REPORT ON OpenDreamKit DELIVERABLE D3.10

Packaging components and user-contributed code for major Linux distributions

NICOLAS M. THIÉRY, VIVIANE PONS, FLORENT HIVERT, SAMUEL LELIÈVRE, ERIK BRAY, LOÏC GOUARIN, FLORIAN RABE, VINCENT DELECROIX, VINCENT KLEIN, CLÉMENT PERNET, JEAN-GUILLAUME DUMAS, DANIEL SCHULTZ, DMITRII PASECHNIK STEVE LINTON, ALEXANDER KONOVALOV, MARKUS PFEIFFER, MICHAEL TORPEY, LUCA DE FEO, JOHN E. CREMONA, OLIVIER CAYROL, JULIEN CRISTAU, JEROEN DEMEYER, ...

Due on	31/08/2019 (M48)
Delivered on	31/08/2019
Lead	Université de Versailles Saint-Quentin (UVSQ)
Progress on and finalization of this deliverable has been tracked publicly at: `https://github.com/OpenDreamKit/OpenDreamKit/issues/59`	

1. Package repositories in Linux

Operating systems offer several ways to install and/or update software. Since long, Microsoft Windows has accustomed its users with the "installer" pattern: obtain an executable, by downloading it or by running it from external media (e.g., a CD or DVD), which takes care of installing the software in the appropriate locations.

At the same time, Linux *distributions* have privileged a different software delivery mechanism: official (and unofficial) *repositories*, collections of installable *packages* that software called a *package manager* can retrieve (typically via download) and install in the appropriate location. Thanks to ever increasing Internet speeds, the package repository model has gained in popularity, and has notably been replicated by the "app store" concept of mobile OSes, and more recently Windows 10.

While in the installer model the responsibility for packaging and distributing software mostly falls on the software vendor, the package repository model allows for a variety of policies: in the Linux space, it is customary for distributions to have official repositories, where software are packaged by select members of the community, along with unofficial repositories where packages can be contributed by anyone.

1.1. Package ecosystems in Linux

Over time, several different package formats have developed for Linux, with associated package managers. Linux distributions tend to support chiefly one specific package format, eventually providing tools for manipulating packages in other formats. There is usually a distinction between *source packages* and *binary packages*: the former contain source code, and are eventually used to build the latter. Most Linux distributions provide software to end users in the form of binary packages (Gentoo is a notable exception); in the rest of this document we will mostly refer to binary packages.

The two most popular package formats are `.deb` and `.rpm`, respectively created by the Debian and Red Hat distributions. Nowadays, each of them serves as the main format for several popular distributions; Table 1 summarizes the principal Linux distributions with their package format.

Package format	Distributions	Package manager
`.rpm`	RHEL, Fedora, CentOS, OpenMandriva, OpenSUSE, ...	yum
`.deb`	Debian, Ubuntu, Mint, Knoppix, Raspbian, ...	apt
Portage	Gentoo, Chrome OS, ...	emerge
`PKGBUILD`	Arch, Manjaro, ...	pacman
Anaconda	*OS-agnostic*	conda

TABLE 1. Most popular Linux distributions and package formats

Programming language	Official repository	Package managers
Python	PyPI	pip
Julia	General	Pkg2
R	CRAN	install
Perl	CPAN	PPM
JavaScript	NPM	npm
any	Conda-forge	conda

TABLE 2. Some programming languages and their official package systems

One of the main prerogatives of a Linux distribution is to choose what packages to make available through its official repositories, prepare them, and ensure that they are compatible with one another. Different distributions that use the same package format may or may not share the same packages. For example, Debian and Ubuntu have mostly disjoint package sets, prepared independently by the respective communities; however some packages in Ubuntu, notably those that pertain to OpenDreamKit, are provided by the DebianScience team[1].

1.2. **Other packaging paradigms**

Besides "classic" Linux package systems, several other paradigms have emerged over the years.

Most programming languages nowadays come with an official repository where source code or pre-compiled code can be stored in the form of packages, and retrieved using a language-specific package manager, regardless of the OS of the user. Examples of this paradigm are the Python Package Index (PyPI)[2] for Python, or the Comprehensive R Archive Network (CRAN)[3] for R. While language-specific repositories are mainly targeted at code written in the respective programming language, every system also offers facilities to install dependencies written in other programming languages, such as pre-compiled C libraries, thus blurring the boundary between the OS package manager and the programming language one. Table 2 summarizes the official package repository and package manager for some popular programming languages relevant to OpenDreamKit.

