# Continuous Reproducibility in GNSS Signal Processing

Carles Fernández-Prades, Jordi Vilà-Valls, Javier Arribas, Antonio Ramos

*Digital Object Identifier 10.1109/ACCESS.2018.2822835*

# Continuous Reproducibility in GNSS Signal Processing

**CARLES FERNÁNDEZ-PRADES, (Senior Member, IEEE), JORDI VILÀ-VALLS, (Senior Member, IEEE), JAVIER ARRIBAS, (Senior Member, IEEE), AND ANTONIO RAMOS**

Department of Statistical Inference for Communications and Positioning, Centre Tecnològic de Telecomunicacions de Catalunya, 08860 Castelldefels, Spain

Corresponding author: Carles Fernández-Prades (cfernandez@cttc.cat)

This work was supported in part by the Spanish Ministry of Economy and Competitiveness through Project TEC2015-69868-C2-2-R (ADVENTURE) and in part by the Government of Catalonia under Grant 2017–SGR–1479.

**ABSTRACT** This paper discusses the reproducibility of scientific experiments in which global navigation satellite system (GNSS) signals play a role. After analyzing the factors that impact the reproducibility of an experiment in the given context, this paper proposes a methodology that, leveraging on software containerization technologies and the best practices from professional software development, ensures the automated reproduction of scientific experiments involving GNSS data and software-defined GNSS receivers, including the generation of figures or tables of a research publication, while fostering scientific collaboration and contributing to mitigate the effects of software aging in mutating computer environments. In order to show a practical implementation of the proposed work flow, the authors propose a simple experiment based on a freely available open source software-defined receiver and the automation of its execution in a popular online platform.

**INDEX TERMS** Digital signal processing, global navigation satellite system, open source software, reproducibility of results, software radio.

## I. INTRODUCTION

This paper discusses software-defined Global Navigation Satellite System (GNSS) receiver technology as a tool for scientific purposes. Research topics such as space weather [1], GNSS reflectometry for earth observation, GNSS radio occultation [2], ionospheric scintillation monitoring and mitigation [3], earthquake and tsunami prediction [4], precise timing or precision farming [5], to name a few, often require non-standard features from GNSS receivers, as well as a complete description of the signal processing applied from the antenna up to the computation of the desired GNSS product, which could be observables, position, timing, or any other measure from intermediate processing stages, such as the output of the acquisition or tracking stages.

However, researchers need to deal with an increasing complexity and integration level of GNSS integrated circuits (ICs), resulting in the practical impossibility of having an exact model on how the desired measurements were obtained, together with blocked access to modify or even inspect any internal aspect of the receiver. If the required measurement is already provided by an existing manufactured receiver, one can resort to statistical characterization and fit models upon it. If the required measurement or process is not implemented by an existing IC, researchers face the complex and costly task of building a brand new receiver from scratch, which is often far from their field of knowledge or subject of interest.

Software-defined receivers have a longstanding recognition to be the technology of choice when a researcher faces the need of obtaining GNSS products that are not already delivered by commercially available IC-based receivers. This popularity is driven by the notion that replacing hardwired, closed circuits by lines of source code provides researchers with unlimited capability to change and customize any aspect of the receiver, from the overall architecture to the selection of algorithms and the fine-tuning, and thus they are empowered to do anything they need. However, it has been widely recognized that scientific computing requires the observance of a set of best practices, which should be considered from the inception of the research activity, in order to permit the

effective communications not only of the research results, but the tools and data required to verify and extend the generated knowledge [6]–[9].

Commercial software-defined GNSS receivers use to contain proprietary code, distributed in binary form and with no possible inspection by a researcher external to the institution that developed the software. Those programs use to provide an extensive application programming interface (API) that allows for fine adjustment of receiver parameters, and some of them even permit user-defined algorithms for certain processing functions. However, they still can be considered black boxes in which we can inject some stimuli (that is, the complete receiver configuration and the stream of signal samples to be processed) and obtain some outputs (the GNSS products, such as observables and navigation data, and possibly other predefined performance indicators), but only with partial information about the actual signal processing taking place and limited access for modification.

This problem is overcome by software-defined GNSS receivers released under Free and Open Source Software (FOSS) licenses, which allow researchers to share the actual source code and provide users with an explicit permission to modify it. Users can download the source code, make modifications at their wish, create an executable upon it in their own machine, and run any kind of experiment.

While FOSS licenses allow for the possibility of the full inspection of every single step in the signal processing chain, from the sample stream ingestion on the computer platform up to the obtention of the desired GNSS signal products, and its arbitrary modification, sharing the code still does not ensure some basic requirements for any scientific research: *reproducible* experiments and the practical possibility of modifying existing code. These features require *i*) a careful design of the software architecture, allowing for arbitrary expansion, and where testability has been taken into account from the scratch; *ii*) the possibility to interact with many different radio-frequency front-ends, sample formats, data collection topologies and processing platforms; *iii*) comprehensive and updated documentation; public fora to contact other users and developers, or to get professional assistance; and *iv*) possibility to extract measurements in standard formats in order to chain the receiver outputs to other existing tools. This paper discusses aspects of software-defined GNSS receivers required for scientific experimentation, based on the experience and lessons learned by the authors (as developers and scientific users of GNSS-SDR [10], an open source, software-defined GNSS receiver), building on the work presented in [11] and [12] and exposed for public discussion at [13], and expanding it to more specifically scientific-related needs.

The paper is organized as follows. Next section discusses reproducibility of research results in the context of GNSS signal processing from a software-defined receivers' perspective. Then, Section III introduces the concept of *continuous reproducibility*, a methodology which allows the automated execution of experiments and provides an effective way to scientific reporting and collaboration, while contributing to mitigate the effects of software aging. A case study is presented in Section IV, discussing the features related to reproducibility of an open source software-defined GNSS receiver and presenting a working example for the topics discussed in the previous sections. Finally, Section V concludes the paper.

## II. REPRODUCIBILITY

An experiment involving a software-defined GNSS receiver is an experiment that occurs in a computer system. It is well-known that today's computational environments are complex, and accounting for all the possible effects of changes within and across systems is a challenging task [7], [14]–[16]. In computer systems research, an experiment is defined by the workload, the specific system where the workload runs, and the results from a particular execution. A key aspect in order to obtain meaningful conclusions from the experiments is *reproducibility*, which refers to the ability of an entire experiment or study to be reproduced, either by the researcher or by someone else working independently. It is one of the main principles of the scientific method and relies on *ceteris paribus* (other things being equal). Publication of scientific theories, including experimental and observational data on which they are based, permits others to scrutinize them, to replicate experiments, identify errors, to support, reject or refine theories, and to reuse data for further understanding and knowledge. Facilitating sustained and rigorous analysis of evidence and theory is the most rigorous form of peer review, and contributes to science's powerful capacity for self-correction [17].

As described in [18], this feature can be classified into *workload reproducibility* (which requires access to the original code and the particular workload that was used to obtain the original experimental results); *system reproducibility* (which requires access to hardware and software resources that resemble the original dependencies, including the set of hardware devices involved in the experiment such as the antenna, the radio frequency front-end, the bus connection to the host system, the specific processor model, possible computing off-loading devices and network elements, the system configuration, and the entire software stack, from the firmware/kernel and linked libraries versions used in the experiment) and *results reproducibility* (the degree to which the results of the re-execution of an experiment are valid with respect to the original). It follows a discussion of those aspects from the perspective of experiments in which software-defined GNSS receivers are involved.

### A. WORKLOAD REPRODUCIBILITY

In the context of this paper, workload reproducibility refers to the availability of the source code implementing the GNSS receiver, and either a real-time stream of incoming digital signal samples delivered by a radio frequency front-end, or those samples stored in files with a given format and data topology.

Obtaining the exact source code version in which the original experiment was performed is not always straightforward, since the source code is inherently dynamic.

