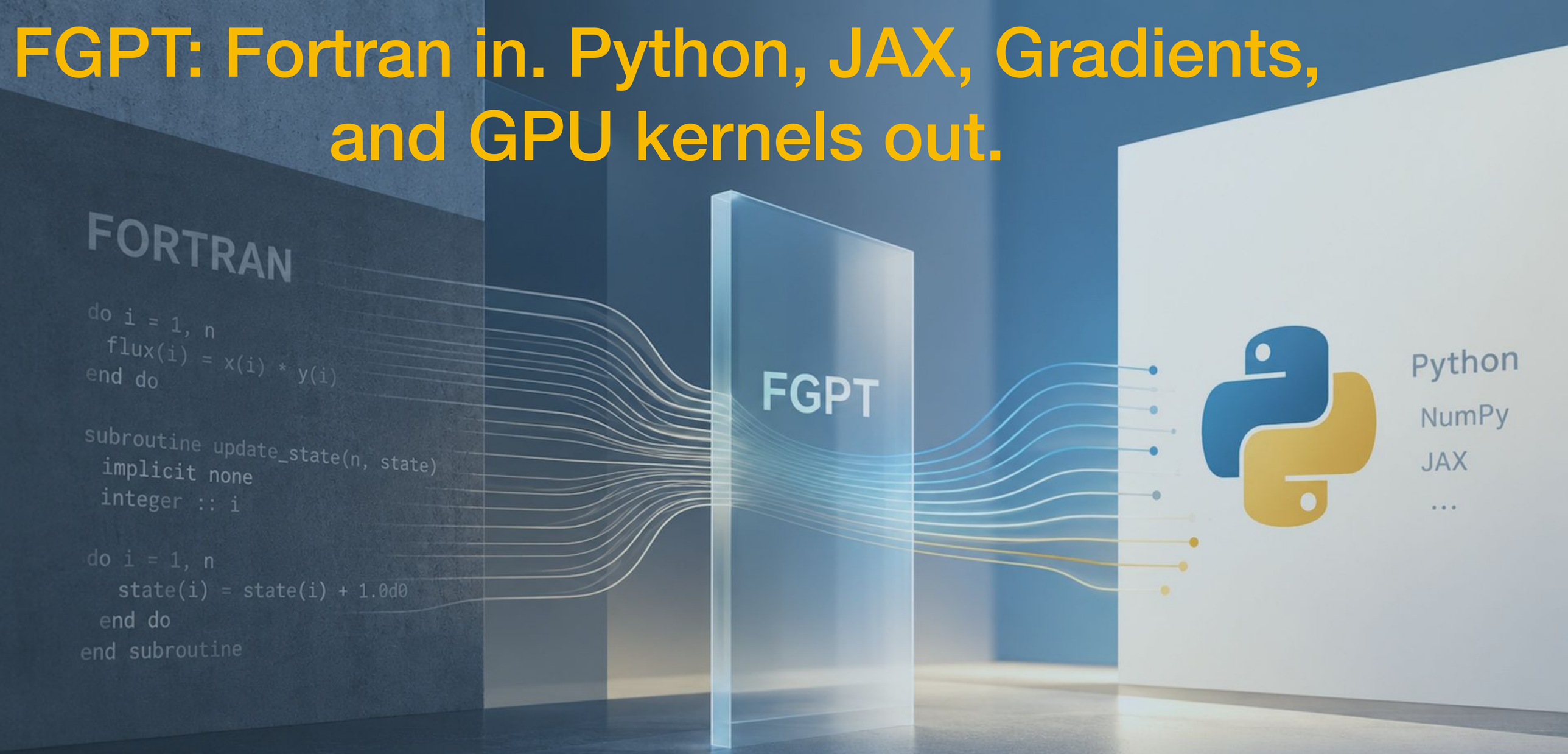
FGPT: Fortran in. Python, JAX, Gradients, and GPU kernels out.



Shivamshan SIVANESAN & Kazem ARDANEH
IPSL-CMC, CNRS, Sorbonne University

Le groupe calcul CNRS, Café Fortran, 5 November 2026

Image: © Legacy IT Center

GROUPE CALCUL

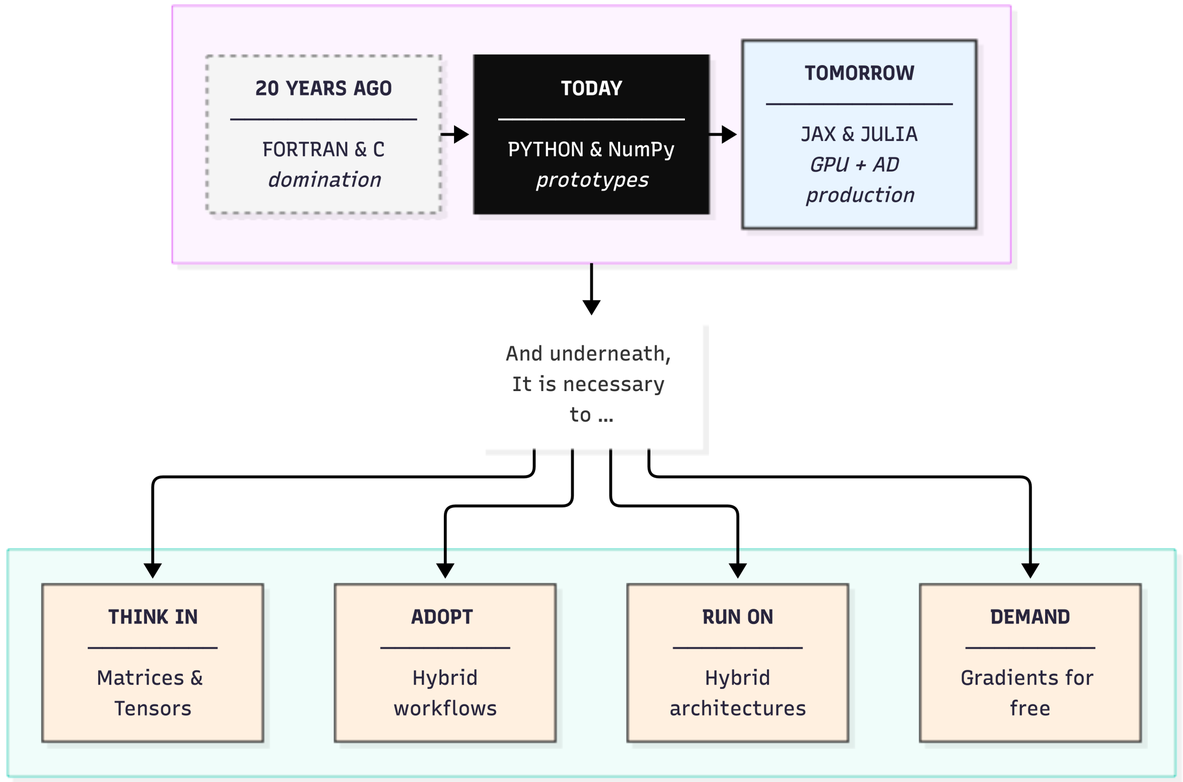
Questions

Why did we develop FGPT?

Why Python and JAX?
Why GPUs?
Why differentiable programming?
How do we transpile?

Why Python and JAX? Why GPUs?

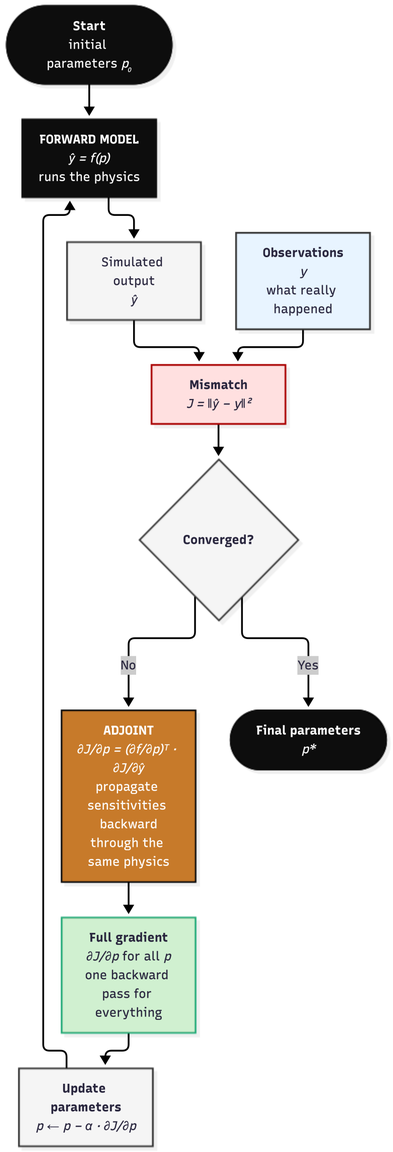
Hardware-aware software design: the essence of HPC



- 80–90% of the performance is on GPUs,
- I/O is the HPC bottleneck,
- Interfaces kill performance,
- Dependencies multiply,
- And always return to the host.

