Université Paris-XIII

Master's Thesis

Pyccel & Parallel Kronecker solvers using Automated Symbolic Analysis

Author:
Said Hadjout

Supervisor:
Dr. Ahmed Ratnani

September 28, 2018

Contents

acknowledgement

I had the wonderful opportunity to do an internship in Max-Planck institute of plasma physics in Munich I would like to thank my Supervisor Dr. Ahmed Ratnani for believing in me all along this period and for being always there when I needed help.
I would like also to thank Mr. Olivier Lafitte and Mr. Emmanuel Audusse for giving me this opportunity and guiding me when I was lost and always encouraging me and my colleagues to always go further to reach our true potential.
I would finally like to say my parents who are the reason of me being who I'm right now to whom I would never succeed without them thank you for everything.

1 Introduction

In this work we present two project's which are based on three articles that will be published. the first paper will present Pyccel a Fortran static compiler for scientific High-Performance Computing. Pyccel was presented in the 11th European Conference on Python in Science Euro-Scipy 2018, it was held on August 31, it will also be presented in PyCon.DE 2018 & PyData Conference on October 24 in Karlsruhe Germany. The second paper will present a Parallel Kronecker solver using symbolic finite elements it is written and used in Python it relies on Pyccel for accelerating its algorithms.
Pyccel is a static compiler for Python, Using Fortran as a back-end language, it can also be viewed as a Translator as it translates the Python code to Fortran, The goal behind developing Pyccel is to allow the user to go from a prototype example written in subset of Python Language, toward a production code in Fortran, without the need of rewriting the whole code, this will give the user the advantage of writing an easy and understandable code in Python that takes much less time in writing compared to a code written in Fortran or C/C++ and it will have the performance of code written in Fortran after the translation,the difference between Pyccel and the accelerating tools is that it provides an understandable translated code is that can be modified an used with other projects in the targeted language, Pyccel can also be used in an embedded mode, inside Python or IPython.
The main job in my master's thesis was to introduce different functionalities into Pyccel such as the oriented object programming, Functional Programming, support some decorators and add the support of different libraries such as MPI4PY, BLAS, LAPACK, and FFTPACK. We also made different benchmarks that are performed with some existing tools such as Pythran, Numba, Hope or Cython. In addition, we present other examples that cannot be (fully) accelerated using these tools, while Pyccel allows for a full conversion.
In future works, we will extend the functional paradigm with task-based parallelism and cache-efficient algorithms for some specific patterns, we also intend to support Parallel programming using GPU's and also support different Languages other than Fortran like C/C++.

For the second paper, we present a Python library that recognizes the Kronecker structure starting from a weak formulation.
The aim was to automatically generate a parallel code for the matrix-vector product function in Python, which is then converted into Fortran using Pyccel. In addition, solving the associated linear systems can be done using different strategies according to the system in order to get an optimal fast solver.
The idea behind it is the same as Pyccel which consist of writing a symbolic code using the Sympy library and generate after that a parallel python code for the matrix assembly, the dot product and different direct solvers or preconditioners that depends on the provided problem, Then the generated python code is translated into Fortran using Pyccel.
Our aim is to have a direct link between the weak formulations defined in the continuous space and their discrete versions. All the existing tools in the market such as Fenics, FreeFem, ...etc, do not infer the properties of the associated linear system to the discrete level. Our goal is not only to provide the assembly procedure, which is generated automatically can be executed in parallel, but also to construct an appropriate linear solver or a Preconditioner associated to a continuous weak formulation.

2 Symbolic Computation

Symbolic computation is an area of computer science that aims to automate a wide range of the computation involved in mathematical problem solving, where we manipulate mathematical expressions and other mathematical objects.
the uses for symbolic computation touches almost all different fields of mathematics and science. its main use is calculations that are made by hand which can be done more efficiently and accurately by symbolic calculations like calculating derivatives, integrals, series and many other areas of mathematics.

2.1 Sympy

Sympy is a Python library for symbolic computations. It aims to become a full-featured computer algebra system (CAS) while keeping the code as simple as possible in order to be comprehensible and easily extensible. Sympy is written entirely in Python.

computing the power series of a function: differentiation:

```
<<< from sympy import Symbol, cos
<<< x = Symbol('x')
<<< e = 1/cos(x)
<<< print e.series(x, 0, 10)
 1 + x**2/2 + 5*x**4/24 + 61*x**6/720 + 277*x**8/8064 + O(x**10)
```

```
<<< from sympy import Symbol,diff, sin,exp
<<< x = Symbol('x')
<<< f = sin(x)*exp(x)
<<< diff(f, x)
  exp(x)*sin(x) + exp(x)*cos(x)
```

2.1.1 Advanced Expression Manipulation

here we are going to take a look of how the mathematical expression are represented in Sympy which will help us in the future to manipulate expressions in Pyccel

Listing 1: Using Numpy for multiple precision

```
>>> from sympy import *
>>> x, y, z = symbols('x y z')
>>> expr = 2**x + x*y
>>> srepr(expr)
"Add(Pow(Integer(2), Symbol('x')), Mul(Symbol('x'), Symbol('y')))"
```

this concept can help us later in calculating the Complexity of Program which may help us in the future in making generating automatic code rather than doing it manually as of right now.

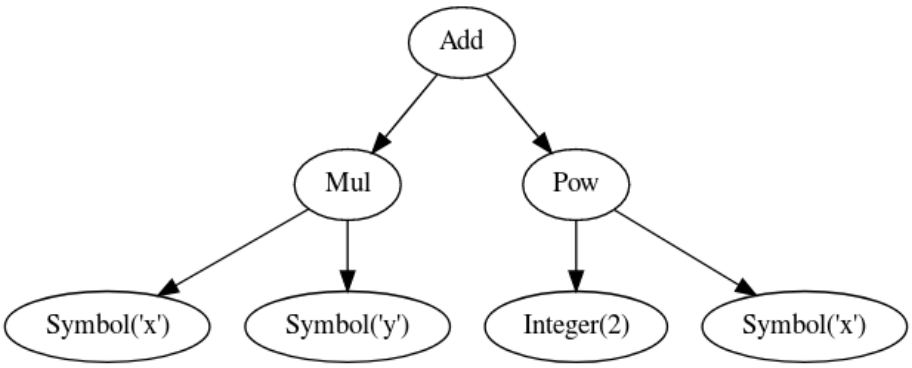

Figure 1: representation of an expression in Sympy

2.1.2 Code Generation in Sympy

In addition to symbolic calculations Sympy introduced a cool concept which is code generation . Sympy provides us with functionalities to generate directly compilable code from Sympy expressions to many other programing languages such as C, C++, Fortran77, Fortran90, Julia, Rust and Octave/Matlab.
The Code Generation in Sympy is very limited to simple symbolic expression, and what Pyccel does is take this concept and generalize it to be able to generate much more complex symbolic expression so that we can use it later to translate whole Python program .
to illustrate this concept we give this example of code generation in Sympy to C language :

```
>>> from sympy.utilities.codegen import codegen
>>> from sympy.abc import x, y, z
>>> [(c_name, c_code), (h_name, c_header)] = codegen(
...     ("f", x+y*z), "C89", "test", header=False, empty=False)
>>> print(c_name)
test.c
>>> print(c_code)
#include "test.h"
#include <math.h>
double f(double x, double y, double z) {
   double f_result;
   f_result = x + y*z;
   return f_result;
}
```

Pyccel, a Fortran static compiler for scientific High-Performance Computing

A. Ratnani[34], S. Hadjout[35]

[3] *Max-Planck Institut für Plasmaphysik, Garching, Germany*

[4] *Technische Universität München, Garching, Germany*

[5] *Université Paris-XIII , Paris, France*

Abstract

We present a static compiler for Python, using Fortran as a backend language. The aim of Pyccel is to allow the user to go from a prototype example, written in Python, toward a production code in Fortran, without the need of rewritting the whole code. Pyccel can also be used in an embedded mode, inside Python or IPython. Different benchmarks are performed with some existing tools such as Pythran, Numba or Cython. In addition, we present other examples (using the Kronecker algebra) than can not (fully) accelerated using these tools, while Pyccel allows for a full conversion.