Bug fixes, improvement and the addition of new features to the code base are frequent, so it is important to have unique identifiers for each source code snapshot. This is solved by version control systems such as Git [19], [20], which has become a standard *de facto* for code sharing due to its support for distributed and non-linear workflows, and its associated hosting services such as GitHub or BitBucket. Git assigns a unique identifier to every repository change (called ''commits'') obtained by a cryptographic SHA-1 hash function of the whole source tree of the commit (not just the changes), the parent commit SHA-1 hash function, the author and committer information (date, name and email), and the commit message. Hence, a researcher can refer to a particular code snapshot and others can retrieve an exact copy of it. This includes the complete description of the receiver's configuration used in the experiment. For significant code releases, another good practice is to assign a digital object identifier (DOI) to them for easier citation.

When applied to software engineering, reproducibility has other additional implications such as in security (*i.e.*, gaining confidence that a distributed binary code is indeed coming from a given verified source code). A build is reproducible if given the same source code, build environment and build instructions, any party can recreate bit-by-bit identical copies of all specified artifacts. In the open source community there are well-established good practices (see [21]) which create a verifiable path from human readable source code to the binary code used by computers. This includes [22]: *i*) the build system needs to be made entirely deterministic: transforming a given source must always create the same result; *ii*) the set of tools used to perform the build and more generally the build environment should either be recorded or predefined; and *iii*) users should be given a way to recreate a close enough build environment, perform the build process, and verify that the output matches the original build.

The other component of the workload is the stream of sampled GNSS signals feeding the software receiver. If the original experiment reported results obtained in real-time with live GNSS signals, using a radio frequency front-end to feed the software receiver input, the exact conditions of the experiment are impossible to reproduce. This can be solved by storing the sample stream in files that can then be read and processed by the software receiver. Storing the samples delivered by the radio frequency front-end ensures that the experiment will be reproducible by others in the future. Although it still poses some technical challenges, such as the availability of high-bandwidth storage systems, storage capacity and means of sharing such large amount of data (as an example: storing L1 signals with a bandwidth of 20 MHz and with 16 bits per sample during 6 months takes more than 1 petabyte of data), it is becoming a trending practice in scientific experiments with software-defined GNSS receivers. In addition to allow reproducibility, sample storage would also allow in the future for corroboration or rebuttal of the obtained conclusions by other yet-to-be-invented methods.

**TABLE 1. Fundamental data collection topologies, as defined by the ION GNSS SDR Standard Working Group [23].**

| Data Topology |
|---|
| Single band, single-stream, single file. |
| Multi-band, single-stream, single file. |
| Multi-stream, single file. |
| Multi-sensor, single file. |
| Temporal splitting of files. |
| Spatial splitting of files. |
| Spatial-temporal splitting. |

There is still no well-established standard for GNSS signals sample storage. A relevant effort is being done by the ION GNSS SDR Standard Working Group, which defines some fundamental data collection topologies for raw GNSS, and possibly other sensors, data stored digitally (reproduced in Table 1), sample resolution (including 1, 2, 4, 8, 16, 32 or 64 bits per sample), encoding (sign, sign-magnitude, signed integer, offset binary or floating-point), sampling frequency, possible intermediate frequency and inverted spectrum indicator. The mentioned Working Group is proposing a metadata standard that defines parameters and a formal XML schema to express the contents of GNSS sampled data files [23], allowing researchers to share the file(s) containing the data and a metadata file containing the format description of such data.

Finally, it is recommended to include a description of the location and date in which the GNSS signals were captured, including the type of receiver according to its antenna dynamics (see Receiver Independent Exchange Format standard, version 3.03 [24, §5.3] for a possible classification), and description of surroundings as needed (nearby buildings and other scatterers, possible presence of interference sources, and any other information considered relevant for the interpretation of the results). A 360-degree picture taken from the antenna location for static receivers, or a 360-degree, time-tagged video for moving platforms could be informative in certain scenarios.

### B. SYSTEM REPRODUCIBILITY

System reproducibility requires the full description of the hardware and software resources that resemble the original experimental setup, including the set of hardware devices involved in the experiment such as the antenna, the radio frequency front-end, specific CPU models, possible computing off-loading devices (such as GPUs or FPGAs) and network elements, system configuration, as well as the entire software stack from the firmware/kernel up to the libraries used by the experiment.

#### 1) HARDWARE DESCRIPTION

The antenna should be described by its manufacturer, identification number and type. In case of multiple antennas, its geometrical arrangement must be provided. Other relevant data is the average antenna phase center relative to the antenna reference point (ARP) for each specific frequency

band and satellite system, the orientation of the antenna zero-direction as well as the direction of its vertical axis (bore-sight), if mounted tilted on a fixed station, or XYZ vector in a body-fixed system, in case of mounted on a moving platform (all units in meters). If the antenna is physically apart from the front-end the cable category and length, as well as the connectors type, should be reported.

In case of using a signal generator instead of live GNSS signals, its brand and model (or version if it is a software-defined generator), as well as the complete set of configuration parameters should be included in the experiment description.

Regarding the description of the radio-frequency front-end originally used to capture the GNSS sampled signals used in the experiment, it should include as many details as possible about its electrical features. There are many commercial off-the-shelf solutions in the market that act as radio frequency front-ends for GNSS signals. Some of them even share the schematics, bill of materials, printed circuit board layout data, and everything that is necessary for its manufacture, provided the access to the required fabrication tools, materials and components. Notable examples are the HackRF board by Great Scott Gadgets, which design is freely available in a git repository, and the BladeRF board by Nuand, which design is also freely available. Such physical artifacts are usually referred to as open-source hardware, and belong to a broader paradigm known as Open Design [25], [26], defined as design artifact projects whose source documentation is made publicly available so that anyone can study, modify, distribute, make, prototype and sell artifacts based on those designs. The artifact's source, the design documentation from which it is made, is available in the preferred format for making modifications to it. Ideally (but not exclusively necessary), Open Design uses readily-available components and materials, standard processes, open infrastructure, unrestricted content, and open-source design tools to maximize the ability of individuals to make and use reproducible hardware.

If custom modifications were made to a commercially available front-end (for instance, replacing and/or disciplining the shipped local oscillator with a more stable clock), those modifications should be also clearly described.

However, having the schematics of a printed circuit board for an RF front-end, or the VHDL model of the FPGA firmware, still does not ensure reproducibility due to *place and route*, a stage in the design of printed circuit boards, integrated circuits, and FPGAs. The first step, placement, involves deciding where to place all electronic components, circuitry, and logic elements in a generally limited amount of space. It assigns logic blocks to specific chip/board locations, trying to minimize the routing distance and therefore allowing successful routing. The second step, routing, decides the exact design of all the wires needed to connect the placed components, optimizing the delay of critical signals. Those operations are usually performed by electronic design automation (EDA) tools, which implement algorithms such as simulated annealing or similar for placing, and pathfinder for routing [27]. Those approaches are heuristic, randomly initialized, and their results are non-deterministic. This means that every run of the route and place process could end up with a different distribution of components in the FPGA or printed circuit board. Although the results are guaranteed to meet the tolerance ranges defined by the designer, this feature poses an extra challenge to reproducibility for printed circuit boards defined by their schematic and for FPGA firmwares defined by their VHDL model.

Even when code and data are shared, it remains difficult to reproduce results due to differing underlying computing platform that is executing the software receiver: processor architecture, memory, storage and communication speed within the different components may vary in different machines. In an experiment report, it is recommended to annotate the processor architecture (*e.g.*, i386, x86_64/amd64, armhf or arm64), manufacturer and type; the available RAM memory and, when relevant, the storage capacity. If computing off-loading devices were used (such as FPGAs or GPUs), its vendor and model should be also specified.

#### 2) SOFTWARE STACK DESCRIPTION

Most software applications, as is the case of software-defined GNSS receivers, do not rely exclusively on their source code. They require features provided by the underlying operating system (for example, Windows, macOS or Linux) as well as a set of supporting libraries (either required or optional) that are called from the main program. This poses a problem to reproducibility, since a researcher replicating the original experiment should install exactly the same version of the operating system, apply the same software upgrades, use the same versions for all the dependency libraries, and the same version of the kernel and other required firmware (that is, the whole software stack that is supporting the main program) than in the original setup. Even if the complete list of dependencies and versions were reported, reproducing the same combination in another machine is not straightforward. While software package managing tools use to ease the upgrading of software components to get the most recent release, they in general do not allow rewinding the system to old versions. In addition, different operating systems (and versions thereof) may require different installation and configuration steps.

