Journal of Open Aviation Science (2023), Vol.**1**
doi:10.59490/joas.2023.7071

SOFTWARE

impunity: Enforcing Physical Unit Consistency at Definition Time in Python

Antoine Chevrot [*] and Xavier Olive

ONERA DTIS, Université de Toulouse, Toulouse, France
*Corresponding author: antoine.chevrot@onera.fr

(Received: 23 June 2023; Revised: 18 July 2023; Accepted: 19 July 2023; Published: 20 July 2023)
(Editor: Martin Strohmeier; Reviewers: Enrico Spinielli, Junzi Sun)

Abstract
We introduce `impunity`, a Python library that enables static analysis of code annotations to ensure the consistency of physical units. It provides a framework for developers to annotate their Python code with physical units and automatically verifies if the units are compatible and adhere to predefined coherence rules. `impunity` comes as a decorator to apply on functions: it analyses the source code to check for consistency of physical dimensions, and applies minimal code rewriting if conversions are necessary. Overall, this approach takes the best of type-checking based methods and dynamic methods and provides a robust approach with no overhead at runtime.

Keywords: Python; physical units; static analysis; code rewriting

1. Statement of need

Programming and scripting has become part of the daily routine of scientists of all domains. Many such programs involve physical quantities such as distance, duration, speed values. The importance of dimensional and unit consistency can be compared to the significance of types in programming. Similar to how passing a program through a type-checker eliminates a potential cause of failure, ensuring the dimensional and unit consistency in equations and formulas serves as an initial validation for their correctness. The catastrophic NASA Mars Climate Orbiter probe in September 1999 sadly resulted from a critical confusion between the SI unit of force, newtons (N), and the US customary unit of force, pound-force (lbf). In the aviation domain, the infamous Air Canada's Boeing 767 flight incident in 1983 crash-landed due partially to a mishap between imperial and metric system.

Many scientists often resort to basic techniques for managing physical units, such as incorporating the unit directly into variable names (e.g., `altitude_in_ft`), adding comments in the code, or in the documentation of their functions and routines. Unfortunately, this error-prone approach is widely adopted, despite the availability of libraries designed to address physical unit consistency. Potential users tend to dismiss such libraries, prioritizing their domain-specific interests over learning new tools, fearing that it may impede their software development process. The challenge of ensuring physical unit consistency is perceived as unattractive to scientists, similar to how enforcing type consistency can be unappealing to programmers from various backgrounds.

This paper presents `impunity`, an open-source Python library that makes use of type annotations defined in PEP 486 to ensure physical consistency in code and to automatically modify the source code if necessary. Annotations with physical units look similar to comments in the code; they are **optional**, and consistency checking is conducted **only when hints are available**, minimizing the mental burden on the user. The library has minimal side effects, imposing virtually **no performance overhead**, and exclusively generating **non-breaking warnings** at definition time.

A typical usage would look as follows:

```
from impunity import impunity

@impunity
def speed(distance: "meter", duration: "seconds") -> "m/s":
    return distance / duration

>>> speed(10, 10)
1
```

This example illustrates the most common use case of the library: compatibility of units of measure is verified at the time of definition. For instance, when evaluating the expression `distance / duration`, the resulting value is expressed in m/s, aligning with the provided annotation. Consequently, the function remains unchanged, eliminating any potential computation overhead during runtime.

In the following example, `impunity` detects that dimensions are correct, but that some conversion is necessary if a result in km/h is to be ensured. This is the only situation when `impunity` edits the source code of the function.

```
@impunity
def speed(distance: "meter", duration: "seconds") -> "km/h":
    # impunity rewrites this as: return 3.6 * (distance / duration)
    return distance / duration

>>> speed(10, 10)
3.6
```

However, if the source code contains anomalies or dimension inconsistencies (e.g., sum distances and durations, annotate the return type of the function as ft, etc.), the function is not edited. However, a warning is raised to notify the developer of the error.

After a software review of similar libraries in many programming languages in Section 2, Section 3 explains the design and rationale of the `impunity` library. Section 4 presents a vision for the future of the development and usage of the library, and concludes.

