8.1.1.6 [Oversampling], page 289, which is much more accurate. The profile is first built in an array completely bounding it with a normalization constant of unity (see Section 8.1.1.4 [Galaxies], page 287). Taking B to be the desired brightness and S to be the sum of the pixels in the created profile, every pixel is then multiplied by B/S so the sum is exactly B.

If the `--individual` option is called, this same array is written to a FITS file. If not, only the overlapping pixels of this array and the output image are kept and added to the output array.

8.1.5 Invoking MakeProfiles

MakeProfiles will make any number of profiles specified in a catalog either individually or in one image. The executable name is `astmkprof` with the following general template

```
$ astmkprof [OPTION ...] [Catalog]
```

One line examples:

```
## Make an image with profiles in catalog.txt (with default size):
$ astmkprof catalog.txt

## Make the profiles in catalog.txt over image.fits:
$ astmkprof --background=image.fits catalog.txt

## Make a Moffat PSF with FWHM 3pix, beta=2.8, truncation=5
$ astmkprof --kernel=moffat,2.8,5 --oversample=1

## Make profiles in catalog, using RA and Dec in the given column:
$ astmkprof --ccol=RA_CENTER --ccol=DEC_CENTER --mode=wcs catalog.txt

## Make a 1500x1500 merged image (oversampled 500x500) image along
## with an individual image for all the profiles in catalog:
$ astmkprof --individual --oversample 3 --mergedsize=500,500 cat.txt
```

The parameters of the mock profiles can either be given through a catalog (which stores the parameters of many mock profiles, see Section 8.1.5.1 [MakeProfiles catalog], page 292), or the `--kernel` option (see Section 8.1.5.3 [MakeProfiles output dataset], page 297). The catalog can be in the FITS ASCII, FITS binary format, or plain text formats (see Section 4.6 [Tables], page 117). A plain text catalog can also be provided using the Standard input (see Section 4.1.3 [Standard input], page 104). The columns related to each parameter can be

determined both by number, or by match/search criteria using the column names, units, or comments. with the options ending in `col`, see below.

Without any file given to the `--background` option, MakeProfiles will make a zero-valued image and build the profiles on that (its size and main WCS parameters can also be defined through the options described in Section 8.1.5.3 [MakeProfiles output dataset], page 297). Besides the main/merged image containing all the profiles in the catalog, it is also possible to build individual images for each profile (only enclosing one full profile to its truncation radius) with the `--individual` option.

If an image is given to the `--background` option, the pixels of that image are used as the background value for every pixel. The flux value of each profile pixel will be added to the pixel in that background value. In this case, the values to all options relating to the output size and WCS will be ignored if specified (for example `--oversample`, `--mergedsize`, and `--prepforconv`) on the command-line or in the configuration files.

The sections below discuss the options specific to MakeProfiles based on context: the input catalog settings which can have many rows for different profiles are discussed in Section 8.1.5.1 [MakeProfiles catalog], page 292, in Section 8.1.5.2 [MakeProfiles profile settings], page 294, we discuss how you can set general profile settings (that are the same for all the profiles in the catalog). Finally Section 8.1.5.3 [MakeProfiles output dataset], page 297, and Section 8.1.5.4 [MakeProfiles log file], page 301, discuss the outputs of MakeProfiles and how you can configure them. Besides these, MakeProfiles also supports all the common Gnuastro program options that are discussed in Section 4.1.2 [Common options], page 95, so please flip through them is well for a more comfortable usage.

Please see Section 2.1 [Sufi simulates a detection], page 17, for a very complete tutorial explaining how one could use MakeProfiles in conjunction with other Gnuastro's programs to make a complete simulated image of a mock galaxy.

8.1.5.1 MakeProfiles catalog

The catalog containing information about each profile can be in the FITS ASCII, FITS binary, or plain text formats (see Section 4.6 [Tables], page 117). The latter can also be provided using standard input (see Section 4.1.3 [Standard input], page 104). Its columns can be ordered in any desired manner. You can specify which columns belong to which parameters using the set of options discussed below. For example through the `--rcol` and `--tcol` options, you can specify the column that contains the radial parameter for each profile and its truncation respectively. See Section 4.6.3 [Selecting table columns], page 121, for a thorough discussion on the values to these options.

The value for the profile center in the catalog (the `--ccol` option) can be a floating point number so the profile center can be on any sub-pixel position. Note that pixel positions in the FITS standard start from 1 and an integer is the pixel center. So a 2D image actually starts from the position (0.5, 0.5), which is the bottom-left corner of the first pixel. When a `--background` image with WCS information is provided or you specify the WCS parameters with the respective options, you may also use RA and Dec to identify the center of each profile (see the `--mode` option below).

In MakeProfiles, profile centers do not have to be in (overlap with) the final image. Even if only one pixel of the profile within the truncation radius overlaps with the final image size, the profile is built and included in the final image image. Profiles that are completely

out of the image will not be created (unless you explicitly ask for it with the `--individual` option). You can use the output log file (created with `--log` to see which profiles were within the image, see Section 4.1.2 [Common options], page 95.

If PSF profiles (Moffat or Gaussian, see Section 8.1.1.2 [Point spread function], page 285) are in the catalog and the profiles are to be built in one image (when `--individual` is not used), it is assumed they are the PSF(s) you want to convolve your created image with. So by default, they will not be built in the output image but as separate files. The sum of pixels of these separate files will also be set to unity (1) so you are ready to convolve, see Section 6.3.1.1 [Convolution process], page 178. As a summary, the position and magnitude of PSF profile will be ignored. This behavior can be disabled with the `--psfinimg` option. If you want to create all the profiles separately (with `--individual`) and you want the sum of the PSF profile pixels to be unity, you have to set their magnitudes in the catalog to the zero-point magnitude and be sure that the central positions of the profiles don't have any fractional part (the PSF center has to be in the center of the pixel).

The list of options directly related to the input catalog columns is shown below.

`--ccol=STR/INT`
: Center coordinate column for each dimension. This option must be called two times to define the center coordinates in an image. For example `--ccol=RA` and `--ccol=DEC` (along with `--mode=wcs`) will inform MakeProfiles to look into the catalog columns named `RA` and `DEC` for the Right Ascension and Declination of the profile centers.

`--fcol=INT/STR`
: The functional form of the profile with one of the values below depending on the desired profile. The column can contain either the numeric codes (for example '`1`') or string characters (for example '`sersic`'). The numeric codes are easier to use in scripts which generate catalogs with hundreds or thousands of profiles.

  The string format can be easier when the catalog is to be written/checked by hand/eye before running MakeProfiles. It is much more readable and provides a level of documentation. All Gnuastro's recognized table formats (see Section 4.6.1 [Recognized table formats], page 117) accept string type columns. To have string columns in a plain text table/catalog, see Section 4.6.2 [Gnuastro text table format], page 119.

  - Sérsic profile with '`sersic`' or '`1`'.
  - Moffat profile with '`moffat`' or '`2`'.
  - Gaussian profile with '`gaussian`' or '`3`'.
  - Point source with '`point`' or '`4`'.
  - Flat profile with '`flat`' or '`5`'.
  - Circumference profile with '`circum`' or '`6`'. A fixed value will be used for all pixels less than or equal to the truncation radius (r_t) and greater than $r_t - w$ (w is the value to the `--circumwidth`).
  - Radial distance profile with '`distance`' or '`7`'. At the lowest level, each pixel only has an elliptical radial distance given the profile's shape and orientation (see Section 8.1.1.1 [Defining an ellipse], page 284). When this profile is chosen, the pixel's elliptical radial distance from the profile center

is written as its value. For this profile, the value in the magnitude column (`--mcol`) will be ignored.

You can use this for checks or as a first approximation to define your own higher-level radial function. In the latter case, just note that the central values are going to be incorrect (see Section 8.1.1.5 [Sampling from a function], page 288).

`--rcol=STR/INT`
The radius parameter of the profiles. Effective radius (r_e) if Sérsic, FWHM if Moffat or Gaussian.

`--ncol=STR/INT`
The Sérsic index (n) or Moffat β.

`--pcol=STR/INT`
The position angle (in degrees) of the profiles relative to the first FITS axis (horizontal when viewed in SAO ds9).

`--qcol=STR/INT`
The axis ratio of the profiles (minor axis divided by the major axis in a 2D ellipse).

`--mcol=STR/INT`
The total pixelated magnitude of the profile within the truncation radius, see Section 8.1.4 [Profile magnitude], page 290.

`--tcol=STR/INT`
The truncation radius of this profile. By default it is in units of the radial parameter of the profile (the value in the `--rcol` of the catalog). If `--tunitinp` is given, this value is interpreted in units of pixels (prior to oversampling) irrespective of the profile.

8.1.5.2 MakeProfiles profile settings

The profile parameters that differ between each created profile are specified through the columns in the input catalog and described in Section 8.1.5.1 [MakeProfiles catalog], page 292. Besides those there are general settings for some profiles that don't differ between one profile and another, they are a property of the general process. For example how many random points to use in the monte-carlo integration, this value is fixed for all the profiles. The options described in this section are for configuring such properties.

`--mode=STR`
Interpret the center position columns (`--ccol` in Section 8.1.5.1 [MakeProfiles catalog], page 292) in image or WCS coordinates. This option thus accepts only two values: `img` and `wcs`. It is mandatory when a catalog is being used as input.

`-r`
`--numrandom`
The number of random points used in the central regions of the profile, see Section 8.1.1.5 [Sampling from a function], page 288.

`-e`
`--envseed`
: Use the value to the `GSL_RNG_SEED` environment variable to generate the random Monte Carlo sampling distribution, see Section 8.1.1.5 [Sampling from a function], page 288, and Section 8.2.1.4 [Generating random numbers], page 304.

`-t FLT`
`--tolerance=FLT`
: The tolerance to switch from Monte Carlo integration to the central pixel value, see Section 8.1.1.5 [Sampling from a function], page 288.

`-p`
`--tunitinp`
: The truncation column of the catalog is in units of pixels. By default, the truncation column is considered to be in units of the radial parameters of the profile (`--rcol`). Read it as 't-unit-in-p' for 'truncation unit in pixels'.

`-f`
`--mforflatpix`
: When making fixed value profiles (flat and circumference, see '`--fcol`'), don't use the value in the column specified by '`--mcol`' as the magnitude. Instead use it as the exact value that all the pixels of these profiles should have. This option is irrelevant for other types of profiles. This option is very useful for creating masks, or labeled regions in an image. Any integer, or floating point value can used in this column with this option, including `NaN` (or '`nan`', or '`NAN`', case is irrelevant), and infinities (`inf`, `-inf`, or `+inf`).

  For example, with this option if you set the value in the magnitude column (`--mcol`) to `NaN`, you can create an elliptical or circular mask over an image (which can be given as the argument), see Section 6.1.3 [Blank pixels], page 154. Another useful application of this option is to create labeled elliptical or circular apertures in an image. To do this, set the value in the magnitude column to the label you want for this profile. This labeled image can then be used in combination with NoiseChisel's output (see Section 7.2.2.3 [NoiseChisel output], page 241) to do aperture photometry with MakeCatalog (see Section 7.4 [MakeCatalog], page 255).

  Alternatively, if you want to mark regions of the image (for example with an elliptical circumference) and you don't want to use NaN values (as explained above) for some technical reason, you can get the minimum or maximum value in the image[10] using Arithmetic (see Section 6.2 [Arithmetic], page 161), then use that value in the magnitude column along with this option for all the profiles.

  Please note that when using MakeProfiles on an already existing image, you have to set '`--oversample=1`'. Otherwise all the profiles will be scaled up based on the oversampling scale in your configuration files (see Section 4.2 [Configuration files], page 106) unless you have accounted for oversampling in your catalog.

[10] The minimum will give a better result, because the maximum can be too high compared to most pixels in the image, making it harder to display.

`--mcolisbrightness`
: The value given in the "magnitude column" (specified by `--mcol`, see Section 8.1.5.1 [MakeProfiles catalog], page 292) must be interpreted as brightness, not magnitude. The zeropoint magnitude (value to the `--zeropoint` option) is ignored and the given value must have the same units as the input dataset's pixels.

  Recall that the total profile magnitude or brightness that is specified with in the `--mcol` column of the input catalog is not an integration to infinity, but the actual sum of pixels in the profile (until the desired truncation radius). See Section 8.1.4 [Profile magnitude], page 290, for more on this point.

`--magatpeak`
: The magnitude column in the catalog (see Section 8.1.5.1 [MakeProfiles catalog], page 292) will be used to find the brightness only for the peak profile pixel, not the full profile. Note that this is the flux of the profile's peak pixel in the final output of MakeProfiles. So beware of the oversampling, see Section 8.1.1.6 [Oversampling], page 289.

  This option can be useful if you want to check a mock profile's total magnitude at various truncation radii. Without this option, no matter what the truncation radius is, the total magnitude will be the same as that given in the catalog. But with this option, the total magnitude will become brighter as you increase the truncation radius.

  In sharper profiles, sometimes the accuracy of measuring the peak profile flux is more than the overall object brightness. In such cases, with this option, the final profile will be built such that its peak has the given magnitude, not the total profile.

  CAUTION: If you want to use this option for comparing with observations, please note that MakeProfiles does not do convolution. Unless you have de-convolved your data, your images are convolved with the instrument and atmospheric PSF, see Section 8.1.1.2 [Point spread function], page 285. Particularly in sharper profiles, the flux in the peak pixel is strongly decreased after convolution. Also note that in such cases, besides de-convolution, you will have to set `--oversample=1` otherwise after resampling your profile with Warp (see Section 6.4 [Warp], page 199), the peak flux will be different.

`-X INT,INT`
`--shift=INT,INT`
: Shift all the profiles and enlarge the image along each dimension. To better understand this option, please see n in Section 8.1.2 [If convolving afterwards], page 289. This is useful when you want to convolve the image afterwards. If you are using an external PSF, be sure to oversample it to the same scale used for creating the mock images. If a background image is specified, any possible value to this option is ignored.

`-c`
`--prepforconv`
: Shift all the profiles and enlarge the image based on half the width of the first Moffat or Gaussian profile in the catalog, considering any possible oversampling see Section 8.1.2 [If convolving afterwards], page 289. `--prepforconv` is only checked and possibly activated if `--xshift` and `--yshift` are both zero (after reading the command-line and configuration files). If a background image is specified, any possible value to this option is ignored.

`-z FLT`
`--zeropoint=FLT`
: The zero-point magnitude of the image.

`-w FLT`
`--circumwidth=FLT`
: The width of the circumference if the profile is to be an elliptical circumference or annulus. See the explanations for this type of profile in `--fcol`.

`-R`
`--replace`
: Do not add the pixels of each profile over the background (possibly crowded by other profiles), replace them. By default, when two profiles overlap, the final pixel value is the sum of all the profiles that overlap on that pixel. When this option is given, the pixels are not added but replaced by the newer profile's pixel and any value under it is lost.

  When order matters, make sure to use this function with '`--numthreads=1`'. When multiple threads are used, the separate profiles are built asynchronously and not in order. Since order does not matter in an addition, this causes no problems by default but has to be considered when this option is given. Using multiple threads is no problem if the profiles are to be used as a mask with a blank or fixed value (see '`--mforflatpix`') since all their pixel values are the same.

  Note that only non-zero pixels are replaced. With radial profiles (for example Sérsic or Moffat) only values above zero will be part of the profile. However, when using flat profiles with the '`--mforflatpix`' option, you should be careful not to give a `0.0` value as the flat profile's pixel value.

8.1.5.3 MakeProfiles output dataset

MakeProfiles takes an input catalog uses basic properties that are defined there to build a dataset, for example a 2D image containing the profiles in the catalog. In Section 8.1.5.1 [MakeProfiles catalog], page 292, and Section 8.1.5.2 [MakeProfiles profile settings], page 294, the catalog and profile settings were discussed. The options of this section, allow you to configure the output dataset (or the canvas that will host the built profiles).

`-k STR`
`--background=STR`
: A background image FITS file to build the profiles on. The extension that contains the image should be specified with the `--backhdu` option, see below.

When a background image is specified, it will be used to derive all the information about the output image. Hence, the following options will be ignored: `--mergedsize`, `--oversample`, `--crpix`, `--crval` (generally, all other WCS related parameters) and the output's data type (see `--type` in Section 4.1.2.1 [Input/Output options], page 95).

The image will act like a canvas to build the profiles on: profile pixel values will be summed with the background image pixel values. With the `--replace` option you can disable this behavior and replace the profile pixels with the background pixels. If you want to use all the image information above, except for the pixel values (you want to have a blank canvas to build the profiles on, based on an input image), you can call `--clearcanvas`, to set all the input image's pixels to zero before starting to build the profiles over it (this is done in memory after reading the input, so nothing will happen to your input file).

`-B STR/INT`
`--backhdu=STR/INT`
The header data unit (HDU) of the file given to `--background`.

`-C`
`--clearcanvas`
When an input image is specified (with the `--background` option, set all its pixels to 0.0 immediately after reading it into memory. Effectively, this will allow you to use all its properties (described under the `--background` option), without having to worry about the pixel values.

`--clearcanvas` can come in handy in many situations, for example if you want to create a labeled image (segmentation map) for creating a catalog (see Section 7.4 [MakeCatalog], page 255). In other cases, you might have modeled the objects in an image and want to create them on the same frame, but without the original pixel values.

`-E STR/INT,FLT[,FLT,[...]]`
`--kernel=STR/INT,FLT[,FLT,[...]]`
Only build one kernel profile with the parameters given as the values to this option. The different values must be separated by a comma (`,`). The first value identifies the radial function of the profile, either through a string or through a number (see description of `--fcol` in Section 8.1.5.1 [MakeProfiles catalog], page 292). Each radial profile needs a different total number of parameters: Sérsic and Moffat functions need 3 parameters: radial, Sérsic index or Moffat β, and truncation radius. The Gaussian function needs two parameters: radial and truncation radius. The point function doesn't need any parameters and flat and circumference profiles just need one parameter (truncation radius).

The PSF or kernel is a unique (and highly constrained) type of profile: the sum of its pixels must be one, its center must be the center of the central pixel (in an image with an odd number of pixels on each side), and commonly it is circular, so its axis ratio and position angle are one and zero respectively. Kernels are commonly necessary for various data analysis and data manipulation steps (for example see Section 6.3 [Convolve], page 177, and Section 7.2 [NoiseChisel], page 225. Because of this it is inconvenient to define a catalog with one row

and many zero valued columns (for all the non-necessary parameters). Hence, with this option, it is possible to create a kernel with MakeProfiles without the need to create a catalog. Here are some examples:

`--kernel=moffat,3,2.8,5`
: A Moffat kernel with FWHM of 3 pixels, $\beta = 2.8$ which is truncated at 5 times the FWHM.

`--kernel=gaussian,2,3`
: A Gaussian kernel with FWHM of 2 pixels and truncated at 3 times the FWHM.

`-x INT,INT`
`--mergedsize=INT,INT`
: The number of pixels along each axis of the output, in FITS order. This is before over-sampling. For example if you call MakeProfiles with `--mergedsize=100,150 --oversample=5` (assuming no shift due for later convolution), then the final image size along the first axis will be 500 by 750 pixels. Fractions are acceptable as values for each dimension, however, they must reduce to an integer, so `--mergedsize=150/3,300/3` is acceptable but `--mergedsize=150/4,300/4` is not.

  When viewing a FITS image in DS9, the first FITS dimension is in the horizontal direction and the second is vertical. As an example, the image created with the example above will have 500 pixels horizontally and 750 pixels vertically.

  If a background image is specified, this option is ignored.

`-s INT`
`--oversample=INT`
: The scale to over-sample the profiles and final image. If not an odd number, will be added by one, see Section 8.1.1.6 [Oversampling], page 289. Note that this `--oversample` will remain active even if an input image is specified. If your input catalog is based on the background image, be sure to set `--oversample=1`.

`--psfinimg`
: Build the possibly existing PSF profiles (Moffat or Gaussian) in the catalog into the final image. By default they are built separately so you can convolve your images with them, thus their magnitude and positions are ignored. With this option, they will be built in the final image like every other galaxy profile. To have a final PSF in your image, make a point profile where you want the PSF and after convolution it will be the PSF.

`-i`
`--individual`
: If this option is called, each profile is created in a separate FITS file within the same directory as the output and the row number of the profile (starting from zero) in the name. The file for each row's profile will be in the same directory as the final combined image of all the profiles and will have the final image's name as a suffix. So for example if the final combined image is named `./out/fromcatalog.fits`, then the first profile that will be created with this option will be named `./out/0_fromcatalog.fits`.

Since each image only has one full profile out to the truncation radius the profile is centered and so, only the sub-pixel position of the profile center is important for the outputs of this option. The output will have an odd number of pixels. If there is no oversampling, the central pixel will contain the profile center. If the value to `--oversample` is larger than unity, then the profile center is on any of the central `--oversample`'d pixels depending on the fractional value of the profile center.

If the fractional value is larger than half, it is on the bottom half of the central region. This is due to the FITS definition of a real number position: The center of a pixel has fractional value 0.00 so each pixel contains these fractions: .5 – .75 – .00 (pixel center) – .25 – .5.

`-m`
`--nomerged`
Don't make a merged image. By default after making the profiles, they are added to a final image with side lengths specified by `--mergedsize` if they overlap with it.

The options below can be used to define the world coordinate system (WCS) properties of the MakeProfiles outputs. The option names are deliberately chosen to be the same as the FITS standard WCS keywords. See Section 8 of Pence et al [2010] (`https://doi.org/10.1051/0004-6361/201015362`) for a short introduction to WCS in the FITS standard[11].

If you look into the headers of a FITS image with WCS for example you will see all these names but in uppercase and with numbers to represent the dimensions, for example `CRPIX1` and `PC2_1`. You can see the FITS headers with Gnuastro's Section 5.1 [Fits], page 128, program using a command like this: `$ astfits -p image.fits`.

If the values given to any of these options does not correspond to the number of dimensions in the output dataset, then no WCS information will be added.

`--crpix=FLT,FLT`
The pixel coordinates of the WCS reference point. Fractions are acceptable for the values of this option.

`--crval=FLT,FLT`
The WCS coordinates of the Reference point. Fractions are acceptable for the values of this option.

`--cdelt=FLT,FLT`
The resolution (size of one data-unit or pixel in WCS units) of the non-oversampled dataset. Fractions are acceptable for the values of this option.

[11] The world coordinate standard in FITS is a very beautiful and powerful concept to link/associate datasets with the outside world (other datasets). The description in the FITS standard (link above) only touches the tip of the ice-burg. To learn more please see Greisen and Calabretta [2002] (`https://doi.org/10.1051/0004-6361:20021326`), Calabretta and Greisen [2002] (`https://doi.org/10.1051/0004-6361:20021327`), Greisen et al. [2006] (`https://doi.org/10.1051/0004-6361:20053818`), and Calabretta et al. (`http://www.atnf.csiro.au/people/mcalabre/WCS/dcs_20040422.pdf`)

`--pc=FLT,FLT,FLT,FLT`
: The PC matrix of the WCS rotation, see the FITS standard (link above) to better understand the PC matrix.

`--cunit=STR,STR`
: The units of each WCS axis, for example `deg`. Note that these values are part of the FITS standard (link above). MakeProfiles won't complain if you use non-standard values, but later usage of them might cause trouble.

`--ctype=STR,STR`
: The type of each WCS axis, for example `RA---TAN` and `DEC--TAN`. Note that these values are part of the FITS standard (link above). MakeProfiles won't complain if you use non-standard values, but later usage of them might cause trouble.

8.1.5.4 MakeProfiles log file

Besides the final merged dataset of all the profiles, or the individual datasets (see Section 8.1.5.3 [MakeProfiles output dataset], page 297), if the `--log` option is called MakeProfiles will also create a log file in the current directory (where you run MockProfiles). See Section 4.1.2 [Common options], page 95, for a full description of `--log` and other options that are shared between all Gnuastro programs. The values for each column are explained in the first few commented lines of the log file (starting with `#` character). Here is a more complete description.

- An ID (row number of profile in input catalog).
- The total magnitude of the profile in the output dataset. When the profile does not completely overlap with the output dataset, this will be different from your input magnitude.
- The number of pixels (in the oversampled image) which used Monte Carlo integration and not the central pixel value, see Section 8.1.1.5 [Sampling from a function], page 288.
- The fraction of flux in the Monte Carlo integrated pixels.
- If an individual image was created, this column will have a value of `1`, otherwise it will have a value of `0`.

8.2 MakeNoise

Real data are always buried in noise, therefore to finalize a simulation of real data (for example to test our observational algorithms) it is essential to add noise to the mock profiles created with MakeProfiles, see Section 8.1 [MakeProfiles], page 284. Below, the general principles and concepts to help understand how noise is quantified is discussed. MakeNoise options and argument are then discussed in Section 8.2.2 [Invoking MakeNoise], page 305.

8.2.1 Noise basics

Deep astronomical images, like those used in extragalactic studies, seriously suffer from noise in the data. Generally speaking, the sources of noise in an astronomical image are photon counting noise and Instrumental noise which are discussed in Section 8.2.1.1 [Photon counting noise], page 302, and Section 8.2.1.2 [Instrumental noise], page 303. This review finishes with Section 8.2.1.4 [Generating random numbers], page 304, which is a short

introduction on how random numbers are generated. We will see that while software random number generators are not perfect, they allow us to obtain a reproducible series of random numbers through setting the random number generator function and seed value. Therefore in this section, we'll also discuss how you can set these two parameters in Gnuastro's programs (including MakeNoise).

8.2.1.1 Photon counting noise

With the very accurate electronics used in today's detectors, photon counting noise[12] is the most significant source of uncertainty in most datasets. To understand this noise (error in counting), we need to take a closer look at how a distribution produced by counting can be modeled as a parametric function.

Counting is an inherently discrete operation, which can only produce positive (including zero) integer outputs. For example we can't count 3.2 or -2 of anything. We only count 0, 1, 2, 3 and so on. The distribution of values, as a result of counting efforts is formally known as the Poisson distribution (`https://en.wikipedia.org/wiki/Poisson_distribution`). It is associated to Siméon Denis Poisson, because he discussed it while working on the number of wrongful convictions in court cases in his 1837 book[13].

Let's take λ to represent the expected mean count of something. Furthermore, let's take k to represent the result of one particular counting attempt. The probability density function of getting k counts (in each attempt, given the expected/mean count of λ) can be written as:

$$f(k) = \frac{\lambda^k}{k!}e^{-\lambda}, \quad k \in \{0, 1, 2, 3, \ldots\}$$

Because the Poisson distribution is only applicable to positive values (note the factorial operator, which only applies to non-negative integers), naturally it is very skewed when λ is near zero. One qualitative way to understand this behavior is that there simply aren't enough integers smaller than λ, than integers that are larger than it. Therefore to accommodate all possibilities/counts, it has to be strongly skewed when λ is small.

As λ becomes larger, the distribution becomes more and more symmetric. A very useful property of the Poisson distribution is that the mean value is also its variance. When λ is very large, say $\lambda > 1000$, then the Normal (Gaussian) distribution (`https://en.wikipedia.org/wiki/Normal_distribution`), is an excellent approximation of the Poisson distribution with mean $\mu = \lambda$ and standard deviation $\sigma = \sqrt{\lambda}$. In other words, a Poisson distribution (with a sufficiently large λ) is simply a Gaussian that only has one free parameter ($\mu = \lambda$ and $\sigma = \sqrt{\lambda}$), instead of the two parameters (independent μ and σ) that it originally has.

In real situations, the photons/flux from our targets are added to a certain background flux (observationally, the *Sky* value). The Sky value is defined to be the average flux of a region in the dataset with no targets. Its physical origin can be the brightness of

[12] In practice, we are actually counting the electrons that are produced by each photon, not the actual photons.

[13] [From Wikipedia] Poisson's result was also derived in a previous study by Abraham de Moivre in 1711. Therefore some people suggest it should rightly be called the de Moivre distribution.

the atmosphere (for ground-based instruments), possible stray light within the imaging instrument, the average flux of undetected targets, or etc. The Sky value is thus an ideal definition, because in real datasets, what lies deep in the noise (far lower than the detection limit) is never known[14]. To account for all of these, the sky value is defined to be the average count/value of the undetected regions in the image. In a mock image/dataset, we have the luxury of setting the background (Sky) value.

In each element of the dataset (pixel in an image), the flux is the sum of contributions from various sources (after convolution by the PSF, see Section 8.1.1.2 [Point spread function], page 285). Let's name the convolved sum of possibly overlapping objects, I_{nn}. nn representing 'no noise'. For now, let's assume the background (B) is constant and sufficiently high for the Poisson distribution to be approximated by a Gaussian. Then the flux after adding noise is a random value taken from a Gaussian distribution with the following mean (μ) and standard deviation (σ):

$$\mu = B + I_{nn}, \quad \sigma = \sqrt{B + I_{nn}}$$

Since this type of noise is inherent in the objects we study, it is usually measured on the same scale as the astronomical objects, namely the magnitude system, see Section 8.1.3 [Flux Brightness and magnitude], page 290. It is then internally converted to the flux scale for further processing.

8.2.1.2 Instrumental noise

While taking images with a camera, a dark current is fed to the pixels, the variation of the value of this dark current over the pixels, also adds to the final image noise. Another source of noise is the readout noise that is produced by the electronics in the detector. Specifically, the parts that attempt to digitize the voltage produced by the photo-electrons in the analog to digital converter. With the current generation of instruments, this source of noise is not as significant as the noise due to the background Sky discussed in Section 8.2.1.1 [Photon counting noise], page 302.

Let C represent the combined standard deviation of all these instrumental sources of noise. When only this source of noise is present, the noised pixel value would be a random value chosen from a Gaussian distribution with

$$\mu = I_{nn}, \quad \sigma = \sqrt{C^2 + I_{nn}}$$

This type of noise is independent of the signal in the dataset, it is only determined by the instrument. So the flux scale (and not magnitude scale) is most commonly used for this type of noise. In practice, this value is usually reported in analog-to-digital units or ADUs, not flux or electron counts. The gain value of the device can be used to convert between these two, see Section 8.1.3 [Flux Brightness and magnitude], page 290.

[14] In a real image, a relatively large number of very faint objects can been fully buried in the noise and never detected. These undetected objects will bias the background measurement to slightly larger values. Our best approximation is thus to simply assume they are uniform, and consider their average effect. See Figure 1 (a.1 and a.2) and Section 2.2 in Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`).

8.2.1.3 Final noised pixel value

Based on the discussions in Section 8.2.1.1 [Photon counting noise], page 302, and Section 8.2.1.2 [Instrumental noise], page 303, depending on the values you specify for B and C from the above, the final noised value for each pixel is a random value chosen from a Gaussian distribution with

$$\mu = B + I_{nn}, \quad \sigma = \sqrt{C^2 + B + I_{nn}}$$

8.2.1.4 Generating random numbers

As discussed above, to generate noise we need to make random samples of a particular distribution. So it is important to understand some general concepts regarding the generation of random numbers. For a very complete and nice introduction we strongly advise reading Donald Knuth's "The art of computer programming", volume 2, chapter 3[15]. Quoting from the GNU Scientific Library manual, "If you don't own it, you should stop reading right now, run to the nearest bookstore, and buy it"[16]!

Using only software, we can only produce what is called a psuedo-random sequence of numbers. A true random number generator is a hardware (let's assume we have made sure it has no systematic biases), for example throwing dice or flipping coins (which have remained from the ancient times). More modern hardware methods use atmospheric noise, thermal noise or other types of external electromagnetic or quantum phenomena. All pseudo-random number generators (software) require a seed to be the basis of the generation. The advantage of having a seed is that if you specify the same seed for multiple runs, you will get an identical sequence of random numbers which allows you to reproduce the same final noised image.

The programs in GNU Astronomy Utilities (for example MakeNoise or MakeProfiles) use the GNU Scientific Library (GSL) to generate random numbers. GSL allows the user to set the random number generator through environment variables, see Section 3.3.1.2 [Installation directory], page 79, for an introduction to environment variables. In the chapter titled "Random Number Generation" they have fully explained the various random number generators that are available (there are a lot of them!). Through the two environment variables `GSL_RNG_TYPE` and `GSL_RNG_SEED` you can specify the generator and its seed respectively.

If you don't specify a value for `GSL_RNG_TYPE`, GSL will use its default random number generator type. The default type is sufficient for most general applications. If no value is given for the `GSL_RNG_SEED` environment variable and you have asked Gnuastro to read the seed from the environment (through the `--envseed` option), then GSL will use the default value of each generator to give identical outputs. If you don't explicitly tell Gnuastro programs to read the seed value from the environment variable, then they will use the system time (accurate to within a microsecond) to generate (apparently random) seeds. In this manner, every time you run the program, you will get a different random number distribution.

There are two ways you can specify values for these environment variables. You can call them on the same command-line for example:

```
$ GSL_RNG_TYPE="taus" GSL_RNG_SEED=345 astmknoise input.fits
```

[15] Knuth, Donald. 1998. The art of computer programming. Addison–Wesley. ISBN 0-201-89684-2

[16] For students, running to the library might be more affordable!

In this manner the values will only be used for this particular execution of MakeNoise. Alternatively, you can define them for the full period of your terminal session or script length, using the shell's `export` command with the two separate commands below (for a script remove the `$` signs):

```
$ export GSL_RNG_TYPE="taus"
$ export GSL_RNG_SEED=345
```

The subsequent programs which use GSL's random number generators will hence forth use these values in this session of the terminal you are running or while executing this script. In case you want to set fixed values for these parameters every time you use the GSL random number generator, you can add these two lines to your `.bashrc` startup script[17], see Section 3.3.1.2 [Installation directory], page 79.

NOTE: If the two environment variables `GSL_RNG_TYPE` and `GSL_RNG_SEED` are defined, GSL will report them by default, even if you don't use the `--envseed` option. For example you can see the top few lines of the output of MakeProfiles:

```
$ export GSL_RNG_TYPE="taus"
$ export GSL_RNG_SEED=345
$ astmkprof -s1 --kernel=gaussian,2,5 --envseed
GSL_RNG_TYPE=taus
GSL_RNG_SEED=345
MakeProfiles A.B started on DDD MMM DD HH:MM:SS YYYY
  - Building one gaussian kernel
  - Random number generator (RNG) type: ranlxs1
  - RNG seed for all profiles: 345
  ---- ./kernel.fits created.
MakeProfiles finished in 0.111271 seconds
```

The first two output lines (showing the names of the environment variables) are printed by GSL before MakeProfiles actually starts generating random numbers. The Gnuastro programs will report the values they use independently, you should check them for the final values used. For example if `--envseed` is not given, `GSL_RNG_SEED` will not be used and the last line shown above will not be printed. In the case of MakeProfiles, each profile will get its own seed value.

8.2.2 Invoking MakeNoise

MakeNoise will add noise to an existing image. The executable name is `astmknoise` with the following general template

```
$ astmknoise [OPTION ...] InputImage.fits
```

One line examples:

```
## Add noise with a standard deviation of 100 to image:
$ astmknoise --sigma=100 image.fits
```

[17] Don't forget that if you are going to give your scripts (that use the GSL random number generator) to others you have to make sure you also tell them to set these environment variable separately. So for scripts, it is best to keep all such variable definitions within the script, even if they are within your `.bashrc`.

```
## Add noise to input image assuming a background magnitude (with
## zeropoint magnitude of 0) and a certain instrumental noise:
$ astmknoise --background=-10 -z0 --instrumental=20 mockimage.fits
```

If actual processing is to be done, the input image is a mandatory argument. The full list of options common to all the programs in Gnuastro can be seen in Section 4.1.2 [Common options], page 95. The type (see Section 4.5 [Numeric data types], page 115) of the output can be specified with the `--type` option, see Section 4.1.2.1 [Input/Output options], page 95. The header of the output FITS file keeps all the parameters that were influential in making it. This is done for future reproducibility.

`-s FLT`

`--sigma=FLT`
: The total noise sigma in the same units as the pixel values. With this option, the `--background`, `--zeropoint` and `--instrumental` will be ignored. With this option, the noise will be independent of the pixel values (which is not realistic, see Section 8.2.1.1 [Photon counting noise], page 302). Hence it is only useful if you are working on low surface brightness regions where the change in pixel value (and thus real noise) is insignificant.

`-b FLT`
`--background=FLT`
: The background pixel value for the image in units of magnitudes, see Section 8.2.1.1 [Photon counting noise], page 302, and Section 8.1.3 [Flux Brightness and magnitude], page 290.

`-z FLT`
`--zeropoint=FLT`
: The zeropoint magnitude used to convert the value of `--background` (in units of magnitude) to flux, see Section 8.1.3 [Flux Brightness and magnitude], page 290.

`-i FLT`
`--instrumental=FLT`
: The instrumental noise which is in units of flux, see Section 8.2.1.2 [Instrumental noise], page 303.

`-e`
`--envseed`
: Use the `GSL_RNG_SEED` environment variable for the seed used in the random number generator, see Section 8.2.1.4 [Generating random numbers], page 304. With this option, the output image noise is always going to be identical (or reproducible).

`-d`
`--doubletype`
: Save the output in the double precision floating point format that was used internally. This option will be most useful if the input images were of integer types.

9 High-level calculations

After the reduction of raw data (for example with the programs in Chapter 6 [Data manipulation], page 151) you will have reduced images/data ready for processing/analyzing (for example with the programs in Chapter 7 [Data analysis], page 208). But the processed/analyzed data (or catalogs) are still not enough to derive any scientific result. Even higher-level analysis is still needed to convert the observed magnitudes, sizes or volumes into physical quantities that we associate with each catalog entry or detected object which is the purpose of the tools in this section.

9.1 CosmicCalculator

To derive higher-level information regarding our sources in extra-galactic astronomy, cosmological calculations are necessary. In Gnuastro, CosmicCalculator is in charge of such calculations. Before discussing how CosmicCalculator is called and operates (in Section 9.1.3 [Invoking CosmicCalculator], page 312), it is important to provide a rough but mostly self sufficient review of the basics and the equations used in the analysis. In Section 9.1.1 [Distance on a 2D curved space], page 307, the basic idea of understanding distances in a curved and expanding 2D universe (which we can visualize) are reviewed. Having solidified the concepts there, in Section 9.1.2 [Extending distance concepts to 3D], page 312, the formalism is extended to the 3D universe we are trying to study in our research.

The focus here is obtaining a physical insight into these equations (mainly for the use in real observational studies). There are many books thoroughly deriving and proving all the equations with all possible initial conditions and assumptions for any abstract universe, interested readers can study those books.

9.1.1 Distance on a 2D curved space

The observations to date (for example the Planck 2015 results), have not measured[1] the presence of significant curvature in the universe. However to be generic (and allow its measurement if it does in fact exist), it is very important to create a framework that allows non-zero uniform curvature. However, this section is not intended to be a fully thorough and mathematically complete derivation of these concepts. There are many references available for such reviews that go deep into the abstract mathematical proofs. The emphasis here is on visualization of the concepts for a beginner.

As 3D beings, it is difficult for us to mentally create (visualize) a picture of the curvature of a 3D volume. Hence, here we will assume a 2D surface/space and discuss distances on that 2D surface when it is flat and when it is curved. Once the concepts have been created/visualized here, we will extend them, in Section 9.1.2 [Extending distance concepts to 3D], page 312, to a real 3D spatial *slice* of the Universe we live in and hope to study.

To be more understandable (actively discuss from an observer's point of view) let's assume there's an imaginary 2D creature living on the 2D space (which *might* be curved in 3D). Here, we will be working with this creature in its efforts to analyze distances in its 2D universe. The start of the analysis might seem too mundane, but since it is difficult to

[1] The observations are interpreted under the assumption of uniform curvature. For a relativistic alternative to dark energy (and maybe also some part of dark matter), non-uniform curvature may be even be more critical, but that is beyond the scope of this brief explanation.

imagine a 3D curved space, it is important to review all the very basic concepts thoroughly for an easy transition to a universe that is more difficult to visualize (a curved 3D space embedded in 4D).

To start, let's assume a static (not expanding or shrinking), flat 2D surface similar to Figure 9.1 and that the 2D creature is observing its universe from point A. One of the most basic ways to parameterize this space is through the Cartesian coordinates (x, y). In Figure 9.1, the basic axes of these two coordinates are plotted. An infinitesimal change in the direction of each axis is written as dx and dy. For each point, the infinitesimal changes are parallel with the respective axes and are not shown for clarity. Another very useful way of parameterizing this space is through polar coordinates. For each point, we define a radius (r) and angle (ϕ) from a fixed (but arbitrary) reference axis. In Figure 9.1 the infinitesimal changes for each polar coordinate are plotted for a random point and a dashed circle is shown for all points with the same radius.

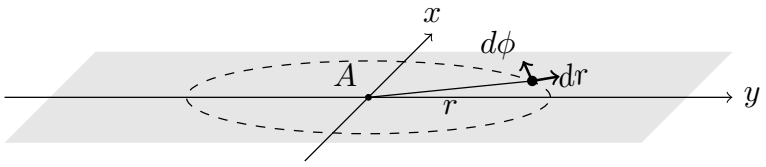


Figure 9.1: Two dimensional Cartesian and polar coordinates on a flat plane.

Assuming an object is placed at a certain position, which can be parameterized as (x, y), or (r, ϕ), a general infinitesimal change in its position will place it in the coordinates $(x+dx, y+dy)$ and $(r+dr, \phi+d\phi)$. The distance (on the flat 2D surface) that is covered by this infinitesimal change in the static universe (ds_s, the subscript signifies the static nature of this universe) can be written as:

$$ds_s = dx^2 + dy^2 = dr^2 + r^2 d\phi^2$$

The main question is this: how can the 2D creature incorporate the (possible) curvature in its universe when it's calculating distances? The universe that it lives in might equally be a curved surface like Figure 9.2. The answer to this question but for a 3D being (us) is the whole purpose to this discussion. Here, we want to give the 2D creature (and later, ourselves) the tools to measure distances if the space (that hosts the objects) is curved.

Figure 9.2 assumes a spherical shell with radius R as the curved 2D plane for simplicity. The 2D plane is tangent to the spherical shell and only touches it at A. This idea will be generalized later. The first step in measuring the distance in a curved space is to imagine a third dimension along the z axis as shown in Figure 9.2. For simplicity, the z axis is assumed to pass through the center of the spherical shell. Our imaginary 2D creature cannot visualize the third dimension or a curved 2D surface within it, so the remainder of this discussion is purely abstract for it (similar to us having difficulty in visualizing a 3D curved space in 4D). But since we are 3D creatures, we have the advantage of visualizing the following steps. Fortunately the 2D creature is already familiar with our mathematical constructs, so it can follow our reasoning.

With the third axis added, a generic infinitesimal change over *the full* 3D space corresponds to the distance:

$$ds_s^2 = dx^2 + dy^2 + dz^2 = dr^2 + r^2 d\phi^2 + dz^2.$$

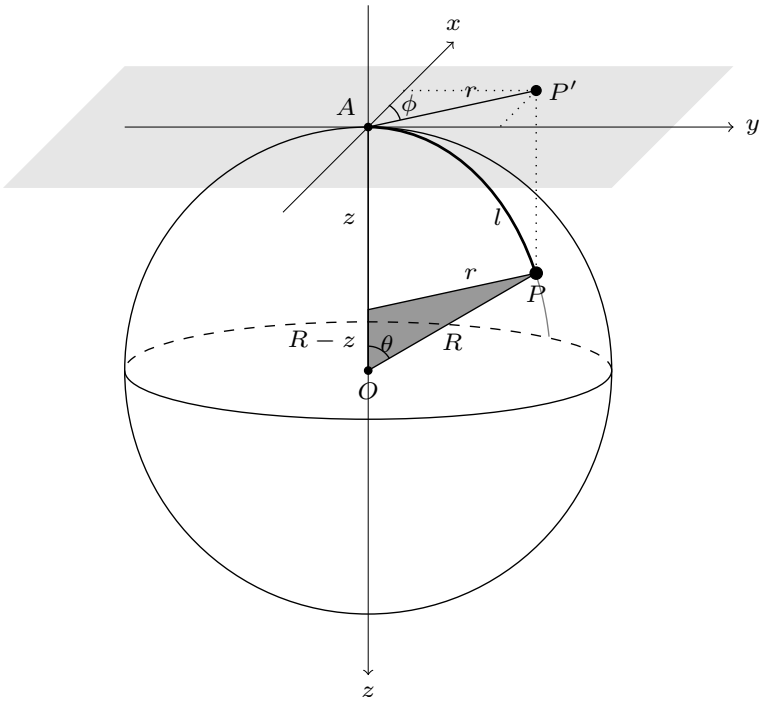


Figure 9.2: 2D spherical shell (centered on O) and flat plane (light gray) tangent to it at point A.

It is very important to recognize that this change of distance is for *any* point in the 3D space, not just those changes that occur on the 2D spherical shell of Figure 9.2. Recall that our 2D friend can only do measurements on the 2D surfaces, not the full 3D space. So we have to constrain this general change to any change on the 2D spherical shell. To do that, let's look at the arbitrary point P on the 2D spherical shell. Its image (P') on the flat plain is also displayed. From the dark gray triangle, we see that

$$\sin\theta = \frac{r}{R}, \quad \cos\theta = \frac{R-z}{R}.$$

These relations allow the 2D creature to find the value of z (an abstract dimension for it) as a function of r (distance on a flat 2D plane, which it can visualize) and thus eliminate z. From $\sin^2\theta + \cos^2\theta = 1$, we get $z^2 - 2Rz + r^2 = 0$ and solving for z, we find:

$$z = R\left(1 \pm \sqrt{1 - \frac{r^2}{R^2}}\right).$$

The $\pm$ can be understood from Figure 9.2: For each r, there are two points on the sphere, one in the upper hemisphere and one in the lower hemisphere. An infinitesimal change in r, will create the following infinitesimal change in z:

$$dz = \frac{\mp r}{R}\left(\frac{1}{\sqrt{1-r^2/R^2}}\right)dr.$$

Using the positive signed equation instead of dz in the ds_s^2 equation above, we get:

$$ds_s^2 = \frac{dr^2}{1-r^2/R^2} + r^2 d\phi^2.$$

The derivation above was done for a spherical shell of radius R as a curved 2D surface. To generalize it to any surface, we can define $K = 1/R^2$ as the curvature parameter. Then the general infinitesimal change in a static universe can be written as:

$$ds_s^2 = \frac{dr^2}{1-Kr^2} + r^2 d\phi^2.$$

Therefore, when $K > 0$ (and curvature is the same everywhere), we have a finite universe, where r cannot become larger than R as in Figure 9.2. When $K = 0$, we have a flat plane (Figure 9.1) and a negative K will correspond to an imaginary R. The latter two cases may be infinite in area (which is not a simple concept, but mathematically can be modeled with r extending infinitely), or finite-area (like a cylinder is flat everywhere with $ds_s^2 = dx^2 + dy^2$, but finite in one direction in size).

A very important issue that can be discussed now (while we are still in 2D and can actually visualize things) is that $\overrightarrow{r}$ is tangent to the curved space at the observer's position. In other words, it is on the gray flat surface of Figure 9.2, even when the universe if curved: $\overrightarrow{r} = P' - A$. Therefore for the point P on a curved space, the raw coordinate r is the distance to P', not P. The distance to the point P (at a specific coordinate r on the flat plane) over the curved surface (thick line in Figure 9.2) is called the *proper distance* and is displayed with l. For the specific example of Figure 9.2, the proper distance can be calculated with: $l = R\theta$ (θ is in radians). using the $\sin\theta$ relation found above, we can find l as a function of r:

$$\theta = \sin^{-1}\left(\frac{r}{R}\right) \quad \rightarrow \quad l(r) = R\sin^{-1}\left(\frac{r}{R}\right)$$

R is just an arbitrary constant and can be directly found from K, so for cleaner equations, it is common practice to set $R = 1$, which gives: $l(r) = \sin^{-1} r$. Also note that when $R = 1$, then $l = \theta$. Generally, depending on the the curvature, in a *static* universe the proper distance can be written as a function of the coordinate r as (from now on we are assuming $R = 1$):

$$l(r) = \sin^{-1}(r) \quad (K > 0), \qquad l(r) = r \quad (K = 0), \qquad l(r) = \sinh^{-1}(r) \quad (K < 0).$$

With l, the infinitesimal change of distance can be written in a more simpler and abstract form of

$$ds_s^2 = dl^2 + r^2 d\phi^2.$$

Until now, we had assumed a static universe (not changing with time). But our observations so far appear to indicate that the universe is expanding (it isn't static). Since there is no reason to expect the observed expansion is unique to our particular position of the universe, we expect the universe to be expanding at all points with the same rate at the same time. Therefore, to add a time dependence to our distance measurements, we can include a multiplicative scaling factor, which is a function of time: $a(t)$. The functional form of $a(t)$ comes from the cosmology, the physics we assume for it: general relativity, and the choice of whether the universe is uniform ('homogeneous') in density and curvature or inhomogeneous. In this section, the functional form of $a(t)$ is irrelevant, so we can avoid these issues.

With this scaling factor, the proper distance will also depend on time. As the universe expands, the distance between two given points will shift to larger values. We thus define a distance measure, or coordinate, that is independent of time and thus doesn't 'move'. We call it the *comoving distance* and display with χ such that: $l(r,t) = \chi(r)a(t)$. We have therefore, shifted the r dependence of the proper distance we derived above for a static universe to the comoving distance:

$$\chi(r) = \sin^{-1}(r) \quad (K > 0), \qquad \chi(r) = r \quad (K = 0), \qquad \chi(r) = \sinh^{-1}(r) \quad (K < 0).$$

Therefore, $\chi(r)$ is the proper distance to an object at a specific reference time: $t = t_r$ (the r subscript signifies "reference") when $a(t_r) = 1$. At any arbitrary moment ($t \neq t_r$) before or after t_r, the proper distance to the object can be scaled with $a(t)$.

Measuring the change of distance in a time-dependent (expanding) universe only makes sense if we can add up space and time[2]. But we can only add bits of space and time together if we measure them in the same units: with a conversion constant (similar to how 1000 is used to convert a kilometer into meters). Experimentally, we find strong support for the hypothesis that this conversion constant is the speed of light (or gravitational waves[3]) in a vacuum. This speed is postulated to be constant[4] and is almost always written as c. We can thus parameterize the change in distance on an expanding 2D surface as

$$ds^2 = c^2 dt^2 - a^2(t) ds_s^2 = c^2 dt^2 - a^2(t)(d\chi^2 + r^2 d\phi^2).$$

[2] In other words, making our space-time consistent with Minkowski space-time geometry. In this geometry, different observers at a given point (event) in space-time split up space-time into 'space' and 'time' in different ways, just like people at the same spatial position can make different choices of splitting up a map into 'left–right' and 'up–down'. This model is well supported by twentieth and twenty-first century observations.

[3] The speed of gravitational waves was recently found to be very similar to that of light in vacuum, see arXiv:1710.05834 (`https://arxiv.org/abs/1710.05834`).

[4] In *natural units*, speed is measured in units of the speed of light in vacuum.

9.1.2 Extending distance concepts to 3D

The concepts of Section 9.1.1 [Distance on a 2D curved space], page 307, are here extended to a 3D space that *might* be curved. We can start with the generic infinitesimal distance in a static 3D universe, but this time in spherical coordinates instead of polar coordinates. θ is shown in Figure 9.2, but here we are 3D beings, positioned on O (the center of the sphere) and the point O is tangent to a 4D-sphere. In our 3D space, a generic infinitesimal displacement will correspond to the following distance in spherical coordinates:

$$ds_s^2 = dx^2 + dy^2 + dz^2 = dr^2 + r^2(d\theta^2 + \sin^2\theta d\phi^2).$$

Like the 2D creature before, we now have to assume an abstract dimension which we cannot visualize easily. Let's call the fourth dimension w, then the general change in coordinates in the *full* four dimensional space will be:

$$ds_s^2 = dr^2 + r^2(d\theta^2 + \sin^2\theta d\phi^2) + dw^2.$$

But we can only work on a 3D curved space, so following exactly the same steps and conventions as our 2D friend, we arrive at:

$$ds_s^2 = \frac{dr^2}{1 - Kr^2} + r^2(d\theta^2 + \sin^2\theta d\phi^2).$$

In a non-static universe (with a scale factor a(t)), the distance can be written as:

$$ds^2 = c^2dt^2 - a^2(t)[d\chi^2 + r^2(d\theta^2 + \sin^2\theta d\phi^2)].$$

9.1.3 Invoking CosmicCalculator

CosmicCalculator will calculate cosmological variables based on the input parameters. The executable name is `astcosmiccal` with the following general template

```
$ astcosmiccal [OPTION...] ...
```

One line examples:

```
## Print basic cosmological properties at redshift 2.5:
$ astcosmiccal -z2.5

## Only print Comoving volume over 4pi stradian to z (Mpc^3):
$ astcosmiccal --redshift=0.8 --volume

## Print luminosity distance, angular diameter distance and age
## of universe in one row at redshift 0.4
$ astcosmiccal -z0.4 -LAg

## Assume Lambda and matter density of 0.7 and 0.3 and print
```

```
## basic cosmological parameters for redshift 2.1:
$ astcosmiccal -l0.7 -m0.3 -z2.1
```

The input parameters (for example current matter density and etc) can be given as command-line options or in the configuration files, see Section 4.2 [Configuration files], page 106. For a definition of the different parameters, please see the sections prior to this. If no redshift is given, CosmicCalculator will just print its input parameters and abort. For a full list of the input options, please see Section 9.1.3.1 [CosmicCalculator input options], page 313.

When only a redshift is given, CosmicCalculator will print all calculations (one per line) with some explanations before each. This can be good when you want a general feeling of the conditions at a specific redshift. Alternatively, if any specific calculations are requested, only the requested values will be calculated and printed with one character space between them. In this case, no description will be printed. See Section 9.1.3.2 [CosmicCalculator specific calculations], page 313, for the full list of these options along with some explanations how when/how they can be useful.

9.1.3.1 CosmicCalculator input options

The inputs to CosmicCalculator can be specified with the following options:

`-z FLT`
`--redshift=FLT`
: The redshift of interest.

`-H FLT`
`--H0=FLT`
: Current expansion rate (in km $\sec^{-1}$ Mpc^{-1}).

`-l FLT`
`--olambda=FLT`
: Cosmological constant density divided by the critical density in the current Universe ($\Omega_{\Lambda,0}$).

`-m FLT`
`--omatter=FLT`
: Matter (including massive neutrinos) density divided by the critical density in the current Universe ($\Omega_{m,0}$).