JAX: Python-based, high-performance scientific computing.

Why Differentiable Programming?



Gradients are how models learn from data!

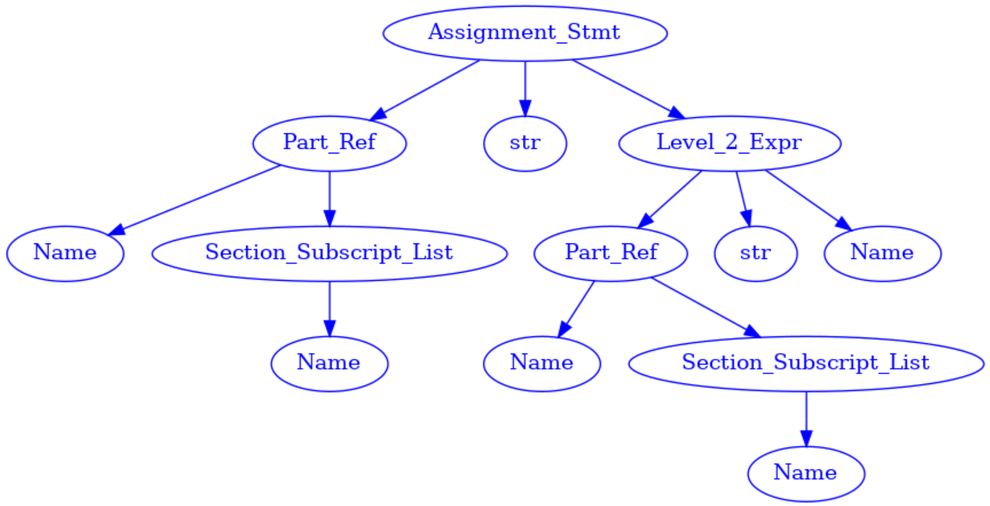
- Data tells you what is reality,
- Gradients tell you how to improve the model,
- So you need to differentiate programs,
- Tangent linear: propagates small input changes forward,
- Adjoint: propagates sensitivities backward.

Turn programs into learning machines.

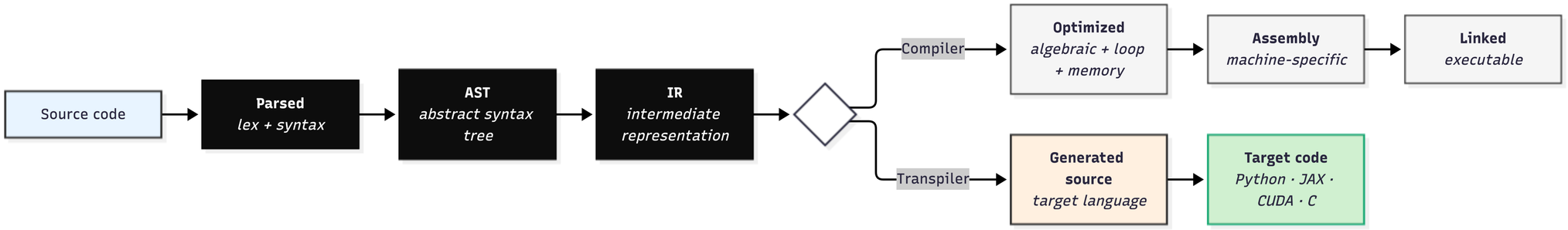
How do we transpile?

Tree is all you need!

How does a compiler see « a(j) = a(j) + j » ?



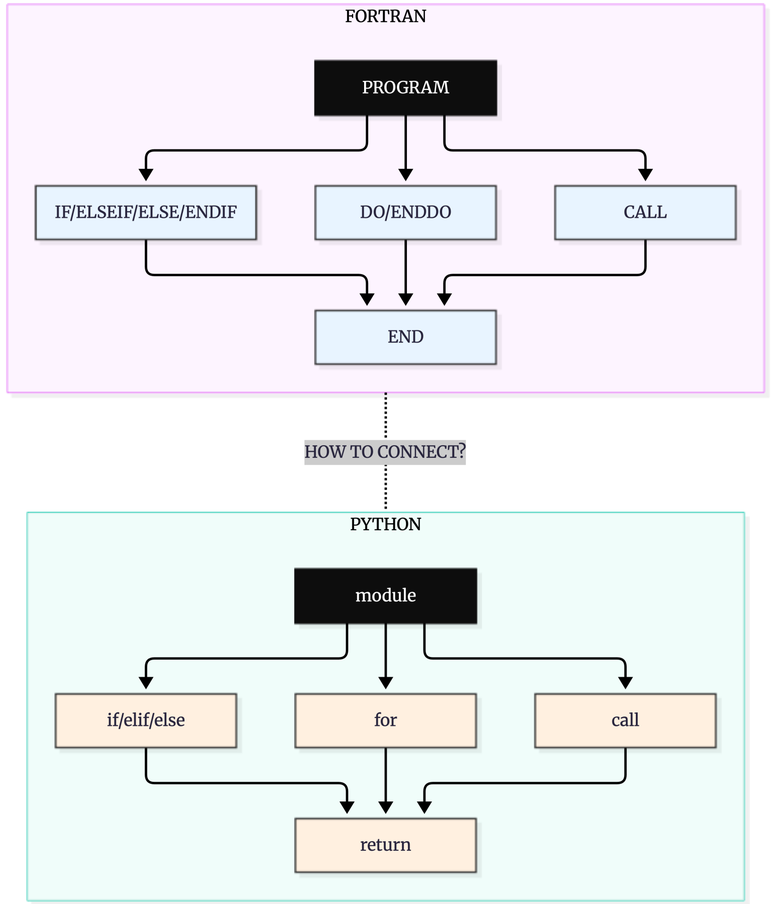
- That tree is called an Abstract Syntax Tree (AST),
- Every language has one: Fortran, C, Python,
- It's the structure underneath the syntax,
- It's how machines understand your code.



Same tree, two destinations: compiler or transpiler.

How to transpile a tree?

Different languages demand a translator!

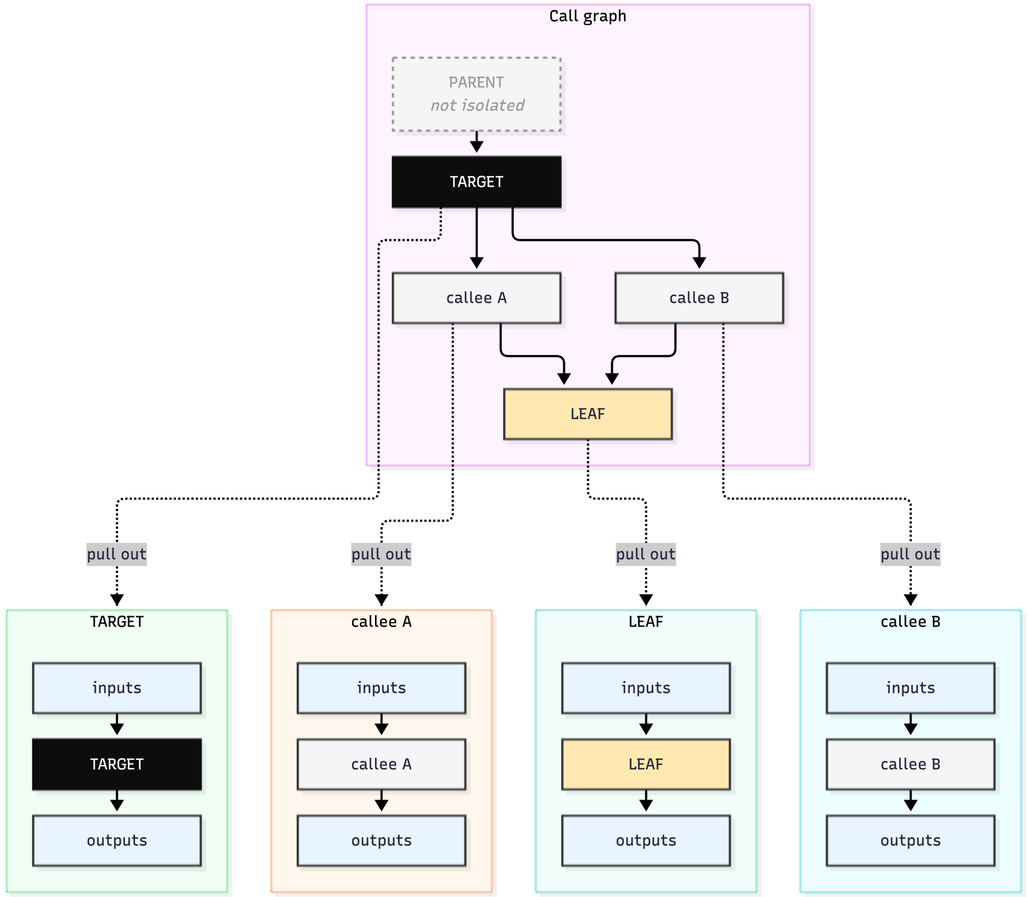


- Fortran → Fparser → Tree,
- Python → AST → Tree,
- Same idea, different representation,
- The real problem: the translator!