2. Related works

The absence of built-in support for units of measurement (UoM) in major programming languages has led to the development of specialized libraries. In a recent study [1], Mc Keever et al. conducted a comprehensive review and evaluation of UoM libraries. They identified 38 open-source libraries and explored their features and evaluation strategies. The study also investigated the perceptions of developers and practitioners regarding the usability of UoM libraries. The findings highlighted barriers to adoption, such as limited awareness, usability concerns, performance issues, and development processes that exclude unit information. In this section, we present various strategies to take UoM into account in various languages, and how well those suit different programming paradigms. This description of the various strategies that have been explored should help understand the design choices made for `impunity`.

2.1 Type-checking based methods

First attempts to include physical dimensions in the type checking process date back to 1994 with works by Kennedy [2] and Goubault [3] in ML based languages. After Andrew Kennedy was hired by Microsoft, they developed the F# language, with a native support of dimensions in the typing system[1]. In the basic following example, `float` is indexed by a `Measure` generic type.

```
[<Measure>] type meter
[<Measure>] type second

let speed (distance: float<meter>) (duration: float<second>) =
    distance / duration

[<EntryPoint>]
let main argv =
    let distance = 12.<meter>
    let duration = 4.<second>
    let value = speed distance duration
    printfn "Speed: %f m/s" value
    0
```

Basic arithmetic is implemented to check that addition can only be applied on the same units, but multiplication creates a new dimension. If the division operator in `distance / duration` is replaced by an addition with `distance + duration`, a typing error is raised:

```
error FS0001: The unit of measure 'second' does not match the unit of measure 'meter'
```

However, it is important to note that no checking is performed during the automatic formatting of the `value` into a `float` using the `%f` placeholder. In this case, the physical unit remains hardcoded within the string. In 2017, Garrigue and Ly [4] implemented a minimal patch in the OCaml language[2] (which has not been merged into the main branch) with similar functionalities.

```
let speed (distance : <m> dfloat) (time : <s> dfloat) = distance /: time
(* val speed: <m> Dim.dfloat -> <s> Dim.dfloat -> <m / s> Dim.dfloat = fun *)
```

Overall, the advantage of this approach is that dimensions are embedded into types which disappear at runtime: computations are only performed with native types and the unit checking does not create any overhead. As with F#, automatic conversion to commensurable units is not possible, and converting units with properly typed functions falls back to the responsibility of the programmer:

```
let foot : <m/ft> dfloat = create 0.3048
let feet_to_meters (x : <ft> dfloat) = x *: foot
```

2.2 Template based methods

Template programming provided by C++ provides a code generation that is well suited to such a language equipped with a preprocessor and providing function overloading. There are two advantages with this approach: code is generated at compilation time with native types only, so the overhead is minimal, if any. Boost[3] is a C++ library for zero-overhead dimensional analysis and unit/quantity manipulation and conversion.

[1] https://learn.microsoft.com/en-us/dotnet/fsharp/language-reference/units-of-measure
[2] https://github.com/tournemire/ocaml/blob/dim/typing/units.ml
[3] https://www.boost.org/

```cpp
#include <iostream>
#include <boost/units/systems/si.hpp>

using namespace boost::units;

quantity<si::velocity> speed(quantity<si::length> distance,
                                        quantity<si::time> duration)
{
    return distance / duration;
}

int main() {
    double distance = 12, duration = 4;
    quantity<si::velocity> value = speed(distance * si::meters,
                                        duration * si::seconds);
    std::cout << "Speed: " << value.value() << " m/s" << std::endl;
    return 0;
}
```

Errors are raised when operations are not available, at preprocessing time, in a similar fashion to the type checking time in strictly-typed ML family languages. With C++, the relevant part of the error raised during template expansion would look as follows:

```
return distance + duration;
       ~~~~~~~~ ^ ~~~~~~~~
       |          |
       |          quantity<unit<list<dim<boost::units::time_base_dimension,
                                    [...]>,[...]>,[...]>>
       quantity<unit<list<dim<boost::units::length_base_dimension,
                                    [...]>,[...]>,[...]>>
```

A notable caveat specific to the template-based conversion method is its susceptibility to floating precision errors arising from unnecessary conversions to the SI system, unlike other methods that employ compatibility checks to avoid such conversions when not required:

```python
>>> m_to_feet = 0.3048
>>> ((890 * m_to_feet) + (890 * m_to_feet)) / m_to_feet  # expect 1780
1779.9999999999998
```

2.3 The quantity pattern

The quantity pattern [5] is based on the implementation of a class which acts as a container model for physical quantities, consisting of a magnitude and a unit. This approach provides a convenient and structured way to handle units of measurement within a program. This pattern is the most widely implemented in UoM management libraries in Python as it is compatible with the dynamic nature of the language. For every operation, the physical units of all parameters are checked at runtime, and conversion is dynamically performed if needed.

