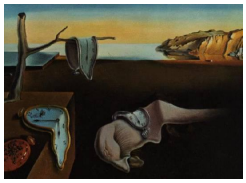

Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

Nix for Polyglot, Reproducible Data Science Workflows

Bruno Rodrigues
Ministry of Research and Higher education, Luxembourg

Abstract

Reproducible analysis requires more than clean, well-documented code: researchers must also manage software dependencies, computational environments, and workflow execution. Existing tools like **Docker** and **renv** address parts of this challenge, but no single solution handles polyglot environments, system dependencies, and workflow orchestration together. I present two R packages that leverage **Nix** to provide both reproducible polyglot environments and declarative workflow orchestration. **rix** generates **Nix** expressions that define reproducible environments spanning R, Python, Julia, and system dependencies. Building on these environments, **rixpress** orchestrates polyglot pipelines where each computational step runs in its own hermetically sealed environment, with automatic caching and dependency tracking. This approach enables researchers to develop analyses interactively while maintaining bit-for-bit reproducibility, supports collaboration across heterogeneous computational environments, and ensures that analyses remain executable years into the future.

Keywords: reproducibility, R, **Nix**.

1. Introduction: Reproducibility is also about software

Peng (2011) introduced the concept of reproducibility as a *continuum*. At one end lies the least reproducible state, where only a paper describing the study is available. Reproducibility improves when authors share the original source code, improves further when they include the underlying data, and reaches its highest level when what Roger Peng called *linked and executable code and data* are provided.

By *linked and executable code and data*, Peng referred to compiled source code and runnable scripts. In this paper, I interpret this notion more broadly as the *computational environment*: the complete set of software required to execute an analysis. Here too, a continuum

exists. At the minimal end, authors might only name the main software used—say, the R programming language. More careful authors might also specify the version of R, or list the additional packages and their versions. Rarely, however, do authors specify the operating system on which the analysis was performed, even though differences in operating systems can lead to divergent results when using the same code and software versions, as shown by Bhandari Neupane, Neupane, Luo, Yoshida, Sun, and Williams (2019). It is even less common for authors to provide step-by-step installation instructions for the required software stack, an omission often driven by institutional constraints. Journals, for instance, can inadvertently work against reproducibility by imposing strict page or word limits that leave no room for the necessary technical documentation, effectively discouraging thoroughness in the name of brevity.

Even when such instructions are given, they often fail across different platforms or versions of the same platform. This lack of portability not only hinders reproducibility but also complicates everyday research workflows. Researchers working across multiple machines must be able to recreate the same environment consistently, and collaborators must share identical computational setups to avoid inconsistencies.

Finally, once the execution environment is correctly configured, additional clarity is needed on how to *run* the project itself. Which packages should be loaded first? Which scripts should be executed, and in what order? Without clear documentation or automated orchestration, these operational details become yet another barrier to reproducibility.

This paper focuses specifically on two critical but often overlooked aspects of reproducibility: *computational environment management* and *workflow orchestration*. I present a comprehensive framework that addresses both challenges using the **Nix** package manager, making it accessible to researchers through two R packages: **rix** and **rixpress**. Before introducing these packages, I survey existing tools and their limitations to contextualize my contribution.

A range of tools now exist to help researchers approach the gold standard of full reproducibility, or to consistently deploy the same development environment across multiple machines. Let us first consider the most basic step in this process: listing the software used. In R, the `sessionInfo()` function provides a concise summary of the software environment, including the R version, platform details, and all loaded packages. Its output can be saved to a file and included as part of a study's reproducibility record. Below is an example output from `sessionInfo()`:

```
R> sessionInfo()

R version 4.3.2 (2023-10-31)
Platform: aarch64-unknown-linux-gnu (64-bit)
Running under: Ubuntu 22.04.3 LTS

Matrix products: default
BLAS:   /usr/lib/aarch64-linux-gnu/openblas-pthread/libblas.so.3
LAPACK: /usr/lib/aarch64-linux-gnu/openblas-pthread/libopenblasp[...]

locale:
 LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
 LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
```

```
 ...

attached base packages:
 stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
 nnet_7.3-19  mgcv_1.9-0   nlme_3.1-163

loaded via a namespace (and not attached):
 compiler_4.3.2  Matrix_1.6-1.1  tools_4.3.2     splines_4.3.2
 grid_4.3.2      lattice_0.21-9
```

When an author includes this information, others attempting to reproduce the study (future readers, collaborators, or even the author at a later time) can easily see which version of R and which packages (with their versions) were used. However, reproducing the environment still requires manually installing the correct package versions—a process that becomes particularly difficult when packages depend on system-level libraries. For example, the **sf** package requires **GDAL**, **GEOS**, and **PROJ** system libraries. Installing and configuring these dependencies varies substantially across operating systems and can be prohibitively difficult on some platforms.

A more robust approach than simply listing package versions is to use the **renv** package. **renv** captures the project's software state and writes it to a `renv.lock` file, which includes the exact versions of R and all required packages. This lockfile serves as a blueprint for restoring the environment automatically, ensuring that others can recreate the same setup with minimal effort. Below is an abbreviated example of an `renv.lock` file:

```
{
  "R": {
    "Version": "4.2.2",
    "Repositories": [
      {
        "Name": "CRAN",
        "URL": "https://packagemanager.rstudio.com/all/latest"
      }
    ]
  },
  "Packages": {
    "MASS": {
      "Package": "MASS",
      "Version": "7.3-58.1",
      "Source": "Repository",
      "Repository": "CRAN",
      "Hash": "762e1804143a332333c054759f89a706",
      "Requirements": []
    },
    "Matrix": {
      "Package": "Matrix",
```

```
      "Version": "1.5-1",
      "Source": "Repository",
      "Repository": "CRAN",
      "Hash": "539dc0c0c05636812f1080f473d2c177",
      "Requirements": ["lattice"]
    }
    ... additional packages omitted ...
  }
}
```

This lockfile lists each package alongside its version and the repository from which it was downloaded. Creating it requires only a single command: `renv::init()`. Others can then reproduce the same package library by running `renv::restore()`, which installs the exact package versions in an isolated, project-specific library that does not interfere with the user's global R library.