Keywords: Python, HPC, Fortran, DSL, MPI, OpenMP, OpenAcc

1 Introduction

The Python language has gained a significant popularity as a language for scientific computing and data science, mainly because it is easy to learn and provides many scientific libraries, including parallel ones. However, Python interpreter/compiler does not perform any optimization and is not meant to write fast codes. Therefor, different approachs have been proposed to accelerate computation-intensive parts of Python codes. Cython [2] is among the standard tools to accelerate Python codes allowing the user to call the Python C API by introducing a static typing approach. However, the low-level code depends on the Python runtime to execute and no backward compatibilty is ensured. More recently Pythran [7] allows to convert the dynamic Python code into a static C++ code while providing types as comments. The Hope [1] library provides a JIT compiler to convert the Python code to C++, where the arguments types are only known at the execution time. Numba[5] follows the same idea of bringing JIT compiling to Python while generating machine code based on LLVM, which can run on either CPU or GPU. Both Numba and Hope rely heavily on the use of simple decorators to instruct the Python package to compile a given function. They also use the available type information at runtime to generate byte code. Another completly different approach is given by PyPy[3], where a new a Python interpreter was written in a restricted subset of the Python language itself (using an internal language called RPython). The aim of PyPy is to provide speed and efficiency at runtime using a JIT compiler.
All the modern solutions to accelerate Python codes are based on C/C++ and so far nothing has been based on the Fortran language. A drawback of using Fortran as a backend language is mainly the lack of meta-programming. While modern scientific codes tend to use C++, mainly because of the meta-programming paradigms, in many applications the main programming language remains Fortran. It is the case for the Plasma Physics community where production softwares rely on years of legacy codes, for which scientists developped specific expertises. Therefor, collaborations with mathematicians is constrained by a huge time investment in code development which is not appropriate for prototyping

new algorithms or numerical schemes. A mathematician dream would be to go instantly from a prototype, written in high level language like Python, to a production code that can be shared with physicists, in Fortran/C++ for example.
In order to simplify the process of going from a prototype to a production code, it is important that the *compiler* allows the use legacy codes or some Python scientific libraries such as numpy, scipy (including blas, lapack and fft), mpi4py, etc ... Moreover, this should be presented in a user-friendly way while ensuring that the new compiler is not difficult to maintain; it is then necessary to rely on some well established python libraries.
On the other hand, Fortran is a computer language for scientific programming that is tailored for efficient run-time execution on a wide variety of processors. Even if the 2003 and 2008 standards added major improvements like *OOP*, *Coarrays*, *Submodules*, *do concurrent*, etc ... they are not covered by all available compilers. Moreover, the Fortran developer still suffers from the lack of *meta-programming* compared to C++. Therefor, it becomes increasingly difficult for applied mathematicians and computational physicists to write applications at the *state of art* while implementing complicated algorithms or (new) numerical schemes.
For this purpose, Pyccel was designed to be used in two cases: (1) accelerate Python code by converting it to Fortran and calling *f2py*, (2) generate portable HPC Fortran codes from a DSL using the Python syntax. In order to achieve the second point, we developped an internal DSL for types and macros. The later is used to map sentences based on *mpi4py*, *blas* or *lapack* (from scipy) onto the appropriate calls in Fortran. Two other parsers for *OpenMP* and *OpenACC* were added too, allowing explicit parallelism through the use of pragmas.
In this work, we present a new Fortran static compiler for Python. The aim of *Pyccel* is to allow computational scientists to get a low level Fortran code from a (valid) Python code. It is completely built on the top of Sympy. The input Python code is then viewed as a symbolic expression for which we extended the associated Fortran printer and provided simple ways to use legacy codes and third party Python libraries such as numpy, scipy and mpi4py. Extending the covered libraries is not difficult; following the idea of Haskell's [9] design, Pyccel is also written in Pyccel.
This paper is organized as the following: in Section 2, we describe the internal design of Pyccel, introduce its standard library, the internal DSL and the parallel computing capabilities. In Section 3, we present different examples of benchmarks and compare how Pyccel performs with respect to a selection of similar existing tools.

2 Pyccel internal design

In this section, we shall explain the Pyccel workflow and give more details on how it handles legacy code, OpenMP/OpenAcc, MPI and some third party Python libraries.

2.1 Pyccel workflow

Starting from a Python code, the *full syntax tree* (FST) is constructed using the *redbaron* [12] python library. In opposition to the classical *Asbtract Syntax Tree* (AST), the FST is a tree representation of the abstract syntactic structure of source code providing comments as nodes. Therefor, it is possible to extend the Python language with additional syntax and semantic statements allowing for static typing, *OpenMP/OpenACC* pragmas and *types/macros* instructions.

Syntax Analysis

Redbaron allows to check the Python syntax and raises an error for a non-valid Python code. Other errors due to Pyccel limitations can also be raised, for example when using `try/except`.

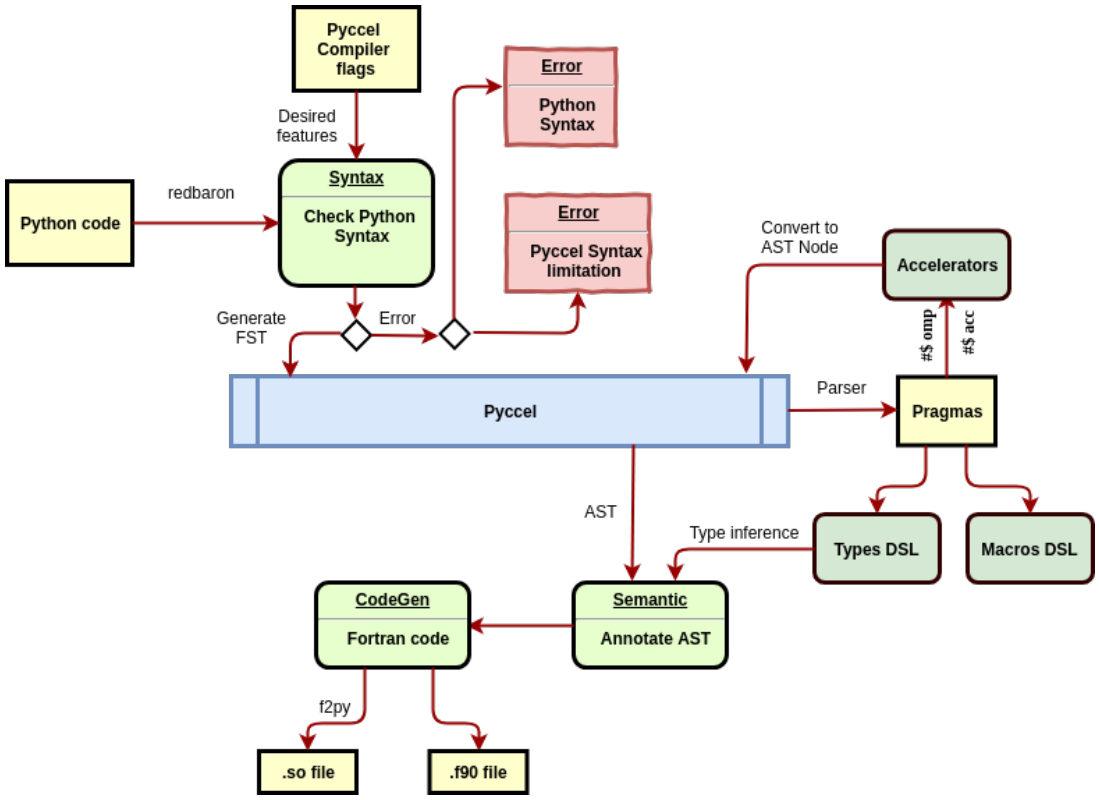

Figure 1: Pyccel workflow

During this stage, comments that start with '`#$`' are parsed using the internal DSL of Pyccel. We distinguish between 3 cases:

- **Accelerators**: OpenMP and OpenAcc statements can be accessed using pragmas of the form '`#$ omp`' and '`#$ acc`'
- **Types** Functions and class methods need associated headers to define their types.
- **Macros** This last feature allows Pyccel to map third libraries calls (like scipy.linalg.blas) to their low level representations (Pyccel internal dsl).

At the end of this stage, Redbaron and *Pragmas* nodes are converted to Sympy nodes representing the new AST.

Annotation and semantic analysis

The second step is to annotate the AST using the available information about types. Declared variables are implicitly typed, while functions types are now available thanks to their headers. For functions that return results, their types are infered automaticaly (but can still be provided). Notice that a function can return multiple variables, but must have a fixed number of results. Another important remark is that the header (+ results types) provides a **type** for a function
Source code 1 shows a typical definition of a Python function. Notice that the type of the result `y` is not given since it is computed using the *Inference Type process*.

```
# header function decr(int)
def decr(x):
    y = x - 1
    return y
```

Source Code 1: Example of function declaration

A function can have different interfaces (in the Fortran spirit). The source code 2 provides such an example where, the `decr` function is now available for `int` and simple and double precision `real` numbers.

```
# this function will generate 3 functions
#$ header function decr(int|double|float)
def decr(x):
    y = x - 1
    return y
```

Source Code 2: Example of function declaration

Remark 2.1 *A function without argument and result, does not need a header.*

In some cases, Pyccel needs to create symbolic variables that will not appear in the Fortran code. A simple example is the following assignement `rx = range(0, 10)`. Such a statement does not have an equivalent in Fortran. However, Pyccel allows to manipulate it and use the variable `rx` in a loop.
At the end of this stage, the annotated AST is ready for code-generation.