A `Quantity` class (see example below) typically consists of at least two attributes for the magnitude (a floating point number) and a unit (a string, or a specific class). The class usually implements unit conversion (which returns a new Quantity object) and all arithmetic operations, by implementing all adequate dunder methods `__add__` (the `+` operator), `__sub__` (the `-` operator), `__mul__` (the `*` operator), `__div__` (the `/` operator), `__pow__` (the `**` operator), etc.

```
from dataclasses import dataclass
import numpy as np

@dataclass
class Quantity:
    magnitude: float | np.ndarray
    unit: str

    def convert_to(self, new_unit: str) -> Quantity:
        ...

    def __add__(self, other: Quantity) -> Quantity:  # and other operators
        converted = other.convert_to(self.unit)
        return Quantity(self.magnitude + converted.magnitude, self.unit)


>>> result = Quantity(1, 'm') + Quantity(1, 'cm')
>>> result
Quantity(1.01, 'm')
>>> result.magnitude
1.01
```

3. Structure of the library

This section introduces the core structure of the `impunity` library, a more advanced example compatible with the static analysis `mypy` tool, and assesses the performance of the library in comparison to state-of-the-art tools.

3.1 The `@impunity` decorator

`impunity` aims to process physical units in annotated code. It ensures the consistency and accuracy of functions, their parameter variables and return values as long as they are annotated with physical units. `impunity` uses annotations and a decorator function called `@impunity`.

In Python, a decorator function in Python is a higher order function which changes the behaviour of a given function. Let's consider an illustrative code snippet to understand how `impunity` operates in practice. Given the following Python function:

```
def speed(distance: "meters", duration: "seconds") -> "km/h":
    res = distance / duration
    return res
```

If the `decorator` function is used at definition time, the following code is executed and the same function is returned:

```
>>> def decorator(function):
...     print(f"Function definition for {function.__name__}")
...     return function
```

```
>>> @decorator
... def speed(distance: "meters", duration: "seconds") -> "km/h":
...     ...
Function definition for speed
```

```
>>> speed = decorator(speed)  # the @decorator is equal to running this line
Function definition for speed
```

In practice, decorator functions are mostly used to change the behaviour of functions at runtime. A common example is the logging of the execution of a function, or the timing of its execution. Then a new (nested) function must be defined based on the old one:

```
def logger(function):
    def new_function(*args):
        print(f"Executing function {function.__name__} with parameters {args}")
        return function(*args)
    return new_function

@logger
def speed(distance: "meters", duration: "seconds") -> "km/h":
    res = distance / duration
    return res

>>> speed(1, 2)
Executing function speed with parameters (1, 2)
0.5
```

The `impunity` library provides the `@impunity` decorator in order to check the coherence of physical units defined within the code: it traverses (and sometimes modifies) the Abstract Syntax Tree (AST) of the code for the function. Annotated variables and functions are logged for future reference. The AST of the speed function can be depicted in a visual representation, as shown in Figure 1 below:

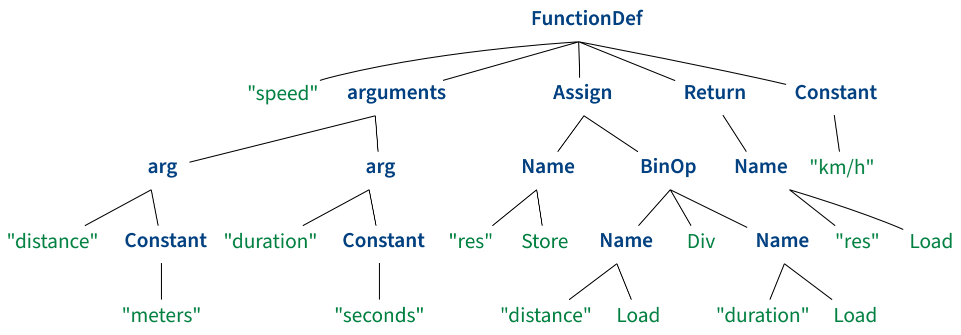

Figure 1. Abstract Syntax Tree (AST) of the annotated speed function

As the decorator function walks through the AST, variables annotated with units of measures (i.e. `distance` and `duration`) are logged. Then, each time a call to an annotated function is detected, `@impunity` compares the expected units of measures from function parameters and return values with the units specified in the function definition (Figure 3). When a mismatch is detected, indicating an inconsistency in units, `impunity` takes one of the following actions:

- if the two units are commensurable, `@impunity` modifies the AST to include a conversion operation;
- if the two units are not commensurable, an `IncommensurableUnits` warning is raised.

In the case of the speed function example, the expected return UoM is `"km/h"` while the UoM inferred from the division between distance and duration variables is `"m/s"`. `impunity` identifies this discrepancy and takes action by modifying the AST accordingly. It introduces a *binary operation* (`BinOp`) node to convert the result to the proper UoM, as depicted in Figure 2.

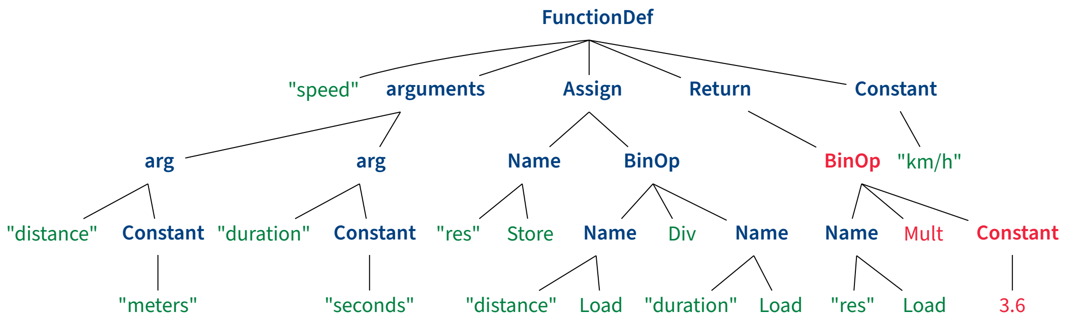

Figure 2. Modified Abstract Syntax Tree (AST)

Here, the constant value of 3.6 is calculated by determining the conversion factor between the two units `"m/s"` and `"km/h"`. `Impunity` leverages the capabilities of the sister Pint library: however, the Pint functionalities are called **only once at definition time**, and not at runtime (i.e. every time the function is executed) resulting in a tremendous gain in performance (see Section 3.3).

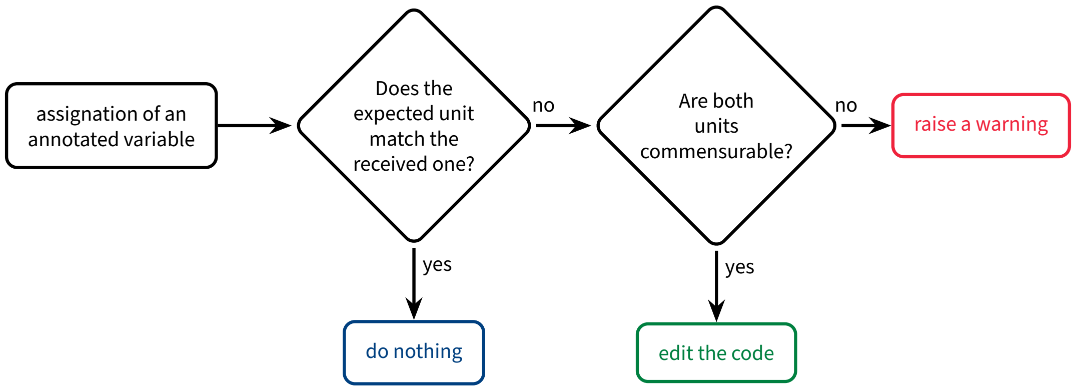
assignation of an annotated variable
Does the expected unit match the received one?
no
Are both units commensurable?
no
raise a warning
yes
yes
do nothing
edit the code

Figure 3. Decision diagram when `@impunity` detects an assignation of a variable with a unit of measure.

