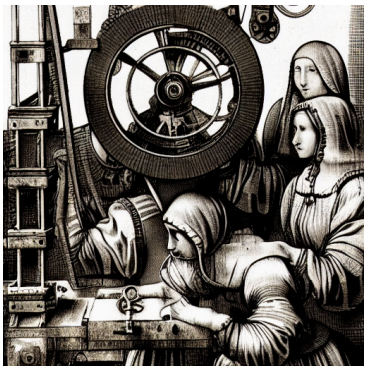
Leibniz MMS Days 2025

The Julia language: Reproducibility infrastructure and project workflows

Jürgen Fuhrmann
WIAS Berlin

The two language problem

- Combination of performant core simulation tools, e.g. in C/Fortran/C++ and glueing/postprocessing tools in a scripting language, e.g. Python.
- But ... this at least a **three language problem**: consider the **build system** in yet another language, e.g. **CMake**
- Claim: **Each project needs its own guru** to maintain the build system and to help to compile the code on new machines or to maintain docker containers
- Claim: Python APIs are easy to explain to general users, efficient algorithms are implemented in C/C++. **Python project codebases are intransparent for many of their users**
- Claim: **Containers are the new binaries**


dreamstudio.ai CC01.0

Akin to an **exponential boundary layer hitting a wall** regarding the complexity of this.

Julia tries to provide an alternative.

Julia

- Syntax comparable to matlab, python/numpy
- Just-ahead-of time compilation to native code ⇒ performance without need to vectorize or to call a computational kernel in another language
- Performant multi-dimensional arrays
- Comprehensive linear algebra
- Parallelization: SIMD, multithreading, distributed
- Interoperability with C, C++, python, R ...
- Use of modern knowledge in language design
- Open source (MIT License)
- Excellent **package management**
- Current stable release: 1.11
- Current LTS release: 1.10

https://julialang.org

https://julialang.org/downloads/#install_julia

julia

Install julia

Install the latest Julia version (v1.11.4 March 10, 2025) by running this in your terminal:

```
$ curl -fsSL https://install.julialang.org | sh
```

It looks like you're using a Unix/Linux-type system. For Windows instructions click here

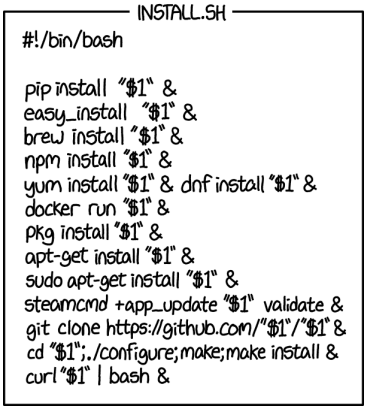
Packages

Extend Julia's core functionality. Each package is a git repository with standardized structure.

- **Package Registries** provide the infrastructure for finding package repositories via package names
- **General Registry**: ≈ 11500 packages (October 2024)
- Pkg.jl **Package Manager** is part of the Julia standard library
- **Composability** of packages due to generic Julia source code (like C++ header only libs)
- **Easy to install** via `Pkg`:

```
julia> using Pkg
julia> Pkg.add("QuantumFrobnicate")
julia> using QuantumFrobnicate
julia> QuantumFrobnicate.minimize_uncertainty!()
```

xkcd.com



Standard structure of a Julia package

- Locally, each package is stored in a directory named e.g. `MyPack` for package `MyPack.jl`.
- Structure of a package directory:
  - `MyPack/src`: subdirectory for package source code
    - `MyPack/src/MyPack.jl`: code defining a module `MyPack`
    - Further Julia sources included by `MyPack.jl`
  - `MyPack/ext`: Source code of package extensions
  - `MyPack/test`: code for unit testing
  - `MyPack/docs`: markdown sources + code for documentation
  - LICENSE.md: (open source) license
  - README.md: README - basic introduction text
  - `MyPack/.github`: control files for github CI
  - Project.toml: Metadata, list of dependencies

```
PcZ48
├─ .github
│  ⋮
├─ .gitignore
├─ LICENSE.md
├─ Project.toml
├─ README.md
├─ benchmarks
│  ⋮
├─ docs
│  ⋮
├─ ext
│  ⋮
├─ src
│  ⋮
└─ test
   ⋮
```