However, **renv** has important limitations. It does not restore the version of R itself—installing the correct R version must be done separately using tools like **rig** (R Infrastructure 2023). More critically, **renv** does not handle system-level dependencies. If **sf** requires **GDAL** version 3.0 but the system has version 2.4 installed, **renv** cannot resolve this conflict. Users must manually install system libraries, and the process differs across operating systems. Further details are discussed in the **renv** documentation[1].

Other packages provide similar functionality with different trade-offs. **groundhog** by Simonsohn and Gruson (2023) makes it easy to install packages as they existed on CRAN at a given date:

```
R> groundhog.library("
+   library('purrr')
+   library('ggplot2')",
+   "2017-10-04",
+   tolerate.R.version = "4.2.2")
```

groundhog places packages in project-specific libraries but does not install R itself. Without the `tolerate.R.version` argument, it will refuse to proceed if the R version does not match expectations. **rang**, developed by hong Chan and Schoch (2023), offers similar date-based package installation. The Posit Package Manager provides another approach through dated CRAN snapshots, configured via the `.Rprofile` file:

```
R> options(repos =
+    c(REPO_NAME =
+      "https://packagemanager.posit.co/cran/__linux__/jammy/2023-06-30"
+    )
+  )
```

This approach installs all packages from the specified snapshot date, but unless users explicitly manage separate libraries for different projects, all projects will share the same package versions.

[1] See https://rstudio.github.io/renv/articles/renv.html.

These tools represent significant progress in reproducibility, but they share a common limitation: none addresses system-level dependencies or supports polyglot environments. A project using R, Python, and system tools like **Quarto** or LaTeXrequires coordinating multiple package managers (**renv** for R, virtual environments for Python, manual installation for system tools), each with its own configuration and potential for conflicts.

The most comprehensive approach to reproducibility is containerization with **Docker**. **Docker** packages a *data product* together with all its dependencies—including the operating system, system libraries, programming languages, and packages—into a self-contained image. A statistical analysis can be viewed as such a data product. The steps to create an image are specified in a `Dockerfile`, which defines the structure and content of the **Docker** image.

Once the image is built, the analysis can be executed inside a *container*, which is a running instance of the image. Containers are typically run non-interactively, allowing the complete computational environment to be instantiated and executed with a single command.

The strength of **Docker** lies in its ability to bundle not only the correct versions of R and its packages but also system-level dependencies. Since **Docker** images are effectively minimal Linux systems, they naturally include libraries required by packages like **sf**. This ensures that future replicators have access to the same software environment, including all necessary system components. Moreover, these images can be easily shared via registries like Docker Hub, enabling seamless reproducibility across machines and collaborators.

The Rocker project, introduced by Boettiger and Eddelbuettel (2017), provides a collection of pre-built **Docker** images for the R community. These images come with specific R versions and, in some cases, preinstalled packages, making them convenient base images for building reproducible analysis environments without starting from scratch.

Despite its strengths, **Docker** presents several challenges for statistical computing workflows:

1. Poor interactive support: While containers can be modified at runtime, changes are lost when the container stops unless explicitly saved. Running graphical applications like RStudio Desktop from containers requires complex X11 configuration that works reliably only on Linux. Instead, web-based IDEs (like RStudio Server) are used, but come with the inherent limitations of web-based solutions.

2. Steep learning curve: **Docker** images are minimal Linux systems, so writing reliable `Dockerfile`s requires familiarity with Linux system administration. To ensure reproducibility, `Dockerfile`s should reference specific image *digests* rather than *tags*, but this practice is rarely followed. Packages like **dockerfiler** (Fay, Guyader, Parry, and Rochette 2024) help automate `Dockerfile` creation but cannot eliminate the need for basic Linux knowledge.

3. Post-hoc reproducibility: A common workflow is to develop analyses interactively using standard R installations, then create a `Dockerfile` afterward to enable reproduction. While this achieves post-hoc reproducibility, it fails to address the challenge of deploying consistent environments across multiple machines *during* development, a critical need for collaborative projects.

4. Single-language focus: While **Docker** can support polyglot environments, orchestrating them requires manual coordination. There is no standard way to specify that one step

should run in a Python environment while another runs in R.

5. Incomplete Reproducibility: Reproducibility should be understood as a spectrum rather than a binary property, a notion emphasised in "Reproducibility in Software Engineering" by Dellaiera (2024). Docker illustrates this spectrum well: it provides strong run-time reproducibility—that is, consistency in space (where the same image behaves identically across different systems) but weak build-time reproducibility, which concerns consistency over time. As defined in the thesis, reproducibility at build time means being able to produce an identical build artefact regardless of when or where it is built. However, Docker inherently fails to guarantee this property. Its reliance on mutable base images and non-deterministic instructions such as `apt-get update` leads to temporal drift: rebuilding the same `Dockerfile` at different points in time can produce distinct outputs. In the experimental evaluation conducted in the thesis, even with pinned base images, checksums of the resulting containers varied across builds and machines. Thus, while Docker offers partial reproducibility: strong at run time (space) and weak at build time (time), it cannot on its own ensure verifiable, deterministic software artifacts. Achieving true reproducibility requires pinning image digests, freezing dependency versions, and employing deterministic build systems such as Nix or Guix for complete control over both space and time (Malka, Zacchiroli, and Zimmermann 2024).

Even with a perfectly reproducible environment, researchers face another challenge: reliably executing the analysis workflow. Which scripts run first? What are the dependencies between steps? Manual execution is error-prone and poorly documented. Build automation tools like **Make** address this by defining analyses as ordered, reproducible steps.

Within R, the **targets** package by Landau (2021) provides a modern, declarative approach to workflow management. **targets** tracks dependencies between code and data, caches intermediate results, and only recomputes steps affected by changes. This dramatically improves efficiency for complex analyses.

However, **targets** operates within a single R session. While it can call Python via **reticulate** or execute system commands, all steps share the same computational environment. For truly polyglot pipelines (where different steps may require incompatible dependencies or different language versions) this limitation becomes problematic. One might run **targets** inside a **Docker** container to ensure reproducibility, but this requires the entire pipeline to use a single environment.

McDermott (2021) provides an excellent example of achieving the gold standard of reproducibility. The accompanying GitHub repository[2] demonstrates the state of the art:

- Package versions recorded in an `renv.lock` file
- A `Makefile` automating the full analysis
- A `Dockerfile` providing a complete computational environment

However, achieving this required mastering multiple complex tools: **renv** for package management, **Docker** for environment containerization, and **Make** for workflow orchestration. Each tool has its own syntax, concepts, and failure modes. The complexity compounds for poly-

[2]https://github.com/grantmcdermott/skeptic-priors

glot projects; for instance, a Python project would require learning virtual environments, and coordinating R and Python in a single analysis requires additional tooling.