A possible solution for researchers is to share not only the source code of the main program and the input data but the whole software stack, including the entire operating system and all software, scripts, code and data necessary to execute the experiment [28]. This approach is known as software virtualization, which consists of an encapsulation of the whole software stack that can be executed on practically any desktop, laptop or server, irrespectively of the main (''host'') operating system installed on the computer that is actually executing it.

A virtualized software application is a program that can be executed regardless the underlying computer platform (*i.e.*, processor architecture, operating system and installed

library versions) that is executing it. This can be achieved by packaging the application and all its software requirements (the operating system and all the application-required supporting libraries and programs) in a single, self-contained and isolated software entity, that can be then run on any platform. Hence, for instance, using virtualization tools a complete Windows system can be run on a Linux machine, or on another version of Windows. An instance of a software-defined GNSS receiver executed in a virtual environment can then be called a *virtualized GNSS receiver* [29]. This is a very convenient strategy for sharing the full software stack as well as data and all the scripts required to reproduce the plots from the original paper. There are two main approaches to software virtualization: virtual machines and software containers.

A virtual machine (VM) is a software-based environment designed to simulate a hardware-based environment, for the sake of the applications it will host. A VM emulates a computer architecture and provides the functionality of a physical computer. Within each virtual machine runs a full operating system, so conventional software applications expecting to be managed by an operating system and executed by a set of processor cores (*e.g.*, a software-defined GNSS receiver) can run within a VM without any required change. In addition, the researcher has full control over the virtual (''guest'') operating system, and thus can install software, examine scripts and code, and modify configuration settings as necessary.

Recently, however, software containers are replacing VMs as the preferred supporting software stack system for virtualized software applications because of the faster and more lightweight nature of the former. An application running in a container can be more efficient in making use of the underlying hardware than when it is executed on a VM (since it operates directly with the real processing units instead of against an emulated layer, avoiding its overhead [30]), and many more containers than VMs can be put onto a single server, thus optimizing the investment in compute resources. The concept of containerization was originally developed as a mechanism to segregate namespaces in a Linux operating system for security purposes, isolating process groups (a process and possible descendant processes) from the outside world. The first approach consisted of producing partitions (sometimes called ''jails'') within which applications of questionable security or authenticity could be executed without risk to the kernel. The kernel was still responsible for execution, though a layer of abstraction was inserted between the kernel and the workload. Once the environment within these partitions was minimized for efficiency's sake, the concept expanded to make the contents of those partitions portable. Hence, this technology can be seen as an advanced implementation of the standard chroot mechanism in UNIX-like systems. The first container system was Linux Containers (LXC), followed by a container hypervisor (LXD), and then by other projects such as Docker or Ubuntu Snaps. These latter systems provide native environments with no hypervisor but a daemon that supplements the host kernel and that maintains the compartmentalization between containers, while connecting the kernel to their workloads.

In [31], Beaulieu-Jones and Greene proposed a continuous analysis process combining Docker containers with continuous integration, a popular software development technique consisting of the verification of each new commit to the source code by an automated build, to automatically re-run computational analysis whenever relevant changes are made to the source code. This allows results to be reproduced quickly, accurately and without needing to contact the original authors. It also provides an audit trail for analyses that use data with sharing restrictions.

### C. RESULTS REPRODUCIBILITY

In computer science, a deterministic algorithm is an algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states. If the software-defined receiver is implemented as a single-threaded program (for instance, as a MATLAB script with no parallelization), where the instructions are always called in the same order, without using external states other than the input, operates in a way that is no timing-sensitive and without unexpected hardware errors, the program can be deterministic, which offers many benefits for debugging, fault tolerance, and security, in addition to make reproducibility easier.

However, the computational load required by software-defined GNSS receivers, and their inherently parallelizable architecture design (e.g., many channels performing very similar operations over the same input sample stream), drive to concurrent programming in order to exploit the full capacity of the underlying processors and to improve the processing efficiency. Parallelism poses many correctness challenges, from dealing with concurrency errors like data races, atomicity violations, and ordering violations, to coping with the nondeterminism inherent in most parallel systems. In such cases, a thread scheduler can play a key role in achieving efficiency (that is, real-time processing) while avoiding problems related to nondeterministic executions. It is then relevant to analyze *run-to-run* reproducibility.

Software-defined receivers can be formally represented as flow graph of nodes. Each node represents a signal processing block, whereas links between nodes represents a flow of data. The concept of a flow graph can be viewed as an acyclic directional graph with one or more source blocks (to insert samples into the flow graph), one or more sink blocks (to terminate or export samples from the flow graph), and any signal processing blocks in between. Those flow graph computations can be jointly modeled as a Kahn process [32], [33]. A Kahn process describes a model of computation where processes are connected by communication channels to form a network. Processes produce data elements or tokens and send them along a communication channel where they are consumed by the waiting destination process. Communication channels are the only method processes may use to exchange information. Kahn requires the execution of a process to be

suspended when it attempts to get data from an empty input channel. A process may not, for example, test an input for the presence or absence of data. At any given point, a process can be either enabled or blocked waiting for data on only one of its input channels: it cannot wait for data from more than one channel.

Systems that obey Kahn's mathematical model are determinate: the history of tokens produced on the communication channels does not depend on the execution order [32]. With a proper scheduling policy, it is possible to implement software defined radio process networks holding two key properties: *i*) *non-termination*: understood as an infinite running flow graph process without deadlocks situations, and *ii*) *strictly bounded*: the number of data elements buffered on the communication channels remains bounded for all possible execution orders. An analysis of such process networks scheduling was provided in [34].

An open source example of a runtime scheduler fulfilling the requirements described in [34] is found in GNU Radio, a software framework for programming software-radio applications. A detailed description of such scheduler implementation (memory management, requirement computations, and other related algorithms and parameters) can be found in [35]. Under this scheme, software-defined signal processing blocks read the available samples in their input memory buffer(s), process them as fast as they can, and place the result in the corresponding output memory buffer(s), each of them being executed in its own, independent thread. This strategy results in a software receiver that always attempts to process signal at the maximum processing capacity, since each block in the flow graph runs as fast as the processor, data flow and buffer space allows, regardless of its input data rate, while allowing for a determinate system. However, even using a deterministic scheduler, it still does not ensure the exact replication of results from two different runs. Many factors can affect the obtained numerical results, most notably *i*) the workload and resources of the machine running the experiment; *ii*) the receiver's *a priori* knowledge of time, ephemeris, almanac and rough position; *iii*) the availability to data external to the receiver (as is the case of DGNSS and A-GNSS solutions, or the combination with other sensors); and *iv*) the use of other programming practices that violate Kahn's model and result in a nondeterminate system (*e.g.*, use of shared variables to circumvent the communication channels).

Another factor that has an impact on GNSS-related experiments' results is satellites' geometry. In order to make the measurement as independent as possible of this effect, measurements should be spread in an interval of 8 hours (see [36]).

Hence, for the all the reasons described along this Section, getting exactly the same numerical results in two different experiments can be extremely difficult, if not impossible, to achieve. In such cases, it makes sense to define equivalence criteria for different result sets. A possible tool for the assessment of equivalence between the outputs of different experiments are equivalence tests, a variation of hypothesis tests used to draw statistical inferences from observed data. In equivalence tests, the null hypothesis is defined as an effect large enough to be deemed interesting, specified by an equivalence bound. The alternative hypothesis is any effect that is less extreme than said equivalence bound. The observed data is statistically compared against the equivalence bounds. Examples are the two sample t-test and the two one-sided t-test (TOST) [37].

