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)

SFSCON - 2023

Beginner friendly

Easy/fast to use

Libraries

Portable

Large community

Slow

Memory intensive

Limited recursion

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

```
$ python3 -m timeit -s 'from ackermann import ackermann' 'ackermann(3,6)'
50 loops, best of 5: 9.36 msec per loop
$ python3 -m timeit -s 'from ackermann import ackermann' 'ackermann(3,7)'
Traceback (most recent call last):
...
RecursionError: maximum recursion depth exceeded in comparison
```

Pyccel was born out of frustration when going from prototype (Python) code to production (Fortran) scientific code.

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

Using the C-Python API, translations are exposed back to Python creating an ***accelerator***.

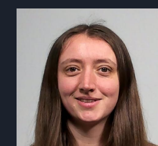
Pyccel is therefore also a static compiler for Python 3, using Fortran or C as backend language.

https://github.com/pyccel/pyccel

Original author :
Ahmed Ratnani
Director of the UM6P Vanguard Center

Original co-author :
Said Hadjout
PhD student at the TUM Munich

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



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

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

Pyccel : The best of both worlds?

Solving a Linear Convection Equation with Finite Differences

```python
import numpy as np

def linearconv_1d(nx: int, dt: float, nt: int):
    c  = 1.0
    dx = 2 / (nx-1)
    x  = np.linspace(0, 2, nx)
    u  = np.ones(nx)
    u[int(0.5/dx) : int(1/dx + 1)] = 2

    cp = c * dt / dx
    un = np.zeros(nx)

    for _ in range(nt):
        un[:] = u[:]
        for i in range(1, nx):
            u[i] = un[i] - cp * (un[i] - un[i-1])

    return x, u

if __name__ == '__main__':
    x, u = linearconv_1d(2001, 0.0003, 3000)
```

```
$ python3.10 -m timeit -s 'from linearconv_1d_mod import linearconv_1d'
'x, u = linearconv_1d(2001, 0.0003, 3000)'
1 loop, best of 5: 1.47 sec per loop
$ pyccel linearconv_1d_mod.py
$ python3.10 -m timeit -s 'from linearconv_1d_mod import linearconv_1d'
'x, u = linearconv_1d(2001, 0.0003, 3000)'
200 loops, best of 5: 1.33 msec per loop
```

x 1105 !

Maximum memory (as measured by massif):

python3.10 linearconv_mod.py : 6.4 MB

./linearconv_mod : 61 KB

x 105 !

Pyccel : The best of both worlds?

Solving a Linear Convection Equation with Finite Differences

Fortran translation

```fortran
module linearconv_1d_mod

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

  contains
  !........................................
  subroutine linearconv_1d(nx, dt, nt, x, u)
    implicit none
    real(f64), allocatable, intent(out) :: x(:)
    real(f64), allocatable, intent(out) :: u(:)
    integer(i64), value :: nx
    real(f64), value :: dt
    integer(i64), value :: nt
    real(f64) :: c
    real(f64) :: dx
    real(f64) :: cp
    real(f64), allocatable :: un(:)
    integer(i64) :: Dummy_0000
    integer(i64) :: i
    integer(i64) :: linspace_index
    c = 1._f64
    dx = 2._f64 / Real((nx - 1_i64), f64)
    allocate(x(0:nx - 1_i64))
    x = [((0_i64 + linspace_index*Real((2_i64 - 0_i64), f64) / Real((nx &
          - 1_i64), f64)), linspace_index = 0_i64,nx - 1_i64)]
    x(nx - 1_i64) = 2._f64
    allocate(u(0:nx - 1_i64))
    u = 1._f64
    u(Int(0._f64 / dx, i64):Int(1_i64 / dx + 1_i64, i64) - 1_i64) = &
          2_i64
    cp = c * dt / dx
    allocate(un(0:nx - 1_i64))
    un = 0._f64
    do Dummy_0000 = 0_i64, nt - 1_i64, 1_i64
      un(:) = u(:)
      do i = 0_i64, nx - 1_i64, 1_i64
        u(i) = un(i) - cp * (un(i) - un(i - 1_i64))
      end do
    end do
    if (allocated(un)) then
      deallocate(un)
    end if
    return

  end subroutine linearconv_1d
  !........................................
end module linearconv_1d_mod
```