Code generation

Since Pyccel AST is based on Sympy, writing the Fortran code generator can be done easily by extending the Sympy *fcode* printer. At this point, and thanks to our type inference system, the generated Fortran will compile without errors, except when using OpenMP/OpenAcc, for which we rely on the Fortran compiler to check some semantic errors. Notice that the same approach can be extended in order to generate code in other languages thanks to the different printers available in Sympy.

2.2 Pyccel Internal DSL

Types

In table 1, we give the correspondance between native numeric types in Python and Fortran. Floating numbers are considered to be in double precision. Multiple precision will be described later in the subsection 2.8.

`int`	`int(kind=4)`
`float`	`real(kind=8)`
`complex`	`complex(kind=16)`
`bool`	`logical`
`char`	`character`
`str`	`character(:)`

Table 1: Correspondance between native numeric types in Python and Fortran

Header files

Header files can be used to link against an existing library. These files must have the extension `.pyh` and when parsed, they allow to expose functions or variables that are present in a given library. In source code 3, we show how Pyccel provides the functions `sdaxpy` and `ddaxpy` from *blas*.

```
#$ header metavar module_version='3.8'
#$ header metavar ignore_at_import=True
#$ header metavar libraries='${BLAS_LIBRARIES}'

#$ header function saxpy(int, float, float [:], int, float [:], int)
#$ header function daxpy(int, double, double [:], int, double [:], int)
```

Source Code 3: Example of a header file, for BLAS

In code source 4, we show how one can import and call `daxpy`, in a DSL mode; this means that such a code will not work in Python. The notion of *macro* will be used later to ensure a backward compatibility with Python.

```
from pyccel.stdlib.internal.blas import daxpy
from numpy import zeros

n = 5
sa = 1.0

incx = 1
sx = zeros(n)

incy = 1
sy = zeros(n)

sx[0] = 1.0
sx[1] = 3.0
sx[3] = 5.0

sy[0] = 2.0
sy[1] = 4.0
sy[3] = 6.0

daxpy(n, sa, sx, incx, sy, incy)
```

Source Code 4: daxpy example in a DSL mode usage

Meta-Variables

Pyccel provides the notion of *meta-variables* in orther to have more control when linking to legacy code. In Tab. 2, we list the available meta-variables and their meaning.

Macros

In order to make Pyccel understand `scipy.linalg.blas` sentences, we introduced the notion of **macros**. In source code 5, we give the implemetation of daxpy as a macro, allowing Pyccel to understand a subset of `scipy.linalg.blas` statements.

meta-variable	meaning
`module_name`	module name for the generated code (example `openmp` → `omp_lib`)
`module_version`	module version
`ignore_at_import`	do not print the `use` statement, if `True`
`libraries`	Environement variable for libraries to link to

Table 2: Pyccel meta-variables and their meaning

```
1 from pyccel.stdlib.internal.blas import daxpy as _daxpy
2
3 #$ header macro (y), daxpy(x, y, alpha=1, n=x.shape, incx=1, incy=1) := _daxpy(n, alpha, x, ↩
      incx, y, incy)
4 b = daxpy(a, b, alpha)
```

Source Code 5: Implementation of daxpy using **macro** DSL to make `scipy.linalg.blas` available in Pyccel.

2.3 Python/Pyccel statements

Pyccel covers most of Python statements, when using imperative programming. Simple statements are comprised within a single logical line and are given in Tab. 3. In Tab. 4, we give the available compound statements covered by Pyccel. These statements contain a group of other statements (simple or compoun). `try` statement is not covered due since they are not suited for HPC applications, although they can be implemented in Fortran. For the moment, `async` statements are not covered, but they may be useful in the future.

`assert`	limited
`assignment`	✓
`augmented assignment`	✓
`pass`	✓
`del`	✓
`return`	✓
`yield`	✗
`raise`	✗
`break`	✓
`continue`	✓
`import`	✓
`global`	✗
`nonlocal`	✗

Table 3: Available simple statements in Pyccel

`if`	✓
`for`	✓
`while`	✓
`try`	✗
`funcdef`	✓
`classdef`	✓
`async_with_stmt`	✗
`async_for_stmt`	✗
`async_funcdef`	✗

Table 4: Available compound statements in Pyccel

Oriented Object Programing

OOP is partially available in Pyccel. In source code 6, we define a `class` and its associated methods. Polymorphism is not yet available, because of the current implementation of type inference algorithm inheritence is also not available yet. Notice the use of the keyword `method` when declaring the function type, in order to *bound* the method `translate` to the class `Point`. Fortran 2003 is then used to generate the associated code.

```
#$ header class Point(public)
#$ header method __init__(Point, [double])
#$ header method __del__(Point)
#$ header method translate(Point, [double])

class Point(object):
    def __init__(self, x):
        self.x = x

    def __del__(self):
        pass

    def translate(self, a):
        self.x = self.x + a

x = [1.,1.,1.]
p = Point (x)

a = [0.,0.,0.]
a[0] = 3
p.translate(a)
print(p.x)

b = p.x[0]
b = p.x[0] + 1
b = 2 * p.x[0] + 1
b = 2 * ( p.x[0] + 1 )
print(b)
c= p.x

p.x[1] = 2.0

del p
```

Source Code 6: Class definition and usage.

Decorators

Pyccel gives you the power to use some decorators in order to have some special functionalities,we here offer four decorators `types`, `sympy`, `python` ,`vectorize`,`inline` and `property` for the OOP.

- types

the `types` decorator does the same work as a type header instead of writing a header you can use the decorator types in order to have the same mechanisme.

```
@types(int)
def g1(x):
    y = x+1
    return y
```

Source Code 7: Examples of *types* decorator.

- vectorize

the `vectorize` decorator allow us to extend a function that acts on a scaler argument to act on a vector

```
#$ header function f(int)
@vectorize(z)
def f(z):
    x= 5+z
    return x
```

Source Code 8: Examples of *vectorize* decorator.

- sympy

the `sympy` decorator allow us to extend the notion of a lambda expressions where you can write complex symbolic code using Sympy objects and as lambda expression these functions will not be generated only after the call of Pyccel built-in function lambdify in addition to declaring its type.

```
@sympy
def f2_sympy(x):
    #This function must return a sympy expression
    #that depends on the arguments of the function
    from sympy import diff
    u = x**2 + 2*x
    u_dx = diff(u, x)
    return u_dx
```

Source Code 9: Examples of *sympy* decorator.

- python

due to the limitations of Pyccel the python decorator gives you the freedom to write whatever Python code you want and will be executed dynamically during the code generation process .

```
@python
def f(x):
    x = [None]*5
    x = 5
    x = 5.
    return x
```

Source Code 10: Examples of *python* decorator.

- inline

the `inline` decorator gives us the option of inlining a function which will help in increasing the execution time.

```
#$ header function f(int)
@inline
def f(t):
    x = 5*t
    return x

m = f(5)
```

Source Code 11: Examples of *inline* decorator.

- property

we also support the `property` decorator used in methods for classes in the Oriented Object programing case.

```
#$ header class Shape(public)
#$ header method __init__(Shape, double, double)
#$ header method area(Shape) results(double)
#$ header method perimeter(Shape) results(double)
#$ header method describe(Shape,str)
#$ header method authorName(Shape,str)
#$ header method scaleSize(Shape, double)

class Shape:

    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.description = "This shape has not been described yet"
        self.author = "Nobody has claimed to make this shape yet"

    @property
    def area(self):
        y = self.x * self.y
        return y

    @property
    def perimeter(self):
        x = 2 * self.x + 2 * self.y
        return x

    def describe(self, text):
        self.description = text

    def authorName(self, text):
        self.author = text

    def scaleSize(self, scale):
        self.x = self.x * scale
        self.y = self.y * scale

rectangle = Shape(100., 45.)
#finding the area of your rectangle:
print(rectangle.area)
```

Source Code 12: Examples of *property* decorator.

2.4 Functional programming

Functional programming is partially covered by Pyccel. It allows for code manipulation, in opposition to the source-to-source approach of Pyccel when dealing with imperative programming. One of the main goals of adding this capability, is to provide different interpretations and then allow for *task based parallelism* in the future. *lambda expressions* are then very useful, since they should have no *side-effect* which is an important requirement to have automatic parallelisation.

Lambda expression

lambda expressions are small anonymous functions. They do not containt any assignement. One big advantage of *lambda* expression is that they allow for currying functions with multiple arguments. The later capability is still very limited in Pyccel and future work will be done to enhance it. In source code 13, we show different *lambda* expressions; Notice that the functions `f1`, `f2` and `g1` are treated as symbolic functions and they can not be generated in Fortran. While the function `m1` is exported after calling the Pyccel built-in function `lambdify` in addition to declaring its type.

```
f1 = lambda x: x**2 + 1
f2 = lambda x,y: x**2 + y**2 + 1
g1 = lambda x: f1(x)**2 + 1

#$ header m1(double)
m1 = lambdify(g1)
```

Source Code 13: Examples of *lambda* expressions.