## III. CONTINUOUS REPRODUCIBILITY

A software container provides a snapshot of a full software stack, adequately packaged and ready to be run in another computer. While this is a convenient feature for reproducibility, it also clenches the results to the possible bugs present in that particular version of the research source code, the linked libraries, other used executables and the OS. Ideally, an experiment should be reproducible not only in the very specific computing environment used by the original researcher but in a broad range of platforms, even in those yet-to-be-released at the time the experiment was conducted for the first time. In other words, it is desirable to achieve long-term reproducibility. Here, ''long-term'' must be framed in the context of Software Engineering, in which changes happen at a rapid pace. For instance, Ubuntu (one of the most popular GNU/Linux distributions) releases a new version of the OS every 6 months. Other distributions such as Arch Linux and Gentoo Linux are rolling release systems, making packages available to the distribution a short time (days or weeks) after they are released upstream. As a consequence, software environments are constantly mutating.

In 2008, some popular Operating Systems were Windows Vista, Ubuntu 8.04 LTS and Mac OS X 10.5 Leopard. One decade later, those OS releases have reached its end of life, and users are no longer receiving new security and maintenance updates. Although the reproduction of such environments in a software container is still possible, the resulting software stack contains known bugs and could compromise safety, reliability and compatibility when executed in a modern machine. On the other hand, trying to build source code written in 2008 in a 2018 environment is likely to fail due to the API-breaking features introduced in OSs, compilers and libraries. This implies that maintaining long-term reproducibility requires changes both in the source code of interest and in the surrounding software environment.

This issue can be addressed with Continuous Integration, a concept firstly introduced in [38] that consists of automating the build and testing of code every time a researcher commits changes to the version control system [39]–[41]. Continuous Integration encourages developers to share their code and unit tests by merging their changes into a shared version control repository after every small task completion. Committing code triggers an automated build system to grab the latest code from the shared repository and to build, test, and validate the obtained numerical results.

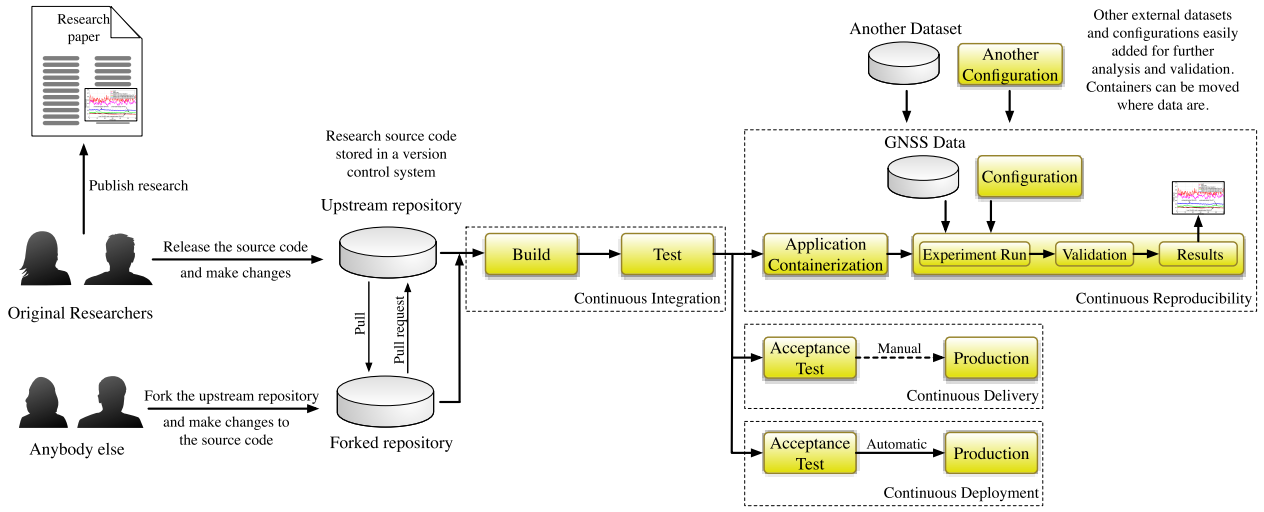


**FIGURE 1. Continuous Reproducibility in the context of other practices such as Continuous Integration, Continuous Delivery and Continuous Deployment.**

For instance, it is possible to set up a Continuous Integration system which makes use of different OSs and their respective versions. Then, every time a new commit is done in the source code tree, the system automatically builds the code and performs the numerical experiment in each of the targeted OS versions. This helps to identify integration and backward/forward compatibility problems as soon as the offending instruction is committed or a new OS appears, so they are easier to fix, and ensures that research code performs as expected in different environments.

The combination of software containerization technology and Continuous Integration systems provides a very flexible, affordable, automated and audited path from each change in the source code tree to the regeneration of the experiment results in a wide range of computing systems. If the scripts required for that automation are made public, anyone external to the original research team can reproduce the entire experiment, make further validations or corrections, and, most importantly, make changes and build upon it.

Leveraging on Continuous Integration, other practices have extended within the software development industry. The organizations' need to frequently and reliably release new features and products has bred the proliferation of Continuous Delivery procedures, aimed at ensuring an application is always at production-ready state after successfully passing automated tests and quality checks. Continuous Deployment practices go a step further and automatically and continuously deploy the application to production or customer environments [42]. A toolchain implementing Continuous Integration, Continuous Delivery and Continuous Deployment is shown in Figure 1.

Those practices and tools can be adapted and applied to the context of research communication in GNSS Signal Processing, enabling effective *Continuous Reproducibility* of the research results. This includes, in addition of releasing the source code under a FOSS license, to share the data set on which the experiments were done (usually, files containing raw GNSS signal samples and possibly other sensors), the full experiment configuration, and the scripts used to analyze the results and generate the figures and tables appearing in the research paper in which the experiment was presented.

The proposed steps performed by a Continuous Reproducibility system each time a new change is committed to the source code repository can be described as a pipeline of automated stages:

1) **Build:** This step can be iterated over a list of supported OS, versions and processor architectures.
   - Install an image of a given OS, with specific brand, version (or snapshot in case of rolling releases) and processor architecture.
   - Install all the required software dependencies.
   - Grab the research code from the upstream repository.
   - In case of compilable code, configure, build and install the required executables.
2) **Test:** Execute Quality Assessment (QA) executables (unit, integration and system testing of the published software). Again, this step can be iterated over a list of supported OS, versions and processor architectures.
3) **Create a software container:**
   - Generation of a software image (*e.g.*, a Docker container) with a snapshot of the full software stack, including other analysis and graphical representation tools required for the generation of the figures appearing in the research paper.
   - Tagging and publication of the newly created container image.
4) **Reproduce the experiment:**
   - Install the image of the container created in the previous step.

- Grab a data set (files or stream containing raw GNSS signal samples).
- Execute the experiment.
- Validation of results: Check whether the numerical results of this instance of the experiment are significantly worse, comparable or better to those of the original experiment, in some defined metric.
- Produce experiment results:
  - Generate the figure(s) that appeared in the research paper with the newly obtained results.
  - Post the resulting figures/numerical results online.

This proposed pipeline is shown in Figure 1. Each stage can have multiple parallel jobs, but the execution is sequential from stage to stage. Each job (usually consisting of a script to be run by a Unix command-line interpreter) is executed in an isolated computer environment. The list of tested OS can be expanded a soon as a new release appears, thus detecting forward compatibility problems as early as possible. If the execution fails at some step, the pipeline stops and a full report is sent to the original research team or any other user triggering the process. The automation of the full process ensures that anybody can reproduce the experiment without need of contacting the original authors, and to experiment with new changes with a rapid evaluation of the results in fair conditions.

The generation of a software container for each particular execution is specially interesting in case of experiments using large data sets, in which moving the container to the physical computing system storing the data can be cheaper than transmitting such data over a communication network, or in cases where data sets have not been publicly released.

Nowadays, most commercial Continuous Integration/ Continuous Delivery platforms offer their services for free to open source projects. There are also high-quality open source tools such as Jenkins, Travis CI, Buildboot and GitLab, among others [43]. Most of them can easily accommodate a Continuous Reproducibility system such as the one described above. If the pipeline is deployed in a server with public access, anyone can reproduce the experiment, make changes and apply the pipeline to other data sets (public or private). It is also possible to run all or some of the jobs at users' premises, thus providing an audit trail for private implementations and data sets.