First steps with Pyccel

Pyccel is available on Pypi:

```
pip install pyccel
```

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

```
pyccel linearconv_mod.py
```

Multiple files are generated:

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

```
.
├── linearconv_mod.cpython-310-x86_64-linux-gnu.so
├── linearconv_mod.py
└── __pyccel__/
    ├── linearconv_mod.mod
    ├── linearconv_mod.f90
    ├── linearconv_mod.f90.lock
    ├── linearconv_mod.o
    ├── linearconv_mod.o.lock
    ├── linearconv_mod_wrapper.c
    ├── linearconv_mod_wrapper.c.lock
    ├── linearconv_mod_wrapper.o
    ├── linearconv_mod_wrapper.o.lock
    ├── bind_c_linearconv_mod.f90
    ├── bind_c_linearconv_mod.f90.lock
    ├── bind_c_linearconv_mod.mod
    ├── bind_c_linearconv_mod.o
    ├── bind_c_linearconv_mod.o.lock
    ├── cwrapper/
    ├── cwrapper.lock
    ├── cwrapper_ndarrays/
    └── ndarrays/
```

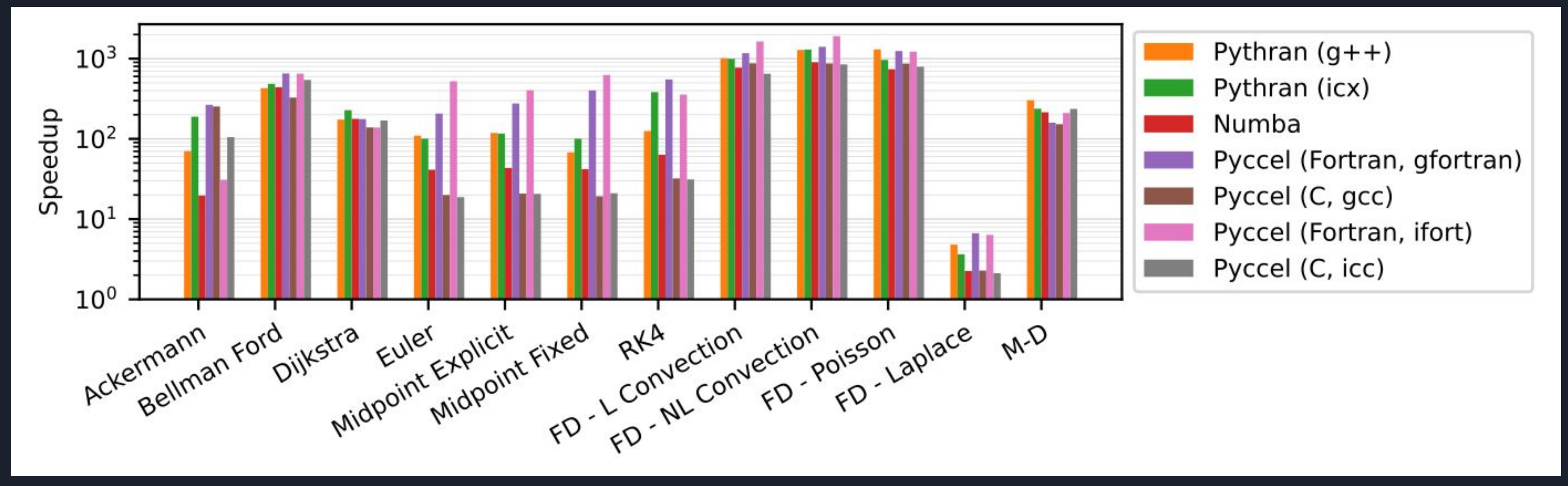
Is Pyccel the right tool for me?