Package extensions: ext/

- Contain code of MyPkg which is used only if some other package is loaded.
- E.g. `MyPack/ext/MyPkgOtherPkgExt.jl` is loaded only if `OtherPkg` is used
- Avoid extensive strong dependencies and precompilation of code which is not always used.
  - E.g. load plotting code only if the plotting package is used.
- `OtherPkg` needs to be recorded in the project metadata as "weak dependency"

Continuous integration: test/runtests.jl

- `test/runtests.jl` contains code for testing package functionality
- Invoked by `Pkg.test("MyPackage")`
- `@test` macro allows to test code
- `test/Project.toml` can contain additional dependencies for testing

```
using Test
```

```
Test Passed
```

```
@test 1+1 == 2
```

Documentation generation: docs/make.jl

- `Documenter.jl` Julia package for documentation generation from docstrings
- Part of standard Julia CI: automatically upon package releases

sinsquare

```
sinsquare(x)
```

Return the squared sinus of x

```
"""
    sinsquare(x)
Return the squared sinus of x
"""
function sinsquare(x)
    return sin(x)^2
end
```

Package metadata & dependencies: Project.toml

Contents of `Project.toml` for the ForwardDiff.jl package

```toml
name = "ForwardDiff"
uuid = "f6369f11-7733-5829-9624-2563aa707210"
version = "0.10.36"

[deps]
CommonSubexpressions = "bbf7d656-a473-5ed7-a52c-81e309532950"
DiffResults = "163ba53b-c6d8-5494-b064-1a9d43ac40c5"
DiffRules = "b552c78f-8df3-52c6-915a-8e097449b14b"
LinearAlgebra = "37e2e46d-f89d-539d-b4ee-838fcccc9c8e"
LogExpFunctions = "2ab3a3ac-af41-5b50-aa03-7779005ae688"
NaNMath = "77ba4419-2d1f-58cd-9bb1-8ffee604a2e3"
Preferences = "21216c6a-2e73-6563-6e65-726566657250"
Printf = "de0858da-6303-5e67-8744-51eddeeeb8d7"
Random = "9a3f8284-a2c9-5f02-9a11-845980a1fd5c"
SpecialFunctions = "276daf66-3868-5448-9aa4-cd146d93841b"
StaticArrays = "90137ffa-7385-5640-81b9-e52037218182"

[compat]
Calculus = "0.2, 0.3, 0.4, 0.5"
CommonSubexpressions = "0.3"
DiffResults = "0.0.1, 0.0.2, 0.0.3, 0.0.4, 1.0.1"
DiffRules = "1.4.0"
DiffTests = "0.0.1, 0.1"
LogExpFunctions = "0.3"
NaNMath = "0.2.2, 0.3, 1"
Preferences = "1"
SpecialFunctions = "0.8, 0.9, 0.10, 1.0, 2"
StaticArrays = "0.8.3, 0.9, 0.10, 0.11, 0.12, 1.0"
julia = "1"

[extensions]
ForwardDiffStaticArraysExt = "StaticArrays"

[extras]
Calculus = "49dc2e85-a5d0-5ad3-a950-438e2897f1b9"
DiffTests = "de460e47-3fe3-5279-bb4a-814414816d5d"
InteractiveUtils = "b77e0a4c-d291-57a0-90e8-8db25a27a240"
SparseArrays = "2f01184e-e22b-5df5-ae63-d93ebab69eaf"
StaticArrays = "90137ffa-7385-5640-81b9-e52037218182"
Test = "8dfed614-e22c-5e08-85e1-65c5234f0b40"

[targets]
test = ["Calculus", "DiffTests", "SparseArrays", "Test", "InteractiveUtils", "StaticArrays"]
```