For the original researcher team the initial investment is not negligible, although beneficial in many aspects. Setting up a Continuous Reproducibility system requires an effort in getting familiar with the related tools and in developing the automation scripts that allow the execution of the experiment and the production of the results with no human intervention. In addition, exposing the implementation of a newly invented algorithm to public scrutiny can be problematic in terms of intellectual property rights, or by other factors such as the reluctance to publish poor-written code, to attend people demanding support and bug fixes, and to share implementation details with possible competitors [44].

As a counterpart, adopting such practices redounds in improved credibility, usability and perceived quality of the research communication [45], while effectively contributing to extend software lifetime and to mitigate the effects of software aging [46]. Last but not least, this methodology implements the automation of continuous improvement cycles, which constitutes an approach that is aligned with the scientific method [47] and allows others to build upon it in the terms established by the source code license.

## IV. CASE STUDY: GNSS-SDR

Reproducibility requirements have an impact in how a software-defined GNSS receiver is designed, implemented, documented and shared. While strictly speaking reproducibility is only possible with exactly the same setup as in the original experiment, in practice it is desirable to allow the reproduction in the widest range of systems as possible, in order to make possible such replication to researchers with access to other kind of equipment (for instance, a computer with different features, operating system or library versions). This section describes some of the design choices and features related to reproducibility taken in GNSS-SDR, an open source project that implements a GNSS software-defined receiver. With no claim of optimality, this is aimed to provide a working example for the topics discussed in Section II and Section III.

- **License:** The source code is released under the GNU General Public License version 3. Some specific files are released under other compatible licenses. All those licenses provide users with the freedom to run the program as they wish, for any purpose; the freedom to study how the program works, and to make modifications at their wish; and the freedom to redistribute copies (and the modified versions) to others.
- **Version control system:** The source code is kept under a version control system by Git, a free and open source tool that automates the process of keeping an annotated history of the project, allowing reversion of code changes, easy branching and merging, sharing and change tracking, and the coordination of work on those files among multiple people in a distributed fashion.
- **Branching model:** GNSS-SDR's reference Git repository, also referred to as ''upstream'', is hosted online by GitHub (see https://github.com/gnss-sdr/gnss-sdr), a web-based Git repository hosting service that offers all the distributed revision control and source code management functionality of Git as well as adding its own features, such as review changes, comment on lines of code, report issues, and plan the evolution of the project with discussion tools, all in a rich web-based graphical interface. In this repository there are two development branches with infinite lifetime:
  - `master` is the main, most stable branch that contains the latest software release plus some occasional, portability-related bug fixes, and

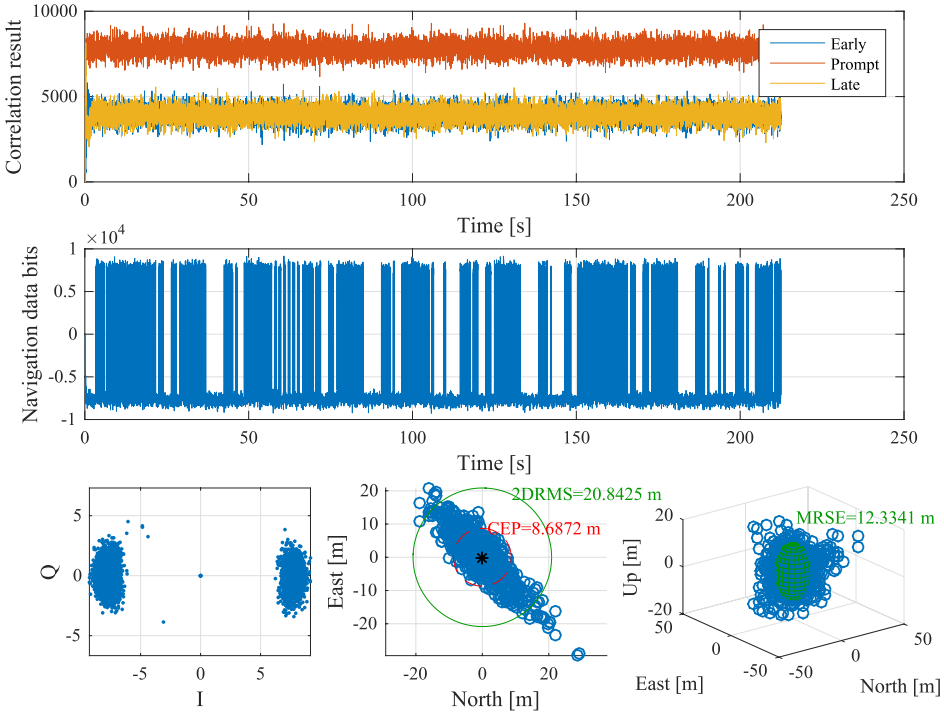


**FIGURE 2. Results of GPS L2C signal processing with GNSS-SDR. Figure taken as example for the Continuous Reproducibility demonstration.**

  - `next` is where all the code development is happening, containing the most updated code that will eventually form part of the next stable release.

  In addition, the repository offers source code releases packet in a single file and its digital signature, to make easier their distribution and future reference.
- **Digital Object Identifier:** A DOI is issued by Zenodo.org for each software release (*e.g.*, [48]).
- **Build system:** In order to address reproducibility of the source code building rules (compiler flags, linking, etc.), GNSS-SDR uses CMake as its building system generator. CMake is used to control the software compilation process using platform and compiler independent configuration files, from which it generates native *makefiles* and *workspaces* that can be used in a wide range of compiler environments. Popular open source build automation tools, such as Make and Ninja, can be used to automatically build the required executable programs and libraries from the source code with the aid of the building files generated by CMake. Using popular, widely available cross-platform tools helps to ensure portability among different systems and processor architectures.
- **Compilers:** The source code can be compiled with the two most popular open source compilers, GCC and LLVM/Clang. In general, it is desirable to be able to build the source code with different compilers, since it improves the overall quality of code by providing different checks and alerts.
- **Programming standards:** GNSS-SDR's source code honors the C++11 and C++14 standards [49], thus ensuring the validity of the source code in a wide range of processing platforms and compilers for a long time span (in computer engineering's time scale).
- **Packaging:** GNSS-SDR was accepted as a software package by the Debian Project, and it is available in their most recent releases. This secures software availability in a wide list of other popular GNU/Linux distributions (*e.g.*, Ubuntu) and ensures correctness in terms of licensing and the availability of software dependencies in a large list of processor architectures. A Macports package is also available for macOS.
- **Containerization:** A "Dockerfile" example for the generation of Docker container images with a working executable of GNSS-SDR is available at https://github.com/carlesfernandez/docker-gnsssdr. Docker is an open source tool designed to make it easier to create, deploy, and run applications by using software containers. Docker containers wrap a piece of software in a complete filesystem that contains everything needed to run: code, runtime, system tools and system libraries, and ship it all out as one package. This guarantees that the software will always run the same, regardless of any customized settings that the executing machine might have that could differ from the machine used for writing and testing the code.
- **Documentation:** Project website at http://gnss-sdr.org, containing building instructions, tutorials, and the