FGPT, One Transpiler pipeline for multiple purposes.

FGPT

Everything is wrapped around an isolation engine!

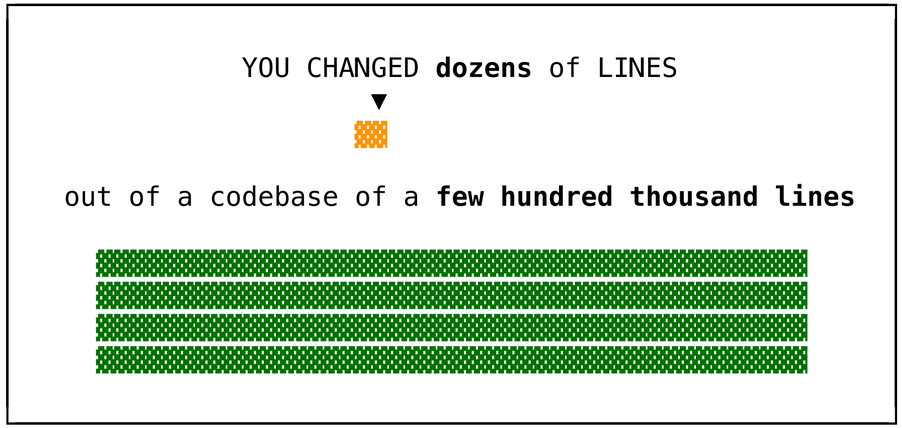


- It takes the Fortran code,
- It resolves the dependencies graph,
- It isolates the procedure from bottom to top,
- It generates standalone units,
- It compiles, runs, and generates benchmarks.

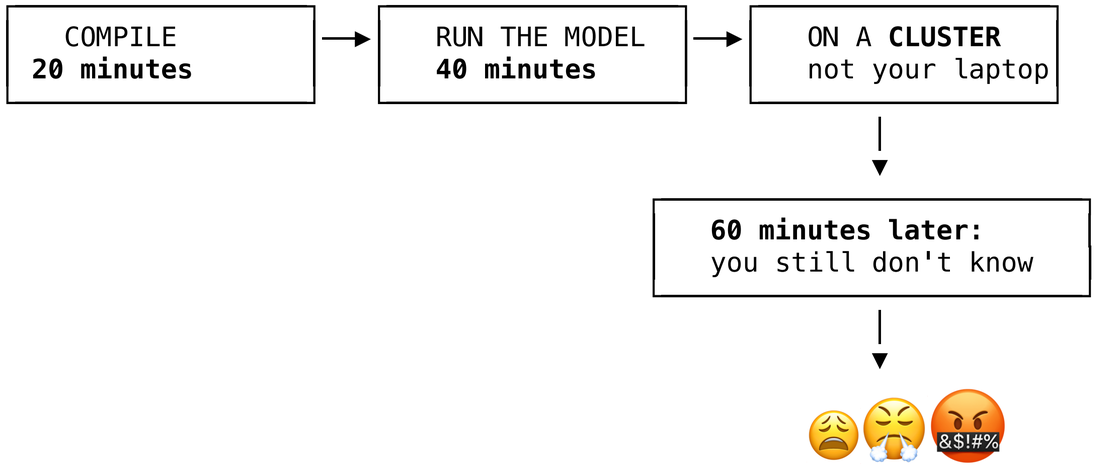
All automatic!

FGPT: Why Isolation?

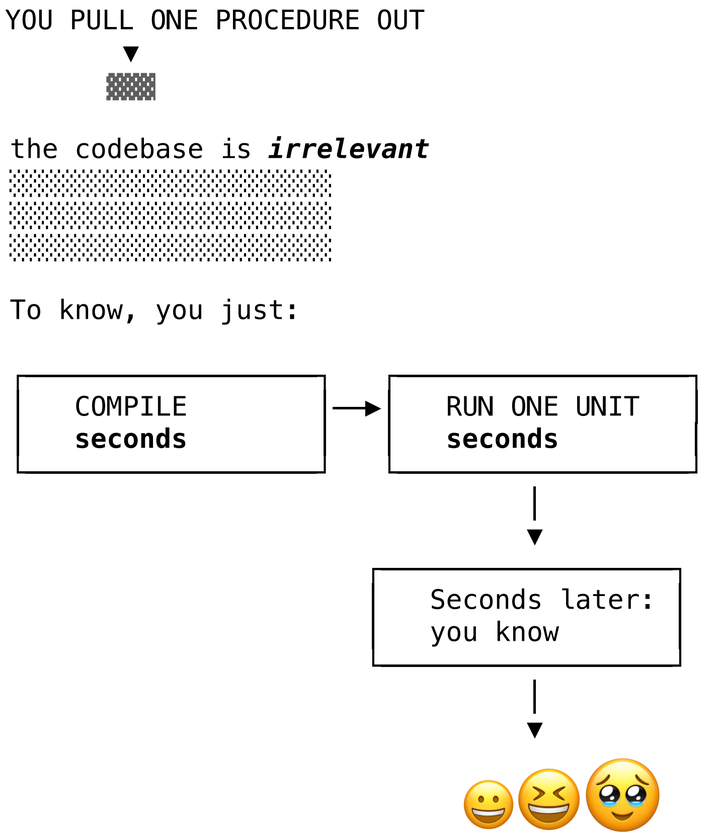
Verification is not optional!



To find out if it works, you have to ***turn on every debug option***:



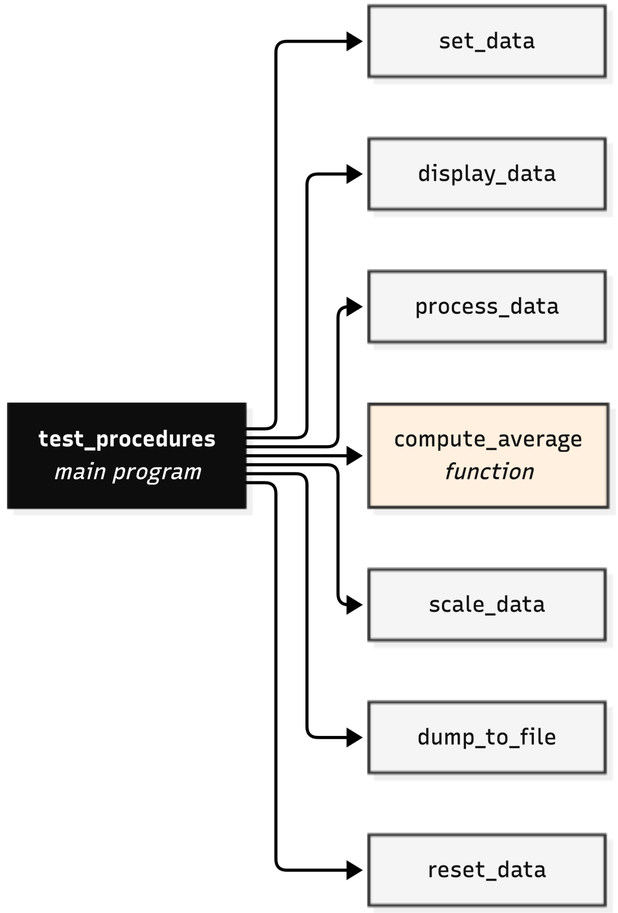
A standalone unit



Develop, Design unit test, Verify, Or don't bother!

FGPT: Isolation

Fgpt/examples/example_01.f90



- Run the isolator. Point it at the procedure you want,
- It does;
  - Backs up the original as _org.fgpt,
  - Instruments the original with benchmark I/O,
  - Compiles and runs the modified Fortran,
  - Generates dummy.bin, output.bin, time.txt,
- Run the isolated procedure. Same inputs. Compare the outputs.

```
$ python examples/example_01.py
└── dump_to_file
    ├── dump_to_file
    ├── dump_to_file.txt
    ├── main_dump_to_file.f90
    ├── mod
    │   └── module_global_dump_to_file.mod
    ├── module_global_dump_to_file.f90
    └── obj
        ├── adStack.o
        ├── main_dump_to_file.o
        └── module_global_dump_to_file.o
```