`-r FLT`
`--oradiation=FLT`
: Radiation density divided by the critical density in the current Universe ($\Omega_{r,0}$).

9.1.3.2 CosmicCalculator specific calculations

By default, when no specific calculations are requested, CosmicCalculator will print a complete set of all its calculators (one line for each calculation, see Section 9.1.3 [Invoking CosmicCalculator], page 312). The full list of calculations can be useful when you don't want any specific value, but just a general view. In other contexts (for example in a batch script or during a discussion), you know exactly what you want and don't want to be distracted by all the extra information.

You can use any number of the options described below in any order. When any of these options are requested, CosmicCalculator's output will just be a single line with a single space

between the (possibly) multiple values. In the example below, only the tangential distance along one arcsecond (in kpc), absolute magnitude conversion, and age of the universe at redshift 2 are printed (recall that you can merge short options together, see Section 4.1.1.2 [Options], page 93).

```
$ astcosmiccal -z2 -sag
8.585046 44.819248 3.289979
```

Here is one example of using this feature in scripts: by adding the following two lines in a script to keep/use the comoving volume with varying redshifts:

```
z=3.12
vol=$(astcosmiccal --redshift=$z --volume)
```

In a script, this operation might be necessary for a large number of objects (several of galaxies in a catalog for example). So the fact that all the other default calculations are ignored will also help you get to your result faster.

If you are indeed dealing with many (for example thousands) of redshifts, using CosmicCalculator is not the best/fastest solution. Because it has to go through all the configuration files and preparations for each invocation. To get the best efficiency (least overhead), we recommend using Gnuastro's cosmology library (see Section 11.3.27 [Cosmology library (`cosmology.h`)], page 436). CosmicCalculator also calls the library functions defined there for its calculations, so you get the same result with no overhead. Gnuastro also has libraries for easily reading tables into a C program, see Section 11.3.10 [Table input output (`table.h`)], page 370. Afterwards, you can easily build and run your C program for the particular processing with Section 11.2 [BuildProgram], page 328.

If you just want to inspect the value of a variable visually, the description (which comes with units) might be more useful. In such cases, the following command might be better. The other calculations will also be done, but they are so fast that you will not notice on modern computers (the time it takes your eye to focus on the result is usually longer than the processing: a fraction of a second).

```
$ astcosmiccal --redshift=0.832 | grep volume
```

The full list of CosmicCalculator's specific calculations is present below. In case you have forgot the units, you can use the `--help` option which has the units along with a short description.

`-G`
`--agenow` The current age of the universe (given the input parameters) in Ga (Giga annum, or billion years).

`-C`
`--criticaldensitynow`
The current critical density (given the input parameters) in grams per centimeter-cube (g/cm^3).

`-d`
`--properdistance`
The proper distance (at current time) to object at the given redshift in Megaparsecs (Mpc). See Section 9.1.1 [Distance on a 2D curved space], page 307, for a description of the proper distance.

`-A`
`--angulardimdist`
: The angular diameter distance to object at given redshift in Megaparsecs (Mpc).

`-s`
`--arcsectandist`
: The tangential distance covered by 1 arcseconds at the given redshift in kiloparsecs (Kpc). This can be useful when trying to estimate the resolution or pixel scale of an instrument (usually in units of arcseconds) at a given redshift.

`-L`
`--luminositydist`
: The luminosity distance to object at given redshift in Megaparsecs (Mpc).

`-u`
`--distancemodulus`
: The distance modulus at given redshift.

`-a`
`--absmagconv`
: The conversion factor (addition) to absolute magnitude. Note that this is practically the distance modulus added with $-2.5\log(1+z)$ for the the desired redshift based on the input parameters. Once the apparent magnitude and redshift of an object is known, this value may be added with the apparent magnitude to give the object's absolute magnitude.

`-g`
`--age`
: Age of the universe at given redshift in Ga (Giga annum, or billion years).

`-b`
`--lookbacktime`
: The look-back time to given redshift in Ga (Giga annum, or billion years). The look-back time at a given redshift is defined as the current age of the universe (`--agenow`) subtracted by the age of the universe at the given redshift.

`-c`
`--criticaldensity`
: The critical density at given redshift in grams per centimeter-cube (g/cm^3).

`-v`
`--onlyvolume`
: The comoving volume in Megaparsecs cube (Mpc^3) until the desired redshift based on the input parameters.

10 Installed scripts

Gnuastro's programs (introduced in previous chapters) are designed to be highly modular and thus mainly contain lower-level operations on the data. However, in many contexts, higher-level operations (for example a sequence of calls to multiple Gnuastro programs, or a special way of running a program and using the outputs) are also very similar between various projects.

To facilitate data analysis on these higher-level steps also, Gnuastro also installs some scripts on your system with the (`astscript-`) prefix (in contrast to the other programs that only have the `ast` prefix). In this chapter, these scripts and their usage details are described.

Like all of Gnuastro's source code, these scripts are also heavily commented. They are written in GNU Bash, which doesn't need compilation (so they are actually human-readable), and is the same language that is mainly used when typing on the command-line. Because of these factors, Bash is much more widely known and used than C (the language of other Gnuastro programs). They also do higher-level operations, so customizing these scripts for a special project will be more common than the programs. You can always inspect them (to customize, check, or educate your self) with this command (just replace `emacs` with your favorite text editor):

```
$ emacs $(which astscript-NAME)
```

These scripts also accept options and are in many ways similar to the programs (see Section 4.1.2 [Common options], page 95) with some minor differences:

- Currently they don't accept configuration files themselves. However, the configuration files of the Gnuastro programs they call are indeed parsed and used by those programs.

  As a result, they don't have the following options: `--checkconfig`, `--config`, `--lastconfig`, `--onlyversion`, `--printparams`, `--setdirconf` and `--setusrconf`.
- They don't directly allocate any memory, so there is no `--minmapsize`.
- They don't have an independent `--usage` option: when called with `--usage`, they just recommend running `--help`.
- The output of `--help` is not configurable like the programs (see Section 4.3.2 [`--help`], page 110).

10.1 Sort FITS files by night

FITS images usually contain (several) keywords for preserving important dates. In particular, for lower-level data, this is usually the observation date and time (for example, stored in the `DATE-OBS` keyword value). When analyzing observed datasets, many calibration steps (like the dark, bias or flat-field), are commonly calculated on a per-observing-night basis.

However, the FITS standard's date format (`YYYY-MM-DDThh:mm:ss.ddd`) is based on the western (Gregorian) calendar. Dates that are stored in this format are complicated for automatic processing: a night starts in the final hours of one calendar day, and extends to the early hours of the next calendar day. As a result, to identify datasets from one night, we commonly need to search for two dates. However calendar peculiarities can make this identification very difficult. For example when an observation is done on the night separating two months (like the night starting on March 31st and going into April 1st),

or two years (like the night starting on December 31st 2018 and going into January 1st, 2019). To account for such situations, it is necessary to keep track of how many days are in a month, and leap years, and etc.

Gnuastro's `astscript-sort-by-night` script is created to help in such important scenarios. It uses Section 5.1 [Fits], page 128, to convert the FITS date format into the Unix epoch time (number of seconds since 00:00:00 of January 1st, 1970), using the `--datetosec` option. The Unix epoch time is a single number (integer, if not given in sub-second precision), enabling easy comparison and sorting of dates after January 1st, 1970.

You can use this script as a basis for making a much more highly customized sorting script. Here are some examples

- If you need to copy the files, but only need a single extension (not the whole file), you can add a step just before the making of the symbolic links, or copies, and change it to only copy a certain extension of the FITS file using the Fits program's `--copy` option, see Section 5.1.1.1 [HDU manipulation], page 131.
- If you need to classify the files with finer detail (for example the purpose of the dataset), you can add a step just before the making of the symbolic links, or copies, to specify a file-name prefix based on other certain keyword values in the files. For example when the FITS files have a keyword to specify if the dataset is a science, bias, or flat-field image. You can read it and to add a `sci-`, `bias-`, or `flat-` to the created file (after the `--prefix`) automatically.

  For example, let's assume the observing mode is stored in the hypothetical `MODE` keyword, which can have three values of `BIAS-IMAGE`, `SCIENCE-IMAGE` and `FLAT-EXP`. With the step below, you can generate a mode-prefix, and add it to the generated link/copy names (just correct the filename and extension of the first line to the script's variables):

```
modepref=$(astfits infile.fits -h1 \
                   | sed -e"s/'/ /g" \
                   | awk '$1=="MODE"{ \
                       if($3=="BIAS-IMAGE") print "bias-"; \
                       else if($3=="SCIENCE-IMAGE") print "sci-"; \
                       else if($3==FLAT-EXP) print "flat-"; \
                       else print $3, "NOT recognized"; exit 1}')
```

  Here is a description of it. We first use `astfits` to print all the keywords in extension `1` of `infile.fits`. In the FITS standard, string values (that we are assuming here) are placed in single quotes (`'`) which are annoying in this context/use-case. Therefore, we pipe the output of `astfits` into `sed` to remove all such quotes (substituting them with a blank space). The result is then piped to AWK for giving us the final mode-prefix: with `$1=="MODE"`, we ask AWK to only consider the line where the first column is `MODE`. There is an equal sign between the key name and value, so the value is the third column (`$3` in AWK). We thus use a simple `if-else` structure to look into this value and print our custom prefix based on it. The output of AWK is then stored in the `modepref` shell variable which you can add to the link/copy name.

  With the solution above, the increment of the file counter for each night will be independent of the mode. If you want the counter to be mode-dependent, you can add a different counter for each mode and use that counter instead of the generic counter for

each night (based on the value of `modepref`). But we'll leave the implementation of this step to you as an exercise.

10.1.1 Invoking astscript-sort-by-night

This script will read a FITS date formatted value from the given keyword, and classify the input FITS files into individual nights. This script can be used with the following general template:

```
$ astscript-sort-by-night [OPTION...] FITS-files
```

One line examples:

```
## Use the DATE-OBS keyword
$ astscript-sort-by-night --key=DATE-OBS /path/to/data/*.fits

## Make links to the input files with the 'img-' prefix
$ astscript-sort-by-night --link --prefix=img- /path/to/data/*.fits
```

This script will look into a HDU/extension (`--hdu`) for a keyword (`--key`) in the given FITS files and interpret the value as a date. The inputs will be separated by "night"s (9:00a.m to next day's 8:59:59a.m, spanning two calendar days, exact hour can be set with `--hour`).

The default output is a list of all the input files along with the following two columns: night number and file number in that night (sorted by time). With `--link` a symbolic link will be made (one for each input) that contains the night number, and number of file in that night (sorted by time), see the description of `--link` for more. When `--copy` is used instead of a link, a copy of the inputs will be made instead of symbolic link.

Below you can see one example where all the `target-*.fits` files in the `data` directory should be separated by observing night according to the `DATE-OBS` keyword value in their second extension (number `1`, recall that HDU counting starts from 0). You can see the output after the `ls` command.

```
$ astscript-sort-by-night -pimg- -h1 -kDATE-OBS data/target-*.fits
$ ls
img-n1-1.fits img-n1-2.fits img-n2-1.fits ...
```

The outputs can be placed in a different (already existing) directory by including that directory's name in the `--prefix` value, for example `--prefix=sorted/img-` will put them all under the `sorted` directory.

This script can be configured like all Gnuastro's programs (through command-line options, see Section 4.1.2 [Common options], page 95), with some minor differences that are described in Chapter 10 [Installed scripts], page 316. The particular options to this script are listed below:

`-h STR`
`--hdu=STR`
: The HDU/extension to use in all the given FITS files. All of the given FITS files must have this extension.

`-k STR`
`--key=STR`
: The keyword name that contains the FITS date format to classify/sort by.

`-H FLT`
`--hour=FLT`
The hour that defines the next "night". By default, all times before 9:00a.m are considered to belong to the previous calendar night. If a sub-hour value is necessary, it should be given in units of hours, for example `--hour=9.5` corresponds to 9:30a.m.

`-l`
`--link` Create a symbolic link for each input FITS file. This option cannot be used with `--copy`. The link will have a standard name in the following format (variable parts are written in `CAPITAL` letters and described after it):

`PnN-I.fits`

`P` This is the value given to `--prefix`. By default, its value is `./` (to store the links in the directory this script was run in). See the description of `--prefix` for more.

`N` This is the night-counter: starting from 1. `N` is just incremented by 1 for the next night, no matter how many nights (without any dataset) there are between two subsequent observing nights (its just an identifier for each night which you can easily map to different calendar nights).

`I` File counter in that night, sorted by time.

`-c`
`--copy` Make a copy of each input FITS file with the standard naming convention described in `--link`. With this option, instead of making a link, a copy is made. This option cannot be used with `--link`.

`-p STR`
`--prefix=STR`
Prefix to append before the night-identifier of each newly created link or copy. This option is thus only relevant with the `--copy` or `--link` options. See the description of `--link` for how its used. For example, with `--prefix=img-`, all the created file names in the current directory will start with `img-`, making outputs like `img-n1-1.fits` or `img-n3-42.fits`.

`--prefix` can also be used to store the links/copies in another directory relative to the directory this script is being run (it must already exist). For example `--prefix=/path/to/processing/img-` will put all the links/copies in the `/path/to/processing` directory, and the files (in that directory) will all start with `img-`.

11 Library

Each program in Gnuastro that was discussed in the prior chapters (or any program in general) is a collection of functions that is compiled into one executable file which can communicate directly with the outside world. The outside world in this context is the operating system. By communication, we mean that control is directly passed to a program from the operating system with a (possible) set of inputs and after it is finished, the program will pass control back to the operating system. For programs written in C and C++, the unique `main` function is in charge of this communication.

Similar to a program, a library is also a collection of functions that is compiled into one executable file. However, unlike programs, libraries don't have a `main` function. Therefore they can't communicate directly with the outside world. This gives you the chance to write your own `main` function and call library functions from within it. After compiling your program into a binary executable, you just have to *link* it to the library and you are ready to run (execute) your program. In this way, you can use Gnuastro at a much lower-level, and in combination with other libraries on your system, you can significantly boost your creativity.

This chapter starts with a basic introduction to libraries and how you can use them in Section 11.1 [Review of library fundamentals], page 320. The separate functions in the Gnuastro library are then introduced (classified by context) in Section 11.3 [Gnuastro library], page 332. If you end up routinely using a fixed set of library functions, with a well-defined input and output, it will be much more beneficial if you define a program for the job. Therefore, in its Section 3.2.2 [Version controlled source], page 72, Gnuastro comes with the Section 12.4.2 [The TEMPLATE program], page 454, to easily define your own programs(s).

11.1 Review of library fundamentals

Gnuastro's libraries are written in the C programming language. In Section 12.1 [Why C programming language?], page 445, we have thoroughly discussed the reasons behind this choice. C was actually created to write Unix, thus understanding the way C works can greatly help in effectively using programs and libraries in all Unix-like operating systems. Therefore, in the following subsections some important aspects of C, as it relates to libraries (and thus programs that depend on them) on Unix are reviewed. First we will discuss header files in Section 11.1.1 [Headers], page 321, and then go onto Section 11.1.2 [Linking], page 324. This section finishes with Section 11.1.3 [Summary and example on libraries], page 327. If you are already familiar with these concepts, please skip this section and go directly to Section 11.3 [Gnuastro library], page 332.

In theory, a full operating system (or any software) can be written as one function. Such a software would not need any headers or linking (that are discussed in the subsections below). However, writing that single function and maintaining it (adding new features, fixing bugs, documentation and etc) would be a programmer or scientist's worst nightmare! Furthermore, all the hard work that went into creating it cannot be reused in other software: every other programmer or scientist would have to re-invent the wheel. The ultimate purpose behind libraries (which come with headers and have to be linked) is to address this problem and increase modularity: "the degree to which a system's components may

be separated and recombined" (from Wikipedia). The more modular the source code of a program or library, the easier maintaining it will be, and all the hard work that went into creating it can be reused for a wider range of problems.

11.1.1 Headers

C source code is read from top to bottom in the source file, therefore program components (for example variables, data structures and functions) should all be *defined* or *declared* closer to the top of the source file: before they are used. *Defining* something in C or C++ is jargon for providing its full details. *Declaring* it, on the other-hand, is jargon for only providing the minimum information needed for the compiler to pass it temporarily and fill in the detailed definition later.

For a function, the *declaration* only contains the inputs and their data-types along with the output's type[1]. The *definition* adds to the declaration by including the exact details of what operations are done to the inputs to generate the output. As an example, take this simple summation function:

```
double
sum(double a, double b)
{
  return a + b;
}
```

What you see above is the *definition* of this function: it shows you (and the compiler) exactly what it does to the two `double` type inputs and that the output also has a `double` type. Note that a function's internal operations are rarely so simple and short, it can be arbitrarily long and complicated. This unreasonably short and simple function was chosen here for ease of reading. The declaration for this function is:

```
double
sum(double a, double b);
```

You can think of a function's declaration as a building's address in the city, and the definition as the building's complete blueprints. When the compiler confronts a call to a function during its processing, it doesn't need to know anything about how the inputs are processed to generate the output. Just as the postman doesn't need to know the inner structure of a building when delivering the mail. The declaration (address) is enough. Therefore by *declaring* the functions once at the start of the source files, we don't have to worry about *defining* them after they are used.

Even for a simple real-world operation (not a simple summation like above!), you will soon need many functions (for example, some for reading/preparing the inputs, some for the processing, and some for preparing the output). Although it is technically possible, managing all the necessary functions in one file is not easy and is contrary to the modularity principle (see Section 11.1 [Review of library fundamentals], page 320), for example the functions for preparing the input can be usable in your other projects with a different processing. Therefore, as we will see later (in Section 11.1.2 [Linking], page 324), the functions don't necessarily need to be defined in the source file where they are used. As long as their definitions are ultimately linked to the final executable, everything will be fine. For now, it is just important to remember that the functions that are called within

[1] Recall that in C, functions only have one output.

one source file must be declared within the source file (declarations are mandatory), but not necessarily defined there.

In the spirit of modularity, it is common to define contextually similar functions in one source file. For example, in Gnuastro, functions that calculate the median, mean and other statistical functions are defined in `lib/statistics.c`, while functions that deal directly with FITS files are defined in `lib/fits.c`.

Keeping the definition of similar functions in a separate file greatly helps their management and modularity, but this fact alone doesn't make things much easier for the caller's source code: recall that while definitions are optional, declarations are mandatory. So if this was all, the caller would have to manually copy and paste (*include*) all the declarations from the various source files into the file they are working on now. To address this problem, programmers have adopted the header file convention: the header file of a source code contains all the declarations that a caller would need to be able to use any of its functions. For example, in Gnuastro, `lib/statistics.c` (file containing function definitions) comes with `lib/gnuastro/statistics.h` (only containing function declarations).

The discussion above was mainly focused on functions, however, there are many more programming constructs such as pre-processor macros and data structures. Like functions, they also need to be known to the compiler when it confronts a call to them. So the header file also contains their definitions or declarations when they are necessary for the functions.

Pre-processor macros (or macros for short) are replaced with their defined value by the pre-processor before compilation. Conventionally they are written only in capital letters to be easily recognized. It is just important to understand that the compiler doesn't see the macros, it sees their fixed values. So when a header specifies macros you can do your programming without worrying about the actual values. The standard C types (for example `int`, or `float`) are very low-level and basic. We can collect multiple C types into a *structure* for a higher-level way to keep and pass-along data. See Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346, for some examples of macros and data structures.

The contents in the header need to be *include*d into the caller's source code with a special pre-processor command: `#include <path/to/header.h>`. As the name suggests, the *pre-processor* goes through the source code prior to the processor (or compiler). One of its jobs is to include, or merge, the contents of files that are mentioned with this directive in the source code. Therefore the compiler sees a single entity containing the contents of the main file and all the included files. This allows you to include many (sometimes thousands of) declarations into your code with only one line. Since the headers are also installed with the library into your system, you don't even need to keep a copy of them for each separate program, making things even more convenient.

Try opening some of the `.c` files in Gnuastro's `lib/` directory with a text editor to check out the include directives at the start of the file (after the copyright notice). Let's take `lib/fits.c` as an example. You will notice that Gnuastro's header files (like `gnuastro/fits.h`) are indeed within this directory (the `fits.h` file is in the `gnuastro/` directory). You will notice that files like `stdio.h`, or `string.h` are not in this directory (or anywhere within Gnuastro).

On most systems the basic C header files (like `stdio.h` and `string.h` mentioned above) are located in `/usr/include/`[2]. Your compiler is configured to automatically search that directory (and possibly others), so you don't have to explicitly mention these directories. Go ahead, look into the `/usr/include` directory and find `stdio.h` for example. When the necessary header files are not in those specific libraries, the pre-processor can also search in places other than the current directory. You can specify those directories with this pre-processor option[3]:

`-I DIR` "Add the directory `DIR` to the list of directories to be searched for header files. Directories named by '-I' are searched before the standard system include directories. If the directory `DIR` is a standard system include directory, the option is ignored to ensure that the default search order for system directories and the special treatment of system headers are not defeated..." (quoted from the GNU Compiler Collection manual). Note that the space between `I` and the directory is optional and commonly not used.

If the pre-processor can't find the included files, it will abort with an error. In fact a common error when building programs that depend on a library is that the compiler doesn't not know where a library's header is (see Section 3.3.5 [Known issues], page 89). So you have to manually tell the compiler where to look for the library's headers with the `-I` option. For a small software with one or two source files, this can be done manually (see Section 11.1.3 [Summary and example on libraries], page 327). However, to enhance modularity, Gnuastro (and most other bin/libraries) contain many source files, so the compiler is invoked many times[4]. This makes manual addition or modification of this option practically impossible.

To solve this problem, in the GNU build system, there are conventional environment variables for the various kinds of compiler options (or flags). These environment variables are used in every call to the compiler (they can be empty). The environment variable used for the C Pre-Processor (or CPP) is `CPPFLAGS`. By giving `CPPFLAGS` a value once, you can be sure that each call to the compiler will be affected. See Section 3.3.5 [Known issues], page 89, for an example of how to set this variable at configure time.

As described in Section 3.3.1.2 [Installation directory], page 79, you can select the top installation directory of a software using the GNU build system, when you `./configure` it. All the separate components will be put in their separate sub-directory under that, for example the programs, compiled libraries and library headers will go into `$prefix/bin` (replace `$prefix` with a directory), `$prefix/lib`, and `$prefix/include` respectively. For enhanced modularity, libraries that contain diverse collections of functions (like GSL, WCSLIB, and Gnuastro), put their header files in a sub-directory unique to themselves. For example all Gnuastro's header files are installed in `$prefix/include/gnuastro`. In your source code, you need to keep the library's sub-directory when including the headers from such libraries, for example `#include <gnuastro/fits.h>`[5]. Not all libraries need to follow this convention, for example CFITSIO only has one header (`fitsio.h`) which is directly installed in `$prefix/include`.

[2] The `include/` directory name is taken from the pre-processor's `#include` directive, which is also the motivation behind the 'I' in the `-I` option to the pre-processor.

[3] Try running Gnuastro's `make` and find the directories given to the compiler with the `-I` option.

[4] Nearly every command you see being executed after running `make` is one call to the compiler.

[5] the top `$prefix/include` directory is usually known to the compiler

11.1.2 Linking

To enhance modularity, similar functions are defined in one source file (with a `.c` suffix, see Section 11.1.1 [Headers], page 321, for more). After running `make`, each human-readable, `.c` file is translated (or compiled) into a computer-readable "object" file (ending with `.o`). Note that object files are also created when building programs, they aren't particular to libraries. Try opening Gnuastro's `lib/` and `bin/progname/` directories after running `make` to see these object files[6]. Afterwards, the object files are *linked* together to create an executable program or a library.

The object files contain the full definition of the functions in the respective `.c` file along with a list of any other function (or generally "symbol") that is referenced there. To get a list of those functions you can use the `nm` program which is part of GNU Binutils. For example from the top Gnuastro directory, run:

```
$ nm bin/arithmetic/arithmetic.o
```

This will print a list of all the functions (more generally, 'symbols') that were called within `bin/arithmetic/arithmetic.c` along with some further information (for example a `T` in the second column shows that this function is actually defined here, `U` says that it is undefined here). Try opening the `.c` file to check some of these functions for your self. Run `info nm` for more information.

To recap, the *compiler* created the separate object files mentioned above for each `.c` file. The *linker* will then combine all the symbols of the various object files (and libraries) into one program or library. In the case of Arithmetic (a program) the contents of the object files in `bin/arithmetic/` are copied (and re-ordered) into one final executable file which we can run from the operating system.

There are two ways to *link* all the necessary symbols: static and dynamic/shared. When the symbols (computer-readable function definitions in most cases) are copied into the output, it is called *static* linking. When the symbols are kept in their original file and only a reference to them is kept in the executable, it is called *dynamic*, or *shared* linking.

Let's have a closer look at the executable to understand this better: we'll assume you have built Gnuastro without any customization and installed Gnuastro into the default `/usr/local/` directory (see Section 3.3.1.2 [Installation directory], page 79). If you tried the `nm` command on one of Arithmetic's object files above, then with the command below you can confirm that all the functions that were defined in the object file above (had a `T` in the second column) are also defined in the `astarithmetic` executable:

```
$ nm /usr/local/bin/astarithmetic
```

These symbols/function have been statically linked (copied) in the final executable. But you will notice that there are still many undefined symbols in the executable (those with a `U` in the second column). One class of such functions are Gnuastro's own library functions that start with '`gal_`':

```
$ nm /usr/local/bin/astarithmetic | grep gal_
```

These undefined symbols (functions) are present in another file and will be linked to the Arithmetic program every time you run it. Therefore they are known as dynamically

[6] Gnuastro uses GNU Libtool for portable library creation. Libtool will also make a `.lo` file for each `.c` file when building libraries (`.lo` files are human-readable).

linked libraries[7]. As we saw above, static linking is done when the executable is being built. However, when a program is dynamically linked to a library, at build-time, the library's symbols are only checked with the available libraries: they are not actually copied into the program's executable. Every time you run the program, the (dynamic) linker will be activated and will try to link the program to the installed library before the program starts.

If you want all the libraries to be statically linked to the executables, you have to tell Libtool (which Gnuastro uses for the linking) to disable shared libraries at configure time[8]:

```
$ configure --disable-shared
```

Try configuring Gnuastro with the command above, then build and install it (as described in Section 1.1 [Quick start], page 1). Afterwards, check the `gal_` symbols in the installed Arithmetic executable like before. You will see that they are actually copied this time (have a `T` in the second column). If the second column doesn't convince you, look at the executable file size with the following command:

```
$ ls -lh /usr/local/bin/astarithmetic
```

It should be around 4.2 Megabytes with this static linking. If you configure and build Gnuastro again with shared libraries enabled (which is the default), you will notice that it is roughly 100 Kilobytes!

This huge difference would have been very significant in the old days, but with the roughly Terabyte storage drives commonly in use today, it is negligible. Fortunately, output file size is not the only benefit of dynamic linking: since it links to the libraries at run-time (rather than build-time), you don't have to re-build a higher-level program or library when an update comes for one of the lower-level libraries it depends on. You just install the new low-level library and it will automatically be used/linked next time in the programs that use it. To be fair, this also creates a few complications[9]:

- Reproducibility: Even though your high-level tool has the same version as before, with the updated library, you might not get the same results.
- Broken links: if some functions have been changed or removed in the updated library, then the linker will abort with an error at run-time. Therefore you need to re-build your higher-level program or library.

To see a list of all the shared libraries that are needed for a program or a shared library to run, you can use GNU C library's `ldd`[10] program, for example:

```
$ ldd /usr/local/bin/astarithmetic
```

Library file names (in their installation directory) start with a `lib` and their ending (suffix) shows if they are static (`.a`) or dynamic (`.so`), as described below. The name of the library is in the middle of these two, for example `libgsl.a` or `libgnuastro.a` (GSL

[7] Do not confuse dynamically *linked* libraries with dynamically *loaded* libraries. The former (that is discussed here) are only loaded once at the program startup. However, the latter can be loaded anytime during the program's execution, they are also known as plugins.

[8] Libtool is very common and is commonly used. Therefore, you can use this option to configure on most programs using the GNU build system if you want static linking.

[9] Both of these can be avoided by joining the mailing lists of the lower-level libraries and checking the changes in newer versions before installing them. Updates that result in such behaviors are generally heavily emphasized in the release notes.

[10] If your operating system is not using the GNU C library, you might need another tool.

and Gnuastro's static libraries), and `libgsl.so.23.0.0` or `libgnuastro.so.4.0.0` (GSL and Gnuastro's shared library, the numbers may be different).

- A static library is known as an archive file and has the `.a` suffix. A static library is not an executable file.
- A shared library ends with the `.so.X.Y.Z` suffix and is executable. The three numbers in the suffix, describe the version of the shared library. Shared library versions are defined to allow multiple versions of a shared library simultaneously on a system and to help detect possible updates in the library and programs that depend on it by the linker.

  It is very important to mention that this version number is different from the software version number (see Section 1.5 [Version numbering], page 7), so do not confuse the two. See the "Library interface versions" chapter of GNU Libtool for more.

  For each shared library, we also have two symbolic links ending with `.so.X` and `.so`. They are automatically set by the installer, but you can change them (point them to another version of the library) when you have multiple versions of a library on your system.

Libraries that are built with GNU Libtool (including Gnuastro and its dependencies), build both static and dynamic libraries by default and install them in `prefix/lib/` directory (for more on `prefix`, see Section 3.3.1.2 [Installation directory], page 79). In this way, programs depending on the libraries can link with them however they prefer. See the contents of `/usr/local/lib` with the command below to see both the static and shared libraries available there, along with their executable nature and the symbolic links:

```
$ ls -l /usr/local/lib/
```

To link with a library, the linker needs to know where to find the library. *At compilation time*, these locations can be passed to the linker with two separate options (see Section 11.1.3 [Summary and example on libraries], page 327, for an example) as described below. You can see these options and their usage in practice while building Gnuastro (after running `make`):

`-L DIR` Will tell the linker to look into `DIR` for the libraries. For example `-L/usr/local/lib`, or `-L/home/yourname/.local/lib`. You can make multiple calls to this option, so the linker looks into several directories at compilation time. Note that the space between `L` and the directory is optional and commonly ignored (written as `-LDIR`).

`-lLIBRARY`
Specify the unique library identifier/name (not containing directory or shared/dynamic nature) to be linked with the executable. As discussed above, library file names have fixed parts which must not be given to this option. So `-lgsl` will guide the linker to either look for `libgsl.a` or `libgsl.so` (depending on the type of linking it is suppose to do). You can link many libraries by repeated calls to this option.

Very important: The place of this option on the compiler's command matters. This is often a source of confusion for beginners, so let's assume you have asked the linker to link with library A using this option. As soon as the linker confronts this option, it looks into the list of the undefined symbols it has found until that

point and does a search in library A for any of those symbols. If any pending undefined symbol is found in library A, it is used. After the search in undefined symbols is complete, the contents of library A are completely discarded from the linker's memory. Therefore, if a later object file or library uses an unlinked symbol in library A, the linker will abort after it has finished its search in all the input libraries or object files.

As an example, Gnuastro's `gal_fits_img_read` function depends on the `fits_read_pix` function of CFITSIO (specified with `-lcfitsio`, which in turn depends on the cURL library, called with `-lcurl`). So the proper way to link something that uses this function is `-lgnuastro -lcfitsio -lcurl`. If instead, you give: `-lcfitsio -lgnuastro` the linker will complain and abort. To avoid such linking complexities when using Gnuastro's library, we recommend using Section 11.2 [BuildProgram], page 328.

If you have compiled and linked your program with a dynamic library, then the dynamic linker also needs to know the location of the libraries after building the program: *every time* the program is run afterwards. Therefore, it may happen that you don't get any errors when compiling/linking a program, but are unable to run your program because of a failure to find a library. This happens because the dynamic linker hasn't found the dynamic library *at run time*.

To find the dynamic libraries at run-time, the linker looks into the paths, or directories, in the `LD_LIBRARY_PATH` environment variable. For a discussion on environment variables, especially search paths like `LD_LIBRARY_PATH`, and how you can add new directories to them, see Section 3.3.1.2 [Installation directory], page 79.

11.1.3 Summary and example on libraries

After the mostly abstract discussions of Section 11.1.1 [Headers], page 321, and Section 11.1.2 [Linking], page 324, we'll give a small tutorial here. But before that, let's recall the general steps of how your source code is prepared, compiled and linked to the libraries it depends on so you can run it:

1. The **pre-processor** includes the header (`.h`) files into the function definition (`.c`) files, expands pre-processor macros and generally prepares the human-readable source for compilation (reviewed in Section 11.1.1 [Headers], page 321).
2. The **compiler** will translate (compile) the human-readable contents of each source (merged `.c` and the `.h` files, or generally the output of the pre-processor) into the computer-readable code of `.o` files.
3. The **linker** will link the called function definitions from various compiled files to create one unified object. When the unified product has a `main` function, this function is the product's only entry point, enabling the operating system or user to directly interact with it, so the product is a program. When the product doesn't have a `main` function, the linker's product is a library and its exported functions can be linked to other executables (it has many entry points).

The GNU Compiler Collection (or GCC for short) will do all three steps. So as a first example, from Gnuastro's source, go to `tests/lib/`. This directory contains the library tests, you can use these as some simple tutorials. For this demonstration, we will compile and run the `arraymanip.c`. This small program will call Gnuastro library for some simple

operations on an array (open it and have a look). To compile this program, run this command inside the directory containing it.

```
$ gcc arraymanip.c -lgnuastro -lm -o arraymanip
```

The two `-lgnuastro` and `-lm` options (in this order) tell GCC to first link with the Gnuastro library and then with C's math library. The `-o` option is used to specify the name of the output executable, without it the output file name will be `a.out` (on most OSs), independent of your input file name(s).

If your top Gnuastro installation directory (let's call it `$prefix`, see Section 3.3.1.2 [Installation directory], page 79) is not recognized by GCC, you will get pre-processor errors for unknown header files. Once you fix it, you will get linker errors for undefined functions. To fix both, you should run GCC as follows: additionally telling it which directories it can find Gnuastro's headers and compiled library (see Section 11.1.1 [Headers], page 321, and Section 11.1.2 [Linking], page 324):

```
$ gcc -I$prefix/include -L$prefix/lib arraymanip.c -lgnuastro -lm     \
      -o arraymanip
```

This single command has done all the pre-processor, compilation and linker operations. Therefore no intermediate files (object files in particular) were created, only a single output executable was created. You are now ready to run the program with:

```
$ ./arraymanip
```

The Gnuastro functions called by this program only needed to be linked with the C math library. But if your program needs WCS coordinate transformations, needs to read a FITS file, needs special math operations (which include its linear algebra operations), or you want it to run on multiple CPU threads, you also need to add these libraries in the call to GCC: `-lgnuastro -lwcs -lcfitsio -lgsl -lgslcblas -pthread -lm`. In Section 11.3 [Gnuastro library], page 332, where each function is documented, it is mentioned which libraries (if any) must also be linked when you call a function. If you feel all these linkings can be confusing, please consider Gnuastro's Section 11.2 [BuildProgram], page 328, program.

11.2 BuildProgram

The number and order of libraries that are necessary for linking a program with Gnuastro library might be too confusing when you need to compile a small program for one particular job (with one source file). BuildProgram will use the information gathered during configuring Gnuastro and link with all the appropriate libraries on your system. This will allow you to easily compile, link and run programs that use Gnuastro's library with one simple command and not worry about which libraries to link to, or the linking order.

BuildProgram uses GNU Libtool to find the necessary libraries to link against (GNU Libtool is the same program that builds all of Gnuastro's libraries and programs when you run `make`). So in the future, if Gnuastro's prerequisite libraries change or other libraries are added, you don't have to worry, you can just run BuildProgram and internal linking will be done correctly.

BuildProgram requires GNU Libtool: BuildProgram depends on GNU Libtool, other implementations don't have some necessary features. If GNU Libtool isn't available at Gnuastro's configure time, you will get a notice at the end of the configuration step and BuildProgram will not be built or installed. Please see Section 3.1.2 [Optional dependencies], page 64, for more information.

11.2.1 Invoking BuildProgram

BuildProgram will compile and link a C source program with Gnuastro's library and all its dependencies, greatly facilitating the compilation and running of small programs that use Gnuastro's library. The executable name is `astbuildprog` with the following general template:

```
$ astbuildprog [OPTION...] C_SOURCE_FILE
```

One line examples:

```
## Compile, link and run 'myprogram.c':
$ astbuildprog myprogram.c

## Similar to previous, but with optimization and compiler warnings:
$ astbuildprog -Wall -O2 myprogram.c

## Compile and link 'myprogram.c', then run it with 'image.fits'
## as its argument:
$ astbuildprog myprogram.c image.fits

## Also look in other directories for headers and linking:
$ astbuildprog -Lother -Iother/dir myprogram.c

## Just build (compile and link) 'myprogram.c', don't run it:
$ astbuildprog --onlybuild myprogram.c
```

If BuildProgram is to run, it needs a C programming language source file as input. By default it will compile and link the program to build the a final executable file and run it. The built executable name can be set with the optional `--output` option. When no output name is set, BuildProgram will use Gnuastro's Section 4.8 [Automatic output], page 124, and remove the suffix of the input and use that as the output name. For the full list of options that BuildProgram shares with other Gnuastro programs, see Section 4.1.2 [Common options], page 95. You may also use Gnuastro's Section 4.2 [Configuration files], page 106, to specify other libraries/headers to use for special directories and not have to type them in every time.

The first argument is considered to be the C source file that must be compiled and linked. Any other arguments (non-option tokens on the command-line) will be passed onto the program when BuildProgram wants to run it. Recall that by default BuildProgram will run the program after building it. This behavior can be disabled with the `--onlybuild` option.

When the `--quiet` option (see Section 4.1.2.3 [Operating mode options], page 100) is not called, BuildPrograms will print the compilation and running commands. Once your program grows and you break it up into multiple files (which are much more easily managed with Make), you can use the linking flags of the non-quiet output in your `Makefile`.

`-I STR`
`--includedir=STR`
: Directory to search for files that you `#include` in your C program. Note that headers relating to Gnuastro and its dependencies don't need this option. This is only necessary if you want to use other headers. It may be called multiple times and order matters. This directory will be searched before those of Gnuastro's build and also the system search directories. See Section 11.1.1 [Headers], page 321, for a thorough introduction.

  From the GNU C Pre-Processor manual: "Add the directory `STR` to the list of directories to be searched for header files. Directories named by `-I` are searched before the standard system include directories. If the directory `STR` is a standard system include directory, the option is ignored to ensure that the default search order for system directories and the special treatment of system headers are not defeated".

`-L STR`
`--linkdir=STR`
: Directory to search for compiled libraries to link the program with. Note that all the directories that Gnuastro was built with will already be used by BuildProgram (GNU Libtool). This option is only necessary if your libraries are in other directories. Multiple calls to this option are possible and order matters. This directory will be searched before those of Gnuastro's build and also the system search directories. See Section 11.1.2 [Linking], page 324, for a thorough introduction.

`-l STR`
`--linklib=STR`
: Library to link with your program. Note that all the libraries that Gnuastro was built with will already be linked by BuildProgram (GNU Libtool). This option is only necessary if you want to link with other directories. Multiple calls to this option are possible and order matters. This library will be linked before Gnuastro's library or its dependencies. See Section 11.1.2 [Linking], page 324, for a thorough introduction.

`-O INT/STR`
`--optimize=INT/STR`
: Compiler optimization level: 0 (for no optimization, good debugging), 1, 2, 3 (for the highest level of optimizations). From the GNU Compiler Collection (GCC) manual: "Without any optimization option, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a break point between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you expect from the source code. Turning on optimization flags makes the compiler attempt to improve the performance and/or code size at the expense of compilation time and possibly the ability to debug the program." Please see your compiler's manual for the full list of acceptable values to this option.

`-g`
`--debug` Emit extra information in the compiled binary for use by a debugger. When calling this option, it is best to explicitly disable optimization with `-O0`. To combine both options you can run `-gO0` (see Section 4.1.1.2 [Options], page 93, for how short options can be merged into one).

`-W STR`
`--warning=STR`
Print compiler warnings on command-line during compilation. "Warnings are diagnostic messages that report constructions that are not inherently erroneous but that are risky or suggest there may have been an error." (from the GCC manual). It is always recommended to compile your programs with warnings enabled.

All compiler warning options that start with `W` are usable by this option in BuildProgram also, see your compiler's manual for the full list. Some of the most common values to this option are: `pedantic` (Warnings related to standard C) and `all` (all issues the compiler confronts).

`-t`
`--tag=STR`
The language configuration information. Libtool can build objects and libraries in many languages. In many cases, it can identify the language automatically, but when it doesn't you can use this option to explicitly notify Libtool of the language. The acceptable values are: `CC` for C, `CXX` for C++, `GCJ` for Java, `F77` for Fortran 77, `FC` for Fortran, `GO` for Go and `RC` for Windows Resource. Note that the Gnuastro library is not yet fully compatible with all these languages.

`-b`
`--onlybuild`
Only build the program, don't run it. By default, the built program is immediately run afterwards.

`-d`
`--deletecompiled`
Delete the compiled binary file after running it. This option is only relevant when the compiled program is run after being built. In other words, it is only relevant when `--onlybuild` is not called. It can be useful when you are busy testing a program or just want a fast result and the actual binary/compiled file is not of later use.

`-a STR`
`--la=STR` Use the given `.la` file (Libtool control file) instead of the one that was produced from Gnuastro's configuration results. The Libtool control file keeps all the necessary information for building and linking a program with a library built by Libtool. The default `prefix/lib/libgnuastro.la` keeps all the information necessary to build a program using the Gnuastro library gathered during configure time (see Section 3.3.1.2 [Installation directory], page 79, for prefix). This option is useful when you prefer to use another Libtool control file.

11.3 Gnuastro library

Gnuastro library's programming constructs (function declarations, macros, data structures, or global variables) are classified by context into multiple header files (see Section 11.1.1 [Headers], page 321)[11]. In this section, the functions in each header will be discussed under a separate sub-section, which includes the name of the header. Assuming a function declaration is in `headername.h`, you can include its declaration in your source code with:

```
# include <gnuastro/headername.h>
```

The names of all constructs in `headername.h` are prefixed with `gal_headername_` (or `GAL_HEADERNAME_` for macros). The `gal_` prefix stands for *G*NU *A*stronomy *L*ibrary.

Gnuastro library functions are compiled into a single file which can be linked on the command-line with the `-lgnuastro` option. See Section 11.1.2 [Linking], page 324, and Section 11.1.3 [Summary and example on libraries], page 327, for an introduction on linking and some fully working examples of the libraries.

Gnuastro's library is a high-level library which depends on lower level libraries for some operations (see Section 3.1 [Dependencies], page 61). Therefore if at least one of Gnuastro's functions in your program use functions from the dependencies, you will also need to link those dependencies after linking with Gnuastro. See Section 11.2 [BuildProgram], page 328, for a convenient way to deal with the dependencies. BuildProgram will take care of the libraries to link with your program (which uses the Gnuastro library), and can even run the built program afterwards. Therefore it allows you to conveniently focus on your exciting science/research when using Gnuastro's libraries.

Libraries are still under heavy development: Gnuastro was initially created to be a collection of command-line programs. However, as the programs and their the shared functions grew, internal (not installed) libraries were added. Since the 0.2 release, the libraries are install-able. Hence the libraries are currently under heavy development and will significantly evolve between releases and will become more mature and stable in due time. It will stabilize with the removal of this notice. Check the `NEWS` file for interface changes. If you use the Info version of this manual (see Section 4.3.4 [Info], page 111), you don't have to worry: the documentation will correspond to your installed version.

11.3.1 Configuration information (`config.h`)

The `gnuastro/config.h` header contains information about the full Gnuastro installation on your system. Gnuastro developers should note that this is the only header that is not available within Gnuastro, it is only available to a Gnuastro library user *after* installation. Within Gnuastro, `config.h` (which is included in every Gnuastro `.c` file, see Section 12.3 [Coding conventions], page 448) has more than enough information about the overall Gnuastro installation.

[11] Within Gnuastro's source, all installed `.h` files in `lib/gnuastro/` are accompanied by a `.c` file in `/lib/`.

`GAL_CONFIG_VERSION` [Macro]

This macro can be used as a string literal[12] containing the version of Gnuastro that is being used. See Section 1.5 [Version numbering], page 7, for the version formats. For example:

```
printf("Gnuastro version: %s\n", GAL_CONFIG_VERSION);
```

or

```
char *gnuastro_version=GAL_CONFIG_VERSION;
```

`GAL_CONFIG_HAVE_LIBGIT2` [Macro]

Libgit2 is an optional dependency of Gnuastro (see Section 3.1.2 [Optional dependencies], page 64). When it is installed and detected at configure time, this macro will have a value of `1` (one). Otherwise, it will have a value of `0` (zero). Gnuastro also comes with some wrappers to make it easier to use libgit2 (see Section 11.3.26 [Git wrappers (`git.h`)], page 435).

`GAL_CONFIG_HAVE_FITS_IS_REENTRANT` [Macro]

This macro will have a value of 1 when the CFITSIO of the host system has the `fits_is_reentrant` function (available from CFITSIO version 3.30). This function is used to see if CFITSIO was configured to read a FITS file simultaneously on different threads.

`GAL_CONFIG_HAVE_WCSLIB_VERSION` [Macro]

WCSLIB is the reference library for world coordinate system transformation (see Section 3.1.1.3 [WCSLIB], page 63, and Section 11.3.13 [World Coordinate System (`wcs.h`)], page 392). However, only more recent versions of WCSLIB also provide its version number. If the WCSLIB that is installed on the system provides its version (through the possibly existing `wcslib_version` function), this macro will have a value of one, otherwise it will have a value of zero.

`GAL_CONFIG_HAVE_PTHREAD_BARRIER` [Macro]

The POSIX threads standard define barriers as an optional requirement. Therefore, some operating systems choose to not include it. As one of the `./configure` step checks, Gnuastro we check if your system has this POSIX thread barriers. If so, this macro will have a value of `1`, otherwise it will have a value of `0`. see Section 11.3.2.1 [Implementation of `pthread_barrier`], page 334, for more.

`GAL_CONFIG_SIZEOF_LONG` [Macro]
`GAL_CONFIG_SIZEOF_SIZE_T` [Macro]

The size of (number of bytes in) the system's `long` and `size_t` types. Their values are commonly either 4 or 8 for 32-bit and 64-bit systems. You can also get this value with the expression '`sizeof size_t`' for example without having to include this header.

11.3.2 Multithreaded programming (`threads.h`)

In recent years, newer CPUs don't have significantly higher frequencies any more. However, CPUs are being manufactured with more cores, enabling more than one operation (thread)

[12] `https://en.wikipedia.org/wiki/String_literal`

at each instant. This can be very useful to speed up many aspects of processing and in particular image processing.

Most of the programs in Gnuastro utilize multi-threaded programming for the CPU intensive processing steps. This can potentially lead to a significant decrease in the running time of a program, see Section 4.4.1 [A note on threads], page 113. In terms of reading the code, you don't need to know anything about multi-threaded programming. You can simply follow the case where only one thread is to be used. In these cases, threads are not used and can be completely ignored.

When the C language was defined (the K&R's book was written), using threads was not common, so C's threading capabilities aren't introduced there. Gnuastro uses POSIX threads for multi-threaded programming, defined in the `pthread.h` system wide header. There are various resources for learning to use POSIX threads. An excellent tutorial (`https://computing.llnl.gov/tutorials/pthreads/`
) is provided by the Lawrence Livermore National Laboratory, with abundant figures to better understand the concepts, it is a very good start. The book 'Advanced programming in the Unix environment'[13], by Richard Stevens and Stephen Rago, Addison-Wesley, 2013 (Third edition) also has two chapters explaining the POSIX thread constructs which can be very helpful.

An alternative to POSIX threads was OpenMP, but POSIX threads are low level, allowing much more control, while being easier to understand, see Section 12.1 [Why C programming language?], page 445. All the situations where threads are used in Gnuastro currently are completely independent with no need of coordination between the threads. Such problems are known as "embarrassingly parallel" problems. They are some of the simplest problems to solve with threads and are also the ones that benefit most from them, see the LLNL introduction[14].

One very useful POSIX thread concept is `pthread_barrier`. Unfortunately, it is only an optional feature in the POSIX standard, so some operating systems don't include it. Therefore in Section 11.3.2.1 [Implementation of `pthread_barrier`], page 334, we introduce our own implementation. This is a rather technical section only necessary for more technical readers and you can safely ignore it. Following that, we describe the helper functions in this header that can greatly simplify writing a multi-threaded program, see Section 11.3.2.2 [Gnuastro's thread related functions], page 335, for more.

11.3.2.1 Implementation of `pthread_barrier`

One optional feature of the POSIX Threads standard is the `pthread_barrier` concept. It is a very useful high-level construct that allows for independent threads to "wait" behind a "barrier" for the rest after they finish. Barriers can thus greatly simplify the code in a multi-threaded program, so they are heavily used in Gnuastro. However, since its an optional feature in the POSIX standard, some operating systems don't include it. So to make Gnuastro portable, we have written our own implementation of those `pthread_barrier` functions.

[13] Don't let the title scare you! The two chapters on Multi-threaded programming are very self-sufficient and don't need any more knowledge than K&R.

[14] `https://computing.llnl.gov/tutorials/parallel_comp/`

At `./configure` time, Gnuastro will check if `pthread_barrier` constructs are available on your system or not. If `pthread_barrier` is not available, our internal implementation will be compiled into the Gnuastro library and the definitions and declarations below will be usable in your code with `#include <gnuastro/threads.h>`.

`pthread_barrierattr_t` [Type]
Type to specify the attributes of a POSIX threads barrier.

`pthread_barrier_t` [Type]
Structure defining the POSIX threads barrier.

`int` [Function]
`pthread_barrier_init` (*pthread_barrier_t* `*b`, *pthread_barrierattr_t* `*attr`, *unsigned int* `limit`)
Initialize the barrier `b`, with the attributes `attr` and total `limit` (a number of) threads that must wait behind it. This function must be called before spinning off threads.

`int` [Function]
`pthread_barrier_wait` (*pthread_barrier_t* `*b`)
This function is called within each thread, just before it is ready to return. Once a thread's function hits this, it will "wait" until all the other functions are also finished.

`int` [Function]
`pthread_barrier_destroy` (*pthread_barrier_t* `*b`)
Destroy all the information in the barrier structure. This should be called by the function that spun-off the threads after all the threads have finished.

> **Destroy a barrier before re-using it:** It is very important to destroy the barrier before (possibly) reusing it. This destroy function not only destroys the internal structures, it also waits (in 1 microsecond intervals, so you will not notice!) until all the threads don't need the barrier structure any more. If you immediately start spinning off new threads with a not-destroyed barrier, then the internal structure of the remaining threads will get mixed with the new ones and you will get very strange and apparently random errors that are extremely hard to debug.

11.3.2.2 Gnuastro's thread related functions

The POSIX Threads functions offered in the C library are very low-level and offer a great range of control over the properties of the threads. So if you are interested in customizing your tools for complicated thread applications, it is strongly encouraged to get a nice familiarity with them. Some resources were introduced in Section 11.3.2 [Multithreaded programming (`threads.h`)], page 333.

However, in many cases used in astronomical data analysis, you don't need communication between threads and each target operation can be done independently. Since such operations are very common, Gnuastro provides the tools below to facilitate the creation and management of jobs without any particular knowledge of POSIX Threads for such operations. The most interesting high-level functions of this section are the `gal_threads_number` and `gal_threads_spin_off` that identify the number of threads on the system and

spin-off threads. You can see a demonstration of using these functions in Section 11.4.3 [Library demo - multi-threaded operation], page 439.

`gal_threads_params` [C `struct`]

Structure keeping the parameters of each thread. When each thread is created, a pointer to this structure is passed to it. The `params` element can be the pointer to a structure defined by the user which contains all the necessary parameters to pass onto the worker function. The rest of the elements within this structure are set internally by `gal_threads_spin_off` and are relevant to the worker function.

```
struct gal_threads_params
{
  size_t            id; /* Id of this thread.                  */
  void         *params; /* User-identified pointer.            */
  size_t       *indexs; /* Target indexs given to this thread. */
  pthread_barrier_t *b; /* Barrier for all threads.            */
};
```

`size_t` [Function]
`gal_threads_number ()`

Return the number of threads that the operating system has available for your program. This number is usually fixed for a single machine and doesn't change. So this function is useful when you want to run your program on different machines (with different CPUs).

`void` [Function]
`gal_threads_spin_off (`*void* `*(*worker)(void *)`, *void* `*caller_params`, *size_t* `numactions`, *size_t* `numthreads)`

Distribute `numactions` jobs between `numthreads` threads and spin-off each thread by calling the `worker` function. The `caller_params` pointer will also be passed to `worker` as part of the `gal_threads_params` structure. For a fully working example of this function, please see Section 11.4.3 [Library demo - multi-threaded operation], page 439.

`void` [Function]
`gal_threads_attr_barrier_init (`*pthread_attr_t* `*attr`, *pthread_barrier_t* `*b`, *size_t* `limit)`

This is a low-level function in case you don't want to use `gal_threads_spin_off`. It will initialize the general thread attribute `attr` and the barrier `b` with `limit` threads to wait behind the barrier. For maximum efficiency, the threads initialized with this function will be detached. Therefore no communication is possible between these threads and in particular `pthread_join` won't work on these threads. You have to use the barrier constructs to wait for all threads to finish.

`void` [Function]
`gal_threads_dist_in_threads (`*size_t* `numactions`, *size_t* `numthreads`, *size_t* `**outthrds`, *size_t* `*outthrdcols)`

This is a low-level function in case you don't want to use `gal_threads_spin_off`. Identify the "index"es (starting from 0) of the actions to be done on each thread in

the `outthrds` array. `outthrds` is treated as a 2D array with `numthreads` rows and `outthrdcols` columns. The indexs in each row, identify the actions that should be done by one thread. Please see the explanation below to understand the purpose of this operation.

Let's assume you have A actions (where there is only one function and the input values differ for each action) and T threads available to the system with $A > T$ (common values for these two would be $A > 1000$ and $T < 10$). Spinning off a thread is not a cheap job and requires a significant number of CPU cycles. Therefore, creating A threads is not the best way to address such a problem. The most efficient way to manage the actions is such that only T threads are created, and each thread works on a list of actions identified for it in series (one after the other). This way your CPU will get all the actions done with minimal overhead.