Agnostic package systems push the concept one step further, by being unrelated to both an OS and a programming language. Currently, one of the most popular package systems is Anaconda, which consists of an open-source package manager (conda), and a community-led package repository (Conda-forge[4]), backed by a commercial company named Anaconda. Originally born as a package system for Python, it has evolved into a OS-and-language-agnostic package system targeted at scientific computing and data science. Centered around JupyterLab and the Jupyter

[1] `https://wiki.debian.org/DebianScience`.
[2] `https://pypi.org/`
[3] `https://cran.r-project.org/`
[4] `https://conda-forge.org/`

notebook for its user interface, Anaconda offers a one-stop solution to install all programming languages, libraries and tools to work on scientific projects.

Being primarily targeted at developers, rather than end users, both language-specific and language-agnostic package repositories are usually open to contributions from anyone at any time.[5] Unlike OS package repositories, they do not aim at having a full set of software all compatible with one another at any given time, but rather at having all versions (including bleeding edge) of any given software installable at all time. To better cater to development workflows, these package managers usually provide an isolation mechanism (often called *environments*) that permits to have several versions of the same software installed and running at the same time. This way, even if two software have conflicting dependencies, they can be installed at the same time by having two separate environments for them.

While potentially more demanding in resources, the installation model based on isolated environments allows more flexibility, and is especially interesting for *reproducible* software builds. It has thus been replicated in the OS space by some Linux distributions, in particular NixOS[6] and GUIX[7]. A more extreme mechanism uses *containerization* to isolate software so that distinct environments do not even share system resources such as file-systems or process tables. This is the approach, taken for example by Red Hat's Container Linux[8] (formerly CoreOS), which is primarily targeted at cloud infrastructures.

1.3. **Relevance for OpenDreamKit**

The software made by OpenDreamKit is primarily available as source code and precompiled binaries from the respective project web pages. Typically, software projects (as opposed to libraries) also provide executable installers from their web pages, thus installation by the installer pattern is always an option for the end user, and in some cases (e.g., installation on Windows) even the preferred one (see D3.7).

Packaging software for package repositories is rarely the responsibility of the software developer, and indeed, outside a few exceptional cases, OpenDreamKit software is not packaged by OpenDreamKit members. While not under our direct responsibility, it is nevertheless of primary importance that our software is made available through as many package repositories as possible. Indeed, while most end-users (Windows users in particular) who install software on their personal computers are perfectly happy with the installer pattern, OpenDreamKit's goal is to reach a much larger audience: shared server and cloud infrastructures, the vast majority of which runs under Linux, are extremely reliant on package managers (both OS-specific and OS-agnostic) for installing software.

This deliverable describes the efforts made by OpenDreamKit to have all its software components packaged for the official repositories of all major Linux distributions. While not precisely within the scope of this deliverable, we also report on packaging for OS-agnostic repositories, and on the impact on software design.

A different packaging issue is having a package system *for* a OpenDreamKit software, in the same vein of language-specific package systems. For example, GAP has developed a package format, a package manager, and maintains an official repository, so that users can publish and share their code written in GAP with other users. Since this aspect of packaging is inextricably linked with the former, we also report on it.

[5] In some cases, like for Anaconda, a separate curated package repository may be offered to paying customers.
[6] `https://nixos.org/`
[7] `https://guix.gnu.org/`
[8] `https://coreos.com/`

Distribution	Maintainers
Debian, Ubuntu	Bill Allombert, Debian Science
Fedora	Paulo César Pereira de Andrade *et al.*
Arch	Antonio Rojas, Felix Yan
Gentoo	François Bissey *et al.*
Conda-forge	Isuru Fernando *et al.*

TABLE 3. Package maintainers for various distributions

Codename	Number	Date
Jessie	8	April 26th, 2015
Stretch	9	June 17th, 2017
Buster	10	July 6th, 2019

TABLE 4. Debian releases during the OpenDreamKit project

2. LINUX PACKAGES FOR OPENDREAMKIT COMPONENTS

The goal of this deliverable was to have official packages for all major Linux distributions for all OpenDreamKit components. Before the start of OpenDreamKit, availability varied greatly among distributions and software. To begin, it is necessary to define what is meant by "major" distribution; in an inevitably biased way, we chose to target the following distributions:

- Debian, Fedora: for their popularity among advanced end-users and system administrators;
- Ubuntu: for its popularity among beginners and casual users, as well as in the server and cloud space;
- Arch, Gentoo: for their popularity among power-users.