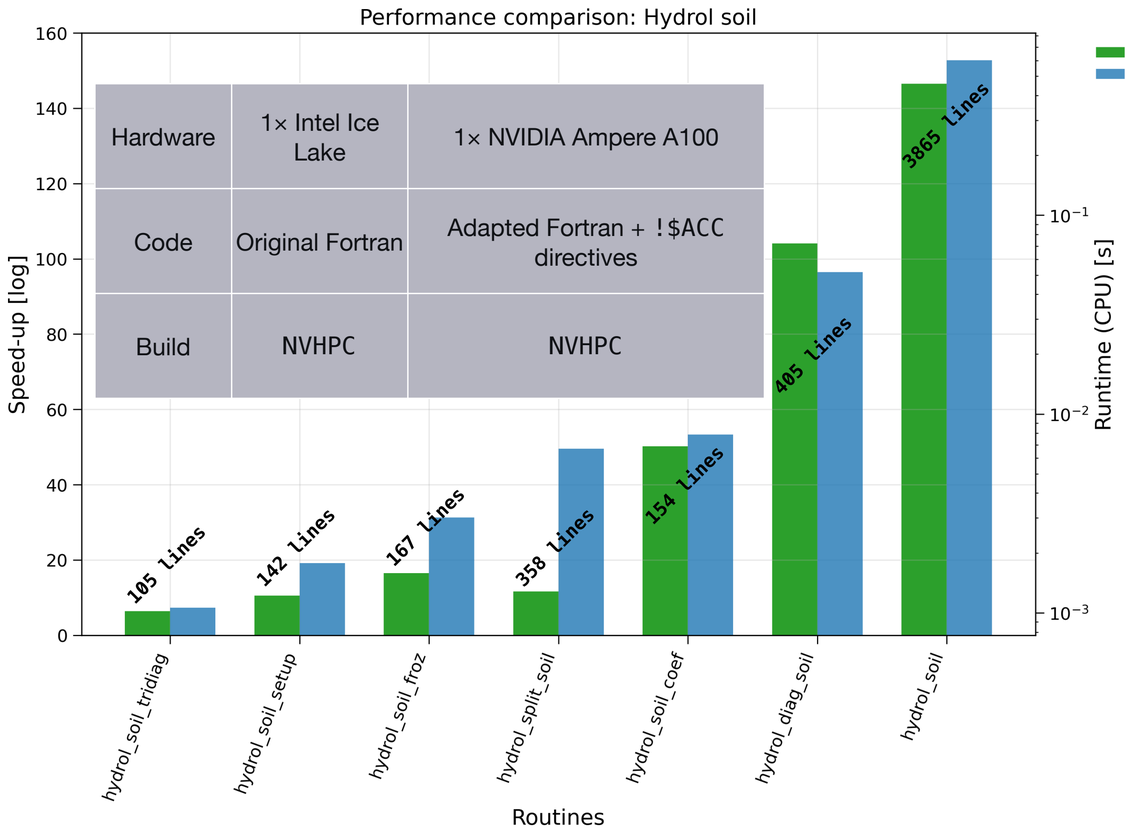
Standalone unit → Easy and rapid to verify!

FGPT: Isolation → GPU porting

90% of GPU porting is code adaptation and cleanup!

Guidelines for Fortran on heterogeneous architectures:
https://hal.science/hal-04193779/document

$$S = \frac{T_{\mathrm{CPU}}}{T_{\mathrm{GPU}}} > \frac{P_{\mathrm{node,GPU}}}{P_{\mathrm{node,CPU}}} \equiv S_{\mathrm{min}}, \qquad \frac{u_{\mathrm{G}}}{u_{\mathrm{C}}} > \frac{S_{\mathrm{min}}}{Q}, \qquad Q = \frac{R^{\mathrm{peak}}_{\mathrm{GPU}}}{R^{\mathrm{peak}}_{\mathrm{CPU}}}$$



Machine	P_node,GPU / P_node,CPU		Required u_G/u_C	Minimum speedup S_min
Jean Zay V100 (4×)	1,500 / 300 = 5.0	9.75	> 0.51	≈ 5×
Jean Zay A100 (8×)	3,650 / 450 = 8.1	27.0	> 0.30	≈ 8×
Jean Zay H100 (4×)	3,400 / 600 = 5.67	21.1	> 0.27	≈ 5.7×
Leonardo A100 (×4)	1,850 / 250 = 7.4	14.6	> 0.51	≈ 7.4×

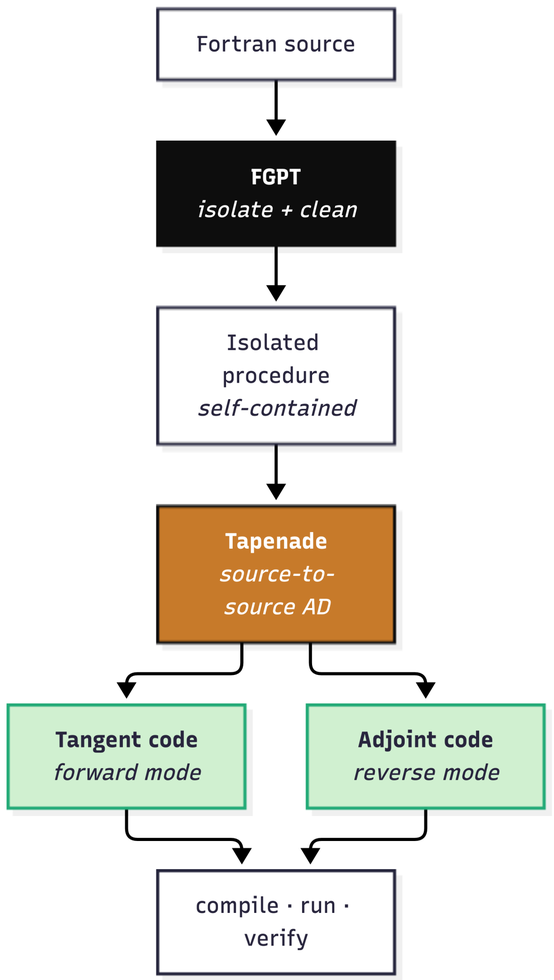
Estimates are for whole-node. Using a fraction of the node (e.g., 1 out of 32 CPU cores + 1GPU) → the threshold rises by some factor (e.g., 32/4 x 7.4 ≈ **60**).



CPU vs GPU is Speed vs Throughput.



FGPT: Isolation → Fortran AD



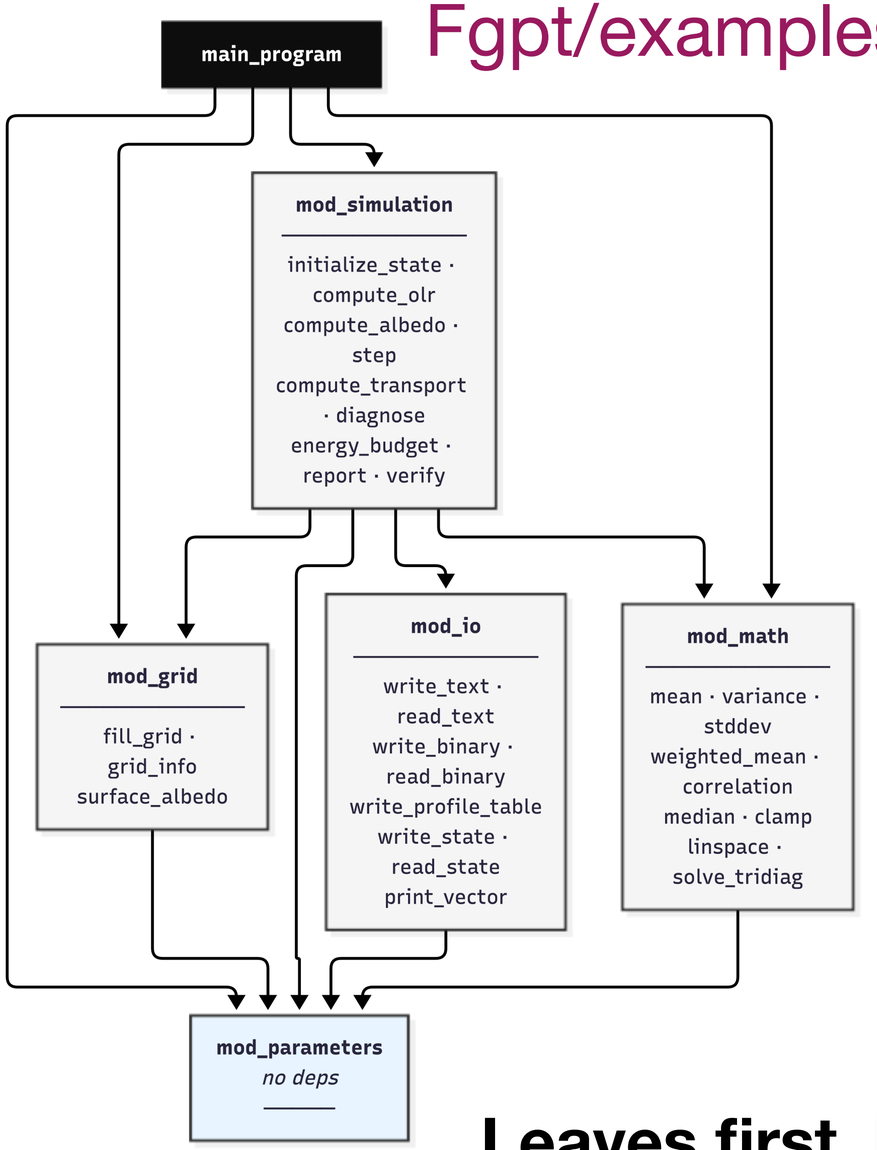
Tapenade AD: Free, Powerful, Built at INRIA

- Free,
- Fortran / C, Tangent / Adjoint mode, Source-to-source,
- Proven by MITGCM, NASA ECCO, Ocean circulation, etc,
- Analytical derivatives, Exact to machine precision.