The purpose of this function is to do what we explained above: each row in the `outthrds` array contains the indexs of actions which must be done by one thread. `outthrds` contains `outthrdcols` columns. In using `outthrds`, you don't have to know the number of columns. The `GAL_BLANK_SIZE_T` macro has a role very similar to a string's `\0`: every row finishes with this macro, so can easily stop parsing the indexes in the row when you confront it. Please see the example program in `tests/lib/multithread.c` for a demonstration.

11.3.3 Library data types (`type.h`)

Data in astronomy can have many types, numeric (numbers) and strings (names, identifiers). The former can also be divided into integers and floats, see Section 4.5 [Numeric data types], page 115, for a thorough discussion of the different numeric data types and which one is useful for different contexts.

To deal with the very large diversity of types that are available (and used in different contexts), in Gnuastro each type is identified with global integer variable with a fixed name, this variable is then passed onto functions that can work on any type or is stored in Gnuastro's Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346, as one piece of meta-data.

The actual values within these integer constants is irrelevant and you should never rely on them. When you need to check, explicitly use the named variable in the table below. If you want to check with more than one type, you can use C's `switch` statement.

Since Gnuastro heavily deals with file input-output, the types it defines are fixed width types, these types are portable to all systems and are defined in the standard C header `stdint.h`. You don't need to include this header, it is included by any Gnuastro header that deals with the different types. However, the most commonly used types in a C (or C++) program (for example `int` or `long`) are not defined by their exact width (storage size), but by their minimum storage. So for example on some systems, `int` may be 2 bytes (16-bits, the minimum required by the standard) and on others it may be 4 bytes (32-bits, common in modern systems).

With every type, a unique "blank" value (or place holder showing the absence of data) can be defined. Please see Section 11.3.5 [Library blank values (`blank.h`)], page 343, for constants that Gnuastro recognizes as a blank value for each type. See Section 4.5 [Numeric data types], page 115, for more explanation on the limits and particular aspects of each type.

`GAL_TYPE_INVALID` [Global integer]
This is just a place holder to specifically mark that no type has been set.

`GAL_TYPE_BIT` [Global integer]
Identifier for a bit-stream. Currently no program in Gnuastro works directly on bits, but features will be added in the future.

`GAL_TYPE_UINT8` [Global integer]
Identifier for an unsigned, 8-bit integer type: `uint8_t` (from `stdint.h`), or an `unsigned char` in most modern systems.

`GAL_TYPE_INT8` [Global integer]
Identifier for a signed, 8-bit integer type: `int8_t` (from `stdint.h`), or an `signed char` in most modern systems.

`GAL_TYPE_UINT16` [Global integer]
Identifier for an unsigned, 16-bit integer type: `uint16_t` (from `stdint.h`), or an `unsigned short` in most modern systems.

`GAL_TYPE_INT16` [Global integer]
Identifier for a signed, 16-bit integer type: `int16_t` (from `stdint.h`), or a `short` in most modern systems.

`GAL_TYPE_UINT32` [Global integer]
Identifier for an unsigned, 32-bit integer type: `uint32_t` (from `stdint.h`), or an `unsigned int` in most modern systems.

`GAL_TYPE_INT32` [Global integer]
Identifier for a signed, 32-bit integer type: `int32_t` (from `stdint.h`), or an `int` in most modern systems.

`GAL_TYPE_UINT64` [Global integer]
Identifier for an unsigned, 64-bit integer type: `uint64_t` (from `stdint.h`), or an `unsigned long` in most modern 64-bit systems.

`GAL_TYPE_INT64` [Global integer]
Identifier for a signed, 64-bit integer type: `int64_t` (from `stdint.h`), or an `long` in most modern 64-bit systems.

`GAL_TYPE_SIZE_T` [Global integer]
Identifier for a `size_t` type. This is just an alias to `uint32`, or `uint64` types for 32-bit, or 64-bit systems respectively.

`GAL_TYPE_ULONG` [Global integer]
Identifier for a `unsigned long` type. This is just an alias to `uint32`, or `uint64` types for 32-bit, or 64-bit systems respectively.

`GAL_TYPE_LONG` [Global integer]
Identifier for a `long` type. This is just an alias to `int32`, or `int64` types for 32-bit, or 64-bit systems respectively.

`GAL_TYPE_FLOAT32` [Global integer]
Identifier for a 32-bit single precision floating point type or `float` in C.

`GAL_TYPE_FLOAT64` [Global integer]
Identifier for a 64-bit double precision floating point type or `double` in C.

`GAL_TYPE_COMPLEX32` [Global integer]
Identifier for a complex number composed of two `float` types. Note that the complex type is not yet fully implemented in all Gnuastro's programs.

`GAL_TYPE_COMPLEX64` [Global integer]
Identifier for a complex number composed of two `double` types. Note that the complex type is not yet fully implemented in all Gnuastro's programs.

`GAL_TYPE_STRING` [Global integer]
Identifier for a string of characters (`char *`).

`GAL_TYPE_STRLL` [Global integer]
Identifier for a linked list of string of characters (`gal_list_str_t`, see Section 11.3.8.1 [List of strings], page 358).

The functions below are defined to make working with the integer constants above easier. In the functions below, the constants above can be used for the `type` input argument.

`size_t` [Function]
`gal_type_sizeof` (*uint8_t* `type`)
Return the number of bytes occupied by `type`. Internally, this function uses C's `sizeof` operator to measure the size of each type.

`char *` [Function]
`gal_type_name` (*uint8_t* `type`, *int* `long_name`)
Return a string literal that contains the name of `type`. It can return both short and long formats of the type names (for example `f32` and `float32`). If `long_name` is non-zero, the long format will be returned, otherwise the short name will be returned. The output string is statically allocated, so it should not be freed. This function is the inverse of the `gal_type_from_name` function. For the full list of names/strings that this function will return, see Section 4.5 [Numeric data types], page 115.

`uint8_t` [Function]
`gal_type_from_name` (*char* `*str`)
Return the Gnuastro integer constant that corresponds to the string `str`. This function is the inverse of the `gal_type_name` function and accepts both the short and long formats of each type. For the full list of names/strings that this function will return, see Section 4.5 [Numeric data types], page 115.

`void` [Function]
`gal_type_min` (*uint8_t* `type`, *void* `*in`)
Put the minimum possible value of `type` in the space pointed to by `in`. Since the value can have any type, this function doesn't return anything, it assumes the space for the given type is available to `in` and writes the value there. Here is one example

```
int32_t min;
```

```
gal_type_min(GAL_TYPE_INT32, &min);
```

Note: Do not use the minimum value for a blank value of a general (initially unknown) type, please use the constants/functions provided in Section 11.3.5 [Library blank values (`blank.h`)], page 343, for the definition and usage of blank values.

`void` [Function]
`gal_type_max` (*uint8_t* `type`, *void* `*in`)

Put the maximum possible value of `type` in the space pointed to by `in`. Since the value can have any type, this function doesn't return anything, it assumes the space for the given type is available to `in` and writes the value there. Here is one example

```
uint16_t max;
gal_type_max(GAL_TYPE_INT16, &max);
```

Note: Do not use the maximum value for a blank value of a general (initially unknown) type, please use the constants/functions provided in Section 11.3.5 [Library blank values (`blank.h`)], page 343, for the definition and usage of blank values.

`int` [Function]
`gal_type_is_int` (*uint8_t* `type`)

Return 1 if the type is an integer (any width and any sign).

`int` [Function]
`gal_type_is_list` (*uint8_t* `type`)

Return 1 if the type is a linked list and zero otherwise.

`int` [Function]
`gal_type_out` (*int* `first_type`, *int* `second_type`)

Return the larger of the two given types which can be used for the type of the output of an operation involving the two input types.

`char *` [Function]
`gal_type_bit_string` (*void* `*in`, *size_t* `size`)

Return the bit-string in the `size` bytes that `in` points to. The string is dynamically allocated and must be freed afterwards. You can use it to inspect the bits within one region of memory. Here is one short example:

```
int32_t a=2017;
char *bitstr=gal_type_bit_string(&a, 4);
printf("%d: %s (%X)\n", a, bitstr, a);
free(bitstr);
```

which will produce:

```
2017: 11100001000001110000000000000000  (7E1)
```

As the example above shows, the bit-string is not the most efficient way to inspect bits. If you are familiar with hexadecimal notation, it is much more compact, see `https://en.wikipedia.org/wiki/Hexadecimal`. You can use `printf`'s `%x` or `%X` to print integers in hexadecimal format.

`char *` [Function]
`gal_type_to_string` (*void* `*ptr`, *uint8_t* `type`, *int* `quote_if_str_has_space`);

Read the contents of the memory that `ptr` points to (assuming it has type `type` and print it into an allocated string which is returned.

If the memory is a string of characters and `quote_if_str_has_space` is non-zero, the output string will have double-quotes around it if it contains space characters. Also, note that in this case, `ptr` must be a pointer to an array of characters (or `char **`), as in the example below (which will put `"sample string"` into `out`):

```
char *out, *string="sample string"
out = gal_type_to_string(&string, GAL_TYPE_STRING, 1);
```

`int` [Function]
`gal_type_from_string` (*void* `**out`, *char* `*string`, *uint8_t* `type`)

Read a string as a given data type and put a the pointer to it in `*out`. When `*out!=NULL`, then it is assumed to be already allocated and the value will be simply put the memory. If `*out==NULL`, then space will be allocated for the given type and the string will be read into that type.

Note that when we are dealing with a string type, `*out` should be interpreted as `char **` (one element in an array of pointers to different strings). In other words, `out` should be `char ***`.

This function can be used to fill in arrays of numbers from strings (in an already allocated data structure), or add nodes to a linked list (if the type is a list type). For an array, you have to pass the pointer to the `i`th element where you want the value to be stored, for example `&(array[i])`.

If the string was successfully parsed to the requested type, this function will return a `0` (zero), otherwise it will return `1` (one). This output format will help you check the status of the conversion in a code like the example below:

```
if( gal_type_from_string(&out, string, GAL_TYPE_FLOAT32) )
  {
    fprintf(stderr, "%s couldn't be read as float32.\n", string);
    exit(EXIT_FAILURE);
  }
```

`void *` [Function]
`gal_type_string_to_number` (*char* `*string`, *uint8_t* `*type`)

Read `string` into smallest type that can host the number, the allocated space for the number will be returned and the type of the number will be put into the memory that `type` points to. If `string` couldn't be read as a number, this function will return `NULL`.

For the ranges acceptable by each type see Section 4.5 [Numeric data types], page 115. For integers it is clear, for floating point types, this function will count the number of significant digits and determine if the given string is single or double precision as described in that section.

11.3.4 Pointers (`pointer.h`)

Pointers play an important role in the C programming language. As the name suggests, they *point* to a byte in memory (like an address in a city). The C programming language

gives you complete freedom in how to use the byte (and the bytes that follow it). Pointers are thus a very powerful feature of C. However, as the saying goes: "With great power comes great responsibility", so they must be approached with care. The functions in this header are not very complex, they are just wrappers over some basic pointer functionality regarding pointer arithmetic and allocation (in memory or HDD/SSD).

`void *` [Function]
`gal_pointer_increment` (*void* `*pointer`, *size_t* `increment`, *uint8_t* `type`)

Return a pointer to an element that is `increment` elements ahead of `pointer`, assuming each element has type of `type`. For the type codes, see Section 11.3.3 [Library data types (`type.h`)], page 337.

When working with the `array` elements of `gal_data_t`, we are actually dealing with `void *` pointers. However, pointer arithmetic doesn't apply to `void *`, because the system doesn't know how many bytes there are in each element to increment the pointer respectively. This function will use the given `type` to calculate where the incremented element is located in memory.

`size_t` [Function]
`gal_pointer_num_between` (*void* `*earlier`, *void* `*later`, *uint8_t* `type`)

Return the number of elements (in the given `type`) between `earlier` and `later`. For the type codes, see Section 11.3.3 [Library data types (`type.h`)], page 337).

`void *` [Function]
`gal_pointer_allocate` (*uint8_t* `type`, *size_t* `size`, *int* `clear`, *const char* `*funcname`, *const char* `*varname`)

Allocate an array of type `type` with `size` elements in RAM (for the type codes, see Section 11.3.3 [Library data types (`type.h`)], page 337). If `clear!=0`, then the allocated space is set to zero (cleared). This is effectively just a wrapper around C's `malloc` or `calloc` functions but takes Gnuastro's integer type codes and will also abort with a clear error if there the allocation was not successful.

When space cannot be allocated, this function will abort the program with a message containing the reason for the failure. `funcname` (name of the function calling this function) and `varname` (name of variable that needs this space) will be used in this error message if they are not `NULL`. In most modern compilers, you can use the generic `__func__` variable for `funcname`. In this way, you don't have to manually copy and paste the function name or worry about it changing later (`__func__` was standardized in C99).

`void *` [Function]
`gal_pointer_allocate_mmap` (*size_t* `size`, *uint8_t* `type`, *int* `clear`, *char* `**mmapname`)

Allocate the necessary space to keep `size` elements of type `type` in HDD/SSD (a file, not in RAM). for the type codes, see Section 11.3.3 [Library data types (`type.h`)], page 337. If `clear!=0`, then the allocated space will also be cleared. The allocation is done using C's `mmap` function. The name of the file containing the allocated space is an allocated string that will be put in `*mmapname`.

Note that the kernel doesn't allow an infinite number of memory mappings to files. So it is not recommended to use this function with every allocation. The best case

scenario to use this function is for large arrays that are very large and can fill up the RAM. Keep the smaller arrays in RAM, which is faster and can have a (theoretically) unlimited number of allocations.

When you are done with the dataset and don't need it anymore, don't use `free` (the dataset isn't in RAM). Just delete the file (and the allocated space for the filename) with the commands below:

```
remove(mmapname);
free(mmapname);
```

11.3.5 Library blank values (`blank.h`)

When the position of an element in a dataset is important (for example a pixel in an image), a place-holder is necessary for the element if we don't have a value to fill it with (for example the CCD cannot read those pixels). We cannot simply shift all the other pixels to fill in the one we have no value for. In other cases, it often occurs that the field of sky that you are studying is not a clean rectangle to nicely fit into the boundaries of an image. You need a way to separate the pixels outside your scientific field from those inside it. Blank values act as these place holders in a dataset. They have no usable value but they have a position.

Every type needs a corresponding blank value (see Section 4.5 [Numeric data types], page 115, and Section 11.3.3 [Library data types (`type.h`)], page 337). Floating point types have a unique value identified by IEEE known as Not-a-Number (or NaN) which is a unique value that is recognized by the compiler. However, integer and string types don't have any standard value. For integers, in Gnuastro we take an extremum of the given type: for signed types (that allow negatives), the minimum possible value is used as blank and for unsigned types (that only accept positives), the maximum possible value is used. To be generic and easy to read/write we define a macro for these blank values and strongly encourage you only use these, and never make any assumption on the value of a type's blank value.

The IEEE NaN blank value type is defined to fail on any comparison, so if you are dealing with floating point types, you cannot use equality (a NaN will *not* be equal to a NaN). If you know your dataset if floating point, you can use the `isnan` function in C's `math.h` header. For a description of numeric data types see Section 4.5 [Numeric data types], page 115. For the constants identifying integers, please see Section 11.3.3 [Library data types (`type.h`)], page 337.

`GAL_BLANK_UINT8` [Global integer]
Blank value for an unsigned, 8-bit integer.

`GAL_BLANK_INT8` [Global integer]
Blank value for a signed, 8-bit integer.

`GAL_BLANK_UINT16` [Global integer]
Blank value for an unsigned, 16-bit integer.

`GAL_BLANK_INT16` [Global integer]
Blank value for a signed, 16-bit integer.

`GAL_BLANK_UINT32` [Global integer]
Blank value for an unsigned, 32-bit integer.

`GAL_BLANK_INT32` [Global integer]
Blank value for a signed, 32-bit integer.

`GAL_BLANK_UINT64` [Global integer]
Blank value for an unsigned, 64-bit integer.

`GAL_BLANK_INT64` [Global integer]
Blank value for a signed, 64-bit integer.

`GAL_BLANK_LONG` [Global integer]
Blank value for `long` type (`int32_t` or `int64_t` in 32-bit or 64-bit systems).

`GAL_BLANK_ULONG` [Global integer]
Blank value for `unsigned long` type (`uint32_t` or `uint64_t` in 32-bit or 64-bit systems).

`GAL_BLANK_SIZE_T` [Global integer]
Blank value for `size_t` type (`uint32_t` or `uint64_t` in 32-bit or 64-bit systems).

`GAL_BLANK_FLOAT32` [Global integer]
Blank value for a single precision, 32-bit floating point type (IEEE NaN value).

`GAL_BLANK_FLOAT64` [Global integer]
Blank value for a double precision, 64-bit floating point type (IEEE NaN value).

`GAL_BLANK_STRING` [Global integer]
Blank value for string types (this is itself a string, it isn't the `NULL` pointer).

The functions below can be used to work with blank pixels.

`void` [Function]
`gal_blank_write` (*void* `*pointer`, *uint8_t* `type`)
Write the blank value for the given `type` into the space that `pointer` points to. This can be used when the space is already allocated (for example one element in an array or a statically allocated variable).

`void *` [Function]
`gal_blank_alloc_write` (*uint8_t* `type`)
Allocate the space required to keep the blank for the given data type `type`, write the blank value into it and return the pointer to it.

`void` [Function]
`gal_blank_initialize` (*gal_data_t* `*input`)
Initialize all the elements in the `input` dataset to the blank value that corresponds to its type. If `input` is a tile over a larger dataset, only the region that the tile covers will be set to blank.

`char *` [Function]
`gal_blank_as_string` (*uint8_t* `type`, *int* `width`)
Write the blank value for the given data type `type` into a string and return it. The space for the string is dynamically allocated so it must be freed after you are done with it. If `width!=0`, then the final string will be padded with white space characters to have the requested width if it is smaller.

`int` [Function]
`gal_blank_is` (*void* `*pointer`, *uint8_t* `type`)

Return 1 if the contents of `pointer` (assuming a type of `type`) is blank. Otherwise, return 0. Note that this function only works on one element of the given type. So if `pointer` is an array, only its first element will be checked. Therefore for strings, the type of `pointer` is assumed to be `char *`. To check if an array/dataset has blank elements or to find which elements in an array are blank, you can use `gal_blank_present` or `gal_blank_flag` respectively (described below).

`int` [Function]
`gal_blank_present` (*gal_data_t* `*input`, *int* `updateflag`)

Return 1 if the dataset has a blank value and zero if it doesn't. Before checking the dataset, this function will look at `input`'s flags. If the `GAL_DATA_FLAG_BLANK_CH` bit of `input->flag` is on, this function will not do any check and will just use the information in the flags. This can greatly speed up processing when a dataset needs to be checked multiple times.

When the dataset's flags were not used and `updateflags` is non-zero, this function will set the flags appropriately to avoid having to re-check the dataset in future calls. When `updateflags==0`, this function has no side-effects on the dataset: it will not toggle the flags.

If you want to re-check a dataset with the blank-value-check flag already set (for example if you have made changes to it), then explicitly set the `GAL_DATA_FLAG_BLANK_CH` bit to zero before calling this function. When there are no other flags, you can just set the flags to zero (`input->flags=0`), otherwise you can use this expression:

```
input->flags &= ~GAL_DATA_FLAG_BLANK_CH;
```

`size_t` [Function]
`gal_blank_number` (*gal_data_t* `*input`, *int* `updateflag`)

Return the number of blank elements in `input`. If `updateflag!=0`, then the dataset blank keyword flags will be updated. See the description of `gal_blank_present` (above) for more on these flags. If `input==NULL`, then this function will return `GAL_BLANK_SIZE_T`.

`gal_data_t *` [Function]
`gal_blank_flag` (*gal_data_t* `*input`)

Create a dataset of the the same size as the input, but with an `uint8_t` type that has a value of 1 for data that are blank and 0 for those that aren't.

`void` [Function]
`gal_blank_flag_apply` (*gal_data_t* `*input`, *gal_data_t* `*flag`)

Set all non-zero and non-blank elements of `flag` to blank in `input`. `flag` has to have an unsigned 8-bit type and be the same size as `input`.

`void` [Function]
`gal_blank_remove` (*gal_data_t* `*input`)

Remove blank elements from a dataset, convert it to a 1D dataset, adjust the size properly (the number of non-blank elements), and toggle the blank-value-related bit-flags. In practice this function doesn't `realloc` the input array, it just shifts the

blank elements to the end and adjusts the size elements of the `gal_data_t`, see Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346.

If all the elements were blank, then `input->size` will be zero. This is thus a good parameter to check after calling this function to see if there actually were any non-blank elements in the input or not and take the appropriate measure. This check is highly recommended because it will avoid strange bugs in later steps.

11.3.6 Data container (`data.h`)

Astronomical datasets have various dimensions, for example 1D spectra or table columns, 2D images, or 3D Integral field data cubes. Datasets can also have various numeric data types, depending on the operation/purpose, for example processed images are commonly stored in floating point format, but their mask images are integers (allowing bit-wise flags to identify certain classes of pixels to keep or mask, see Section 4.5 [Numeric data types], page 115). Certain other information about a dataset are also commonly necessary, for example the units of the dataset, the name of the dataset and some comments. To deal with any generic dataset, Gnuastro defines the `gal_data_t` as input or output.

11.3.6.1 Generic data container (`gal_data_t`)

To be able to deal with any dataset (various dimensions, numeric data types, units and higher-level structures), Gnuastro defines the `gal_data_t` type which is the input/output container of choice for many of Gnuastro library's functions. It is defined in `gnuastro/data.h`. If you will be using ('`# include`'ing) those libraries, you don't need to include this header explicitly, it is already included by any library header that uses `gal_data_t`.

`gal_data_t` [Type (C `struct`)]

The main container for datasets in Gnuastro. It can host data of any dimensions, with any numeric data type. It is actually a structure, but `typedef`'d as a new type to avoid having to write the `struct` before any declaration. The actual structure is shown below which is followed by a description of each element.

```
typedef struct gal_data_t
{
  void     *restrict array;  /* Basic array information.   */
  uint8_t             type;
  size_t              ndim;
  size_t             *dsize;
  size_t              size;
  char           *mmapname;
  size_t        minmapsize;

  int                 nwcs;  /* WCS information.           */
  struct wcsprm       *wcs;

  uint8_t             flag;  /* Content description.       */
  int               status;
  char               *name;
```

```
  char               *unit;
  char             *comment;

  int              disp_fmt;  /* For text printing.         */
  int            disp_width;
  int        disp_precision;

  struct gal_data_t  *next;  /* For higher-level datasets. */
  struct gal_data_t *block;
} gal_data_t;
```

The list below contains a description for each `gal_data_t` element.

`void *restrict array`
: This is the pointer to the main array of the dataset containing the raw data (values). All the other elements in this data-structure are actually meta-data enabling us to use/understand the series of values in this array. It must allow data of any type (see Section 4.5 [Numeric data types], page 115), so it is defined as a `void *` pointer. A `void *` array is not directly usable in C, so you have to cast it to proper type before using it, please see Section 11.4.1 [Library demo - reading a FITS image], page 437, for a demonstration.

  The `restrict` keyword was formally introduced in C99 and is used to tell the compiler that at any moment only this pointer will modify what it points to (a pixel in an image for example)[15]. This extra piece of information can greatly help in compiler optimizations and thus the running time of the program. But older compilers might not have this capability, so at `./configure` time, Gnuastro checks this feature and if the user's compiler doesn't support `restrict`, it will be removed from this definition.

`uint8_t type`
: A fixed code (integer) used to identify the type of data in `array` (see Section 4.5 [Numeric data types], page 115). For the list of acceptable values to this variable, please see Section 11.3.3 [Library data types (`type.h`)], page 337.

`size_t ndim`
: The dataset's number of dimensions.

`size_t *dsize`
: The size of the dataset along each dimension. This is an array (with `ndim` elements), of positive integers in row-major order[16] (based on C). When a data file is read into memory with Gnuastro's libraries, this array is dynamically allocated based on the number of dimensions that the dataset has.

  It is important to remember that C's row-major ordering is the opposite of the FITS standard which is in column-major order: in the FITS standard the fastest dimension's size is specified by `NAXIS1`, and slower dimensions follow. The FITS standard was defined mainly based on the FORTRAN language which is the

[15] Also see `https://en.wikipedia.org/wiki/Restrict`.

[16] Also see `https://en.wikipedia.org/wiki/Row-_and_column-major_order`.

opposite of C's approach to multi-dimensional arrays (and also starts counting from 1 not 0). Hence if a FITS image has `NAXIS1==20` and `NAXIS2==50`, the `dsize` array must be filled with `dsize[0]==50` and `dsize[1]==20`.

The fastest dimension is the one that is contiguous in memory: to increment by one along that dimension, just go to the next element in the array. As we go to slower dimensions, the number of memory cells we have to skip for an increment along that dimension becomes larger.

`size_t size`

The total number of elements in the dataset. This is actually a multiplication of all the values in the `dsize` array, so it is not an independent parameter. However, low-level operations with the dataset (irrespective of its dimensions) commonly need this number, so this element is designed to avoid calculating it every time.

`char *mmapname`

Name of file hosting the `mmap`'d contents of `array`. If the value of this variable is `NULL`, then the contents of `array` are actually stored in RAM, not in a file on the HDD/SSD. See the description of `minmapsize` below for more.

If a file is used, it will be kept in the hidden `.gnuastro`, or `.gnuastro_mmap` directories with a randomly selected name to allow multiple arrays to be kept there at the same time, see description of `--minmapsize` in Section 4.1.2.2 [Processing options], page 98. When `gal_data_free` is called the randomly named file will be deleted.

`size_t minmapsize`

The minimum size of an array (in bytes) to store the contents of `array` as a file (on the non-volatile HDD/SSD), not in RAM. This can be very useful for large datasets which can be very memory intensive and the user's RAM might not be sufficient to keep/process it. A random filename is assigned to the array which is available in the `mmapname` element of `gal_data_t` (above), see there for more. `minmapsize` is stored in each `gal_data_t`, so it can be passed on to subsequent/derived datasets.

See the description of the `--minmapsize` option in Section 4.1.2.2 [Processing options], page 98, for more on using this value.

`nwcs` The number of WCS coordinate representations (for WCSLIB).

`struct wcsprm *wcs`

The main WCSLIB structure keeping all the relevant information necessary for WCSLIB to do its processing and convert data-set positions into real-world positions. When it is given a `NULL` value, all possible WCS calculations/measurements will be ignored.

`uint8_t flag`

Bit-wise flags to describe general properties of the dataset. The number of bytes available in this flag is stored in the `GAL_DATA_FLAG_SIZE` macro. Note that you should use bit-wise operators[17] to check these flags. The currently recognized bits are stored in these macros:

[17] See `https://en.wikipedia.org/wiki/Bitwise_operations_in_C`.

`GAL_DATA_FLAG_BLANK_CH`
: Marking that the dataset has been checked for blank values or not. When a dataset doesn't have any blank values, the `GAL_DATA_FLAG_HASBLANK` bit will be zero. But upon initialization, all bits also get a value of zero. Therefore, a checker needs this flag to see if the value in `GAL_DATA_FLAG_HASBLANK` is reliable (dataset has actually been parsed for a blank value) or not.

  Also, if it is necessary to re-check the presence of flags, you just have to set this flag to zero and call `gal_blank_present` for example to parse the dataset and check for blank values. Note that for improved efficiency, when this flag is set, `gal_blank_present` will not actually parse the dataset, it will just use `GAL_DATA_FLAG_HASBLANK`.

`GAL_DATA_FLAG_HASBLANK`
: This bit has a value of `1` when the given dataset has blank values. If this bit is `0` and `GAL_DATA_FLAG_BLANK_CH` is `1`, then the dataset has been checked and it didn't have any blank values, so there is no more need for further checks.

`GAL_DATA_FLAG_SORT_CH`
: Marking that the dataset is already checked for being sorted or not and thus that the possible `0` values in `GAL_DATA_FLAG_SORTED_I` and `GAL_DATA_FLAG_SORTED_D` are meaningful. The logic behind this is similar to that in `GAL_DATA_FLAG_BLANK_CH`.

`GAL_DATA_FLAG_SORTED_I`
: This bit has a value of `1` when the given dataset is sorted in an increasing manner. If this bit is `0` and `GAL_DATA_FLAG_SORT_CH` is `1`, then the dataset has been checked and wasn't sorted (increasing), so there is no more need for further checks.

`GAL_DATA_FLAG_SORTED_D`
: This bit has a value of `1` when the given dataset is sorted in a decreasing manner. If this bit is `0` and `GAL_DATA_FLAG_SORT_CH` is `1`, then the dataset has been checked and wasn't sorted (decreasing), so there is no more need for further checks.

The macro `GAL_DATA_FLAG_MAXFLAG` contains the largest internally used bit-position. Higher-level flags can be defined with the bit-wise shift operators using this macro to define internal flags for libraries/programs that depend on Gnuastro without causing any possible conflict with the internal flags discussed above or having to check the values manually on every release.

`int status`
: A context-specific status values for this data-structure. This integer will not be set by Gnuastro's libraries. You can use it keep some additional information about the dataset (with integer constants) depending on your applications.

`char *name`
: The name of the dataset. If the dataset is a multi-dimensional array and read/written as a FITS image, this will be the value in the `EXTNAME` FITS keyword. If the dataset is a one-dimensional table column, this will be the column name. If it is set to `NULL` (by default), it will be ignored.

`char *unit`
: The units of the dataset (for example `BUNIT` in the standard FITS keywords) that will be read from or written to files/tables along with the dataset. If it is set to `NULL` (by default), it will be ignored.

`char *comment`
: Any further explanation about the dataset which will be written to any output file if present.

`disp_fmt` Format to use for printing each element of the dataset to a plain text file, the acceptable values to this element are defined in Section 11.3.10 [Table input output (`table.h`)], page 370. Based on C's `printf` standards.

`disp_width`
: Width of printing each element of the dataset to a plain text file, the acceptable values to this element are defined in Section 11.3.10 [Table input output (`table.h`)], page 370. Based on C's `printf` standards.

`disp_precision`
: Precision of printing each element of the dataset to a plain text file, the acceptable values to this element are defined in Section 11.3.10 [Table input output (`table.h`)], page 370. Based on C's `printf` standards.

`gal_data_t *next`
: Through this pointer, you can link a `gal_data_t` with other datasets related datasets, for example the different columns in a dataset each have one `gal_data_t` associate with them and they are linked to each other using this element. There are several functions described below to facilitate using `gal_data_t` as a linked list. See Section 11.3.8 [Linked lists (`list.h`)], page 357, for more on these wonderful high-level constructs.

`gal_data_t *block`
: Pointer to the start of the complete allocated block of memory. When this pointer is not `NULL`, the dataset is not treated as a contiguous patch of memory. Rather, it is seen as covering only a portion of the larger patch of memory that `block` points to. See Section 11.3.15 [Tessellation library (`tile.h`)], page 399, for a more thorough explanation and functions to help work with tiles that are created from this pointer.

11.3.6.2 Dataset allocation

Gnuastro's main data container was defined in Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346. The functions listed in this section describe the most basic operations on `gal_data_t`: those related to allocation and freeing. These functions are declared in `gnuastro/data.h` which is also visible from the function names (see Section 11.3 [Gnuastro library], page 332).

`gal_data_t *` [Function]
`gal_data_alloc` (*void* `*array`, *uint8_t* `type`, *size_t* `ndim`, *size_t* `*dsize`, *struct wcsprm* `*wcs`, *int* `clear`, *size_t* `minmapsize`, *char* `*name`, *char* `*unit`, *char* `*comment`)

Dynamically allocate a `gal_data_t` and initialize it will all the given values. See the description of `gal_data_initialize` and Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346, for more information. This function will often be the most frequently used because it allocates the `gal_data_t` hosting all the values *and* initializes it. Once you are done with the dataset, be sure to clean up all the allocated spaces with `gal_data_free`.

`void` [Function]
`gal_data_initialize` (*gal_data_t* `*data`, *void* `*array`, *uint8_t* `type`, *size_t* `ndim`, *size_t* `*dsize`, *struct wcsprm* `*wcs`, *int* `clear`, *size_t* `minmapsize`, *char* `*name`, *char* `*unit`, *char* `*comment`)

Initialize the given data structure (`data`) with all the given values. Note that the raw input `gal_data_t` must already have been allocated before calling this function. For a description of each variable see Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346. It will set the values and do the necessary allocations. If they aren't `NULL`, all input arrays (`dsize`, `wcs`, `name`, `unit`, `comment`) are separately copied (allocated) by this function for usage in `data`, so you can safely use one value to initialize many datasets or use statically allocated variables in this function call. Once you are done with the dataset, you can free all the allocated spaces with `gal_data_free_contents`.

If `array` is not `NULL`, it will be directly copied into `data->array` (based on the total number of elements calculated from `dsize`) and no new space will be allocated for the array of this dataset, this has many low-level advantages and can be used to work on regions of a dataset instead of the whole allocated array (see the description under `block` in Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346, for one example). If the given pointer is not the start of an allocated block of memory or it is used in multiple datasets, be sure to set it to `NULL` (with `data->array=NULL`) before cleaning up with `gal_data_free_contents`.

`ndim` may be zero. In this case no allocation will occur, `data->array` and `data->dsize` will be set to `NULL` and `data->size` will be zero. However (when necessary) `dsize` must not have any zero values (a dimension of length zero is not defined).

`void` [Function]
`gal_data_free_contents` (*gal_data_t* `*data`)

Free all the non-`NULL` pointers in `gal_data_t` except for `next` and `block`. If `data` is actually a tile (`data->block!=NULL`, see Section 11.3.15 [Tessellation library (`tile.h`)], page 399), then `data->array` is not freed. For a complete description of `gal_data_t` and its contents, see Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346.

`void` [Function]
`gal_data_free` (*gal_data_t* `*data`)

Free all the non-`NULL` pointers in `gal_data_t`, then free the actual data structure.

11.3.6.3 Arrays of datasets

Gnuastro's generic data container (`gal_data_t`) is a very versatile structure that can be used in many higher-level contexts. One such higher-level construct is an array of `gal_data_t` structures to simplify the allocation (and later cleaning) of several `gal_data_t`s that are related.

For example, each column in a table is usually represented by one `gal_data_t` (so it has its own name, data type, units and etc). A table (with many columns) can be seen as an array of `gal_data_t`s (when the number of columns is known a-priori). The functions below are defined to create a cleared array of data structures and to free them when none are necessary any more. These functions are declared in `gnuastro/data.h` which is also visible from the function names (see Section 11.3 [Gnuastro library], page 332).

`gal_data_t *` [Function]
`gal_data_array_calloc` (*size_t* `size`)

Allocate an array of `gal_data_t` with `size` elements. This function will also initialize all the values (`NULL` for pointers and 0 for other types). You can use `gal_data_initialize` to fill each element of the array afterwards. The following code snippet is one example of doing this.

```
size_t i;
gal_data_t *dataarr;
dataarr=gal_data_array_calloc(10);
for(i=0;i<10;++i) gal_data_initialize(&dataarr[i], ...);
...
gal_data_array_free(dataarr, 10, 1);
```

`void` [Function]
`gal_data_array_free` (*gal_data_t* `*dataarr`, *size_t* `num`, *int* `free_array`)

Free all the `num` elements within `dataarr` and the actual allocated array. If `free_array` is not zero, then the `array` element of all the datasets will also be freed, see Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346.

11.3.6.4 Copying datasets

The functions in this section describes Gnuastro's facilities to copy a given dataset into another. The new dataset can have a different type (including a string), it can be already allocated (in which case only the values will be written into it). In all these cases, if the input dataset is a tile or a list, only the data within the given tile, or the given node in a list, are copied. If the input is a list, the `next` pointer will also be copied to the output, see Section 11.3.8.9 [List of `gal_data_t`], page 368.

In many of the functions here, it is possible to copy the dataset to a new numeric data type (see Section 4.5 [Numeric data types], page 115. In such cases, Gnuastro's library is going to use the native conversion by C. So if you are converting to a smaller type, it is up to you to make sure that the values fit into the output type.

`gal_data_t *` [Function]
`gal_data_copy` (*gal_data_t* `*in`)

Return a new dataset that is a copy of `in`, all of `in`'s meta-data will also copied into the output, except for `block`. If the dataset is a tile/list, only the given tile/node will be copied, the `next` pointer will also be copied however.

`gal_data_t *` [Function]
`gal_data_copy_to_new_type` (*gal_data_t* `*in`, *uint8_t* `newtype`)

Return a copy of the dataset `in`, converted to `newtype`, see Section 11.3.3 [Library data types (`type.h`)], page 337, for Gnuastro library's type identifiers. The returned dataset will have all meta-data except their type and `block` equal to the input's metadata. If the dataset is a tile/list, only the given tile/node will be copied, the `next` pointer will also be copied however.

`gal_data_t *` [Function]
`gal_data_copy_to_new_type_free` (*gal_data_t* `*in`, *uint8_t* `newtype`)

Return a copy of the dataset `in` that is converted to `newtype` and free the input dataset. See Section 11.3.3 [Library data types (`type.h`)], page 337, for Gnuastro library's type identifiers. The returned dataset will have all meta-data, except their type, equal to the input's metadata (including `next`). Note that if the input is a tile within a larger block, it will not be freed. This function is similar to `gal_data_copy_to_new_type`, except that it will free the input dataset.

`void` [Function]
`gal_data_copy_to_allocated` (*gal_data_t* `*in`, *gal_data_t* `*out`)

Copy the contents of the array in `in` into the already allocated array in `out`. The types of the input and output may be different, type conversion will be done internally. When `in->size != out->size` this function will behave as follows:

`out->size < in->size`
: This function won't re-allocate the necessary space, it will abort with an error, so please check before calling this function.

`out->size > in->size`
: This function will write the values in `out->size` and `out->dsize` from the same values of `in`. So if you want to use a pre-allocated space/dataset multiple times with varying input sizes, be sure to reset `out->size` before every call to this function.

`gal_data_t *` [Function]
`gal_data_copy_string_to_number` (*char* `*string`)

Read `string` into the smallest type that can store the value (see Section 4.5 [Numeric data types], page 115). This function is just a wrapper for the `gal_type_string_to_number`, but will put the value into a single-element dataset.

11.3.7 Dimensions (`dimension.h`)

An array is a contiguous region of memory. Hence, at the lowest level, every element of an array just has one single-valued position: the number of elements that lie between it and the first element in the array. This is also known as the *index* of the element within

the array. A dataset's number of dimensions is high-level abstraction (meta-data) that we project onto that contiguous patch of memory. When the array is interpreted as a one-dimensional dataset, this index is also the *coordinate* of the element. But once we associate the patch of memory with a higher dimension, there must also be one coordinate for each dimension.

The functions and macros in this section provide you with the tools to convert an index into a coordinate and vice-versa along with several other issues for example issues with the neighbors of an element in a multi-dimensional context.

`size_t` [Function]
`gal_dimension_total_size` (*size_t* `ndim`, *size_t* `*dsize`)
Return the total number of elements for a dataset with `ndim` dimensions that has `dsize` elements along each dimension.

`int` [Function]
`gal_dimension_is_different` (*gal_data_t* `*first`, *gal_data_t* `*second`)
Return `1` (one) if the two datasets don't have the same size along all dimensions. This function will also return `1` when the number of dimensions of the two datasets are different.

`size_t *` [Function]
`gal_dimension_increment` (*size_t* `ndim`, *size_t* `*dsize`)
Return an allocated array that has the number of elements necessary to increment an index along every dimension. For example along the fastest dimension (last element in the `dsize` and returned arrays), the value is `1` (one).

`size_t` [Function]
`gal_dimension_num_neighbors` (*size_t* `ndim`)
The maximum number of neighbors (any connectivity) that a data element can have in `ndim` dimensions. Effectively, this function just returns $3^n - 1$ (where n is the number of dimensions).

`GAL_DIMENSION_FLT_TO_INT (FLT)` [Function-like macro]
Calculate the integer pixel position that the floating point `FLT` number belongs to. In the FITS format (and thus in Gnuastro), the center of each pixel is allocated on an integer (not it edge), so the pixel which hosts a floating point number cannot simply be found with internal type conversion.

`void` [Function]
`gal_dimension_add_coords` (*size_t* `*c1`, *size_t* `*c2`, *size_t* `*out`, *size_t* `ndim`)
For every dimension, add the coordinates in `c1` with `c2` and put the result into `out`. In other words, for dimension `i` run `out[i]=c1[i]+c2[i];`. Hence `out` may be equal to any one of `c1` or `c2`.

`size_t` [Function]
`gal_dimension_coord_to_index` (*size_t* `ndim`, *size_t* `*dsize`, *size_t* `*coord`)
Return the index (counting from zero) from the coordinates in `coord` (counting from zero) assuming the dataset has `ndim` elements and the size of the dataset along each dimension is in the `dsize` array.

`void` [Function]
`gal_dimension_index_to_coord` (*size_t* `index`, *size_t* `ndim`, *size_t* `*dsize`, *size_t* `*coord`)

Fill in the `coord` array with the coordinates that correspond to `index` assuming the dataset has `ndim` elements and the size of the dataset along each dimension is in the `dsize` array. Note that both `index` and each value in `coord` are assumed to start from `0` (zero). Also that the space which `coord` points to must already be allocated before calling this function.

`size_t` [Function]
`gal_dimension_dist_manhattan` (*size_t* `*a`, *size_t* `*b`, *size_t* `ndim`)

Return the manhattan distance (see Wikipedia (`https://en.wikipedia.org/wiki/Taxicab_geometry`)) between the two coordinates `a` and `b` (each an array of `ndim` elements).

`float` [Function]
`gal_dimension_dist_radial` (*size_t* `*a`, *size_t* `*b`, *size_t* `ndim`)

Return the radial distance between the two coordinates `a` and `b` (each an array of `ndim` elements).

`gal_data_t *` [Function]
`gal_dimension_collapse_sum` (*gal_data_t* `*in`, *size_t* `c_dim`, *gal_data_t* `*weight`)

Collapse the input dataset (`in`) along the given dimension (`c_dim`, in C definition: starting from zero, from the slowest dimension), by summing all elements in that direction. If `weight!=NULL`, it must be a single-dimensional array, with the same size as the dimension to be collapsed. The respective weight will be multiplied to each element during the collapse.

For generality, the returned dataset will have a `GAL_TYPE_FLOAT64` type. See Section 11.3.6.4 [Copying datasets], page 352, for converting the returned dataset to a desired type. Also, for more on the application of this function, see the Arithmetic program's `collapse-sum` operator (which uses this function) in Section 6.2.2 [Arithmetic operators], page 162.

`gal_data_t *` [Function]
`gal_dimension_collapse_mean` (*gal_data_t* `*in`, *size_t* `c_dim`, *gal_data_t* `*weight`)

Similar to `gal_dimension_collapse_sum` (above), but the collapse will be done by calculating the mean along the requested dimension, not summing over it.

`gal_data_t *` [Function]
`gal_dimension_collapse_number` (*gal_data_t* `*in`, *size_t* `c_dim`)

Collapse the input dataset (`in`) along the given dimension (`c_dim`, in C definition: starting from zero, from the slowest dimension), by counting how many non-blank elements there are along that dimension.

For generality, the returned dataset will have a `GAL_TYPE_INT32` type. See Section 11.3.6.4 [Copying datasets], page 352, for converting the returned dataset to a desired type. Also, for more on the application of this function, see the Arithmetic program's `collapse-number` operator (which uses this function) in Section 6.2.2 [Arithmetic operators], page 162.

`gal_data_t *` [Function]
`gal_dimension_collapse_minmax` (*gal_data_t* `*in`, *size_t* `c_dim`, *int* `max1_min0`)

Collapse the input dataset (`in`) along the given dimension (`c_dim`, in C definition: starting from zero, from the slowest dimension), by using the largest/smallest non-blank value along that dimension. If `max1_min0` is non-zero, then the collapsed dataset will have the maximum value along the given dimension and if it is zero, the minimum.

`GAL_DIMENSION_NEIGHBOR_OP (index, ndim, dsize,` [Function-like macro]
`connectivity, dinc, operation)`

Parse the neighbors of the element located at `index` and do the requested operation on them. This is defined as a macro to allow easy definition of any operation on the neighbors of a given element without having to use loops within your source code (the loops are implemented by this macro). For an example of using this function, please see Section 11.4.2 [Library demo - inspecting neighbors], page 438. The input arguments to this function-like macro are described below:

`index` Distance of this element from the first element in the array on a contiguous patch of memory (starting from 0), see the discussion above.

`ndim` The number of dimensions associated with the contiguous patch of memory.

`dsize` The full array size along each dimension. This must be an array and is assumed to have the same number elements as `ndim`. See the discussion under the same element in Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346.

`connectivity`
Most distant neighbors to consider. Depending on the number of dimensions, different neighbors may be defined for each element. This function-like macro distinguish between these different neighbors with this argument. It has a value between `1` (one) and `ndim`. For example in a 2D dataset, 4-connected neighbors have a connectivity of `1` and 8-connected neighbors have a connectivity of `2`. Note that this is inclusive, so in this example, a connectivity of `2` will also include connectivity `1` neighbors.

`dinc` An array keeping the length necessary to increment along each dimension. You can make this array with the following function. Just don't forget to free the array after you are done with it:

```
size_t *dinc=gal_dimension_increment(ndim, dsize);
```

`dinc` depends on `ndim` and `dsize`, but it must be defined outside this function-like macro since it involves allocation to help in performance.

`operation`
Any C operation that you would like to do on the neighbor. This macro will provide you a `nind` variable that can be used as the index of the neighbor that is currently being studied. It is defined as '`size_t ndim;`'. Note that `operation` will be repeated the number of times there is a neighbor for this element.

This macro works fully within its own `{}` block and except for the `nind` variable that shows the neighbor's index, all the variables within this macro's block start with `gdn_`.

11.3.8 Linked lists (`list.h`)

An array is a contiguous region of memory that is very efficient and easy to use for recording and later accessing any random element as fast as any other. This makes array the primary data container when you have many elements (for example an image which has millions of pixels). One major problem with an array is that the number of elements that go into it must be known in advance and adding or removing an element will require a re-set of all the other elements. For example if you want to remove the 3rd element in a 1000 element array, all 997 subsequent elements have to pulled back by one position, the reverse will happen if you need to add an element.

In many contexts such situations never come up, for example you don't want to shift all the pixels in an image by one or two pixels from some random position in the image: their positions have scientific value. But in other contexts you will find your self frequently adding/removing an a-priori unknown number of elements. Linked lists (or *lists* for short) are the data-container of choice in such situations. As in a chain, each *node* in a list is an independent C structure, keeping its own data along with pointer(s) to its immediate neighbor(s). Below, you can see one simple linked list node structure along with an ASCII art schematic of how we can use the `next` pointer to add any number of elements to the list that we want. By convention, a list is terminated when `next` is the `NULL` pointer.

```
struct list_float          /*     ---------    ---------           */
{                          /*     | Value |    | Value |           */
  float             value; /*     |  ---  |    |  ---  |           */
  struct list_float *next; /*     |  next-|--> |  next-|--> NULL   */
}                          /*     ---------    ---------           */
```

The schematic shows another great advantage of linked lists: it is very easy to add or remove/pop a node anywhere in the list. If you want to modify the first node, you just have to change one pointer. If it is in the middle, you just have to change two. You initially define a variable of this type with a `NULL` pointer as shown below:

```
struct list_float *mylist=NULL;
```

To add or remove/pop a node from the list you can use functions provided for the respective type in the sections below.

When you add an element to the list, it is conventionally added to the "top" of the list: the general list pointer will point to the newly created node, which will point to the previously created node and so on. So when you "pop" from the top of the list, you are actually retrieving the last value you put in and changing the list pointer to the next youngest node. This is thus known as a "last-in-first-out" list. This is the most efficient type of linked list (easier to implement and faster to process). Alternatively, you can add each newly created node at the end of the list. If you do that, you will get a "first-in-first-out" list. But that will force you to go through the whole list for each new element that is created (this will slow down the processing)[18].

The node example above creates the simplest kind of a list. We can define each node with two pointers to both the next and previous neighbors, this is called a "Doubly linked

[18] A better way to get a first-in-first-out is to first keep the data as last-in-first-out until they are all read. Afterwards, reverse the list by popping each node and immediately add it to the new list. This practically reverses the last-in-first-out list to a first-in-first-out one. All the list types discussed in this chapter have a function with a `_reverse` suffix for this job.

list". In general, lists are very powerful and simple constructs that can be very useful. But going into more detail would be out of the scope of this short introduction in this book. Wikipedia (`https://en.wikipedia.org/wiki/Linked_list`) has a nice and more thorough discussion of the various types of lists. To appreciate/use the beauty and elegance of these powerful constructs even further, see Chapter 2 (Information Structures, in volume 1) of Donald Knuth's "The art of computer programming".

In this section we will review the functions and structures that are available in Gnuastro for working on lists. They differ by the type of data that each node can keep. For each linked-list node structure, we will first introduce the structure, then the functions for working on the structure. All these structures and functions are defined and declared in `gnuastro/list.h`.

11.3.8.1 List of strings

Probably one of the most common lists you will be using are lists of strings. They are the best tools when you are reading the user's inputs, or when adding comments to the output files. Below you can see Gnuastro's string list type and several functions to help in adding, removing/popping, reversing and freeing the list.

`gal_list_str_t` [Type (C `struct`)]

A single node in a list containing a string of characters.

```
typedef struct gal_list_str_t
{
  char *v;
  struct gal_list_str_t *next;
} gal_list_str_t;
```

`void` [Function]
`gal_list_str_add` (*gal_list_str_t* `**list`, *char* `*value`, *int* `allocate`)

Add a new node to the list of strings (`list`) and update it. The new node will contain the string `value`. If `allocate` is not zero, space will be allocated specifically for the string of the new node and the contents of `value` will be copied into it. This can be useful when your string may be changed later in the program, but you want your list to remain. Here is one short/simple example of initializing and adding elements to a string list:

```
gal_list_str_t *strlist=NULL;
gal_list_str_add(&strlist, "bottom of list.");
gal_list_str_add(&strlist, "second last element of list.");
```

`char *` [Function]
`gal_list_str_pop` (*gal_list_str_t* `**list`)

Pop the top element of `list`, change `list` to point to the next node in the list, and return the string that was in the popped node. If `*list==NULL`, then this function will also return a `NULL` pointer.

`size_t` [Function]
`gal_list_str_number` (*gal_list_str_t* `*list`)

Return the number of nodes in `list`.

`void` [Function]
`gal_list_str_print` (*gal_list_str_t* `*list`)
Print the strings within each node of `*list` on the standard output in the same order that they are stored. Each string is printed on one line. This function is mainly good for checking/debugging your program. For program outputs, its best to make your own implementation with a better, more user-friendly, format. For example the following code snippet.

```
size_t i;
gal_list_str_t *tmp;
for(tmp=list; tmp!=NULL; tmp=tmp->next)
  printf("String %zu: %s\n", i, tmp->v);
```

`void` [Function]
`gal_list_str_reverse` (*gal_list_str_t* `**list`)
Reverse the order of the list such that the top node in the list before calling this function becomes the bottom node after it.

`void` [Function]
`gal_list_str_free` (*gal_list_str_t* `*list`, *int* `freevalue`)
Free every node in `list`. If `freevalue` is not zero, also free the string within the nodes.

11.3.8.2 List of `int32_t`

Signed integers are the best types when you are dealing with a positive or negative integers. The are generally useful in many contexts, for example when you want to keep the order of a series of states (each state stored as a given number in an `enum` for example). On many modern systems, `int32_t` is just an alias for `int`, so you can use them interchangeably. To make sure, check the size of `int` on your system:

`gal_list_i32_t` [Type (C `struct`)]
A single node in a list containing a 32-bit signed integer (see Section 4.5 [Numeric data types], page 115).

```
typedef struct gal_list_i32_t
{
  int32_t v;
  struct gal_list_i32_t *next;
} gal_list_i32_t;
```

`void` [Function]
`gal_list_i32_add` (*gal_list_i32_t* `**list`, *int32_t* `value`)
Add a new node (containing `value`) to the top of the `list` of `int32_t`s (`uint32_t` is equal to `int` on many modern systems), and update `list`. Here is one short example of initializing and adding elements to a string list:

```
gal_list_i32_t *i32list=NULL;
gal_list_i32_add(&i32list, 52);
gal_list_i32_add(&i32list, -4);
```

`int32_t` [Function]
`gal_list_i32_pop` (*gal_list_i32_t* `**list`)
Pop the top element of `list` and return the value. This function will also change `list` to point to the next node in the list. If `*list==NULL`, then this function will also return `GAL_BLANK_INT32` (see Section 11.3.5 [Library blank values (`blank.h`)], page 343).

`size_t` [Function]
`gal_list_i32_number` (*gal_list_i32_t* `*list`)
Return the number of nodes in `list`.

`void` [Function]
`gal_list_i32_print` (*gal_list_i32_t* `*list`)
Print the integers within each node of `*list` on the standard output in the same order that they are stored. Each integer is printed on one line. This function is mainly good for checking/debugging your program. For program outputs, its best to make your own implementation with a better, more user-friendly format. For example the following code snippet. You can also modify it to print all values in one line, and etc, depending on the context of your program.

```
size_t i;
gal_list_i32_t *tmp;
for(tmp=list; tmp!=NULL; tmp=tmp->next)
  printf("String %zu: %s\n", i, tmp->v);
```

`void` [Function]
`gal_list_i32_reverse` (*gal_list_i32_t* `**list`)
Reverse the order of the list such that the top node in the list before calling this function becomes the bottom node after it.

`int32_t *` [Function]
`gal_list_i32_to_array` (*gal_list_i32_t* `*list`, *int* `reverse`, *size_t* `*num`)
Dynamically allocate an array and fill it with the values in `list`. The function will return a pointer to the allocated array and put the number of elements in the array into the `num` pointer. If `reverse` has a non-zero value, the array will be filled in the opposite order of elements in `list`. This function can be useful after you have finished reading an initially unknown number of values and want to put them in an array for easy random access.

`void` [Function]
`gal_list_i32_free` (*gal_list_i32_t* `*list`)
Free every node in `list`.