Being the lone project depending upon all other OpenDreamKit components, SAGE is automatically the most challenging to package. Hence, packagers for SAGE are usually, though not always, also responsible for packaging most OpenDreamKit components within a distribution. Table 3 summarizes the currently active package maintainers for each of the distributions above; with the exception of Bill Allombert, and some occasional members of the Debian Science group, all package maintainers are external to the OpenDreamKit project.

With the exception of Debian and Ubuntu, all distributions above have a history of regularly packaging SAGE (and thus all its dependencies) since at least 2013, well before the start of OpenDreamKit. Hence, the strategy adopted for this deliverable was to have SAGE packaged for Debian, in collaboration with the Debian Science team, with all SAGE dependencies and packages for Ubuntu automatically following from this work.

Debian has a two year long *release cycle*, each cycle beginning at the middle of odd numbered years. This means that every two years the community decides on which packages should be included in the next stable release, at which point a *freeze* happens and packages within a release can only be updated to fix bugs and security issues. "Missing the train" for one release cycle means that a software cannot enter Debian for the next two years. Table 4 summarizes the Debian releases that happened during the 2015-2019 period.

Ubuntu has a similar two year long *long-term-support (LTS)* release cycle, but also a shorter six-month cycle for quicker releases, thus allowing more chances to release a package.

2.1. **Status before OpenDreamKit**

At the start of OpenDreamKit, only packages for PARI/GP and GAP were available in Debian/Ubuntu, thanks to the continued efforts of Bill Allombert, a long time member of the Debian community.

Software	Stable version	Version in Debian	Maintainer	Dependencies	Dependents
GAP	4.7.8	4r7p5-2	Bill Allombert	13	9
PARI/GP	2.7.4	2.7.2-1	Bill Allombert	36	6
SAGE	6.8	—	—	—	—
SINGULAR	4.0.1	—	—	—	—

TABLE 5. OpenDreamKit packages in Debian Jessie, September 1, 2015. Source `http://snapshot.debian.org/`.

Software	Stable version	Version in Debian	Maintainer	Dependencies	Dependents
GAP	4.8.8	4r8p6-2	Bill Allombert	14	20
PARI/GP	2.9.3	2.9.1-1	Bill Allombert	39	9
SAGE	8.0	7.4-9	Debian Science	653	3
SINGULAR	4.0.3	1:4.0.3-p3+ds-5	Debian Science	21	4

TABLE 6. OpenDreamKit packages in Debian Stretch, August 31, 2017. Source `http://snapshot.debian.org/`.

Software	Stable version	Version in Debian	Maintainer	Dependencies	Dependents
GAP	4.10.2	4r10p0-7	Bill Allombert	15	13
PARI/GP	2.11.2	2.11.1-2	Bill Allombert	41	9
SAGE	8.8	8.6-6	Debian Science	593	3
SINGULAR	4.1.2	1:4.1.1-p2+ds-3	Debian Science	23	4

TABLE 7. OpenDreamKit packages in Debian Buster, August 29, 2019. Source `http://snapshot.debian.org/`.

Information on the packages available at the time are summarized in Table 5; for succinctness, we only list the four core computer algebra systems (CAS) distributed by OpenDreamKit, leaving libraries out of the picture.[9] To give an idea of the complexity of the task, we list the number of packages in the dependency tree of each software (i.e., the total number of packages that would get installed on a fresh Debian system with no other software than the package manger); we do the same for the number of *dependent* packages (packages that depend on the software).

SAGE (version 3.0 at the time) had been packaged in Debian for the last time in 2010, but fell off the official distribution due to incompatible dependencies. Ubuntu used to have an unofficial SAGE package, also installing SINGULAR and other dependencies alongside, but this package offered little in terms of compatibility and integration with the rest of the system.

2.2. **Current status**

The first occasion to have all OpenDreamKit components packaged for Debian was offered by the *freeze* for the Debian 9.0 (codenamed "Stretch") release in 2017. Thanks to the intense collaboration between OpenDreamKit and the Debian Science team, the occasion was not missed, thus achieving the goal of the deliverable 24 months in advance of the projected deadline. Table 6 summarizes information on the packages available in Debian Stretch. The same packages were added to the official Ubuntu repositories at the same time.

The next Debian release cycle came to completion in July 2019; all OpenDreamKit packages were updated and improved for the occasion, with a marked simplification of the dependency tree. Table 7 summarizes information on the packages available in Debian 10 (codenamed "Buster").

[9] Libraries typically have fewer dependencies and are easier to package.