Emulator has its limits (accumulation error is not trivial)
→ The adjoint model isn't dead!

FGPT: Isolation → Fortran AD

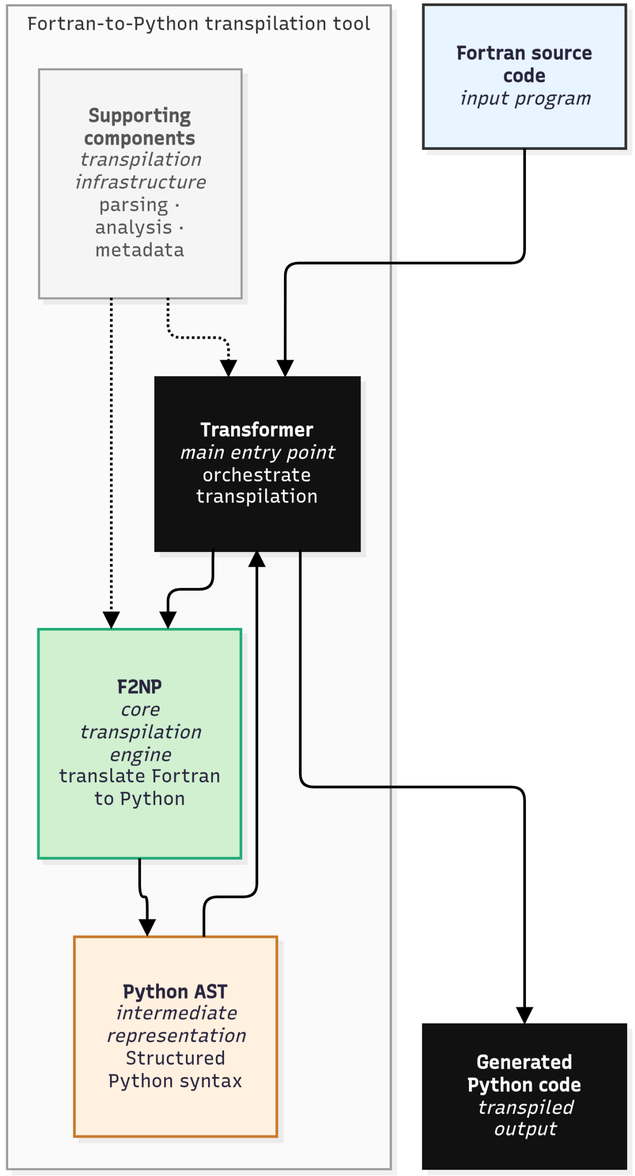
Fgpt/examples/dependencies/main_program.f90




```
$ python examples/dependencies.py
main_program/main_program/
├── main_main_program.f90          ← the driver (generated)
├── main_program                   ← the compiled executable
├── main_program.txt               ← compile + run log
├── mod
│   ├── module_global_main_program.mod
│   ├── module_global_main_program..._adj.mod
│   └── module_global_main_program..._tgt.mod
├── module_global_main_program_b.f90   ← Tapenade: adjoint model
├── module_global_main_program_b.msg   ← Tapenade report (adjoint)
├── module_global_main_program_d.f90   ← Tapenade: tangent model
├── module_global_main_program_d.msg   ← Tapenade report (tangent)
├── module_global_main_program.f90     ← the global module (generated)
└── obj
    ├── adStack.o                      ← Tapenade runtime
    ├── main_main_program.o
    ├── module_global_main_program_b.o
    ├── module_global_main_program_d.o
    └── module_global_main_program.o
```

Leaves first. Main last. Every unit, differentiated.

FGPT: Isolation → Fortran to Python

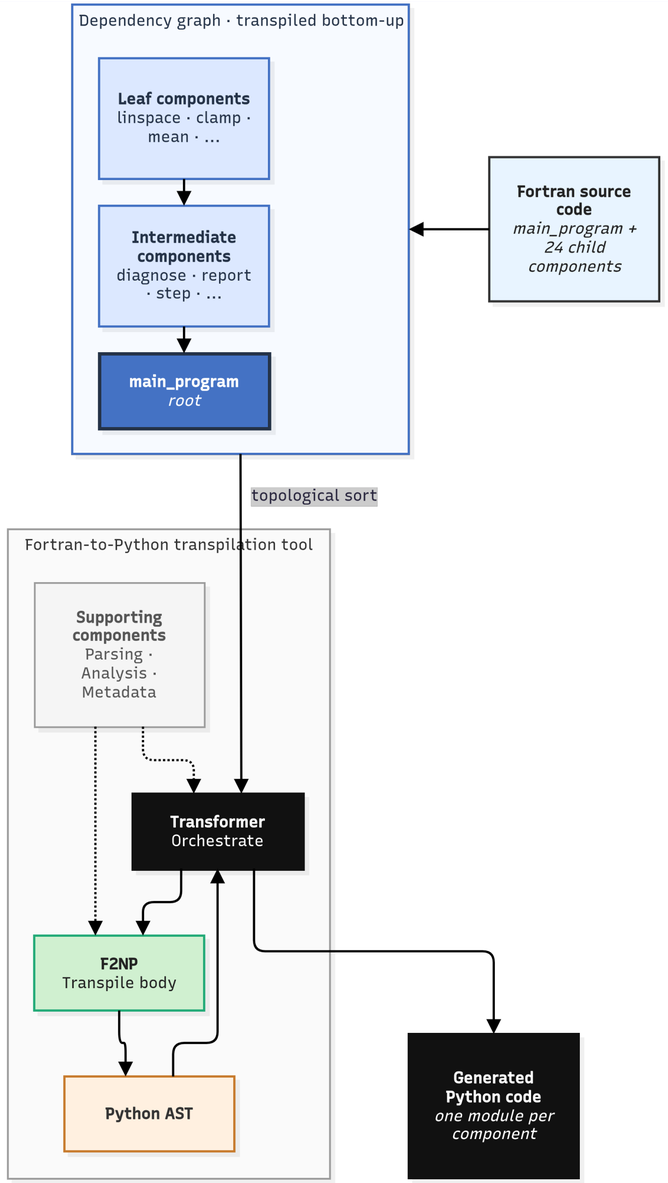


Fortran, reborn as Python!

- Bridging two worlds: Legacy Fortran becomes Python,
- Clean separation: The Transformer runs the pipeline, F2NP does the Transpilation,
- Semantics preserved: Same logic, Same structure, Just Python,
- No manual work: Testing, debugging, development, all automatic.
- Modernization unlocked: Legacy science code, native in the Python ecosystem.

Keep the algorithm. Transform the code.

