Pyccel
Write Python code, get Fortran speed

Emily Bourne (**Sc**ientific **IT** and **A**pplication **S**upport, EPFL),
Yaman Güçlü (Max Planck **I**nstitute for **P**lasma **P**hysics, Garching)

Why Python?

- High-level programming language, interpreted, general-purpose, object-oriented
- Simple syntax with emphasis on code readability
- Widely used as integration language and for advanced scripting
- Extensive Python Standard Library, included in most Python distributions
- Huge ecosystem of third-party libraries (linear algebra, scientific computing, machine learning, etc.)

- Since 2021 most popular programming language worldwide, according to TIOBE & PYPL indices
    - Followed by: C, C++, Java, . . . , Fortran (#11), . . . , Julia (#25), . . .
    - Easy to find excellent students who already know Python, or want to learn it
    - Easy to find online documentation (examples, tutorials, manuals, best practice guidelines, etc.)

Why use an accelerator? Fortran vs Python

A is a large n × n real matrix with n=10000.

We compute the sum of its elements without using built-in or library functions.

Fortran is approx. ~145 times faster than Python.

- Python is powerful and concise, but slow with loops
- Fortran is verbose and must be compiled, but it is a great number cruncher

Python (pure):

```python
def sum(A, n):
    s = 0.
    for i in range(n):
        for j in range(n):
            s += A[i,j]
    return s
```

Time to solution ≈ 17s

Fortran

```fortran
function sum(A, n) result(s)
  implicit none
  integer, parameter :: wp = 8

  real(wp), intent(in) :: A(:,:)
  integer,  intent(in) :: n
  real(wp)             :: s

  integer :: i, j

  s = 0._wp
  do i = 1, n
    do j = 1, n
      s = s + A(i,j)
    end do
  end do
end function
```

Time to solution ≈ 0.1s

Pyccel was born out of frustration when going from prototype to production scientific code.

Pyccel is firstly a ***transpiler***. It currently translates to Fortran or C.

Using wrapping, translations are exposed back to Python creating an ***accelerator***.

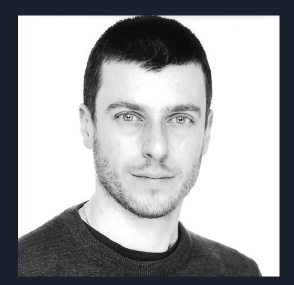
Pyccel can therefore be described as a static compiler for Python 3, using Fortran or C as backend language.

https://github.com/pyccel/pyccel

Who is Pyccel?

Original author :
Ahmed Ratnani
Director of the UM6P Vanguard Center

Original co-author :
Said Hadjout
PhD student at the Technische Universität München



Maintainer and administrator :
Yaman Güçlü
Staff Scientist at the Max Planck Institute for Plasma Physics

Maintainer and main developer :
Emily Bourne
HPC Application Expert at SCITAS, EPFL

1337

Current and previous Junior developers :
Students from 1337 New Generation Coding School, Morocco