2.3. **Beyond Linux packages**

Having achieved the goals of the deliverable largely in advance, we took the occasion to explore other package repositories. At the OpenDreamKit workshop in Cernay (see D2.15), besides experiments with GUIX and Nix packages, a sprint took place to have SAGE packaged for Conda-forge. At the time of writing, the latest SAGE 8.8 and all its dependencies are available on Conda-forge for the Linux and MacOS platforms. This packaging effort is still experimental, and will only become stable after SAGE is officially migrated to Python 3 (see below).

3. WORK ACCOMPLISHED

If a software can be installed on a system, then, in principle, it can be packaged for it. And, indeed, a SAGE package for Ubuntu had been distributed through an unofficial channel for many years.

The difficulty is not packaging a software, as much as packaging it *properly*. Indeed, a package system tries to fulfill several goals:

- Code reuse: by having a separate package for a component shared by several software, the component can be installed only once and reused many times. This obviously saves disk space, and in the case of shared libraries it also saves memory and cpu, since the library needs only be loaded once in shared memory.
- Compatibility: different software may depend on different versions of a same component, and break when incompatible versions are used. By ensuring that all software within a release is mutually compatible, Linux distributions provide a smooth experience for the user.

  Nevertheless, some times it is impossible to satisfy all dependent software with a single version of a component, in this case, it is not unusual for Linux distributions to provide two different versions of the same component installable side to side (e.g., Python 2 and Python 3 are both available in all Linux distributions at the time of writing). While this approach is in tension with the goal of code reuse, it is sometimes the most pragmatic one.
- Flexibility: a software may rely on two components to do the same job, using indifferently one or the other according to which is available in the system. By having separate packages, a user can freely choose which component he prefers to have installed, and even change his mind later, without breaking dependent software.

It is clear that, the higher the number of dependencies, the higher the complexity of packaging a software is. In the case of OpenDreamKit, a single software may interact with hundreds of packages, as illustrated in Tables 5–7.

In this sense, the legacy unofficial SAGE package for Ubuntu, installing all of SAGE and its dependencies nearly as a unique huge package, was not playing by the rules, which explains why it was not included in the official repositorites. Furthermore, the strong traditions of the Debian community (length of the release cycle, large quantity of scientific software already packaged, reluctance to host alternative/modified versions of packages), exacerbated the problem and made packaging SAGE and its dependencies extremely challenging.

The successful packaging of all OpenDreamKit components in Debian/Ubuntu was thus the result of several actions undertaken by the OpenDreamKit members in concert with the concerned communities. We list below the most important actions.

Workshops dedicated to packaging: A series of workshop specifically targeted at packaging was organized in Cernay (near Paris):

- SAGE days 77, April 4–8, 2016, see D2.2;
- SAGE days 85, April 30 – May 4, 2018, see D2.11;
- SAGE days 101, June 17–21, 2019, see D2.15.

They were the occasion to gather the various actors involved in packaging OpenDreamKit components, exchange ideas, run experiments, and sprint on development. A mailing list `sage-packaging@googlegroups.com` was created to keep exchanging on issues related to packaging after the workshops.

Limiting *patches*: When a project depends on an external component, it may need, in some instances, to *patch* the external dependency, i.e., create a slightly modified version of the official component that interfaces better with the project. Several different reasons may lead a project to patch a dependency:

- A bug is found in the dependency, but its developers either disagree with the proposed fix, or have a timeline for releasing it that is incompatible with that of the dependent project.
- The dependency makes assumptions about its environment (file paths, system resources, ...) that are unsatisfied within the dependent project.
- Two or more dependencies are incompatible, but the dependent project needs all; a patch to one of the dependencies may help them coexist.

While patches may be unavoidable, they complicate the work for package maintainers: which version of a component should be included in the official repository, the original or the patched one? In some cases, it may be necessary to include both, which negates the benefits cited previously. For this reason, Linux distributions usually discourage projects patching dependencies, keeping that prerogative to themselves.

SAGE, owing to its large number of dependencies, has traditionally behaved very badly in this respect, maintaining a large set of patches. An great amount of work has gone into *pushing patches upstream* (i.e., having them accepted in the original project) whenever possible, and in removing useless or avoidable patches.

Updating dependencies: As dependencies evolve, it is necessary to ensure that the dependent project is as compatible as possible with the latest version. Failing which, package maintainers would be faced with the dilemma of updating the dependency and thus breaking the dependent, or keeping the dependency at an obsolete version.