FGPT: Isolation → Fortran to Python



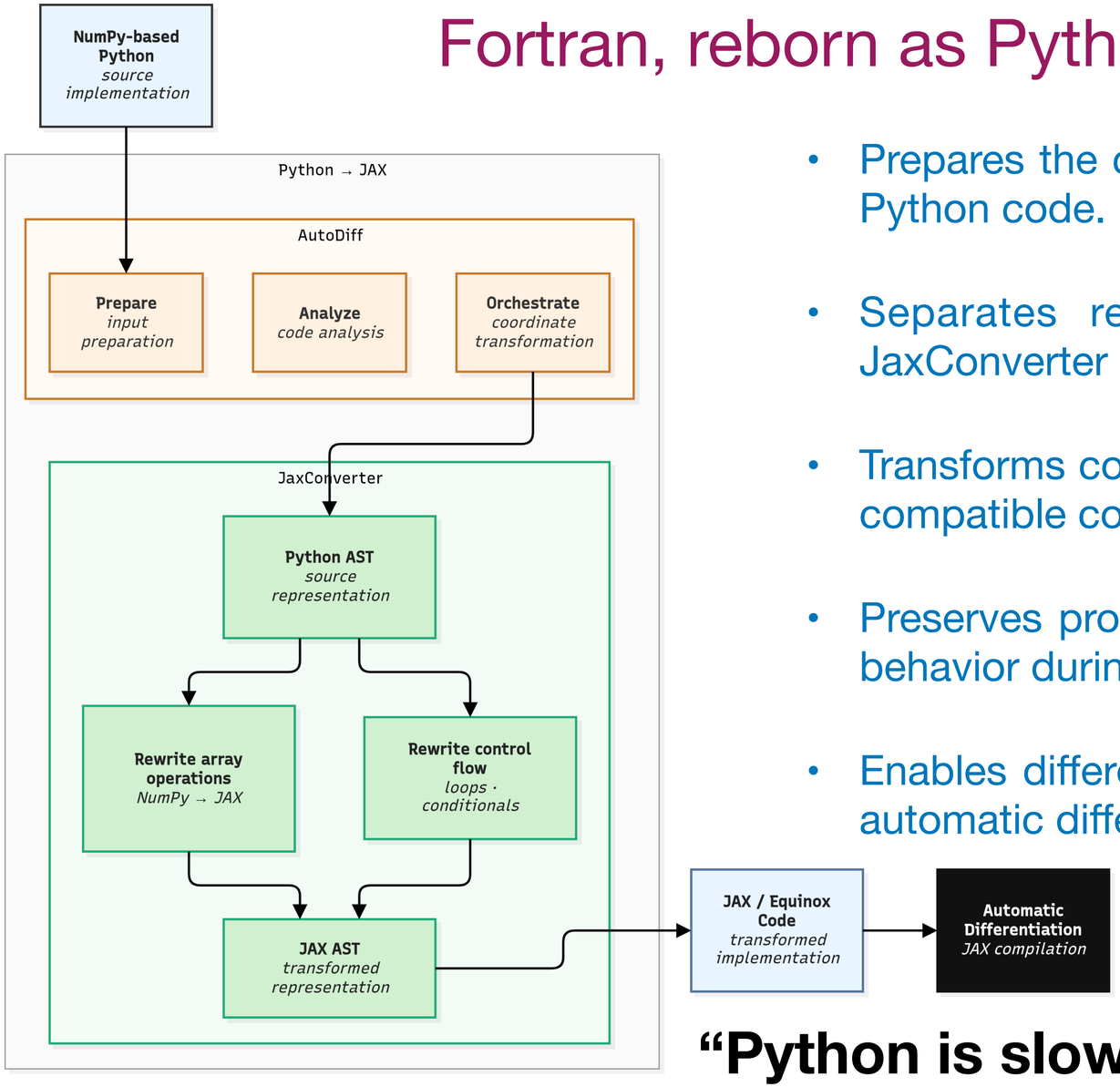
Fgpt/examples/dependencies

```
$ python examples/dependencies.py
main_program/main_program/
├── global_module_main_program.py ← Generated Python global class
├── main_main_program.f90
├── main_program
├── main_main_program.py          ← Generated Python main script
├── main_program.txt
├── module_global_main_program.f90
├── mod
│   └── module_global_main_program.mod
└── obj
    ├── adStack.o
    ├── main_main_program.o
    └── module_global_main_program.o
```

Full transformed and validated.

FGPT: Isolation → Fortran to JAX

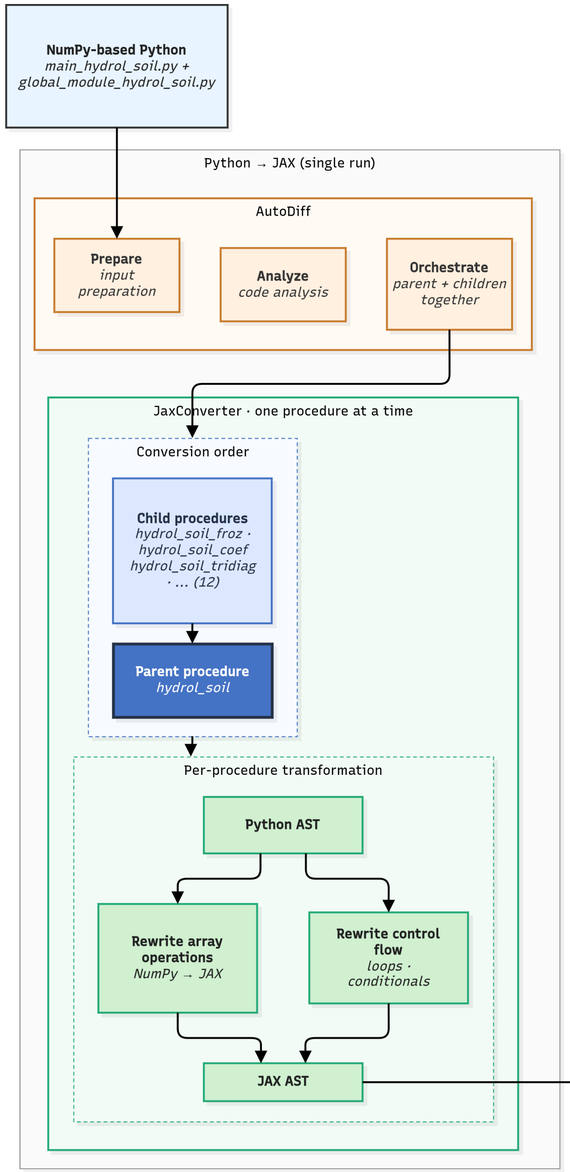
Fortran, reborn as Python. Accelerated by JAX.



- Prepares the code: AutoDiff prepares and analyzes the NumPy-based Python code.
- Separates responsibilities: AutoDiff orchestrates the workflow, JaxConverter handles code transformation.
- Transforms code: Rewrites control flow and array operations into JAX-compatible constructs.
- Preserves program logic: Aims to maintain the original computational behavior during conversion.
- Enables differentiation: Generates JAX/Equinox code compatible with automatic differentiation and compilation.

“Python is slow” is no longer valid.

FGPT: Isolation → Fortran to JAX



hydrol_soil/hydrol_soil

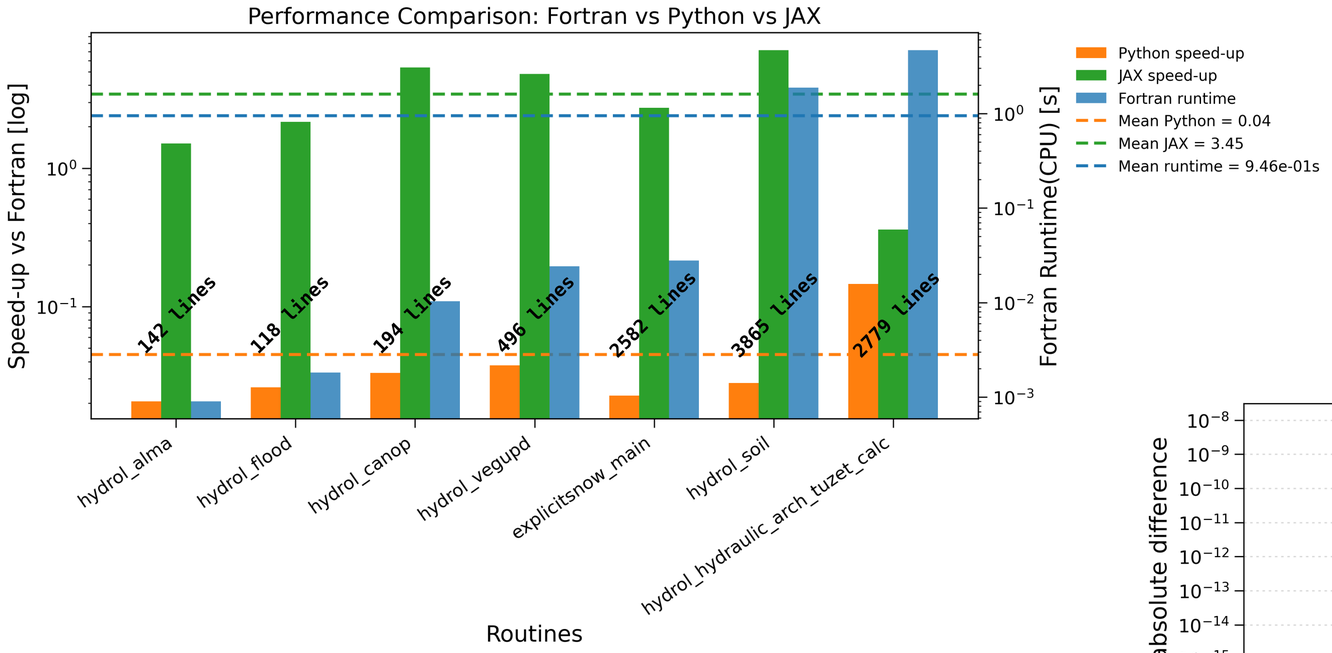
```
$ fgpt autodiff \
    --class_file hydrol/hydrol_soil/global_module_hydrol_soil.py \ ← Transformation command
    --main_file hydrol/hydrol_soil/main_hydrol_soil.py \
    --mode jax
        └── hydrol_soil
            ├── hydrol_soil
            ├── hydrol_soil.txt
            ├── global_module_hydrol_soil.py
            ├── global_module_hydrol_soil_jax.py      ← Generated JAX global class
            ├── main_hydrol_soil.f90
            ├── main_hydrol_soil.py
            ├── main_hydrol_soil_jax.py               ← Generated JAX main script
            ├── mod
            │   └── module_global_hydrol_soil.mod
            ├── module_global_hydrol_soil.f90
            └── obj
                ├── adStack.o
                ├── main_hydrol_soil.o
                └── module_hydrol_soil.o

$ python hydrol/hydrol_soil/main_hydrol_soil_jax.py ← Execution command
```

Full transformed and validated.