Package metadata: Project.toml

- Package name
- UUID to identify package, name is secondary
  - ⇒ manage different packages with the same name
- Version according to Semantic Versioning
- `[deps]`: list of dependecies with UUIDs
- `[weakdeps]`: list of weak dependecies with UUIDs
- `[extensions]`: list of package extensions and corresponding weak dependencies
- `[compat]` section: version compatibility bounds for strong and weak dependencies and julia
- `[sources]` section (since julia 1.11): path or git url of unregistered packages
- Further info: author, additional packages for testing ...

dreamstudio.ai CC01.0

Semantic versioning: [compat]

With a package version `X.Y.Z`, increment: https://semver.org

- "Major" `X` when incompatible API changes are made
- "Minor" `Y` when functionality is added in a backward compatible manner
- "Patch" `Z` when bug fixes are made

- Julia: major version is 0 ⇒ minor version acts like major
- `[compat]` section describes the compatibility bounds
- `Pkg.add` selects highest possible version
- Resolver looks for optimal solution of dependency versions

```
[compat]
julia = "1.6"    # versions >=1.6 , < 2.0
Example = "0.5"  # versions >=0.5 , < 0.6
Foo = "1.2, 2"   # versions >=1.2 , < 3.0
Bar = "~1.2.23"  # only version 1.2.23
```

What happens when adding a package ?

```
julia> Pkg.add("MyPkg")
```

1. Package name and UUID are looked up in a **registry**
2. Package git repo URL read from registry (nowadays packages are cached and served from a package server)
3. Calculate version compatibility for package and dependencies
4. Package and dependencies downloaded to `~/.julia/packages/`
5. Package and dependencies recorded in current active **environment**
6. Some package code is precompiled

dreamstudio.ai CC01.0

```
julia> using MyPkg
```

7. Precompile more of the package code
8. Make the exported symbols available for use

Manifest.toml

- Automatically generated and updated
- Lists recursively all package dependencies with their installed versions
- Since 1.11: Julia version specific manifest files: `Manifest-v1.11.toml`, `Manifest-v1.12.toml` …

```
# This file is machine-generated - editing it directly is not advised

julia_version = "1.11.4"
manifest_format = "2.0"
project_hash = "12904ab294d8d0868d4e89eb25cd4f87eb77d079"

[[deps.AbstractPlutoDingetjes]]
deps = ["Pkg"]
git-tree-sha1 = "6e1d2a35f2f90a4bc7c2ed98079b2ba09c35b83a"
uuid = "6e696c72-6542-2067-7265-42206c756150"
version = "1.3.2"

[[deps.AbstractTrees]]
git-tree-sha1 = "2d9c9a55f9c93e8887ad391fbae72f8ef55e1177"
uuid = "1520ce14-60c1-5f80-bbc7-55ef81b5835c"
version = "0.4.5"

[[deps.Adapt]]
deps = ["LinearAlgebra", "Requires"]
git-tree-sha1 = "d80af0733c99ea80575f612813fa6aa71022d33a"
uuid = "79e6a3ab-5dfb-504d-930d-738a2a938a0e"
version = "4.1.0"
weakdeps = ["StaticArrays"]

    [deps.Adapt.extensions]
    AdaptStaticArraysExt = "StaticArrays"

[[deps.ArgTools]]
uuid = "0dad84c5-d112-42e6-8d28-ef12dabb789f"
version = "1.1.2"

[[deps.ArnoldiMethod]]
deps = ["LinearAlgebra", "Random", "StaticArrays"]
git-tree-sha1 = "d57bd3762d308bded22c3b82d033bff85f6195c6"
uuid = "ec485272-7323-5ecc-a04f-4719b315124d"
version = "0.4.0"

[[deps.Artifacts]]
uuid = "56f22d72-fd6d-98f1-02f0-08ddc0907c33"
version = "1.11.0"

[[deps.AxisAlgorithms]]
deps = ["LinearAlgebra", "Random", "SparseArrays", "WoodburyMatrices"]
git-tree-sha1 = "01b8ccb13d68535d73d2b0c23e39bd23155fb712"
```