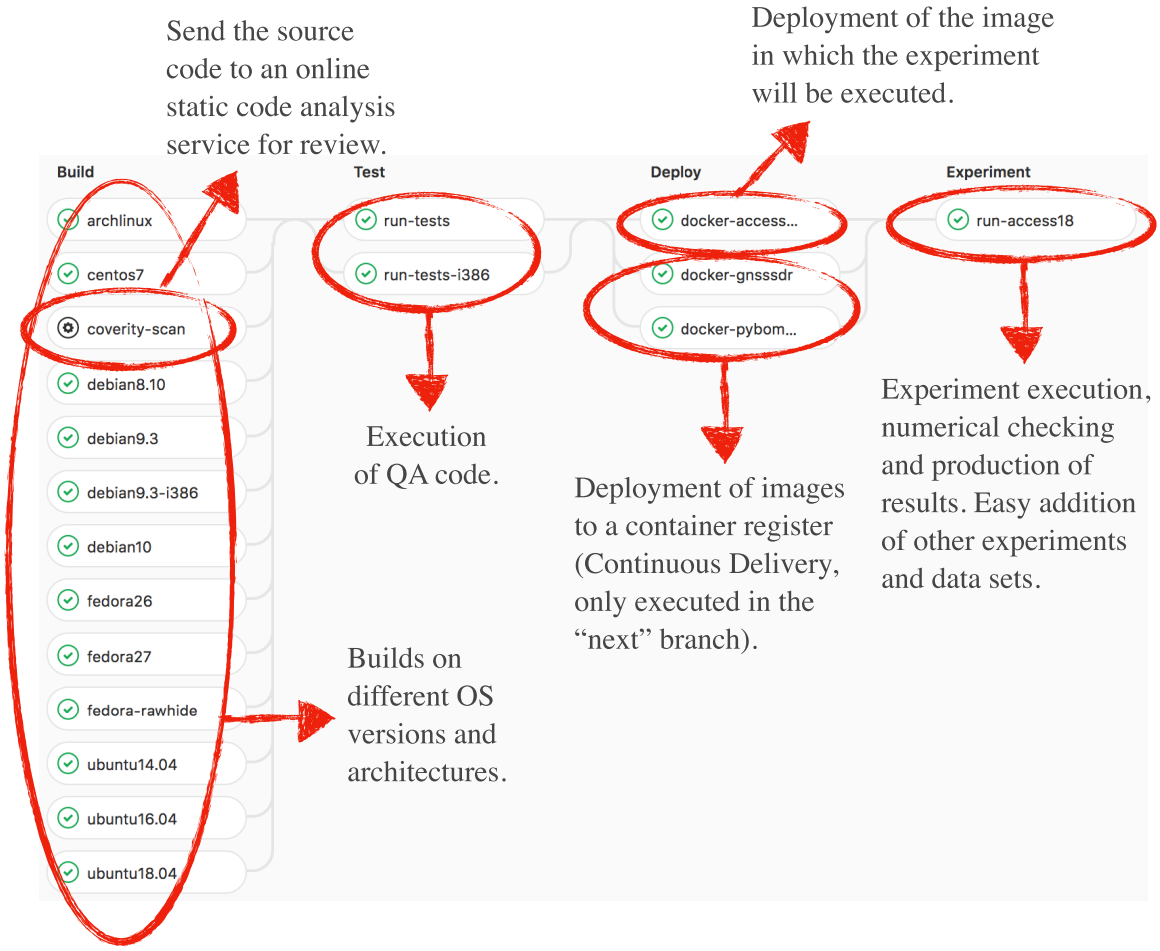


**FIGURE 3. Continuous Reproducibility pipeline for GNSS-SDR, as implemented on GitLab. Publicly available at https://gitlab.com/gnss-sdr/gnss-sdr.**

documentation about the receiver's configuration. The website source content is available as a Git repository at https://github.com/gnss-sdr/geniuss-place.

- **Interoperability:** GNSS signal products delivered in standard output formats such as RINEX files (navigation and observables) in versions 2.11 and 3.03, and RTCM v3.2 messages, with configurable rates.
- **Configuration:** Full receiver configuration in a single text file, allowing easy exchange and reference of receiver's full configuration.
- **Scalability:** Modular design and runtime scheduler inherited from GNU Radio (see https://www.gnuradio.org), a well-established open source software framework for software-defined radio applications.

In order to demonstrate the feasibility of a Continuous Reproducibility system as proposed in Section III, we propose an example implemented in GitLab, a web-based Git repository manager. The experiment considered for this example consists of processing raw signal samples captured with a radio frequency front-end. The input signal is a raw data collection in the L2 band carried out at Fraunhofer IIS, Nuremberg (Germany), with a Flexiband receiver [50] and Spirent GSS8000 signal generator. The data set is available online [51]. The results of processing that signal with GNSS-SDR configured as a GPS L2C receiver in a cold start are shown in Figure 2. The receiver was able to get position fixes in single point positioning mode, with metrics given in circular error probability (CEP, the radius of circle centered at mean of the obtained values, containing the horizontal position estimate with probability of 50%, in m) and twice the distance root mean square (2DRMS, containing the horizontal position estimate with probability of 95%) for 2D positioning, and the mean radial spherical error (MRSE, containing the three-dimensional position estimate with probability of 61%, in m) for 3D positioning (center and right bottom plots in Figure 2).

The implementation of the proposed Continuous Reproducibility system in GitLab consists of the creation of a file at the root folder, named ".gitlab-ci.yml", that defines the automated pipeline. The process consists of four stages, each one populated with one or more jobs, which can be mandatory, optional (manually triggered) or "allowed to fail". Each job consists of a set of commands executed in an independent virtual computer environment.

```
# Download and run the software container
docker pull carlesfernandez/docker-gnsssdr:access18
docker run -it -v $PWD/access18:/home/access18 carlesfernandez/docker-gnsssdr:access18

# Download receiver's configuration, experiment scripts and GNSS data
git clone https://github.com/gnss-sdr/gnss-sdr
cd gnss-sdr
git checkout next
mkdir -p exp-access18/data
cd exp-access18/data
curl https://zenodo.org/record/1184601/files/L2_signal_samples.tar.xz --output L2_signal_samples.tar.xz
tar xvfJ L2_signal_samples.tar.xz
echo "3a04c1eeb970776bb77f5e3b7eaff2df  L2_signal_samples.tar.xz" > data.md5
md5sum -c data.md5
cd ..
cp ../src/utils/reproducibility/ieee-access18/L2-access18.conf .
cp ../src/utils/reproducibility/ieee-access18/plot_dump.m .
cp -r ../src/utils/matlab/libs/geoFunctions .

# Run the experiment, generate figure and export it outside the container
gnss-sdr --c=L2-access18.conf
octave --no-gui plot_dump.m
epspdf Figure2.eps Figure2.pdf
cp Figure2.pdf /home/access18/
exit
```

**FIGURE 4.** **Terminal commands to reproduce Figure 2 of this paper in any machine with Docker installed and running.**

Stages are defined as follows:

1) **Build:** Multiple mandatory jobs building the source code in most popular GNU/Linux distributions (Arch Linux, CentOS, Debian, Fedora, Ubuntu), in different versions and architectures. Additional versions can be easily added as soon as they appear. An optional job performs a compilation with a specific compiler and sends the results to Coverity Scan, a static code analysis service which is free for FOSS projects and provides valuable feedback on code quality and identification of defects. Checking the building process is highly important in GNSS-SDR, which is written in C++ and links to other FOSS libraries which API change along time. This also holds in case of using scripting languages (such as Matlab/Octave or Python), where this step is recommended in order to check whether the code runs as expected in the different versions of the language interpreter and associated packages shipped with different OS.
2) **Test:** Set of jobs in charge of the execution in different environments of the quality assessment code, which consists of a set of noninteractive procedures evaluating assertions and leading to pass/fail decisions based upon some predefined requirements. This includes *unit tests*, checking certain functions and areas (or units) of the source code, and *system tests*, conducted on a complete, integrated system to evaluate the software receiver's requirements compliance.
3) **Deploy:** One job in this stage creates a software container image that is ready to execute the experiment. This includes an OS, all the required external software library dependencies and executables required to run GNSS-SDR, the scripts required to execute the software receiver over a data set and to generate the figures and tables of the research paper (in this example, Figure 2), and all the graphical representation tools and packages required to generate it (in this case, Octave and some LaTeX–related packages for the generation of files in PDF format). More jobs creating images for other experiments with different requirements can be easily added. The data set itself is intentionally left out in this image, in order to ease the insertion of other data sets in the next stage. Two additional jobs implement a Continuous Delivery system for GNSS-SDR, and are only executed in the "next" branch. All jobs in this stage end publishing a tagged image in a public container register (in this case, https://hub.docker.com).
4) **Experiment:** A job installing the image created in the previous stage, grabbing the data set, executing the experiment and generating Figure 2 from the results. Other jobs executing different experiments can be easily added.