11.3.8.3 List of `size_t`

The `size_t` type is a unique type in C: as the name suggests it is defined to store sizes, or more accurately, the distances between memory locations. Hence it is always positive (an `unsigned` type) and it is directly related to the address-able spaces on the host system: on 32-bit and 64-bit systems it is an alias for `uint32_t` and `uint64_t`, respectively (see Section 4.5 [Numeric data types], page 115).

`size_t` is the default compiler type to index an array (recall that an array index in C is just a pointer increment of a given *size*). Since it is unsigned, its a great type for counting (where negative is not defined), you are always sure it will never exceed the system's (virtual) memory and since its name has the word "size" inside it, it provides a good level of documentation[19]. In Gnuastro, we do all counting and array indexing with this type, so this list is very handy. As discussed above, `size_t` maps to different types on different machines, so a portable way to print them with `printf` is to use C99's `%zu` format.

`gal_list_sizet_t` [Type (C `struct`)]

A single node in a list containing a `size_t` value (which maps to `uint32_t` or `uint64_t` on 32-bit and 64-bit systems), see Section 4.5 [Numeric data types], page 115.

```
typedef struct gal_list_sizet_t
{
  size_t v;
  struct gal_list_sizet_t *next;
} gal_list_sizet_t;
```

`void` [Function]
`gal_list_sizet_add` (*gal_list_sizet_t* `**list`, *size_t* `value`)

Add a new node (containing `value`) to the top of the `list` of `size_t`s and update `list`. Here is one short example of initializing and adding elements to a string list:

```
gal_list_sizet_t *slist=NULL;
gal_list_sizet_add(&slist, 45493);
gal_list_sizet_add(&slist, 930484);
```

`sizet_t` [Function]
`gal_list_sizet_pop` (*gal_list_sizet_t* `**list`)

Pop the top element of `list` and return the value. This function will also change `list` to point to the next node in the list. If `*list==NULL`, then this function will also return `GAL_BLANK_SIZE_T` (see Section 11.3.5 [Library blank values (`blank.h`)], page 343).

`size_t` [Function]
`gal_list_sizet_number` (*gal_list_sizet_t* `*list`)

Return the number of nodes in `list`.

`void` [Function]
`gal_list_sizet_print` (*gal_list_sizet_t* `*list`)

Print the values within each node of `*list` on the standard output in the same order that they are stored. Each integer is printed on one line. This function is mainly good for checking/debugging your program. For program outputs, its best to make your own implementation with a better, more user-friendly format. For example, the following code snippet. You can also modify it to print all values in one line, and etc, depending on the context of your program.

```
size_t i;
gal_list_sizet_t *tmp;
```

[19] So you know that a variable of this type is not used to store some generic state for example.

```
for(tmp=list; tmp!=NULL; tmp=tmp->next)
  printf("String %zu: %zu\n", i, tmp->v);
```

`void` [Function]
`gal_list_sizet_reverse` (*gal_list_sizet_t* `**list`)
Reverse the order of the list such that the top node in the list before calling this function becomes the bottom node after it.

`size_t *` [Function]
`gal_list_sizet_to_array` (*gal_list_sizet_t* `*list`, *int* `reverse`, *size_t* `*num`)
Dynamically allocate an array and fill it with the values in `list`. The function will return a pointer to the allocated array and put the number of elements in the array into the `num` pointer. If `reverse` has a non-zero value, the array will be filled in the inverse of the order of elements in `list`. This function can be useful after you have finished reading an initially unknown number of values and want to put them in an array for easy random access.

`void` [Function]
`gal_list_sizet_free` (*gal_list_sizet_t* `*list`)
Free every node in `list`.

11.3.8.4 List of `float`

Single precision floating point numbers can accurately store real number until 7.2 decimals and only consume 4 bytes (32-bits) of memory, see Section 4.5 [Numeric data types], page 115. Since astronomical data rarely reach that level of precision, single precision floating points are the type of choice to keep and read data. However, when processing the data, it is best to use double precision floating points (since errors propagate).

`gal_list_f32_t` [Type (C `struct`)]
A single node in a list containing a 32-bit single precision `float` value: see Section 4.5 [Numeric data types], page 115.

```
typedef struct gal_list_f32_t
{
  float v;
  struct gal_list_f32_t *next;
} gal_list_f32_t;
```

`void` [Function]
`gal_list_f32_add` (*gal_list_f32_t* `**list`, *float* `value`)
Add a new node (containing `value`) to the top of the `list` of `float`s and update `list`. Here is one short example of initializing and adding elements to a string list:

```
gal_list_f32_t *flist=NULL;
gal_list_f32_add(&flist, 3.89);
gal_list_f32_add(&flist, 1.23e-20);
```

`float` [Function]
`gal_list_f32_pop` (*gal_list_f32_t* `**list`)
Pop the top element of `list` and return the value. This function will also change `list` to point to the next node in the list. If `*list==NULL`, then this function will

return `GAL_BLANK_FLOAT32` (NaN, see Section 11.3.5 [Library blank values (`blank.h`)], page 343).

`size_t` [Function]
`gal_list_f32_number` (*gal_list_f32_t* `*list`)
Return the number of nodes in `list`.

`void` [Function]
`gal_list_f32_print` (*gal_list_f32_t* `*list`)
Print the values within each node of `*list` on the standard output in the same order that they are stored. Each floating point number is printed on one line. This function is mainly good for checking/debugging your program. For program outputs, its best to make your own implementation with a better, more user-friendly format. For example, in the following code snippet. You can also modify it to print all values in one line, and etc, depending on the context of your program.

```
size_t i;
gal_list_f32_t *tmp;
for(tmp=list; tmp!=NULL; tmp=tmp->next)
  printf("Node %zu: %f\n", i, tmp->v);
```

`void` [Function]
`gal_list_f32_reverse` (*gal_list_f32_t* `**list`)
Reverse the order of the list such that the top node in the list before calling this function becomes the bottom node after it.

`float *` [Function]
`gal_list_f32_to_array` (*gal_list_f32_t* `*list`, *int* `reverse`, *size_t* `*num`)
Dynamically allocate an array and fill it with the values in `list`. The function will return a pointer to the allocated array and put the number of elements in the array into the `num` pointer. If `reverse` has a non-zero value, the array will be filled in the inverse of the order of elements in `list`. This function can be useful after you have finished reading an initially unknown number of values and want to put them in an array for easy random access.

`void` [Function]
`gal_list_f32_free` (*gal_list_f32_t* `*list`)
Free every node in `list`.

11.3.8.5 List of `double`

Double precision floating point numbers can accurately store real number until 15.9 decimals and consume 8 bytes (64-bits) of memory, see Section 4.5 [Numeric data types], page 115. This level of precision makes them very good for serious processing in the middle of a program's execution: in many cases, the propagation of errors will still be insignificant compared to actual observational errors in a data set. But since they consume 8 bytes and more CPU processing power, they are often not the best choice for storing and transferring of data.

`gal_list_f64_t` [Type (C `struct`)]

A single node in a list containing a 64-bit double precision `double` value: see Section 4.5 [Numeric data types], page 115.

```
typedef struct gal_list_f64_t
{
  double v;
  struct gal_list_f64_t *next;
} gal_list_f64_t;
```

`void` [Function]
`gal_list_f64_add` (*gal_list_f64_t* `**list`, *double* `value`)

Add a new node (containing `value`) to the top of the `list` of `double`s and update `list`. Here is one short example of initializing and adding elements to a string list:

```
gal_list_f64_t *dlist=NULL;
gal_list_f64_add(&dlist, 3.8129395763193);
gal_list_f64_add(&dlist, 1.239378923931e-20);
```

`double` [Function]
`gal_list_f64_pop` (*gal_list_f64_t* `**list`)

Pop the top element of `list` and return the value. This function will also change `list` to point to the next node in the list. If `*list==NULL`, then this function will return `GAL_BLANK_FLOAT64` (NaN, see Section 11.3.5 [Library blank values (`blank.h`)], page 343).

`size_t` [Function]
`gal_list_f64_number` (*gal_list_f64_t* `*list`)

Return the number of nodes in `list`.

`void` [Function]
`gal_list_f64_print` (*gal_list_f64_t* `*list`)

Print the values within each node of `*list` on the standard output in the same order that they are stored. Each floating point number is printed on one line. This function is mainly good for checking/debugging your program. For program outputs, its best to make your own implementation with a better, more user-friendly format. For example, in the following code snippet. You can also modify it to print all values in one line, and etc, depending on the context of your program.

```
size_t i;
gal_list_f64_t *tmp;
for(tmp=list; tmp!=NULL; tmp=tmp->next)
  printf("Node %zu: %f\n", i, tmp->v);
```

`void` [Function]
`gal_list_f64_reverse` (*gal_list_f64_t* `**list`)

Reverse the order of the list such that the top node in the list before calling this function becomes the bottom node after it.

`double *` [Function]
`gal_list_f64_to_array` (*gal_list_f64_t* `*list`, *int* `reverse`, *size_t* `*num`)
Dynamically allocate an array and fill it with the values in `list`. The function will return a pointer to the allocated array and put the number of elements in the array into the `num` pointer. If `reverse` has a non-zero value, the array will be filled in the inverse of the order of elements in `list`. This function can be useful after you have finished reading an initially unknown number of values and want to put them in an array for easy random access.

`void` [Function]
`gal_list_f64_free` (*gal_list_f64_t* `*list`)
Free every node in `list`.

11.3.8.6 List of `void *`

In C, `void *` is the most generic pointer. Usually pointers are associated with the type of content they point to. For example `int *` means a pointer to an integer. This ancillary information about the contents of the memory location is very useful for the compiler, catching bad errors and also documentation (it helps the reader see what the address in memory actually contains). However, `void *` is just a raw address (pointer), it contains no information on the contents it points to.

These properties make the `void *` very useful when you want to treat the contents of an address in different ways. You can use the `void *` list defined in this section and its function on any kind of data: for example you can use it to keep a list of custom data structures that you have built for your own separate program. Each node in the list can keep anything and this gives you great versatility. But in using `void *`, please beware that "with great power comes great responsibility".

`gal_list_void_t` [Type (C `struct`)]
A single node in a list containing a `void *` pointer.

```
typedef struct gal_list_void_t
{
  void *v;
  struct gal_list_void_t *next;
} gal_list_void_t;
```

`void` [Function]
`gal_list_void_add` (*gal_list_void_t* `**list`, *void* `*value`)
Add a new node (containing `value`) to the top of the `list` of `void *`s and update `list`. Here is one short example of initializing and adding elements to a string list:

```
gal_list_void_t *vlist=NULL;
gal_list_f64_add(&vlist, some_pointer);
gal_list_f64_add(&vlist, another_pointer);
```

`void *` [Function]
`gal_list_void_pop` (*gal_list_void_t* `**list`)
Pop the top element of `list` and return the value. This function will also change `list` to point to the next node in the list. If `*list==NULL`, then this function will return `NULL`.

`size_t` [Function]
`gal_list_void_number` (*gal_list_void_t* `*list`)
Return the number of nodes in `list`.

`void` [Function]
`gal_list_void_reverse` (*gal_list_void_t* `**list`)
Reverse the order of the list such that the top node in the list before calling this function becomes the bottom node after it.

`void` [Function]
`gal_list_void_free` (*gal_list_void_t* `*list`)
Free every node in `list`.

11.3.8.7 Ordered list of `size_t`

Positions/sizes in a dataset are conventionally in the `size_t` type (see Section 11.3.8.3 [List of `size_t`], page 360) and it sometimes occurs that you want to parse and read the values in a specific order. For example you want to start from one pixel and add pixels to the list based on their distance to that pixel. So that ever time you pop an element from the list, you know it is the nearest that has not yet been studied. The `gal_list_osizet_t` type and its functions in this section are designed to facilitate such operations.

`gal_list_osizet_t` [Type (C `struct`)]
Each node in this singly-linked list contains a `size_t` value and a floating point value. The floating point value is used as a reference to add new nodes in a sorted manner. At any moment, the first popped node in this list will have the smallest `tosort` value, and subsequent nodes will have larger to values.

```
typedef struct gal_list_osizet_t
{
  size_t v;                     /* The actual value. */
  float  s;                     /* The parameter to sort by. */
  struct gal_list_osizet_t *next;
} gal_list_osizet_t;
```

`void` [Function]
`gal_list_osizet_add` (*gal_list_osizet_t* `**list`, *size_t* `value`, *float* `tosort`)
Allocate space for a new node in `list`, and store `value` and `tosort` into it. The new node will not necessarily be at the "top" of the list. If `*list!=NULL`, then the `tosort` values of existing nodes is inspected and the given node is placed in the list such that the top element (which is popped with `gal_list_osizet_pop`) has the smallest `tosort` value.

`size_t` [Function]
`gal_list_osizet_pop` (*gal_list_osizet_t* `**list`, *float* `*sortvalue`)
Pop a node from the top of `list`, return the node's `value` and put its sort value in the space that `sortvalue` points to. This function will also free the allocated space for the popped node and after this function, `list` will point to the next node (which has a larger `tosort` element).

`void` [Function]
`gal_list_osizet_to_sizet_free` (*gal_list_osizet_t* `*in`, *gal_list_sizet_t* `**out`)
Convert the ordered list of `size_t`s into an ordinary `size_t` linked list. This can be useful when all the elements have been added and you just need to pop-out elements and don't care about the sorting values any more. After the conversion is done, this function will free the input list. Note that the `out` list doesn't have to be empty. If it already contains some nodes, the new nodes will be added on top of them.

11.3.8.8 Doubly linked ordered list of `size_t`

An ordered list of indexs is required in many contexts, one example was discussed at the beginning of Section 11.3.8.7 [Ordered list of `size_t`], page 366. But the list that was introduced there only has one point of entry: you can always only parse the list from smallest to largest. In this section, the doubly-linked `gal_list_dosizet_t` node is defined which will allow us to parse the values in ascending or descending order.

`gal_list_dosizet_t` [Type (C `struct`)]
Doubly-linked, ordered `size_t` list node structure. Each node in this Doubly-linked list contains a `size_t` value and a floating point value. The floating point value is used as a reference to add new nodes in a sorted manner. In the functions here, this linked list can be pointed to by two pointers (largest and smallest) with the following format:

```
                largest pointer
                |
   NULL <-- (v0,s0) <--> (v1,s1) <--> ... (vn,sn) --> NULL
                                                |
                                  smallest pointer
```

At any moment, the two pointers will point to the nodes containing the "largest" and "smallest" values and the rest of the nodes will be sorted. This is useful when an unknown number of nodes are being added continuously and during the operations it is important to have the nodes in a sorted format.

```
typedef struct gal_list_dosizet_t
{
  size_t v;                       /* The actual value. */
  float s;                        /* The parameter to sort by. */
  struct gal_list_dosizet_t *prev;
  struct gal_list_dosizet_t *next;
} gal_list_dosizet_t;
```

`void` [Function]
`gal_list_dosizet_add` (*gal_list_dosizet_t* `**largest`, *gal_list_dosizet_t* `**smallest`, *size_t* `value`, *float* `tosort`)
Allocate space for a new node in `list`, and store `value` and `tosort` into it. If the list is empty, both `largest` and `smallest` must be `NULL`.

`size_t` [Function]
`gal_list_dosizet_pop_smallest` (*gal_list_dosizet_t* `**largest`, *gal_list_dosizet_t* `**smallest`, *float* `tosort`)
Pop the value with the smallest reference from the doubly linked list and store the reference into the space pointed to by `tosort`. Note that even though only the smallest pointer will be popped, when there was only one node in the list, the `largest` pointer also has to change, so we need both.

`void` [Function]
`gal_list_dosizet_print` (*gal_list_dosizet_t* `*largest`, *gal_list_dosizet_t* `*smallest`)
Print the largest and smallest values sequentially until the list is parsed.

`void` [Function]
`gal_list_dosizet_to_sizet` (*gal_list_dosizet_t* `*in`, *gal_list_sizet_t* `**out`)
Convert the doubly linked, ordered `size_t` list into a singly-linked list of `size_t`.

`void` [Function]
`gal_list_dosizet_free` (*gal_list_dosizet_t* `*largest`)
Free the doubly linked, ordered `sizet_t` list.

11.3.8.9 List of `gal_data_t`

Gnuastro's generic data container has a `next` element which enables it to be used as a singly-linked list (see Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346). The ability to connect the different data containers offers great advantages. For example each column in a table in an independent dataset: with its own name, units, numeric data type (see Section 4.5 [Numeric data types], page 115). Another application is in Tessellating an input dataset into separate tiles or only studying particular regions, or tiles, of a larger dataset (see Section 4.7 [Tessellation], page 123, and Section 11.3.15 [Tessellation library (`tile.h`)], page 399). Each independent tile over the dataset can be connected to the others as a linked list and thus any number of tiles can be represented with one variable.

`void` [Function]
`gal_list_data_add` (*gal_data_t* `**list`, *gal_data_t* `*newnode`)
Add an already allocated dataset (`newnode`) to top of `list`. Note that if `newnode->next!=NULL` (`newnode` is itself a list), then `list` will be added to its end.

In this example multiple images are linked together as a list:

```
size_t minmapsize=-1;
gal_data_t *tmp, *list=NULL;
tmp = gal_fits_img_read("file1.fits", "1", minmapsize);
gal_list_data_add( &list, tmp );
tmp = gal_fits_img_read("file2.fits", "1", minmapsize);
gal_list_data_add( &list, tmp );
```

`void` [Function]
`gal_list_data_add_alloc` (*gal_data_t* `**list`, *void* `*array`, *uint8_t* `type`, *size_t* `ndim`, *size_t* `*dsize`, *struct wcsprm* `*wcs`, *int* `clear`, *size_t* `minmapsize`, *char* `*name`, *char* `*unit`, *char* `*comment`)

Allocate a new dataset (with `gal_data_alloc` in Section 11.3.6.2 [Dataset allocation], page 350) and put it as the first element of `list`. Note that if this is the first node to be added to the list, `list` must be `NULL`.

`gal_data_t *` [Function]
`gal_list_data_pop` (*gal_data_t* `**list`)

Pop the top node from `list` and return it.

`void` [Function]
`gal_list_data_reverse` (*gal_data_t* `**list`)

Reverse the order of the list such that the top node in the list before calling this function becomes the bottom node after it.

`size_t` [Function]
`gal_list_data_number` (*gal_data_t* `*list`)

Return the number of nodes in `list`.

`void` [Function]
`gal_list_data_free` (*gal_data_t* `*list`)

Free all the datasets in `list` along with all the allocated spaces in each.

11.3.9 Array input output

Getting arrays (commonly images or cubes) from a file into your program or writing them after the processing into an output file are some of the most common operations. The functions in this section are designed for such operations with the known file types. The functions here are thus just wrappers around functions of lower-level file type functions of this library, for example Section 11.3.11 [FITS files (`fits.h`)], page 374, or Section 11.3.12.2 [TIFF files (`tiff.h`)], page 388. If the file type of the input/output file is already known, you can use the functions in those sections respectively.

`int` [Function]
`gal_array_name_recognized` (*char* `*filename`)

Return 1 if the given file name corresponds to one of the recognized file types for reading arrays.

`int` [Function]
`gal_array_name_recognized_multiext` (*char* `*filename`)

Return 1 if the given file name corresponds to one of the recognized file types for reading arrays which may contain multiple extensions (for example FITS or TIFF) formats.

`gal_data_t` [Function]
`gal_array_read` (*char* `*filename`, *char* `*extension`, *gal_list_str_t* `*lines`, *size_t* `minmapsize`)

Read the array within the given extension (`extension`) of `filename`, or the `lines` list (see below). If the array is larger than `minmapsize` bytes, then it won't be read into

RAM, but a file on the HDD/SSD (no difference for the programmer). `extension` will be ignored for files that don't support them (for example JPEG or text). For FITS files, `extension` can be a number or a string (name of the extension), but for TIFF files, it has to be number. In both cases, counting starts from zero.

For multi-channel formats (like RGB images in JPEG or TIFF), this function will return a Section 11.3.8.9 [List of `gal_data_t`], page 368: one data structure per channel. Thus if you just want a single array (and want to check if the user hasn't given a multi-channel input), you can check the `next` pointer of the returned `gal_data_t`.

`lines` is a list of strings with each node representing one line (including the new-line character), see Section 11.3.8.1 [List of strings], page 358. It will mostly be the output of `gal_txt_stdin_read`, which is used to read the program's input as separate lines from the standard input (see Section 11.3.12.1 [Text files (`txt.h`)], page 386). Note that `filename` and `lines` are mutually exclusive and one of them must be `NULL`.

`void` [Function]
`gal_array_read_to_type` (*char* `*filename`, *char* `*extension`, *gal_list_str_t* `*lines`, *uint8_t* `type`, *size_t* `minmapsize`)

Similar to `gal_array_read`, but the output data structure(s) will have a numeric data type of `type`, see Section 4.5 [Numeric data types], page 115.

`void` [Function]
`gal_array_read_one_ch` (*char* `*filename`, *char* `*extension`, *gal_list_str_t* `*lines`, *size_t* `minmapsize`)

Read the dataset within `filename` (extension/hdu/dir `extension`) and make sure it is only a single channel. This is just a simple wrapper around `gal_array_read` that checks if there was more than one dataset and aborts with an informative error if there is more than one channel in the dataset.

Formats like JPEG or TIFF support multiple channels per input, but it may happen that your program only works on a single dataset. This function can be a convenient way to make sure that the data that comes into your program is only one channel.

`void` [Function]
`gal_array_read_one_ch_to_type` (*char* `*filename`, *char* `*extension`, *gal_list_str_t* `*lines`, *uint8_t* `type`, *size_t* `minmapsize`)

Similar to `gal_array_read_one_ch`, but the output data structure will has a numeric data type of `type`, see Section 4.5 [Numeric data types], page 115.

11.3.10 Table input output (`table.h`)

Tables are a collection of one dimensional datasets that are packed together into one file. They are the single most common format to store high-level (processed) information, hence they play a very important role in Gnuastro. For a more thorough introduction, please see Section 5.3 [Table], page 147. Gnuastro's Table program, and all the other programs that can read from and write into tables, use the functions of this section for reading and writing their input/output tables. For a simple demonstration of using the constructs introduced here, see Section 11.4.4 [Library demo - reading and writing table columns], page 442.

Currently only plain text (see Section 4.6.2 [Gnuastro text table format], page 119) and FITS (ASCII and binary) tables are supported by Gnuastro. However, the low-level table infra-structure is written such that accommodating other formats is also possible and in future releases more formats will hopefully be supported. Please don't hesitate to suggest your favorite format so it can be implemented when possible.

`GAL_TABLE_DEF_WIDTH_STR` [Macro]
`GAL_TABLE_DEF_WIDTH_INT` [Macro]
`GAL_TABLE_DEF_WIDTH_LINT` [Macro]
`GAL_TABLE_DEF_WIDTH_FLT` [Macro]
`GAL_TABLE_DEF_WIDTH_DBL` [Macro]
`GAL_TABLE_DEF_PRECISION_INT` [Macro]
`GAL_TABLE_DEF_PRECISION_FLT` [Macro]
`GAL_TABLE_DEF_PRECISION_DBL` [Macro]

The default width and precision for generic types to use in writing numeric types into a text file (plain text and FITS ASCII tables). When the dataset doesn't have any pre-set width and precision (see `disp_width` and `disp_precision` in Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346) these will be directly used in C's `printf` command to write the number as a string.

`GAL_TABLE_DISPLAY_FMT_STRING` [Macro]
`GAL_TABLE_DISPLAY_FMT_DECIMAL` [Macro]
`GAL_TABLE_DISPLAY_FMT_UDECIMAL` [Macro]
`GAL_TABLE_DISPLAY_FMT_OCTAL` [Macro]
`GAL_TABLE_DISPLAY_FMT_HEX` [Macro]
`GAL_TABLE_DISPLAY_FMT_FLOAT` [Macro]
`GAL_TABLE_DISPLAY_FMT_EXP` [Macro]
`GAL_TABLE_DISPLAY_FMT_GENERAL` [Macro]

The display format used in C's `printf` to display data of different types. The `_STRING` and `_DECIMAL` are unique for printing strings and signed integers, they are mainly here for completeness. However, unsigned integers and floating points can be displayed in multiple formats:

Unsigned integer

For unsigned integers, it is possible to choose from `_UDECIMAL` (unsigned decimal), `_OCTAL` (octal notation, for example `125` in decimal will be displayed as `175`), and `_HEX` (hexadecimal notation, for example `125` in decimal will be displayed as `7D`).

Floating point

For floating point, it is possible to display the number in `_FLOAT` (floating point, for example `1500.345`), `_EXP` (exponential, for example `1.500345e+03`), or `_GENERAL` which is the best of the two for the given number.

`GAL_TABLE_FORMAT_INVALID` [Macro]
`GAL_TABLE_FORMAT_TXT` [Macro]
`GAL_TABLE_FORMAT_AFITS` [Macro]

`GAL_TABLE_FORMAT_BFITS` [Macro]

All the current acceptable table formats to Gnuastro. The `AFITS` and `BFITS` represent FITS ASCII tables and FITS Binary tables. You can use these anywhere you see the `tableformat` variable.

`GAL_TABLE_SEARCH_INVALID` [Macro]
`GAL_TABLE_SEARCH_NAME` [Macro]
`GAL_TABLE_SEARCH_UNIT` [Macro]
`GAL_TABLE_SEARCH_COMMENT` [Macro]

When the desired column is not a number, these values determine if the string to match, or regular expression to search, be in the *name*, *units* or *comments* of the column meta data. These values should be used for the `searchin` variables of the functions.

`gal_data_t *` [Function]
`gal_table_info` (*char* `*filename`, *char* `*hdu`, *gal_list_str_t* `*lines`, *size_t* `*numcols`, *size_t* `*numrows`, *int* `*tableformat`)

Store the information of each column of a table into an array of data structures with `numcols` datasets (one data structure for each column). The number of rows is stored in `numrows`. The format of the table (e.g., ASCII text file, or FITS binary or ASCII table) will be put in `tableformat` (macros defined above). If the `filename` is not a FITS file, then `hdu` will not be used (can be `NULL`).

The input must be either a file (specified by `filename`) or a list of strings (`lines`). `lines` is a list of strings with each node representing one line (including the new-line character), see Section 11.3.8.1 [List of strings], page 358. It will mostly be the output of `gal_txt_stdin_read`, which is used to read the program's input as separate lines from the standard input (see Section 11.3.12.1 [Text files (`txt.h`)], page 386). Note that `filename` and `lines` are mutually exclusive and one of them must be `NULL`.

In the output datasets, only the meta-data strings (column name, units and comments), will be allocated and set. This function is just for column information (meta-data), not column contents.

`void` [Function]
`gal_table_print_info` (*gal_data_t* `*allcols`, *size_t* `numcols`, *size_t* `numrows`)

This program will print the column information for all the columns (output of `gal_table_info`). The output is in the same format as this command with Gnuastro Table program (see Section 5.3 [Table], page 147):

```
$ asttable --info table.fits
```

`gal_data_t *` [Function]
`gal_table_read` (*char* `*filename`, *char* `*hdu`, *gal_list_str_t* `*lines`, *gal_list_str_t* `*cols`, *int* `searchin`, *int* `ignorecase`, *size_t* `minmapsize`, *size_t* `*colmatch`)

Read the specified columns in a file (named `filename`), or list of strings (`lines`) into a linked list of data structures. If the file is FITS, then `hdu` will also be used, otherwise, `hdu` is ignored.

`lines` is a list of strings with each node representing one line (including the new-line character), see Section 11.3.8.1 [List of strings], page 358. It will mostly be the output

of `gal_txt_stdin_read`, which is used to read the program's input as separate lines from the standard input (see Section 11.3.12.1 [Text files (`txt.h`)], page 386). Note that `filename` and `lines` are mutually exclusive and one of them must be `NULL`.

The information to search for columns should be specified by the `cols` list of strings (see Section 11.3.8.1 [List of strings], page 358). The string in each node of the list may be a number, an exact match to a column name, or a regular expression (in GNU AWK format) enclosed in `/ /`. The `searchin` value must be one of the macros defined above. If `cols` is NULL, then this function will read the full table.

The output is an individually allocated list of datasets (see Section 11.3.8.9 [List of `gal_data_t`], page 368) with the same order of the `cols` list. Note that one column node in the `cols` list might give multiple columns (for example from regular expressions), in this case, the order of output columns that correspond to that one input, are in order of the table (which column was read first). So the first requested column is the first popped data structure and so on.

if `colmatch!=NULL`, it is assumed to be an array that has at least the same number of elements as nodes in the `cols` list. The number of columns that matched each input column will be stored in each element.

`gal_list_sizet_t *` [Function]
`gal_table_list_of_indexs` (*gal_list_str_t* `*cols`, *gal_data_t* `*allcols`, *size_t* `numcols`, *int* `searchin`, *int* `ignorecase`, *char* `*filename`, *char* `*hdu`, *size_t* `*colmatch`)

Returns a list of indexs (starting from 0) of the input columns that match the names/numbers given to `cols`. This is a low-level operation which is called by `gal_table_read` (described above), see there for more on each argument's description. `allcols` is the returned array of `gal_table_info`.

`void` [Function]
`gal_table_comments_add_intro` (*gal_list_str_t* `**comments`, *char* `*program_string`, *time_t* `*rawtime`)

Add some basic information to the list of `comments`. This basic information includes the following information

- If the program is run in a Git version controlled directory, Git's description is printed (see description under `COMMIT` in Section 4.9 [Output FITS files], page 125).
- The calendar time that is stored in `rawtime` (`time_t` is C's calendar time format defined in `time.h`). You can calculate the time in this format with the following expressions:

```
time_t rawtime;
time(&rawtime);
```

- The name of your program in `program_string`. If it is `NULL`, this line is ignored.

`void` [Function]
`gal_table_write` (*gal_data_t* `*cols`, *gal_list_str_t* `*comments`, *int* `tableformat`, *char* `*filename`, *char* `*extname`, *uint8_t* `colinfoinstdout`)

Write `cols` (a list of datasets, see Section 11.3.8.9 [List of `gal_data_t`], page 368) into a table stored in `filename`. The format of the table can be determined with

`tableformat` that accepts the macros defined above. When `filename==NULL`, the column information will be printed on the standard output (command-line).

If `comments!=NULL`, the list of comments (see Section 11.3.8.1 [List of strings], page 358) will also be printed into the output table. When the output table is a plain text file, every node of `comments` will be printed after a `#` (so it can be considered as a comment) and in FITS table they will follow a `COMMENT` keyword.

If a file named `filename` already exists, the operation depends on the type of output. When `filename` is a FITS file, the table will be added as a new extension after all existing extensions. If `filename` is a plain text file, this function will abort with an error.

If `filename` is a FITS file, the table extension will have the name `extname`.

When `colinfoinstdout!=0` and `filename==NULL` (columns are printed in the standard output), the dataset metadata will also printed in the standard output. When printing to the standard output, the column information can be piped into another program for further processing and thus the meta-data (lines starting with a `#`) must be ignored. In such cases, you only print the column values by passing `0` to `colinfoinstdout`.

`void` [Function]
`gal_table_write_log` (*gal_data_t* `*logll`, *char* `*program_string`, *time_t* `*rawtime`, *gal_list_str_t* `*comments`, *char* `*filename`, *int* `quiet`)

Write the `logll` list of datasets into a table in `filename` (see Section 11.3.8.9 [List of `gal_data_t`], page 368). This function is just a wrapper around `gal_table_comments_add_intro` and `gal_table_write` (see above). If `quiet` is non-zero, this function will print a message saying that the `filename` has been created.

11.3.11 FITS files (`fits.h`)

The FITS format is the most common format to store data (images and tables) in astronomy. The CFITSIO library already provides a very good low-level collection of functions for manipulating FITS data. The low-level nature of CFITSIO is defined for versatility and portability. As a result, even a simple and basic operation, like reading an image or table column into memory, will require a special sequence of CFITSIO function calls which can be inconvenient and buggy to manage in separate locations. To ease this process, Gnuastro's library provides wrappers for CFITSIO functions. With these, it much easier to read, write, or modify FITS file data, header keywords and extensions. Hence, if you feel these functions don't exactly do what you want, we strongly recommend reading the CFITSIO manual to use its great features directly (afterwards, send us your wrappers so we can include it here for others to benefit also).

All the functions and macros introduced in this section are declared in `gnuastro/fits.h`. When you include this header, you are also including CFITSIO's `fitsio.h` header. So you don't need to explicitly include `fitsio.h` anymore and can freely use any of its macros or functions in your code along with those discussed here.

11.3.11.1 FITS Macros, errors and filenames

Some general constructs provided by Gnuastro's FITS handling functions are discussed here. In particular there are several useful functions about FITS file names.

`GAL_FITS_MAX_NDIM` [Macro]
: The maximum number of dimensions a dataset can have in FITS format, according to the FITS standard this is 999.

`void` [Function]
`gal_fits_io_error (int status, char *message)`
: If `status` is non-zero, this function will print the CFITSIO error message corresponding to status, print `message` (optional) in the next line and abort the program. If `message==NULL`, it will print a default string after the CFITSIO error.

`int` [Function]
`gal_fits_name_is_fits (char *name)`
: If the `name` is an acceptable CFITSIO FITS filename return `1` (one), otherwise return `0` (zero). The currently acceptable FITS suffixes are `.fits`, `.fit`, `.fits.gz`, `.fits.Z`, `.imh`, `.fits.fz`. IMH is the IRAF format which is acceptable to CFITSIO.

`int` [Function]
`gal_fits_suffix_is_fits (char *suffix)`
: Similar to `gal_fits_name_is_fits`, but only for the suffix. The suffix doesn't have to start with '`.`': this function will return `1` (one) for both `fits` and `.fits`.

`char *` [Function]
`gal_fits_name_save_as_string (char *filename, char *hdu)`
: If the name is a FITS name, then put a `(hdu: ...)` after it and return the string. If it isn't a FITS file, just print the name, if `filename==NULL`, then return the string `stdin`. Note that the output string's space is allocated.

  This function is useful when you want to report a random file to the user which may be FITS or not (for a FITS file, simply the filename is not enough, the HDU is also necessary).

11.3.11.2 CFITSIO and Gnuastro types

Both Gnuastro and CFITSIO have special identifiers for each type that they accept. Gnuastro's type identifiers are fully described in Section 11.3.3 [Library data types (`type.h`)], page 337, and are usable for all kinds of datasets (images, table columns and etc) as part of Gnuastro's Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346. However, following the FITS standard, CFITSIO has different identifiers for images and tables. Following CFITSIO's own convention, we will use `bitpix` for image type identifiers and `datatype` for its internal identifiers (and mainly used in tables). The functions introduced in this section can be used to convert between CFITSIO and Gnuastro's type identifiers.

One important issue to consider is that CFITSIO's types are not fixed width (for example `long` may be 32-bits or 64-bits on different systems). However, Gnuastro's types are defined by their width. These functions will use information on the host system to do the proper conversion. To have a portable (usable on different systems) code, is thus recommended to use these functions and not to assume a fixed correspondence between CFITSIO and Gnuastro's types.

`uint8_t` [Function]
`gal_fits_bitpix_to_type` (*int* `bitpix`)
Return the Gnuastro type identifier that corresponds to CFITSIO's `bitpix` on this system.

`int` [Function]
`gal_fits_type_to_bitpix` (*uint8_t* `type`)
Return the CFITSIO `bitpix` value that corresponds to Gnuastro's `type`.

`char` [Function]
`gal_fits_type_to_bin_tform` (*uint8_t* `type`)
Return the FITS standard binary table `TFORM` character that corresponds to Gnuastro's `type`.

`int` [Function]
`gal_fits_type_to_datatype` (*uint8_t* `type`)
Return the CFITSIO `datatype` that corresponds to Gnuastro's `type` on this machine.

`uint8_t` [Function]
`gal_fits_datatype_to_type` (*int* `datatype`, *int* `is_table_column`)
Return Gnuastro's type identifier that corresponds to the CFITSIO `datatype`. Note that when dealing with CFITSIO's `TLONG`, the fixed width type differs between tables and images. So if the corresponding dataset is a table column, put a non-zero value into `is_table_column`.

11.3.11.3 FITS HDUs

A FITS file can contain multiple HDUs/extensions. The functions in this section can be used to get basic information about the extensions or open them. Note that `fitsfile` is defined in CFITSIO's `fitsio.h` which is automatically included by Gnuastro's `gnuastro/fits.h`.

`fitsfile *` [Function]
`gal_fits_open_to_write` (*char* `*filename`)
If `filename` exists, open it and return the `fitsfile` pointer that corresponds to it. If `filename` doesn't exist, the file will be created which contains a blank first extension and the pointer to its next extension will be returned.

`size_t` [Function]
`gal_fits_hdu_num` (*char* `*filename`)
Return the number of HDUs/extensions in `filename`.

`int` [Function]
`gal_fits_hdu_format` (*char* `*filename`, *char* `*hdu`)
Return the format of the HDU as one of CFITSIO's recognized macros: `IMAGE_HDU`, `ASCII_TBL`, or `BINARY_TBL`.

`fitsfile *` [Function]
`gal_fits_hdu_open` (*char* `*filename`, *char* `*hdu`, *int* `iomode`)
Open the HDU/extension `hdu` from `filename` and return a pointer to CFITSIO's `fitsfile`. `iomode` determines how the FITS file will be opened using CFITSIO's macros: `READONLY` or `READWRITE`.

The string in `hdu` will be appended to `filename` in square brackets so CFITSIO only opens this extension. You can use any formatting for the `hdu` that is acceptable to CFITSIO. See the description under `--hdu` in Section 4.1.2.1 [Input/Output options], page 95, for more.

`fitsfile *` [Function]
`gal_fits_hdu_open_format` (*char* `*filename`, *char* `*hdu`, *int* `img0_tab1`)

Open (in read-only format) the `hdu` HDU/extension of `filename` as an image or table. When `img0_tab1` is `0`(zero) but the HDU is a table, this function will abort with an error. It will also abort with an error when `img0_tab1` is `1` (one), but the HDU is an image.

A FITS HDU may contain both tables or images. When your program needs one of these formats, you can call this function so if the user provided the wrong HDU/file, it will abort and inform the user that the file/HDU is has the wrong format.

11.3.11.4 FITS header keywords

Each FITS extension/HDU contains a raw dataset which can either be a table or an image along with some header keywords. The keywords can be used to store meta-data about the actual dataset. The functions in this section describe Gnuastro's high-level functions for reading and writing FITS keywords. Similar to all Gnuastro's FITS-related functions, these functions are all wrappers for CFITSIO's low-level functions.

The necessary meta-data (header keywords) for a particular dataset are commonly numerous, it is much more efficient to list them in one variable and call the reading/writing functions once. Hence the functions in this section use linked lists, a thorough introduction to them is given in Section 11.3.8 [Linked lists (`list.h`)], page 357. To reading FITS keywords, these functions use a list of Gnuastro's generic dataset format that is discussed in Section 11.3.8.9 [List of `gal_data_t`], page 368. To write FITS keywords we define the `gal_fits_list_key_t` node that is defined below.

`gal_fits_list_key_t` [Type (C `struct`)]

Structure for writing FITS keywords. This structure is used for one keyword and you don't need to set all elements. With the `next` element, you can link it to another keyword thus creating a linked list to add any number of keywords easily and at any step during your program (see Section 11.3.8 [Linked lists (`list.h`)], page 357, for an introduction on lists). See the functions below for adding elements to the list.

```
typedef struct gal_fits_list_key_t
{
  int                      kfree;   /* ==1, free name.      */
  int                      vfree;   /* ==1, free value.     */
  int                      cfree;   /* ==1, free comment.   */
  uint8_t                   type;   /* Keyword value type.  */
  char                   *keyname;  /* Keyword Name.        */
  void                     *value;  /* Keyword value.       */
  char                   *comment;  /* Keyword comment.     */
  char                      *unit;  /* Keyword unit.        */
  struct gal_fits_list_key_t *next;  /* Next keyword.        */
} gal_fits_list_key_t;
```

`void *` [Function]
`gal_fits_key_img_blank` (*uint8_t* `type`)

Returns a pointer to an allocated space containing the value to the FITS `BLANK` header keyword, when the input array has a type of `type`. This is useful when you want to write the `BLANK` keyword using CFITSIO's `fits_write_key` function.

According to the FITS standard: "If the `BSCALE` and `BZERO` keywords do not have the default values of 1.0 and 0.0, respectively, then the value of the `BLANK` keyword must equal the actual value in the FITS data array that is used to represent an undefined pixel and not the corresponding physical value". Therefore a special `BLANK` value is needed for datasets containing signed 8-bit, unsigned 16-bit, unsigned 32-bit, and unsigned 64-bit integers (types that are defined with `BSCALE` and `BZERO` in the FITS standard).

Not usable when reading a dataset: As quoted from the FITS standard above, the value returned by this function can only be generically used for the writing of the `BLANK` keyword header. It *must not* be used as the blank pointer when when reading a FITS array using CFITSIO. When reading an array with CFITSIO, you can use `gal_blank_alloc_write` to generate the necessary pointer.

`void` [Function]
`gal_fits_key_clean_str_value` (*char* `*string`)

Remove the single quotes and possible extra spaces around the keyword values that CFITSIO returns when reading a string keyword. CFITSIO doesn't remove the two single quotes around the string value of a keyword. Hence the strings it reads are like: `'value '`, or `'some_very_long_value'`. To use the value during your processing, it is commonly necessary to remove the single quotes (and possible extra spaces). This function will do this within the allocated space of the string.

`char *` [Function]
`gal_fits_key_date_to_struct_tm` (*char* `*fitsdate`, *struct tm* `*tp`)

Parse `fitsdate` as a FITS date format string (most generally: `YYYY-MM-DDThh:mm:ss.ddd...`) into the C library's broken-down time structure, or `struct tm` (declared in `time.h`) and return a pointer to the remainder of the string (containing the optional sub-second portion of `fitsdate`, or the `.ddd...` of the format).

The returned `char *` points to part of the `fitsdate` string, so it must not be freed. For example, if `fitsdate` contains no sub-second portion, then the returned `char *` will point to the NULL-character of `fitsdate`.

Note that the FITS date format mentioned above is the most complete representation. The following two formats are also acceptable: `YYYY-MM-DDThh:mm:ss` and `YYYY-MM-DD`. This option can also interpret the older FITS date format where only two characters are given to the year and the date format is reversed (`DD/MM/YYThh:mm:ss.ddd...`). In this case (following the GNU C Library), this option will make the following assumption: values 68 to 99 correspond to the years 1969 to 1999, and values 0 to 68 as the years 2000 to 2068.

`size_t` [Function]
`gal_fits_key_date_to_seconds` (*char* `*fitsdate`, *char* `**subsecstr`, *double* `*subsec`)

Return the Unix epoch time (number of seconds that have passed since 00:00:00 Thursday, January 1st, 1970) corresponding to the FITS date format string `fitsdate` (see description of `gal_fits_key_date_to_struct_tm` above).

The Unix epoch time is in units of seconds, but the FITS date format allows sub-second accuracy. The last two arguments are for the (optional) sub-second portion. If `fitsdate` contains sub-second accuracy, then the starting of the sub-second part is stored in the `char *` pointer that `subsecstr` points to, and `subsec` will the corresponding numerical value (between 0 and 1, in double precision floating point).

This is a very useful function for operations on the FITS date values, for example sorting FITS files by their dates, or finding the time difference between two FITS files. The advantage of working with the Unix epoch time is that you don't have to worry about calendar details (for example the number of days in different months, or leap years, and etc).

`void` [Function]
`gal_fits_key_read_from_ptr` (*fitsfile* `*fptr`, *gal_data_t* `*keysll`, *int* `readcomment`, *int* `readunit`)

Read the list of keyword values from a FITS pointer. The input should be a linked list of Gnuastro's generic data container (`gal_data_t`). Before calling this function, you just have to set the `name` and desired `type` values of each node in the list to the keyword you want it to keep the value of. The given `name` value will be directly passed to CFITSIO to read the desired keyword name. This function will allocate space to keep the value. If `readcomment` and `readunit` are non-zero, this function will also try to read the possible comments and units of the keyword.

Here is one example of using this function:

```
/* Allocate an array of datasets. */
gal_data_t *keysll=gal_data_array_calloc(N);

/* Make the array usable as a list too (by setting 'next'). */
for(i=0;i<N-1;++i) keysll[i].next=keysll[i+1];

/* Fill the datasets with a 'name' and a 'type'. */
keysll[0].name="NAME1";     keysll[0].type=GAL_TYPE_INT32;
keysll[1].name="NAME2";     keysll[1].type=GAL_TYPE_STRING;
...
...

/* Call this function. */
gal_fits_key_read_from_ptr(fptr, keysll, 0, 0);

/* Use the values as you like... */

/* Free all the allocated spaces. Note that 'name' wasn't allocated
```

```
   in this example, so we should explicitly set it to NULL before
   calling 'gal_data_array_free'. */
for(i=0;i<N;++i) keysll[i].name=NULL;
gal_data_array_free(keysll, N, 1);
```

If the `array` pointer of each keyword's dataset is not `NULL`, then it is assumed that the space to keep the value has already been allocated. If it is `NULL`, space will be allocated for the value by this function.

Strings need special consideration: the reason is that generally, `gal_data_t` needs to also allow for array of strings (as it supports arrays of integers for example). Hence when reading a string value, two allocations may be done by this function (one if `array!=NULL`).

Therefore, when using the values of strings after this function, `keysll[i].array` must be interpreted as `char **`: one allocation for the pointer, one for the actual characters. If you use something like the example, above you don't have to worry about the freeing, `gal_data_array_free` will free both allocations. So to read a string, one easy way would be the following:

```
char *str, **strarray;
strarr = keysll[i].array;
str    = strarray[0];
```

If CFITSIO is unable to read a keyword for any reason the `status` element of the respective `gal_data_t` will be non-zero. If it is zero, then the keyword was found and successfully read. Otherwise, it is a CFITSIO status value. You can use CFITSIO's error reporting tools or `gal_fits_io_error` (see Section 11.3.11.1 [FITS Macros, errors and filenames], page 374) for reporting the reason of the failure. A tip: when the keyword doesn't exist, CFITSIO's status value will be `KEY_NO_EXIST`.

CFITSIO will start searching for the keywords from the last place in the header that it searched for a keyword. So it is much more efficient if the order that you ask for keywords is based on the order they are stored in the header.

`void` [Function]
`gal_fits_key_read` (*char* `*filename`, *char* `*hdu`, *gal_data_t* `*keysll`, *int* `readcomment`, *int* `readunit`)

Same as `gal_fits_read_keywords_fptr` (see above), but accepts the filename and HDU as input instead of an already opened CFITSIO `fitsfile` pointer.

`void` [Function]
`gal_fits_key_list_add` (*gal_fits_list_key_t* `**list`, *uint8_t* `type`, *char* `*keyname`, *int* `kfree`, *void* `*value`, *int* `vfree`, *char* `*comment`, *int* `cfree`, *char* `*unit`)

Add a keyword to the top of list of header keywords that need to be written into a FITS file. In the end, the keywords will have to be freed, so it is important to know before hand if they were allocated or not (hence the presence of the arguments ending in `free`). If the space for the respective element is not allocated, set these arguments to `0` (zero).

Important note for strings: the value should be the pointer to the string its-self (`char *`), not a pointer to a pointer (`char **`).

`void` [Function]
`gal_fits_key_list_add_end` (*gal_fits_list_key_t* `**list`, *uint8_t* `type`, *char* `*keyname`, *int* `kfree`, *void* `*value`, *int* `vfree`, *char* `*comment`, *int* `cfree`, *char* `*unit`)

Similar to `gal_fits_key_list_add` (see above) but add the keyword to the end of the list. Use this function if you want the keywords to be written in the same order that you add nodes to the list of keywords.

`void` [Function]
`gal_fits_key_list_reverse` (*gal_fits_list_key_t* `**list`)

Reverse the input list of keywords.

`void` [Function]
`gal_fits_key_write_title_in_ptr` (*char* `*title`, *fitsfile* `*fptr`)

Add two lines of "title" keywords to the given CFITSIO `fptr` pointer. The first line will be blank and the second will have the string in `title` roughly in the middle of the line (a fixed distance from the start of the keyword line). A title in the list of keywords helps in classifying the keywords into groups and inspecting them by eye. If `title==NULL`, this function won't do anything.

`void` [Function]
`gal_fits_key_write_filename` (*char* `*keynamebase`, *char* `*filename`, *gal_fits_list_key_t* `**list`, *int* `top1end0`)

Put `filename` into the `gal_fits_list_key_t` list (possibly broken up into multiple keywords) to later write into a HDU header. The `keynamebase` string will be appended with a `_N` (N>0) and used as the keyword name. If `top1end0!=0`, then the keywords containing the filename will be added to the top of the list.

The FITS standard sets a maximum length for the value of a keyword. This creates problems with file names (which include directories). Because file names/addresses can become very long. Therefore, when `filename` is longer than the maximum length of a FITS keyword value, this function will break it into several keywords (breaking up the string on directory separators).

`void` [Function]
`gal_fits_key_write_wcsstr` (*fitsfile* `*fptr`, *char* `*wcsstr`, *int* `nkeyrec`)

Write the WCS header string (produced with WCSLIB's `wcshdo` function) into the CFITSIO `fitsfile` pointer. `nkeyrec` is the number of FITS header keywords in `wcsstr`. This function will put a few blank keyword lines along with a comment `WCS information` before writing each keyword record.

`void` [Function]
`gal_fits_key_write` (*gal_fits_list_key_t* `**keylist`, *char* `*title`, *char* `*filename`, *char* `*hdu`)

Write the list of keywords in `keylist` into the `hdu` extension of the file called `filename` (it must already exist).

`void` [Function]
`gal_fits_key_write_in_ptr` (*gal_fits_list_key_t* `**keylist`, *fitsfile* `*fptr`)

Write the list of keywords in `keylist` into the given CFITSIO `fitsfile` pointer.

`void` [Function]
`gal_fits_key_write_version` (*gal_fits_list_key_t* `**keylist`, *char* `*title`, *char* `*filename`, *char* `*hdu`)

Write the (optional, when `keylist!=NULL`) given list of keywords under the optional FITS keyword `title`, then print all the important version and date information. This is basically, just a wrapper over `gal_fits_key_write_version_in_ptr`.

`void` [Function]
`gal_fits_key_write_version_in_ptr` (*gal_fits_list_key_t* `**keylist`, *char* `*title`, *fitsfile* `*fptr`)

Write or update (all the) keyword(s) in `headers` into the FITS pointer, but also the date, name of your program (`program_name`), along with the versions of CFITSIO, WCSLIB (when available), GSL, Gnuastro, and (the possible) commit information into the header as described in Section 4.9 [Output FITS files], page 125.

Since the data processing depends on the versions of the libraries you have used, it is strongly recommended to include this information in every FITS output. `gal_fits_img_write` and `gal_fits_tab_write` will automatically use this function.

`void` [Function]
`gal_fits_key_write_config` (*gal_fits_list_key_t* `**keylist`, *char* `*title`, *char* `*extname`, *char* `*filename`, *char* `*hdu`)

Write the given keyword list (`keylist`) into the `hdu` extension of `filename`, ending it with version information. This function will write `extname` as the name of the extension (value to the standard `EXTNAME` FITS keyword). The list of keywords will then be printed under a title called `title`.

This function is used by many Gnuastro programs and is primarily intended for writing configuration settings of a program into the zero-th extension of their FITS outputs (which is empty when the FITS file is created by Gnuastro's program and this library).

11.3.11.5 FITS arrays (images)

Images (or multi-dimensional arrays in general) are one of the common data formats that is stored in FITS files. Only one image may be stored in each FITS HDU/extension. The functions described here can be used to get the information of, read, or write images in FITS files.

`void` [Function]
`gal_fits_img_info` (*fitsfile* `*fptr`, *int* `*type`, *size_t* `*ndim`, *size_t* `**dsize`, *char* `**name`, *char* `**unit`)

Read the type (see Section 11.3.3 [Library data types (`type.h`)], page 337), number of dimensions, and size along each dimension of the CFITSIO `fitsfile` into the `type`, `ndim`, and `dsize` pointers respectively. If `name` and `unit` are not `NULL` (point to a `char *`), then if the image has a name and units, the respective string will be put in these pointers.

`gal_data_t *` [Function]
`gal_fits_img_read` (*char* `*filename`, *char* `*hdu`, *size_t* `minmapsize`)

Read the contents of the `hdu` extension/HDU of `filename` into a Gnuastro generic data container (see Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346) and return it. If the necessary space is larger than `minmapsize`, then don't keep the data in RAM, but in a file on the HDD/SSD. For more on `minmapsize` see the description under the same name in Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346.

Note that this function only reads the main data within the requested FITS extension, the WCS will not be read into the returned dataset. To read the WCS, you can use `gal_wcs_read` function as shown below. Afterwards, the `gal_data_free` function will free both the dataset and any WCS structure (if there are any).

```
data=gal_fits_img_read(filename, hdu, -1);
data->wcs=gal_wcs_read(filename, hdu, 0, 0, &data->wcs->nwcs);
```

`gal_data_t *` [Function]
`gal_fits_img_read_to_type` (*char* `*inputname`, *char* `*inhdu`, *uint8_t* `type`, *size_t* `minmapsize`)

Read the contents of the `hdu` extension/HDU of `filename` into a Gnuastro generic data container (see Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346) of type `type` and return it.

This is just a wrapper around `gal_fits_img_read` (to read the image/array of any type) and `gal_data_copy_to_new_type_free` (to convert it to `type` and free the initially read dataset). See the description there for more.

`gal_data_t *` [Function]
`gal_fits_img_read_kernel` (*char* `*filename`, *char* `*hdu`, *size_t* `minmapsize`)

Read the `hdu` of `filename` as a convolution kernel. A convolution kernel must have an odd size along all dimensions, it must not have blank (NaN in floating point types) values and must be flipped around the center to make the proper convolution (see Section 6.3.1.1 [Convolution process], page 178). If there are blank values, this function will change the blank values to `0.0`. If the input image doesn't have the other two requirements, this function will abort with an error describing the condition to the user. The finally returned dataset will have a `float32` type.

`fitsfile *` [Function]
`gal_fits_img_write_to_ptr` (*gal_data_t* `*input`, *char* `*filename`)

Write the `input` dataset into a FITS file named `filename` and return the corresponding CFITSIO `fitsfile` pointer. This function won't close `fitsfile`, so you can still add other extensions to it after this function or make other modifications.