Environments: Project.toml & Manifest.toml

Environment: directory with `Project.toml` and `Manifest.toml`


dreamstudio.ai CC0 1.0

- `Project.toml`: name + UUIDs of all packages added
- `Manifest.toml`: name + UUID + version + git-hash of package *and all of its depedencies and their dependencies*
- Julia codes always runs in an environment
  - `$ julia`: use **default environment** for julia version, e.g. `~/.julia/environments/v1.11`
  - `$ julia --project=@xyz`: activate **shared environment** in `~/.julia/environments/xyz`
  - `$ julia --project=dir` or `julia> Pkg.activate("dir")`: activate **project environment** in directory `dir`

Registries

Registry: directory collecting metadata of packages for look-up

- Maintained as git repository
- Default: https://github.com/JuliaRegistries/General
  - Like blockchain: no deletions, continued forever
  - Packages must be open source
  - Automated heuristic decision process for new packages to be registered
- Local copy kept up-to-date for each Julia installation
- Multiple registries are possible

dreamstudio.ai CC01.0

Publishing a package in the General Registry

- Add Julia registrator app to github repository
- Increse version number in `Project.toml`
- Comment the relevant commit on github:

```
@JuliaRegistrator register

Release notes:
- Removed all the bugs
```

- Creates pull request to `https://github.com/JuliaRegistries/General`
- If the package is new, merge it after 3-days waiting period and heuristic feasibility check
  - Github repo with standard structure
  - Reasonable name
  - OSI approved Open Source license
- Otherwise, merge automatically after some heuristic check

I can't publish in the General Registry!

- LocalRegistry.jl supports user maintained registries
- Institutions or project cooperations can maintain their own package registries

```
julia> Pkg.registry.add("https://github.com/xyz/XYZRegistry")
julia> Pkg.add("PackageFromXYZRegistry")
```

- Uses full capabilities of `[compat]` mechanism - easy to manage by users
- Need to **trust** the repository maintainers
- `[sources]` section in Project.toml since Julia 1.11 allows to specify path or URL of package source

```
[sources]
PkgA = {url="https://github.com/xyz/PkgA.jl"}
PkgB = {path="packages/PkgB"}
```

Binary packages

- **Foreign Function Interface (FFI)** allows to call binary code compliant with the C ABI
- `CxxWrap.jl`, `CXX.jl` provide ways to interoperate with code compiled from C++
- **Binary packages:** "jll's" and Julia wrappers around them

```
1 using GSL_jll
```

```
0.15064525725099692
```

```
1  ccall((:gsl_sf_bessel_J0, GSL_jll.libgsl),Cdouble, (Cdouble,),6.0)
```

```
1 using GSL
```

```
0.15064525725099692
```

```
1 GSL.sf_bessel_J0(6.0)
```

BinaryBuilder & Yggdrasil

- `BinaryBuilder.jl` provides a container like infrastructure to build and maintain **binary packages** for **all architectures supported by Julia**
  - Leverage of software libraries from the world of compiled languages
  - Mesh generators, numerics packages, GUI toolkits etc.
- https://github.com/JuliaPackaging/Yggdrasil: collection of ≈ 1400 build scripts for binary packages

https://github.com/JuliaPackaging/Yggdrasil

Hey, let us work together!

```
# coolscript.jl
using Plots
X=range(0,10, length=2001)
Y=sin.(X)+0.1*rand(length(X))
plot(X,Y)
```

```
julia> include("coolscript.jl")
julia> using Plots
┌ Package Plots not found, but a package named Plots is available from a
│ registry.
│ Install package?
│   (@v1.11) pkg> add Plots
└ (y/n/o) [y]:
```

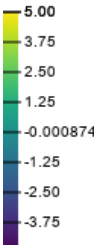
- **Default environment** cluttered, possible incompatible compatibility requirements
- Need to tell collaborator: "before running the script, you need to install this and that"

Reproducible Notebooks: Pluto.jl

Browser based notebooks implemented in Julia and Javascript

- **Easy installation**: installed as a single Julia package on Linux, MacOS, Windows
- **Reactive**: cell results are automatically recalculated
- **Version controllable**: no results in the notebook
- **Reproducible**: notebooks contain their own environment
- Efficient interaction with HTML+Javascript
- Created by Fons van der Plas & his friends