The process is shown in Figure 3, where each column represents a stage in the Continuous Reproducibility workflow. All the mandatory jobs in each stage must be completed successfully before starting the execution of next stage jobs. Each time a commit is pushed to any branch of the GitLab repository containing a file in the root folder named ".gitlab-ci.yml", the workflow is automatically triggered. If a mandatory job fails, the process is stopped and the user that triggered it receives an email with a report. If all the mandatory jobs in the pipeline end successfully, the job at the last stage will have reproduced a downloadable version of Figure 2.

The complete system is available online (see https://gitlab.com/gnss-sdr/gnss-sdr/pipelines), so the experiment can be readily reproduced by others in a fully transparent

procedure (see Fig. 4), and changes can be made in a traceable manner.

## V. CONCLUSIONS

This paper discussed the reproducibility of scientific experiments in which a software-defined GNSS receiver plays a role. While reproducibility is largely recognized as an essential feature of the scientific method, it is usual to find scientific papers that fail to provide enough information for the replication of the original experiment by other researchers. Those aspects were analyzed in terms of workload, system and results reproducibility, providing recommendations for reporting experiments in a way that can be reproduced by others. Then, leveraging on software containerization technologies and the best practices from professional software development, this paper proposed a methodology that allows for the automated execution of experiments, provides an effective way to scientific reporting and collaboration, and contributes to mitigate software aging by detecting forward compatibility problems as soon as possible. A practical example implemented in a popular online platform was also provided.

## REFERENCES


[1] X. Yue *et al.*, "Space weather observations by GNSS radio occultation: From FORMOSAT-3/COSMIC to FORMOSAT-7/COSMIC-2," *Space Weather*, vol. 12, no. 11, pp. 616–621, Nov. 2014, doi: 10.1002/2014SW001133.

[2] L. Lestarquit *et al.*, "Reflectometry with an open-source software GNSS receiver: Use case with carrier phase altimetry," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 9, no. 10, pp. 4843–4853, Oct. 2016, doi: 10.1109/JSTARS.2016.2568742.

[3] J. Vila-Valls, P. Closas, C. Fernandez-Prades, and J. T. Curran, "On the ionospheric scintillation mitigation in advanced GNSS receivers," *IEEE Trans. Aerosp. Electron. Syst.*, to be published, doi: 10.1109/TAES.2018.2798480.

[4] L. He and K. Heki, "Three-dimensional distribution of ionospheric anomalies prior to three large earthquakes in Chile," *Geophys. Res. Lett.*, vol. 43, no. 14, pp. 7287–7293, Jul. 2016, doi: 10.1002/2016GL069863.

[5] H. J. Heege, "Precision in guidance of farm machinery," in *Precision in Crop Farming*, H. J. Heege, Ed. Dordrecht, The Netherlands: Springer, 2013, ch. 4, pp. 35–50, doi: 10.1007/978-94-007-6760-7_4.

[6] G. K. Sandve, A. Nekrutenko, J. Taylor, and E. Hovig, "Ten simple rules for reproducible computational research," *PLoS Comput. Biol.*, vol. 9, no. 10, pp. 1–4, Oct. 2013, doi: 10.1371/journal.pcbi.1003285.

[7] V. Stodden and S. Miguez, "Best practices for computational science: Software infrastructure and environments for reproducible and extensible research," *J. Open Res. Softw.*, vol. 2, no. 1, pp. 1–6, 2014, doi: 10.5334/jors.ay.

[8] G. Wilson *et al.*, "Best practices for scientific computing," *PLoS Biol.*, vol. 12, no. 1, p. e1001745, Jan. 2014, doi: 10.1371/journal.pbio.1001745.

[9] G. Wilson, J. Bryan, K. Cranston, J. Kitzes, L. Nederbragt, and T. K. Teal, "Good enough practices in scientific computing," *PLoS Comput. Biol.*, vol. 13, no. 6, p. e1005510, Jun. 2017, doi: 10.1371/journal.pcbi.1005510.

[10] GNSS-SDR. (2017). *An Open Source Global Navigation Satellite Systems Software Defined Receiver*. Accessed: Apr. 2, 2018. [Online]. Available: http://gnss-sdr.org

[11] C. Fernández-Prades, J. Arribas, and P. Closas, "Assessment of software-defined GNSS receivers," in *Proc. 8th Ed. NAVITEC, ESA/ESTEC*, Noodwijk, The Netherlands, Dec. 2016, pp. 1–9, doi: 10.1109/NAVITEC.2016.7931740.

[12] C. Fernández-Prades, J. Arribas, M. Majoral, J. Vilà-Valls, A. García-Rigo, M. Hernández-Pajares, "An open path from the antenna to scientific-grade GNSS products," in *Proc. 6th Intl. Colloq. Sci. Fundam. Aspects GNSS/Galileo*, Valencia, Spain, Oct. 2017, pp. 1–8

[13] (2017). *GNSS-SDR Website: 16 Design Forces for Software-Defined GNSS Receivers*. Accessed: Apr. 2, 2018. [Online]. Available: http://gnss-sdr.org/design-forces/

[14] P. Vandewalle, J. Kovacević, and M. Vetterli, "Reproducible research in signal processing," *IEEE Signal Process. Mag.*, vol. 26, no. 3, pp. 37–47, May 2009, doi: 10.1109/MSP.2009.932122.

[15] R. D. Peng, "Reproducible research in computational science," *Science*, vol. 334, no. 6060, pp. 1226–1227, Dec. 2011, doi: 10.1126/science.1213847.

[16] D. Irving, "A minimum standard for publishing computational results in the weather and climate sciences," *Bull. Amer. Meteorol. Soc.*, vol. 97, no. 7, pp. 1149–1158, Jul. 2016, doi: 10.1175/BAMS-D-15-00010.1.

[17] The Royal Society, "Science as an open enterprise," Sci. Policy Centre, London, U.K., Tech. Rep. 02/12, Jun. 2012.

[18] I. Jimenez *et al.*, "The role of container technology in reproducible computer systems research," in *Proc. IEEE Int. Conf. Cloud Eng.*, Tempe, AZ, USA, Mar. 2015, pp. 379–385, doi: 10.1109/IC2E.2015.75.

[19] S. Chacon and B. Straub, *Pro Git*, 2nd ed. New York, NY, USA: Apress, 2014, Accessed: Apr. 2, 2018. [Online]. Available: https://git-scm.com/book/en/v2

[20] J. D. Blischak, E. R. Davenport, and G. Wilson, "A quick introduction to version control with Git and GitHub," *PLoS Comput. Biol.*, vol. 12, no. 1, p. e1004668, Jan. 2016, doi: 10.1371/journal.pcbi.1004668.

[21] (2017). *Reproducible Builds*. Accessed: Apr. 2, 2018. [Online]. Available: https://reproducible-builds.org

[22] J. Bobbio, "How to make your software build reproducibly," in *Chaos Communication Camp*. Zehdenick, Germany: Mildenberg, 2015.

[23] ION GNSS SDR Standard Working Group. (Aug. 2017). *Global Navigation Satellite Systems Software Defined Radio Sampled Data Metadata Standard Revision 1.0*. Accessed: Apr. 2, 2018. [Online]. Available: https://github.com/IonMetadataWorkingGroup

[24] International GNSS Service (IGS) and RINEX Working Group and Radio Technical Commission for Maritime Services Special Committee, "RINEX—The receiver independent exchange format, version 3.03," Accessed: Apr. 2, 2018. [Online]. Available: ftp://igs.org/pub/data/format/rinex303.pdf

[25] M. Avital, "The generative bedrock of open design," in *Open Design Now: Why Design Cannot Remain Exclusive*, B. van Abel, L. Evers, R. Klaassen, and P. Troxler, Eds. Amsterdam, The Netherlands: BIS Publishers, 2011, pp. 48–58.