What researchers need is a tool that:

1. Manages complete environments including programming languages, packages, and system dependencies;
2. Supports multiple languages natively, not through workarounds;
3. Works interactively without the complexity of containers;
4. Provides step-level isolation so different pipeline steps can use different environments;
5. Ensures bit-for-bit reproducibility through deterministic builds;
6. Remains simple enough for researchers without systems administration expertise.

The **Nix** package manager provides these capabilities. **Nix** ensures reproducible software installation by deploying *component closures*—packages bundled with all their dependencies and transitive dependencies (Dolstra, De Jonge, and Visser 2004). This creates self-contained software environments that work identically across machines and over time. **Nix** can replace **Docker** for environment isolation, **renv** for package management, and even **Make** for workflow orchestration—all within a single, unified framework.

However, **Nix** has a steep learning curve. It is a complex system with its own purely functional programming language designed to declaratively define how software is built and configured. While this functional approach ensures reproducibility, it creates a significant barrier to adoption.

To make **Nix** accessible to researchers, I developed two R packages:

- **rix** generates **Nix** expressions from intuitive R function calls, eliminating the need to learn the Nix language. It handles environment definition including R, Python, Julia, system tools, and their dependencies.

- **rixpress** orchestrates polyglot analytical pipelines using **Nix** as the build engine. Each pipeline step runs in its own hermetically sealed environment with automatic caching and dependency tracking. This enables true polyglot workflows where R, Python and Julia steps coexist with different dependencies. Python user can use **ryxpress**, a port of **rixpress**.

Together, these packages provide what existing tools cannot: a single framework for managing reproducible, polyglot computational environments and executing complex analytical workflows with step-level isolation. While **Nix** remains complex for advanced use cases, **rix** and **rixpress** abstract this complexity for common research workflows, making deep reproducibility accessible without requiring systems administration expertise.

The remainder of this paper proceeds as follows. Section 2 introduces the **Nix** package manager and explains why its functional approach enables reproducibility. Section 3 presents **rix** and demonstrates environment definition for various use cases. Section 4 discusses the `rstats-on-nix` fork of the official package repository and how it addresses practical limitations. Section 5 introduces **rixpress** and demonstrates polyglot pipeline orchestration through a complete example. Section 6 concludes with discussion of appropriate use cases and future directions.

2. The Nix package manager

Nix is a powerful, cross-platform package manager designed for installing and building software in a fully reproducible and reliable manner. Unlike traditional package managers, **Nix** emphasizes immutable infrastructure and a functional approach, which significantly enhances the consistency of computational environments. As of writing, the primary Nix package collection, `nixpkgs`, provides access to over 120,000 packages, including nearly all of CRAN and Bioconductor. According to Repology, a service that tracks package repositories across distributions, `nixpkgs` is both the largest and among the most up-to-date package repositories available (see Figure Figure 1).

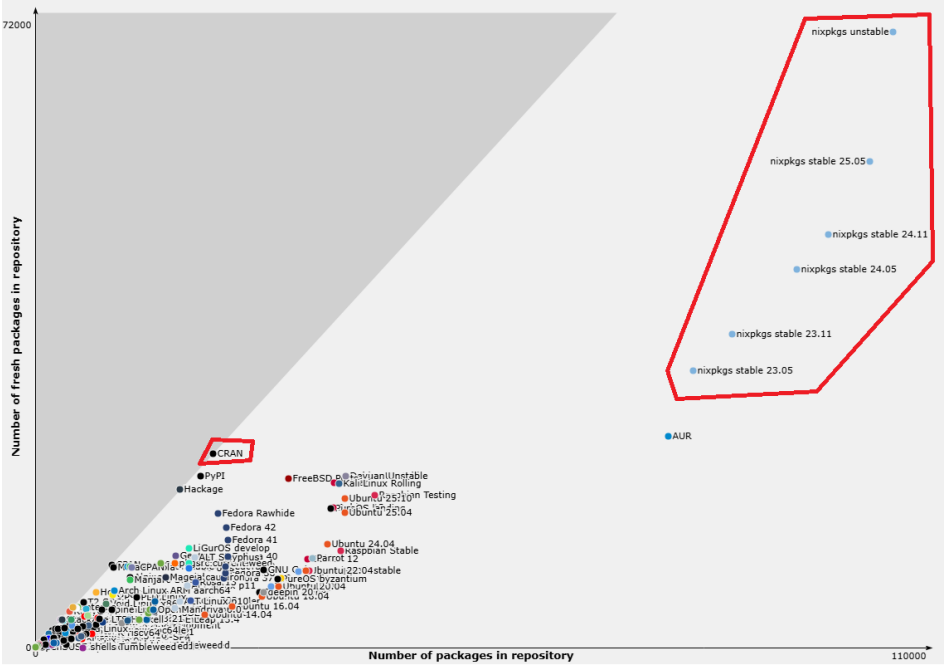

Figure 1: Package repository size and freshness according to Repology. `nixpkgs` leads in both total package count and update recency, making it well-suited for reproducible research environments.

This extensive coverage allows users to install not only R itself but also all its required packages and system-level dependencies for any given project.

While **Nix** is the default package manager for the NixOS Linux distribution, it can be installed as a standalone tool on other Linux distributions, macOS, and effectively on Windows via the Windows Subsystem for Linux (WSL2), treating these as equivalent platforms in practice.

A key advantage of using **Nix** to install R packages, as opposed to the standard `install.packages()` function, lies in its comprehensive dependency management. **Nix** ensures that all dependencies of each package are installed, irrespective of whether they are other R packages or underlying system libraries.

Consider, for example, the **sf** package, which is fundamental for spatial data analysis in R. **sf** relies on several complex external system libraries such as GDAL, GEOS, and PROJ.

Manually installing and configuring these libraries to be compatible with **sf** can be challenging and platform-specific. With **Nix**, however, the user merely needs to declare **sf** as a project requirement. **Nix** then automatically installs and configures all necessary system libraries as transitive dependencies. This seamless process is possible because the maintainers of the R programming language for **Nix** have declaratively specified these dependencies for **sf** in `nixpkgs`, allowing the entire software stack to be provisioned effortlessly from an end-user perspective.