`void` [Function]
`gal_fits_img_write` (*gal_data_t* `*data`, *char* `*filename`, *gal_fits_list_key_t* `*headers`, *char* `*program_string`)

Write the `input` dataset into the FITS file named `filename`. Also add the `headers` keywords to the newly created HDU/extension it along with your program's name (`program_string`).

`void` [Function]
`gal_fits_img_write_to_type` (*gal_data_t* `*data`, *char* `*filename`, *gal_fits_list_key_t* `*headers`, *char* `*program_string`, *int* `type`)

Convert the `input` dataset into `type`, then write it into the FITS file named `filename`. Also add the `headers` keywords to the newly created HDU/extension along with your program's name (`program_string`). After the FITS file is written, this function will free the copied dataset (with type `type`) from memory.

This is just a wrapper for the `gal_data_copy_to_new_type` and `gal_fits_img_write` functions.

`void` [Function]
`gal_fits_img_write_corr_wcs_str` (*gal_data_t* `*data`, *char* `*filename`, *char* `*wcsstr`, *int* `nkeyrec`, *double* `*crpix`, *gal_fits_list_key_t* `*headers`, *char* `*program_string`)

Write the `input` dataset into `filename` using the `wcsstr` while correcting the `CRPIX` values.

This function is mainly useful when you want to make FITS files in parallel (from one main WCS structure, with just differing CRPIX). This can happen in the following cases for example:

- When a large number of FITS images (with WCS) need to be created in parallel, it can be much more efficient to write the header's WCS keywords once at first, write them in the FITS file, then just correct the CRPIX values.
- WCSLIB's header writing function is not thread safe. So when writing FITS images in parallel, we can't write the header keywords in each thread.

11.3.11.6 FITS tables

Tables are one of the common formats of data that is stored in FITS files. Only one table may be stored in each FITS HDU/extension, but each table column must be viewed as a different dataset (with its own name, units and numeric data type for example). The only constraint of the column datasets in a table is that they must be one-dimensional and have the same number of elements as the other columns. The functions described here can be used to get the information of, read, or write columns into FITS tables.

`void` [Function]
`gal_fits_tab_size` (*fitsfile* `*fitsptr`, *size_t* `*nrows`, *size_t* `*ncols`)

Read the number of rows and columns in the table within CFITSIO's `fitsptr`.

`int` [Function]
`gal_fits_tab_format` (*fitsfile* `*fitsptr`)

Return the format of the FITS table contained in CFITSIO's `fitsptr`. Recall that FITS tables can be in binary or ASCII formats. This function will return `GAL_TABLE_FORMAT_AFITS` or `GAL_TABLE_FORMAT_BFITS` (defined in Section 11.3.10 [Table input output (`table.h`)], page 370). If the `fitsptr` is not a table, this function will abort the program with an error message informing the user of the problem.

`gal_data_t *` [Function]
`gal_fits_tab_info` (*char* `*filename`, *char* `*hdu`, *size_t* `*numcols`, *size_t* `*numrows`, *int* `*tableformat`)

Store the information of each column in `hdu` of `filename` into an array of data structures with `numcols` elements (one data structure for each column) see Section 11.3.6.3 [Arrays of datasets], page 352. The total number of rows in the table is also put into the memory that `numrows` points to. The format of the table (e.g., FITS binary or ASCII table) will be put in `tableformat` (macros defined in Section 11.3.10 [Table input output (`table.h`)], page 370).

This function is just for column information. Therefore it only stores meta-data like column name, units and comments. No actual data (contents of the columns for example the `array` or `dsize` elements) will be allocated by this function. This is a low-level function particular to reading tables in FITS format. To be generic, it is recommended to use `gal_table_info` which will allow getting information from a variety of table formats based on the filename (see Section 11.3.10 [Table input output (`table.h`)], page 370).

`gal_data_t *` [Function]
`gal_fits_tab_read` (*char* `*filename`, *char* `*hdu`, *size_t* `numrows`, *gal_data_t* `*colinfo`, *gal_list_sizet_t* `*indexll`, *size_t* `minmapsize`)

Read the columns given in the list `indexll` from a FITS table (in `filename` and HDU/extension `hdu`) into the returned linked list of data structures, see Section 11.3.8.3 [List of `size_t`], page 360, and Section 11.3.8.9 [List of `gal_data_t`], page 368. If the necessary space for each column is larger than `minmapsize`, don't keep it in the RAM, but in a file in the HDD/SSD, see the description under the same name in Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346.

Each column will have `numrows` rows and `colinfo` contains any further information about the columns (returned by `gal_fits_tab_info`, described above). Note that this is a low-level function, so the output data linked list is the inverse of the input indexs linked list. It is recommended to use `gal_table_read` for generic reading of tables, see Section 11.3.10 [Table input output (`table.h`)], page 370.

`void` [Function]
`gal_fits_tab_write` (*gal_data_t* `*cols`, *gal_list_str_t* `*comments`, *int* `tableformat`, *char* `*filename`, *char* `*extname`)

Write the list of datasets in `cols` (see Section 11.3.8.9 [List of `gal_data_t`], page 368) as separate columns in a FITS table in `filename`. If `filename` already exists then this function will write the table as a new extension called `extname`, after all existing ones. The format of the table (ASCII or binary) may be specified with the `tableformat` (see Section 11.3.10 [Table input output (`table.h`)], page 370). If `comments!=NULL`, each node of the list of strings will be written as a `COMMENT` keywords in the output FITS file (see Section 11.3.8.1 [List of strings], page 358.

This is a low-level function for tables. It is recommended to use `gal_table_write` for generic writing of tables in a variety of formats, see Section 11.3.10 [Table input output (`table.h`)], page 370.

11.3.12 File input output

The most commonly used file format in astronomical data analysis is the FITS format (see Section 5.1 [Fits], page 128, for an introduction), therefore Gnuastro's library provides a large and separate collection of functions to read/write data from/to them (see Section 11.3.11 [FITS files (`fits.h`)], page 374). However, FITS is not well recognized outside the astronomical community and cannot be imported into documents or slides. Therefore, in this section, we discuss the other different file formats that Gnuastro's library recognizes.

11.3.12.1 Text files (`txt.h`)

The most universal and portable format for data storage are plain text files. They can be viewed and edited on any text editor or even on the command-line. This section are describes some functions that help in reading from and writing to plain text files.

Lines are one of the most basic building blocks (delimiters) of a text file. Some operating systems like Microsoft Windows, terminate their ASCII text lines with a carriage return character and a new-line character (two characters, also known as CRLF line terminators). While Unix-like operating systems just use a single new-line character. The functions below that read an ASCII text file are able to identify lines with both kinds of line terminators.

Gnuastro defines a simple format for metadata of table columns in a plain text file that is discussed in Section 4.6.2 [Gnuastro text table format], page 119. The functions to get information from, read from and write to plain text files also follow those conventions.

`GAL_TXT_LINESTAT_INVALID` [Macro]
`GAL_TXT_LINESTAT_BLANK` [Macro]
`GAL_TXT_LINESTAT_COMMENT` [Macro]
`GAL_TXT_LINESTAT_DATAROW` [Macro]

Status codes for lines in a plain text file that are returned by `gal_txt_line_stat`. Lines which have a `#` character as their first non-white character are considered to be comments. Lines with nothing but white space characters are considered blank. The remaining lines are considered as containing data.

`int` [Function]
`gal_txt_line_stat` (*char* `*line`)

Check the contents of `line` and see if it is a blank, comment, or data line. The returned values are the macros that start with `GAL_TXT_LINESTAT`.

`gal_data_t *` [Function]
`gal_txt_table_info` (*char* `*filename`, *gal_list_str_t* `*lines`, *size_t* `*numcols`, *size_t* `*numrows`)

Store the information of each column in a text file `filename`, or list of strings (`lines`) into an array of data structures with `numcols` elements (one data structure for each column) see Section 11.3.6.3 [Arrays of datasets], page 352. The total number of rows in the table is also put into the memory that `numrows` points to.

`lines` is a list of strings with each node representing one line (including the new-line character), see Section 11.3.8.1 [List of strings], page 358. It will mostly be the output of `gal_txt_stdin_read`, which is used to read the program's input as separate lines

from the standard input (see below). Note that `filename` and `lines` are mutually exclusive and one of them must be `NULL`.

This function is just for column information. Therefore it only stores meta-data like column name, units and comments. No actual data (contents of the columns for example the `array` or `dsize` elements) will be allocated by this function. This is a low-level function particular to reading tables in plain text format. To be generic, it is recommended to use `gal_table_info` which will allow getting information from a variety of table formats based on the filename (see Section 11.3.10 [Table input output (`table.h`)], page 370).

`gal_data_t *` [Function]
`gal_txt_table_read` (*char* `*filename`, *gal_list_str_t* `*lines`, *size_t* `numrows`, *gal_data_t* `*colinfo`, *gal_list_sizet_t* `*indexll`, *size_t* `minmapsize`)

Read the columns given in the list `indexll` from a plain text file (`filename`) or list of strings (`lines`), into a linked list of data structures (see Section 11.3.8.3 [List of `size_t`], page 360, and Section 11.3.8.9 [List of `gal_data_t`], page 368). If the necessary space for each column is larger than `minmapsize`, don't keep it in the RAM, but in a file on the HDD/SSD, see the description under the same name in Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346.

`lines` is a list of strings with each node representing one line (including the new-line character), see Section 11.3.8.1 [List of strings], page 358. It will mostly be the output of `gal_txt_stdin_read`, which is used to read the program's input as separate lines from the standard input (see below). Note that `filename` and `lines` are mutually exclusive and one of them must be `NULL`.

Note that this is a low-level function, so the output data list is the inverse of the input indexs linked list. It is recommended to use `gal_table_read` for generic reading of tables in any format, see Section 11.3.10 [Table input output (`table.h`)], page 370.

`gal_data_t *` [Function]
`gal_txt_image_read` (*char* `*filename`, *gal_list_str_t* `*lines`, *size_t* `minmapsize`)

Read the 2D plain text dataset in file (`filename`) or list of strings (`lines`) into a dataset and return the dataset. If the necessary space for the image is larger than `minmapsize`, don't keep it in the RAM, but in a file on the HDD/SSD, see the description under the same name in Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346.

`lines` is a list of strings with each node representing one line (including the new-line character), see Section 11.3.8.1 [List of strings], page 358. It will mostly be the output of `gal_txt_stdin_read`, which is used to read the program's input as separate lines from the standard input (see below). Note that `filename` and `lines` are mutually exclusive and one of them must be `NULL`.

`gal_list_str_t *` [Function]
`gal_txt_stdin_read` (*long* `timeout_microsec`)

Read the complete standard input and return a list of strings with each line (including the new-line character) as one node of that list. If the standard input is already filled (for example connected to another program's output with a pipe), then this function will parse the whole stream.

If Standard input is not pre-configured and the *first line* is typed/written in the terminal before `timeout_microsec` micro-seconds, it will continue parsing until reaches an end-of-file character (`CTRL-D` after a new-line on the keyboard) with no time limit. If nothing is entered before `timeout_microsec` micro-seconds, it will return `NULL`.

All the functions that can read plain text tables will accept a filename as well as a list of strings (intended to be the output of this function for using Standard input). The reason for keeping the standard input is that once something is read from the standard input, it is hard to put it back. We often need to read a text file several times: once to count how many columns it has and which ones are requested, and another time to read the desired columns. So it easier to keep it all in allocated memory and pass it on from the start for each round.

`void` [Function]
`gal_txt_write` (*gal_data_t* `*cols`, *gal_list_str_t* `*comment`, *char* `*filename`, *uint8_t* `colinfoinstdout`)

Write `cols` in a plain text file `filename`. `cols` may have one or two dimensions which determines the output:

1D `cols` is treated as a column and a list of datasets (see Section 11.3.8.9 [List of `gal_data_t`], page 368): every node in the list is written as one column in a table.

2D `cols` is a two dimensional array, it cannot be treated as a list (only one 2D array can currently be written to a text file). So if `cols->next!=NULL` the next nodes in the list are ignored and will not be written.

This is a low-level function for tables. It is recommended to use `gal_table_write` for generic writing of tables in a variety of formats, see Section 11.3.10 [Table input output (`table.h`)], page 370.

If `filename` already exists this function will abort with an error and will not write over the existing file. Before calling this function make sure if the file exists or not. If `comments!=NULL`, a `#` will be put at the start of each node of the list of strings and will be written in the file before the column meta-data in `filename` (see Section 11.3.8.1 [List of strings], page 358).

When `filename==NULL`, the column information will be printed on the standard output (command-line). When `colinfoinstdout!=0` and `filename==NULL` (columns are printed in the standard output), the dataset metadata will also printed in the standard output. When printing to the standard output, the column information can be piped into another program for further processing and thus the meta-data (lines starting with a `#`) must be ignored. In such cases, you only print the column values by passing `0` to `colinfoinstdout`.

11.3.12.2 TIFF files (`tiff.h`)

Outside of astronomy, the TIFF standard is arguably the most commonly used format to store high-precision data/images. Unlike FITS however, the TIFF standard only supports images (not tables), but like FITS, it has support for all standard data types (see Section 4.5 [Numeric data types], page 115) which is the primary reason other fields use it.

Another similarity of the TIFF and FITS standards is that TIFF supports multiple images in one file. The TIFF standard calls each one of these images (and their accompanying meta-data) a 'directory' (roughly equivalent to the FITS extensions). Unlike FITS however, the directories can only be identified by their number (counting from zero), recall that in FITS you can also use the extension name to identify it.

The functions described here allow easy reading (and later writing) of TIFF files within Gnuastro or for users of Gnuastro's libraries. Currently only reading is supported, but if you are interested, please get in touch with us.

`int` [Function]
`gal_tiff_name_is_tiff` (*char* `*name`)
Return `1` if `name` has a TIFF suffix. This can be used to make sure that a given input file is TIFF. See `gal_tiff_suffix_is_tiff` for a list of recognized suffixes.

`int` [Function]
`gal_tiff_suffix_is_tiff` (*char* `*name`)
Return `1` if `suffix` is a recognized TIFF suffix. The recognized suffixes are `tif`, `tiff`, `TIFF` and `TIFF`.

`size_t` [Function]
`gal_tiff_dir_string_read` (*char* `*string`)
Return the number within `string` as a `size_t` number to identify a TIFF directory. Note that the directories start counting from zero.

`gal_data_t *` [Function]
`gal_tiff_read` (*char* `*filename`, *size_t* `dir`, *size_t* `minmapsize`)
Read the `dir` directory within the TIFF file `filename` and return the contents of that TIFF directory as `gal_data_t`. If the directory's image contains multiple channels, the output will be a list (see Section 11.3.8.9 [List of `gal_data_t`], page 368).

11.3.12.3 JPEG files (`jpeg.h`)

The JPEG file format is one of the most common formats for storing and transferring images, recognized by almost all image rendering and processing programs. In particular, because of its lossy compression algorithm, JPEG files can have low volumes, making it used heavily on the internet. For more on this file format, and a comparison with others, please see Section 5.2.1 [Recognized file formats], page 138.

For scientific purposes, the lossy compression and very limited dynamic range (8-bit integers) make JPEG very unattractive for storing of valuable data. However, because of its commonality, it will inevitably be needed in some situations. The functions here can be used to read and write JPEG images into Gnuastro's Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346. If the JPEG file has more than one color channel, each channel is treated as a separate node in a list of datasets (see Section 11.3.8.9 [List of `gal_data_t`], page 368).

`int` [Function]
`gal_jpeg_name_is_jpeg` (*char* `*name`)
Return `1` if `name` has a JPEG suffix. This can be used to make sure that a given input file is JPEG. See `gal_jpeg_suffix_is_jpeg` for a list of recognized suffixes.

`int` [Function]
`gal_jpeg_suffix_is_jpeg` (*char* `*name`)
Return `1` if `suffix` is a recognized JPEG suffix. The recognized suffixes are `.jpg`, `.JPG`, `.jpeg`, `.JPEG`, `.jpe`, `.jif`, `.jfif` and `.jfi`.

`gal_data_t *` [Function]
`gal_jpeg_read` (*char* `*filename`, *size_t* `minmapsize`)
Read the JPEG file `filename` and return the contents as `gal_data_t`. If the directory's image contains multiple colors/channels, the output will be a list with one node per color/channel (see Section 11.3.8.9 [List of `gal_data_t`], page 368).

`void` [Function]
`gal_jpeg_write` (*gal_data_t* `*in`, *char* `*filename`, *uint8_t* `quality`, *float* `widthincm`)
Write the given dataset (`in`) into `filename` (a JPEG file). If `in` is a list, then each node in the list will be a color channel, therefore there can only be 1, 3 or 4 nodes in the list. If the number of nodes is different, then this function will abort the program with a message describing the cause. The lossy JPEG compression level can be set through `quality` which is a value between 0 and 100 (inclusive, 100 being the best quality). The display width of the JPEG file in units of centimeters (to suggest to viewers/users, only a meta-data) can be set through `widthincm`.

11.3.12.4 EPS files (`eps.h`)

The Encapsulated PostScript (EPS) format is commonly used to store images (or individual/single-page parts of a document) in the PostScript documents. For a more complete introduction, please see Section 5.2.1 [Recognized file formats], page 138. To provide high quality graphics, the Postscript language is a vectorized format, therefore pixels (elements of a "rasterized" format) aren't defined in their context.

To display rasterized images, PostScript does allow arrays of pixels. However, since the over-all EPS file may contain many vectorized elements (for example borders, text, or other lines over the text) and interpreting them is not trivial or necessary within Gnuastro's scope, Gnuastro only provides some functions to write a dataset (in the `gal_data_t` format, see Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346) into EPS.

`int` [Function]
`gal_eps_name_is_eps` (*char* `*name`)
Return `1` if `name` has an EPS suffix. This can be used to make sure that a given input file is EPS. See `gal_eps_suffix_is_eps` for a list of recognized suffixes.

`int` [Function]
`gal_eps_suffix_is_eps` (*char* `*name`)
Return `1` if `suffix` is a recognized EPS suffix. The recognized suffixes are `.eps`, `.EPS`, `.epsf`, `.epsi`.

`void` [Function]
`gal_eps_to_pt` (*float* `widthincm`, *size_t* `*dsize`, *size_t* `*w_h_in_pt`)
Given a specific width in centimeters (`widthincm` and the number of the dataset's pixels in each dimension (`dsize`) calculate the size of the output in PostScript points.

The output values are written in the `w_h_in_pt` array (which has to be allocated before calling this function). The first element in `w_h_in_pt` is the width and the second is the height of the image.

`void` [Function]
`gal_eps_write` (*gal_data_t* `*in`, *char* `*filename`, *float* `widthincm`, *uint32_t* `borderwidth`, *int* `hex`, *int* `dontoptimize`, *int* `forpdf`)

Write the `in` dataset into an EPS file called `filename`. `in` has to be an unsigned 8-bit character type (`GAL_TYPE_UINT8`, see Section 4.5 [Numeric data types], page 115). The desired width of the image in human/non-pixel units (to help the displayer) can be set with the `widthincm` argument. If `borderwidth` is non-zero, it is interpreted as the width (in points) of a solid black border around the image. A border can helpful when importing the EPS file into a document.

EPS files are plain-text (can be opened/edited in a text editor), therefore there are different encodings to store the data (pixel values) within them. Gnuastro supports the Hexadecimal and ASCII85 encoding. ASCII85 is more efficient (producing small file sizes), so it is the default encoding. To use Hexadecimal encoding, set `hex` to a non-zero value. Currently If you don't directly want to import the EPS file into a PostScript document but want to later compile it into a PDF file, set the `forpdf` argument to `1`.

By default, when the dataset only has two values, this function will use the PostScript optimization that allows setting the pixel values per bit, not byte (Section 5.2.1 [Recognized file formats], page 138). This can greatly help reduce the file size. However, when `dontoptimize!=0`, this optimization is disabeled: even though there are only two values (is binary), the difference between them does not correspond to the full contrast of black and white.

11.3.12.5 PDF files (`pdf.h`)

The portable document format (PDF) has arguably become the most common format used for distribution of documents. In practice, a PDF file is just a compiled PostScript file. For a more complete introduction, please see Section 5.2.1 [Recognized file formats], page 138. To provide high quality graphics, the PDF is a vectorized format, therefore pixels (elements of a "rasterized" format) aren't defined in their context. As a result, similar to Section 11.3.12.4 [EPS files (`eps.h`)], page 390, Gnuastro only writes datasets to a PDF file, not vice-versa.

`int` [Function]
`gal_pdf_name_is_pdf` (*char* `*name`)

Return `1` if `name` has an PDF suffix. This can be used to make sure that a given input file is PDF. See `gal_pdf_suffix_is_pdf` for a list of recognized suffixes.

`int` [Function]
`gal_pdf_suffix_is_pdf` (*char* `*name`)

Return `1` if `suffix` is a recognized PDF suffix. The recognized suffixes are `.pdf` and `.PDF`.

`void` [Function]
`gal_pdf_write` (*gal_data_t* `*in`, *char* `*filename`, *float* `widthincm`, *uint32_t* `borderwidth`, *int* `dontoptimize`)

Write the `in` dataset into an EPS file called `filename`. `in` has to be an unsigned 8-bit character type (`GAL_TYPE_UINT8`, see Section 4.5 [Numeric data types], page 115). The desired width of the image in human/non-pixel units (to help the displayer) can be set with the `widthincm` argument. If `borderwidth` is non-zero, it is interpreted as the width (in points) of a solid black border around the image. A border can helpful when importing the PDF file into a document.

This function is just a wrapper for the `gal_eps_write` function in Section 11.3.12.4 [EPS files (`eps.h`)], page 390. After making the EPS file, Ghostscript (with a version of 9.10 or above, see Section 3.1.2 [Optional dependencies], page 64) will be used to compile the EPS file to a PDF file. Therefore if GhostScript doesn't exist, doesn't have the proper version, or fails for any other reason, the EPS file will remain. It can be used to find the cause, or use another converter or PostScript compiler.

By default, when the dataset only has two values, this function will use the PostScript optimization that allows setting the pixel values per bit, not byte (Section 5.2.1 [Recognized file formats], page 138). This can greatly help reduce the file size. However, when `dontoptimize!=0`, this optimization is disabeled: even though there are only two values (is binary), the difference between them does not correspond to the full contrast of black and white.

11.3.13 World Coordinate System (`wcs.h`)

The FITS standard defines the world coordinate system (WCS) as a mechanism to associate physical values to positions within a dataset. For example, it can be used to convert pixel coordinates in an image to celestial coordinates like the right ascension and declination. The functions in this section are mainly just wrappers over CFITSIO, WCSLIB and GSL library functions to help in common applications.

`struct wcsprm *` [Function]
`gal_wcs_read_fitsptr` (*fitsfile* `*fptr`, *size_t* `hstartwcs`, *size_t* `hendwcs`, *int* `*nwcs`)

[**Not thread-safe**] Return the WCSLIB `wcsprm` structure that is read from the CFITSIO `fptr` pointer to an opened FITS file. Also put the number of coordinate representations found into the space that `nwcs` points to. To read the WCS structure directly from a filename, see `gal_wcs_read` below. After processing has finished, you can free the returned structure with WCSLIB's `wcsvfree` keyword:

```
status = wcsvfree(&nwcs,&wcs);
```

If you don't want to search the full FITS header for WCS-related FITS keywords (for example due to conflicting keywords), but only a specific range of the header keywords you can use the `hstartwcs` and `hendwcs` arguments to specify the keyword number range (counting from zero). If `hendwcs` is larger than `hstartwcs`, then only keywords in the given range will be checked. Hence, to ignore this feature (and search the full FITS header), give both these arguments the same value.

If the WCS information couldn't be read from the FITS file, this function will return a `NULL` pointer and put a zero in `nwcs`. A WCSLIB error message will also be printed in `stderr` if there was an error.

This function is just a wrapper over WCSLIB's `wcspih` function which is not thread-safe. Therefore, be sure to not call this function simultaneously (over multiple threads).

`struct wcsprm *` [Function]
`gal_wcs_read` (*char* `*filename`, *char* `*hdu`, *size_t* `hstartwcs`, *size_t* `hendwcs`, *int* `*nwcs`)

[**Not thread-safe**] Return the WCSLIB structure that is read from the HDU/extension `hdu` of the file `filename`. Also put the number of coordinate representations found into the space that `nwcs` points to. Please see `gal_wcs_read_fitsptr` for more.

`struct wcsprm *` [Function]
`gal_wcs_copy` (*struct wcsprm* `*wcs`)

Return a fully allocated (independent) copy of `wcs`.

`void` [Function]
`gal_wcs_on_tile` (*gal_data_t* `*tile`)

Create a WCSLIB `wcsprm` structure for `tile` using WCS parameters of the tile's allocated block dataset, see Section 11.3.15 [Tessellation library (`tile.h`)], page 399, for the definition of tiles. If `tile` already has a WCS structure, this function won't do anything.

In many cases, tiles are created for internal/low-level processing. Hence for performance reasons, when creating the tiles they don't have any WCS structure. When needed, this function can be used to add a WCS structure to each tile tile by copying the WCS structure of its block and correcting the reference point's coordinates within the tile.

`double *` [Function]
`gal_wcs_warp_matrix` (*struct wcsprm* `*wcs`)

Return the Warping matrix of the given WCS structure as an array of double precision floating points. This will be the final matrix, irrespective of the type of storage in the WCS structure. Recall that the FITS standard has several methods to store the matrix. The output is an allocated square matrix with each side equal to the number of dimensions.

`void` [Function]
`gal_wcs_decompose_pc_cdelt` (*struct wcsprm* `*wcs`)

Decompose the `PCi_j` and `CDELTi` elements of `wcs`. According to the FITS standard, in the `PCi_j` WCS formalism, the rotation matrix elements m_{ij} are encoded in the `PCi_j` keywords and the scale factors are encoded in the `CDELTi` keywords. There is also another formalism (the `CDi_j` formalism) which merges the two into one matrix.

However, WCSLIB's internal operations are apparently done in the `PCi_j` formalism. So its outputs are also all in that format by default. When the input is a `CDi_j`, WCSLIB will still read the matrix directly into the `PCi_j` matrix and the `CDELTi` values are set to `1` (one). This function is designed to correct such issues: after it is

finished, the `CDELTi` values in `wcs` will correspond to the pixel scale, and the `PCi_j` will correction show the rotation.

`double` [Function]
`gal_wcs_angular_distance_deg` (*double* `r1`, *double* `d1`, *double* `r2`, *double* `d2`)
Return the angular distance (in degrees) between a point located at (`r1`, `d1`) to (`r2`, `d2`). All input coordinates are in degrees. The distance (along a great circle) on a sphere between two points is calculated with the equation below.

$$\cos(d) = \sin(d_1)\sin(d_2) + \cos(d_1)\cos(d_2)\cos(r_1 - r_2)$$

However, since the the pixel scales are usually very small numbers, this function won't use that direct formula. It will be use the Haversine formula (`https://en.wikipedia.org/wiki/Haversine_formula`) which is better considering floating point errors:

$$\frac{\sin^2(d)}{2} = \sin^2\left(\frac{d_1 - d_2}{2}\right) + \cos(d_1)\cos(d_2)\sin^2\left(\frac{r_1 - r_2}{2}\right)$$

`double *` [Function]
`gal_wcs_pixel_scale` (*struct wcsprm* `*wcs`)
Return the pixel scale for each dimension of `wcs` in degrees. The output is an allocated array of double precision floating point type with one element for each dimension. If its not successful, this function will return `NULL`.

`double` [Function]
`gal_wcs_pixel_area_arcsec2` (*struct wcsprm* `*wcs`)
Return the pixel area of `wcs` in arcsecond squared. If the input WCS structure is not two dimensional and the units (`CUNIT` keywords) are not `deg` (for degrees), then this function will return a NaN.

`gal_data_t *` [Function]
`gal_wcs_world_to_img` (*gal_data_t* `*coords`, *struct wcsprm* `*wcs`, *int* `inplace`)
Convert the linked list of world coordinates in `coords` to a linked list of image coordinates given the input WCS structure. `coords` must be a linked list of data structures of float64 ('double') type, seeSection 11.3.8 [Linked lists (`list.h`)], page 357, and Section 11.3.8.9 [List of `gal_data_t`], page 368. The top (first popped/read) node of the linked list must be the first WCS coordinate (RA in an image usually) and etc. Similarly, the top node of the output will be the first image coordinate (in the FITS standard).

If `inplace` is zero, then the output will be a newly allocated list and the input list will be untouched. However, if `inplace` is non-zero, the output values will be written into the input's already allocated array and the returned pointer will be the same pointer to `coords` (in other words, you can ignore the returned value). Note that in the latter case, only the values will be changed, things like units or name (if present) will be untouched.

`gal_data_t *` [Function]
`gal_wcs_img_to_world` (*gal_data_t* `*coords`, *struct wcsprm* `*wcs`, *int* `inplace`)
Convert the linked list of image coordinates in `coords` to a linked list of world coordinates given the input WCS structure. See the description of `gal_wcs_world_to_img` for more details.

11.3.14 Arithmetic on datasets (`arithmetic.h`)

When the dataset's type and other information are already known, any programming language (including C) provides some very good tools for various operations (including arithmetic operations like addition) on the dataset with a simple loop. However, as an author of a program, making assumptions about the type of data, its dimensions and other basic characteristics will come with a large processing burden.

For example if you always read your data as double precision floating points for a simple operation like addition with an integer constant, you will be wasting a lot of CPU and memory when the input dataset is `int32` type for example (see Section 4.5 [Numeric data types], page 115). This overhead may be small for small images, but as you scale your process up and work with hundred/thousands of files that can be very large, this overhead will take a significant portion of the processing power. The functions and macros in this section are designed precisely for this purpose: to allow you to do any of the defined operations on any dataset with no overhead (in the native type of the dataset).

Gnuastro's Arithmetic program uses the functions and macros of this section, so please also have a look at the Section 6.2 [Arithmetic], page 161, program and in particular Section 6.2.2 [Arithmetic operators], page 162, for a better description of the operators discussed here.

The main function of this library is `gal_arithmetic` that is described below. It can take an arbitrary number of arguments as operands (depending on the operator, similar to `printf`). Its first two arguments are integers specifying the flags and operator. So first we will review the constants for the recognized flags and operators and discuss them, then introduce the actual function.

`GAL_ARITHMETIC_INPLACE` [Macro]
`GAL_ARITHMETIC_FREE` [Macro]
`GAL_ARITHMETIC_NUMOK` [Macro]
`GAL_ARITHMETIC_FLAGS_ALL` [Macro]
Bit-wise flags to pass onto `gal_arithmetic` (see below). To pass multiple flags, use the bitwise-or operator, for example `GAL_ARITHMETIC_INPLACE | GAL_ARITHMETIC_FREE`. `GAL_ARITHMETIC_FLAGS_ALL` is a combination of all flags to shorten your code if you want all flags activated. Each flag is described below:

`GAL_ARITHMETIC_INPLACE`
Do the operation in-place (in the input dataset, thus modifying it) to improve CPU and memory usage. If this flag is used, after `gal_arithmetic` finishes, the input dataset will be modified. It is thus useful if you have no more need for the input after the operation.

`GAL_ARITHMETIC_FREE`
Free (all the) input dataset(s) after the operation is done. Hence the inputs are no longer usable after `gal_arithmetic`.

`GAL_ARITHMETIC_NUMOK`
: It is acceptable to use a number and an array together. For example if you want to add all the pixels in an image with a single number you can pass this flag to avoid having to allocate a constant array the size of the image (with all the pixels having the same number).

`GAL_ARITHMETIC_OP_PLUS` [Macro]
`GAL_ARITHMETIC_OP_MINUS` [Macro]
`GAL_ARITHMETIC_OP_MULTIPLY` [Macro]
`GAL_ARITHMETIC_OP_DIVIDE` [Macro]
`GAL_ARITHMETIC_OP_LT` [Macro]
`GAL_ARITHMETIC_OP_LE` [Macro]
`GAL_ARITHMETIC_OP_GT` [Macro]
`GAL_ARITHMETIC_OP_GE` [Macro]
`GAL_ARITHMETIC_OP_EQ` [Macro]
`GAL_ARITHMETIC_OP_NE` [Macro]
`GAL_ARITHMETIC_OP_AND` [Macro]
`GAL_ARITHMETIC_OP_OR` [Macro]
Binary operators (requiring two operands) that accept datasets of any recognized type (see Section 4.5 [Numeric data types], page 115). When `gal_arithmetic` is called with any of these operators, it expects two datasets as arguments. For a full description of these operators with the same name, see Section 6.2.2 [Arithmetic operators], page 162. The first dataset/operand will be put on the left of the operator and the second will be put on the right. The output type of the first four is determined from the input types (largest type of the inputs). The rest (which are all conditional operators) will output a binary `uint8_t` (or `unsigned char`) dataset with values of either `0` (zero) or `1` (one).

`GAL_ARITHMETIC_OP_NOT` [Macro]
The logical NOT operator. When `gal_arithmetic` is called with this operator, it only expects one operand (dataset), since this is a unary operator. The output is `uint8_t` (or `unsigned char`) dataset of the same size as the input. Any non-zero element in the input will be `0` (zero) in the output and any `0` (zero) will have a value of `1` (one).

`GAL_ARITHMETIC_OP_ISBLANK` [Macro]
A unary operator with output that is `1` for any element in the input that is blank, and `0` for any non-blank element. When `gal_arithmetic` is called with this operator, it will only expect one input dataset. The output dataset will have `uint8_t` (or `unsigned char`) type.

`gal_arithmetic` with this operator is just a wrapper for the `gal_blank_flag` function of Section 11.3.5 [Library blank values (`blank.h`)], page 343, and this operator is just included for completeness in arithmetic operations. So in your program, it might be easier to just call `gal_blank_flag`.

`GAL_ARITHMETIC_OP_WHERE` [Macro]
The three-operand *where* operator thoroughly discussed in Section 6.2.2 [Arithmetic operators], page 162. When `gal_arithmetic` is called with this operator, it will only

expect three input datasets: the first (which is the same as the returned dataset) is the array that will be modified. The second is the condition dataset (that must have a `uint8_t` or `unsigned char` type), and the third is the value to be used if condition is non-zero.

As a result, note that the order of operands when calling `gal_arithmetic` with `GAL_ARITHMETIC_OP_WHERE` is the opposite of running Gnuastro's Arithmetic program with the `where` operator (see Section 6.2 [Arithmetic], page 161). This is because the latter uses the reverse-Polish notation which isn't necessary when calling a function (see Section 6.2.1 [Reverse polish notation], page 162).

`GAL_ARITHMETIC_OP_SQRT` [Macro]
`GAL_ARITHMETIC_OP_LOG` [Macro]
`GAL_ARITHMETIC_OP_LOG10` [Macro]

Unary operator functions for calculating the square root ($\sqrt{i}$), $ln(i)$ and $log(i)$ mathematic operators on each element of the input dataset. The returned dataset will have a floating point type, but its precision is determined from the input: if the input is a 64-bit floating point, the output will also be 64-bit. Otherwise, the returned dataset will be 32-bit floating point. See Section 4.5 [Numeric data types], page 115, for the respective precision.

If you want your output to be 64-bit floating point but your input is a different type, you can convert the input to a floating point type with `gal_data_copy_to_new_type` or `gal_data_copy_to_new_type_free`(see Section 11.3.6.4 [Copying datasets], page 352).

`GAL_ARITHMETIC_OP_MINVAL` [Macro]
`GAL_ARITHMETIC_OP_MAXVAL` [Macro]
`GAL_ARITHMETIC_OP_NUMBERVAL` [Macro]
`GAL_ARITHMETIC_OP_SUMVAL` [Macro]
`GAL_ARITHMETIC_OP_MEANVAL` [Macro]
`GAL_ARITHMETIC_OP_STDVAL` [Macro]
`GAL_ARITHMETIC_OP_MEDIANVAL` [Macro]

Unary operand statistical operators that will return a single value for datasets of any size. These are just wrappers around similar functions in Section 11.3.21 [Statistical operations (`statistics.h`)], page 416, and are included in `gal_arithmetic` only for completeness (to use easily in Section 6.2 [Arithmetic], page 161). In your programs, it will probably be easier if you use those `gal_statistics_` functions directly.

`GAL_ARITHMETIC_OP_ABS` [Macro]

Unary operand absolute-value operator.

`GAL_ARITHMETIC_OP_MIN` [Macro]
`GAL_ARITHMETIC_OP_MAX` [Macro]
`GAL_ARITHMETIC_OP_NUMBER` [Macro]
`GAL_ARITHMETIC_OP_SUM` [Macro]
`GAL_ARITHMETIC_OP_MEAN` [Macro]
`GAL_ARITHMETIC_OP_STD` [Macro]

`GAL_ARITHMETIC_OP_MEDIAN` [Macro]
Multi-operand statistical operations. When `gal_arithmetic` is called with any of these operators, it will expect only a single operand that will be interpreted as a list of datasets (see Section 11.3.8.9 [List of `gal_data_t`], page 368. The output will be a single dataset with each of its elements replaced by the respective statistical operation on the whole list. These operators can work on multiple threads using the `numthreads` argument. See the discussion under the `min` operator in Section 6.2.2 [Arithmetic operators], page 162.

`GAL_ARITHMETIC_OP_SIGCLIP_STD` [Macro]
`GAL_ARITHMETIC_OP_SIGCLIP_MEAN` [Macro]
`GAL_ARITHMETIC_OP_SIGCLIP_MEDIAN` [Macro]
`GAL_ARITHMETIC_OP_SIGCLIP_NUMBER` [Macro]
Similar to the operands above (including `GAL_ARITHMETIC_MIN`), except that when `gal_arithmetic` is called with these operators, it requires two arguments. The first is the list of datasets like before, and the second is the 2-element list of sigma-clipping parameters. The first element in the parameters list is the multiple of sigma and the second is the termination criteria (see Section 7.1.2 [Sigma clipping], page 209).

`GAL_ARITHMETIC_OP_POW` [Macro]
Binary operator to-power operator. When `gal_arithmetic` is called with any of these operators, it will expect two operands: raising the first by the second. This operator only accepts floating point inputs and the output is also floating point.

`GAL_ARITHMETIC_OP_BITAND` [Macro]
`GAL_ARITHMETIC_OP_BITOR` [Macro]
`GAL_ARITHMETIC_OP_BITXOR` [Macro]
`GAL_ARITHMETIC_OP_BITLSH` [Macro]
`GAL_ARITHMETIC_OP_BITRSH` [Macro]
`GAL_ARITHMETIC_OP_MODULO` [Macro]
Binary integer-only operand operators. These operators are only defined on integer data types. When `gal_arithmetic` is called with any of these operators, it will expect two operands: the first is put on the left of the operator and the second on the right. The ones starting with `BIT` are the respective bit-wise operators in C and `MODULO` is the modulo/remainder operator. For a discussion on these operators, please see Section 6.2.2 [Arithmetic operators], page 162.

The output type is determined from the input types and C's internal conversions: it is strongly recommended that both inputs have the same type (any integer type), otherwise the bit-wise behavior will be determined by your compiler.

`GAL_ARITHMETIC_OP_BITNOT` [Macro]
The unary bit-wise NOT operator. When `gal_arithmetic` is called with any of these operators, it will expect one operand of an integer type and preform the bitwise-NOT operation on it. The output will have the same type as the input.

`GAL_ARITHMETIC_OP_TO_UINT8` [Macro]
`GAL_ARITHMETIC_OP_TO_INT8` [Macro]
`GAL_ARITHMETIC_OP_TO_UINT16` [Macro]

`GAL_ARITHMETIC_OP_TO_INT16` [Macro]
`GAL_ARITHMETIC_OP_TO_UINT32` [Macro]
`GAL_ARITHMETIC_OP_TO_INT32` [Macro]
`GAL_ARITHMETIC_OP_TO_UINT64` [Macro]
`GAL_ARITHMETIC_OP_TO_INT64` [Macro]
`GAL_ARITHMETIC_OP_TO_FLOAT32` [Macro]
`GAL_ARITHMETIC_OP_TO_FLOAT64` [Macro]

Unary type-conversion operators. When `gal_arithmetic` is called with any of these operators, it will expect one operand and convert it to the requested type. Note that with these operators, `gal_arithmetic` is just a wrapper over the `gal_data_copy_to_new_type` or `gal_data_copy_to_new_type_free` that are discussed in `Copying datasets`. It accepts these operators only for completeness and easy usage in Section 6.2 [Arithmetic], page 161. So in your programs, it might be preferable to directly use those functions.

`gal_data_t *` [Function]
`gal_arithmetic` (*int* `operator`, *size_t* `numthreads`, *int* `flags`, ...)

Do the arithmetic operation of `operator` on the given operands (the third argument and any further argument). If the operator can work on multiple threads, the number of threads can be specified with `numthreads`. When the operator is single-threaded, `numthreads` will be ignored. Special conditions can also be specified with the `flag` operator (a bit-flag with bits described above, for example `GAL_ARITHMETIC_INPLACE` or `GAL_ARITHMETIC_FREE`). The acceptable values for `operator` are also defined in the macros above.

`gal_arithmetic` is a multi-argument function (like C's `printf`). In other words, the number of necessary arguments is not fixed and depends on the value to `operator`. Here are a few examples showing this variability:

```
out_1=gal_arithmetic(GAL_ARITHMETIC_OP_LOG,   1, 0, in_1);
out_2=gal_arithmetic(GAL_ARITHMETIC_OP_PLUS,  1, 0, in_1, in_2);
out_3=gal_arithmetic(GAL_ARITHMETIC_OP_WHERE, 1, 0, in_1, in_2, in_3);
```

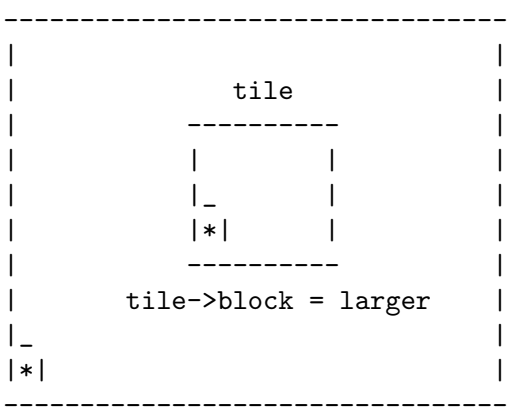
The number of necessary operands for each operator (and thus the number of necessary arguments to `gal_arithmetic`) are described above under each operator.

11.3.15 Tessellation library (`tile.h`)

In many contexts, it is desirable to slice the dataset into subsets or tiles (overlapping or not). In such a way that you can work on each tile independently. One method would be to copy that region to a separate allocated space, but in many contexts this isn't necessary and in fact can be a big burden on CPU/Memory usage. The `block` pointer in Gnuastro's Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346, is defined for such situations: where allocation is not necessary. You just want to read the data or write to it independently (or in coordination with) other regions of the dataset. Added with parallel processing, this can greatly improve the time/memory consumption.

See the figure below for example: assume the `larger` dataset is a contiguous block of memory that you are interpreting as a 2D array. But you only want to work on the smaller `tile` region.

`larger`



To use `gal_data_t`'s `block` concept, you allocate a `gal_data_t *tile` which is initialized with the pointer to the first element in the sub-array (as its `array` argument). Note that this is not necessarily the first element in the larger array. You can set the size of the tile along with the initialization as you please. Recall that, when given a non-`NULL` pointer as `array`, `gal_data_initialize` (and thus `gal_data_alloc`) do not allocate any space and just uses the given pointer for the new `array` element of the `gal_data_t`. So your `tile` data structure will not be pointing to a separately allocated space.

After the allocation is done, you just point `tile->block` to the `larger` dataset which hosts the full block of memory. Where relevant, Gnuastro's library functions will check the `block` pointer of their input dataset to see how to deal with dimensions and increments so they can always remain within the tile. The tools introduced in this section are designed to help in defining and working with tiles that are created in this manner.

Since the block structure is defined as a pointer, arbitrary levels of tessellation/grid-ing are possible (`tile->block` may itself be a tile in an even larger allocated space). Therefore, just like a linked-list (see Section 11.3.8 [Linked lists (`list.h`)], page 357), it is important to have the `block` pointer of the largest (allocated) dataset set to `NULL`. Normally, you won't have to worry about this, because `gal_data_initialize` (and thus `gal_data_alloc`) will set the `block` element to `NULL` by default, just remember not to change it. You can then only change the `block` element for the tiles you define over the allocated space.

Below, we will first review constructs for Section 11.3.15.1 [Independent tiles], page 400, and then define the current approach to fully tessellating a dataset (or covering every pixel/data-element with a non-overlapping tile grid in Section 11.3.15.2 [Tile grid], page 405. This approach to dealing with parts of a larger block was inspired from a similarly named concept in the GNU Scientific Library (GSL), see its "Vectors and Matrices" chapter for their implementation.

11.3.15.1 Independent tiles

The most general application of tiles is to treat each independently, for example they may overlap, or they may not cover the full image. This section provides functions to help in checking/inspecting such tiles. In Section 11.3.15.2 [Tile grid], page 405, we will discuss functions that define/work-with a tile grid (where the tiles don't overlap and fully cover the input dataset). Therefore, the functions in this section are general and can be used for the tiles produced by that section also.

`void` [Function]
`gal_tile_start_coord` (*gal_data_t* `*tile`, *size_t* `*start_coord`)
Calculate the starting coordinates of a tile in its allocated block of memory and write them in the memory that `start_coord` points to (which must have `tile->ndim` elements).

`void` [Function]
`gal_tile_start_end_coord` (*gal_data_t* `*tile`, *size_t* `*start_end`, *int* `rel_block`)
Put the starting and ending (end point is not inclusive) coordinates of `tile` into the `start_end` array. It is assumed that a space of `2*tile->ndim` has been already allocated (static or dynamic) for `start_end` before this function is called.

`rel_block` (or relative-to-block) is only relevant when `tile` has an intermediate tile between it and the allocated space (like a channel, see `gal_tile_full_two_layers`). If it doesn't (`tile->block` points the allocated dataset), then the value to `rel_block` is irrelevant.

When `tile->block` is its self a larger block and `rel_block` is set to 0, then the starting and ending positions will be based on the position within `tile->block`, not the allocated space.

`void *` [Function]
`gal_tile_start_end_ind_inclusive` (*gal_data_t* `*tile`, *gal_data_t* `*work`, *size_t* `*start_end_inc`)
Put the indexs of the first/start and last/end pixels (inclusive) in a tile into the `start_end` array (that must have two elements). NOTE: this function stores the index of each point, not its coordinates. It will then return the pointer to the start of the tile in the `work` data structure (which doesn't have to be equal to `tile->block`.

The outputs of this function are defined to make it easy to parse over an n-dimensional tile. For example, this function is one of the most important parts of the internal processing of in `GAL_TILE_PARSE_OPERATE` function-like macro that is described below.

`gal_data_t *` [Function]
`gal_tile_series_from_minmax` (*gal_data_t* `*block`, *size_t* `*minmax`, *size_t* `number`)
Construct a list of tile(s) given coordinates of the minimum and maximum of each tile. The minimum and maximums are assumed to be inclusive and in C order (slowest dimension first). The returned pointer is an allocated `gal_data_t` array that can later be freed with `gal_data_array_free` (see Section 11.3.6.3 [Arrays of datasets], page 352). Internally, each element of the output array points to the next element, so the output may also be treated as a list of datasets (see Section 11.3.8.9 [List of `gal_data_t`], page 368) and passed onto the other functions described in this section.

The array keeping the minimum and maximum coordinates for each tile must have the following format. So in total `minmax` must have `2*ndim*number` elements.

```
| min0_d0 | min0_d1 | max0_d0 | max0_d1 | ...
                  ... | minN_d0 | minN_d1 | maxN_d0 | maxN_d1 |
```

`gal_data_t *` [Function]
`gal_tile_block` (*gal_data_t* `*tile`)

Return the dataset that contains `tile`'s allocated block of memory. If tile is immediately defined as part of the allocated block, then this is equivalent to `tile->block`. However, it is possible to have multiple layers of tiles (where `tile->block` is itself a tile). So this function is the most generic way to get to the actual allocated dataset.

`size_t` [Function]
`gal_tile_block_increment` (*gal_data_t* `*block`, *size_t* `*tsize`, *size_t* `num_increment`, *size_t* `*coord`)

Return the increment necessary to start at the next contiguous patch memory associated with a tile. `block` is the allocated block of memory and `tsize` is the size of the tile along every dimension. If `coord` is `NULL`, it is ignored. Otherwise, it will contain the coordinate of the start of the next contiguous patch of memory.

This function is intended to be used in a loop and `num_increment` is the main variable to this function. For the first time you call this function, it should be `1`. In subsequent calls (while you are parsing a tile), it should be increased by one.

`gal_data_t *` [Function]
`gal_tile_block_write_const_value` (*gal_data_t* `*tilevalues`, *gal_data_t* `*tilesll`, *int* `withblank`, *int* `initialize`)

Write a constant value for each tile over the area it covers in an allocated dataset that is the size of `tile`'s allocated block of memory (found through `gal_tile_block` described above). The arguments to this function are:

`tilevalues`
: This must be an array that has the same number of elements as the nodes in in `tilesll` and in the same order that 'tilesll' elements are parsed (from top to bottom, see Section 11.3.8 [Linked lists (`list.h`)], page 357). As a result the array's number of dimensions is irrelevant, it will be parsed contiguously.

`tilesll`
: The list of input tiles (see Section 11.3.8.9 [List of `gal_data_t`], page 368). Internally, it might be stored as an array (for example the output of `gal_tile_series_from_minmax` described above), but this function doesn't care, it will parse the `next` elements to go to the next tile. This function will not pop-from or free the `tilesll`, it will only parse it from start to end.

`withblank`
: If the block containing the tiles has blank elements, those blank elements will be blank in the output of this function also, hence the array will be initialized with blank values when this option is called (see below).

`initialize`
: Initialize the allocated space with blank values before writing in the constant values. This can be useful when the tiles don't cover the full allocated block.

`gal_data_t *` [Function]
`gal_tile_block_check_tiles` (*gal_data_t* `*tilesll`)

Make a copy of the memory block and fill it with the index of each tile in `tilesll` (counting from 0). The non-filled areas will have blank values. The output dataset will have a type of `GAL_TYPE_INT32` (see Section 11.3.3 [Library data types (`type.h`)], page 337).

This function can be used when you want to check the coverage of each tile over the allocated block of memory. It is just a wrapper over the `gal_tile_block_write_const_value` (with `withblank` set to zero).

`void *` [Function]
`gal_tile_block_relative_to_other` (*gal_data_t* `*tile`, *gal_data_t* `*other`)

Return the pointer corresponding to the start of the region covered by `tile` over the `other` dataset. See the examples in `GAL_TILE_PARSE_OPERATE` for some example applications of this function.

`void` [Function]
`gal_tile_block_blank_flag` (*gal_data_t* `*tilell`, *size_t* `numthreads`)

Check if each tile in the list has blank values and update its `flag` to mark this check and its result (see Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346). The operation will be done on `numthreads` threads.

`GAL_TILE_PARSE_OPERATE` (`IN`, `OTHER`, `PARSE_OTHER`, `CHECK_BLANK`, `OP`) [Function-like macro]

Parse `IN` (which can be a tile or a fully allocated block of memory) and do the `OP` operation on it. `OP` can be any combination of C expressions. If `OTHER!=NULL`, `OTHER` will be interpreted as a dataset and this macro will allow access to its element(s) and it can optionally be parsed while parsing over `IN`.

If `OTHER` is a fully allocated block of memory (not a tile), then the same region that is covered by `IN` within its own block will be parsed (the same starting pixel with the same number of pixels in each dimension). Hence, in this case, the blocks of `OTHER` and `IN` must have the same size. When `OTHER` is a tile it must have the same size as `IN` and parsing will start from its starting element/pixel. Also, the respective allocated blocks of `OTHER` and `IN` (if different) may have different sizes. Using `OTHER` (along with `PARSE_OTHER`), this function-like macro will thus enable you to parse and define your own operation on two fixed size regions in one or two blocks of memory. In the latter case, they may have different numeric data types, see Section 4.5 [Numeric data types], page 115).

The input arguments to this macro are explained below, the expected type of each argument are also written following the argument name:

`IN (gal_data_t)`
: Input dataset, this can be a tile or an allocated block of memory.

`OTHER (gal_data_t)`
: Dataset (`gal_data_t`) to parse along with `IN`. It can be `NULL`. In that case, `o` (see description of `OP` below) will be `NULL` and should not be used. If `PARSE_OTHER` is zero, only its first element will be used and the size of this dataset is irrelevant.

When `OTHER` is a block of memory, it has to have the same size as the allocated block of `IN`. When its a tile, it has to have the same size as `IN`.

`PARSE_OTHER (int)`
: Parse the other dataset along with the input. When this is non-zero and `OTHER!=NULL`, then the `o` pointer will be incremented to cover the `OTHER` tile at the same rate as `i`, see description of `OP` for `i` and `o`.

`CHECK_BLANK (int)`
: If it is non-zero, then the input will be checked for blank values and `OP` will only be called when we are not on a blank element.

`OP`
: Operator: this can be any number of C expressions. This macro is going to define a `itype *i` variable which will increment over each element of the input array/tile. `itype` will be replaced with the C type that corresponds to the type of `INPUT`. As an example, if `INPUT`'s type is `GAL_DATA_UINT16` or `GAL_DATA_FLOAT32`, `i` will be defined as `uint16` or `float` respectively.

  This function-like macro will also define an `otype *o` which you can use to access an element of the `OTHER` dataset (if `OTHER!=NULL`). `o` will correspond to the type of `OTHER` (similar to `itype` and `INPUT` discussed above). If `PARSE_OTHER` is non-zero, then `o` will also be incremented to the same index element but in the other array. You can use these along with any other variable you define before this macro to process the input and/or the other.

  All variables within this function-like macro begin with `tpo_` except for the three variables listed below. Therefore, as long as you don't start the names of your variables with this prefix everything will be fine. Note that `i` (and possibly `o`) will be incremented once by this function-like macro, so don't increment them within `OP`.

  `i`
  : Pointer to the element of `INPUT` that is being parsed with the proper type.

  `o`
  : Pointer to the element of `OTHER` that is being parsed with the proper type. `o` can only be used if `OTHER!=NULL` and it will be parsed/incremented if `PARSE_OTHER` is non-zero.

  `b`
  : Blank value in the type of `INPUT`.