List Comprehensions

List Comprehensions gives us an easy way to create lists ,we only support list of integers, real, complex numbers and tuples or lists we don't support list Comprehensions inside list Comprehensions we also don't support if conditions but we allow the use of the ternary conditional operator as show in the example below.

```
x = [i*j for i in range(1000) for j in range(0,i,2) for k in range(0,3)]

y = [5.]*50

z = [i*j*k1 for i in range(200) for j in range(0,i,2) for k1 in y]

s = [(x1, y1, z1)  for x1 in range(1,30) for y1 in range(x1,30) for z1 in range(y1,1000)]

t = [i if i>5 else 5 for i in range(1000)]
```

Source Code 14: Examples *List Comprehensions*.

Iterators

Thanks to the use of *magic methods* and OOP, it is possible to define classes that behave like iterators. In Python, this is simply done by implementing the methods `__iter__` and `__next__`. However, usage of iterators is still very experimental in Pyccel and additional work should be done, especially since one may use the `yield` statement, which is not supported yet in Pyccel.

2.5 Error and Warning messages

A considerable work have been done to ensure that Pyccel works like traditional compilers. Errors and warnings are returned by Pyccel depending on their *severity*. In addition, since Redbaron provides a *bounding-box* notion, lines/columns are printed for some errors/warnings.

2.6 Use of legacy code

One of the main advantages of Pyccel compared to the existing accerators tools, is the capability of using legacy code and importing them in different ways. Following the same idea that has been shown for BLAS (source code 3), one can link to any existing library in C/Fortran. Notice that this is done at the low level and does not rely on Python execution runtime.
In addition, one may implement Pyccel code (using the DSL mode) in order to have a better interface. This approach has been used for FFTPACK.

2.7 Pyccel standard library

Pyccel standard library consists of a small subset of utilities from the Python standard library in addition to `blas`, `lapack`, `fftpack`, `mpi`, `openmp` and `openacc` that are accessed through the DSL mode. Other libraries may be added in the future.

Built-in types, functions and constants

Built-in types were already introduced in Tab. 1. Python functions are not fully covered and only those marked in Tab. 5 are treated.

`abs`	✓	`delattr`	✗	`hash`	✗	`memoryview`	✗	`set`	✗
`all`	✗	`dict`	✗	`help`	✗	`min`	✓	`setattr`	✗
`any`	✗	`dir`	✗	`hex`	✗	`next`	✗	`slice`	?
`ascii`	✗	`divmod`	✗	`id`	✗	`object`	✗	`sorted`	✗
`bin`	✗	`enumerate`	✓	`input`	✓	`oct`	✗	`staticmethod`	✗
`bool`	✓	`eval`	✗	`int`	✓	`open`	✓	`str`	✓
`breakpoint`	✗	`exec`	✗	`isinstance`	✗	`ord`	✗	`sum`	✓
`bytearray`	✗	`filter`	✗	`issubclass`	✗	`pow`	✓	`super`	✓
`bytes`	✗	`float`	✓	`iter`	✗	`print`	✓	`tuple`	✓
`callable`	✗	`format`	✗	`len`	✓	`property`	✓	`type`	?
`chr`	✓	`frozenset`	✗	`list`	✓	`range`	✓	`vars`	✗
`classmethod`	✗	`getattr`	✗	`locals`	✗	`repr`	✓	`zip`	✓
`compile`	✗	`globals`	✗	`map`	✓	`reversed`	✓	`__import__`	✗
`complex`	✓	`hasattr`	✗	`max`	✓	`round`	✓		

Table 5: Available Python functions in Pyccel

Built-in Mathematical functions and constants

Python provides different mathematical functions through its standard library. These functions can be imported from the `math` package. In tables 6, 7, 8, 9, 10 and 11 we show the available real functions. Tab. 12 shows the available math constants.

ceil	✓	isclose	✗
copysign	✗	isfinite	✗
fabs	✗	isinf	✗
factorial	✗	isnan	✗
floor	✓	ldexp	✗
fmod	✓	modf	✓
frexp	✗	remainder	✗
fsum	✗	trunc	✗
gcd	✗		

Table 6: Available Number-theoretic and representation functions in Pyccel

exp	✓
expm1	✗
log	✓
log1p	✗
log2	✓
log10	✓
pow	✓
sqrt	✓

Table 7: Available Power and logarithmic functions in Pyccel

acos	✓
asin	✓
atan	✓
atan2	✓
cos	✓
hypot	✓
sin	✓
tan	✓

Table 8: Available trigonometric functions in Pyccel

acosh	✓
asinh	✓
atanh	✓
cosh	✓
sinh	✓
tannh	✓

Table 9: Available hyperbolic functions in Pyccel

degrees	✓?
radians	✓?

Table 10: Available angular conversion functions in Pyccel

erf	✓
erfc	✓
gamma	✓
lgamma	✓

Table 11: Available special functions in Pyccel

pi	✓
e	✓
tau	✗
inf	✗
nan	✗

Table 12: Available math constants in Pyccel

itertools package

`itertools` was originaly inspired by ideas from APL, Haskell and SML. Functional programming relies heavily on the use of iterators and exposing it will improve the capabilities of Pyccel. For the moment, `product` is the only available function. Source code 17 shows an example of using `product` function from `itertools` package.

Pyccel Internal libraries

Many HPC applications rely on the use of well established and optimized libraries like BLAS and LAPACK. Moreover, distribued computing is generaly based on MPI. In order to take into account these kinds of dependencies, Pyccel was designed in such a way it makes it easy to use these libraries in an internal dsl mode. In addition, enriching them is straithforward using types and macros dsl. Since Python provides good bindings for BLAS, LAPACK, MPI and FFTPACK (FFTW is not yet available in Pyccel) that are easier to use than the low-level implementations, we decided to make Pyccel understands and map high-level calls to the low-level ones, through the notion of *macros*. Tab. 13 lists the available libraries through the internal dsl mode in Pyccel. Other libraries will be added in the future, such as the *Thrust* [8] library.
In source code 18, we show how to call MPI in the dsl mode. Notice that such a code will not run using Python, in which case, one needs to use the *mpi4py* package.

`BLAS`	✓
`LAPACK`	✓
`MPI`	✓
`OpenMP`	✓
`OpenAcc`	✓
`FFTPACK`	✓
`FFTW`	✗

Table 13: Available internal libraries in Pyccel

2.8 Third party libraries

Thanks to the concept of *macros*, although it is limited, one can use third party libraries and still convert the Python code to Fortran, using Pyccel. In the sequel, we describe the available features in this context. These features will be extended in the future depending on the applications using Pyccel.

numpy

Numpy [11] is actually the standard package when dealing with N-dimensional arrays. It provides the object `ndarray` and many useful and advanced functions. `ndarray` object is available in Pyccel as an abstract node (an extension of the `Basic` class from *Sympy*).
Moreover, Pyccel covers Numpy multiple precision. In source code 16, we show how to create/declare variables with different precisions using numpy. Multiple precision for numeric types is also available for functions. In source code 15, we show a function that can operate on arrays with different precisions and types. In addition to these types, some functions from Numpy and covered by Pyccel are described in Tab. 15.

int	int(kind=4)
int32	int(kind=4)
int64	int(kind=8)
float	real(kind=8)
float32	real(kind=4)
float64	real(kind=8)
complex	complex(kind=16)
complex64	complex(kind=8)
complex128	complex(kind=16)

Table 14: Correspondance between numeric types in Numpy and Fortran

zeros	✓	sqrt	✓
ones	✓	asin	✓
array	✓	acos	✓
empty	✓	asec	✓
zeros_like	✓	atan	✓
ones_like	✓	acot	✓
rand	✓	acsc	✓
sum	✓	log	✓
shape	✓		

Table 15: Available numpy functions in Pyccel

```
1 #$ header function decr_(int*4 | int*8 | int*4 [:] | int*8 [:])
2 def decr(y):
3     y = y-1
```

Source Code 15: Function definition with different precisions and types

scipy

Scipy library [10] provides different mathematical algorithms and user-friendly functions built on the Numpy package. Tables 16 and 17 show the available constants and functions from scipy. Notice that due to the use of *macros*, BLAS and LAPACK calls must be used *in-place*, without creating new variables; this is not a limitation for HPC applications, since the user should manage himself the memory usage.

pi	✓

Table 16: Available scipy constants in Pyccel

scipy.linalg.blas	✓
scipy.linalg.lapack	✓
scipy.fftpack	✓

Table 17: Available scipy functions/sub-packages in Pyccel

mpi4py

mpi4py [4], provides Python bindings of *Message Passing Interface (MPI)* [6] standard allowing Python codes to run in parallel on *multiple processors*.
In source code 21, we show a parallel implementation for matrix-matrix multiplication using *mpi4py*.mpi4py uses the mecanimse of macros which provide us with a one on one mapping between mpi4py calls and Fortran calls Which comes with certain limitations if the mpi4py call is much more complex ,like the irecv method which need a buffer in Fortran while it doesn't in mpi4py

3 Examples and Benchmarks

In this section, we compare Pyccel to other available tools, such as: Hope, PyPy, Cython, Numba and Pythran. Finally, we give a more complex example that is of interest in HPC: parallel dot and solve for a Kronecker product matrix. Since it is involving different librairies in Python, the comparison will be restricted to few tools. In order to address the portability of the generated code by Pyccel, we use both gcc and intel to compile the Fortran code. In this case, the examples were only compiled with the flag `-O2`.