This concept is central to **Nix**'s design, where packages are referred to as *component closures*. As Dolstra *et al.* (2004) explains:

> The idea is to always deploy component closures: if we deploy a component, then we must also deploy its dependencies, their dependencies, and so on. That is, we must always deploy a set of components that is closed under the ''depends on'' relation. Since closures are selfcontained, they are the units of complete software deployment. After all, if a set of components is not closed, it is not safe to deploy, since using them might cause other components to be referenced that are missing on the target system.

The core of **Nix**'s reproducibility lies in its functional package management paradigm and its unique approach to builds. When installing software with **Nix**, it evaluates an *expression* written in the Nix language. These expressions define *derivations*, which are declarative blueprints describing how to build a package. A derivation specifies all inputs: source code, build commands, and crucially, all direct and transitive dependencies. This declarative nature ensures that every build process is fully specified and independent of the environment where it's executed. Furthermore, all **Nix** expressions for official packages are hosted in the `nixpkgs` GitHub repository. By referencing a specific commit of `nixpkgs` (a process known as *pinning a revision*) users guarantee that all packages installed are derived from the exact same set of build instructions. This means that if an environment is built today using a pinned revision, it will produce the identical set of software versions and configurations if rebuilt tomorrow, next year, or on a different machine, as long as the `nixpkgs` repository remains accessible.

Pinning is crucial, but it is not the only reason **Nix** supports reproducibility. **Nix** is a *functional* package manager in the programming sense: it treats software builds as pure functions. Given the same inputs (source code, dependencies, build scripts), a **Nix** build will always produce the identical output, regardless of when or where it's executed. This is achieved by disallowing side effects and ensuring that builds operate within hermetic (isolated) environments, with no hidden dependencies on global system state. While this functional approach greatly enhances reproducibility, it can sometimes introduce complexity for package maintainers, especially for software that needs to download external assets during installation (e.g., certain Bioconductor packages). For end-users, however, this complexity is largely abstracted away. Additionally, **Nix** supports multiple versions or *variants* of a package on the same system. Each package installed into the "Nix store" (typically in `/nix/store`) has a unique identifier derived from a hash of all its inputs. This isolation ensures that projects always use their intended software versions, allowing multiple versions of R to coexist and preventing "dependency hell" and global environment pollution.

For a more in-depth technical discussion of **Nix**'s design principles, see Dolstra *et al.* (2004).

By leveraging these principles, **Nix** can effectively replace and integrate the functionalities of

tools like **renv** and **Docker** for R projects, or `requirements.txt` files and virtual environments for Python projects. Furthermore, **Nix** excels at building polyglot environments, which can seamlessly combine R, Python, Julia, a LaTeX distribution, or any of the numerous other tools available in `nixpkgs`. This enables the creation of a truly complete, project-specific, and deeply reproducible environment that can be used interactively for development or non-interactively for automated analysis. As long as the `nixpkgs` repository (or a suitable fork, as discussed in Section Section 4) remains accessible, the environment can be rebuilt reliably in the future.

Nix provides strong cross-platform support across major operating systems. It runs natively on Linux (x86_64) with full functionality and optimal integration. On macOS (Intel and Apple Silicon), however, reproducibility is less reliable due to dependencies on impure system frameworks and Xcode toolchains outside the Nix store. As a result, macOS users may need to update project pins after system or Xcode upgrades to restore builds. On Windows, **Nix** works well through the Windows Subsystem for Linux (WSL2), though graphical applications may need extra setup. **Nix** environments on WSL can be used interactively via VS Code or Positron for a smooth development experience. It is also important to note that **Nix** can be installed within a **Docker** image to deploy and reproduce environments reliably in containerized workflows.

While **Nix** is a powerful system, it comes with notable challenges. The steepest is its learning curve: **Nix** uses its own declarative language (also called Nix), which can be difficult for those new to functional or declarative paradigms. Complexity becomes most apparent when writing custom derivations or troubleshooting intricate build issues. Build times can also be long, especially for large environments, when pre-built binaries aren't available. Additionally, because **Nix** stores all package outputs immutably in the Nix store, disk usage tends to be higher than with traditional package managers. Finally, while `nixpkgs` is extensive, some new or niche packages may be missing or fail to build across platforms.

To make these capabilities more approachable for R users seeking reproducible, project-specific environments without learning the full Nix language, I created the **rix** and **rixpress** packages.

3. Reproducible development environments with Nix

As mentioned, **Nix** expressions are written in the Nix programming language, which is purely functional. Here is a simple example that creates a shell environment containing version 4.3.1 of R:

```
let
  pkgs = import (fetchTarball
    "https://github.com/NixOS/nixpkgs/archive/976fa336.tar.gz"
  ) {};
  system_packages = builtins.attrValues {
    inherit (pkgs) R;
  };
in
  pkgs.mkShell {
    buildInputs = [ system_packages ];
```

```
    shellHook = "R --vanilla";
  }
```

In this expression, the `let` keyword is used to define variables. The variable `pkgs` imports the set of packages from the `nixpkgs` repository at the specified commit `976fa336`. The variable `system_packages` lists the packages to include in the environment; in this case, it is just the R programming language, along with all its dependencies and their transitive dependencies. The `mkShell` function then creates a development shell with the specified packages. The `shellHook` is set to `"R --vanilla"`, meaning that entering the shell automatically starts R in vanilla mode, ignoring any startup options.

This expression can be saved in a file called `default.nix`. The environment can then be built on a system with **Nix** installed using the `nix-build` command.[3]. Once the build completes, the user can enter the interactive shell with `nix-shell`. This shell contains all the packages specified in `default.nix` and can be used for development, similar to activating a virtual environment in the Python ecosystem.

Writing **Nix** expressions can be challenging for users unfamiliar with the Nix language. However, the ability to define a fully reproducible development environment in a single text file and then rebuild it anywhere is highly appealing. To lower the barrier to adoption of **Nix** for reproducibility, I developed the **rix** package.

rix provides the `rix()` function, which simplifies generating **Nix** expressions. It is available on CRAN and can be installed like any other R package. Additionally, it can bootstrap an R development environment on a system where R is not yet installed but **Nix** is available. This can be done by running (inside of a terminal):

```
$> nix-shell -I \
+    nixpkgs=https://github.com/rstats-on-nix/nixpkgs/tarball/2025-10-20 -p \
+    R rPackages.rix
```

(the `-I` flag allows one to pass a specific revision of `nixpkgs`, ensuring temporary shells are also reproducible).