Two main questions:

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

Tool	Code must be translated	Easy to install without sudo privileges	Type annotations needed	Compatible bottlenecks need isolating	Handles multiple folders	Interface with pure Python code	Interface with low level 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	+++

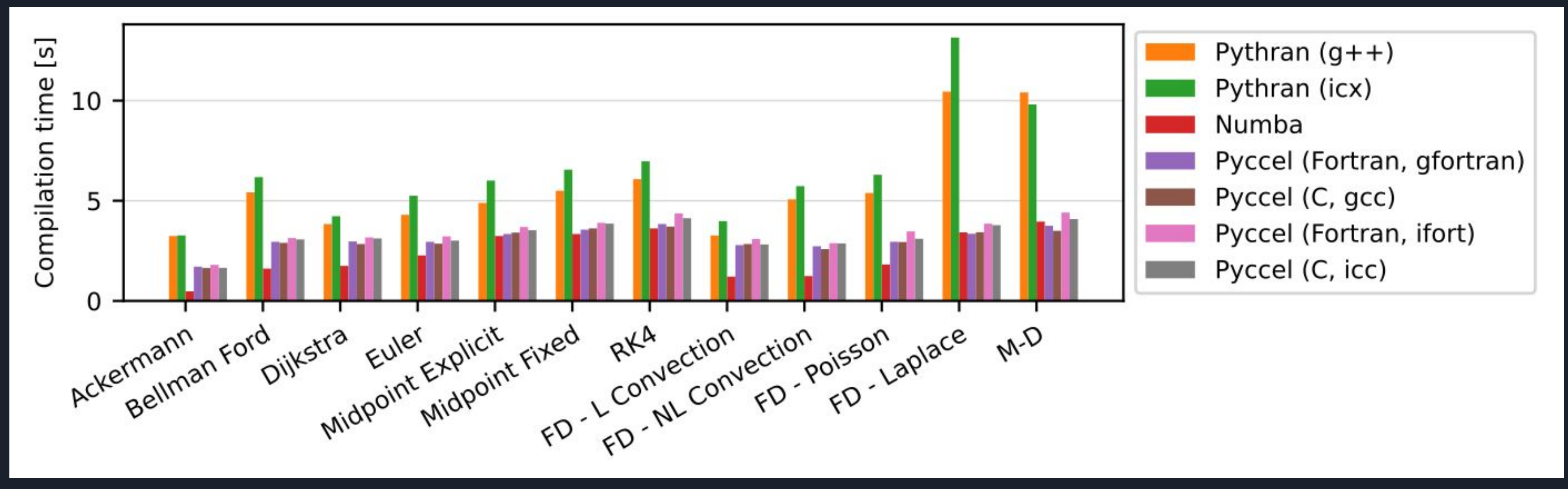
How much can I gain vs NumPy?



Using most recent versions of the codes on Pypi as of 7th November with Python 3.11

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

How long does that take?



Using most recent versions of the codes on Pypi as of 7th November with Python 3.11

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

Conclusions

https://www.github.com/pyccel/pyccel

- Python + Pyccel = speedy development **and** speedy execution (up to x1000)
- Simple interface via type annotations
- Human-readable code generated in Fortran or C
- Has support for parallelisation paradigms (OpenMP, cupy available in a fork)
- Open to Open Source contributors
- Well-maintained docs
  https://github.com/pyccel/pyccel/tree/devel/docs
- Discord server for community interactions
  https://discord.gg/2Q6hwjfFVb
- Used by other open-source projects:
  - Struphy Plasma Physics PDEs
  - Psydac: a Python IGA library for large-scale simulations
- Published in the Journal of Open Source Software
  https://joss.theoj.org/papers/10.21105/joss.04991

Also presenting at SFSCON!