[26] Open DesignWorking Group. (2017). *The Open Design Definition v. 0.5*. Accessed: Apr. 2, 2018. [Online]. Available: https://github.com/OpenDesign-WorkingGroup

[27] T. Feist, "Vivado design suite," Xilinx, Inc., San Jose, CA, Tech. Rep. WP416 (v1.1), Jun. 2014.

[28] S. R. Piccolo and M. B. Frampton, "Tools and techniques for computational reproducibility," *GigaScience*, vol. 30, no. 5, pp. 1–13, Jul. 2016, doi: 10.1186/s13742-016-0135-4.

[29] C. Fernández-Prades *et al.*, "A cloud optical access network for virtualized GNSS receivers," in *Proc. 30th Int. Techn. Meeting Satellite Division Inst. Navigat.*, Portland, OR, USA, Sep. 2017, pp. 3796–3815.

[30] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Philadelphia, PA, USA, Mar. 2015, pp. 171–172, doi: 10.1109/ISPASS.2015.7095802.

[31] B. K. Beaulieu-Jones and C. S. Greene, "Reproducibility of computational workflows is automated using continuous analysis," *Nature Biotechnol.*, vol. 35, no. 4, pp. 342–346, Apr. 2017, doi: 10.1038/nbt.3780.

[32] G. Kahn, "The semantics of a simple language for parallel programming," in *Information Processing*, J. L. Rosenfeld, Ed. Stockholm, Sweden: North Holland, Aug. 1974, pp. 471–475.

[33] G. Kahn and D. B. MacQueen, "Coroutines and networks of parallel processes," in *Information Processing*, B. Gilchrist, Ed. Amsterdam, The Netherlands: North Holland, 1977, pp. 993–998.

[34] T. M. Parks, "Bounded scheduling of process networks," Ph.D. dissertation, Univ. California, Berkeley, Berkeley, CA, USA, Dec. 1995.

[35] T. W. Rondeau. (Sep. 2013). *Explaining the GNU Radio Scheduler*. Accessed: Apr. 2, 2018. [Online]. Available: http://www.trondeau.com/blog/2013/9/15/explaining-the-gnu-radio-scheduler.html

[36] C. Hay, "Standardized GPS simulation scenarios for SPS receiver testing," in *Proc. IEEE/ION Position, Location, Navigat. Symp.*, Apr. 2006, pp. 1080–1085, doi: 10.1109/PLANS.2006.1650713.

[37] S. Wellek, *Testing Statistical Hypotheses of Equivalence and Noninferiority*, 2nd ed. Boca Raton, FL, USA: CRC Press, Jun. 2010.

[38] G. Booch, *Object Oriented Design With Applications*. New York, NY, USA: Benjamin Cummings, 1991.
[39] P. Duvall, S. Matyas, and A. Glover, "Continuous Integration," in *Improving Software Quality and Reducing Risk*. Upper Saddle River, NJ, USA: Addison Wesley, 2007.
[40] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Upper Saddle River, NJ, USA: Addison Wesley, 2011.
[41] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, Mar. 2017, doi: 10.1109/ACCESS.2017.2685629.
[42] P. Rodríguez *et al.*, "Continuous deployment of software intensive products and services: A systematic mapping study," *J. Syst. Softw.*, vol. 123, pp. 263–291, Jan. 2017, doi: 10.1016/j.jss.2015.12.015.
[43] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *Proc. 31st IEEE/ACM Intl. Conf. Automated Softw. Eng.*, Singapore, Sep. 2016, pp. 426–437, doi: 10.1145/2970276.2970358.
[44] N. Barnes, "Publish your computer code: It is good enough," *Nature*, vol. 467, no. 7317, p. 753, Oct. 2010, doi: 10.1038/467753a.
[45] M. Ihle, I. S. Winney, A. Krystalli, and M. Croucher, "Striving for transparent and credible research: Practical guidelines for behavioral ecologists," *Behavioral Ecol.*, vol. 28, no. 2, pp. 348–354, Apr. 2017, doi: 10.1093/beheco/arx003.
[46] F. Machida, J. Xiang, K. Tadano, and Y. Maeno, "Lifetime extension of software execution subject to aging," *IEEE Trans. Rel.*, vol. 66, no. 1, pp. 123–134, Mar. 2017, doi: 10.1109/TR.2016.2615880.
[47] G. J. Langley, R. D. Moen, K. M. Nolan, T. W. .Nolan, C. L. Norman, and L. P. Provost, *The Improvement Guide*. San Francisco, CA, USA: Jossey-Bass, 2009.
[48] C. Fernández-Prades, J. Arribas, and L. Esteve, *GNSS-SDR v0.0.9*, Zenodo, Feb. 2017, doi: 10.5281/zenodo.291371.
[49] *Information Technology—Programming languages—C++*, International Organization Standardization, Standard ISO/IEC 14882:2014(E), Geneva, Switzerland, Dec. 2014.
[50] A. Rügamer, F. Förster, M. Stahl, and G. Rohmer, "Features and applications of the adaptable Flexiband USB3.0 front-end," in *Proc. 27th Int. Tech. Meeting Satellite Division Inst. Navigat.*, Tampa, FL, USA, Sep. 2014, pp. 330–362.
[51] A. Ramos and C. Fernández-Prades. (Feb. 2018). *GNSS Signal Samples in the L2 Band*, Zenodo. Accessed: Apr. 2, 2018. [Online]. Available: https://zenodo.org/record/1184601, doi: 10.5281/zenodo.1172670.

**CARLES FERNÁNDEZ-PRADES** (S'02–M'06–SM'12) received the M.S. and Ph.D. *(cum-laude)* degrees in electrical engineering from the Universitat Politècnica de Catalunya (UPC), in 2001 and 2006, respectively. In 2001, he joined the Department of Signal Theory and Communication, UPC, as a Research Assistant, getting involved in European and National research projects both with technical and managerial duties. He also was a Teaching Assistant in the field of analog and digital communications, UPC, from 2001 to 2005. In 2006, he joined the Centre Tecnològic de Telecomunicacions de Catalunya, where he is currently a Senior Researcher and serves as the Head of the Statistical Inference for Communications and Positioning Department. He was an Advisor of two theses acknowledged with the EURASIP Best Ph.D. Thesis Award in 2014 and 2015. His primary areas of interest include Bayesian estimation, signal processing, communication systems, GNSS, software-defined radio, and design of RF front-ends.

**JORDI VILÀ-VALLS** (SM'17) received the Ph.D. degree in electrical engineering from Grenoble INP, France, in 2010. He is currently a Senior Researcher with the Statistical Inference for Communications and Positioning Department, Centre Tecnològic de Telecomunicacions de Catalunya, and also a Lecturer with the Telecommunications and Aerospace Engineering School, Universitat Politècnica de Catalunya, Barcelona, Spain. His primary areas of interest include robust statistical signal processing at large, nonlinear Bayesian inference, computational and robust statistics, with applications to: GNSS, indoor positioning/localization, tracking and sensor fusion systems, wireless communications, and aerospace science.

**JAVIER ARRIBAS** (S'09–M'12–SM'14) received the M.Sc. degree in telecommunication engineering from La Salle University in 2004 and the Ph.D. degree from the Universitat Politècnica de Catalunya in 2012. He holds the position of a Senior Researcher with the Statistical Inference for Communications and Positioning Department, Centre Tecnològic de Telecomunicacions de Catalunya. His primary areas of interest include statistical signal processing, GNSS synchronization, detection and estimation theory, software defined receivers, FPGA prototyping and the design of RF front-ends. He was a recipient of the 2015 EURASIP Best Ph.D. Thesis Award.

**ANTONIO RAMOS** received the M.Sc. degree in telecommunication engineering from the Universitat Politècnica de Catalunya in 2013. He is currently a Research Assistant with the Statistical Inference for Communications and Positioning Department, Centre Tecnològic de Telecomunicacions de Catalunya. His primary areas of interest include statistical signal processing with application in spectroscopy analysis techniques and GNSS software defined receivers.