3.2 A fully commented running example

In the following, we comment the code printed below to go through various functionalities of the `impunity` library. The code is fully checked with both `mypy` (for types) and `impunity` (for physical units). The full annotated code is available in the repository as `scripts/sample_code.py`.

```python
from typing import TypeVar
from typing_extensions import Annotated  # ①

import numpy as np
from impunity import impunity

GAMMA: Annotated[float, "dimensionless"] = 1.40  # ②
R: Annotated[float, "m^2 / (s^2 * C)"] = 287.05287
STRATOSPHERE_TEMP: Annotated[float, "K"] = 216.65

F = TypeVar("F", float, np.ndarray)  # ③

@impunity
def temperature(h: Annotated[F, "m"]) -> Annotated[F, "K"]:
    """Temperature of ISA atmosphere."""
    temp_0: Annotated[float, "K"] = 288.15
    c: Annotated[float, "K/m"] = 0.0065
    temp: Annotated[F, "K"] = np.maximum(  # ④
        temp_0 - c * h,  # ⑤
        STRATOSPHERE_TEMP,
    )
    return temp

@impunity(rewrite="sound_speed.py")  # ⑥
def sound_speed(h: Annotated[F, "ft"]) -> Annotated[F, "kts"]:
    """Speed of sound in ISA atmosphere."""
    temp = temperature(h)  # ⑦
    a: Annotated[F, "m/s"] = np.sqrt(GAMMA * R * temp)  # ⑧
    return a

@impunity  # ⑨
def main() -> None:
    altitude: "m" = 1000
    print(sound_speed(altitude))  # ⑩ a.
    # 653.9753225425684

    altitude_array: "ft" = np.array([0, 3280.84, 5000, 10000, 30000])
    x = sound_speed(altitude_array)  # ⑩ b.
    print(x)
    # [661.47859444 653.9753223  650.00902555 638.3334048  589.32227624]

    y: "m/s" = sound_speed(altitude_array)  # ⑩ c.
    print(y)
    # [340.29398803 336.43397136 334.39353203 328.3870738  303.173571  ]

if __name__ == "__main__":
    main()
```

① `typing.Annotated` has been supported since Python 3.11. In earlier versions, the object is part of the `typing_extensions` package. Annotated allows the developer to add extra information to type annotations. `impunity` supports this notation to provide unit annotations which do not conflict with static type checkers such as `mypy` or `pylance`. Both notations are supported by `impunity`, but only `Annotated` is supported by static checkers.

② Global variables can be annotated without being wrapped in an `@impunity` decorated function. At the time being, decorated global variables are only partially supported due to limitations of the language:

```
>>> from constants import GAMMA  # @impunity cannot find the unit information
>>> import constants
>>> constants.GAMMA  # @impunity finds the unit information
```

③ *(about the typing module in Python)* Such `TypeVar` definition can be convenient for typing purposes: in the `temperature` function, if `h` is a float, the return value is a float; if `h` is a `NumPy` array, the return value is also a `NumPy` array.

④ Some functions (e.g. all NumPy functions) cannot be fully *impunified*. Here, the return value of `np.maximum()` is **undefined**: in order to propagate the physical unit information to the return type, and possibly convert the return value of the function into a compatible unit, it **must** be annotated.

⑤ Inside the function call, unit consistency for the expression `temp_0 - c * h` is ensured.

⑥ The `@impunity` decorator can also be called with arguments. Here, the amended version of the `sound_speed` function is dumped in `sound_speed.py`: the dumped code may not be executable as is; this mode is intended for debugging purposes.
Here is the result of the *impunification* (as printed in the output file):

```
def sound_speed(h: Annotated[F, 'ft']) -> Annotated[F, 'kts']:
    """Speed of sound in ISA atmosphere"""
    temp = temperature(h * 0.30479999999999996)
    a: Annotated[F, 'm/s'] = np.sqrt(GAMMA * R * temp)
    return a * 1.9438444924406049
```

⑦ *(to be compared with ④)* The annotation of the `temp` variable is here **optional**: `impunity` knows the physical unit for the return value of the `temperature` function from its signature. As a result, a conversion for `h` is triggered, and the physical unit for `temp` is considered to be "K". The `temp` variable may still be annotated as `Annotated[F, "K"]` for redundancy, or as another compatible unit (e.g., `Annotated[F, "deg_C"]`) in order to include a conversion operation at the time of assignment.