You can use a given tile (`tile` on a dataset that it was not initialized with but has the same size, let's call it `new`) with the following steps:

```
void *tarray;
gal_data_t *tblock;

/* 'tile->block' must be corrected AFTER 'tile->array'. */
tarray      = tile->array;
tblock      = tile->block;
tile->array = gal_tile_block_relative_to_other(tile, new);
tile->block = new;
```

```
/* Parse and operate over this region of the 'new' dataset. */
GAL_TILE_PARSE_OPERATE(tile, NULL, 0, 0, {
    YOUR_PROCESSING;
  });

/* Reset 'tile->block' and 'tile->array'. */
tile->array=tarray;
tile->block=tblock;
```

You can work on the same region of another block in one run of this function-like macro. To do that, you can make a fake tile and pass that as the `OTHER` argument. Below is a demonstration, `tile` is the actual tile that you start with and `new` is the other block of allocated memory.

```
size_t zero=0;
gal_data_t *faketile;

/* Allocate the fake tile, these can be done outside a loop
 * (over many tiles). */
faketile=gal_data_alloc(NULL, new->type, 1, &zero,
                        NULL, 0, -1, NULL, NULL, NULL);
free(faketile->array);               /* To keep things clean. */
free(faketile->dsize);               /* To keep things clean. */
faketile->block = new;
faketile->ndim  = new->ndim;

/* These can be done in a loop (over many tiles). */
faketile->size  = tile->size;
faketile->dsize = tile->dsize;
faketile->array = gal_tile_block_relative_to_other(tile, new);

/* Do your processing.... in a loop (over many tiles). */
GAL_TILE_PARSE_OPERATE(tile, faketile, 1, 1, {
    YOUR_PROCESSING_EXPRESSIONS;
  });

/* Clean up (outside the loop). */
faketile->array=NULL;
faketile->dsize=NULL;
gal_data_free(faketile);
```

11.3.15.2 Tile grid

One very useful application of tiles is to completely cover an input dataset with tiles. Such that you know every pixel/data-element of the input image is covered by only one tile. The constructs in this section allow easy definition of such a tile structure. They will create lists of tiles that are also usable by the general tools discussed in Section 11.3.15.1 [Independent tiles], page 400.

As discussed in Section 4.7 [Tessellation], page 123, (mainly raw) astronomical images will mostly require two layers of tessellation, one for amplifier channels which all have the same size and another (smaller tile-size) tessellation over each channel. Hence, in this section we define a general structure to keep the main parameters of this two-layer tessellation and help in benefiting from it.

`gal_tile_two_layer_params` [Type (C `struct`)]

The general structure to keep all the necessary parameters for a two-layer tessellation.

```
struct gal_tile_two_layer_params
{
  /* Inputs */
  size_t             *tilesize;  /*******************************/
  size_t          *numchannels;  /* These parameters have to be */
  float          remainderfrac;  /* filled manually before      */
  uint8_t           workoverch;  /* calling the functions in    */
  uint8_t           checktiles;  /* this section.               */
  uint8_t       oneelempertile;  /*******************************/

  /* Internal parameters. */
  size_t                  ndim;
  size_t              tottiles;
  size_t          tottilesinch;
  size_t           totchannels;
  size_t          *channelsize;
  size_t             *numtiles;
  size_t         *numtilesinch;
  char          *tilecheckname;
  size_t          *permutation;
  size_t           *firsttsize;

  /* Tile and channel arrays (which are also lists). */
  gal_data_t            *tiles;
  gal_data_t         *channels;
};
```

`size_t *` [Function]
`gal_tile_full` (*gal_data_t* `*input`, *size_t* `*regular`, *float* `remainderfrac`, *gal_data_t* `**out`, *size_t* `multiple`, *size_t* `**firsttsize`)

Cover the full dataset with (mostly) identical tiles and return the number of tiles created along each dimension. The regular tile size (along each dimension) is determined from the `regular` array. If `input`'s size is not an exact multiple of `regular` for each dimension, then the tiles touching the edges in that dimension will have a different size to fully cover every element of the input (depending on `remainderfrac`).

The output is an array with the same dimensions as `input` which contains the number of tiles along each dimension. See Section 4.7 [Tessellation], page 123, for a description of its application in Gnuastro's programs and `remainderfrac`, just note that this function defines only one layer of tiles.

This is a low-level function (independent of the `gal_tile_two_layer_params` structure defined above). If you want a two-layer tessellation, directly call `gal_tile_full_two_layers` that is described below. The input arguments to this function are:

`input` The main dataset (allocated block) which you want to create a tessellation over (only used for its sizes). So `input` may be a tile also.

`regular` The the size of the regular tiles along each of the input's dimensions. So it must have the same number of elements as the dimensions of `input` (or `input->ndim`).

`remainderfrac`
The significant fraction of the remainder space to see if it should be split into two and put on both sides of a dimension or not. This is thus only relevant `input` length along a dimension isn't an exact multiple of the regular tile size along that dimension. See Section 4.7 [Tessellation], page 123, for a more thorough discussion.

`out` Pointer to the array of data structures that will keep all the tiles (see Section 11.3.6.3 [Arrays of datasets], page 352). If `*out==NULL`, then the necessary space to keep all the tiles will be allocated. If not, then all the tile information will be filled from the dataset that `*out` points to, see `multiple` for more.

`multiple` When `*out==NULL` (and thus will be allocated by this function), allocate space for `multiple` times the number of tiles needed. This can be very useful when you have several more identically sized `inputs`, and you want all their tiles to be allocated (and thus indexed) together, even though they have different `block` datasets (that then link to one allocated space). See the definition of channels in Section 4.7 [Tessellation], page 123, and `gal_tile_full_two_layers` below.

`firsttsize`
The size of the first tile along every dimension. This is only different from the regular tile size when `regular` is not an exact multiple of `input`'s length along every dimension. This array is allocated internally by this function.

`void` [Function]
`gal_tile_full_sanity_check` (*char* `*filename`, *char* `*hdu`, *gal_data_t* `*input`, *struct gal_tile_two_layer_params* `*tl`)
Make sure that the input parameters (in `tl`, short for two-layer) correspond to the input dataset. `filename` and `hdu` are only required for error messages. Also, allocate and fill the `tl->channelsize` array.

`void` [Function]
`gal_tile_full_two_layers` (*gal_data_t* `*input`, *struct gal_tile_two_layer_params* `*tl`)
Create the two layered tessellation in `tl`. The general set of steps you need to take to define the two-layered tessellation over an image can be seen in the example code below.

```
gal_data_t *input;
struct gal_tile_two_layer_params tl;
char *filename="input.fits", *hdu="1";

/* Set all the inputs shown in the structure definition. */
...

/* Read the input dataset. */
input=gal_fits_img_read(filename, hdu, -1);

/* Do a sanity check and preparations. */
gal_tile_full_sanity_check(filename, hdu, input, &tl);

/* Build the two-layer tessellation*/
gal_tile_full_two_layers(input, &tl);

/* 'tl.tiles' and 'tl.channels' are now a lists of tiles.*/
```

`void` [Function]
`gal_tile_full_permutation` (*struct gal_tile_two_layer_params* `*tl`)

Make a permutation to allow the conversion of tile location in memory to its location in the full input dataset and put it in `tl->permutation`. If a permutation has already been defined for the tessellation, this function will not do anything. If permutation won't be necessary (there is only one channel or one dimension), then this function will not do anything (`tl->permutation` must have been initialized to `NULL`).

When there is only one channel OR one dimension, the tiles are allocated in memory in the same order that they represent the input data. However, to make channel-independent processing possible in a generic way, the tiles of each channel are allocated contiguously. So, when there is more than one channel AND more than one dimension, the index of the tile does not correspond to its position in the grid covering the input dataset.

The example below may help clarify: assume you have a 6x6 tessellation with two channels in the horizontal and one in the vertical. On the left you can see how the tile IDs correspond to the input dataset. NOTE how '03' is on the second row, not on the first after '02'. On the right, you can see how the tiles are stored in memory (and shown if you simply write the array into a FITS file for example).

```
Corresponding to input              In memory
----------------------             --------------
  15 16 17 33 34 35                30 31 32 33 34 35
  12 13 14 30 31 32                24 25 26 27 28 29
  09 10 11 27 28 29                18 19 20 21 22 23
  06 07 08 24 25 26      <--       12 13 14 15 16 17
  03 04 05 21 22 23                06 07 08 09 10 11
  00 01 02 18 19 20                00 01 02 03 04 05
```

As a result, if your values are stored in same order as the tiles, and you want them in over-all memory (for example to save as a FITS file), you need to permute the values:

```
gal_permutation_apply(values, tl->permutation);
```

If you have values over-all and you want them in tile-order, you can apply the inverse permutation:

```
gal_permutation_apply_inverse(values, tl->permutation);
```

Recall that this is the definition of permutation in this context:

```
permute:    IN_ALL[ i       ]   =   IN_MEMORY[ perm[i] ]
inverse:    IN_ALL[ perm[i] ]   =   IN_MEMORY[ i       ]
```

`void` [Function]
`gal_tile_full_values_write` (*gal_data_t* `*tilevalues`, *struct gal_tile_two_layer_params* `*tl`, *int* `withblank`, *char* `*filename`, *gal_fits_list_key_t* `*keys`, *char* `*program_string`)

Write one value for each tile into a file. It is important to note that the values in `tilevalues` must be ordered in the same manner as the tiles, so `tilevalues->array[i]` is the value that should be given to `tl->tiles[i]`. The `tl->permutation` array must have been initialized before calling this function with `gal_tile_full_permutation`.

If `withblank` is non-zero, then block structure of the tiles will be checked and all blank pixels in the block will be blank in the final output file also.

`gal_data_t *` [Function]
`gal_tile_full_values_smooth` (*gal_data_t* `*tilevalues`, *struct gal_tile_two_layer_params* `*tl`, *size_t* `width`, *size_t* `numthreads`)

Smooth the given values with a flat kernel of the given `width`. This cannot be done manually because if `tl->workoverch==0`, tiles in different channels must not be mixed/smoothed. Also the tiles are contiguous within the channel, not within the image, see the description under `gal_tile_full_permutation`.

`size_t` [Function]
`gal_tile_full_id_from_coord` (*struct gal_tile_two_layer_params* `*tl`, *size_t* `*coord`)

Return the ID of the tile that corresponds to the coordinates `coord`. Having this ID, you can use the `tl->tiles` array to get to the proper tile or read/write a value into an array that has one value per tile.

`void` [Function]
`gal_tile_full_free_contents` (*struct gal_tile_two_layer_params* `*tl`)

Free all the allocated arrays within `tl`.

11.3.16 Bounding box (`box.h`)

Functions related to reporting a the bounding box of certain inputs are declared in `gnuastro/box.h`. All coordinates in this header are in the FITS format (first axis is the horizontal and the second axis is vertical).

`void` [Function]
`gal_box_bound_ellipse_extent` (*double* `a`, *double* `b`, *double* `theta_deg`, *double* `*extent`)

Return the maximum extent along each dimension of the given ellipse from the center of the ellipse. Therefore this is half the extent of the box in each dimension. `a` is the ellipse semi-major axis, `b` is the semi-minor axis, `theta_deg` is the position angle in degrees. The extent in each dimension is in floating point format and stored in `extent` which must already be allocated before this function.

`void` [Function]
`gal_box_bound_ellipse` (*double* `a`, *double* `b`, *double* `theta_deg`, *long* `*width`)

Any ellipse can be enclosed into a rectangular box. This function will write the height and width of that box where `width` points to. It assumes the center of the ellipse is located within the central pixel of the box. `a` is the ellipse semi-major axis length, `b` is the semi-minor axis, `theta_deg` is the position angle in degrees. The `width` array will contain the output size in long integer type. `width[0]`, and `width[1]` are the number of pixels along the first and second FITS axis. Since the ellipse center is assumed to be in the center of the box, all the values in `width` will be an odd integer.

`void` [Function]
`gal_box_border_from_center` (*double* `center`, *size_t* `ndim`, *long* `*width`, *long* `*fpixel`, *long* `*lpixel`)

Given the center coordinates in `center` and the `width` (along each dimension) of a box, return the coordinates of the first (`fpixel`) and last (`lpixel`) pixels. All arrays must have `ndim` elements (one for each dimension).

`int` [Function]
`gal_box_overlap` (*long* `*naxes`, *long* `*fpixel_i`, *long* `*lpixel_i`, *long* `*fpixel_o`, *long* `*lpixel_o`, *size_t* `ndim`)

An `ndim`-dimensional dataset of size `naxes` (along each dimension, in FITS order) and a box with first and last (inclusive) coordinate of `fpixel_i` and `lpixel_i` is given. This box doesn't necessarily have to lie within the dataset, it can be outside of it, or only partially overlap. This function will change the values of `fpixel_i` and `lpixel_i` to exactly cover the overlap in the input dataset's coordinates.

This function will return 1 if there is an overlap and 0 if there isn't. When there is an overlap, the coordinates of the first and last pixels of the overlap will be put in `fpixel_o` and `lpixel_o`.

11.3.17 Polygons (`polygon.h`)

Polygons are commonly necessary in image processing. In Gnuastro, they are used in Crop (see Section 6.1 [Crop], page 151) for cutting out non-rectangular regions of a image. Warp (see Section 6.4 [Warp], page 199) uses them to warp the images into a new pixel grid. The polygon related Gnuastro library macros and functions are introduced here.

In all the functions here the vertices (and points) are defined as an array. So a polygon with 4 vertices will be identified with an array of 8 elements with the first two elements keeping the 2D coordinates of the first vertice and so on.

`GAL_POLYGON_MAX_CORNERS` [Macro]

The largest number of vertices a polygon can have in this library.

`GAL_POLYGON_ROUND_ERR` [Macro]

We have to consider floating point round-off errors when dealing with polygons. For example we will take `A` as the maximum of `A` and `B` when `A>B-GAL_POLYGON_ROUND_ERR`.

`void` [Function]
`gal_polygon_ordered_corners` (*double* `*in`, *size_t* `n`, *size_t* `*ordinds`)

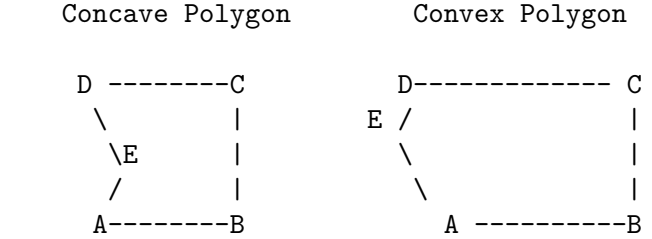
We have a simple polygon (that can result from projection, so its edges don't collide or it doesn't have holes) and we want to order its corners in an anticlockwise fashion. This is necessary for clipping it and finding its area later. The input vertices can have practically any order.

The input (`in`) is an array containing the coordinates (two values) of each vertice. `n` is the number of corners. So `in` should have `2*n` elements. The output (`ordinds`) is an array with `n` elements specifying the indexs in order. This array must have been allocated before calling this function. The indexes are output for more generic usage, for example in a homographic transform (necessary in warping an image, see Section 6.4.1 [Warping basics], page 200), the necessary order of vertices is the same for all the pixels. In other words, only the positions of the vertices change, not the way they need to be ordered. Therefore, this function would only be necessary once.

As a summary, the input is unchanged, only `n` values will be put in the `ordinds` array. Such that calling the input coordinates in the following fashion will give an anti-clockwise order when there are 4 vertices:

```
1st vertice: in[ordinds[0]*2], in[ordinds[0]*2+1]
2nd vertice: in[ordinds[1]*2], in[ordinds[1]*2+1]
3rd vertice: in[ordinds[2]*2], in[ordinds[2]*2+1]
4th vertice: in[ordinds[3]*2], in[ordinds[3]*2+1]
```

The implementation of this is very similar to the Graham scan in finding the Convex Hull. However, in projection we will never have a concave polygon (the left condition below, where this algorithm will get to E before D), we will always have a convex polygon (right case) or E won't exist!



This is because we are always going to be calculating the area of the overlap between a quadrilateral and the pixel grid or the quadrilateral its self.

The `GAL_POLYGON_MAX_CORNERS` macro is defined so there will be no need to allocate these temporary arrays separately. Since we are dealing with pixels, the polygon can't really have too many vertices.

`double` [Function]
`gal_polygon_area` (*double* `*v`, *size_t* `n`)
Find the area of a polygon with vertices defined in `v`. `v` points to an array of doubles which keep the positions of the vertices such that `v[0]` and `v[1]` are the positions of the first vertice to be considered.

`int` [Function]
`gal_polygon_pin` (*double* `*v`, *double* `*p`, *size_t* `n`)
Return `1` if the point `p` is within the polygon whose vertices are defined by `v` and `0` otherwise. Note that the vertices of the polygon have to be sorted in an anti-clock-wise manner.

`int` [Function]
`gal_polygon_ppropin` (*double* `*v`, *double* `*p`, *size_t* `n`)
Similar to `gal_polygon_pin`, except that if the point `p` is on one of the edges of a polygon, this will return `0`.

`void` [Function]
`gal_polygon_clip` (*double* `*s`, *size_t* `n`, *double* `*c`, *size_t* `m`, *double* `*o`, *size_t* `*numcrn`)
Clip (find the overlap of) two polygons. This function uses the Sutherland-Hodgman (`https://en.wikipedia.org/wiki/Sutherland%E2%80%93Hodgman_algorithm`) polygon clipping algorithm. Note that the vertices of both polygons have to be sorted in an anti-clock-wise manner.

11.3.18 Qsort functions (`qsort.h`)

When sorting a dataset is necessary, the C programming language provides the `qsort` (Quick sort) function. `qsort` is a generic function which allows you to sort any kind of data structure (not just a single array of numbers). To define "greater" and "smaller" (for sorting), `qsort` needs another function, even for simple numerical types. The functions introduced in this section are to passed onto `qsort`.

The first class of functions below (with `TYPE` in their names) can be used for sorting a simple numeric array. Just replace `TYPE` with the dataset's numeric datatype. The second set of functions can be used to sort indexs (leave the actual numbers untouched). To use the second set of functions, a global variable or structure are also necessary as described below.

`gal_qsort_index_single` [Global variable]
Pointer to an array (for example `float *` or `int *`) to use as a reference in `gal_qsort_index_single_TYPE_d` or `gal_qsort_index_single_TYPE_i`, see the explanation of these functions for more. Note that if *more than one* array is to be sorted in a multi-threaded operation, these functions will not work as expected. However, when all the threads just sort the indexs based on a *single array*, this global variable can safely be used in a multi-threaded scenario.

`gal_qsort_index_multi` [Type (C `struct`)]
Structure to get the sorted indexs of multiple datasets on multiple threads with `gal_qsort_index_multi_d` or `gal_qsort_index_multi_i`. Note that the `values` array

will not be changed by these functions, it is only read. Therefore all the `values` elements in the (to be sorted) array of `gal_qsort_index_multi` must point to the same place.

```
struct gal_qsort_index_multi
{
  float *values;         /* Array of values (same in all).      */
  size_t  index;         /* Index of each element to be sorted. */
};
```

`int` [Function]
`gal_qsort_TYPE_d` (*const void* `*a`, *const void* `*b`)

When passed to `qsort`, this function will sort a `TYPE` array in decreasing order (first element will be the largest). Please replace `TYPE` (in the function name) with one of the Section 4.5 [Numeric data types], page 115, for example `gal_qsort_int32_d`, or `gal_qsort_float64_d`.

`int` [Function]
`gal_qsort_TYPE_i` (*const void* `*a`, *const void* `*b`)

When passed to `qsort`, this function will sort a `TYPE` array in increasing order (first element will be the smallest). Please replace `TYPE` (in the function name) with one of the Section 4.5 [Numeric data types], page 115, for example `gal_qsort_int32_i`, or `gal_qsort_float64_i`.

`int` [Function]
`gal_qsort_index_single_TYPE_d` (*const void* `*a`, *const void* `*b`)

When passed to `qsort`, this function will sort a `size_t` array based on decreasing values in the `gal_qsort_index_single`. The global `gal_qsort_index_single` pointer has a `void *` pointer which will be cast to the proper type based on this function: for example `gal_qsort_index_single_uint16_d` will cast the array to an unsigned 16-bit integer type. The array that `gal_qsort_index_single` points to will not be changed, it is only read. For example, see this demo program:

```
#include <stdio.h>
#include <stdlib.h>           /* qsort is defined in stdlib.h. */
#include <gnuastro/qsort.h>

int
main (void)
{
  size_t s[4]={0, 1, 2, 3};
  float f[4]={1.3,0.2,1.8,0.1};
  gal_qsort_index_single=f;
  qsort(s, 4, sizeof(size_t), gal_qsort_index_single_float_d);
  printf("%zu, %zu, %zu, %zu\n", s[0], s[1], s[2], s[3]);
  return EXIT_SUCCESS;
}
```

The output will be: `2, 0, 1, 3`.

`int` [Function]
`gal_qsort_index_single_TYPE_i` (*const void* `*a`, *const void* `*b`)
Similar to `gal_qsort_index_single_TYPE_d`, but will sort the indexes such that the values of `gal_qsort_index_single` can be parsed in increasing order.

`int` [Function]
`gal_qsort_index_multi_d` (*const void* `*a`, *const void* `*b`)
When passed to `qsort` with an array of `gal_qsort_index_multi`, this function will sort the array based on the values of the given indexs. The sorting will be ordered according to the `values` pointer of `gal_qsort_index_multi`. Note that `values` must point to the same place in all the structures of the `gal_qsort_index_multi` array.

This function is only useful when the the indexs of multiple arrays on multiple threads are to be sorted. If your program is single threaded, or all the indexs belong to a single array (sorting different sub-sets of indexs in a single array on multiple threads), it is recommended to use `gal_qsort_index_single_d`.

`int` [Function]
`gal_qsort_index_multi_i` (*const void* `*a`, *const void* `*b`)
Similar to `gal_qsort_index_multi_d`, but the result will be sorted in increasing order (first element will have the smallest value).

11.3.19 Permutations (`permutation.h`)

Permutation is the technical name for re-ordering of values. The need for permutations occurs a lot during (mainly low-level) processing. To do permutation, you must provide two inputs: an array of values (that you want to re-order in place) and a permutation array which contains the new index of each element (let's call it `perm`). The diagram below shows the input array before and after the re-ordering.

```
permute:    AFTER[ i       ] = BEFORE[ perm[i] ]     i = 0 .. N-1
inverse:    AFTER[ perm[i] ] = BEFORE[ i       ]     i = 0 .. N-1
```

The functions here are a re-implementation of the GNU Scientific Library's `gsl_permute` function. The reason we didn't use that function was that it uses system-specific types (like `long` and `int`) which can have different widths on different systems, hence are not easily convertible to Gnuastro's fixed width types (see Section 4.5 [Numeric data types], page 115). There is also a separate function for each type, heavily using macros to allow a `base` function to work on all the types. Thus it is hard to read/understand. Hence, Gnuastro contains a re-write of their steps in a new type-agnostic method which is a single function that can work on any type.

As described in GSL's source code and manual, this implementation comes from Donald Knuth's *Art of computer programming* book, in the "Sorting and Searching" chapter of Volume 3 (3rd ed). Exercise 10 of Section 5.2 defines the problem and in the answers, Knuth describes the solution. So if you are interested, please have a look there for more.

We are in contact with the GSL developers and in the future[20] we will submit these implementations to GSL. If they are finally incorporated there, we will delete this section in future versions.

`void` [Function]
`gal_permutation_check` (*size_t* `*permutation`, *size_t* `size`)
Print how `permutation` will re-order an array that has `size` elements for each element in one one line.

`void` [Function]
`gal_permutation_apply` (*gal_data_t* `*input`, *size_t* `*permutation`)
Apply `permutation` on the `input` dataset (can have any type), see above for the definition of permutation.

`void` [Function]
`gal_permutation_apply_inverse` (*gal_data_t* `*input`, *size_t* `*permutation`)
Apply the inverse of `permutation` on the `input` dataset (can have any type), see above for the definition of permutation.

11.3.20 Matching (`match.h`)

Matching is often necessary when the measurements have been done using different instruments, different software or different configurations of the same software. The functions in this part of Gnuastro's library will be growing to allow matching of images and finding a match between different catalogs (register them). Currently it only provides the The high-level measurements are stored in tables with positions (commonly in RA and Dec with units of degrees).

`gal_data_t *` [Function]
`gal_match_coordinates` (*gal_data_t* `*coord1`, *gal_data_t* `*coord2`, *double* `*aperture`, *int* `sorted_by_first`, *int* `inplace`, *size_t* `minmapsize`, *size_t* `*nummatched`)
Return the permutations that when applied, the first `nummatched` rows of both inputs match with each other (are the nearest within the given aperture). The two inputs (`coord1` and `coord2`) must be Section 11.3.8.9 [List of `gal_data_t`], page 368. Each `gal_data_t` node in the list should be a single dimensional dataset (column in a table). The dimensions of the coordinates is determined by the number of `gal_data_t` nodes in the two input lists (which must be equal). Note that the number of rows (or the number of elements in each `gal_data_t`) in the columns of `coord1` and `coord2` can be different.

The matching aperture is defined by the `aperture` array. If several points of one catalog lie within this aperture of a point in the other, the nearest is defined as the match. In a 2D situation (where the input lists have two nodes), for the most generic case, it must have three elements: the major axis length, axis ratio and position angle (see Section 8.1.1.1 [Defining an ellipse], page 284). If `aperture[1]==1`, the aperture

[20] Gnuastro's Task 14497 (`http://savannah.gnu.org/task/?14497`). If this task is still "postponed" when you are reading this and you are interested to help, your help would be very welcome. Both Gnuastro and GSL developers are very busy, hence both would appreciate your help.

will be a circle of radius `aperture[0]` and the third value won't be used. When the aperture is an ellipse, distances between the points are also calculated in the respective elliptical distances (r_{el} in Section 8.1.1.1 [Defining an ellipse], page 284).

To speed up the search, this function will sort the input coordinates by their first column (first axis). If *both* are already sorted by their first column, you can avoid the sorting step by giving a non-zero value to `sorted_by_first`.

When sorting is necessary and `inplace` is non-zero, the actual input columns will be sorted. Otherwise, an internal copy of the inputs will be made, used (sorted) and later freed before returning. Therefore, when `inplace==0`, inputs will remain untouched, but this function will take more time and memory.

If internal allocation is necessary and the space is larger than `minmapsize`, the space will be not allocated in the RAM, but in a file, see description of `--minmapsize` in Section 4.1.2.2 [Processing options], page 98.

The number of matches will be put in the space pointed by `nummatched`. If there wasn't any match, this function will return `NULL`. If match(s) were found, a list with three `gal_data_t` nodes will be returned. The top two nodes in the list are the permutations that must be applied to the first and second inputs respectively. After applying the permutations, the top `nummatched` elements will match with each other. The third node is the distances between the respective match. Note that the three nodes of the list are all one-dimensional (a column) and can have different lengths.

Output permutations ignore internal sorting: the output permutations will correspond to the initial inputs. Therefore, even when `inplace!=0` (and this function re-arranges the inputs), the output permutation will correspond to original (possibly non-sorted) inputs.

The reason for this is that you rarely want the actual positional columns after the match. Usually, you also have other columns (measurements, for example magnitudes) for higher-level processing after the match (that correspond to the input order before sorting). Once you have the permutations, they can be applied to those other columns (see Section 11.3.19 [Permutations (`permutation.h`)], page 414) and the higher-level processing can continue.

When you read the coordinates from a table using `gal_table_read` (see Section 11.3.10 [Table input output (`table.h`)], page 370), and only ask for the coordinate columns, the inputs to this function are the returned `gal_data_t *` from two different tables.

11.3.21 Statistical operations (`statistics.h`)

After reading a dataset into memory from a file or fully simulating it with another process, the most common processes that will be done on it are statistical operations to let you quantify different aspects of the data. the functions in this section describe Gnuastro's current set of tools for this job. All these functions can work on any numeric data type natively (see Section 4.5 [Numeric data types], page 115) and can also work on tiles over a dataset. Hence the inputs and outputs are in Gnuastro's Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346.

`GAL_STATISTICS_SIG_CLIP_MAX_CONVERGE` [Macro]
: The maximum number of clips, when σ-clipping should be done by convergence. If the clipping does not converge before making this many clips, all sigma-clipping outputs will be NaN.

`GAL_STATISTICS_MODE_GOOD_SYM` [Macro]
: The minimum acceptable symmetricity of the mode calculation. If the symmetricity of the derived mode is less than this value, all the returned values by `gal_statistics_mode` will have a value of NaN.

`GAL_STATISTICS_BINS_INVALID` [Macro]
`GAL_STATISTICS_BINS_REGULAR` [Macro]
`GAL_STATISTICS_BINS_IRREGULAR` [Macro]
: Macros used to identify if the regularity of the bins when defining bins.

`gal_data_t *` [Function]
`gal_statistics_number` (*gal_data_t* `*input`)
: Return a single-element dataset with type `size_t` which contains the number of non-blank elements in `input`.

`gal_data_t *` [Function]
`gal_statistics_minimum` (*gal_data_t* `*input`)
: Return a single-element dataset containing the minimum non-blank value in `input`. The numerical datatype of the output is the same as `input`.

`gal_data_t *` [Function]
`gal_statistics_maximum` (*gal_data_t* `*input`)
: Return a single-element dataset containing the maximum non-blank value in `input`. The numerical datatype of the output is the same as `input`.

`gal_data_t *` [Function]
`gal_statistics_sum` (*gal_data_t* `*input`)
: Return a single-element (`double` or `float64`) dataset containing the sum of the non-blank values in `input`.

`gal_data_t *` [Function]
`gal_statistics_mean` (*gal_data_t* `*input`)
: Return a single-element (`double` or `float64`) dataset containing the mean of the non-blank values in `input`.

`gal_data_t *` [Function]
`gal_statistics_std` (*gal_data_t* `*input`)
: Return a single-element (`double` or `float64`) dataset containing the standard deviation of the non-blank values in `input`.

`gal_data_t *` [Function]
`gal_statistics_mean_std` (*gal_data_t* `*input`)
: Return a two-element (`double` or `float64`) dataset containing the mean and standard deviation of the non-blank values in `input`. The first element of the returned dataset is the mean and the second is the standard deviation.

This function will calculate both values in one pass over the dataset. Hence when both the mean and standard deviation of a dataset are necessary, this function is much more efficient than calling `gal_statistics_mean` and `gal_statistics_std` separately.

`gal_data_t *` [Function]
`gal_statistics_median` (*gal_data_t* `*input`, *int* `inplace`)

Return a single-element dataset containing the median of the non-blank values in `input`. The numerical datatype of the output is the same as `input`.

Calculating the median involves sorting the dataset and removing blank values, for better performance (and less memory usage), you can give a non-zero value to the `inplace` argument. In this case, the sorting and removal of blank elements will be done directly on the input dataset. However, after this function the original dataset may have changed (if it wasn't sorted or had blank values).

`size_t` [Function]
`gal_statistics_quantile_index` (*size_t* `size`, *double* `quantile`)

Return the index of the element that has a quantile of `quantile` assuming the dataset has `size` elements.

`size_t` [Function]
`gal_statistics_quantile` (*gal_data_t* `*input`, *double* `quantile`, *int* `inplace`)

Return a single-element dataset containing the value with in a quantile `quantile` of the non-blank values in `input`. The numerical datatype of the output is the same as `input`. See `gal_statistics_median` for a description of `inplace`.

`size_t` [Function]
`gal_statistics_quantile_function_index` (*gal_data_t* `*input`, *gal_data_t* `*value`, *int* `inplace`)

Return the index of the quantile function (inverse quantile) of `input` at `value`. In other words, this function will return the index of the nearest element (of a sorted and non-blank) `input` to `value`. If the value is outside the range of the input, then this function will return `GAL_BLANK_SIZE_T`.

`gal_data_t *` [Function]
`gal_statistics_quantile_function` (*gal_data_t* `*input`, *gal_data_t* `*value`, *int* `inplace`)

Return a single-element (`double` or `float64`) dataset containing the quantile function of the non-blank values in `input` at `value`. In other words, this function will return the quantile of `value` in `input`. If the value is smaller than the input's smallest element, the returned value will be zero. If the value is larger than the input's largest element, then the returned value is 1. See `gal_statistics_median` for a description of `inplace`.

`gal_data_t *` [Function]
`gal_statistics_mode` (*gal_data_t* `*input`, *float* `mirrordist`, *int* `inplace`)

Return a four-element (`double` or `float64`) dataset that contains the mode of the `input` distribution. This function implements the non-parametric algorithm to find

the mode that is described in Appendix C of Akhlaghi and Ichikawa [2015] (`https://arxiv.org/abs/1505.01664`).

In short it compares the actual distribution and its "mirror distribution" to find the mode. In order to be efficient, you can determine how far the comparison goes away from the mirror through the `mirrordist` parameter (think of it as a multiple of sigma/error). See `gal_statistics_median` for a description of `inplace`.

The output array has the following elements (in the given order, note that counting in C starts from 0).

```
array[0]: mode
array[1]: mode quantile.
array[2]: symmetricity.
array[3]: value at the end of symmetricity.
```

`gal_data_t *` [Function]
`gal_statistics_mode_mirror_plots` (*gal_data_t* `*input`, *gal_data_t* `*value`, *size_t* `numbins`, *int* `inplace`, *double* `*mirror_val`)

Make a mirrored histogram and cumulative frequency plot (with `numbins`) with the mirror distribution of the `input` having a value in `value`. If all the input elements are blank, or the mirror value is outside the range of the input, this function will return a `NULL` pointer.

The output is a list of data structures (see Section 11.3.8.9 [List of `gal_data_t`], page 368): the first is the bins with one bin at the mirror point, the second is the histogram with a maximum of one and the third is the cumulative frequency plot (with a maximum of one).

`int` [Function]
`gal_statistics_is_sorted` (*gal_data_t* `*input`, *int* `updateflags`)

Return `0` if the input is not sorted, if it is sorted, this function will return `1` and `2` if it is increasing or decreasing, respectively. This function will abort with an error if `input` has zero elements and will return `1` (sorted, increasing) when there is only one element. This function will only look into the dataset if the `GAL_DATA_FLAG_SORT_CH` bit of `input->flag` is `0`, see Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346.

When the flags don't indicate a previous check *and* `updateflags` is non-zero, this function will set the flags appropriately to avoid having to re-check the dataset in future calls (this can be very useful when repeated checks are necessary). When `updateflags==0`, this function has no side-effects on the dataset: it will not toggle the flags.

If you want to re-check a dataset with the blank-value-check flag already set (for example if you have made changes to it), then explicitly set the `GAL_DATA_FLAG_SORT_CH` bit to zero before calling this function. When there are no other flags, you can simply set the flags to zero (with `input->flags=0`), otherwise you can use this expression:

```
input->flags &= ~GAL_DATA_FLAG_SORT_CH;
```

`void` [Function]
`gal_statistics_sort_increasing` (*gal_data_t* `*input`)
Sort the input dataset (in place) in an increasing order and toggle the sort-related bit flags accordingly.

`void` [Function]
`gal_statistics_sort_decreasing` (*gal_data_t* `*input`)
Sort the input dataset (in place) in a decreasing order and toggle the sort-related bit flags accordingly.

`gal_data_t *` [Function]
`gal_statistics_no_blank_sorted` (*gal_data_t* `*input`, *int* `inplace`)
Remove all the blanks and sort the input dataset. If `inplace` is non-zero this will happen on the input dataset (in the allocated space of the input dataset). However, if `inplace` is zero, this function will allocate a new copy of the dataset and work on that. Therefore if `inplace==0`, the input dataset will be modified.

This function uses the bit flags of the input, so if you have modified the dataset, set `input->flags=0` before calling this function. Also note that `inplace` is only for the dataset elements. Therefore even when `inplace==0`, if the input is already sorted *and* has no blank values, then the flags will be updated to show this.

If all the elements were blank, then the returned dataset's `size` will be zero. This is thus a good parameter to check after calling this function to see if there actually were any non-blank elements in the input or not and take the appropriate measure. This can help avoid strange bugs in later steps. The flags of a zero-sized returned dataset will indicate that it has no blanks and is sorted in an increasing order. Even if having blank values or being sorted is not defined on a zero-element dataset, it is up to the caller to choose what they will do with a zero-element dataset. The flags have to be set after this function any way.

`gal_data_t *` [Function]
`gal_statistics_regular_bins` (*gal_data_t* `*input`, *gal_data_t* `*inrange`, *size_t* `numbins`, *double* `onebinstart`)
Generate an array of regularly spaced elements as a 1D array (column) of type `double` (i.e., `float64`, it has to be double to account for small differences on the bin edges). The input arguments are described below

`input` The dataset you want to apply the bins to. This is only necessary if the range argument is not complete, see below. If `inrange` has all the necessary information, you can pass a `NULL` pointer for this.

`inrange` This dataset keeps the desired range along each dimension of the input data structure, it has to be in `float` (i.e., `float32`) type.

- If you want the full range of the dataset (in any dimensions, then just set `inrange` to `NULL` and the range will be specified from the minimum and maximum value of the dataset (`input` cannot be `NULL` in this case).
- If there is one element for each dimension in range, then it is viewed as a quantile (Q), and the range will be: 'Q to 1-Q'.

- If there are two elements for each dimension in range, then they are assumed to be your desired minimum and maximum values. When either of the two are NaN, the minimum and maximum will be calculated for it.

`numbins` The number of bins: must be larger than 0.

`onebinstart`
A desired value for onebinstart. Note that with this option, the bins won't start and end exactly on the given range values, it will be slightly shifted to accommodate this request.

`gal_data_t *` [Function]
`gal_statistics_histogram` (*gal_data_t* `*input`, *gal_data_t* `*bins`, *int* `normalize`, *int* `maxone`)

Make a histogram of all the elements in the given dataset with bin values that are defined in the `inbins` structure (see `gal_statistics_regular_bins`, they currently have to be equally spaced). `inbins` is not mandatory, if you pass a `NULL` pointer, the bins structure will be built within this function based on the `numbins` input. As a result, when you have already defined the bins, `numbins` is not used.

Let's write the center of the ith element of the bin array as b_i, and the fixed half-bin width as h. Then element j of the input array (in_j) will be counted in b_i if $(b_i - h) \leq in_j < (b_i + h)$. However, if in_j is somewhere in the last bin, the condition changes to $(b_i - h) \leq in_j \leq (b_i + h)$.

`gal_data_t *` [Function]
`gal_statistics_cfp` (*gal_data_t* `*input`, *gal_data_t* `*bins`, *int* `normalize`)

Make a cumulative frequency plot (CFP) of all the elements in `input` with bin values that are defined in the `bins` structure (see `gal_statistics_regular_bins`).

The CFP is built from the histogram: in each bin, the value is the sum of all previous bins in the histogram. Thus, if you have already calculated the histogram before calling this function, you can pass it onto this function as the data structure in `bins->next` (see `List of gal_data_t`). If `bin->next!=NULL`, then it is assumed to be the histogram. If it is `NULL`, then the histogram will be calculated internally and freed after the job is finished.

When a histogram is given and it is normalized, the CFP will also be normalized (even if the normalized flag is not set here): note that a normalized CFP's maximum value is 1.

`gal_data_t *` [Function]
`gal_statistics_sigma_clip` (*gal_data_t* `*input`, *float* `multip`, *float* `param`, *int* `inplace`, *int* `quiet`)

Apply σ-clipping on a given dataset and return a dataset that contains the results. For a description of σ-clipping see Section 7.1.2 [Sigma clipping], page 209. `multip` is the multiple of the standard deviation (or σ, that is used to define outliers in each round of clipping).

The role of `param` is determined based on its value. If `param` is larger than `1` (one), it must be an integer and will be interpreted as the number clips to do. If it is less than `1` (one), it is interpreted as the tolerance level to stop the iteration.

The returned dataset (let's call it `out`) contains a four-element array with type `GAL_TYPE_FLOAT32`. The final number of clips is stored in the `out->status`.

```
float *array=out->array;
array[0]: Number of points used.
array[1]: Median.
array[2]: Mean.
array[3]: Standard deviation.
```

If the σ-clipping doesn't converge or all input elements are blank, then this function will return NaN values for all the elements above.

`gal_data_t *` [Function]
`gal_statistics_outlier_positive` (*gal_data_t* `*input`, *size_t* `window_size`, *float* `sigma`, *float* `sigclip_multip`, *float* `sigclip_param`, *int* `inplace`, *int* `quiet`)

Find the first positive outlier in the `input` distribution. The returned dataset contains a single element: the first positive outlier. It is one of the dataset's elements, in the same type as the input. If the process fails for any reason (for example no outlier was found), a `NULL` pointer will be returned.

All (possibly existing) blank elements are first removed from the input dataset, then it is sorted. A sliding window of `window_size` elements is parsed over the dataset. Starting from the `window_size`-th element of the dataset, in the direction of increasing values. This window is used as a reference. The first element where the distance to the previous (sorted) element is `sigma` units away from the distribution of distances in its window is considered an outlier and returned by this function.

Formally, if we assume there are N non-blank elements. They are first sorted. Searching for the outlier starts on element W. Let's take v_i to be the i-th element of the sorted input (with no blank values) and m and σ as the σ-clipped median and standard deviation from the distances of the previous W elements (not including v_i). If the value given to `sigma` is displayed with s, the i-th element is considered as an outlier when the condition below is true.

$$\frac{(v_i - v_{i-1}) - m}{\sigma} > s$$

The `sigclip_multip` and `sigclip_param` arguments specify the properties of the σ-clipping (see Section 7.1.2 [Sigma clipping], page 209, for more). You see that by this definition, the outlier cannot be any of the lower half elements. The advantage of this algorithm compared to σ-clippign is that it only looks backwards (in the sorted array) and parses it in one direction.

If `inplace!=0`, the removing of blank elements and sorting will be done within the input dataset's allocated space. Otherwise, this function will internally allocate (and later free) the necessary space to keep the intermediate space that this process requires.

If `quiet!=0`, this function will report the parameters every time it moves the window as a separate line with several columns. The first column is the value, the second (in square brackets) is the sorted index, the third is the distance of this element from

the previous one. The Fourth and fifth (in parenthesis) are the median and standard deviation of the sigma-clipped distribution within the window and the last column is the difference between the third and fourth, divided by the fifth.

`gal_data_t *` [Function]
`gal_statistics_outlier_flat_cfp` (*gal_data_t* `*input`, *size_t* `dist`, *float* `thresh`, *float* `width`, *int* `inplace`, *int* `quiet`, *size_t* `*index`)

Return the first element in the given dataset where the cumulative frequency plot first becomes flat (below a given threshold on the slope) for a sufficient width (percentage of whole dataset's range). The returned dataset only has one element (with the same type as the input). If `index!=NULL`, the index (counting from zero, after sorting dataset and removing any blanks) is written in the space that `index` points to. If no sufficiently flat portion is found, the returned pointer will be `NULL`.

The operation of this function can be best visualized using the cumulative frequency plot (CFP, see Section 7.1.1 [Histogram and Cumulative Frequency Plot], page 208). Imagine setting the horizontal axis of the input's CFP to a range of 0 to 100, and finding the first part where its slope is constantly/contiguously flat for a certain fraction/width of the whole dataset's range. Below we'll describe this in more detail.

This function will first remove all the blank elements and sort the remaining elements. If `inplace` is non-zero this step will be done in place: re-organizing/permuting the input dataset. Using `inplace` can result in faster processing and less RAM consumption, but only when the input dataset is no longer needed in its original permutation.

The input dataset will then be internally scaled from 0 to 100 (so `thresh` and `width` can be in units of percent). For each element, this function will then estimate the slope of the cumulative distribution function by using the `dist` elements before and after it. In other words, if d is the value of `dist`, then the slope over the i'th element (a_i) is estimated using this formula: $1/(a_{i+d} - a_{i-d})$. Therefore, when this slope is larger than 1, the distance between the two checked points is smaller than 1% of the dataset's range.

All points that have a slope less than `thresh` are then marked. The first (parsing from the smallest to the largest values) contiguous patch of marked elements that is larger than `width` wide (in units of percentage of the dataset's range) will be considered as the boundary region of outliers and the first element in that patch will be returned.

11.3.22 Binary datasets (`binary.h`)

Binary datasets only have two (usable) values: 0 (also known as background) or 1 (also known as foreground). They are created after some binary classification is applied to the dataset. The most common is thresholding: for example in an image, pixels with a value above the threshold are given a value of 1 and those with a value less than the threshold are assigned a value of 0.

Since there is only two values, in the processing of binary images, you are usually concerned with the positioning of an element and its vicinity (neighbors). When a dataset has more than one dimension, multiple classes of immediate neighbors (that are touching the element) can be defined for each data-element. To separate these different classes of immediate neighbors, we define *connectivity*.

The classification is done by the distance from element center to the neighbor's center. The nearest immediate neighbors have a connectivity of 1, the second nearest class of neighbors have a connectivity of 2 and so on. In total, the largest possible connectivity for data with `ndim` dimensions is `ndim`. For example in a 2D dataset, 4-connected neighbors (that share an edge and have a distance of 1 pixel) have a connectivity of 1. The other 4 neighbors that only share a vertice (with a distance of $\sqrt{2}$ pixels) have a connectivity of 2. Conventionally, the class of connectivity-2 neighbors also includes the connectivity 1 neighbors, so for example we call them 8-connected neighbors in 2D datasets.

Ideally, one bit is sufficient for each element of a binary dataset. However, CPUs are not designed to work on individual bits, the smallest unit of memory addresses is a byte (containing 8 bits on modern CPUs). Therefore, in Gnuastro, the type used for binary dataset is `uint8_t` (see Section 4.5 [Numeric data types], page 115). Although it does take 8-times more memory, this choice offers much better performance and the some extra (useful) features.

The advantage of using a full byte for each element of a binary dataset is that you can also have other values (that will be ignored in the processing). One such common "other" value in real datasets is a blank value (to mark regions that should not be processed because there is no data). The constant `GAL_BLANK_UINT8` value must be used in these cases (see Section 11.3.5 [Library blank values (`blank.h`)], page 343). Another is some temporary value(s) that can be given to a processed pixel to avoid having another copy of the dataset as in `GAL_BINARY_TMP_VALUE` that is described below.

`GAL_BINARY_TMP_VALUE` [Macro]

The functions described below work on a `uint8_t` type dataset with values of 1 or 0 (no other pixel will be touched). However, in some cases, it is necessary to put temporary values in each element during the processing of the functions. This temporary value has a special meaning for the operation and will be operated on. So if your input datasets have values other than 0 and 1 that you don't want these functions to work on, be sure they are not equal to this macro's value. Note that this value is also different from `GAL_BLANK_UINT8`, so your input datasets may also contain blank elements.

`gal_data_t *` [Function]
`gal_binary_erode` (*gal_data_t* `*input`, *size_t* `num`, *int* `connectivity`, *int* `inplace`)

Do `num` erosions on the `connectivity`-connected neighbors of `input` (see above for the definition of connectivity).

If `inplace` is non-zero *and* the input's type is `GAL_TYPE_UINT8`, then the erosion will be done within the input dataset and the returned pointer will be `input`. Otherwise, `input` is copied (and converted if necessary) to `GAL_TYPE_UINT8` and erosion will be done on this new dataset which will also be returned. This function will only work on the elements with a value of 1 or 0. It will leave all the rest unchanged.

Erosion (inverse of dilation) is an operation in mathematical morphology where each foreground pixel that is touching a background pixel is flipped (changed to background). The `connectivity` value determines the definition of "touching". Erosion will thus decrease the area of the foreground regions by one layer of pixels.

`gal_data_t *` [Function]
`gal_binary_dilate` (*gal_data_t* `*input`, *size_t* `num`, *int* `connectivity`, *int* `inplace`)

Do `num` dilations on the `connectivity`-connected neighbors of `input` (see above for the definition of connectivity). For more on `inplace` and the output, see `gal_binary_erode`.

Dilation (inverse of erosion) is an operation in mathematical morphology where each background pixel that is touching a foreground pixel is flipped (changed to foreground). The `connectivity` value determines the definition of "touching". Dilation will thus increase the area of the foreground regions by one layer of pixels.

`gal_data_t *` [Function]
`gal_binary_open` (*gal_data_t* `*input`, *size_t* `num`, *int* `connectivity`, *int* `inplace`)

Do `num` openings on the `connectivity`-connected neighbors of `input` (see above for the definition of connectivity). For more on `inplace` and the output, see `gal_binary_erode`.

Opening is an operation in mathematical morphology which is defined as erosion followed by dilation (see above for the definitions of erosion and dilation). Opening will thus remove the outer structure of the foreground. In this implementation, `num` erosions are going to be applied on the dataset, then `num` dilations.

`size_t` [Function]
`gal_binary_connected_components` (*gal_data_t* `*binary`, *gal_data_t* `**out`, *int* `connectivity`)

Return the number of connected components in `binary` through the breadth first search algorithm (finding all pixels belonging to one component before going on to the next). Connection between two pixels is defined based on the value to `connectivity`. `out` is a dataset with the same size as `binary` with `GAL_TYPE_INT32` type. Every pixel in `out` will have the label of the connected component it belongs to. The labeling of connected components starts from 1, so a label of zero is given to the input's background pixels.

When `*out!=NULL` (its space is already allocated), it will be cleared (to zero) at the start of this function. Otherwise, when `*out==NULL`, the necessary dataset to keep the output will be allocated by this function.

`binary` must have a type of `GAL_TYPE_UINT8`, otherwise this function will abort with an error. Other than blank pixels (with a value of `GAL_BLANK_UINT8` defined in Section 11.3.5 [Library blank values (`blank.h`)], page 343), all other non-zero pixels in `binary` will be considered as foreground (and will be labeled). Blank pixels in the input will also be blank in the output.

`gal_data_t *` [Function]
`gal_binary_connected_adjacency_matrix` (*gal_data_t* `*adjacency`, *size_t* `*numconnected`)

Find the number of connected labels and new labels based on an adjacency matrix, which must be a square binary array (type `GAL_TYPE_UINT8`). The returned dataset is a list of new labels for each old label. In other words, this function will find the objects that are connected (possibly through a third object) and in the output array,

the respective elements for all input labels is going to have the same value. The total number of connected labels is put into the space that `numconnected` points to.

An adjacency matrix defines connection between two labels. For example, let's assume we have 5 labels and we know that labels 1 and 5 are connected to label 3, but are not connected with each other. Also, labels 2 and 4 are not touching any other label. So in total we have 3 final labels: one combined object (merged from labels 1, 3, and 5) and the initial labels 2 and 4. The input adjacency matrix would look like this (note the extra row and column for a label 0 which is ignored):

```
                  INPUT                                  OUTPUT
                  =====                                  ======
               in_lab  1  2  3  4  5   |
                                       |       numconnected = 3
                    0  0  0  0  0  0   |
     in_lab 1 -->   0  0  0  1  0  0   |
     in_lab 2 -->   0  0  0  0  0  0   |  Returned: new labels for the
     in_lab 3 -->   0  1  0  0  0  1   |       5 initial objects
     in_lab 4 -->   0  0  0  0  0  0   |   | 0 | 1 | 2 | 1 | 3 | 1 |
     in_lab 5 -->   0  0  0  1  0  0   |
```

Although the adjacency matrix as used here is symmetric, currently this function assumes that it is filled on both sides of the diagonal.

`gal_data_t *` [Function]
`gal_binary_holes_label` (*gal_data_t* `*input`, *int* `connectivity`, *size_t* `*numholes`)

Label all the holes in the foreground (non-zero elements in input) as independent regions. Holes are background regions (zero-valued in input) that are fully surrounded by the foreground, as defined by `connectivity`. The returned dataset has a 32-bit signed integer type with the size of the input. All holes in the input will have labels/counters greater or equal to `1`. The rest of the background regions will still have a value of `0` and the initial foreground pixels will have a value of `-1`. The total number of holes will be written where `numholes` points to.

`void` [Function]
`gal_binary_holes_fill` (*gal_data_t* `*input`, *int* `connectivity`, *size_t* `maxsize`)

Fill all the holes (0 valued pixels surrounded by 1 valued pixels) of the binary `input` dataset. The connectivity of the holes can be set with `connectivity`. Holes larger than `maxsize` are not filled. This function currently only works on a 2D dataset.

11.3.23 Labeled datasets (`label.h`)

A labeled dataset is one where each element/pixel has an integer label (or counter). The label identifies the group/class that the element belongs to. This form of labeling allows the higher-level study of all pixels within a certain class.

For example, to detect objects/targets in an image/dataset, you can apply a threshold to separate the noise from the signal (to detect diffuse signal, a threshold is useless and more advanced methods are necessary, for example Section 7.2 [NoiseChisel], page 225). But the output of detection is a binary dataset (which is just a very low-level labeling of `0` for noise and `1` for signal).

The raw detection map is therefore hardly useful for any kind of analysis on objects/targets in the image. One solution is to use a connected-components algorithm (see `gal_binary_connected_components` in Section 11.3.22 [Binary datasets (`binary.h`)], page 423). It is a simple and useful way to separate/label connected patches in the foreground. This higher-level (but still elementary) labeling therefore allows you to count how many connected patches of signal there are in the dataset and is a major improvement compared to the raw detection.

However, when your objects/targets are touching, the simple connected components algorithm is not enough and a still higher-level labeling mechanism is necessary. This brings us to the necessity of the functions in this part of Gnuastro's library. The main inputs to the functions in this section are already labeled datasets (for example with the connected components algorithm above).

Each of the labeled regions are independent of each other (the labels specify different classes of targets). Therefore, especially in large datasets, it is often useful to process each label on independent CPU threads in parallel rather than in series. Therefore the functions of this section actually use an array of pixel/element indexs (belonging to each label/class) as the main identifier of a region. Using indexs will also allow processing of overlapping labels (for example in deblending problems). Just note that overlapping labels are not yet implemented, but planned. You can use `gal_label_indexs` to generate lists of indexs belonging to separate classes from the labeled input.

`GAL_LABEL_INIT` [Macro]
`GAL_LABEL_RIVER` [Macro]
`GAL_LABEL_TMPCHECK` [Macro]

Special negative integer values used internally by some of the functions in this section. Recall that meaningful labels are considered to be positive integers (≥ 1). Zero is conventionally kept for regions with no labels, therefore negative integers can be used for any extra classification in the labeled datasets.

`gal_data_t *` [Function]
`gal_label_indexs` (*gal_data_t* `*labels`, *size_t* `numlabs`, *size_t* `minmapsize`)

Return an array of `gal_data_t` containers, each containing the pixel indexs of the respective label (see Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346). `labels` contains the label of each element and has to have an `GAL_TYPE_INT32` type (see Section 11.3.3 [Library data types (`type.h`)], page 337). Only positive (greater than zero) values in `labels` will be used/indexed, other elements will be ignored.

