# prometeo Documentation

***Release 0.0.5***

**Andrea Zanelli**

**May 19, 2022**

# Contents:

This is prometeo, an experimental modeling tool for embedded high-performance computing. prometeo provides a domain specific language (DSL) based on a subset of the Python language that allows one to conveniently write scientific computing programs in a high-level language (Python itself) that can be transpiled to high-performance self-contained C code easily deployable on embedded devices.

# CHAPTER 1

# features

1. **Python compatible syntax :** prometeo is a DSL embedded into the Python language. prometeo programs can be executed from the Python interpreter.
2. **efficient :** prometeo programs transpile to high-performance C code.
3. **statically typed :** prometeo uses Python's native type hints to strictly enforce static typing.
4. **deterministic memory usage :** a specific program structure is required and enforced through static analysis. In this way prometeo transpiled programs have a guaranteed maximum heap usage.
5. **fast memory menagement :** thanks to its static analysis, prometeo can avoid allocating and garbage-collecting memory, resulting in faster and safer execution.
6. **self-contained and embeddable :** unlike other similar tools and languages, prometeo targets specifically embedded applications and programs written in prometeo transpile to self-contained C code that does not require linking against the Python run-time library.

## 1.1 Installation

### 1.1.1 PyPI installation

prometeo can be installed through PyPI with *pip install prometeo-dsl*. Notice that, since prometeo makes extensive use of type hints to equip Python code with static typing information, the minimum Python version required is 3.6.

### 1.1.2 manual installation

If you want to install prometeo building the sources on your local machine you can proceed as follows:

- Run *git submodule update –init* to clone the submodules.
- Run *make install_shared* from *<prometeo_root>/prometeo/cpmt* to compile and install the shared library associated with the C backend. Notice that the default installation path is *<prometeo_root>/prometeo/cpmt/install*.

- You need Python 3.6. or later.
- Optional: to keep things clean you can setup a virtual environment with *virtualenv –python=<path_to_python3.6> <path_to_new_virtualenv>*.
- Run *pip install -e .* from *<prometeo_root>* to install the Python package.

Finally, you can run the examples in *<root>/examples* with *pmt <example_name>.py –cgen=<True/False>*, where the *–cgen* flag determines whether the code is executed by the Python interpreter or C code is generated compiled and run.

## 1.2 Python syntax

prometeo is an embedded domain specific language based on Python. Hence, its syntax is based on Python. Below you find details regarding the most common supported Python constructs that prometeo is able to transpile to C.

### 1.2.1 variable declaration

A variable can be declared as follows

```
<var_name> : <type> = <value>
```

where *<var_name>* must be a valid identifier *<type>* must be a valid prometeo built-in type or a user-defined type and *<value>* must be an valid expression of type *<type>*.

Example:

```
a : int = 1
```

Notice that, unlike in Python, type hints are strictly mandatory as they will instruct prometeo's parser regarding the type of the variables being defined.

### 1.2.2 *if* statement

An *if* statement takes the form

```
if <cond>:
    ...
```

### 1.2.3 *for* loop

A *for* loop takes the form

```
for i in range([<start>], <end>)
    ...
```

where the optional parameter *<start>* must be an expression of type *int* (default value 0) and defines the starting value of the loop's index and *<end>* must be an expression of type *<int>* which defines its final value.

### 1.2.4 function definition

Functions can be defined as follows

```
def <function_name> (<arg1> :  <arg_1_type>, ...) -> <ret_type> :

    ...

    return <ret_value>
```

### 1.2.5 class definition

prometeo supports basic classes of the following form

```
class <name>:
    def __init__(self, <arg1> : <arg_1_type>, ...) -> None:
        self.<attribute> : <type> = <value>
        ...

    def <method_name> (self,  <arg1> : <arg_1_type>, ...) -> <ret_type>:
        ...

        return <ret_value>
```

### 1.2.6 main function

For consistency all main functions need to be defined as follows

```
def main() -> int:

    ...

return 0
```

### 1.2.7 pure Python blocks

In order to be able to use the full potential of the Python language and its vast pool of libraries, it is possible to write *pure Python* blocks that are run only when prometeo code is executed directly from the Python intepreter (when –cgen is set to false). In particular, any line that is enclosed within *# pure >* and *# pure <* will be run only by the Python interpreter, but completely discarded by prometeo's parser.

```
# some prometeo code
A : pmat = pmat(n,n)
...

# pure >

# this is only run by the Python interpreter (--cgen=False)
# and will not be transpiled)

# some Python code
```

(continues on next page)

(continued from previous page)

```
import numpy as np

M = np.array([[1.0, 2.0],[0.0, 0.5]])
print(np.linalg.eigvals(M))
...

# pure <

# some more prometeo code
for i in range(n):
    for j in range(n):
        A[i, j] = 1.0
...
```

## 1.3 BLAS/LAPACK API

Below a description of prometeo's BLAS/LAPACK API can be found:

### 1.3.1 LEVEL 1 BLAS

### 1.3.2 LEVEL 2 BLAS

- General matrix-vector multiplication (GEMV)

$$z \leftarrow \beta \cdot y + \alpha \cdot \text{op}(A)x$$

```
pmt_gemv(A[.T], x, [y], z, [alpha=1.0], [beta=0.0])
```