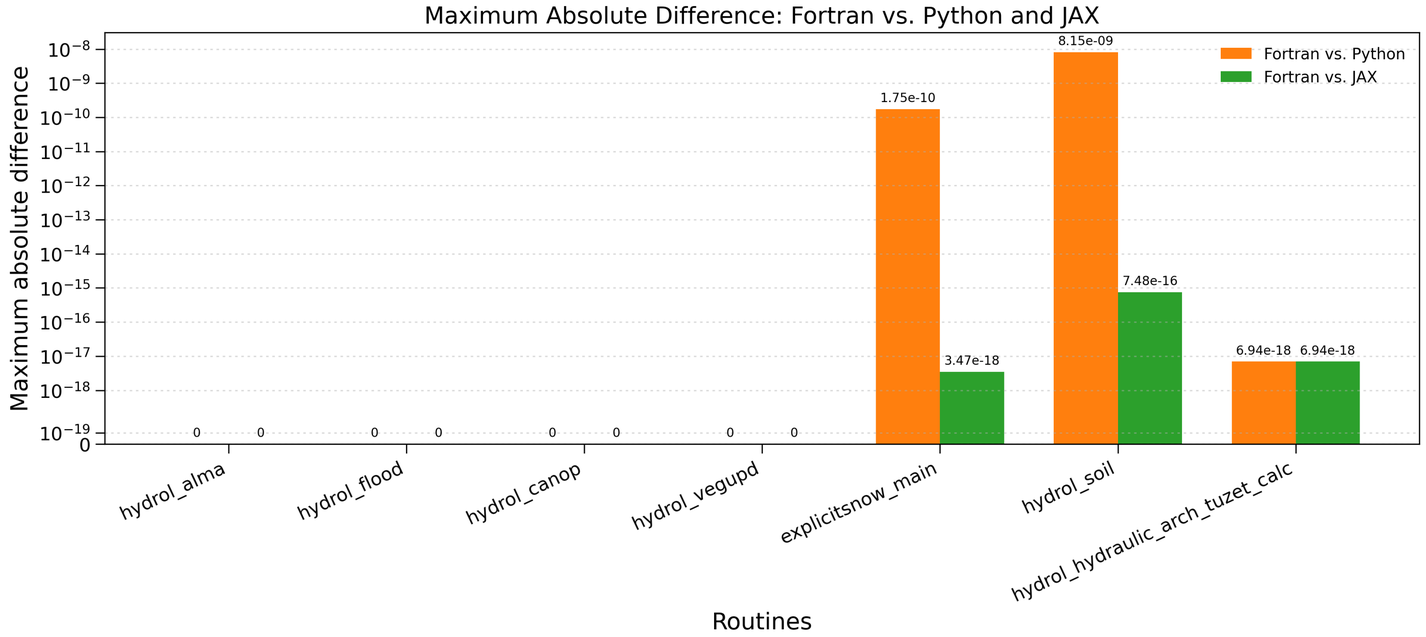
FGPT: Isolation → Fortran to JAX

JAX: Fortran speed, Python flexibility!



JAX outperforms both Python and Fortran, achieving an average speedup of **3.45**× to Fortran.

JAX achieves near-identical results to Fortran, with higher numerical agreement than Python.

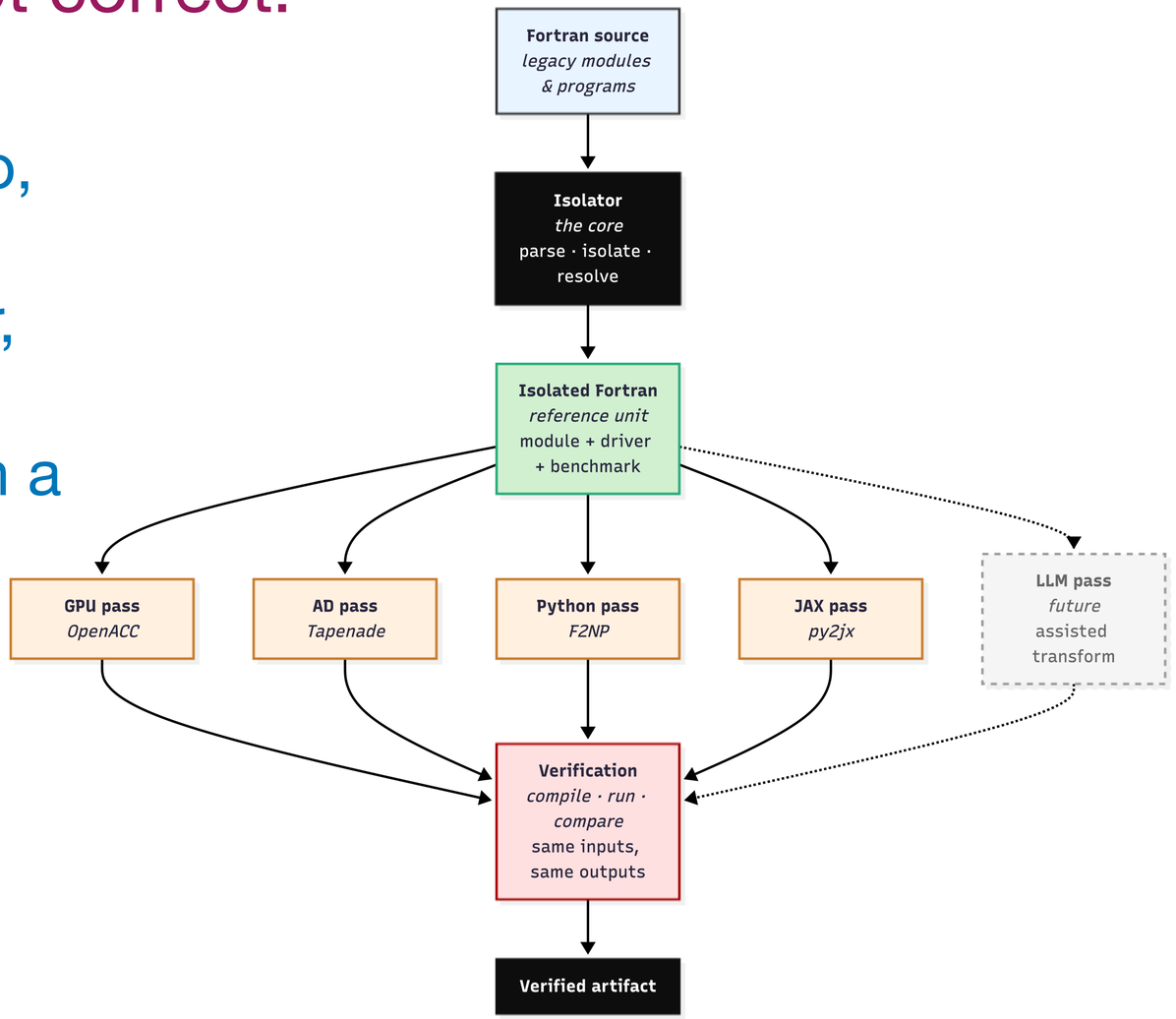


Forget ‘Python is slow.’ Just use JAX.

What about LLMs?

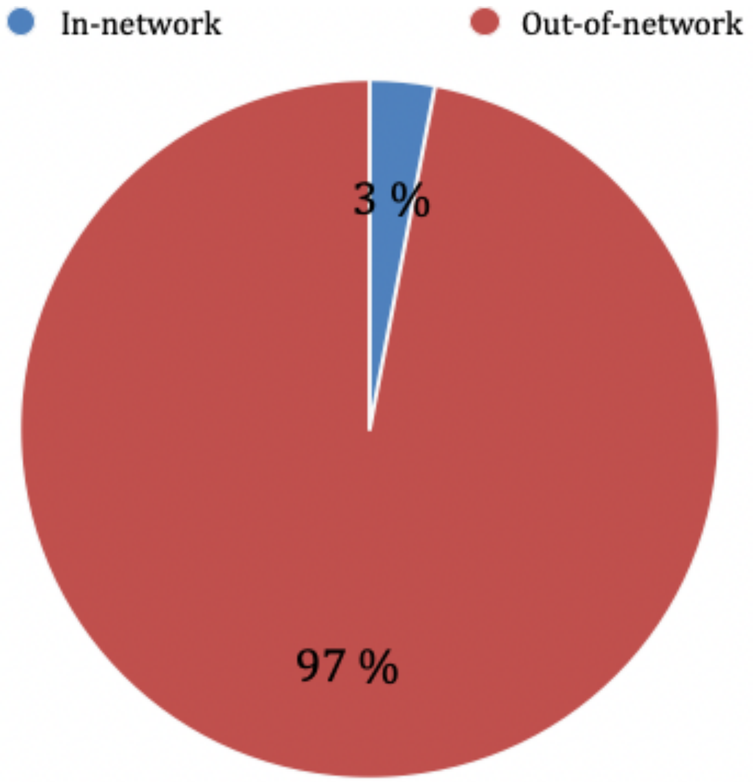
Plausible is not correct!