This command opens a temporary R session with **rix** available.[4] From there, users can generate new **Nix** expressions for building environments. For example, the following generates a `default.nix` file that installs R 4.3.1 along with the **dplyr** and **chronicler** packages:

```
R> library('rix')

R> rix(r_ver = "4.3.1",
+    r_pkgs = c("dplyr", "chronicler"),
+    project_path = ".",
+    overwrite = TRUE)
```

rix can also handle more complex setups, and users can provide a date instead of a specific R version:

[3]For installing **Nix**, I recommend the Determinate Systems installer: https://determinate.systems/posts/determinate-nix-installer

[4]`nix-shell -p` starts an interactive shell with the specified packages.

```
R> rix(date = "2025-10-20",
+    r_pkgs = c("rix", "dplyr", "chronicler", "AER@1.2-8"),
+    system_pkgs = c("quarto", "git"),
+    tex_pkgs = c(
+          "amsmath",
+          "framed",
+          "fvextra",
+          "environ",
+          "fontawesome5",
+          "orcidlink",
+          "pdfcol",
+          "tcolorbox",
+          "tikzfill"
+    ),
+    git_pkgs = list(
+        package_name = "fusen",
+        repo_url = "https://github.com/ThinkR-open/fusen",
+        commit = "60346860111be79fc2beb33c53e195f97504a667"
+    ),
+    ide = "positron",
+    project_path = ".",
+    overwrite = TRUE)
```

This call to `rix()` generates a `default.nix` file for a development shell that encapsulates a complete and reproducible research environment. It provides R and its packages (including **AER** at version 1.2-8), several TeXLive packages for LaTeXdocument authoring, development versions of **rix** and **fusen** pulled directly from GitHub at a specific commit, and the Positron editor. The reproducibility of this environment is guaranteed by pinning all components to a single point in time: the R packages are resolved from the CRAN snapshot of October 20, 2025, while all other system tools are fixed to the version of `nixpkgs` from that same date. Typing `nix-shell` in a terminal within the folder that contains this `default.nix` will drop the user into a new shell. It should be noted that it uses the `nixpkgs` from that same date. It is also possible to configure IDEs to dynamically load this new shell to provide the proper development tooling.

rix can generate **Nix** expressions even if **Nix** is not installed on the system. This is useful for continuous integration and continuous deployment (CI/CD) workflows on platforms such as GitHub Actions. For instance, the repository containing the source code for this article[5] uses GitHub Actions to compile the paper. Each time a push is made to the master branch, a runner installs **Nix**, generates the environment from the hosted `default.nix` file, and compiles the paper using **Quarto** within the reproducible environment. This ensure that *exactly* the same environment is used on the author's computer and on the CI/CD without any additional, platform-specific, configuration.

Instead of first entering a **Nix** shell, it is also possible to run a program directly from the environment:

[5] https://github.com/b-rodrigues/rix_paper

```
cd /path/to/project/ && nix-shell default.nix --run "Rscript analysis.R"
```

This command runs `Rscript` and executes the `analysis.R` script, which in this example should be located in the same directory as `default.nix`.

4. The rstats-on-nix fork of nixpkgs

As explained earlier, **Nix** uses expressions from the `nixpkgs` GitHub repository to build software. However, when generating expressions with **rix**, the fork `rstats-on-nix/nixpkgs` is used instead.

Using a fork offers several advantages. First, it provides flexibility that the official `nixpkgs` repository cannot always accommodate. **Nix** is primarily the package manager for the NixOS Linux distribution, and governance and technical choices made upstream can limit what **rix** aims to provide.

For instance, while **Nix** can theoretically support multiple versions (or *variants*) of the same package, in practice maintainers cannot provide several variants for all R packages, given the size of the ecosystem (over 20,000 CRAN and Bioconductor packages). This makes it difficult to install a specific version of an R package not included in a particular `nixpkgs` commit. With **rix**, users can install a specific package version from source, e.g.:

```
R> rix(..., r_pkgs = "dplyr@1.0.7", ...)
```

However, installing from source might fail, especially if the package needs to be compiled.

Additionally, updating the full R package set on **Nix** daily is impractical. While CRAN and Bioconductor update daily, the R packages in `nixpkgs` are only updated with new R releases. This limitation is due to **Nix**'s governance as a Linux distribution package manager.

The `rstats-on-nix` fork allows me to circumvent these limitations. For example, it provides daily snapshots of CRAN. Each day, the R package set is updated and committed to a dated branch using GitHub Actions. Users can select a specific date with:

```
R> rix(date = "2024-12-14", ...)
```

I strive to provide an available date per week: each Monday, a GitHub Action tests popular packages on Linux and macOS, and only if all tests succeed is the date added to the list of available dates in **rix**. This ensures users can reliably install packages, and allows me to backport fixes if needed. For example, when RStudio was temporarily broken due to a dependency issue (`boost`), a pull request was submitted to the official `nixpkgs` repository. I backported the fix to the `rstats-on-nix` fork, making RStudio available to users of **rix** earlier than upstream, as merging PRs in the official repository can take some time.

I have backported fixes to the `rstats-on-nix nixpkgs` fork as far back as March 2019. The process involves checking out a `nixpkgs` commit on the selected date, updating the R package set using Posit CRAN and Bioconductor snapshots, backporting fixes, and ensuring popular packages work on both x86-linux (including WSL2) and aarch64-darwin (Apple Silicon). These changes are committed to a dated branch in `rstats-on-nix/nixpkgs`. Users can see all available dates with `rix::available_dates()`.

A drawback of forking `nixpkgs` is that backported packages are not included upstream and thus are not prebuilt by Nix's CI platform, Hydra. Users may need to build many packages from source, which can be time-consuming. To mitigate this, I provide a binary cache sponsored by Cachix, complementing the public Nix cache. Instructions for using Cachix are in **rix**'s documentation. Using the cache significantly speeds up installations, as prebuilt packages are downloaded rather than compiled.

The fork also allows me to catch issues (such as packages' builds breaking) early on, and prepare fixes that can then be contributed upstream.