3.1 Rosen der

This example tests how the compiler is accessing data in a 1d array, while computing an expression that involves local terms (direct neighbors) whithin a for loop. In Tab. 18, we give the cpu time and the resulting speedup for double precision. We notice that Pyccel (gcc) is two times faster than Pythran and three times faster when using intel.

Tool	Python	Hope	PyPy	Cython	Numba	Pythran	Pyccel-gcc	Pyccel-intel
Timing (μs)	229.85	811.67	70.00	2.06	4.73	2.07	**0.98**	**0.64**
Speedup	–	× 0.28	× 3.28	× 111.43	× 48.57	× 110.98	× **232.94**	× **353.94**

Table 18: Rosen-Der (double precision): CPU time and speedup with respect to Python

3.2 Black-Scholes

We consider the Black-Scholes formula for European options. The closed form is used an evaluated for 10^6 strikes. In Tab. 19, we give the cpu time and the resulting speedup for double precision. As we can see, Pyccel outperforms all existing tools, especially when using intel compiler.

Tool	Python	Hope	PyPy	Cython	Numba	Pythran	Pyccel-gcc	Pyccel-intel
Timing (μs)	180.44	9.27	1.55	309.67	3.0	1.1	**1.04**	**6.56 10^{-2}**
Speedup	–	× 19.46	× 115.72	× 0.58	× 60.06	× 163.8	× **172.35**	× **2748.71**

Table 19: Black-Scholes (double precision): CPU time and speedup with respect to Python

3.3 Laplace

This example shows the perfomance on accessing a 2d array to solve the Laplace equation using second order central Finite Differences (access to direct neighbors). In Tab. 20, we give the cpu time and the resulting speedup for double precision. We notice that Pythran is better than Pyccel-gcc. The reason is that Pythran is performing code optimizations. Unlike gcc, the intel version is more than twice time faster than Pythran, without setting any optimization flags!!

Tool	Python	Hope	PyPy	Cython	Numba	Pythran	Pyccel-gcc	Pyccel-intel
Timing (μs)	57.71	745.27	1.51	7.98	6.46 10^{-2}	6.28 10^{-2}	**8.02 10^{-2}**	**2.81 10^{-2}**
Speedup	–	× 0.07	× 37.98	× 7.22	× 892.02	× 918.56	× **719.32**	× **2048.65**

Table 20: Laplace (double precision): CPU time and speedup with respect to Python

3.4 Growcut

The GrowCut example is an image processing algorithm based on segmentation according to the similarity of the adjacent pixels. In addition to using a 3d array, inner loops bounds are not fixed and may change with respect to the outer loops. In Tab. 21, we give the cpu time and the resulting speedup for double precision. In this example, we notice that Pyccel is only 1.3 time faster than Pythran.

Tool	Python	Hope	PyPy	Cython	Numba	Pythran	Pyccel-gcc	Pyccel-intel
Timing (s)	54.39	0.8	1.0	$1.02\ 10^{-1}$	$4.67\ 10^{-1}$	$8.57\ 10^{-2}$	$\mathbf{6.27\ 10^{-2}}$	$\mathbf{6.54\ 10^{-2}}$
Speedup	–	× 63.75	× 54.28	× 532.37	× 116.45	× 634.32	× **866.49**	× **831.7**

Table 21: Growcut (double precision): CPU time and speedup with respect to Python

3.5 Kronecker product

This example implements the matrix-vector product for a matrix that is a Kronecker product of two (stencil) matrices. In Tab. 22, we give the cpu time and the resulting speedup for double precision. We notice that Pyccel (using intel compiler) is about 6 and 9 times faster than Cython and Pythran.

Tool	Python	Numba	Hope	Cython	Pythran	Pyccel-gcc	Pyccel-intel
Timing (μs)	624.3	80.52	9.17	$8.3\ 10^{-1}$	1.248	$\mathbf{9.89\ 10^{-1}}$	$\mathbf{1.38\ 10^{-1}}$
Speedup	–	× 7.75	× 68.08	× 751.89	× 500.3	× **631.1**	× **4506.45**

Table 22: Kronecker-product dot (double precision): CPU time and speedup with respect to Python for a matrix 100×100 and a padding of 10

3.6 Discussion

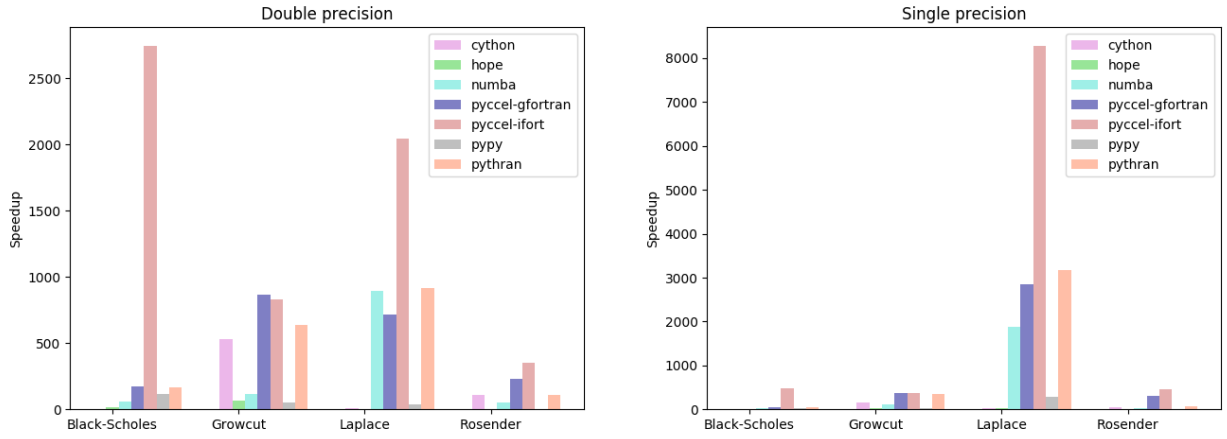

Figure 2: Speedup provided by different accelerator tools using double and single precision for the four considered examples.

3.7 Parallel matrix-matrix multiplication

In the sequel, we present 3 implementations of the matrix-matrix multiplication using MPI, OpenMP and OpenAcc. In Fig. 3, we plot the CPU time in seconds and the speedup on one single Node, for the OpenMP implementation, on the LRZ cluster.

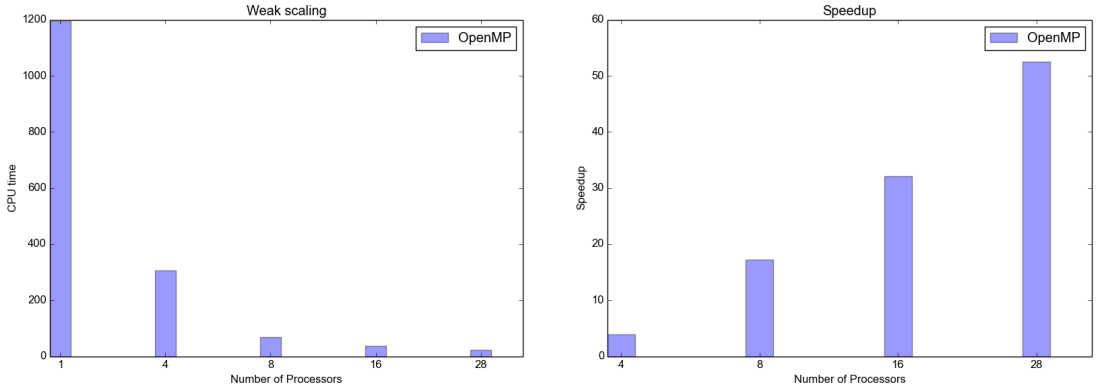

Figure 3: Matrix-multiplication results (using OpenMP) for (n,m,p) = (5000,7000,5000). (left)CPU time in seconds (right) speedup on one Node (28 cores)

4 Future work

- Functional programing
- Parallel computing
  - OpenAcc
  - Task based parallelism using OpenMP
- Code generation
- standard library: functools
- Third party libraries: fftw,

5 Conclusion

In this work, we presented a new static compiler for Python using Fortran as a backend language. We presented different benchmarks for which Pyccel outperforms most of the existing solutions mostly relying on c or c++. Although Pyccel does not perform any internal optimization on the generated code, the time execution is better thanks to the well known fact that fortran compilers produce optimized machine code. Pyccel is now used for different internal projects at the NMPP such as automatic code generation for finite elements methods using a formal languague for variational formulations. Other applications include a 4d parallel vlasov solver, GLT symbolic computation for BSplines and machine learning for pdes. In future work, we will extend the fonctional paradigm with task based parallelism and cache efficient algorithms for some specific patterns.

