Pyccel: Automating Translation of Python Prototypes to C/Fortran Production Codes

Emily Bourne[1], Mohamed Jalal Maaouni[2], Yaman Güçlü[3]

[1]Scitas, EPFL, Switzerland; [2]UM6P, Morocco; [3]NMPP division, MPI for Plasma Physics, Germany

What is Pyccel[1] ?

New numerical methods are often developed in Python due to its simplicity, and the availability of a variety of packages that simplify testing. However, using these methods in high-performance computing (HPC) applications typically requires converting them to a low-level language like Fortran or C; a process that can become a significant bottleneck.

Pyccel automates this conversion, generating human-readable Fortran or C code directly from Python. It also creates a C Python extension module, so the compiled code remains callable from Python, allowing the same testing suite to be used for both versions.

The scientific libraries Struphy[2] and Psydac[3] leverage this to allow their Python libraries to run at usable speeds.

[1] Bourne, E., Güçlü, Y., Hadjout, S., & Ratnani, A. (2023). Pyccel: a Python-to-X transpiler for scientific high-performance computing. Journal of Open Source Software, 8(83), 4991.
[2] Possanner, S., Holderied, F., Li, Y., Na, B. K., Bell, D., Hadjout, S., & Güçlü, Y. (2023, August). High-Order Structure-Preserving Algorithms for Plasma Hybrid Models. In International Conference on Geometric Science of Information (pp. 263-271). Cham: Springer Nature Switzerland.
[3] Güçlü, Y., Hadjout, S., & Ratnani, A. (2022). PSYDAC: a high-performance IGA library in Python. In 8th European Congress on Computational Methods in Applied Sciences and Engineering.

What's new in v2 ?

Pyccel Version 1 supported the objects most commonly used in scientific simulations: NumPy arrays and scalars. However occasionally more complex structures are required.

Pyccel Version 2 expands the existing support to include:

- Support for homogeneous containers thanks to STC[1] (for C) and gFTL[2] (for Fortran)
  - List support `list[T]`
  - Dictionary support `dict[K, V]`
  - Set support `set[T]`
- Class support
  - magic methods (e.g. `__add__`)
  - read-only attributes with `@property`
- Improved type specification closer to Python's native syntax (`typing` module)

```
T = TypeVar['T', float, complex]
def myfunc(x: Final[T], i: int) -> T:
    ...
```

[1] STC: github.com/stclib/stc
[2] gFTL: github.com/goddard-/gftl

An example: Implementing splines

Original Python code

```
from typing import Final
import numpy as np
from pyccel.decorators import inline

class Spline:
    def __init__(self, degree : int, knots : 'float[:]', coeffs : 'float[:]'):
        self._degree = degree
        self._knots = knots
        self._coeffs = coeffs

    @inline
    @property
    def degree(self):
        return self._degree

    def _basis_funcs(self, x: 'float', span: 'int', values: 'float[:]'):
        """ Compute non-zero basis functions at x following Algorithm A2.2
        from the NURBS book [1]. """
        left = np.empty(self.degree)
        right = np.empty(self.degree)

        values[0] = 1.0

        for j in range(0, self.degree):
            left[j] = x - self._knots[span-j]
            right[j] = self._knots[span+1+j] - x
            saved = 0.0

            for r in range(0, j+1):
                temp = values[r] / (right[r] + left[j-r])
                values[r] = saved + right[r] * temp
                saved = left[j-r] * temp

            values[j+1] = saved

    def eval(self, x : 'Final[float[:]]', y : 'float[:]'):
        """ Evaluate spline at non-zero basis elements: sum_i N_i(x) * c_i.
        """
        ...

    def _find_span(self, x: float) -> int:
        ...
```

Auto-genenerated translation (trimmed for readability)

```
struct Spline {
    int64_t private_degree;
    array_double_1d private_knots;
    array_double_1d private_coeffs;
    bool is_freed;
};
/*_____________________________________________________________*/
/* Compute non-zero basis functions at x following Algorithm A2.2*/
/*        from the NURBS book [1].                             */
/*_____________________________________________________________*/
void Spline__private_basis_funcs(struct Spline* self, double x, int64_t span,
                                 array_double_1d values)
{
    array_double_1d left = {0};
    array_double_1d right = {0};
    int64_t j;
    double saved;
    int64_t r;
    double temp;
    double* left_ptr;
    double* right_ptr;
    left_ptr = malloc(sizeof(double) * (self->private_degree));
    left = (array_double_1d)cspan_md_layout(c_ROWMAJOR, left_ptr, self->private_degree);
    right_ptr = malloc(sizeof(double) * (self->private_degree));
    right = (array_double_1d)cspan_md_layout(c_ROWMAJOR, right_ptr, self->private_degree);
    (*cspan_at(&values, INT64_C(0))) = 1.0;
    for (j = INT64_C(0); j < self->private_degree; j += INT64_C(1))
    {
        (*cspan_at(&left, j)) = x - (*cspan_at(&self->private_knots, span - j));
        (*cspan_at(&right, j)) = (*cspan_at(&self->private_knots, span + INT64_C(1) + j)) - x;
        saved = 0.0;
        for (r = INT64_C(0); r < j + INT64_C(1); r += INT64_C(1))
        {
            temp = (*cspan_at(&values, r)) / ((*cspan_at(&right, r)) +
                   (*cspan_at(&left, j - r)));
            (*cspan_at(&values, r)) = saved + (*cspan_at(&right, r)) * temp;
            saved = (*cspan_at(&left, j - r)) * temp;
        }
        (*cspan_at(&values, j + INT64_C(1))) = saved;
    }
    free(left.data);
    left.data = NULL;
    free(right.data);
    right.data = NULL;
}
```