5. Orchestrating the workflow with rixpress

Defining a reproducible environment with **rix** addresses the first major challenge of reproducibility. The second challenge is reliably and efficiently executing the analysis workflow within that environment, which is the role of the **rixpress** package (or **ryxpress** for Python users).

As mentioned in the introduction, a build automation tool like **targets** is invaluable for managing complex analyses. It tracks dependencies between code and data, caches results, and only recomputes steps that have changed. One can run a **targets** pipeline inside a Nix environment to make it reproducible. However, this approach has limitations: the entire pipeline must run in a single environment, and orchestrating steps across different languages (e.g., R and Python) requires manual handling via packages like **reticulate**.

rixpress overcomes these limitations by using Nix not just as a package manager, but as the build automation engine itself. In a **rixpress** pipeline, each step is defined as a Nix derivation, providing two key benefits:

1. True Polyglot Pipelines: Each step can have its own Nix environment. A Python step can run in a pure Python environment, an R step in an R environment, and a Quarto rendering step in yet another, all within the same pipeline.
2. Deep Reproducibility: Each step is a hermetically sealed Nix derivation whose output is cached in the Nix store based on the hash of all its inputs. All artifacts are direct children of the computational environment (because the computational environment is actually a dependency of the artifacts), ensuring they are rebuilt if the environment changes, keeping environment and outputs always in sync.

To illustrate these capabilities with a realistic research task, I present a polyglot pipeline that simulates a Real Business Cycle (RBC) model in Julia (Bezanson, Edelman, Karpinski, and Shah 2017), uses the resulting data to train an **XGBoost** forecasting model in Python, visualizes the results in R with **ggplot2**, and finally compiles a **Quarto** report. The code for this example can be found in the following repository[6], with additional details provided in the Appendix of this manuscript.

The user defines the pipeline in an R script (`gen-pipeline.R`) using functions inspired by **targets**. The underlying logic for each step is encapsulated in separate helper scripts (`functions.jl`, `functions.py`, and `functions.R`). Further details are provided as well as

[6]https://github.com/b-rodrigues/rixpress_demos/tree/master/rbc

step-by-step instructions in the Appendix. The high-level orchestration script demonstrates how `{rixpress}` defines a granular, multi-step machine learning workflow that crosses language boundaries:

```
R> library('rixpress')

R> pipeline_steps <- list(
+  # STEP 0: Define RBC Model Parameters in Julia
+  rxp_jl(alpha, 0.3),
+  rxp_jl(beta, 1 / 1.01),
+  # ... (other parameters omitted for brevity) ...
+
+  # STEP 1: Julia - Simulate the RBC model
+  rxp_jl(
+    name = simulated_rbc_data,
+    expr = "simulate_rbc_model(alpha, beta, delta, rho, sigma, sigma_z)",
+    user_functions = "functions/functions.jl",
+    encoder = "arrow_write"
+  ),
+
+  # STEP 2.1: Python - Prepare features
+  rxp_py(
+    name = processed_data,
+    expr = "prepare_features(simulated_rbc_data)",
+    user_functions = "functions/functions.py",
+    decoder = "pyarrow.feather.read_feather"
+  ),
+
+  # STEP 2.2: Python - Split data (X_train, y_train, etc.)
+  rxp_py(name = X_train, expr = "get_X_train(processed_data)", ...),
+  # ... (other data splits omitted for brevity) ...
+
+  # STEP 2.3: Python - Train the XGBoost model
+  rxp_py(
+    name = trained_model,
+    expr = "train_model(X_train, y_train)",
+    user_functions = "functions/functions.py"
+  ),
+
+  # STEP 2.4: Python - Make predictions and format results
+  # ... (prediction and formatting steps omitted for brevity) ...
+  rxp_py(
+    name = final_predictions_df,
+    expr = "format_results(y_test, model_predictions)",
+    user_functions = "functions/functions.py",
+    encoder = "save_arrow"
+  ),
```

```
+
+  # STEP 3: R - Visualize the predictions
+  rxp_r(
+    name = output_plot,
+    expr = plot_predictions(final_predictions_df),
+    user_functions = "functions/functions.R",
+    decoder = arrow::read_feather
+  ),
+
+  # STEP 4: Quarto - Compile the final report
+  rxp_qmd(
+    name = final_report,
+    qmd_file = "readme.qmd"
+  )
+)

# Generate the 'pipeline.nix' file from the R list
R> rxp_populate(pipeline_steps, build = TRUE)
```

The `rxp_populate()` function translates this R list into a `pipeline.nix` file, which declaratively defines the entire workflow. Data flows from derivation to derivation by being serialized into the efficient and language-agnostic Arrow format using the pair of `encoder/decoder` functions. Other universal formats, such as `csv` or `json` could have been used.

As a sidenote: Python users who wish to use **ryxpress** define pipelines using the same R-based DSL shown above. This design choice keeps the pipeline definition language consistent across both R and Python ecosystems. **ryxpress** calls R and **rixpress** under the hood to generate and build the `pipeline.nix` file.

The package can then generate a visual representation of the pipeline's directed acyclic graph by running `rxp_dag()`, as can be seen in Figure 2.

To execute the pipeline, one can either set `build = TRUE` in `rxp_populate()` or call `rxp_make()` separately. **Nix** executes each step in order, building dependencies as needed. Outputs are cached, so subsequent runs only recompute steps with changed inputs or code. This provides the efficiency of **targets** with polyglot support and bit-for-bit reproducibility. Artifacts can be inspected interactively in R using `rxp_read("artifact_name")` or `rxp_load("artifact_name")`.

For Python users, a port called **ryxpress** allows building the same pipelines and inspecting outputs from Python sessions. **rixpress** also includes several additional features not covered here for brevity. As mentioned previously, it is also possible to configure popular IDEs to work interactively and seamlessly with both **rix** and **rixpress**, enabling a smooth, reproducible workflow from within the development environment. Detailed setup instructions are provided in the vignettes of both packages.