Appendix: Python codes

```
1  from numpy import int, int32, int64
2  from numpy import float, float32, float64
3  from numpy import complex, complex64, complex128
4
5  x1 = int(6)
6  x2 = int32(6)
7  x3 = int64(6)
8  y1 = float(6)
9  y2 = float32(6)
10 y3 = float64(6)
11 z1 = complex(6)
12 z2 = complex64(6)
13 z3 = complex128(6)
```

Source Code 16: Using Numpy for multiple precision

```
1  from itertools import product
2
3  x1 = [1, 2, 3]
4  y1 = [4, 5, 6]
5
6  for i2,j2 in product(x1, y1):
7      print i2,j2
```

Source Code 17: Example of using `product` function from `itertools` package.

```
from pyccel.stdlib.internal.mpi import mpi_init
from pyccel.stdlib.internal.mpi import mpi_finalize
from pyccel.stdlib.internal.mpi import mpi_comm_size
from pyccel.stdlib.internal.mpi import mpi_comm_rank
from pyccel.stdlib.internal.mpi import mpi_comm_world
from pyccel.stdlib.internal.mpi import mpi_status_size
from pyccel.stdlib.internal.mpi import mpi_allreduce
from pyccel.stdlib.internal.mpi import MPI_INTEGER
from pyccel.stdlib.internal.mpi import MPI_PROD

# we need to declare these variables somehow,
# since we are calling mpi subroutines
ierr = -1
size = -1
rank = -1

mpi_init(ierr)

comm = mpi_comm_world
mpi_comm_size(comm, size, ierr)
mpi_comm_rank(comm, rank, ierr)

if rank == 0:
    value = 1000
else:
    value = rank

product_value = 0
mpi_allreduce (value, product_value, 1, MPI_INTEGER, MPI_PROD, comm, ierr)

print('I, process ', rank,', have the global product value ', product_value)

mpi_finalize(ierr)
```

Source Code 18: Calling MPI in a DSL mode usage

```
def kron_dot_dense(A,B,X):
    n = A.shape[0] ; m = B.shape[1]
    X_tmp = np.zeros_like(X)
    for i in range(n):
        for j in range(m):
            for k in range(m):
                X_tmp[i,j] += X[i,k]*B[k,j]

    Y = np.zeros_like(X)
    for j in range(m):
        for i in range(n):
            for k in range(n):
                Y[i,j] += A[k,i]*X_tmp[k,j]
    return Y
```

Source Code 19: Kronecker dot product using dense matrices

```
def kron_dot_stencil(starts, ends, pads, X, X_tmp, Y, A, B):
    s1 = starts[0]
    s2 = starts[1]
    e1 = ends[0]
    e2 = ends[1]
    p1 = pads[0]
    p2 = pads[1]

    for j1 in range(s1-p1, e1+p1+1):
        for i2 in range(s2, e2+1):
            X_tmp[j1+p1-s1, i2-s2+p2] = sum(X[j1+p1-s1, i2-s2+k]*B[i2,k]
                                            for k in range(2*p2+1))

    for i1 in range(s1, e1+1):
        for i2 in range(s2, e2+1):
            Y[i1-s1+p1,i2-s2+p2] = sum(A[i1, k]*X_tmp[i1-s1+k, i2-s2+p2]
                                       for k in range(2*p1+1))

    return Y
```

Source Code 20: Kronecker dot product using stencil matrices

```
def dot(my_A,my_B,my_C)
import numpy import floor,sqrt,dot,empty_like,zeros_like
from mpi4py import MPI

my_N = 3000
my_M = 3000

NORTH = 0
SOUTH = 1
EAST = 2
WEST = 3

def pprint(string, comm=MPI.COMM_WORLD):
    if comm.rank == 0:
        print(string)

comm = MPI.COMM_WORLD

mpi_rows = int(floor(sqrt(comm.size)))
mpi_cols = comm.size // mpi_rows
if mpi_rows*mpi_cols > comm.size:
    mpi_cols -= 1
if mpi_rows*mpi_cols > comm.size:
    mpi_rows -= 1

pprint("Creating a %d x %d processor grid..." % (mpi_rows, mpi_cols) )

ccomm = comm.Create_cart( (mpi_rows, mpi_cols), periods=(True, True), reorder=True)

my_mpi_row, my_mpi_col = ccomm.Get_coords( ccomm.rank )
neigh = [0,0,0,0]

neigh[NORTH], neigh[SOUTH] = ccomm.Shift(0, 1)
neigh[EAST],  neigh[WEST]  = ccomm.Shift(1, 1)

# Create matrices
my_C = zeros_like(my_A)
tile_A_ = empty_like(my_A)
tile_B_ = empty_like(my_A)
req = [-1]*4

for r in range(mpi_rows):
    req[EAST]  = ccomm.Isend(tile_A , neigh[EAST])
    req[WEST]  = ccomm.Irecv(tile_A_, neigh[WEST])
    req[SOUTH] = ccomm.Isend(tile_B , neigh[SOUTH])
    req[NORTH] = ccomm.Irecv(tile_B_, neigh[NORTH])

    my_C += dot(tile_A, tile_B)

    req[0].Waitall(req)
comm.barrier()
```

Source Code 21: Matrix-matrix multiplication using MPI

```
from numpy import zeros

n = 5000
m = 7000
p = 5000

a = zeros(shape=(n,m))
b = zeros(shape=(m,p))
c = zeros(shape=(n,p))

#$ omp parallel
#$ omp do schedule(runtime)
for i in range(0, n):
    for j in range(0, m):
        a[i,j] = i-j
#$ omp end do nowait

#$ omp do schedule(runtime)
for i in range(0, m):
    for j in range(0, p):
        b[i,j] = i+j
#$ omp end do nowait

#$ omp do schedule(runtime)
for i in range(0, n):
    for j in range(0, p):
        for k in range(0, p):
            c[i,j] = c[i,j] + a[i,k]*b[k,j]
#$ omp end do
#$ omp end parallel
```

Source Code 22: Matrix-matrix multiplication using OpenMP

References

[1] J. Akeret, L. Gamper, A. Amara, and A. Refregier. Hope: A python just-in-time compiler for astrophysical computations. *Astronomy and Computing*, 10:1 – 8, 2015.

[2] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith. Cython: The best of both worlds. *Computing in Science Engineering*, 13(2):31–39, March 2011.

[3] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '09, pages 18–25, New York, NY, USA, 2009. ACM.

[4] Lisandro D. Dalcin, Rodrigo R. Paz, Pablo A. Kler, and Alejandro Cosimo. Parallel distributed computing using python. *Advances in Water Resources*, 34(9):1124 – 1139, 2011. New Computational Methods and Software Tools.

[5] T. Oliphant et al. Numba. http://numba.pydata.org.

[6] Message P Forum. Mpi: A message-passing interface standard. Technical report, Knoxville, TN, USA, 1994.

[7] Serge Guelton, Pierrick Brunet, Mehdi Amini, Adrien Merlini, Xavier Corbillon, and Alan Raynaud. Pythran: Enabling static optimization of scientific python programs. *Computational Science & Discovery*, 8(1):014001, 2015.

[8] J. Hoberock and N. Bell. https://thrust.github.io/.

[9] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[10] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[11] Travis E. Oliphant. *Guide to NumPy*. CreateSpace Independent Publishing Platform, USA, 2nd edition, 2015.

[12] PyCQA. redbaron. Technical report.

Fast and parallel Kronecker solvers using Automated Symbolic Isogeometric Analysis

A. Ratnani[a], S. Hadjout[b]

[a] *ahmed.ratnani@ipp.mpg.de*
[b] *saidaissa.hajdout@edu.univ-paris13.fr*

Abstract

Keywords: B-Splines, Finite Elements, Computer algebra system, HPC, Python, Symbolics, Kronecker algebra

1. Introduction

Discretizations of partial differential equations based on structured grids, allow to device fast solvers, under some specific assumptions. For linear equations with constant coefficients, the involved matrices, using Finite Difference or Finite Elements for example, can be described using the kronecker algebra; these matrices are in fact a sum of linear combination of Kronecker product of 2 or 3 matrices (for 2d and 3d problems respectively).
In this work, we present a Python library that recognizes the Kronecker structure starting from a weak formulation. The aim is to generate automatically a parallel code for the matrix-vector product function in Python, which is then converted into Fortran using Pyccel. In addition, solving the associated linear systems can be done using different strategies in order to get an optimal fast solver. In [1], such approach was used to provide a solver for Isogeometric Analysis discretization of the quasi-neutral equation in a gyro-kinetic model, using the FFT in the angular direction. In [2], the authors present a parallel implementation when using explicit time scheme, where in this case, they implemented a fasst solver for the mass matrix (which is a Kronecker product of 1d mass matrices). Other strategies such as the use of Sylvester equation was developed in [3]. Another application of such algorithms is the use of low rank approximation to device preconditioners as in [4].