⑧ *(to be compared with ④)* Again, the return unit of the `np.sqrt()` is **undefined**. When clarified that it is expressed in "m/s", automatic conversion to "kts" may be automatically performed.

⑨ Only *impunified* functions can be processed by the unit checking process. If the variables `altitude` or `altitude_array` are defined (even with annotations) in the Python REPL console, or outside a function in a Python file, no checking or transformation can be performed.
Note that in this `main()` function, we switch back to simple unit annotations, supported by `impunity`, but incompatible with `mypy`, which raises errors.

⑩ a. The first sound speed value is directly printed, therefore, it is expressed in the unit specified in the function annotation ("kts"). b. With the array for altitude values (returning variable x), the variable is also automatically labelled as "kts". Redundancy is of course possible. c. If the user expects a result in a different unit (variable y), the result of the function **must** be assigned to an annotated variable so that it can be converted prior to printing.

3.3 Performance assessment

Mc Keever et al. [1] justifies the fact that *Python-enclined* scientists do not use any UoM checking library with the computation overhead they often induce during execution. Python is a dynamic language, and most tools provide dynamic checking of the UoM, every time a function is called with potentially different parameters and different units of measures. impunity statically checks the UoM of variables in the code based on annotations and modifies the AST at definition time. The following assessment shows that `impunity` is a lot quicker than four existing libraries, chosen among other Python libraries dealing with UoM for their support, regular updates and community.

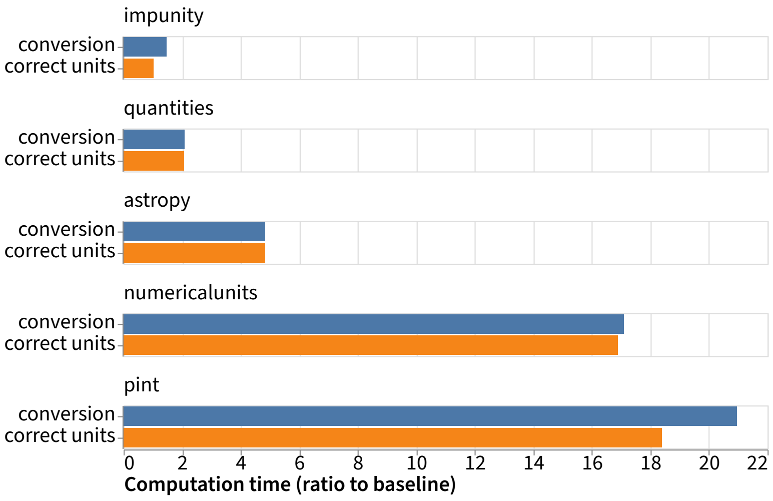

Figure 4. Comparison of performance for several libraries handling units of measurements : `impunity` has no overhead when units are correct, and a minimal one when units need to be converted.

- **numericalunits** (https://github.com/sbyrnes321/numericalunits) uses a complete set of independent base units (meters, kilograms, seconds, coulombs, kelvins) that are defined as *randomly-chosen positive floating-point* numbers, different for all executions. If units are consistent, the randomness disappears; if not, two executions of the same code return different values.

- **astropy** [6] is a Python package offering functionalities aimed at astronomers and astrophysicists. It also includes an implementation of the quantity design pattern. `astropy` also implements a decorator to check calls of functions with quantity parameters.

- **Pint** (https://github.com/hgrecco/pint) also provides an implementation of the quantity design pattern as a standalone library. It is flexible and provides good integration with other scientific libraries like `Pandas` (through extension types) or `NumPy`.

- **Quantities** (https://github.com/python-quantities/python-quantities) is designed to handle arithmetic and conversions of physical quantities, which have a magnitude, dimensionality specified by various units, and possibly an uncertainty. `Quantities` builds on the popular `NumPy` library and is designed to work with `NumPy`'s standard ufuncs, many of which are already supported.