6. Conclusion

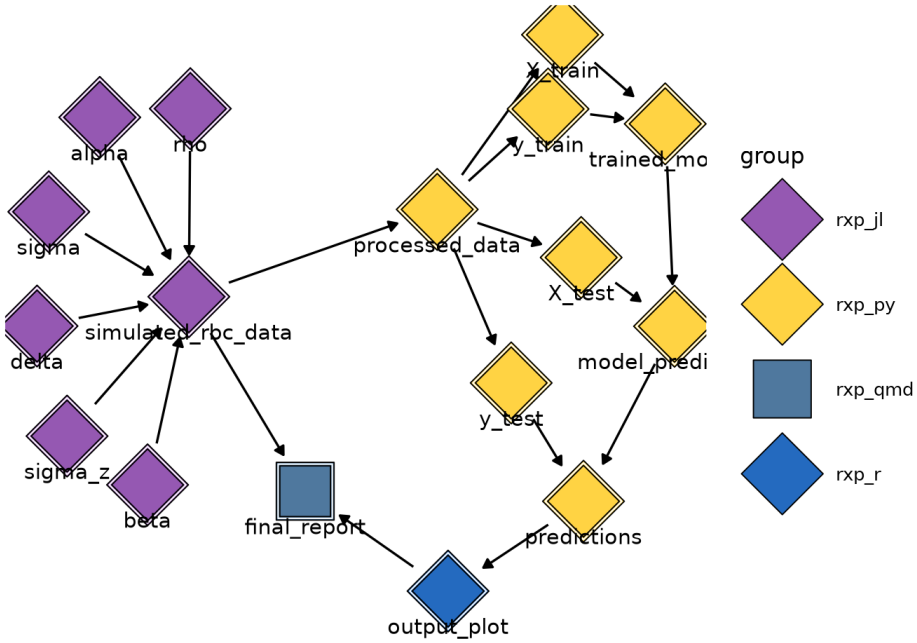

Figure 2: Graphical representation of the polyglot pipeline. Purple nodes are Julia, yellow nodes are **Python**, light blue nodes are R, and dark blue nodes are Quarto derivations.

Many tools exist to improve reproducibility, but **Nix** stands out because it deploys complete software environments *closed under the "depends on" relation*: it installs not only a package, but all its dependencies and their dependencies. This makes **Nix** particularly powerful for reproducible research.

However, solving such a complex problem makes **Nix** a complex tool. With **rix**, I aim to make **Nix** more accessible to R users by providing a familiar interface and workflow. By building reproducible development shells with **Nix**, researchers can accommodate a wide range of use cases: running scripts and pipelines, developing interactive **shiny** applications, or serving **plumber** APIs.

Furthermore, **rixpress** extends this reproducibility to entire analysis pipelines. By leveraging Nix as a build automation engine, **rixpress** allows polyglot workflows where each step runs in its own hermetically sealed environment. This ensures deep reproducibility, efficient caching, and seamless orchestration of polyglot analyses. Most importantly, it provides native polyglot support without the manual coordination required by containerization approaches, making complex multi-language workflows accessible to researchers without systems administration expertise.

While **rix** and **rixpress** (and **ryxpress**) provide significant steps forward, it is important to acknowledge current limitations. **rixpress** (and **ryxpress**) in particular requires further development, especially to enhance the ease of debugging complex pipelines. Its visualization capabilities are also still fairly limited; for instance, they do not currently indicate outdated derivations when a user modifies code that would necessitate a rebuild. These areas represent important avenues for future work to further mature the framework and improve the user

experience.

Acknowledgments

I thank the rOpenSci reviewers and contributors who provided valuable feedback on the development of **rix** and **rixpress**. In particular, I am grateful to David Watkins and Jacob Wujiciak-Jens for their reviews of **rix**, and to William Landau and Anthony Martinez for their reviews of **rixpress**. I also acknowledge the contributions of Philipp Baumann, **rix**'s co-author, Richard J. Acton, Jordi Rosell, Elio Campitelli, László Kupcsik, and Michael Heming for **rix**. Their expertise and feedback greatly improved the quality and usability of these packages. I would also like to thank Pol Dellaiera and Edvin Syk for providing feedback on the manuscript.

References

Bezanson J, Edelman A, Karpinski S, Shah VB (2017). "Julia: A fresh approach to numerical computing." *SIAM review*, **59**(1), 65–98. URL https://doi.org/10.1137/141000671.

Bhandari Neupane J, Neupane RP, Luo Y, Yoshida WY, Sun R, Williams PG (2019). "Characterization of Leptazolines A–D, Polar Oxazolines from the Cyanobacterium Leptolyngbya sp., Reveals a Glitch with the "Willoughby–Hoye" Scripts for Calculating NMR Chemical Shifts." *Organic Letters*, **21**(20), 8449–8453. doi:10.1021/acs.orglett.9b03332.

Boettiger C, Eddelbuettel D (2017). "An Introduction to Rocker: Docker Containers for R." *The R Journal*, **9**(2), 527–536. doi:10.32614/RJ-2017-065.

De Paoli B (2009). "Slides 1: The RBC Model, Analytical and Numerical solutions." https://personal.lse.ac.uk/depaoli/RBC_slides1.pdf. Lecture Slides.

Dellaiera P (2024). "Reproducibility in Software Engineering." doi:10.5281/zenodo.12666898.

Dolstra E, De Jonge M, Visser E (2004). "**Nix**: A Safe and Policy-Free System for Software Deployment." In *18th Large Installation System Administration Conference*, pp. 79–92.

Fay C, Guyader V, Parry J, Rochette S (2024). ***dockerfiler**: Easy Dockerfile Creation from R*. R package version 0.2.5, URL https://thinkr-open.github.io/dockerfiler/.

hong Chan C, Schoch D (2023). "**rang**: Reconstructing Reproducible R Computational Environments." *PLOS ONE*, **18**(6), e0286761. doi:10.1371/journal.pone.0286761.

Landau WM (2021). "The **targets** R Package: A Dynamic Make-like Function-Oriented Pipeline Toolkit for Reproducibility and High-Performance Computing." *Journal of Open Source Software*, **6**(57), 2959. doi:10.21105/joss.02959.

Malka J, Zacchiroli S, Zimmermann T (2024). "Reproducibility of Build Environments through Space and Time." In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER'24, p.

97–101. ACM. doi:10.1145/3639476.3639767. URL http://dx.doi.org/10.1145/3639476.3639767.

McDermott GR (2021). "Skeptic Priors and Climate Consensus." *Climatic Change*, **166**(1-2), 7. doi:10.1007/s10584-021-03114-6.

Peng RD (2011). "Reproducible Research in Computational Science." *Science*, **334**(6060), 1226–1227. doi:10.1126/science.1213847.