Meaningful labels start from `1` and not `0`, therefore the output array of `gal_data_t` will contain `numlabs+1` elements. The first (zero-th) element of the output (`indexs[0]` in the example below) will be initialized to a dataset with zero elements. This will allow easy (non-confusing) access to the indexs of each (meaningful) label.

`numlabs` is the number of labels in the dataset. If it is given a value of zero, then the maximum value in the input (largest label) will be found and used. Therefore if it is given, but smaller than the actual number of labels, this function may/will crash (it will write in unallocated space). `numlabs` is therefore useful in a highly optimized/checked environment.

For example, if the returned array is called `indexs`, then `indexs[10].size` contains the number of elements that have a label of `10` in `labels` and `indexs[10].array`

is an array (after casting to `size_t *`) containing the indexs of each one of those elements/pixels.

By *index* we mean the 1D position: the input number of dimensions is irrelevant (any dimensionality is supported). In other words, each element's index is the number of elements/pixels between it and the dataset's first element/pixel. Therefore it is always greater or equal to zero and stored in `size_t` type.

`size_t` [Function]
`gal_label_watershed` (*gal_data_t* `*values`, *gal_data_t* `*indexs`, *gal_data_t* `*label`, *size_t* `*topinds`, *int* `min0_max1`)

Use the watershed algorithm[21] to "over-segment" the pixels in the `indexs` dataset based on values in the `values` dataset. Internally, each local extrema (maximum or minimum, based on `min0_max1`) and its surrounding pixels will be given a unique label. For demonstration, see Figures 8 and 9 of Akhlaghi and Ichikawa [2015] (`http://arxiv.org/abs/1505.01664`). If `topinds!=NULL`, it is assumed to point to an already allocated space to write the index of each clump's local extrema, otherwise, it is ignored.

The `values` dataset must have a 32-bit floating point type (`GAL_TYPE_FLOAT32`, see Section 11.3.3 [Library data types (`type.h`)], page 337) and will only be read by this function. `indexs` must contain the indexs of the elements/pixels that will be over-segmented by this function and have a `GAL_TYPE_SIZE_T` type, see the description of `gal_label_indexs`, above. The final labels will be written in the respective positions of `labels`, which must have a `GAL_TYPE_INT32` type and be the same size as `values`.

When `indexs` is already sorted, this function will ignore `min0_max1`. To judge if the dataset is sorted or not (by the values the indexs correspond to in `values`, not the actual indexs), this function will look into the bits of `indexs->flag`, for the respective bit flags, see Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346. If `indexs` is not already sorted, this function will sort it according to the values of the respective pixel in `values`. The increasing/decreasing order will be determined by `min0_max1`. Note that if this function is called on multiple threads *and* `values` points to a different array on each thread, this function will not return a reasonable result. In this case, please sort `indexs` prior to calling this function (see `gal_qsort_index_multi_d` in Section 11.3.18 [Qsort functions (`qsort.h`)], page 412).

When `indexs` is decreasing (increasing), or `min0_max1` is `1` (`0`), local minima (maxima), are considered rivers (watersheds) and given a label of `GAL_LABEL_RIVER` (see above).

Note that rivers/watersheds will also be formed on the edges of the labeled regions or when the labeled pixels touch a blank pixel. Therefore this function will need to check for the presence of blank values. To be most efficient, it is thus recommended to use `gal_blank_present` (with `updateflag=1`) prior to calling this function (see Section 11.3.5 [Library blank values (`blank.h`)], page 343. Once the flag has been

[21] The watershed algorithm was initially introduced by Vincent and Soille (`https://doi.org/10.1109/34.87344`). It starts from the minima and puts the pixels in, one by one, to grow them until the touch (create a watershed). For more, also see the Wikipedia article: `https://en.wikipedia.org/wiki/Watershed_%28image_processing%29`.

set, no other function (including this one) that needs special behavior for blank pixels will have to parse the dataset to see if it has blank values any more.

If you are sure your dataset doesn't have blank values (by the design of your software), to avoid an extra parsing of the dataset and improve performance, you can set the two bits manually (see the description of `flags` in Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346):

```
input->flags |=  GAL_DATA_FLAG_BLANK_CH; /* Set bit to 1. */
input->flags &= ~GAL_DATA_FLAG_HASBLANK; /* Set bit to 0. */
```

`void` [Function]
`gal_label_clump_significance` (*gal_data_t* `*values`, *gal_data_t* `*std`, *gal_data_t* `*label`, *gal_data_t* `*indexs`, *struct gal_tile_two_layer_params* `*tl`, *size_t* `numclumps`, *size_t* `minarea`, *int* `variance`, *int* `keepsmall`, *gal_data_t* `*sig`, *gal_data_t* `*sigind`)

This function is usually called after `gal_label_watershed`, and is used as a measure to identify which over-segmented "clumps" are real and which are noise.

A measurement is done on each clump (using the `values` and `std` datasets, see below). To help in multi-threaded environments, the operation is only done on pixels which are indexed in `indexs`. It is expected for `indexs` to be sorted by their values in `values`. If not sorted, the measurement may not be reliable. If sorted in a decreasing order, then clump building will start from their highest value and vice-versa. See the description of `gal_label_watershed` for more on `indexs`.

Each "clump" (identified by a positive integer) is assumed to be surrounded by atleast one river/watershed pixel (with a non-positive label). This function will parse the pixels identified in `indexs` and make a measurement on each clump and over all the river/watershed pixels. The number of clumps (`numclumps`) must be given as an input argument and any clump that is smaller than `minarea` is ignored (because of scatter). If `variance` is non-zero, then the `std` dataset is interpreted as variance, not standard deviation.

The `values` and `std` datasets must have a `float` (32-bit floating point) type. Also, `label` and `indexs` must respectively have `int32` and `size_t` types. `values` and `label` must have the same size, but `std` can have three possible sizes: 1) a single element (which will be used for the whole dataset, 2) the same size as `values` (so a different error can be assigned to every pixel), 3) a single value for each tile, based on the `tl` tessellation (see Section 11.3.15.2 [Tile grid], page 405). In the last case, a tile/value will be associated to each clump based on its flux-weighted (only positive values) center.

The main output is an internally allocated, 1-dimensional array with one value per label. The array information (length, type and etc) will be written into the `sig` generic data container. Therefore `sig->array` must be `NULL` when this function is called. After this function, the details of the array (number of elements, type and size and etc) will be written in to the various components of `sig`, see the definition of `gal_data_t` in Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346. Therefore `sig` must already be allocated before calling this function.

Optionally (when `sigind!=NULL`, similar to `sig`) the clump labels of each measurement in `sig` will be written in `sigind->array`. If `keepsmall` zero, small clumps (where no measurement is made) will not be included in the output table.

This function is initially intended for a multi-threaded environment. In such cases, you will be writing arrays of clump measures from different regions in parallel into an array of `gal_data_t`s. You can simply allocate (and initialize), such an array with the `gal_data_array_calloc` function in Section 11.3.6.3 [Arrays of datasets], page 352. For example if the `gal_data_t` array is called `array`, you can pass `&array[i]` as `sig`.

Along with some other functions in `label.h`, this function was initially written for Section 7.3 [Segment], page 243. The description of the parameter used to measure a clump's significance is fully given in Section 7.3.1 [Segment changes after publication], page 245.

`void` [Function]
`gal_label_grow_indexs` (*gal_data_t* `*labels`, *gal_data_t* `*indexs`, *int* `withrivers`, *int* `connectivity`)

Grow the (positive) labels of `labels` over the pixels in `indexs` (see description of `gal_label_indexs`). The pixels (position in `indexs`, values in `labels`) that must be "grown" must have a value of `GAL_LABEL_INIT` in `labels` before calling this function. For a demonstration see Columns 2 and 3 of Figure 10 in Akhlaghi and Ichikawa [2015] (`http://arxiv.org/abs/1505.01664`).

In many aspects, this function is very similar to over-segmentation (watershed algorithm, `gal_label_watershed`). The big difference is that in over-segmentation local maximums (that aren't touching any already labeled pixel) get a separate label. However, here the final number of labels will not change. All pixels that aren't directly touching a labeled pixel just get pushed back to the start of the loop, and the loop iterates until its size doesn't change any more. This is because in a generic scenario some of the indexed pixels might not be reachable through other indexed pixels.

The next major difference with over-segmentation is that when there is only one label in growth region(s), it is not mandatory for `indexs` to be sorted by values. If there are multiple labeled regions in growth region(s), then values are important and you can use `qsort` with `gal_qsort_index_single_d` to sort the indexs by values in a separate array (see Section 11.3.18 [Qsort functions (`qsort.h`)], page 412).

This function looks for positive-valued neighbors of each pixel in `indexs` and will label a pixel if it touches one. Therefore, it is very important that only pixels/labels that are intended for growth have positive values in `labels` before calling this function. Any non-positive (zero or negative) value will be ignored as a label by this function. Thus, it is recommended that while filling in the 'indexs' array values, you initialize all the pixels that are in `indexs` with `GAL_LABEL_INIT`, and set non-labeled pixels that you don't want to grow to `0`.

This function will write into both the input datasets. After this function, some of the non-positive `labels` pixels will have a new positive label and the number of useful elements in `indexs` will have decreased. The index of those pixels that couldn't be labeled will remain inside `indexs`. If `withrivers` is non-zero, then pixels that are immediately touching more than one positive value will be given a `GAL_LABEL_RIVER` label.

Note that the `indexs->array` is not re-allocated to its new size at the end[22]. But since `indexs->dsize[0]` and `indexs->size` have new values after this function is returned, the extra elements just won't be used until they are ultimately freed by `gal_data_free`.

Connectivity is a value between `1` (fewest number of neighbors) and the number of dimensions in the input (most number of neighbors). For example in a 2D dataset, a connectivity of `1` and `2` corresponds to 4-connected and 8-connected neighbors.

11.3.24 Convolution functions (`convolve.h`)

Convolution is a very common operation during data analysis and is thoroughly described as part of Gnuastro's Section 6.3 [Convolve], page 177, program which is fully devoted to this job. Because of the complete introduction that was presented there, we will directly skip onto the currently available convolution functions in Gnuastro's library.

As of this version, only spatial domain convolution is available in Gnuastro's libraries. We haven't had the time to liberate the frequency domain function convolution and deconvolution functions that are available in the Convolve program[23].

`gal_data_t *` [Function]
`gal_convolve_spatial` (*gal_data_t* `*tiles`, *gal_data_t* `*kernel`, *size_t* `numthreads`, *int* `edgecorrection`, *int* `convoverch`)

Convolve the given `tiles` dataset (possibly a list of tiles, see Section 11.3.8.9 [List of `gal_data_t`], page 368, and Section 11.3.15 [Tessellation library (`tile.h`)], page 399) with `kernel` on `numthreads` threads. When `edgecorrection` is non-zero, it will correct for the edge dimming effects as discussed in Section 6.3.1.2 [Edges in the spatial domain], page 178.

`tiles` can be a single/complete dataset, but in that case the speed will be very slow. Therefore, for larger images, it is recommended to give a list of tiles covering a dataset. To create a tessellation that fully covers an input image, you may use `gal_tile_full`, or `gal_tile_full_two_layers` to also define channels over your input dataset. These functions are discussed in Section 11.3.15.2 [Tile grid], page 405. You may then pass the list of tiles to this function. This is the recommended way to call this function because spatial domain convolution is slow and breaking the job into many small tiles and working on simultaneously on several threads can greatly speed up the processing.

If the tiles are defined within a channel (a larger tile), by default convolution will be done within the channel, so pixels on the edge of a channel will not be affected by their neighbors that are in another channel. See Section 4.7 [Tessellation], page 123, for the necessity of channels in astronomical data analysis. This behavior may be disabled when `convoverch` is non-zero. In this case, it will ignore channel borders (if they exist) and mix all pixels that cover the kernel within the dataset.

[22] Note that according to the GNU C Library, even a `realloc` to a smaller size can also cause a re-write of the whole array, which is not a cheap operation.

[23] Hence any help would be greatly appreciated.

`void` [Function]
`gal_convolve_spatial_correct_ch_edge` (*gal_data_t* `*tiles`, *gal_data_t* `*kernel`, *size_t* `numthreads`, *int* `edgecorrection`, *gal_data_t* `*tocorrect`)

Correct the edges of channels in an already convolved image when it was initially convolved with `gal_convolve_spatial` and `convoverch==0`. In that case, strong boundaries might exist on the channel edges. So if you later need to remove those boundaries at later steps of your processing, you can call this function. It will only do convolution on the tiles that are near the edge and were effected by the channel borders. Other pixels in the image will not be touched. Hence, it is much faster.

11.3.25 Interpolation (`interpolate.h`)

During data analysis, it happens that parts of the data cannot be given a value, but one is necessary for the higher-level analysis. For example a very bright star saturated part of your image and you need to fill in the saturated pixels with some values. Another common usage case are masked sky-lines in 1D spectra that similarly need to be assigned a value for higher-level analysis. In other situations, you might want a value in an arbitrary point: between the elements/pixels where you have data. The functions described in this section are for such operations.

The parametric interpolations discussed below are wrappers around the interpolation functions of the GNU Scientific Library (or GSL, see Section 3.1.1.1 [GNU Scientific library], page 62). To identify the different GSL interpolation types, Gnuastro's `gnuastro/interpolate.h` header file contains macros that are discussed below. The GSL wrappers provided here are not yet complete because we are too busy. If you need them, please consider helping us in adding them to Gnuastro's library. Your would be very welcome and appreciated.

`gal_data_t *` [Function]
`gal_interpolate_close_neighbors` (*gal_data_t* `*input`, *struct gal_tile_two_layer_params* `*tl`, *size_t* `numneighbors`, *size_t* `numthreads`, *int* `onlyblank`, *int* `aslinkedlist`)

Interpolate the values in the image using the median value of their `numneighbors` closest neighbors. This function is non-parametric and thus agnostic to the input's number of dimension. If `onlyblank` is non-zero, then only blank elements will be interpolated and pixels that already have a value will be left untouched. This function is multi-threaded and will run on `numthreads` threads (see `gal_threads_number` in Section 11.3.2 [Multithreaded programming (`threads.h`)], page 333).

`tl` is Gnuastro's two later tessellation structure used to define tiles over an image and is fully described in Section 11.3.15.2 [Tile grid], page 405. When `tl!=NULL`, then it is assumed that the `input->array` contains one value per tile and interpolation will respect certain tessellation properties, for example to not interpolate over channel borders.

If several datasets have the same set of blank values, you don't need to call this function multiple times. When `aslinkedlist` is non-zero, then `input` will be seen as a Section 11.3.8.9 [List of `gal_data_t`], page 368. In this case, the same neighbors will be used for all the datasets in the list. Of course, the values for each dataset will

be different, so a different value will be written in the each dataset, but the neighbor checking that is the most CPU intensive part will only be done once.

This is a non-parametric and robust function for interpolation. The interpolated values are also always within the range of the non-blank values and strong outliers do not get created. However, this type of interpolation must be used with care when there are gradients. This is because it is non-parametric and if there aren't enough neighbors, step-like features can be created.

`GAL_INTERPOLATE_1D_INVALID` [Macro]
This is just a place holder to manage errors.

`GAL_INTERPOLATE_1D_LINEAR` [Macro]
[From GSL:] Linear interpolation. This interpolation method does not require any additional memory.

`GAL_INTERPOLATE_1D_POLYNOMIAL` [Macro]
[From GSL:] Polynomial interpolation. This method should only be used for interpolating small numbers of points because polynomial interpolation introduces large oscillations, even for well-behaved datasets. The number of terms in the interpolating polynomial is equal to the number of points.

`GAL_INTERPOLATE_1D_CSPLINE` [Macro]
[From GSL:] Cubic spline with natural boundary conditions. The resulting curve is piecewise cubic on each interval, with matching first and second derivatives at the supplied data-points. The second derivative is chosen to be zero at the first point and last point.

`GAL_INTERPOLATE_1D_CSPLINE_PERIODIC` [Macro]
[From GSL:] Cubic spline with periodic boundary conditions. The resulting curve is piecewise cubic on each interval, with matching first and second derivatives at the supplied data-points. The derivatives at the first and last points are also matched. Note that the last point in the data must have the same y-value as the first point, otherwise the resulting periodic interpolation will have a discontinuity at the boundary.

`GAL_INTERPOLATE_1D_AKIMA` [Macro]
[From GSL:] Non-rounded Akima spline with natural boundary conditions. This method uses the non-rounded corner algorithm of Wodicka.

`GAL_INTERPOLATE_1D_AKIMA_PERIODIC` [Macro]
[From GSL:] Non-rounded Akima spline with periodic boundary conditions. This method uses the non-rounded corner algorithm of Wodicka.

`GAL_INTERPOLATE_1D_STEFFEN` [Macro]
[From GSL:] Steffen's method[24] guarantees the monotonicity of the interpolating function between the given data points. Therefore, minima and maxima can only occur exactly at the data points, and there can never be spurious oscillations between data points. The interpolated function is piecewise cubic in each interval. The resulting curve and its first derivative are guaranteed to be continuous, but the second derivative may be discontinuous.

[24] `http://adsabs.harvard.edu/abs/1990A%26A...239..443S`

`gsl_spline *` [Function]
`gal_interpolate_1d_make_gsl_spline` (*gal_data_t* `*X`, *gal_data_t* `*Y`, *int* `type_1d`)

Allocate and initialize a GNU Scientific Library (GSL) 1D `gsl_spline` structure using the non-blank elements of `Y`. `type_1d` identifies the interpolation scheme and must be one of the `GAL_INTERPOLATE_1D_*` macros defined above.

If `X==NULL`, the X-axis is assumed to be integers starting from zero (the index of each element in `Y`). Otherwise, the values in `X` will be used to initialize the interpolation structure. Note that when given, `X` must *not* contain any blank elements and it must be sorted (in increasing order).

Each interpolation scheme needs a minimum number of elements to successfully operate. If the number of non-blank values in `Y` is less than this number, this function will return a `NULL` pointer.

To be as generic and modular as possible, GSL's tools are low-level. Therefore before doing the interpolation, many steps are necessary (like preparing your dataset, then allocating and initializing `gsl_spline`). The metadata available in Gnuastro's Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346, make it easy to hide all those preparations within this function.

Once `gsl_spline` has been initialized by this function, the interpolation can be evaluated for any X value within the non-blank range of the input using `gsl_spline_eval` or `gsl_spline_eval_e`.

For example in the small program below, we read the first two columns of the table in `table.txt` and feed them to this function to later estimate the values in the second column for three selected points. You can use Section 11.2 [BuildProgram], page 328, to compile and run this function, see Section 11.4 [Library demo programs], page 437, for more.

```
#include <stdio.h>
#include <stdlib.h>
#include <gnuastro/table.h>
#include <gnuastro/interpolate.h>

int
main(void)
{
  size_t i;
  gal_data_t *X, *Y;
  gsl_spline *spline;
  gsl_interp_accel *acc;
  gal_list_str_t *cols=NULL;

  /* Change the values based on your input table. */
  double points[]={1.8, 2.5, 10.3};

  /* Read the first two columns from 'tab.txt'.
     IMPORTANT: the list is first-in-first-out, so the output
```

```
   column order is the inverse of the input order. */
gal_list_str_add(&cols, "1", 0);
gal_list_str_add(&cols, "2", 0);
Y=gal_table_read("table.txt", NULL, cols, GAL_TABLE_SEARCH_NAME,
                 0, -1, NULL);
X=Y->next;

/* Allocate the GSL interpolation accelerator and make the
   'gsl_spline' structure. */
acc=gsl_interp_accel_alloc();
spline=gal_interpolate_1d_make_gsl_spline(X, Y,
                                GAL_INTERPOLATE_1D_STEFFEN);

/* Calculate the respective value for all the given points,
   if 'spline' could be allocated. */
if(spline)
  for(i=0; i<(sizeof points)/(sizeof *points); ++i)
    printf("%f: %f\n", points[i],
           gsl_spline_eval(spline, points[i], acc));

/* Clean up and return. */
gal_data_free(X);
gal_data_free(Y);
gsl_spline_free(spline);
gsl_interp_accel_free(acc);
gal_list_str_free(cols, 0);
return EXIT_SUCCESS;
}
```

`void` [Function]
`gal_interpolate_1d_blank` (*gal_data_t* `*in`, *int* `type_1d`)

Fill the blank elements of `in` using the rest of the elements and the given interpolation. The interpolation scheme can be set through `type_1d`, which accepts any of the `GAL_INTERPOLATE_1D_*` macros above. The interpolation is internally done in 64-bit floating point type (`double`). However the evaluated/interpolated values (originally blank) will be written (in `in`) with its original numeric datatype, using C's standard type conversion.

By definition, interpolation is only defined "between" valid points. Therefore, if any number of elements on the start or end of the 1D array are blank, those elements will not be interpolated and will remain blank. To see if any blank (non-interpolated) elements remain, you can use `gal_blank_present` on `in` after this function is finished.

11.3.26 Git wrappers (`git.h`)

Git is one of the most common tools for version control and it can often be useful during development, for example see `COMMIT` keyword in Section 4.9 [Output FITS files], page 125. At installation time, Gnuastro will also check for the existence of libgit2, and store the value in the `GAL_CONFIG_HAVE_LIBGIT2`, see Section 11.3.1 [Configuration information (`config.h`)],

page 332, and Section 3.1.2 [Optional dependencies], page 64. `gnuastro/git.h` includes `gnuastro/config.h` internally, so you won't have to include both for this macro.

`char *` [Function]
`gal_git_describe ( )`

When libgit2 is present and the program is called within a directory that is version controlled, this function will return a string containing the commit description (similar to Gnuastro's unofficial version number, see Section 1.5 [Version numbering], page 7). If there are uncommitted changes in the running directory, it will add a '`-dirty`' prefix to the description. When there is no tagged point in the previous commit, this function will return a uniquely abbreviated commit object as fallback. This function is used for generating the value of the `COMMIT` keyword in Section 4.9 [Output FITS files], page 125. The output string is similar to the output of the following command:

```
$ git describe --dirty --always
```

Space for the output string is allocated within this function, so after using the value you have to `free` the output string. If libgit2 is not installed or the program calling this function is not within a version controlled directory, then the output will be the `NULL` pointer.

11.3.27 Cosmology library (`cosmology.h`)

This library does the main cosmological calculations that are commonly necessary in extragalactic astronomical studies. The main variable in this context is the redshift (z). The cosmological input parameters in the functions below are `H0`, `o_lambda_0`, `o_matter_0`, `o_radiation_0` which respectively represent the current (at redshift 0) expansion rate (Hubble constant in units of km/sec/Mpc), cosmological constant (Λ), matter and radiation densities.

All these functions are declared in `gnuastro/cosmology.h`. For a more extended introduction/discussion of the cosmological parameters, please see Section 9.1 [CosmicCalculator], page 307.

`double` [Function]
`gal_cosmology_age` (*double* `z`, *double* `H0`, *double* `o_lambda_0`, *double* `o_matter_0`, *double* `o_radiation_0`)

Returns the age of the universe at redshift `z` in units of Giga years.

`double` [Function]
`gal_cosmology_proper_distance` (*double* `z`, *double* `H0`, *double* `o_lambda_0`, *double* `o_matter_0`, *double* `o_radiation_0`)

Returns the proper distance to an object at redshift `z` in units of Mega parsecs.

`double` [Function]
`gal_cosmology_comoving_volume` (*double* `z`, *double* `H0`, *double* `o_lambda_0`, *double* `o_matter_0`, *double* `o_radiation_0`)

Returns the comoving volume over 4pi stradian to `z` in units of Mega parsecs cube.

`double` [Function]
`gal_cosmology_critical_density` (*double* `z`, *double* `H0`, *double* `o_lambda_0`, *double* `o_matter_0`, *double* `o_radiation_0`)

Returns the critical density at redshift `z` in units of g/cm^3.

`double` [Function]
`gal_cosmology_angular_distance` (*double* `z`, *double* `H0`, *double* `o_lambda_0`, *double* `o_matter_0`, *double* `o_radiation_0`)
Return the angular diameter distance to an object at redshift `z` in units of Mega parsecs.

`double` [Function]
`gal_cosmology_luminosity_distance` (*double* `z`, *double* `H0`, *double* `o_lambda_0`, *double* `o_matter_0`, *double* `o_radiation_0`)
Return the luminosity diameter distance to an object at redshift `z` in units of Mega parsecs.

`double` [Function]
`gal_cosmology_distance_modulus` (*double* `z`, *double* `H0`, *double* `o_lambda_0`, *double* `o_matter_0`, *double* `o_radiation_0`)
Return the distance modulus at redshift `z` (with no units).

`double` [Function]
`gal_cosmology_to_absolute_mag` (*double* `z`, *double* `H0`, *double* `o_lambda_0`, *double* `o_matter_0`, *double* `o_radiation_0`)
Return the conversion from apparent to absolute magnitude for an object at redshift `z`. This value has to be added to the apparent magnitude to give the absolute magnitude of an object at redshift `z`.

11.4 Library demo programs

In this final section of Chapter 11 [Library], page 320, we give some example Gnuastro programs to demonstrate various features in the library. All these programs have been tested and once Gnuastro is installed you can compile and run them with with Gnuastro's Section 11.2 [BuildProgram], page 328, program that will take care of linking issues. If you don't have any FITS file to experiment on, you can use those that are generated by Gnuastro after `make check` in the `tests/` directory, see Section 1.1 [Quick start], page 1.

11.4.1 Library demo - reading a FITS image

The following simple program demonstrates how to read a FITS image into memory and use the `void *array` pointer in of Section 11.3.6.1 [Generic data container (`gal_data_t`)], page 346. For easy linking/compilation of this program along with a first run see Section 11.2 [BuildProgram], page 328. Before running, also change the `filename` and `hdu` variable values to specify an existing FITS file and/or extension/HDU.

This is just intended to demonstrate how to use the `array` pointer of `gal_data_t`. Hence it doesn't do important sanity checks, for example in real datasets you may also have blank pixels. In such cases, this program will return a NaN value (see Section 6.1.3 [Blank pixels], page 154). So for general statistical information of a dataset, it is much better to use Gnuastro's Section 7.1 [Statistics], page 208, program which can deal with blank pixels any many other issues in a generic dataset.

```
#include <stdio.h>
#include <stdlib.h>
#include <gnuastro/fits.h> /* includes gnuastro's data.h and type.h */
```

```
#include <gnuastro/statistics.h>

int
main(void)
{
  size_t i;
  float *farray;
  double sum=0.0f;
  gal_data_t *image;
  char *filename="img.fits", *hdu="1";


  /* Read 'img.fits' (HDU: 1) as a float32 array. */
  image=gal_fits_img_read_to_type(filename, hdu, GAL_TYPE_FLOAT32, -1);


  /* Use the allocated space as a single precision floating
   * point array (recall that 'image->array' has 'void *'
   * type, so it is not directly usable. */
  farray=image->array;


  /* Calculate the sum of all the values. */
  for(i=0; i<image->size; ++i)
    sum += farray[i];


  /* Report the sum. */
  printf("Sum of values in %s (hdu %s) is: %f\n",
         filename, hdu, sum);


  /* Clean up and return. */
  gal_data_free(image);
  return EXIT_SUCCESS;
}
```

11.4.2 Library demo - inspecting neighbors

The following simple program shows how you can inspect the neighbors of a pixel using the `GAL_DIMENSION_NEIGHBOR_OP` function-like macro that was introduced in Section 11.3.7 [Dimensions (`dimension.h`)], page 353. For easy linking/compilation of this program along with a first run see Section 11.2 [BuildProgram], page 328. Before running, also change the file name and HDU (first and second arguments to `gal_fits_img_read_to_type`) to specify an existing FITS file and/or extension/HDU.

```
#include <stdio.h>
#include <gnuastro/fits.h>
```

```
#include <gnuastro/dimension.h>

int
main(void)
{
  double sum;
  float *array;
  size_t i, num, *dinc;
  gal_data_t *input=gal_fits_img_read_to_type("input.fits", "1",
                                              GAL_TYPE_FLOAT32, -1);

  /* To avoid the 'void *' pointer and have 'dinc'. */
  array=input->array;
  dinc=gal_dimension_increment(input->ndim, input->dsize);

  /* Go over all the pixels. */
  for(i=0;i<input->size;++i)
    {
      num=0;
      sum=0.0f;
      GAL_DIMENSION_NEIGHBOR_OP( i, input->ndim, input->dsize,
                                 input->ndim, dinc,
                                 {++num; sum+=array[nind];} );
      printf("%zu: num: %zu, sum: %f\n", i, num, sum);
    }

  /* Clean up and return. */
  gal_data_free(input);
  return EXIT_SUCCESS;
}
```

11.4.3 Library demo - multi-threaded operation

The following simple program shows how to use Gnuastro to simplify spinning off threads and distributing different jobs between the threads. The relevant thread-related functions are defined in Section 11.3.2.2 [Gnuastro's thread related functions], page 335. For easy linking/compilation of this program, along with a first run, see Gnuastro's Section 11.2 [BuildProgram], page 328. Before running, also change the `filename` and `hdu` variable values to specify an existing FITS file and/or extension/HDU.

This is a very simple program to open a FITS image, distribute its pixels between different threads and print the value of each pixel and the thread it was assigned to. The actual operation is very simple (and would not usually be done with threads in a real-life program). It is intentionally chosen to put more focus on the important steps in spinning of threads and how the worker function (which is called by each thread) can identify the job-IDs it should work on.

For example, instead of an array of pixels, you can define an array of tiles or any other context-specific structures as separate targets. The important thing is that each action

should have its own unique ID (counting from zero, as is done in an array in C). You can then follow the process below and use each thread to work on all the targets that are assigned to it. Recall that spinning-off threads is its self an expensive process and we don't want to spin-off one thread for each target (see the description of `gal_threads_dist_in_threads` in Section 11.3.2.2 [Gnuastro's thread related functions], page 335.

There are many (more complicated, real-world) examples of using `gal_threads_spin_off` in Gnuastro's actual source code, you can see them by searching for the `gal_threads_spin_off` function from the top source (after unpacking the tarball) directory (for example with this command):

```
$ grep -r gal_threads_spin_off ./
```

The code of this demonstration program is shown below. This program was also built and run when you ran `make check` during the building of Gnuastro (`tests/lib/multithread.c`, so it is already tested for your system and you can safely use it as a guide.

```
#include <stdio.h>
#include <stdlib.h>

#include <gnuastro/fits.h>
#include <gnuastro/threads.h>


/* This structure can keep all information you want to pass onto the
 * worker function on each thread. */
struct params
{
  gal_data_t *image;            /* Dataset to print values of. */
};




/* This is the main worker function which will be called by the
 * different threads. 'gal_threads_params' is defined in
 * 'gnuastro/threads.h' and contains the pointer to the parameter we
 * want. Note that the input argument and returned value of this
 * function always must have 'void *' type. */
void *
worker_on_thread(void *in_prm)
{
  /* Low-level definitions to be done first. */
  struct gal_threads_params *tprm=(struct gal_threads_params *)in_prm;
  struct params *p=(struct params *)tprm->params;


  /* Subsequent definitions. */
  float *array=p->image->array;
  size_t i, index, *dsize=p->image->dsize;
```

```
  /* Go over all the actions (pixels in this case) that were assigned
   * to this thread. */
  for(i=0; tprm->indexs[i] != GAL_BLANK_SIZE_T; ++i)
    {
      /* For easy reading. */
      index = tprm->indexs[i];


      /* Print the information. */
      printf("(%zu, %zu) on thread %zu: %g\n", index%dsize[1]+1,
             index/dsize[1]+1, tprm->id, array[index]);
    }


  /* Wait for all the other threads to finish, then return. */
  if(tprm->b) pthread_barrier_wait(tprm->b);
  return NULL;
}




/* High-level function (called by the operating system). */
int
main(void)
{
  struct params p;
  char *filename="input.fits", *hdu="1";
  size_t numthreads=gal_threads_number();


  /* Read the image into memory as a float32 data type. We are using
   * '-1' for 'minmapsize' to ensure that the image is read into
   * memory. */
  p.image=gal_fits_img_read_to_type(filename, hdu, GAL_TYPE_FLOAT32,
                                    -1);


  /* Print some basic information before the actual contents: */
  printf("Pixel values of %s (HDU: %s) on %zu threads.\n", filename,
         hdu, numthreads);
  printf("Used to check the compiled library's capability in opening "
         "a FITS file, and also spinning-off threads.\n");
```

```
  /* A small sanity check: this is only intended for 2D arrays (to
   * print the coordinates of each pixel). */
  if(p.image->ndim!=2)
    {
      fprintf(stderr, "only 2D images are supported.");
      exit(EXIT_FAILURE);
    }


  /* Spin-off the threads and do the processing on each thread. */
  gal_threads_spin_off(worker_on_thread, &p, p.image->size, numthreads);


  /* Clean up and return. */
  gal_data_free(p.image);
  return EXIT_SUCCESS;
}
```

11.4.4 Library demo - reading and writing table columns

Tables are some of the most common inputs to, and outputs of programs. This section contains a small program for reading and writing tables using the constructs described in Section 11.3.10 [Table input output (`table.h`)], page 370. For easy linking/compilation of this program, along with a first run, see Gnuastro's Section 11.2 [BuildProgram], page 328. Before running, also set the following file and column names in the first two lines of `main`. The input and output names may be `.txt` and `.fits` tables, `gal_table_read` and `gal_table_write` will be able to write to both formats. For plain text tables see see Section 4.6.2 [Gnuastro text table format], page 119.

This example program reads three columns from a table. The first two columns are selected by their name (`NAME1` and `NAME2`) and the third is selected by its number: column 10 (counting from 1). Gnuastro's column selection is discussed in Section 4.6.3 [Selecting table columns], page 121. The first and second columns can be any type, but this program will convert them to `int32_t` and `float` for its internal usage respectively. However, the third column must be double for this program. So if it isn't, the program will abort with an error. Having the columns in memory, it will print them out along with their sum (just a simple application, you can do what ever you want at this stage). Reading the table finishes here.

The rest of the program is a demonstration of writing a table. While parsing the rows, this program will change the first column (to be counters) and multiply the second by 10 (so the output will be different). Then it will define the order of the output columns by setting the `next` element (to create a Section 11.3.8.9 [List of `gal_data_t`], page 368). Before writing, this function will also set names for the columns (units and comments can be defined in a similar manner). Writing the columns to a file is then done through a simple call to `gal_table_write`.

The operations that are shown in this example program are not necessary all the time. For example, in many cases, you know the numerical data type of the column before writing

your program (see Section 4.5 [Numeric data types], page 115), so type checking and copying to a specific type won't be necessary.

```
#include <stdio.h>
#include <stdlib.h>

#include <gnuastro/table.h>

int
main(void)
{
  /* File names and column names (which may also be numbers). */
  char *c1_name="NAME1", *c2_name="NAME2", *c3_name="10";
  char *inname="input.fits", *hdu="1", *outname="out.fits";

  /* Internal parameters. */
  float *array2;
  double *array3;
  int32_t *array1;
  size_t i, counter=0;
  gal_data_t *c1, *c2;
  gal_data_t tmp, *col, *columns;
  gal_list_str_t *column_ids=NULL;

  /* Define the columns to read. */
  gal_list_str_add(&column_ids, c1_name, 0);
  gal_list_str_add(&column_ids, c2_name, 0);
  gal_list_str_add(&column_ids, c3_name, 0);

  /* The columns were added in reverse, so correct it. */
  gal_list_str_reverse(&column_ids);

  /* Read the desired columns. */
  columns = gal_table_read(inname, hdu, column_ids,
                           GAL_TABLE_SEARCH_NAME, 1, -1, NULL);

  /* Go over the columns, we'll assume that you don't know their type
   * a-priori, so we'll check  */
  counter=1;
  for(col=columns; col!=NULL; col=col->next)
    switch(counter++)
      {
      case 1:              /* First column: we want it as int32_t. */
        c1=gal_data_copy_to_new_type(col, GAL_TYPE_INT32);
        array1 = c1->array;
        break;
```

```
      case 2:              /* Second column: we want it as float.  */
        c2=gal_data_copy_to_new_type(col, GAL_TYPE_FLOAT32);
        array2 = c2->array;
        break;

      case 3:              /* Third column: it MUST be double.     */
        if(col->type!=GAL_TYPE_FLOAT64)
          {
            fprintf(stderr, "Column %s must be float64 type, it is "
                    "%s", c3_name, gal_type_name(col->type, 1));
            exit(EXIT_FAILURE);
          }
        array3 = col->array;
        break;
      }

  /* As an example application we'll just print them out. In the
   * meantime (just for a simple demonstration), change the first
   * array value to the counter and multiply the second by 10. */
  for(i=0;i<c1->size;++i)
    {
      printf("%zu: %d + %f + %f = %f\n", i+1, array1[i], array2[i],
             array3[i], array1[i]+array2[i]+array3[i]);
      array1[i]  = i+1;
      array2[i] *= 10;
    }

  /* Link the first two columns as a list. */
  c1->next = c2;
  c2->next = NULL;

  /* Set names for the columns and write them out. */
  c1->name = "COUNTER";
  c2->name = "VALUE";
  gal_table_write(c1, NULL, GAL_TABLE_FORMAT_BFITS, outname);

  /* The names weren't allocated, so to avoid cleaning-up problems,
   * we'll set them to NULL. */
  c1->name = c2->name = NULL;

  /* Clean up and return.  */
  gal_data_free(c1);
  gal_data_free(c2);
  gal_list_data_free(columns);
  gal_list_str_free(column_ids, 0); /* strings weren't allocated. */
  return EXIT_SUCCESS;
}
```

12 Developing

The basic idea of GNU Astronomy Utilities is for an interested astronomer to be able to easily understand the code of any of the programs or libraries, be able to modify the code if s/he feels there is an improvement and finally, to be able to add new programs or libraries for their own benefit, and the larger community if they are willing to share it. In short, we hope that at least from the software point of view, the "obscurantist faith in the expert's special skill and in his personal knowledge and authority" can be broken, see Section 1.2 [Science and its tools], page 2. With this aim in mind, Gnuastro was designed to have a very basic, simple, and easy to understand architecture for any interested inquirer.

This chapter starts with very general design choices, in particular Section 12.1 [Why C programming language?], page 445, and Section 12.2 [Program design philosophy], page 447. It will then get a little more technical about the Gnuastro code and file/directory structure in Section 12.3 [Coding conventions], page 448, and Section 12.4 [Program source], page 451. Section 12.4.2 [The TEMPLATE program], page 454, discusses a minimal (and working) template to help in creating new programs or easier learning of a program's internal structure. Some other general issues about documentation, building and debugging are then discussed. This chapter concludes with how you can learn about the development and get involved in Section 12.9 [Gnuastro project webpage], page 459, Section 12.10 [Developing mailing lists], page 460, and Section 12.11 [Contributing to Gnuastro], page 461.

12.1 Why C programming language?

Currently the programming language that is most commonly used in scientific applications is C++[1], Python[2], and Julia[3] (which is a newcomer but swiftly gaining ground). One of the main reasons behind this choice is their high-level abstractions. However, GNU Astronomy Utilities is fully written in the C programming language[4]. The reasons can be summarized with simplicity, portability and efficiency/speed. All three are very important in a scientific software and we will discuss them below.

Simplicity can best be demonstrated in a comparison of the main books of C++ and C. The "C programming language"[5] book, written by the authors of C, is only 286 pages and covers a very good fraction of the language, it has also remained unchanged from 1988. C is the main programming language of nearly all operating systems and there is no plan of any significant update. On the other hand, the most recent "C++ programming language"[6] book, also written by its author, has 1366 pages and its fourth edition came out in 2013! As discussed in Section 1.2 [Science and its tools], page 2, it is very important for other scientists to be able to readily read the code of a program at their will with minimum requirements.

In C++, inheriting objects in the object oriented programming paradigm and their internal functions make the code very easy to write for a programmer who is deeply invested in

[1] `https://isocpp.org/`

[2] `https://www.python.org/`

[3] `https://julialang.org/`

[4] `https://en.wikipedia.org/wiki/C_(programming_language)`

[5] Brian Kernighan, Dennis Ritchie. *The C programming language*. Prentice Hall, Inc., Second edition, 1988. It is also commonly known as K&R and is based on the ANSI C and ISO C90 standards.

[6] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley Professional; 4 edition, 2013.

those objects and understands all their relations well. But it simultaneously makes reading the program for a first time reader (a curious scientist who wants to know only how a small step was done) extremely hard. Before understanding the methods, the scientist has to invest a lot of time and energy in understanding those objects and their relations. But in C, everything is done with basic language types for example `int`s or `float`s and their pointers to define arrays. So when an outside reader is only interested in one part of the program, that part is all they have to understand.

Recently it is also becoming common to write scientific software in Python, or a combination of it with C or C++. Python is a high level scripting language which doesn't need compilation. It is very useful when you want to do something on the go and don't want to be halted by the troubles of compiling, linking, memory checking, etc. When the datasets are small and the job is temporary, this ability of Python is great and is highly encouraged. A very good example might be plotting, in which Python is undoubtedly one of the best.

But as the data sets increase in size and the processing becomes more complicated, the speed of Python scripts significantly decrease. So when the program doesn't change too often and is widely used in a large community, mostly on large data sets (like astronomical images), using Python will waste a lot of valuable research-hours. It is possible to wrap C or C++ functions with Python to fix the speed issue. But this creates further complexity, because the interested scientist has to master two programming languages and their connection (which is not trivial).

Like C++, Python is object oriented, so as explained above, it needs a high level of experience with that particular program to reasonably understand its inner workings. To make things worse, since it is mainly for on-the-go programming[7], it can undergo significant changes. One recent example is how Python 2.x and Python 3.x are not compatible. Lots of research teams that invested heavily in Python 2.x cannot benefit from Python 3.x or future versions any more. Some converters are available, but since they are automatic, lots of complications might arise in the conversion[8].

If a research project begins using Python 3.x today, there is no telling how compatible their investments will be when Python 4.x or 5.x will come out. This stems from the core principles of Python, which are very useful when you look in the 'on the go' basis as described before and not future usage. Reproducibility (ability to run the code in the future) is a core principal of any scientific result, or the software that produced that result. Rebuilding all the dependencies of a software in an obsolete language is not easy. Future-proof code (as long as current operating systems will be used) is written in C.

The portability of C is best demonstrated by the fact that both C++ and Python are part of the C-family of programming languages which also include Julia, Java, Perl, and many other languages. C libraries can be immediately included in C++, and it is easy to write wrappers for them in all C-family programming languages. This will allow other scientists to benefit from C libraries using any C-family language that they prefer. As a result, Gnuastro's library is already usable in C and C++, and wrappers will be[9] added for higher-level languages like Python, Julia and Java.

[7] Note that Python is good for fast programming, not fast programs.

[8] For example see Jenness (2017) (`https://arxiv.org/abs/1712.00461`) which describes how LSST is managing the transition.

[9] `http://savannah.gnu.org/task/?13786`

The final reason was speed. This is another very important aspect of C which is not independent of simplicity (first reason discussed above). The abstractions provided by the higher-level languages (which also makes learning them harder for a newcomer) comes at the cost of speed. Since C is a low-level language[10] (closer to the hardware), it is much less complex for both the human reader *and* the computer. The benefits of simplicity for a human were discussed above. Simplicity for the computer translates into more efficient (faster) programs. This creates a much closer relation between the scientist/programmer (or their program) and the actual data and processing. The GNU coding standards[11] also encourage the use of C over all other languages when generality of usage and "high speed" is desired.

12.2 Program design philosophy

The core processing functions of each program (and all libraries) are written mostly with the basic ISO C90 standard. We do make lots of use of the GNU additions to the C language in the GNU C library[12], but these functions are mainly used in the user interface functions (reading your inputs and preparing them prior to or after the analysis). The actual algorithms, which most scientists would be more interested in, are much more closer to ISO C90. For this reason, program source files that deal with user interface issues and those doing the actual processing are clearly separated, see Section 12.4 [Program source], page 451. If anything particular to the GNU C library is used in the processing functions, it is explained in the comments in between the code.

All the Gnuastro programs provide very low level and modular operations (modeled on GNU Coreutils). Almost all the basic command-line programs like `ls`, `cp` or `rm` on GNU/Linux operating systems are part of GNU Coreutils. This enables you to use shell scripting languages (for example GNU Bash) to operate on a large number of files or do very complex things through the creative combinations of these tools that the authors had never dreamed of. We have put a few simple examples in Chapter 2 [Tutorials], page 16.

For example all the analysis output can be saved as ASCII tables which can be fed into your favorite plotting program to inspect visually. Python's Matplotlib is very useful for fast plotting of the tables to immediately check your results. If you want to include the plots in a document, you can use the PGFplots package within LaTeX, no attempt is made to include such operations in Gnuastro. In short, Bash can act as a glue to connect the inputs and outputs of all these various Gnuastro programs (and other programs) in any fashion. Of course, Gnuastro's programs are just front-ends to the main workhorse (Section 11.3 [Gnuastro library], page 332), allowing a user to create their own programs (for example with Section 11.2 [BuildProgram], page 328). So once the functions within programs become mature enough, they will be moved within the libraries for even more general applications.

---

[10] Low-level languages are those that directly operate the hardware like assembly languages. So C is actually a high-level language, but it can be considered one of the lowest-level languages among all high-level languages.

[11] `http://www.gnu.org/prep/standards/`

[12] Gnuastro uses many GNU additions to the C library. However, thanks to the GNU Portability library (Gnulib) which is included in the Gnuastro tarball, users of non-GNU/Linux operating systems can also benefit from all these features when using Gnuastro.

The advantage of this architecture is that the programs become small and transparent: the starting and finishing point of every program is clearly demarcated. For nearly all operations on a modern computer (fast file input-output) with a modest level of complexity, the read/write speed is insignificant compared to the actual processing a program does. Therefore the complexity which arises from sharing memory in a large application is simply not worth the speed gain. Gnuastro's design is heavily influenced from Eric Raymond's "The Art of Unix Programming"[13] which beautifully describes the design philosophy and practice which lead to the success of Unix-based operating systems[14].

12.3 Coding conventions

In Gnuastro, we try our best to follow the GNU coding standards. Added to those, Gnuastro defines the following conventions. It is very important for readability that the whole package follows the same convention.

- The code must be easy to read by eye. So when the order of several lines within a function does not matter (for example when defining variables at the start of a function). You should put the lines in the order of increasing length and group the variables with similar types such that this half-pyramid of declarations becomes most visible. If the reader is interested, a simple search will show them the variable they are interested in. However, this visual aid greatly helps in general inspections of the code and help the reader get a grip of the function's processing.
- A function that cannot be fully displayed (vertically) in your monitor is probably too long and may be more useful if it is broken up into multiple functions. 40 lines is usually a good reference. When the start and end of a function are clearly visible in one glance, the function is much more easier to understand. This is most important for low-level functions (which usually define a lot of variables). Low-level functions do most of the processing, they will also be the most interesting part of a program for an inquiring astronomer. This convention is less important for higher level functions that don't define too many variables and whose only purpose is to run the lower-level functions in a specific order and with checks.

  In general you can be very liberal in breaking up the functions into smaller parts, the GNU Compiler Collection (GCC) will automatically compile the functions as inline functions when the optimizations are turned on. So you don't have to worry about decreasing the speed. By default Gnuastro will compile with the `-O3` optimization flag.
- All Gnuastro hand-written text files (C source code, Texinfo documentation source, and version control commit messages) should not exceed **75** characters per line. Monitors today are certainly much wider, but with this limit, reading the functions becomes much more easier. Also for the developers, it allows multiple files (or multiple views of one file) to be displayed beside each other on wide monitors.

  Emacs's buffers are excellent for this capability, setting a buffer width of 80 with '`C-u 80 C-x 3`' will allow you to view and work on several files or different parts of one file using the wide monitors common today. Emacs buffers can also be used as a shell prompt and compile the program (with `M-x compile`), and 80 characters is the default

[13] Eric S. Raymond, 2004, *The Art of Unix Programming*, Addison-Wesley Professional Computing Series.
[14] KISS principle: Keep It Simple, Stupid!

width in most terminal emulators. If you use Emacs, Gnuastro sets the 75 character `fill-column` variable automatically for you, see cartouche below.

For long comments you can use press `Alt-q` in Emacs to separate them into separate lines automatically. For long literal strings, you can use the fact that in C, two strings immediately after each other are concatenated, for example `"The first part, " "and the second part."`. Note the space character in the end of the first part. Since they are now separated, you can easily break a long literal string into several lines and adhere to the maximum 75 character line length policy.

- The headers required by each source file (ending with `.c`) should be defined inside of it. All the headers a complete program needs should *not* be stacked in another header to include in all source files (for example `main.h`). Although most 'professional' programmers choose this single header method, Gnuastro is primarily written for professional/inquisitive astronomers (who are generally amateur programmers). The list of header files included provides valuable general information and helps the reader. `main.h` may only include the header file(s) that define types that the main program structure needs, see `main.h` in Section 12.4 [Program source], page 451. Those particular header files that are included in `main.h` can of course be ignored (not included) in separate source files.
- The headers should be classified (by an empty line) into separate groups:
  1. `#include <config.h>`: This must be the first code line (not commented or blank) in each source file *within Gnuastro*. It sets macros that the GNU Portability Library (Gnulib) will use for a unified environment (GNU C Library), even when the user is building on a system that doesn't use the GNU C library.
  2. The C library header files, for example `stdio.h`, `stdlib.h`, or `math.h`.
  3. Installed library header files, including Gnuastro's installed headers (for example `cfitsio.h` or `gsl/gsl_rng.h`, or `gnuastro/fits.h`).
  4. Gnuastro's internal headers (that are not installed), for example `gnuastro-internal/options.h`.
  5. For programs, the `main.h` file (which is needed by the next group of headers).
  6. That particular program's header files, for example `mkprof.h`, or `noisechisel.h`.

  As much as order does not matter when you include the header of each group, sort them by length, as described above.
- All function names, variables, etc should be in lower case. Macros and constant global `enum`s should be in upper case.
- For the naming of exported header files, functions, variables, macros, and library functions, we adopt similar conventions to those used by the GNU Scientific Library (GSL)[15]. In particular, in order to avoid clashes with the names of functions and variables coming from other libraries the name-space '`gal_`' is prefixed to them. GAL stands for *G*NU *A*stronomy *L*ibrary.
- All installed header files should be in the `lib/gnuastro` directory (under the top Gnuastro source directory). After installation, they will be put in the `$prefix/include/gnuastro` directory (see Section 3.3.1.2 [Installation directory],

[15] `https://www.gnu.org/software/gsl/design/gsl-design.html#SEC15`

page 79, for `$prefix`). Therefore with this convention Gnuastro's headers can be included in internal (to Gnuastro) and external (a library user) source files with the same line

```
# include <gnuastro/headername.h>
```

Note that the GSL convention for header file names is `gsl_specialname.h`, so your include directive for a GSL header must be something like `#include <gsl/gsl_specialname.h>`. Gnuastro doesn't follow this GSL guideline because of the repeated `gsl` in the include directive. It can be confusing and cause bugs for beginners. All Gnuastro (and GSL) headers must be located within a unique directory and will not be mixed with other headers. Therefore the '`gsl_`' prefix to the header file names is redundant[16].

- All installed functions and variables should also include the base-name of the file in which they are defined as prefix, using underscores to separate words[17]. The same applies to exported macros, but in upper case. For example in Gnuastro's top source directory, the prototype of function `gal_box_border_from_center` is in `lib/gnuastro/box.h`, and the macro `GAL_POLYGON_MAX_CORNERS` is defined in `lib/gnuastro/polygon.h`.

  This is necessary to give any user (who is not familiar with the library structure) the ability to follow the code. This convention does make the function names longer (a little harder to write), but the extra documentation it provides plays an important role in Gnuastro and is worth the cost.
- There should be no trailing white space in a line. To do this automatically every time you save a file in Emacs, add the following line to your `~/.emacs` file.

  ```
  (add-hook 'before-save-hook 'delete-trailing-whitespace)
  ```
- There should be no tabs in the indentation[18].
- Individual, contextually similar, functions in a source file are separated by 5 blank lines to be easily seen to be related in a group when parsing the source code by eye. In Emacs you can use `CTRL-u 5 CTRL-o`.
- One group of contextually similar functions in a source file is separated from another with 20 blank lines. In Emacs you can use `CTRL-u 20 CTRL-o`. Each group of functions has short descriptive title of the functions in that group. This title is surrounded by asterisks (`*`) to make it clearly distinguishable. Such contextual grouping and clear title are very important for easily understanding the code.
- Always read the comments before the patch of code under it. Similarly, try to add as many comments as you can regarding every patch of code. Effectively, we want someone to get a good feeling of the steps, without having to read the C code and only by reading the comments. This follows similar principles as Literate programming (`https://en.wikipedia.org/wiki/Literate_programming`).

[16] For GSL, this prefix has an internal technical application: GSL's architecture mixes installed and not-installed headers in the same directory. This prefix is used to identify their installation status. Therefore this filename prefix in GSL a technical internal issue (for developers, not users).

[17] The convention to use underscores to separate words, called "snake case" (or "snake_case"). This is also recommended by the GNU coding standards.

[18] If you use Emacs, Gnuastro's `.dir-locals.el` file will automatically never use tabs for indentation. To make this a default in all your Emacs sessions, you can add the following line to your `~/.emacs` file: `(setq-default indent-tabs-mode nil)`

The last two conventions are not common and might benefit from a short discussion here. With a good experience in advanced text editor operations, the last two are redundant for a professional developer. However, recall that Gnuastro aspires to be friendly to unfamiliar, and inexperienced (in programming) eyes. In other words, as discussed in Section 1.2 [Science and its tools], page 2, we want the code to appear welcoming to someone who is completely new to coding (and text editors) and only has a scientific curiosity.

Newcomers to coding and development, who are curious enough to venture into the code, will probably not be using (or have any knowledge of) advanced text editors. They will see the raw code in the webpage or on a simple text editor (like Gedit) as plain text. Trying to learn and understand a file with dense functions that are all spaced with one or two blank lines can be very taunting for a newcomer. But when they scroll through the file and see clear titles and meaningful spaces for similar functions, we are helping them find and focus on the part they are most interested in sooner and easier.