PlutoVista.jl+vtk.js+webgl: 132651 nodes




```
X=0:0.1:5;
```

```
f(x,y,z)=sin(x)*cos(3y)*z;
```

```
grid=simplexgrid(X,X,X);
```

More Pluto.jl Benefits

- **Clara** teaches a Julia based course in scientific computing. She prepares the **course material as Pluto notebooks**. After installation with simple instructions, **students run them** on their computers. The package environment automatically installs all packages necessary. HTML and PDF previews available as well.
- **Students** prepare their **exam projects as Pluto notebooks**. Clara can receive their work and run it on her computer.
- Leverage this for simple computational projects

Download Julia and install it according to the procedure on you particular operating system. Invoke Julia and issue the following commands:

```
using Pkg
Pkg.add("Pluto")
using Pluto
Pluto.run()
```

A menu will show up in the browser which allows to start a notebook

Reproducible projects

Transferring `Project.toml` allows to share the information on project dependencies along with compat bounds

- **Alice**, working on Linux, creates a project using Julia and **a number of Julia packages**. She develops the code in a directory which is activated as a Julia environment. She sets up a git repository containing source code, documentation and a `Project.toml` file.
- **Bob**, working on windows, checks out the code from the repo. A call to `Pkg.instantiate()` **installs all packages** with the right compat bounds as Alice intended.
- Upon **publishing**, they create a repo branch which in addition contains `Manifest.toml`, allowing for exact reproducibility of their results

dreamstudio.ai CC0 1.0

A take on project structure

- `Project.toml`: Project metadata + dependencies
- `LICENSE`: License of the project. By default it is the MIT license
- `README.md`: Package summary
- `src`: Subdirectory for project specific code as part of the MyProject package representing the project.
- `test`: Unit tests for project code in `src`. Could include something from `scripts`, `notebooks`.
- `scripts`, `notebooks`: "Frontend" code for creating project results
- `docs`: Sources for the documentation created with `Documenter.jl`
- `etc`: Service code

```
MyProject/
├── src
│   └── MyProject.jl
├── docs
│   ├── make.jl
│   ├── Project.toml
│   └── src
│       └── index.md
├── LICENSE
├── notebooks
│   └── demo-notebook.jl
├── Project.toml
├── README.md
├── scripts
│   └── demo-script.jl
├── etc
│   ├── runpluto.jl
│   └── instantiate.jl
└── test
    └── runtests.jl
```

PkgSkeleton.jl can generate these this: Download and unpack project skeleton project-skeleton.zip. Then invoke

```
julia> using PkgSkeleton
julia> PkgSkeleton.generate("MyProject"; templates=["project-skeleton"])
```

Further infrastructure

- DrWatson.jl manages code and computational results in a Julia project repository
  - Automatic generation of data file names from simulation parameters
- Visual Studio Code integration
- Jupyter notebook support
- Integration with quarto for reproducible publications
- Draft repo for integration with showyourwork

Some Issues

- Package loading and `using` latency due to JIT precompilation aka "Time to first plot"
  - Currently, the Julia community undertakes dedicated successful efforts towards fixing this problem
- Missing formal interface descriptions
  - Julia alternative to C++20 concepts ? Traits ?
  - Bottom up design process, fear to lose opportunities due to too rigid formalizations
- Resources for keeping infrastructure running
  - Many volunteers are involved at central points
  - Competitivity depends on package contributions
  - Server infrastructure costs

dreamstudio.ai CC01.0

Conclusions

Julia provides as well a **fresh approach to reproducibility**, learning from the experiences of of `conda`, `npm` etc.

- Package management is part of the standard Julia workflow, available without further installation
- Transparent package and project source code without the need to know two languages or handling of build systems
- Introductions to Julia at an early stage should explain working with environments etc.
- Package management can be leveraged for reproducible research

SIAM REVIEW
Vol. 59, No. 1, pp. 65–98

Julia: A Fresh Approach to Numerical Computing*

Jeff Bezanson†
Alan Edelman‡
Stefan Karpinski§
Viral B. Shah†