Brief example - Ackermann

```python
def ackermann(m : int, n : int) -> int:
    """
    Solve the ackermann function
    """
    if m == 0:
        return n + 1
    elif n == 0:
        return ackermann(m - 1, 1)
    else:
        return ackermann(m - 1, ackermann(m, n - 1))
```

```
emily@olivier-UX410UQK:~/Code/test$ python3 -m timeit -s 'from ackermann import ackermann' 'ackermann(3,6)'
10 loops, best of 5: 19.6 msec per loop
emily@olivier-UX410UQK:~/Code/test$ python3 -m timeit -s 'from ackermann import ackermann' 'ackermann(3,7)'
Traceback (most recent call last):
  File "/usr/lib/python3.10/timeit.py", line 326, in main
    number, _ = t.autorange(callback)
  File "/usr/lib/python3.10/timeit.py", line 224, in autorange
    time_taken = self.timeit(number)
  File "/usr/lib/python3.10/timeit.py", line 178, in timeit
    timing = self.inner(it, self.timer)
  File "<timeit-src>", line 6, in inner
    ackermann(3,7)
  File "/home/emily/Code/test/./ackermann.py", line 7, in ackermann
    return ackermann(m - 1, ackermann(m, n - 1))
  File "/home/emily/Code/test/./ackermann.py", line 7, in ackermann
    return ackermann(m - 1, ackermann(m, n - 1))
  File "/home/emily/Code/test/./ackermann.py", line 7, in ackermann
    return ackermann(m - 1, ackermann(m, n - 1))
  [Previous line repeated 987 more times]
  File "/home/emily/Code/test/./ackermann.py", line 5, in ackermann
    return ackermann(m - 1, 1)
  File "/home/emily/Code/test/./ackermann.py", line 2, in ackermann
    if m == 0:
RecursionError: maximum recursion depth exceeded in comparison
emily@olivier-UX410UQK:~/Code/test$ pyccel ackermann.py
emily@olivier-UX410UQK:~/Code/test$ python3 -m timeit -s 'from ackermann import ackermann' 'ackermann(3,6)'
2000 loops, best of 5: 136 usec per loop
emily@olivier-UX410UQK:~/Code/test$ python3 -m timeit -s 'from ackermann import ackermann' 'ackermann(3,10)'
10 loops, best of 5: 33.7 msec per loop
emily@olivier-UX410UQK:~/Code/test$ pyccel ackermann.py --language=c
emily@olivier-UX410UQK:~/Code/test$ python3 -m timeit -s 'from ackermann import ackermann' 'ackermann(3,6)'
2000 loops, best of 5: 140 usec per loop
emily@olivier-UX410UQK:~/Code/test$ python3 -m timeit -s 'from ackermann import ackermann' 'ackermann(3,10)'
10 loops, best of 5: 36 msec per loop
emily@olivier-UX410UQK:~/Code/test$
```

Brief example: Ackermann

```c
#include "ackermann.h"
#include <stdlib.h>
#include <stdint.h>


/*.....................................*/
/*________________________________*/
/*                                */
/*    Solve the ackermann function */
/*                                */
/*________________________________*/
int64_t ackermann(int64_t m, int64_t n)
{
    if (m == INT64_C(0))
    {
        return n + INT64_C(1);
    }
    else if (n == INT64_C(0))
    {
        return ackermann(m - INT64_C(1), INT64_C(1));
    }
    else
    {
        return ackermann(m - INT64_C(1), ackermann(m, n - INT64_C(1)));
    }
}
/*.....................................*/
```

```fortran
module ackermann_0001

  use, intrinsic :: ISO_C_Binding, only : i64 => C_INT64_T
  implicit none

  contains

  !.....................................
  !________________________________!
  !                                !
  !    Solve the ackermann function!
  !                                !
  !________________________________!

  recursive function ackermann(m, n) result(Out_0001)

    implicit none

    integer(i64) :: Out_0001
    integer(i64), value :: m
    integer(i64), value :: n

    if (m == 0_i64) then
      Out_0001 = n + 1_i64
      return
    else if (n == 0_i64) then
      Out_0001 = ackermann(m - 1_i64, 1_i64)
      return
    else
      Out_0001 = ackermann(m - 1_i64, ackermann(m, n - 1_i64))
      return
    end if

  end function ackermann
  !.....................................

end module ackermann_0001
```

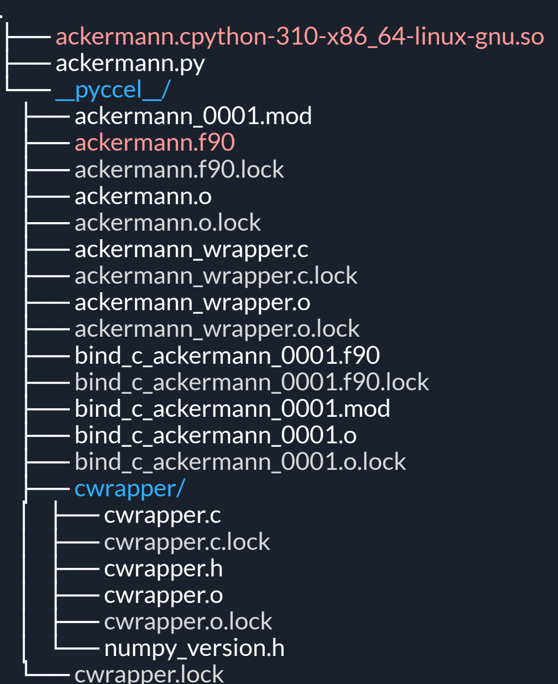
Brief example - Ackermann

The simplest way to run Pyccel is with the command line tool:

pyccel ackermann.py

Multiple files are generated:

- Translated code
- Locks (for thread safety)
- Wrappers (to act as a bridge between languages)
- Shared library (callable from Python)



Brief example : Vector expressions

```python
def triad(a : 'float[:,:]', b : 'float[:]', c : float):
    return a + b*c
```

```fortran
module addition


  use, intrinsic :: ISO_C_Binding, only : f64 => C_DOUBLE , i64 => &
        C_INT64_T
  implicit none

  contains

  !........................................
  subroutine triad(a, b, c, Out_0001)

    implicit none

    real(f64), allocatable, intent(out) :: Out_0001(:,:)
    real(f64), intent(in) :: a(0:,0:)
    real(f64), intent(in) :: b(0:)
    real(f64), value :: c
    integer(i64) :: i

    allocate(Out_0001(0:size(a, 1_i64, i64) - _i64, 0:size(a, 2_i64, &
          i64) - 1_i64))
    do i = 0_i64, size(Out_0001, 2_i64, i64) - 1_i64, 1_i64
      Out_0001(:, i) = a(:, i) + b * c
    end do
    return

  end subroutine triad
  !........................................

end module addition
```

Brief example : Vector expressions

```python
def triad(a : 'float[:,:]', b : 'float[:]', c : float):
    return a + b*c
```

```c
#include "addition.h"
#include <stdlib.h>
#include "ndarrays.h"
#include <stdint.h>


/*........................................*/
t_ndarray triad(t_ndarray a, t_ndarray b, double c)
{
    int64_t i;
    int64_t i_0001;
    t_ndarray Out_0001 = {.shape = NULL};
    Out_0001 = array_create(2, (int64_t[]){a.shape[INT64_C(0)],
                            a.shape[INT64_C(1)]}, nd_double, false, order_c);
    for (i = INT64_C(0); i < Out_0001.shape[INT64_C(0)]; i += INT64_C(1))
    {
        for (i_0001 = INT64_C(0); i_0001 < Out_0001.shape[INT64_C(1)]; i_0001 += INT64_C(1))
        {
            GET_ELEMENT(Out_0001, nd_double, (int64_t)i, (int64_t)i_0001) =
                  GET_ELEMENT(a, nd_double, (int64_t)i, (int64_t)i_0001) +
                  GET_ELEMENT(b, nd_double, (int64_t)i_0001) * c;
        }
    }
    return Out_0001;
}
/*........................................*/
```

Competitors

There are many other tools for accelerating Python. The most popular are:

- Cython : generates fast C code, if an intermediate “Cython” language is used
- PyPy : an alternative interpreter
- Numba : a just-in-time compiler for Python functions, based on LLVM
- Pythran : translates Python code to some heavily templated C++ code

Honourable mention:

- Julia

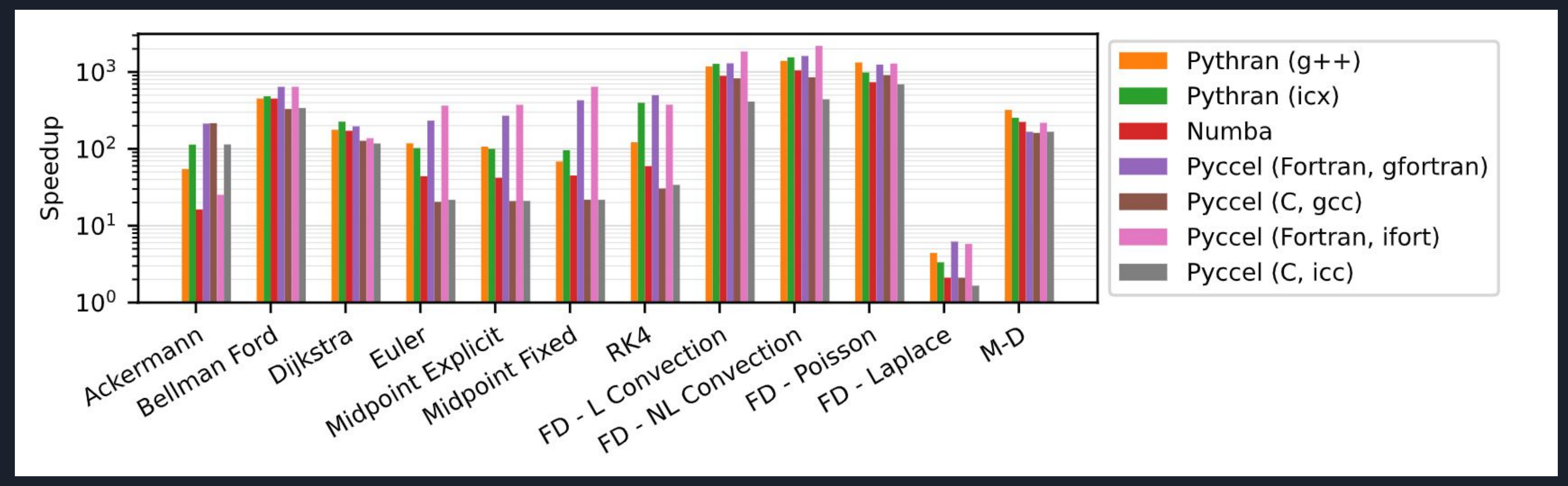
Should I use Pyccel?