- Solve linear system with (lower or upper) triangular matrix coefficient (TRSV)

$$\text{op}(A)\, x = b$$

```
pmt_trsv(A[.T], b, [lower=True])
```

- Matrix-vector multiplication with (lower or upper) triangular matrix coefficient (TRMV)

$$z \leftarrow \text{op}(A)\, x$$

```
pmt_trmv(A[.T], x, z, [lower=True])
```

### 1.3.3 LEVEL 3 BLAS

- General matrix-matrix multiplication (GEMM)

$$D \leftarrow \beta \cdot C + \alpha \cdot \text{op}(A)\, \text{op}(B)$$

```
pmt_gemm(A[.T], B[.T], [C], D, [alpha=1.0], [beta=0.0])
```

- Symmetric rank $k$ update (SYRK)

$$D \leftarrow \beta \cdot C + \alpha \cdot \text{op}(A)\,\text{op}(B)$$

with $C$ and $D$ lower triangular.

```
pmt_syrk(A[.T], B[.T], [C], D, [alpha=1.0], [beta=0.0])
```

- Triangular matrix-matrix multiplication (TRMM)

$$D \leftarrow \alpha \cdot B\,A^\top$$

with $B$ upper triangular or

$$D \leftarrow \alpha \cdot A\,B$$

with $A$ lower triangular.

```
pmt_trmm(A[.T], B, D, [alpha=1.0], [beta=0.0])
```

### 1.3.4 LAPACK

- Cholesky factorization (POTRF)

$$C = D\,D^\top$$

with $D$ lower triangular and $C$ symmetric and positive definite

```
pmt_potrf(C, D)
```

- LU factorization (GETRF)

$$C = L\,P\,U$$

```
pmt_getr(C, D)
```

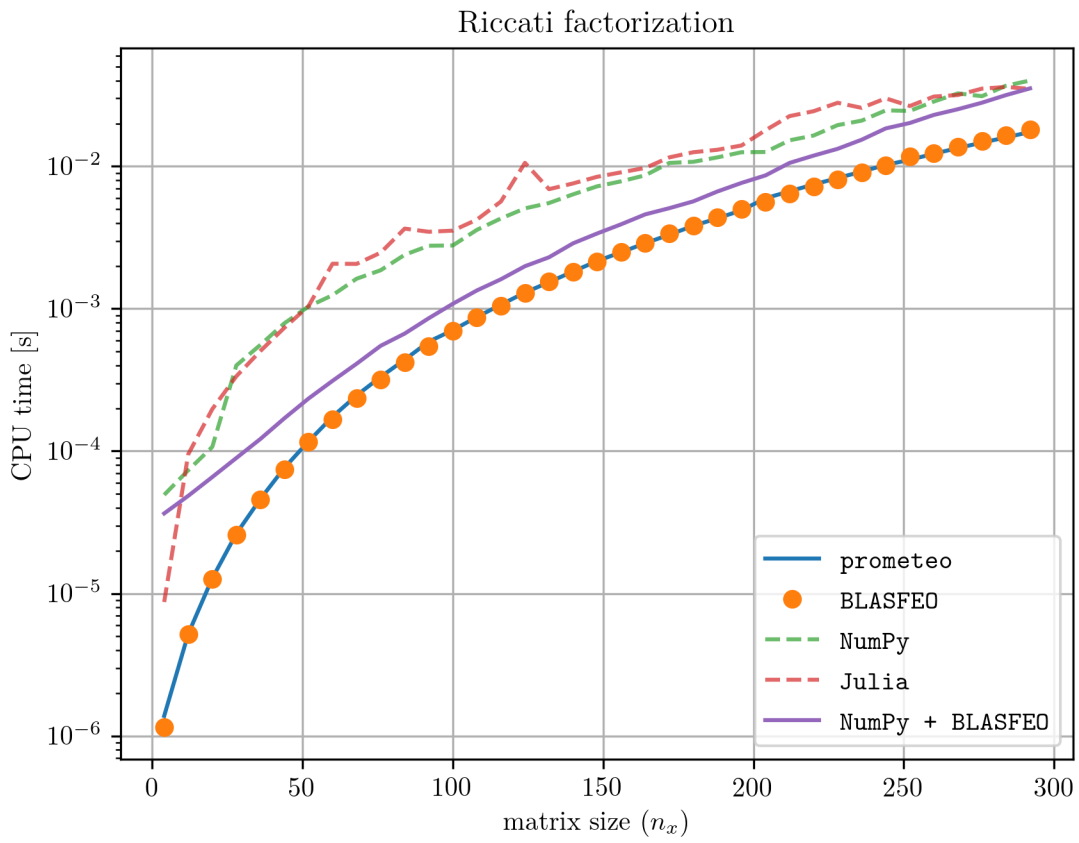
- QR factorization (GEQRF)

$$C = Q\,R$$

```
pmt_geqrf(C, D)
```

## 1.4 Performance

Since prometeo programs transpile to pure C code that calls the high performance linear algebra library BLASFEO (publication, code), execution time can be comparable to hand-written high-performance code. The figure below shows a comparison of the CPU time necessary to carry out a Riccati factorization using highly optimized hand-written C code with calls to BLASFEO and the ones obtained with prometeo transpiled code from this example). The computation times obtained with NumPy and Julia are added too for comparison - notice however that these last two implementations of the Riccati factorization are **not as easily embeddable** as the C code generated by prometeo and the hand-coded C implementation. All the benchmarks have been run on a Dell XPS-9360 equipped with an i7-7560U CPU running at 2.30 GHz (to avoid frequency fluctuations due to thermal throttling).


Riccati factorization
CPU time [s]
$10^{-2}$
$10^{-3}$
$10^{-4}$
$10^{-5}$
$10^{-6}$
0
50
100
150
200
250
300
matrix size ($n_x$)
prometeo
BLASFEO
NumPy
Julia
NumPy + BLASFEO