R Infrastructure (2023). ***rig**: The R Installation Manager*. URL https://github.com/r-lib/rig.

Simonsohn U, Gruson H (2023). *groundhog: Version-Control for CRAN, GitHub, and GitLab Packages*. R package version 3.1.2, https://github.com/CredibilityLab/groundhog, URL https://groundhogr.com/.

7. Appendix

7.1. Reproducing this paper

The source code of this paper is hosted on GitHub and can be found at following link[7]. The paper can be easily compiled by running the following command:

```
$> nix-shell --run "quarto render paper.qmd --to jss-pdf"
```

The `default.nix` file defines the exact computational environment required to compile the manuscript, ensuring full reproducibility of the build process. To guarantee the reproducibility of the manuscript, it is automatically recompiled via GitHub Actions upon each commit, using the same Nix environment. An HTML version is additionally deployed to GitHub Pages, providing an accessible format for viewing on smaller devices at this link[8]. The PDF version of the manuscript can be found at link[9].

7.2. A Complete Polyglot Example with rixpress

This appendix provides a conceptual walk-through of the polyglot pipeline example discussed in Section 5. The pipeline simulates a Real Business Cycle (RBC) model in Julia, trains an **XGBoost** model in Python, visualizes the results in R, and compiles a final report with **Quarto**.

The complete, runnable source code for this example is available in the paper's GitHub repository. A shell script, `run_polyglot_example.sh`, is provided to automate the entire process from environment creation to final output, as described at the end of this section[10].

Project Structure and Components

[7] https://github.com/b-rodrigues/rix_paper
[8] https://b-rodrigues.github.io/rix_paper/
[9] https://b-rodrigues.github.io/rix_paper/paper.pdf
[10] https://github.com/b-rodrigues/rix_paper/blob/master/polyglot-example/run_polyglot_example.sh

The project is organized into a set of scripts, each with a distinct responsibility. The core logic is separated from the environment definition and pipeline orchestration.

- `functions/`: This directory contains the helper scripts with the core analytical code for each language.
  - `functions.jl`: The Julia script for the economic simulation.
  - `functions.py`: The Python script for the machine learning workflow.
  - `functions.R`: The R script for data visualization.
- `gen-env.R`: An R script that defines the reproducible computational environment.
- `gen-pipeline.R`: The main R script that defines and orchestrates the entire polyglot pipeline.

Step 1: The Environment Definition

The foundation of the project is the reproducible environment, defined in `gen-env.R`. This script uses the `rix()` function to programmatically generate a `default.nix` file. This file serves as a complete blueprint for the computational environment, specifying:

- The exact versions of R, Python, and Julia.
- A list of required packages for each language (e.g., **ggplot2** for R, **xgboost** for Python, **DataFrames** for Julia).
- System-level dependencies like **Quarto**.

Crucially, the entire environment is pinned to a specific date (`2025-10-14`), ensuring that anyone who builds the environment, now or in the future, will get the exact same software versions, guaranteeing reproducibility.

By dropping into a temporary shell using the following command:

```
$> nix-shell -I \
+  nixpkgs=https://github.com/rstats-on-nix/nixpkgs/tarball/2025-10-20 -p \
+  R rPackages.rix
```

it is possible to generate the `default.nix` by sourcing `gen-env.R`. Then, one leaves this temporary shell, and builds the environment using the command `nix-shell`, which also drop the user into the development shell.

Step 2: The Core Analytical Logic

The scientific logic for each stage of the pipeline is encapsulated in the separate helper scripts within the `functions/` directory.

- The RBC Model Simulation (`functions.jl`): This Julia script contains a single pure function that implements the state-space solution to the RBC model, based on De Paoli (2009). It takes the model's economic parameters as inputs and returns the simulated time-series data as a **DataFrame**.
- The **XGBoost** Forecasting Model (`functions.py`): This Python script handles the complete machine learning workflow through a series of modular functions. Its responsibilities include feature engineering (creating lagged variables), splitting the data into training and testing sets, training the **XGBoost** model, and generating predictions.

- The Visualization (`functions.R`): This R script contains a function that uses **ggplot2** to create the final visualization. It is designed to take the data frame of actual and predicted values produced by the Python script and generate a plot comparing the two series.

Step 3: Orchestrating the Pipeline

The entire workflow is defined and orchestrated by `gen-pipeline.R`. This script acts as the master plan, using functions from the **rixpress** package to define each computational step as a *derivation*. It declaratively outlines the dependencies between steps:

1. It begins by defining the RBC model parameters in Julia.
2. It specifies that the RBC simulation in `functions.jl` depends on these parameters.
3. It then defines the series of Python steps (feature preparation, training, prediction), making each one dependent on the output of the previous one. The first Python step is explicitly made dependent on the output from the Julia simulation. **rixpress** handles the passing of data between languages, in this case using the **Arrow** file format for efficiency.
4. Finally, it defines the **R visualization** step, which depends on the final predictions from the Python model, and a **Quarto rendering** step that depends on the generated plot.

When this script is run in the development shell (by executing `source("gen-pipeline.R")`), **rixpress** translates the declared pipeline into a master Nix expression that **Nix** can execute, automatically handling the caching of results and re-running only the necessary steps if a piece of code or data changes.

To build the pipeline from an interactive Python session, one would execute the following lines:

```
Python> from ryxpress import rxp_make
Python> rxp_make()
```

Running the Project

This entire example can be executed by running the `run_polyglot_example.sh` script available in the root of the paper's repository. This script automates the full process: 1. It first executes `gen-env.R` inside a temporary **Nix** shell to build the `default.nix` file. 2. It then executes `gen-pipeline.R` inside the newly defined environment. This triggers **Nix** to run the entire polyglot pipeline in the correct order.

Upon completion, the script will have generated all intermediate artifacts and the final `readme.html` report found in the `pipeline-output` folder containing the visualization.

Affiliation:

Bruno Rodrigues
Department of Statistics
18, Montée de la Pétrusse
Luxembourg Luxembourg
E-mail: bruno@brodrigues.co
URL: https://www.brodrigues.co

Journal of Statistical Software
published by the Foundation for Open Access Statistics
MMMMMM YYYY, Volume VV, Issue II
doi:10.18637/jss.v000.i00

http://www.jstatsoft.org/
http://www.foastat.org/
Submitted: yyyy-mm-dd
Accepted: yyyy-mm-dd