Remark 1.1. *This work is still in progress, and only the matrix-vector product was implemented for the moment. Solving the associated linear system is still a work in progress.*

1.1. B-Splines

We start this section by recalling some basic properies about B-splines curves and surfaces.
For a basic introduction to the subject, we refer to the books [5].
A B-Splines family, $(N_i)_{1\leqslant i\leqslant n}$ of order k, can be generated using a non-decreasing sequence of *knots* $T = (t_i)_{1\leqslant i\leqslant n+k}$.

B-Splines series

The j-th B-Spline of order k is defined by the recurrence relation:

$$N_j^k = w_j^k N_j^{k-1} + (1 - w_{j+1}^k) N_{j+1}^{k-1} \tag{1.1}$$

where,

$$w_j^k(x) = \frac{x - t_j}{t_{j+k-1} - t_j} \qquad\qquad N_j^1(x) = \chi_{[t_j, t_{j+1}[}(x) \tag{1.2}$$

We note some important properties of a B-splines basis:

- B-splines are piecewise polynomial of degree $p = k - 1$,

- Compact support; the support of N_j^k is contained in $\left[t_j, t_{j+k}\right]$,
- If $x \in \ ]t_j, t_{j+1}[$, then only the *B-splines* $\{N_{j-k+1}^k, \cdots, N_j^k\}$ are non vanishing at x,
- Positivity: $\forall j \in \{1, \cdots, n\} \ \ N_j(x) > 0, \ \ \forall x \in]t_j, t_{j+k}[$,
- Partition of unity $\sum_{i=1}^n N_i^k(x) = 1, \forall x \in \mathbb{R}$,
- Local linear independence,
- If a knot t_i has a multiplicity m_i then the B-spline is $C^{(p-m_i)}$ at t_i.

2. Kronecker product

The kronecker product and its related properties prodive many examples of algorithms that combine intensive computations with the use of low level libraries. For more details on this subject, we refer the reader to [**?**] and the references therein. For self-consistency of the paper, we shall give a brief recall of the properties used to implement our examples.

Definition 2.1 (The **vec** operator)**.** *Let $A = (a_{ij}) \in \mathcal{M}_{n\times m}$, the* **vec** *operator is defined as,*

$$\mathbf{vec}A = \begin{pmatrix} A_{:,1} \\ \vdots \\ A_{:,m} \end{pmatrix} \in \mathbb{R}^{mn} \tag{2.1}$$

which is simply a vector composed by stacking all the columns of A. Where we denote $A_{:,j}$ the j^{th} column of A. We also define the inverse operator of **vec** *by,*

$$A = \mathbf{vec}^{-1}\mathbf{vec}A \tag{2.2}$$

Definition 2.2 (Kronecker product)**.** *Let $A = (a_{ij}) \in \mathcal{M}_{m\times n}$ and $B = (b_{ij}) \in \mathcal{M}_{r\times s}$ be two matrices. The Kronecker product of A and B, denoted by $A \otimes B \in \mathcal{M}_{mr\times ns}$, defines the following matrix:*

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \vdots & \vdots & & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{pmatrix} \tag{2.3}$$

Proposition 2.1.

$$(A \otimes B)\mathbf{vec}(X) = \mathbf{vec}(BXA^T) \tag{2.4}$$

Proposition 2.2.

$$(A \otimes B)^{-1} = A^{-1} \otimes B^{-1} \tag{2.5}$$

Using Eq. 2.5, we have $(A \otimes B)^{-1}\mathbf{vec}(Y) = A^{-1} \otimes B^{-1}\mathbf{vec}(Y)$, therefor

Proposition 2.3.

$$(A \otimes B)^{-1}\mathbf{vec}(Y) = \mathbf{vec}(B^{-1}YA^{-T}) \tag{2.6}$$

Algorithm 1 Kronecker product-dot (sequential) algorithm

1: **procedure** KRON_DOT_SEQ(A, B, x, y) ▷ $y = (A \otimes B)\, x$
2: $x_{tmp} \leftarrow$ MXM(x, B) ▷ matrix-matrix multiplication
3: $y \leftarrow$ MXM(A, x_{tmp})
4: **return** y
5: **end procedure**

2.1. Algorithms and implementation

In the sequel, we present the serial and parallel algorithm and give there python implementations. Using 2.4, the general form for the matrix-vector product is given in (algorithm 1) for the serial case. Its Python implementation is given in (Source Code 4), when both A and B are dense matrices. In many applications such as numerical discretizations of partial differential equations, both A and B are banded matrices. In this case, one may use the LAPACK xGBMV subroutines and its associated banded storage. Another way, is to use a Stencil format, as described in the (Source Code 5).
For parallel computing, we use MPI cart and its associated subcomminicators to exchange the 1D data. Such algorithms can not be implemented using the existing tools, while ensuring the backward compatibility with Python.

2.2. Results

We present here some comparison results between the timming of the assembly algrithme and the dot product of both the 2d Stencil Matrix and the Kronecker Sum of 1d matrices.

	1d $\otimes$ 1d	2d
Assembly (s)	**0.076s**	60.11s
Dot_Product (s)	**0.045s**	0.168s

Table 1: Comparaison between Kronecker Sum and 2d Format

3. SPL library

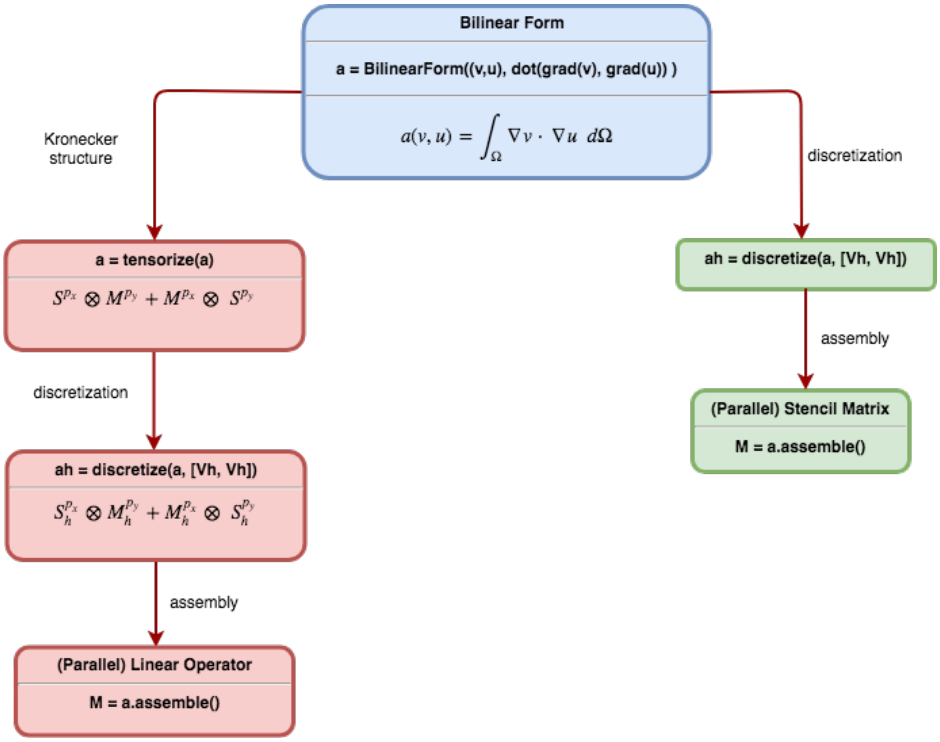

Figure 1: SPL workflow: starting from a bilinear form (right) one can discretize it giving the Spline spaces (left) perform the symbolic computation of the GLT expression or evaluate it.