Two main questions:

1. How much can I gain?
2. How much effort is required?

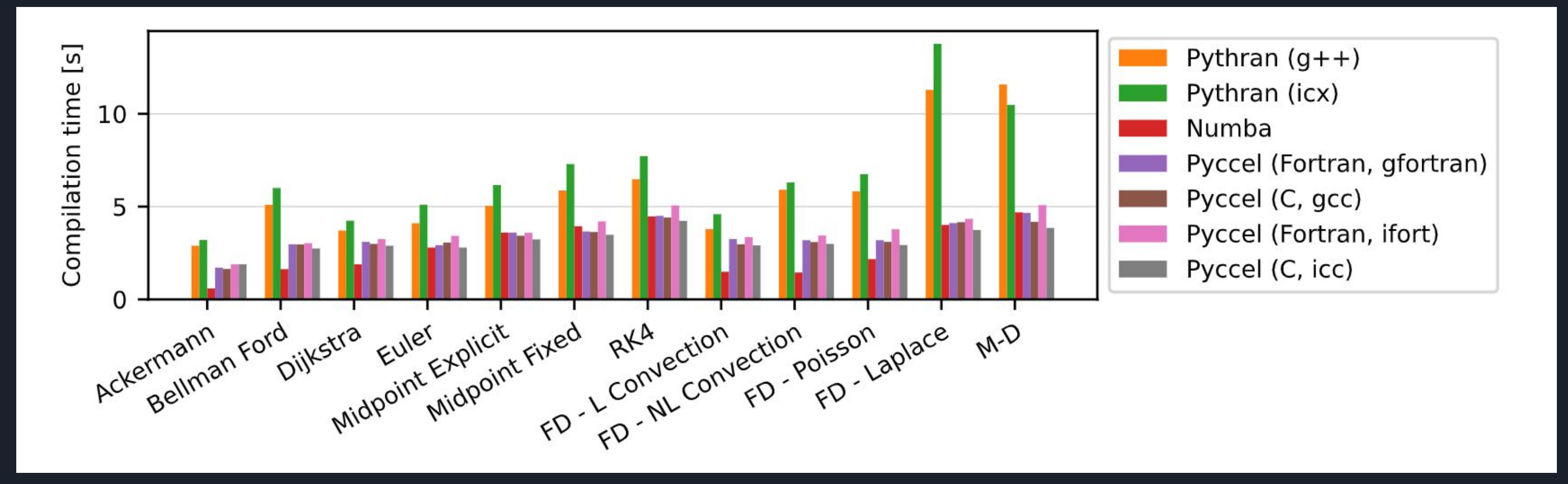
Tool	Requires translation	Easy to use on Supercalculator	Type annotations	Isolate kernels	Handles multiple folders	Interface with pure Python code	Interface with C code	Gain
Cython	Yes	Yes	Yes	Yes	Yes	Yes	Yes	+++
PyPy	No	No	No	No	Yes	No	No	+
Numba	No	No	Yes	Yes	Yes	Yes	No	++
Pythran	No	Yes	Yes	Yes	Only sub-folders	No	No	+++
Pyccel	No	Yes	Yes	Yes	Yes	No	Yes	+++

How much can I gain vs NumPy?



https://github.com/pyccel/pyccel-benchmarks

How long does that take?