While in most cases updating a dependency requires little to no work, major version changes may require considerable effort. Notable examples for SAGE were the upgrade of SINGULAR from version 3 to 4, and the upgrade of Python from version 2 to 3. The former was completed in mid-2016, one year and a half after the initial release of SINGULAR 4; the latter is still underway (see next section).

Modularization of OpenDreamKit software: Identifying and cutting out meaningful components from a large piece of software can also help make packaging easier; delegating work even more.

With the help of OpenDreamKit, the various software projects undertook a broad effort in modularizing their code base, leading to a considerable reduction in duplicated efforts and improved code quality.

Important examples are the adoption of Jupyter as unified user interface, which in the case of SAGE paved the way to the deprecation of the old Notebook. The development of LIBGAP (see D5.15) let SAGE depend on an officially supported C API to GAP, rather than on fragile bindings developed internally. Finally, spinning the projects CySignals and CyPari off mainline SAGE simplified the process of packaging it, and allowed other projects outside OpenDreamKit to depend on them without having to import the full SAGE.

Providing alternate workflows for user-contributed code: Finally, a different issue, mostly relevant to full-fledged computer algebra systems, is the packaging of user-contributed code *for* the system. Indeed, there is a growing demand for workflows that allow to easily share code developed by the users of a system, and this demand can hardly be fulfilled

by a distribution's package repository. In the same vein of language-specific package repositories, the need was felt to have repositories for SAGE and GAP.[10]

GAP already had a package system before the start of OpenDreamKit, but this was the occasion to improve the existing tools and simplify the workflows for the users. See D5.15.

SAGE used to encourage its users to directly contribute code to the mainline, however this could be a frustrating experience for beginners, who had to learn the coding standards of SAGE and go through the standard review process; it also led in some cases to *bit-rotting*: old unmaintained code in SAGE with no designed maintainer. Some SAGE developers advocated distributing user code in the form of SPKGs, the internal format SAGE uses to pack dependencies. Although this is a valid method, the responsibility of hosting the SPKG still falls on the user, and the available tools are rather limited.

Lately, a consensus has developed in the community on encouraging users to use the standard Python packaging tools, and host their packages on PyPI, the main repository for Python projects. Support for installing packages from PyPI has been integrated in SAGE via the `sage -pip` command, and documentation and example packages[11]have been developed to guide the user in the process.

While these improvements do not directly impact packaging for Linux distributions, the added possibilities for the end-users enable faster growth of the projects, at no additional expense for package maintainers.

4. THE FUTURE

The quantity of work spent to have software properly packaged raises the question of its sustainability. Indeed, many tasks are recurring and need to be iterated at each new release cycle: updating dependencies, integrating patches upstream, maintaining interfaces, ... While the end of OpenDreamKit will be felt, we argue that the efforts done until now will not be lost, and will greatly simplify the tasks of package maintainers. Indeed, the need for some of the actions undertaken had been felt for a long time, and OpenDreamKit offered the occasion to address many long-standing issues. With OpenDreamKit coming to an end, many foundational issues have already been solved, and only recurring tasks need to be dealt with for updating packages.

Among the major foundational issues still open, is the migration of SAGE to Python 3.[12] Fortunately, thanks to a titanic effort involving hundreds of entries in the issue tracker,[13] with the help of members of OpenDreamKit,[14] SAGE is now compatible with Python 3, and only a few months away from officially abandoning Python 2.

The move to Python 3 will also be the occasion to advance on several projects that had stalled because of the limitations of Python 2. In particular, packaging for Conda-forge will be simplified after the shift, and it will be easier to build Windows-compatible Anaconda packages. Anaconda packages will also enable easier building of JupyterHub and Binder-compatbile images, further stimulating the development of workflows based on OpenDreamKit components.

Disclaimer: this report, together with its annexes and the reports for the earlier deliverables, is self contained for auditing and reviewing purposes. Hyperlinks to external resources are

[10]Due to its smaller and more specialized user base, a repository for PARI/GP seems unjustified for the moment. SINGULAR is in the process of migrating to Julia, in the context of the OSCAR project, and will likely use the standard Julia package tools when the migration will be complete.

[11]`https://github.com/sagemath/sage_sample`.

[12]Python 2 reaches its end of life in 2020. After that point, it will become nearly impossible to package a Python 2 version of SAGE for any system.

[13]`https://trac.sagemath.org/ticket/26212`.

[14]We are grateful to F. Chapoton for leading this effort.

meant as a convenience for casual readers wishing to follow our progress; such links have been checked for correctness at the time of submission of the deliverable, but there is no guarantee implied that they will remain valid.