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[:]', c: 'T[:,:]'):
      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 weeks)
- 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:

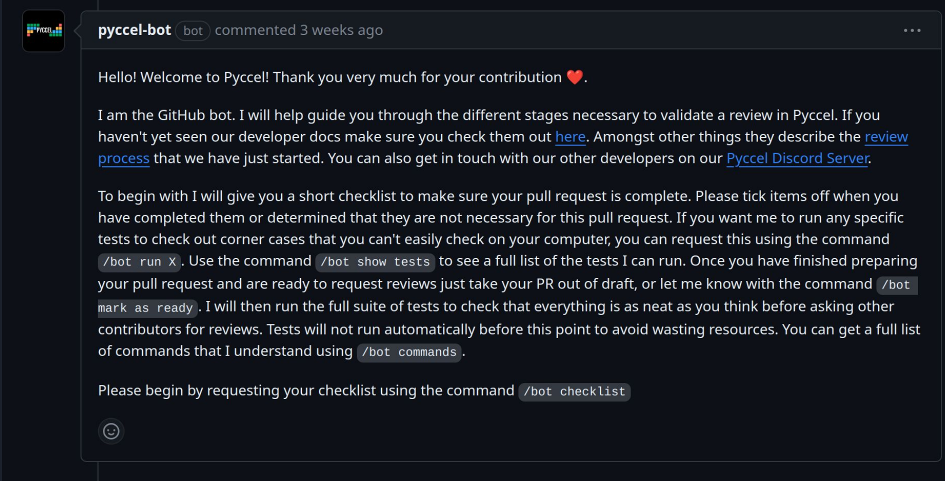
- cmath
- math
- NumPy

Minimal support:

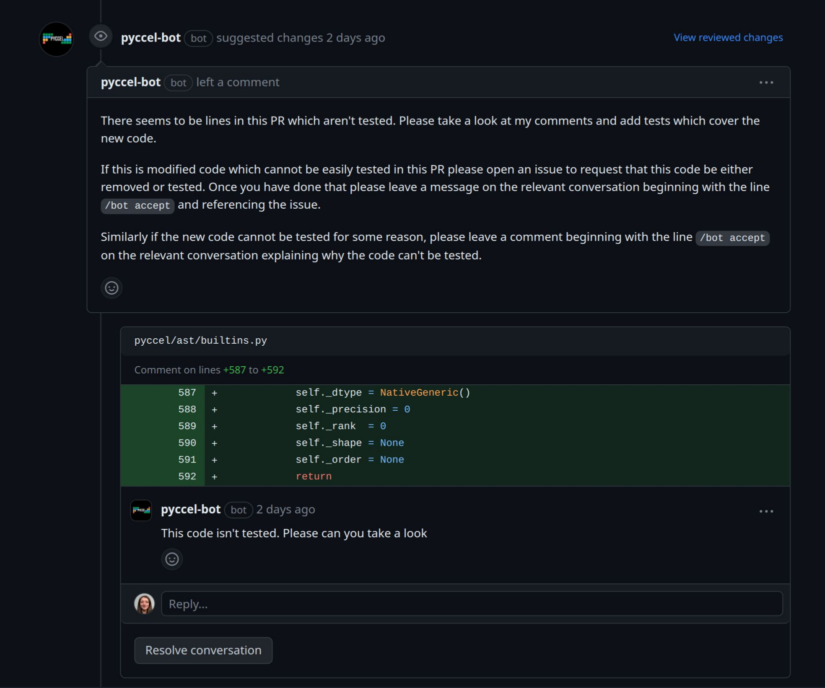
- *product* from itertools
- *exit* from sys

Bots and tests are used to automate repetitive tasks

Tests are only run on request to reduce ecological impact



Managing Open Source Contributions



OpenMP Support

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

OpenMP functions can be imported and accessed via Pyccel

```
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