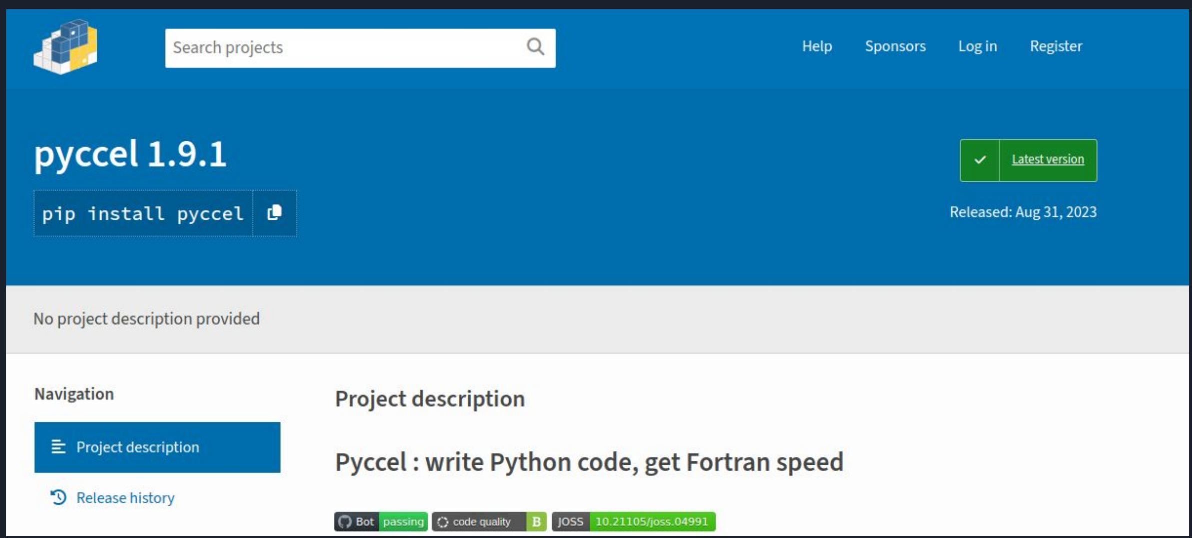


https://github.com/pyccel/pyccel-benchmarks

Installation

1. Install dependencies (compilers)
2. Install using Pip (https://pypi.org/project/pyccel/)

```
pip install pyccel
```



Getting started

1. Isolate bottlenecks which can be handled by Pyccel
2. Ensure correct Pyccel-friendly type annotation is used
3. Run Pyccel on your file with the command : *pyccel my_file.py*
4. Enjoy the speed-up!

Type annotations

Supported types:

- built-in data types: bool, int, float, complex

```
def func(a : int):
```

```
def func(a : 'int'):
```

- NumPy integer types: int8, int16, int32, int64
- NumPy real types: float32, float64, double
- NumPy complex types: complex64, complex128

```
def func(a : 'int16'):
```

- NumPy arrays of any of the above

```
def func(arr : 'int[:,:]'):
```

```
def func(arr : 'int[:,:](order=C)'):
```

```
def func(arr : 'int[:,:](order=F)'):
```

Function templating

The same function can be used with multiple types thanks to the *template* decorator

Similar to the *UnionType* but allows for more fine-grain control

```
from pyccel.decorators import template

@template(name='T', types=[int,float])
@template(name='Z', types=[int,float])
def f(a : 'T', b : 'Z'):
      pass
```

```
from pyccel.decorators import template

@template(name='T', types=[int,float])
def f(a : 'T', b : 'T'):
      pass
```

Limits

Limitations to be fixed soon:

- Classes (available in 1 or 2 release cycles)
- Non-HPC structures (lists/dicts/sets) - Watch this space (hopefully end of the year)
- Union types

Permanent limitations

- Type changes
- Non-homogeneous lists
- Exceptions (unless heavily requested)
- Plotting etc

Supported Python Packages

Well supported:

- cmath
- math
- NumPy

Minimal support:

- *product* from itertools
- *exit* from sys

Other useful information

Pyccel can be tested in interactive mode with epyccel (soon also on a website)

Neat error messages

```
def funct_c( x : 'const int', a : int ):
    x += a
    return x
```

```
ERROR at annotation (semantic) stage
pyccel:
 |fatal [semantic]: CONST_ASSIGNED_ARGUMENT.py [3]| Cannot modify 'const' argument (x)
```

Reasonably extensive documentation : https://github.com/pyccel/pyccel/tree/devel/docs

Support for new features can be requested at https://github.com/pyccel/pyccel/discussions

OpenMP Support

Pyccel contains support for OpenMP 5 pragmas (see docs for details).