GNU Emacs, the recommended text editor: GNU Emacs is an extensible and easily customizable text editor which many programmers rely on for developing due to its countless features. Among them, it allows specification of certain settings that are applied to a single file or to all files in a directory and its sub-directories. In order to harmonize code coming from different contributors, Gnuastro comes with a `.dir-locals.el` file which automatically configures Emacs to satisfy most of the coding conventions above when you are using it within Gnuastro's directories. Thus, Emacs users can readily start hacking into Gnuastro. If you are new to developing, we strongly recommend this editor. Emacs was the first project released by GNU and is still one of its flagship projects. Some resources can be found at:

Official manual
: At `https://www.gnu.org/software/emacs/manual/emacs.html`. This is a great and very complete manual which is being improved for over 30 years and is the best starting point to learn it. It just requires a little patience and practice, but rest assured that you will be rewarded. If you install Emacs, you also have access to this manual on the command-line with the following command (see Section 4.3.4 [Info], page 111).

```
$ info emacs
```

A guided tour of emacs
: At `https://www.gnu.org/software/emacs/tour/`. A short visual tour of Emacs, officially maintained by the Emacs developers.

Unofficial mini-manual
: At `https://tuhdo.github.io/emacs-tutor.html`. A shorter manual which contains nice animated images of using Emacs.

12.4 Program source

Besides the fact that all the programs share some functions that were explained in Chapter 11 [Library], page 320, everything else about each program is completely independent. Recall that Gnuastro is written for an active astronomer/scientist (not a

passive one who just uses a software). It must thus be easily navigable. Hence there are fixed source files (that contain fixed operations) that must be present in all programs, these are discussed fully in Section 12.4.1 [Mandatory source code files], page 452. To easily understand the explanations in this section you can use Section 12.4.2 [The TEMPLATE program], page 454, which contains the bare minimum code for one working program. This template can also be used to easily add new utilities: just copy and paste the directory and change `TEMPLATE` with your program's name.

12.4.1 Mandatory source code files

Some programs might need lots of source files and if there is no fixed convention, navigating them can become very hard for a new inquirer into the code. The following source files exist in every program's source directory (which is located in `bin/progname`). For small programs, these files are enough. Larger programs will need more files and developers are encouraged to define any number of new files. It is just important that the following list of files exist and do what is described here. When creating other source files, please choose filenames that are a complete single word: don't abbreviate (abbreviations are cryptic). For a minimal program containing all these files, see Section 12.4.2 [The TEMPLATE program], page 454.

`main.c` Each executable has a `main` function, which is located in `main.c`. Therefore this file is the starting point when reading any program's source code. No actual processing functions must be defined in this file, the function(s) in this file are only meant to connect the most high level steps of each program. Generally, `main` will first call the top user interface function to read user input and make all the preparations. Then it will pass control to the top processing function for that program. The functions to do both these jobs must be defined in other source files.

`main.h` All the major parameters which will be used in the program must be stored in a structure which is defined in `main.h`. The name of this structure is usually `prognameparams`, for example `cropparams` or `noisechiselparams`. So `#include "main.h"` will be a staple in all the source codes of the program. It is also regularly the first (and only) argument most of the program's functions which greatly helps in readability.

Keeping all the major parameters of a program in this structure has the major benefit that most functions will only need one argument: a pointer to this structure. This will significantly facilitate the job of the programmer, the inquirer and the computer. All the programs in Gnuastro are designed to be low-level, small and independent parts, so this structure should not get too large.

The main root structure of all programs contains at least one instance of the `gal_options_common_params` structure. This structure will keep the values to all common options in Gnuastro's programs (see Section 4.1.2 [Common options], page 95). This top root structure is conveniently called `p` (short for parameters) by all the functions in the programs and the common options parameters within it are called `cp`. With this convention any reader can immediately understand where to look for the definition of one parameter. For example you know that `p->cp->output` is in the common parameters while `p->threshold` is in the program's parameters.

With this basic root structure, source code of functions can potentially become full of structure de-reference operators (`->`) which can make the code very unreadable. In order to avoid this, whenever a structure element is used more than a couple of times in a function, a variable of the same type and with the same name (so it can be searched) as the desired structure element should be defined with the value of the root structure inside of it in definition time. Here is an example.

```
char *hdu=p->cp.hdu;
float threshold=p->threshold;
```

`args.h` The options particular to each program are defined in this file. Each option is defined by a block of parameters in `program_options`. These blocks are all you should modify in this file, leave the bottom group of definitions untouched. These are fed directly into the GNU C library's Argp facilities and it is recommended to have a look at that for better understand what is going on, although this is not required here.

Each element of the block defining an option is described under `argp_option` in `bootstrapped/lib/argp.h` (from Gnuastro's top source file). Note that the last few elements of this structure are Gnuastro additions (not documented in the standard Argp manual). The values to these last elements are defined in `lib/gnuastro/type.h` and `lib/gnuastro-internal/options.h` (from Gnuastro's top source directory).

`ui.h` Besides declaring the exported functions of `ui.c`, this header also keeps the "key"s to every program-specific option. The first class of keys for the options that have a short-option version (single letter, see Section 4.1.1.2 [Options], page 93). The character that is defined here is the option's short option name. The list of available alphabet characters can be seen in the comments. Recall that some common options also take some characters, for those, see `lib/gnuastro-internal/options.h`.

The second group of options are those that don't have a short option alternative. Only the first in this group needs a value (`1000`), the rest will be given a value by C's `enum` definition, so the actual value is irrelevant and must never be used, always use the name.

`ui.c` Everything related to reading the user input arguments and options, checking the configuration files and checking the consistency of the input parameters before the actual processing is run should be done in this file. Since most functions are the same, with only the internal checks and structure parameters differing. We recommend going through the `ui.c` of Section 12.4.2 [The TEMPLATE program], page 454, or several other programs for a better understanding.

The most high-level function in `ui.c` is named `ui_read_check_inputs_setup`. It accepts the raw command-line inputs and a pointer to the root structure for that program (see the explanation for `main.h`). This is the function that `main` calls. The basic idea of the functions in this file is that the processing functions should need a minimum number of such checks. With this convention an inquirer who only wants to understand only one part (mostly the processing part and not user input details and sanity checks) of the code can easily do so

in the later files. It also makes all the errors related to input appear before the processing begins which is more convenient for the user.

`progname.c, progname.h`
: The high-level processing functions in each program are in a file named `progname.c`, for example `crop.c` or `noisechisel.c`. The function within these files which `main` calls is also named after the program, for example

```
void
crop(struct cropparams *p)
```

or

```
void
noisechisel(struct noisechiselparams *p)
```

In this manner, if an inquirer is interested the processing steps, they can immediately come and check this file for the first processing step without having to go through `main.c` and `ui.c` first. In most situations, any failure in any step of the programs will result in an informative error message and an immediate abort in the program. So there is usually no need for return values. Under more complicated situations where a return value might be necessary, `void` will be replaced with an `int` in the examples above. This value must be directly returned by `main`, so it has to be an `int`.

`authors-cite.h`
: This header file keeps the global variable for the program authors and its BibTeX record for citation. They are used in the outputs of the common options `--version` and `--cite`, see Section 4.1.2.3 [Operating mode options], page 100.

12.4.2 The TEMPLATE program

The extra creativity offered by libraries comes at a cost: you have to actually write your `main` function and get your hands dirty in managing user inputs: are all the necessary parameters given a value? is the input in the correct format? do the options and the inputs correspond? and many other similar checks. So when an operation has well-defined inputs and outputs and is commonly needed, it is much more worthwhile to simply do use all the great features that Gnuastro has already defined for such operations.

To make it easier to learn/apply the internal program infra-structure discussed in Section 12.4.1 [Mandatory source code files], page 452, in the Section 3.2.2 [Version controlled source], page 72, Gnuastro ships with a template program . This template program is not available in the Gnuastro tarball so it doesn't confuse people using the tarball. The `bin/TEMPLATE` directory in Gnuastro's Git repository contains the bare-minimum files necessary to define a new program and all the basic/necessary files/functions are pre-defined there.

Below you can see a list of initial steps to take for customizing this template. We just assume that after cloning Gnuastro's history, you have already bootstrapped Gnuastro, if not, please see Section 3.2.2.1 [Bootstrapping], page 73.

1. Select a name for your new program (for example `myprog`).
2. Copy the `TEMPLATE` directory to a directory with your program's name:

```
$ cp -R bin/TEMPLATE bin/myprog
```

3. As with all source files in Gnuastro, all the files in template also have a copyright notice at their top. Open all the files and correct these notices: 1) The first line contains a single-line description of the program. 2) In the second line only the name or your program needs to be fixed and 3) Add your name and email as a "Contributing author". As your program grows, you will need to add new files, don't forget to add this notice in those new files too, just put your name and email under "Original author" and correct the copyright years.
4. Open `configure.ac` in the top Gnuastro source. This file manages the operations that are done when a user runs `./configure`. Going down the file, you will notice repetitive parts for each program. You will notice that the program names follow an alphabetic ordering in each part. There is also a commented line/patch for the `TEMPLATE` program in each part. You can copy one line/patch (from the program above or below your desired name for example) and paste it in the proper place for your new program. Then correct the names of the copied program to your new program name. There are multiple places where this has to be done, so be patient and go down to the bottom of the file. Ultimately add `bin/myprog/Makefile` to `AC_CONFIG_FILES`, only here the ordering depends on the length of the name (it isn't alphabetical).
5. Open `Makefile.am` in the top Gnuastro source. Similar to the previous step, add your new program similar to all the other programs. Here there are only two places: 1) at the top where we define the conditionals (three lines per program), and 2) immediately under it as part of the value for `SUBDIRS`.
6. Open `doc/Makefile.am` and similar to `Makefile.am` (above), add the proper entries for the man-page of your program to be created (here, the variable that keeps all the man-pages to be created is `dist_man_MANS`). Then scroll down and add a rule to build the man-page similar to the other existing rules (in alphabetical order). Don't forget to add a short one-line description here, it will be displayed on top of the man-page.
7. Change `TEMPLATE.c` and `TEMPLATE.h` to `myprog.c` and `myprog.h` in the file names:

```
$ cd bin/myprog
$ mv TEMPLATE.c myprog.c
$ mv TEMPLATE.h myprog.h
```

8. Correct all occurrences of `TEMPLATE` in the input files to `myprog` (in short or long format). You can get a list of all occurrences with the following command. If you use Emacs, it will be able to parse the Grep output and open the proper file and line automatically. So this step can be very easy.

```
$ grep --color -nHi -e template *
```

9. Run the following commands to re-build the configuration and build system, and then to configure and build Gnuastro (which now includes your exciting new program).

```
$ autoreconf -f
$ ./configure
$ make
```

10. You are done! You can now start customizing your new program to do your special processing. When its complete, just don't forget to add checks also, so it can be tested at least once on a user's system with `make check`, see Section 12.7 [Test scripts], page 458. Finally, if you would like to share it with all Gnuastro users, inform us so we merge it into Gnuastro's main history.

12.5 Documentation

Documentation (this book) is an integral part of Gnuastro (see Section 1.2 [Science and its tools], page 2). Documentation is not considered a separate project and must be written by its developers. Users can make edits/corrections, but the initial writing must be by the developer. So, no change is considered valid for implementation unless the respective parts of the book have also been updated. The following procedure can be a good suggestion to take when you have a new idea and are about to start implementing it.

The steps below are not a requirement, the important thing is that when you send your work to be included in Gnuastro, the book and the code have to both be fully up-to-date and compatible, with the purpose of the update very clearly explained. You can follow any strategy you like, the following strategy was what we have found to be most useful until now.

1. Edit the book and fully explain your desired change, such that your idea is completely embedded in the general context of the book with no sense of discontinuity for a first time reader. This will allow you to plan the idea much more accurately and in the general context of Gnuastro (a particular program or library). Later on, when you are coding, this general context will significantly help you as a road-map.

   A very important part of this process is the program/library introduction. These first few paragraphs explain the purposes of the program or library and are fundamental to Gnuastro. Before actually starting to code, explain your idea's purpose thoroughly in the start of the respective/new section you wish to work on. While actually writing its purpose for a new reader, you will probably get some valuable and interesting ideas that you hadn't thought of before. This has occurred several times during the creation of Gnuastro.

   If an introduction already exists, embed or blend your idea's purpose with the existing introduction. We emphasize that doing this is equally useful for you (as the programmer) as it is useful for the user (reader). Recall that the purpose of a program is very important, see Section 12.2 [Program design philosophy], page 447.

   As you have already noticed for every program/library, it is very important that the basics of the science and technique be explained in separate subsections prior to the 'Invoking Programname' subsection. If you are writing a new program or your addition to an existing program involves a new concept, also include such subsections and explain the concepts so a person completely unfamiliar with the concepts can get a general initial understanding. You don't have to go deep into the details, just enough to get an interested person (with absolutely no background) started with some good pointers/links to where they can continue studying if they are more interested. If you feel you can't do that, then you have probably not understood the concept yourself. If you feel you don't have the time, then think about yourself as the reader in one year: you will forget almost all the details, so now that you have done all the theoretical preparations, add a few more hours and document it. Therefore in one year, when you find a bug or want to add a new feature, you don't have to prepare as much. Have in mind that your only limitation in length is the fatigue of the reader after reading a long text, nothing else. So as long as you keep it relevant/interesting for the reader, there is no page number limit/cost.

It might also help if you start discussing the usage of your idea in the 'Invoking ProgramName' subsection (explaining the options and arguments you have in mind) at this stage too. Actually starting to write it here will really help you later when you are coding.

2. After you have finished adding your initial intended plan to the book, then start coding your change or new program within the Gnuastro source files. While you are coding, you will notice that somethings should be different from what you wrote in the book (your initial plan). So correct them as you are actually coding, but don't worry too much about missing a few things (see the next step).
3. After your work has been fully implemented, read the section documentation from the start and see if you didn't miss any change in the coding and to see if the context is fairly continuous for a first time reader (who hasn't seen the book or had known Gnuastro before you made your change).
4. If the change is notable, also update the `NEWS` file.

12.6 Building and debugging

To build the various programs and libraries in Gnuastro, the GNU build system is used which defines the steps in Section 1.1 [Quick start], page 1. It consists of GNU Autoconf, GNU Automake and GNU Libtool which are collectively known as GNU Autotools. They provide a very portable system to check the hosts environment and compile Gnuastro based on that. They also make installing everything in their standard places very easy for the programmer. Most of the small caps files that you see in the top source directory of the tarball are created by these three tools (see Section 3.2.2 [Version controlled source], page 72). To facilitate the building and testing of your work during development, Gnuastro comes with two useful scripts:

`developer-build`
: This is more fully described in Section 3.3.1.4 [Configure and build in RAM], page 84. During development, you will usually run this command only once (at the start of your work).

`tests/during-dev.sh`
: This script is designed to be run each time you make a change and want to test your work (with some possible input and output). The script itself is heavily commented and thoroughly describes the best way to use it, so we won't repeat it here.

  As a short summary: you specify the build directory, an output directory (for the built program to be run in, and also contains the inputs), the program's short name and the arguments and options that it should be run with. This script will then build Gnuastro, go to the output directory and run the built executable from there. One option for the output directory might be your desktop, so you can easily see the output files and delete them when you are finished. The main purpose of these scripts is to keep your source directory clean and facilitate your development.

By default all the programs are compiled with optimization flags for increased speed. A side effect of optimization is that valuable debugging information is lost. All the libraries

are also linked as shared libraries by default. Shared libraries further complicate the debugging process and significantly slow down the compilation (the `make` command). So during development it is recommended to configure Gnuastro as follows:

```
$ ./configure --enable-debug
```

In `developer-build` you can ask for this behavior through the `--debug` option, see Section 3.3.2 [Separate build and source directories], page 85.

In order to understand the building process, you can go through the Autoconf, Automake and Libtool manuals, like all GNU manuals they provide both a great tutorial and technical documentation. The "A small Hello World" section in Automake's manual (in chapter 2) can be a good starting guide after you have read the separate introductions.

12.7 Test scripts

As explained in Section 3.3.3 [Tests], page 88, for every program some simple tests are written to check the various independent features of the program. All the tests are placed in the `tests/` directory. The `tests/prepconf.sh` script is the first 'test' that will be run. It will copy all the configuration files from the various directories to a `tests/.gnuastro` directory (which it will make) so the various tests can set the default values. This script will also make sure the programs don't go searching for user and system wide configuration files to avoid the mixing of values with different Gnuastro version on the system.

For each program, the tests are placed inside directories with the program name. Each test is written as a shell script. The last line of this script is the test which runs the program with certain parameters. The return value of this script determines the fate of the test, see the "Support for test suites" chapter of the Automake manual for a very nice and complete explanation. In every script, two variables are defined at first: `prog` and `execname`. The first specifies the program name and the second the location of the executable.

The most important thing to have in mind about all the test scripts is that they are run from inside the `tests/` directory in the "build tree". Which can be different from the directory they are stored in (known as the "source tree")[19]. This distinction is made by GNU Autoconf and Automake (which configure, build and install Gnuastro) so that you can install the program even if you don't have write access to the directory keeping the source files. See the "Parallel build trees (a.k.a VPATH builds)" in the Automake manual for a nice explanation.

Because of this, any necessary inputs that are distributed in the tarball[20], for example the catalogs necessary for checks in MakeProfiles and Crop, must be identified with the `$topsrc` prefix instead of `../` (for the top source directory that is unpacked). This `$topsrc` variable points to the source tree where the script can find the source data (it is defined in `tests/Makefile.am`). The executables and other test products were built in the build tree (where they are being run), so they don't need to be prefixed with that variable. This is also true for images or files that were produced by other tests.

[19] The `developer-build` script also uses this feature to keep the source and build directories separate (see Section 3.3.2 [Separate build and source directories], page 85).

[20] In many cases, the inputs of a test are outputs of previous tests, this doesn't apply to this class of inputs. Because all outputs of previous tests are in the "build tree".

12.8 Developer's checklist

This is a checklist of things to do after applying your changes/additions in Gnuastro:

1. If the change is non-trivial, write test(s) in the `tests/progname/` directory to test the change(s)/addition(s) you have made. Then add their file names to `tests/Makefile.am`.
2. Run `$ make check` to make sure everything is working correctly.
3. Make sure the documentation (this book) is completely up to date with your changes, see Section 12.5 [Documentation], page 456.
4. Commit the change to your issue branch (see Section 12.11.3 [Production workflow], page 464, and Section 12.11.4 [Forking tutorial], page 465). Afterwards, run Autoreconf to generate the appropriate version number:

   ```
   $ autoreconf -f
   ```

5. Finally, to make sure everything will be built, installed and checked correctly run the following command (after re-configuring, and re-building). To greatly speed up the process, use multiple threads (8 in the example below, change it appropriately)

   ```
   $ make distcheck -j8
   ```

   This command will create a distribution file (ending with `.tar.gz`) and try to compile it in the most general cases, then it will run the tests on what it has built in its own mini-environment. If `$ make distcheck` finishes successfully, then you are safe to send your changes to us to implement or for your own purposes. See Section 12.11.3 [Production workflow], page 464, and Section 12.11.4 [Forking tutorial], page 465.

12.9 Gnuastro project webpage

Gnuastro's central management hub (`https://savannah.gnu.org/projects/gnuastro/`)[21] is located on GNU Savannah (`https://savannah.gnu.org/`)[22]. Savannah is the central software development management system for many GNU projects. Through this central hub, you can view the list of activities that the developers are engaged in, their activity on the version controlled source, and other things. Each defined activity in the development cycle is known as an 'issue' (or 'item'). An issue can be a bug (see Section 1.7 [Report a bug], page 11), or a suggested feature (see Section 1.8 [Suggest new feature], page 12) or an enhancement or generally any *one* job that is to be done. In Savannah, issues are classified into three categories or 'tracker's:

Support
: This tracker is a way that (possibly anonymous) users can get in touch with the Gnuastro developers. It is a complement to the bug-gnuastro mailing list (see Section 1.7 [Report a bug], page 11). Anyone can post an issue to this tracker. The developers will not submit an issue to this list. They will only reassign the issues in this list to the other two trackers if they are valid[23]. Ideally (when the developers have time to put on Gnuastro, please don't forget that Gnuastro is a volunteer effort), there should be no open items in this tracker.

[21] `https://savannah.gnu.org/projects/gnuastro/`

[22] `https://savannah.gnu.org/`

[23] Some of the issues registered here might be due to a mistake on the user's side, not an actual bug in the program.

Bugs This tracker contains all the known bugs in Gnuastro (problems with the existing tools).

Tasks The items in this tracker contain the future plans (or new features/capabilities) that are to be added to Gnuastro.

All the trackers can be browsed by a (possibly anonymous) visitor, but to edit and comment on the Bugs and Tasks trackers, you have to be a registered on Savannah. When posting an issue to a tracker, it is very important to choose the 'Category' and 'Item Group' options accurately. The first contains a list of all Gnuastro's programs along with 'Installation', 'New program' and 'Webpage'. The "Item Group" contains the nature of the issue, for example if it is a 'Crash' in the software (a bug), or a problem in the documentation (also a bug) or a feature request or an enhancement.

The set of horizontal links on the top of the page (Starting with 'Main' and 'Homepage' and finishing with 'News') are the easiest way to access these trackers (and other major aspects of the project) from any part of the project webpage. Hovering your mouse over them will open a drop down menu that will link you to the different things you can do on each tracker (for example, 'Submit new' or 'Browse'). When you browse each tracker, you can use the "Display Criteria" link above the list to limit the displayed issues to what you are interested in. The 'Category' and 'Group Item' (explained above) are a good starting point.

Any new issue that is submitted to any of the trackers, or any comments that are posted for an issue, is directly forwarded to the gnuastro-devel mailing list (`https://lists.gnu.org/mailman/listinfo/gnuastro-devel`, see Section 12.10 [Developing mailing lists], page 460, for more). This will allow anyone interested to be up to date on the over-all development activity in Gnuastro and will also provide an alternative (to Savannah) archiving for the development discussions. Therefore, it is not recommended to directly post an email to this mailing list, but do all the activities (for example add new issues, or comment on existing ones) on Savannah.

Do I need to be a member in Savannah to contribute to Gnuastro? No.

The full version controlled history of Gnuastro is available for anonymous download or cloning. See Section 12.11.3 [Production workflow], page 464, for a description of Gnuastro's Integration-Manager Workflow. In short, you can either send in patches, or make your own fork. If you choose the latter, you can push your changes to your own fork and inform us. We will then pull your changes and merge them into the main project. Please see Section 12.11.4 [Forking tutorial], page 465, for a tutorial.

12.10 Developing mailing lists

To keep the developers and interested users up to date with the activity and discussions within Gnuastro, there are two mailing lists which you can subscribe to:

`gnuastro-devel@gnu.org`
(at `https://lists.gnu.org/mailman/listinfo/gnuastro-devel`)
All the posts made in the support, bugs and tasks discussions of Section 12.9 [Gnuastro project webpage], page 459, are also sent to this mailing address and

archived. By subscribing to this list you can stay up to date with the discussions that are going on between the developers before, during and (possibly) after working on an issue. All discussions are either in the context of bugs or tasks which are done on Savannah and circulated to all interested people through this mailing list. Therefore it is not recommended to post anything directly to this mailing list. Any mail that is sent to it from Savannah to this list has a link under the title "Reply to this item at:". That link will take you directly to the issue discussion page, where you can read the discussion history or join it.

While you are posting comments on the Savannah issues, be sure to update the meta-data. For example if the task/bug is not assigned to anyone and you would like to take it, change the "Assigned to" box, or if you want to report that it has been applied, change the status and so on. All these changes will also be circulated with the email very clearly.

`gnuastro-commits@gnu.org`
(at `https://lists.gnu.org/mailman/listinfo/gnuastro-commits`)
This mailing list is defined to circulate all commits that are done in Gnuastro's version controlled source, see Section 3.2.2 [Version controlled source], page 72. If you have any ideas, or suggestions on the commits, please use the bug and task trackers on Savannah to followup the discussion, do not post to this list. All the commits that are made for an already defined issue or task will state the respective ID so you can find it easily.

12.11 Contributing to Gnuastro

You have this great idea or have found a good fix to a problem which you would like to implement in Gnuastro. You have also become familiar with the general design of Gnuastro in the previous sections of this chapter (see Chapter 12 [Developing], page 445) and want to start working on and sharing your new addition/change with the whole community as part of the official release. This is great and your contribution is most welcome. This section and the next (see Section 12.8 [Developer's checklist], page 459) are written in the hope of making it as easy as possible for you to share your great idea with the community.

In this section we discuss the final steps you have to take: legal and technical. From the legal perspective, the copyright of any work you do on Gnuastro has to be assigned to the Free Software Foundation (FSF) and the GNU operating system, or you have to sign a disclaimer. We do this to ensure that Gnuastro can remain free in the future, see Section 12.11.1 [Copyright assignment], page 462. From the technical point of view, in this section we also discuss commit guidelines (Section 12.11.2 [Commit guidelines], page 463) and the general version control workflow of Gnuastro in Section 12.11.3 [Production workflow], page 464, along with a tutorial in Section 12.11.4 [Forking tutorial], page 465.

Recall that before starting the work on your idea, be sure to checkout the bugs and tasks trackers in Section 12.9 [Gnuastro project webpage], page 459, and announce your work there so you don't end up spending time on something others have already worked on, and also to attract similarly interested developers to help you.

12.11.1 Copyright assignment

Gnuastro's copyright is owned by the FSF. Professor Eben Moglen, of the Columbia University Law School has given a nice summary of the reasons for this at `https://www.gnu.org/licenses/why-assign`. Below we are copying it verbatim for self consistency (in case you are offline or reading in print).

> Under US copyright law, which is the law under which most free software programs have historically been first published, there are very substantial procedural advantages to registration of copyright. And despite the broad right of distribution conveyed by the GPL, enforcement of copyright is generally not possible for distributors: only the copyright holder or someone having assignment of the copyright can enforce the license. If there are multiple authors of a copyrighted work, successful enforcement depends on having the cooperation of all authors.
>
> In order to make sure that all of our copyrights can meet the record keeping and other requirements of registration, and in order to be able to enforce the GPL most effectively, FSF requires that each author of code incorporated in FSF projects provide a copyright assignment, and, where appropriate, a disclaimer of any work-for-hire ownership claims by the programmer's employer. That way we can be sure that all the code in FSF projects is free code, whose freedom we can most effectively protect, and therefore on which other developers can completely rely.

Please get in touch with the Gnuastro maintainer (currently Mohammad Akhlaghi, mohammad -at- akhlaghi -dot- org) to follow the procedures. It is possible to do this for each change (good for for a single contribution), and also more generally for all the changes/additions you do in the future within Gnuastro. So if you have already assigned the copyright of your work on another GNU software to the FSF, it should be done again for Gnuastro. The FSF has staff working on these legal issues and the maintainer will get you in touch with them to do the paperwork. The maintainer will just be informed in the end so your contributions can be merged within the Gnuastro source code.

Gnuastro will gratefully acknowledge (see Section 1.11 [Acknowledgments], page 14) all the people who have assigned their copyright to the FSF and have thus helped to guarantee the freedom and reliability of Gnuastro. The Free Software Foundation will also acknowledge your copyright contributions in the Free Software Supporter: `https://www.fsf.org/free-software-supporter` which will circulate to a very large community (104,444 people in April 2016). See the archives for some examples and subscribe to receive interesting updates. The very active code contributors (or developers) will also be recognized as project members on the Gnuastro project webpage (see Section 12.9 [Gnuastro project webpage], page 459) and can be given a `gnu.org` email address. So your very valuable contribution and copyright assignment will not be forgotten and is highly appreciated by a very large community. If you are reluctant to sign an assignment, a disclaimer is also acceptable.

Do I need a disclaimer from my university or employer? It depends on the contract with your university or employer. From the FSF's `/gd/gnuorg/conditions.text`: "If you are employed to do programming, or have made an agreement with your employer that says it owns programs you write, we need a signed piece of paper from your employer disclaiming rights to" Gnuastro. The FSF's copyright clerk will kindly help you decide, please consult the following email address: "assign -at- gnu -dot- org".

12.11.2 Commit guidelines

To be able to cleanly integrate your work with the other developers, **never commit on the `master` branch** (see Section 12.11.3 [Production workflow], page 464, for a complete discussion and Section 12.11.4 [Forking tutorial], page 465, for a cookbook example). In short, leave `master` only for changes you fetch, or pull from the official repository (see Section 3.2.2.2 [Synchronizing], page 75).

In the Gnuastro commit messages, we strive to follow these standards. Note that in the early phases of Gnuastro's development, we are experimenting and so if you notice earlier commits don't satisfy some of the guidelines below, it is because they predate that guideline.

Commit title

The commits have to start with one short descriptive title. The title is separated from the body with one blank line. Run `git log` to see some of the most recent commit messages as an example. In general, the title should satisfy the following conditions:

- It is best for the title to be short, about 60 (or even 50) characters. Most emulated command-line terminals are about 80 characters wide. However, we should also allow for the commit hashes which are printed in `git log --oneline`, and also branch names or the graph structure outputs of `git log` which are also commonly used.
- The title should not finish with any full-stops or periods ('`.`').

Commit body

The body of the commit message is separated from the title by one empty line. Recall that anyone who has subscribed to `gnuastro-commits` mailing list will get the commit in their email after it has been pushed to `master`. People will also read them when they synchronize with the main Gnuastro repository (see Section 3.2.2.2 [Synchronizing], page 75). Finally, the commit messages will later be used to update the `NEWS` file on each release. Therefore the commit message body plays a very important role in the development of Gnuastro, so please adhere to the following guidelines.

- The body should be very descriptive. Start the commit message body by explaining what changes your commit makes from a user's perspective (added, changed, or removed options, or arguments to programs or libraries, or modified algorithms, or new installation step, or etc).
- Try to explain the committed contents as best as you can. Recall that the readers of your commit message do not necessarily have your current background. After some time you will also forget the context, so this request is not just for others[24]. Therefore be very descriptive and explain

[24] `http://catb.org/esr/writings/unix-koans/prodigy.html`

as much as possible: what the bug/task was, justify the way you fixed it and discuss other possible solutions that you might not have included. For the last item, it is best to discuss them thoroughly as comments in the appropriate section of the code, but only give a short summary in the commit message. Note that all added and removed source code lines will also be circulated in the `gnuastro-commits` mailing list.

- Like all other Gnuastro's text files, the lines in the commit body should not be longer than 75 characters, see Section 12.3 [Coding conventions], page 448. This is to ensure that on standard terminal emulators (with 80 character width), the `git log` output can be cleanly displayed (note that the commit message is indented in the output of `git log`). If you use Emacs, Gnuastro's `.dir-locals.el` file will ensure that your commits satisfy this condition (using `M-q`).
- When the commit is related to a task or a bug, please include the respective ID (in the format of `bug/task #ID`, note the space) in the commit message (from Section 12.9 [Gnuastro project webpage], page 459) for interested people to be able to followup the discussion that took place there. If the commit fixes a bug or finishes a task, the recommended way is to add a line after the body with '`This fixes bug #ID.`', or '`This finishes task #ID.`'. Don't assume that the reader has internet access to check the bug's full description when reading the commit message, so give a short introduction too.

12.11.3 Production workflow

Fortunately 'Pro Git' has done a wonderful job in explaining the different workflows in Chapter 5[25] and in particular the "Integration-Manager Workflow" explained there. The implementation of this workflow is nicely explained in Section 5.2[26] under "Forked-Public-Project". We have also prepared a short tutorial in Section 12.11.4 [Forking tutorial], page 465. Anything on the master branch should always be tested and ready to be built and used. As described in 'Pro Git', there are two methods for you to contribute to Gnuastro in the Integration-Manager Workflow:

1. You can send commit patches by email as fully explained in 'Pro Git'. This is good for your first few contributions. Just note that raw patches (containing only the diff) do not have any meta-data (author name, date and etc). Therefore they will not allow us to fully acknowledge your contributions as an author in Gnuastro: in the `AUTHORS` file and at the start of the PDF book. These author lists are created automatically from the version controlled source.

   To receive full acknowledgment when submitting a patch, is thus advised to use Git's `format-patch` tool. See Pro Git's Public project over email (`https://git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project#Public-Project-over-Email`) section for a nice explanation. If you would like to get more heavily involved in Gnuastro's development, then you can try the next solution.

[25] `http://git-scm.com/book/en/v2/Distributed-Git-Distributed-Workflows`

[26] `http://git-scm.com/book/en/v2/Distributed-Git-Contributing-to-a-Project`

2. You can have your own forked copy of Gnuastro on any hosting site you like (GitHub, GitLab, BitBucket, or etc) and inform us when your changes are ready so we merge them in Gnuastro. This is more suited for people who commonly contribute to the code (see Section 12.11.4 [Forking tutorial], page 465).

In both cases, your commits (with your name and information) will be preserved and your contributions will thus be fully recorded in the history of Gnuastro and in the `AUTHORS` file and this book (second page in the PDF format) once they have been incorporated into the official repository. Needless to say that in such cases, be sure to follow the bug or task trackers (or subscribe to the `gnuastro-devel` mailing list) and contact us before hand so you don't do something that someone else is already working on. In that case, you can get in touch with them and help the job go on faster, see Section 12.9 [Gnuastro project webpage], page 459. This workflow is currently mostly borrowed from the general recommendations of Git[27] and GitHub. But since Gnuastro is currently under heavy development, these might change and evolve to better suit our needs.

12.11.4 Forking tutorial

This is a tutorial on the second suggested method (commonly known as forking) that you can submit your modifications in Gnuastro (see Section 12.11.3 [Production workflow], page 464).

To start, please create an empty repository on your hosting service webpage (we recommend GitLab[28]). If this is your first hosted repository on the webpage, you also have to upload your public SSH key[29] for the `git push` command below to work. Here we'll assume you use the name `janedoe` to refer to yourself everywhere and that you choose `gnuastro-janedoe` as the name of your Gnuastro fork. Any online hosting service will give you an address (similar to the '`git@gitlab.com:...`' below) of the empty repository you have created using their webpage, use that address in the third line below.

```
$ git clone git://git.sv.gnu.org/gnuastro.git
$ cd gnuastro
$ git remote add janedoe git@gitlab.com:janedoe/gnuastro-janedoe.git
$ git push janedoe master
```

The full Gnuastro history is now pushed onto your hosting service and the `janedoe` remote is now also following your `master` branch. If you run `git remote show REMOTENAME` for the `origin` and `janedoe` remotes, you will see their difference: the first has pull access and the second doesn't. This nicely summarizes the main idea behind this workflow: you push to your remote repository, we pull from it and merge it into `master`, then you finalize it by pulling from the main repository.

To test (compile) your changes during your work, you will need to bootstrap the version controlled source, see Section 3.2.2.1 [Bootstrapping], page 73, for a full description. The cloning process above is only necessary for your first time setup, you don't need to repeat

[27] `https://github.com/git/git/blob/master/Documentation/SubmittingPatches`

[28] See `https://www.gnu.org/software/repo-criteria-evaluation.html` for an evaluation of the major existing repositories. Gnuastro uses GNU Savannah (which also has the highest ranking in the evaluation), but for starters, GitLab may be easier.

[29] For example see this explanation provided by GitLab: `http://docs.gitlab.com/ce/ssh/README.html`.

it. However, please repeat the steps below for each independent issue you intend to work on.

Let's assume you have found a bug in `lib/statistics.c`'s median calculating function. Before actually doing anything, please announce it (see Section 1.7 [Report a bug], page 11) so everyone knows you are working on it or to find out others aren't already working on it. With the commands below, you make a branch, checkout to it, correct the bug, check if it is indeed fixed, add it to the staging area, commit it to the new branch and push it to your hosting service. But before all of them, make sure that you are on the `master` branch and that your `master` branch is up to date with the main Gnuastro repository with the first two commands.

```
$ git checkout master
$ git pull
$ git checkout -b bug-median-stats      # Choose a descriptive name
$ emacs lib/statistics.c
$                                       # do your checks here
$ git add lib/statistics.c
$ git commit
$ git push janedoe bug-median-stats
```

Your new branch is now on your hosted repository. Through the respective tacker on Savannah (see Section 12.9 [Gnuastro project webpage], page 459) you can then let the other developers know that your `bug-median-stats` branch is ready. They will pull your work, test it themselves and if it is ready to be merged into the main Gnuastro history, they will merge it into the `master` branch. After that is done, you can simply checkout your local `master` branch and pull all the changes from the main repository. After the pull you can run '`git log`' as shown below, to see how `bug-median-stats` is merged with master. To finalize, you can push all the changes to your hosted repository and delete the branch:

```
$ git checkout master
$ git pull
$ git log --oneline --graph --decorate --all
$ git push janedoe master
$ git branch -d bug-median-stats                # delete local branch
$ git push janedoe --delete bug-median-stats    # delete remote branch
```

Just as a reminder, always keep your work on each issue in a separate local and remote branch so work can progress on them independently. After you make your announcement, other people might contribute to the branch before merging it in to `master`, so this is very important. As a final reminder: before starting each issue branch from `master`, be sure to run `git pull` in `master` as shown above. This will enable you to start your branch (work) from the most recent commit and thus simplify the final merging of your work.

Appendix A Gnuastro programs list

GNU Astronomy Utilities 0.9, contains the following programs. They are sorted in alphabetical order and a short description is provided for each program. The description starts with the executable names in `thisfont` followed by a pointer to the respective section in parenthesis. Throughout this book, they are ordered based on their context, please see the top-level contents for contextual ordering (based on what they do).

Arithmetic
: (`astarithmetic`, see Section 6.2 [Arithmetic], page 161) For arithmetic operations on multiple (theoretically unlimited) number of datasets (images). It has a large and growing set of arithmetic, mathematical, and even statistical operators (for example `+`, `-`, `*`, `/`, `sqrt`, `log`, `min`, `average`, `median`).

BuildProgram
: (`astbuildprog`, see Section 11.2 [BuildProgram], page 328) Compile, link and run programs that depend on the Gnuastro library (see Section 11.3 [Gnuastro library], page 332). This program will automatically link with the libraries that Gnuastro depends on, so there is no need to explicitly mention them every time you are compiling a Gnuastro library dependent program.

ConvertType
: (`astconvertt`, see Section 5.2 [ConvertType], page 138) Convert astronomical data files (FITS or IMH) to and from several other standard image and data formats, for example TXT, JPEG, EPS or PDF.

Convolve
: (`astconvolve`, see Section 6.3 [Convolve], page 177) Convolve (blur or smooth) data with a given kernel in spatial and frequency domain on multiple threads. Convolve can also do de-convolution to find the appropriate kernel to PSF-match two images.

CosmicCalculator
: (`astcosmiccal`, see Section 9.1 [CosmicCalculator], page 307) Do cosmological calculations, for example the luminosity distance, distance modulus, comoving volume and many more.

Crop
: (`astcrop`, see Section 6.1 [Crop], page 151) Crop region(s) from an image and stitch several images if necessary. Inputs can be in pixel coordinates or world coordinates.

Fits
: (`astfits`, see Section 5.1 [Fits], page 128) View and manipulate FITS file extensions and header keywords.

MakeCatalog
: (`astmkcatalog`, see Section 7.4 [MakeCatalog], page 255) Make catalog of labeled image (output of NoiseChisel). The catalogs are highly customizable and adding new calculations/columns is very straightforward.

MakeNoise
: (`astmknoise`, see Section 8.2 [MakeNoise], page 301) Make (add) noise to an image, with a large set of random number generators and any seed.

MakeProfiles
: (`astmkprof`, see Section 8.1 [MakeProfiles], page 284) Make mock 2D profiles in an image. The central regions of radial profiles are made with a configurable 2D Monte Carlo integration. It can also build the profiles on an over-sampled image.

Match
: (`astmatch`, see Section 7.5 [Match], page 279) Given two input catalogs, find the rows that match with each other within a given aperture (may be an ellipse).

NoiseChisel
: (`astnoisechisel`, see Section 7.2 [NoiseChisel], page 225) Detect signal in noise. It uses a technique to detect very faint and diffuse, irregularly shaped signal in noise (galaxies in the sky), using thresholds that are below the Sky value, see arXiv:1505.01664 (`http://arxiv.org/abs/1505.01664`).

Segment
: (`astsegment`, see Section 7.3 [Segment], page 243) Segment detected regions based on the structure of signal and the input dataset's noise properties.

Statistics
: (`aststatistics`, see Section 7.1 [Statistics], page 208) Statistical calculations on the input dataset (column in a table, image or datacube).

Table
: (`asttable`, Section 5.3 [Table], page 147) Convert FITS binary and ASCII tables into other such tables, print them on the command-line, save them in a plain text file, or get the FITS table information.

Warp
: (`astwarp`, see Section 6.4 [Warp], page 199) Warp image to new pixel grid. Any projective transformation or Homography can be applied to the input images.

The programs listed above are designed to be highly modular and generic. Hence, they are naturally for lower-level operations. In Gnuastro, higher-level operations (combining multiple programs, or running a program in a special way), are done with installed Bash scripts (all prefixed with `astscript-`). They can be run just like a program and behave very similarly (with minor differences, see Chapter 10 [Installed scripts], page 316).

`astscript-sort-by-night`
: (See Section 10.1 [Sort FITS files by night], page 316) Given a list of FITS files, and a HDU and keyword name (for a date), this script separates the files in the same night (possibly over two calendar days).

Appendix B Other useful software

In this appendix the installation of programs and libraries that are not direct Gnuastro dependencies are discussed. However they can be useful for working with Gnuastro.

B.1 SAO ds9

SAO ds9[1] is not a requirement of Gnuastro, it is a FITS image viewer. So to check your inputs and outputs, it is one of the best options. Like the other packages, it might already be available in your distribution's repositories. It is already pre-compiled in the download section of its webpage. Once you download it you can unpack and install (move it to a system recognized directory) with the following commands (`x.x.x` is the version number):

```
$ tar xf ds9.linux64.x.x.x.tar.gz
$ sudo mv ds9 /usr/local/bin
```

Once you run it, there might be a complaint about the Xss library, which you can find in your distribution package management system. You might also get an `XPA` related error. In this case, you have to add the following line to your `~/.bashrc` and `~/.profile` file (you will have to log out and back in again for the latter):

```
export XPA_METHOD=local
```

B.1.1 Viewing multiextension FITS images

The FITS definition allows for multiple extensions inside one FITS file, each extension can have a completely independent dataset inside of it. If you just double click on a multi-extension FITS file or run `$ds9 foo.fits`, SAO ds9 will only show you the first extension. If you have a multi-extension file containing 2D images, one way to load and switch between the each 2D extension is to take the following steps in the SAO ds9 window: "File"→"Open Other"→"Open Multi Ext Cube" and then choose the Multi extension FITS file in your computer's file structure.

The method above is a little tedious to do every time you want view a multi-extension FITS file. A different series of steps is also necessary if you the extensions are 3D data cubes. Fortunately SAO ds9 also provides command-line options that you can use to specify a particular behavior. One of those options is `-mecube` which opens a FITS image as a multi-extension data cube (treating each 2D extension as a slice in a 3D cube). This allows you to flip through the extensions easily while keeping all the settings similar.

Try running `$ds9 -mecube foo.fits` to see the effect (for example on the output of Section 7.2 [NoiseChisel], page 225). If the file has multiple extensions, a small window will also be opened along with the main ds9 window. This small window allows you to slide through the image extensions of `foo.fits`. If `foo.fits` only consists of one extension, then SAO ds9 will open as usual. Just to avoid confusion, note that SAO ds9 does not follow the GNU style of separating long and short options as explained in Section 4.1.1 [Arguments and options], page 92. In the GNU style, this 'long' (multi-character) option should have been called like `--mecube`, but SAO ds9 follows its own conventions.

Recall the `-mecube` opens each 2D input extension as a slice in 3D. Therefore, when you want to inspect a multi-extension FITS file containing a 3D dataset, the `-mecube` option is

[1] `http://ds9.si.edu/`

no good any more (it only opens the first slice of the 3D cube in each extension). In that case, we have to use SAO ds9's `-multiframe` option to open each extension as a separate frame. Since the input is a 3D dataset, we get the same small window as the 2D case above for scrolling through the 3D slices. We then have to also ask ds9 to match the frames and lock the slices, so for example zooming in one, will also zoom the others.

We can use a script to automatize this process and make work much easier (and save a lot of time) when opening any generic 2D or 3D dataset. After taking the following steps, when you click on a FITS file in your graphic user interface, ds9 will open in the respective 2D or 3D mode when double clicking a FITS file on the graphic user interface, and an executable will also be available to open ds9 similarly on the command-line. Note that the following solution assumes you already have Gnuastro installed (and in particular the Section 5.1 [Fits], page 128, program).

Let's assume that you want to store this script in `BINDIR` (that is in your `PATH` environment variable, see Section 3.3.1.2 [Installation directory], page 79). [Tip: a good place would be `~/.local/bin`, just don't forget to make sure it is in your `PATH`]. Using your favorite text editor, put the following script into a file called `BINDIR/ds9-multi-ext`. You can change the size of the opened ds9 window by changing the `1800x3000` part of the script below.

```
#! /bin/bash

# To allow generic usage, if no input file is given (the ‘if’ below is
# true), then just open an empty ds9.
if [ "x$1" == "x" ]; then
    ds9
else
    # Make sure we are dealing with a FITS file. We are using shell
    # redirection here to make sure that nothing is printed in the
    # terminal (to standard output when we have a FITS file, or to
    # standard error when we don't). Since we've used redirection,
    # we'll also have to echo the return value of ‘astfits’.
    check=$(astfits $1 -h0 > /dev/null 2>&1; echo $?)

    # If the file was a FITS file, then ‘check’ will be 0.
    if [ "$check" == "0" ]; then

        # Read the number of dimensions.
        n0=$(astfits $1 -h0 | awk '$1=="NAXIS"{print $3}')

        # Find the number of dimensions.
        if [ "$n0" == "0" ]; then
            ndim=$(astfits $1 -h1 | awk '$1=="NAXIS"{print $3}')
        else
            ndim=$n0
        fi;

        # Open DS9 based on the number of dimension.
```

```
            if [ "$ndim" = "2" ]; then
                # 2D multi-extension file: use the "Cube" window to
                # flip/slide through the extensions.
                ds9 -zscale -geometry 1800x3000 -mecube $1         \
                    -zoom to fit -wcs degrees
            else
                # 3D multi-extension file: The "Cube" window will slide
                # between the slices of a single extension. To flip
                # through the extensions (not the slices), press the top
                # row "frame" button and from the last four buttons of the
                # bottom row ("first", "previous", "next" and "last") can
                # be used to switch through the extensions (while keeping
                # the same slice).
                ds9 -zscale -geometry 1800x3000 -wcs degrees       \
                    -multiframe $1 -match frame image              \
                    -lock slice image -lock frame image -single    \
                    -zoom to fit
            fi
        else
            if [ -f $1 ]; then
                echo "'$1' isn't a FITS file."
            else
                echo "'$1' doesn't exist."
            fi
        fi
    fi
```

As described above (also in the comments of the script), if you have opened a multi-extension 2D dataset (image), the "Cube" window can be used to slide/flip through each extension. But when the input is a 3D data cube, the "Cube" window will slide/flip through the slices in each extension (a separate 3D dataset). To flip through the extensions (while keeping the slice fixed), click the "frame" button on the top row of buttons, then use the last four buttons of the bottom row ("first", "previous", "next" and "last") to change between the extensions.

To run this script, you have to activate its executable flag with this command:

```
$ chmod +x BINDIR/ds9-multi-ext
```

If `BINDIR` is within your system's `PATH` environment variable (see Section 3.3.1.2 [Installation directory], page 79), you can now open ds9 conditionally using the script above with this command:

```
$ ds9-multi-ext foo.fits
```

For the graphic user interface, we'll assume you are using GNOME (the most popular graphic user interface for GNU/Linux systems), version 3. For GNOME 2, see below. You can customize GNOME to open specific files with `.desktop` files. For each user, they are stored in `~/.local/share/applications/`. In case you don't have the directory, make it your self (with `mkdir`). Using your favorite text editor, you can now create `~/.local/share/applications/saods9.desktop` with the following contents. Just don't

forget to correct `BINDIR`. If you would also like to have ds9's logo/icon in GNOME, download it, uncomment the `Icon` line, and write its address in the value.

```
[Desktop Entry]
Type=Application
Version=1.0
Name=SAO DS9
Comment=View FITS images
Terminal=false
Categories=Graphics;RasterGraphics;2DGraphics;3DGraphics
#Icon=/PATH/TO/DS9/ICON/ds9.png
Exec=BINDIR/ds9-multi-ext %f
```

The steps above will add SAO ds9 as one of your applications. To make it default, take the following steps (just once is enough). Right click on a FITS file and select Open with other application→View all applications→SAO ds9.

In case you are using GNOME 2 you can take the following steps: right click on a FITS file and choose Properties→Open With→Add button. A list of applications will show up, ds9 might already be present in the list, but don't choose it because it will run with no options. Below the list is an option "Use a custom command". Click on it and write the following command: `BINDIR/ds9-multi-ext` in the box and click "Add". Then finally choose the command you just added as the default and click the "Close" button.

B.2 PGPLOT

PGPLOT is a package for making plots in C. It is not directly needed by Gnuastro, but can be used by WCSLIB, see Section 3.1.1.3 [WCSLIB], page 63. As explained in Section 3.1.1.3 [WCSLIB], page 63, you can install WCSLIB without it too. It is very old (the most recent version was released early 2001!), but remains one of the main packages for plotting directly in C. WCSLIB uses this package to make plots if you want it to make plots. If you are interested you can also use it for your own purposes.

If you want your plotting codes in between your C program, PGPLOT is currently one of your best options. The recommended alternative to this method is to get the raw data for the plots in text files and input them into any of the various more modern and capable plotting tools separately, for example the Matplotlib library in Python or PGFplots in LaTeX. This will also significantly help code readability. Let's get back to PGPLOT for the sake of WCSLIB. Installing it is a little tricky (mainly because it is so old!).

You can download the most recent version from the FTP link in its webpage[2]. You can unpack it with the `tar xf` command. Let's assume the directory you have unpacked it to is `PGPLOT`, most probably it is: `/home/username/Downloads/pgplot/`. open the `drivers.list` file:

```
$ gedit drivers.list
```

Remove the `!` for the following lines and save the file in the end:

```
PSDRIV 1 /PS
PSDRIV 2 /VPS
PSDRIV 3 /CPS
```

[2] `http://www.astro.caltech.edu/~tjp/pgplot/`

```
PSDRIV 4 /VCPS
XWDRIV 1 /XWINDOW
XWDRIV 2 /XSERVE
```

Don't choose GIF or VGIF, there is a problem in their codes.

Open the `PGPLOT/sys_linux/g77_gcc.conf` file:

```
$ gedit PGPLOT/sys_linux/g77_gcc.conf
```

change the line saying: `FCOMPL="g77"` to `FCOMPL="gfortran"`, and save it. This is a very important step during the compilation of the code if you are in GNU/Linux. You now have to create a folder in `/usr/local`, don't forget to replace `PGPLOT` with your unpacked address:

```
$ su
# mkdir /usr/local/pgplot
# cd /usr/local/pgplot
# cp PGPLOT/drivers.list ./
```

To make the Makefile, type the following command:

```
# PGPLOT/makemake PGPLOT linux g77_gcc
```

It should finish by saying: `Determining object file dependencies`. You have done the hard part! The rest is easy: run these three commands in order:

```
# make
# make clean
# make cpg
```

Finally you have to place the position of this directory you just made into the `LD_LIBRARY_PATH` environment variable and define the environment variable `PGPLOT_DIR`. To do that, you have to edit your `.bashrc` file:

```
$ cd ~
$ gedit .bashrc
```

Copy these lines into the text editor and save it:

```
PGPLOT_DIR="/usr/local/pgplot/"; export PGPLOT_DIR
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/pgplot/
export LD_LIBRARY_PATH
```

You need to log out and log back in again so these definitions take effect. After you logged back in, you want to see the result of all this labor, right? Tim Pearson has done that for you, create a temporary folder in your home directory and copy all the demonstration files in it:

```
$ cd ~
$ mkdir temp
$ cd temp
$ cp /usr/local/pgplot/pgdemo* ./
$ ls
```

You will see a lot of pgdemoXX files, where XX is a number. In order to execute them type the following command and drink your coffee while looking at all the beautiful plots! You are now ready to create your own.

```
$ ./pgdemoXX
```

Appendix C GNU Free Doc. License

Version 1.3, 3 November 2008

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released

under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

   It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

   You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

   A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any,

be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their

titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

   You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

   If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

   However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

   Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

   Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `https://www.gnu.org/copyleft/`.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

    "Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

    "CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

    "Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

    An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

    The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.3
or any later version published by the Free Software Foundation;
with no Invariant Sections, no Front-Cover Texts, and no Back-Cover
Texts.  A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with. . . Texts." line with this:

```
with the Invariant Sections being list their titles, with
the Front-Cover Texts being list, and with the Back-Cover Texts
being list.
```

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Appendix D GNU Gen. Pub. License v3

Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program—to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

   "This License" refers to version 3 of the GNU General Public License.

   "Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

   "The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

   To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

   A "covered work" means either the unmodified Program or a work based on the Program.

   To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

   To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

   An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

   The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

   A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

   All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

   You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

   Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

   No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

   You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

   You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

   You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

   a. The work must carry prominent notices stating that you modified it, and giving a relevant date.

   b. The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

   c. You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

   d. If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

   A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

   You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b. Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c. Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d. Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e. Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source.

The information must suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a. Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or
b. Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or
c. Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d. Limiting the use for publicity purposes of names of licensors or authors of the material; or

e. Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f. Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance.

However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

    Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

    An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

    You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

    A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

    A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of this License.

    Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

    In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

    If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so

available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

    If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

    Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year name of author

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or (at
your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see https://www.gnu.org/licenses/.
```

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
program Copyright (C) year name of author
This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type 'show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, your program's commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see `https://www.gnu.org/licenses/`.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read `https://www.gnu.org/licenses/why-not-lgpl.html`.

Index: Macros, structures and functions

All Gnuastro library's exported macros start with `GAL_`, and its exported structures and functions start with `gal_`. This abbreviation stands for *G*NU *A*stronomy *L*ibrary. The next element in the name is the name of the header which declares or defines them, so to use the `gal_array_fset_const` function, you have to `#include <gnuastro/array.h>`. See Section 11.3 [Gnuastro library], page 332, for more. The `pthread_barrier` constructs are our implementation and are only available on systems that don't have them, see Section 11.3.2.1 [Implementation of `pthread_barrier`], page 334.

G

P

Index

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z