C Translation

See the full example:

For large arrays the pure Python code is quite slow:

```
$ python3 -m pyperf timeit -s 'from splines import Spline; import numpy as np;
s = Spline(5, knots = np.linspace(0,1, 1000), coeffs = np.ones(1000)); x =
np.random.rand(100000); y = np.empty(100000)' 's.eval(x, y)'

Mean +- std dev: 1.81 sec +- 0.40 sec
```

But with minimal effort the problem is fixed:

```
$ pyccel splines.py     # This command takes 2.5s
$ ls
__pyccel__   splines.cpython-310-x86_64-linux-gnu.so  splines.py
$ python3 -m pyperf timeit -s 'from splines import Spline; import numpy as np;
s = Spline(5, knots = np.linspace(0,1, 1000), coeffs = np.ones(1000)); x =
np.random.rand(100000); y = np.empty(100000)' 's.eval(x, y)'

Mean +- std dev: 12.6 ms +- 0.0 ms
```

The code can be found in the __pyccel__/ directory meaning it could also be used in an existing low-level project.

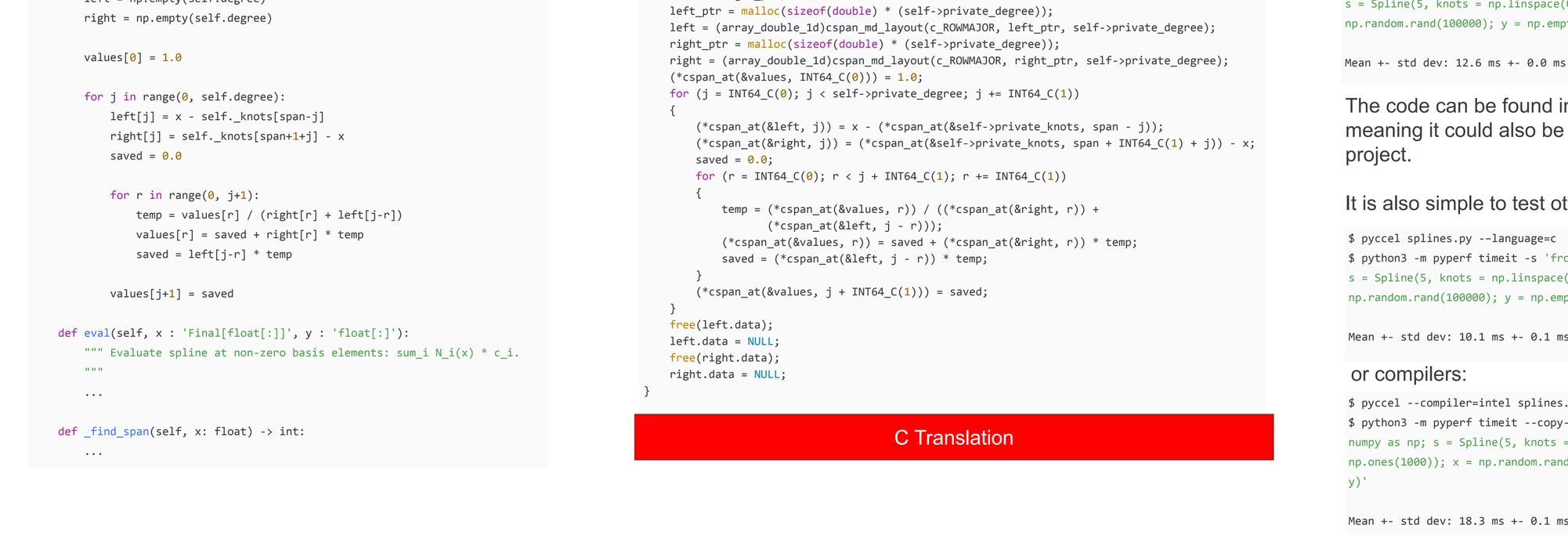
It is also simple to test other languages:

```
$ pyccel splines.py --language=c
$ python3 -m pyperf timeit -s 'from splines import Spline; import numpy as np;
s = Spline(5, knots = np.linspace(0,1, 1000), coeffs = np.ones(1000)); x =
np.random.rand(100000); y = np.empty(100000)' 's.eval(x, y)'

Mean +- std dev: 10.1 ms +- 0.1 ms
```

or compilers:

```
$ pyccel --compiler=intel splines.py
$ python3 -m pyperf timeit --copy-env -s 'from splines import Spline; import
numpy as np; s = Spline(5, knots = np.linspace(0,1, 1000), coeffs =
np.ones(1000)); x = np.random.rand(100000); y = np.empty(100000)' 's.eval(x,
y)'

Mean +- std dev: 18.3 ms +- 0.1 ms
```

Pyccel vs other accelerators : Benchmarks



github.com/pyccel/pyccel

github.com/pyccel/pyccel-benchmarks