OpenMP functions can be imported and accessed via Pyccel

```python
def get_num_threads(n : int):
    from pyccel.stdlib.internal.openmp import omp_set_num_threads,
                                              omp_get_num_threads, omp_get_thread_num
    omp_set_num_threads(n)
    #$ omp parallel
    print("hello from thread number:", omp_get_thread_num())
    result = omp_get_num_threads()
    #$ omp end parallel
    return result

if __name__ == '__main__':
    x = get_num_threads(4)
    print(x)
```

OpenMP - Example

```python
def my_sum(A: 'float[:,:]', n : int):
    s = 0.
    #$ omp parallel for collapse(2) reduction(+:s)
    for i in range(n):
        for j in range(n):
            s += A[i,j]
    return s
```

Time to solution ≈ 17s

```python
def my_sum(A: 'float[:,:]', n : int):
    return sum(A)
```

Time to solution ≈ 0.06s

```fortran
function my_sum(A, n) result(s)

   implicit none

   real(f64) :: s
   real(f64), intent(in) :: A(0:,0:)
   integer(i64), value :: n
   integer(i64) :: i
   integer(i64) :: j

   s = 0._f64
   !$omp parallel do collapse(2) reduction(+:s)
   do i = _i64, n - _i64, _i64
     do j = _i64, n - _i64, _i64
       s = s + A(j, i)
     end do
   end do
   !$omp end parallel do
   return

 end function sum
```

Time to solution without OpenMP ≈ 0.12s
Time to solution with 2 OpenMP threads ≈ 0.06s

Successes

https://joss.theoj.org/papers/10.21105/joss.04991

Pyccel: a Python-to-X transpiler for scientific high-performance computing

Emily Bourne [1*], **Yaman Güçlü** [2*¶], **Said Hadjout** [2,3*], and **Ahmed Ratnani** [4*]

1 CEA, IRFM, F-13108 Saint-Paul-lez-Durance, France **2** NMPP division, Max-Planck-Institut für Plasmaphysik, Garching bei München, Germany **3** Dept. of Mathematics, Technische Universität München, Garching bei München, Germany **4** Lab. MSDA, Mohammed VI Polytechnic University, Benguerir, Morocco ¶ Corresponding author * These authors contributed equally.

Successes

Psydac - Spline finite element library : https://github.com/pyccel/psydac/

Struphy - STRUcture-Preserving HYbrid codes - a Python package for energetic particles in plasma.

https://doi.org/10.1007/978-3-031-38299-4_28

ARTICLE

High-Order Structure-Preserving Algorithms for Plasma Hybrid Models

Authors: Stefan Possanner, Florian Holderied, Yingzhe Li, Byung Kyu Na, Dominik Bell, Said Hadjout, Yaman Güçlü Authors Info & Claims

Geometric Science of Information: 6th International Conference, GSI 2023, St. Malo, France, August 30 – September 1, 2023, Proceedings, Part II • Aug 2023 • Pages 263–271 • https://doi.org/10.1007/978-3-031-38299-4_28

Published: 30 August 2023 Publication History

0 0

PyGyro - Highly parallel drift-kinetic Semi-Lagrangian Simulations in Python (in review)

Study Case - PyGyro

Parallel drift-kinetic Semi-Lagrangian simulation modelling the following equations in 4D:

$$\partial_t f + \{\phi, f\} + v_{\|} \vec{\nabla}_{\|} f - \vec{\nabla}_{\|} \phi \, \partial_{v_{\|}} f = 0$$

$$\{\phi, f\} = -\frac{\partial_\theta \phi}{r B_0} \partial_r f + \frac{\partial_r \phi}{r B_0} \partial_\theta f$$

$$-\left[\partial_r^2 \phi + \left(\frac{1}{r} + \frac{\partial_r n_0}{n_0}\right) \partial_r \phi + \frac{1}{r^2} \partial_\theta^2 \phi\right] + \frac{1}{T_e} \phi = \frac{1}{n_0} \int_{-\infty}^{+\infty} (f - f_{eq}) dv_{\|}$$