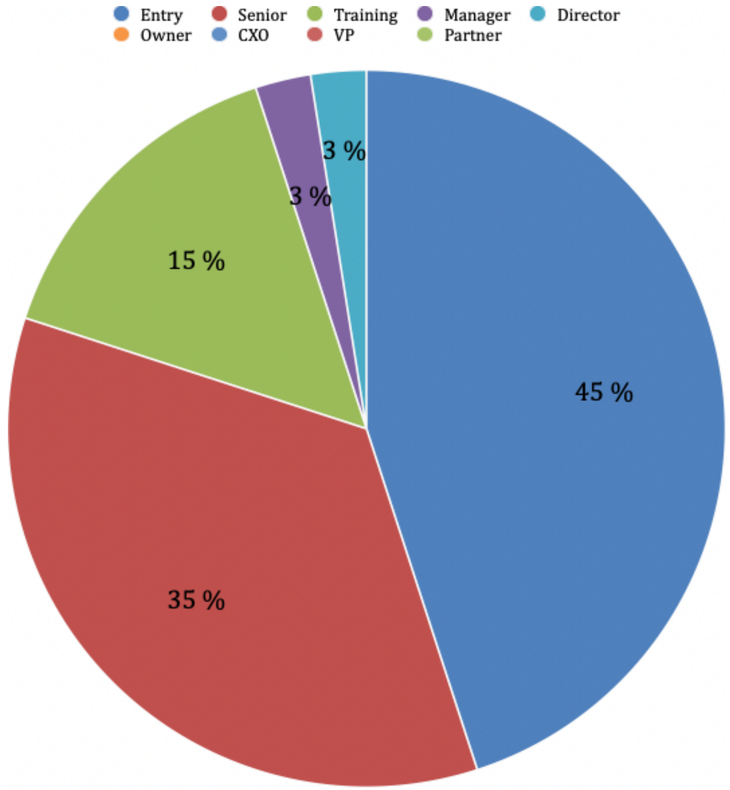
- It doesn't verify/validate the code → You do,
- Same prompt next week → different answer,
- The reasoning lives in the NN weights. Not in a scientific community,
- Every code, every prompt, every dialect, different,
- It can't be owned.

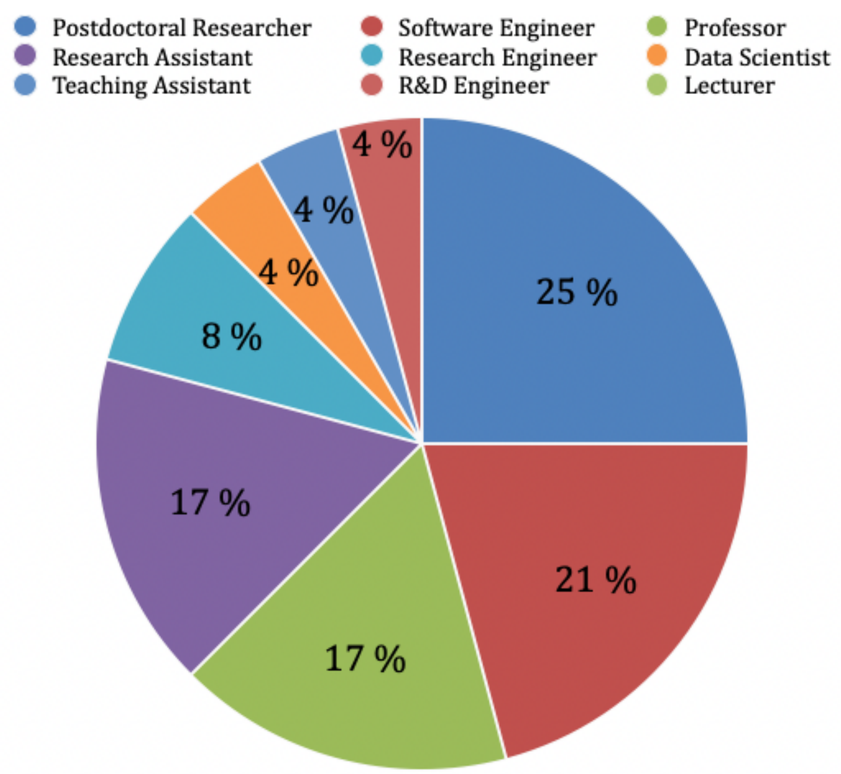


You pay full price N times, and you own nothing!

In the end, who cares?







Reached 44,456 LinkedIn members

Seniors as much as entry levels!

FGPT: Current Limitations

We built the engine, the rest is coming.

- Pointers, classes, blocks. Not yet. The AST handles them; the pipeline doesn't.
- Derived types. Partially supported. Simple cases work. Nested and polymorphic ones don't.
- Interface blocks. Partially supported. Explicit interfaces are read; generic resolution is not.
- NetCDF and HDF5 I/O. Coming. Today it works with standard Fortran I/O only.
- LLM pipeline. A locally installed model on the cluster — no external calls, no data leaving the machine.

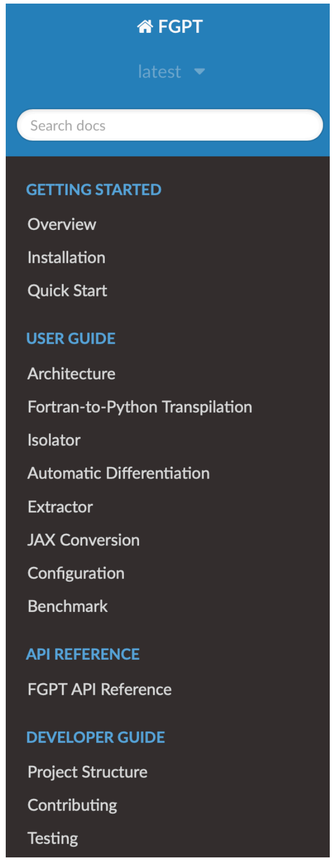
Others works

FGPT sits between compilers and agents: deterministic.

- **f2c**, **f2py**, **Pyccel**: Fortran to C, Python wrappers, Python to Fortran,
- **PSyclone**, **Loki**: Porting Fortran for GPUs, Fortran in, Fortran out,
- **Zhou et al**: GPT-4 translates CESM photosynthesis to JAX, Semi-automated,
- **Lahlou et al**: Agents translate 19,000 lines of CLM-ml to JAX,
- **legoESM**, **Gentine et al**: Agents, mostly Claude, write a JAX Earth system model,
- **Koldunov et al**: Claude ports FESOM2 to C++, MITgcm to JAX.

They wrap it, port it, or delegate to agents. FGPT parses it, then speaks Python and JAX.

Install, Examples, Scripts



Install

```
git clone https://github.com/kardaneh/Fgpt.git
cd Fgpt

uv venv --python 3.8
source .venv/bin/activate

uv pip install -e .
```

Install development dependencies:

```
uv pip install -e ".[dev]"
```

Install documentation dependencies:

```
uv pip install -e ".[docs]"
```

Verify installation:

```
python -c "import fgpt; print(fgpt.__version__)"
fgpt -help
```

Examples

`dependencies/` → **a full multi-module Fortran app**

`mod_parameters · mod_grid · mod_math · mod_io · mod_simulation · main_program`

Complete dependency tree, isolated bottom-up

`autodiff/Fortran/forward/` → **tangent linear AD**

`calibrate.f90 · module_global_multilevel_matrix_d.f90 · plots.py`

Forward-mode differentiation, end to end

`autodiff/Fortran/reverse/` → **adjoint AD**

`calibrate.f90 · module_global_multilevel_matrix_b.f90 · plots.py`

Reverse-mode differentiation, end to end

`example_01.f90 / .py`	→ **A procedure isolation**
`example_02.py`	→ **The full isolation pipeline, bottom-up**
`io_demo.f90 / .py`	→ **Fortran I/O isolated and verified**
`netcdf_demo.f90 / .py`	→ **NetCDF I/O, in progress**

Verify the result

```
python src/fgpt/core/common/executive.py --target_module Name_of_dir
```

Summary

- **Isolates any procedure:** from a module, a program, or a tangle of dependencies,
- **Resolves dependencies:** dummies, globals, shapes, intents, calls,
- **Generates standalone units:** module, driver, benchmark,
- **Verifies automatically:** same inputs, same outputs,
- **Transforms further:** Python, JAX, Fortran GPU, Fortran AD.

Thanks to:
Sven Rodriguez · Redouane Lguensat · Pascal Maugis · Frederic Chevallier · Fabienne Maignan · Philippe Peylin — for the discussions.
Horizon Europe AI4PEX — for the funding.
EuroHPC JU · LEONARDO · CINECA — for the machine.