3.1. Symbolic bilinear forms using sympde

Following the [6] approach, a new library for (tensor) finite elements has been implemented, based on Sympy. Although *sympde* will be presented in a future work, we introduce here the concept of `BilinearForm` that will be used throughout the paper. Compared to UFL, the ambiguity between *test* and *trial* functions is solved by providing the variables to a *lambda*-like expression.
Let M, A and S denote the mass, advection and stiffness matrices in 1d. The following code shows how one can define the Laplace and a curl-div operators using sympde: Computing their Kronecker structure leads to:

$$S_{p_x} \otimes M_{p_y} + M_{p_x} \otimes S_{p_y}$$

for Laplace operator, and

$$\begin{bmatrix} \Sigma_{11} & (-\alpha+\beta)\, A_{p_x} \otimes A_{p_y} \otimes M_{p_z} & (-\alpha+\beta)\, A_{p_x} \otimes A_{p_z} \otimes M_{p_y} \\ (-\alpha+\beta)\, A_{p_x} \otimes A_{p_y} \otimes M_{p_z} & \Sigma_{22} & (-\alpha+\beta)\, A_{p_y} \otimes A_{p_z} \otimes M_{p_x} \\ (-\alpha+\beta)\, A_{p_x} \otimes A_{p_z} \otimes M_{p_y} & (-\alpha+\beta)\, A_{p_y} \otimes A_{p_z} \otimes M_{p_x} & \Sigma_{33} \end{bmatrix}$$

for the Curl-Div operator, where we introduced the matrices

$$\begin{aligned} \Sigma_{11} &= \alpha M_{p_x} \otimes M_{p_y} \otimes S_{p_z} + \alpha M_{p_x} \otimes M_{p_z} \otimes S_{p_y} + \beta M_{p_y} \otimes M_{p_z} \otimes S_{p_x} \\ \Sigma_{22} &= \alpha M_{p_x} \otimes M_{p_y} \otimes S_{p_z} + \alpha M_{p_y} \otimes M_{p_z} \otimes S_{p_x} + \beta M_{p_x} \otimes M_{p_z} \otimes S_{p_y} \\ \Sigma_{33} &= \alpha M_{p_x} \otimes M_{p_z} \otimes S_{p_y} + \alpha M_{p_y} \otimes M_{p_z} \otimes S_{p_x} + \beta M_{p_x} \otimes M_{p_y} \otimes S_{p_z} \end{aligned}$$

In the following example, we consider the 2d anisotropic diffusion problem

```
from sympde.core import grad, dot
from sympde.core import FunctionSpace
from sympde.core import TestFunction
from sympde.core import BilinearForm
from sympde.core import Domain

domain = Domain(r'\Omega', dim=2)

V = FunctionSpace('V', domain=domain)
U = FunctionSpace('U', domain=domain)

v = TestFunction(V, name='v')
u = TestFunction(U, name='u')

a = BilinearForm((v,u), dot(grad(v), grad(u)))
```

Python Code 1: sympde model for the 2D Laplace operator

```
from sympde.core import dot, curl, div
from sympde.core import FunctionSpace
from sympde.core import VectorTestFunction
from sympde.core import BilinearForm
from sympde.core import Constant
from sympde.core import Domain

domain = Domain(r'\Omega', dim=3)

V = VectorFunctionSpace('V', domain=domain)

v = VectorTestFunction(V, name='v')
u = VectorTestFunction(V, name='u')

alpha = Constant('alpha', real=True)
beta  = Constant('beta', real=True)

expr = alpha * dot(curl(v), curl(u))
     + beta * div(v) * div(u)
a = BilinearForm((v,u), expr)
```

Python Code 2: sympde model for the 3D curl-div operator

$$a(v,u) = \int_\Omega \kappa_\parallel (\mathbf{b}\cdot\nabla v)(\mathbf{b}\cdot\nabla u) + \kappa_I \nabla v\cdot\nabla u d\Omega, \quad \forall u, v \in \mathcal{V}_h$$

where $\mathbf{b}$ denotes the unit vector of the magnetic field, Ω is our 2D computational domain and $\mathcal{V}_h \subset H^1(\Omega)$. In typical applications in plasma physics, we are interested in highly anisotropic configurations with $\frac{\kappa_\parallel}{\kappa_I} \simeq 10^6 \gg 1$. The associated Kronecker structure as computed by our symbolic library is

$$\kappa_\parallel b_x^2 M_{p_y} \otimes S_{p_x} + 2\kappa_\parallel b_x b_y A_{p_x} \otimes A_{p_y} + \kappa_\parallel b_y^2 M_{p_x} \otimes S_{p_y} + \kappa_\perp M_{p_x} \otimes S_{p_y} + \kappa_\perp M_{p_y} \otimes S_{p_x}$$

The abstract model using sympde is

4. Conclusion and future work

The aim of this work is to have a direct link between weak formulations defined in the continuous space and their discrete versions. Up to now, all the existing tools based on such approach, such as Fenics, Freefem, etc, do not infere the properties of the associated linear system to the discrete level. Our goal is not only to provide the assembly procedure, which is generated automaticaly and is parallel, but also to construct an appropriate linear solver or preconditioner associated to a continuous weak formulation. Up to now, we were able to compute the Kronecker structure for linear operators and with constant coefficients. In the future, we will extend this work so that it is possible to recognize separable variable coefficients, construct a low rank approximation and generate an appropriate linear solver using some predefined patterns such as the one based on FFT or Sylvester system.

```
from sympy import Tuple
from sympde.core import grad, dot
from sympde.core import FunctionSpace
from sympde.core import TestFunction
from sympde.core import BilinearForm
from sympde.core import Domain
from sympde.core import Constant

domain = Domain(r'\Omega', dim=2)

V = FunctionSpace('V', domain=domain)
U = FunctionSpace('U', domain=domain)

v = TestFunction(V, name='v')
u = TestFunction(U, name='u')

k_par = Constant('\kappa_{\parallel}', real=True)
k_per = Constant('\kappa_{\perp}', real=True)

bx = Constant('b_x', real=True)
by = Constant('b_y', real=True)
b = Tuple(bx, by)

expr = k_par * dot(b, grad(v)) * dot(b, grad(u)) +
       k_per * dot(grad(v), grad(u))
a = BilinearForm((v,u), expr)
```

Python Code 3: sympde model for the 2D anisotropic operator

Appendix

Python codes

```
def kron_dot_dense(A,B,X):
    n = A.shape[0] ; m = B.shape[1]
    X_tmp = np.zeros_like(X)
    for i in range(n):
        for j in range(m):
            for k in range(m):
                X_tmp[i,j] += X[i,k]*B[k,j]

    Y = np.zeros_like(X)
    for j in range(m):
        for i in range(n):
            for k in range(n):
                Y[i,j] += A[k,i]*X_tmp[k,j]
    return Y
```

Python Code 4: Kronecker dot product using dense matrices

```
def kron_dot_stencil(starts, ends, pads, X, X_tmp, Y, A, B):
    s1 = starts[0]
    s2 = starts[1]
    e1 = ends[0]
    e2 = ends[1]
    p1 = pads[0]
    p2 = pads[1]

    for j1 in range(s1-p1, e1+p1+1):
        for i2 in range(s2, e2+1):
             X_tmp[j1+p1-s1, i2-s2+p2] = sum(X[j1+p1-s1, i2-s2+k]*B[i2,k]
                                             for k in range(2*p2+1))

    for i1 in range(s1, e1+1):
        for i2 in range(s2, e2+1):
             Y[i1-s1+p1,i2-s2+p2] = sum(A[i1, k]*X_tmp[i1-s1+k, i2-s2+p2]
                                        for k in range(2*p1+1))

    return Y
```

Python Code 5: Kronecker dot product using stencil matrices

[1] N. Crouseilles, A. Ratnani, E. Sonnendrücker, An isogeometric analysis approach for the study of the gyrokinetic quasi-neutrality equation, Journal of Computational Physics 231 (2012) 373–393.
[2] M. Woźniak, M. Loś, M. Paszyński, L. Dalcin, V. Calo, Parallel fast isogeometric solvers for explicit dynamics, COMPUTING AND INFORMATICS 36 (2).
[3] G. Sangalli, M. Tani, Isogeometric preconditioners based on fast solvers for the sylvester equation, SIAM Journal on Scientific Computing 38 (6) (2016) A3644–A3671. doi:10.1137/16M1062788.
[4] A. Mantzaflaris, B. Jttler, B. N. Khoromskij, U. Langer, Low rank tensor methods in galerkin-based isogeometric analysis, Computer Methods in Applied Mechanics and Engineering 316 (2017) 1062 – 1085, special Issue on Isogeometric Analysis: Progress and Challenges. doi:https://doi.org/10.1016/j.cma.2016.11.013.
URL http://www.sciencedirect.com/science/article/pii/S0045782516315377
[5] W. T. L. Piegl, The NURBS Book, Springer-Verlag, Berlin, Heidelberg, 1995, second ed.
[6] M. S. Alnaes, A. Logg, K. B. Olgaard, M. E. Rognes, G. N. Wells, Unified form language: A domain-specific language for weak formulations of partial differential equations, ACM Trans. Math. Softw. 40 (2) (2014) 9:1–9:37.

5 Conclusion and future work

The aim of this work is to make the transition easy from a prototype code to a productive code.
By providing the Pyccel Compiler we were able to make that transition by translating the Python Code to an **Understandable** Fortran Code that competes well with the existing tools, the next step is to support other languages like C, C++, Lua, Julia, Rust and Octave/Matlab , so that we can have a Language-independent code which is a new concept that has many benefits that we can exploit.
Always in the same objective of simplifying the life of a programmer we introduced the Library Fast and parallel Kronecker solvers using Automated Symbolic Isogeometric Analysis to solve Partial differential equation in the same spirit of FreeFem, Fenics and many other tools but it is more user friendly and more efficient as it introduces many strategies to solve the associated Linear System by using the information provided by the Symbolic Isogeometric Analysis, this work is still in progress and only the matrix-vector product was implemented for the moment we will add Solving the exact associated linear system in the future.