https://github.com/pyccel/pygyro

Study Case - PyGyro

```
pygyro/
├── advection
│   ├── advection.py
│   ├── convergence_test_advection.py
│   ├── __init__.py
│   ├── test_advection.py
│   ├── time_test_advection.py
│   └── visual_test_advection.py
├── initialisation
│   ├── constants.py
│   ├── initialiser.py
│   ├── __init__.py
│   ├── setups.py
│   ├── test_setup.py
│   └── visual_test_setup.py
├── __init__.py
├── model
│   ├── grid.py
│   ├── __init__.py
│   ├── layout.py
│   ├── process_grid.py
│   ├── test_grid.py
│   ├── test_layout.py
│   └── test_process_setup.py
├── poisson
│   ├── convergence_test_poisson_solver.py
│   ├── __init__.py
│   ├── poisson_solver.py
│   ├── test_poisson_solver.py
│   └── visual_test_poisson_solver.py
├── splines
│   ├── __init__.py
│   ├── spline_interpolators.py
│   ├── splines.py
│   └── tests/
└── utilities
    ├── discrete_slider.py
    ├── grid_plotter.py
    ├── __init__.py
    ├── savingTools.py
    ├── test_saveTools.py
    └── visual_test_plots.py
```

https://github.com/pyccel/pygyro

Function	Total time excluding sub functions [s]	Number of calls	Time per call [s]	Total time[s]
method ’Alltoall’ from mpi4py	28.193	250	0.113	28.193
numpy.core.multiarray.array	16.070	5689347	0.000	16.070
bisplev from scipy	10.009	1006666	0.000	37.819
method scipy.interpolate. fitpack. bispev	8.503	1006666	0.000	8.503
atleast 1d from numpy	7.990	2915166	0.000	23.820
splev	7.334	901833	0.000	18.883
reshape from numpy	6.590	3397927	0.000	6.590

https://github.com/pyccel/pygyro

Study Case - PyGyro

Splines are the bottleneck but they come from a library.

- Write a (more targeted) spline implementation
- Translate the implementation

It is not necessary to translate everything, matrix interpolation equations are still handled in Python.

The class wrapper in Python still allows for easy use.

https://github.com/pyccel/pygyro

Study Case - PyGyro

Function	Total time excluding sub functions [s]	Number of calls	Time per call [s]	Total time[s]
method 'Alltoall' from mpi4py	10.461	250	0.042	10.461
numpy.core.multiarray.array	9.340	1624947	0.000	9.340
eval in Spline2D	7.225	1006666	0.000	12.345
step in PoloidalAdvection	5.423	3333	0.002	35.266
eval in Spline1D	4.948	901833	0.000	6.319
step in FluxSurfaceAdvection	4.744	5000	0.001	9.527
_vectorize_call	4.403	38333	0.000	25.365

https://github.com/pyccel/pygyro

Study Case - PyGyro

Advection functions depend on splines. As a result lots of arguments are sometimes necessary (this should improve once classes are implemented).

```
def poloidal_advection_step_expl(f: 'float[:,:]',
                                 dt: 'float', v: 'float',
                                 rPts: 'float[:]', qPts: 'float[:]',
                                 drPhi_0: 'float[:,:]', dthetaPhi_0: 'float[:,:]',
                                 drPhi_k: 'float[:,:]', dthetaPhi_k: 'float[:,:]',
                                 endPts_k1_q: 'float[:,:]', endPts_k1_r: 'float[:,:]', endPts_k2_q: 'float[:,:]', endPts_k2_r: 'float[:,:]',
                                 kts1Phi: 'float[:]', kts2Phi: 'float[:]', coeffsPhi: 'float[:,:]', deg1Phi: 'int', deg2Phi: 'int',
                                 kts1Pol: 'float[:]', kts2Pol: 'float[:]', coeffsPol: 'float[:,:]', deg1Pol: 'int', deg2Pol: 'int',
                                 CN0: 'float', kN0: 'float', deltaRN0: 'float', rp: 'float', CTi: 'float',
                                 kTi: 'float', deltaRTi: 'float', B0: 'float', nulBound: 'bool' = False):
```

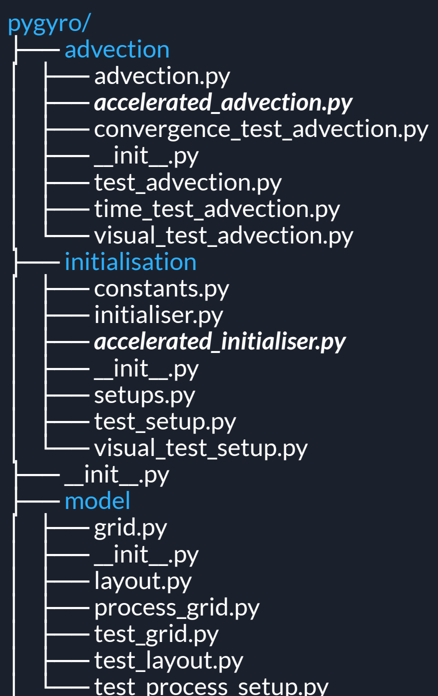
https://github.com/pyccel/pygyro

Study Case - PyGyro : Final timings

Function	Total time excluding sub functions [s]	Number of calls	Time per call [s]	Total time[s]
method 'Alltoall' from mpi4py	2.735	250	0.011	2.735
step in FluxSurfaceAdvection	1.826	5000	0.000	3.355
parallel_gradient in ParallelGradient	1.492	1000	0.001	1.903
step in PoloidalAdvection	1.420	3333	0.002	3.004
Method 'solve' from scipy 'SuperLU'	1.294	96666	0.000	1.294
getPerturbedRho in ParallelGradient	1.022	101	0.010	2.307
_solve_system_nonperiodic in SplineInterpolator1D	0.977	123700	0.000	0.977

https://github.com/pyccel/pygyro

Study Case - PyGyro



- poisson
  - convergence_test_poisson_solver.py
  - __init__.py
  - poisson_solver.py
  - ***accelerated_poisson_solver.py***
  - test_poisson_solver.py
  - visual_test_poisson_solver.py
- splines
  - __init__.py
  - spline_interpolators.py
  - splines.py
  - ***accelerated_splines.py***
  - tests/
- utilities
  - discrete_slider.py
  - grid_plotter.py
  - __init__.py
  - savingTools.py
  - test_saveTools.py
  - visual_test_plots.py

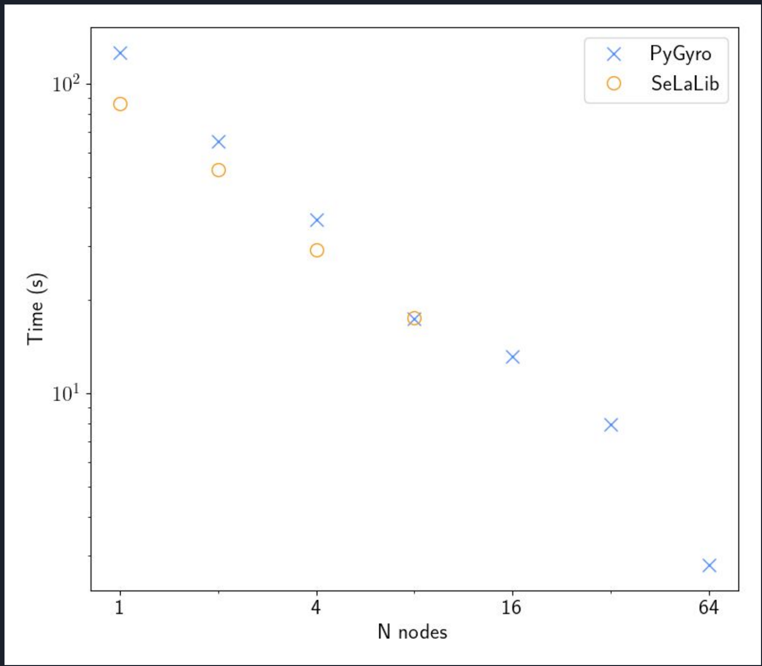
https://github.com/pyccel/pygyro

Study Case - PyGyro vs SeLaLib

Still room for improvement, but all bottlenecks are translated functions.

MPI implementation in Python allowed for easy testing of parallelisation.

The strong scaling result for the prototype was very promising.



https://github.com/pyccel/pygyro