For each library, two different use-cases are considered based on the example `speed` function. One with variables annotated with the correct units (meters and seconds), and one with different but commensurable units (meters and hours). In both cases, two `NumPy` arrays of shape (10000,) are sent as parameters. The computation time over 300 iterations is then averaged. Execution times for both use-cases are displayed in Figure 4.

As observed, the overhead induced by `impunity` is minimal. This is mainly due to the difference between the dynamic checking of the other libraries and the static analysis done by `impunity`. By changing the AST directly before execution, `impunity` limits its overhead to the multiplications added to keep the units coherent between each others. This is also why, when UoM are identical, the overhead is non-existent.

4. Conclusion and future developments

This paper presents `impunity`, a new library for managing units of measure in Python. The most notable improvements compared to other similar libraries boil down to:

- **Integration with legacy code.**
  - The library leverages the existing Python type hinting system: it is **non-intrusive, optional**, and compatible with widely used Python development tools and frameworks;
  - It requires only annotations and a decorator function, not more than what a meticulous developer would write as comments in his code. Annotations naturally appear in the documentation later in the process;
  - Existing code **does not break after a dependency library is *impunified*:** native structures are not changed in the process, and possible inconsistencies only raise warnings.
- **Minimal overhead.** The library is designed with minimal overhead: it does not significantly impact the runtime performance of scientific computations. If annotations do not raise any need for unit conversions, existing code is left as is, **running with native structures**.
- **Completeness and versatility.** `impunity` provides a comprehensive set of predefined physical unit types and support the creation of custom units to accommodate a wide range of scientific domains and applications.

The authors created this library because of numerous unit incompatibility errors in their existing code base, which depends on different aeronautical libraries using both SI (e.g. meters) and non-SI units (feet, knots, nautical miles, etc.). A first attempt at rewriting legacy code with existing libraries caused an unacceptable drop in performance, more than 20 times slower.

The future of the `impunity` library will be mostly led by the development of other existing open-source Python libraries for aviation, e.g. `pitot` (https://github.com/open-aviation/pitot), `OpenAP` [7] and `traffic` [8]. In the long run, a standalone code analysis program, possibly integrated with IDEs, would be a great addition. A complete integration with popular Python code analysis tools, such as `pylance` and `mypy`, would also improve the usability and promote the adoption of best practices for unit management in scientific codebases. We also aim at providing proper annotations for units in typed records, such as named tuples, typed dictionaries, `dataclass` structures or `Pandas` DataFrame.

Author contributions

- Antoine Chevrot: Conceptualization, Methodology, Software, Validation, Writing – Original draft
- Xavier Olive: Conceptualization, Validation, Writing – Original draft

Reproducibility statement

`impunity` is an open-source library under MIT licence. The source code is available on GitHub at https://github.com/achevrot/impunity. The performance assessment presented in this paper can be reproduced according to the readme file in the `scripts/performance` folder.

This software is archived at https://doi.org/10.5281/zenodo.8167457

References

[1] Steve McKeever, Oscar Bennich-Björkman, and Omar-Alfred Salah. "Unit of measurement libraries, their popularity and suitability". In: *Software: Practice and Experience* 51.4 (2021), pp. 711–734. DOI: 10.1002/spe.2926.

[2] Andrew Kennedy. "Dimension Types". In: *Proceedings of the 5th European Symposium on Programming: Programming Languages and Systems*. ESOP '94. 1994. DOI: 10.5555/645390.651419.

[3] Jean Goubault and Michiel Korthals. "Inférence d'unités physiques en ML". In: *5e journées francophones des langages applicatifs*. 1994.

[4] Jacques Garrigue and Dara Ly. "Des unités dans le typeur". In: *28e journées francophones des langages applicatifs*. 2017.

[5] Martin Fowler. *Analysis patterns: reusable object models*. Addison-Wesley Professional, 1997.

[6] Astropy Collaboration et al. "The Astropy Project: Sustaining and Growing a Community-oriented Open-source Project and the Latest Major Release (v5.0) of the Core Package". In: *apj* 935.2, 167 (Aug. 2022), p. 167. DOI: 10.3847/1538-4357/ac7c74. arXiv: 2206.14220 `[astro-ph.IM]`.

[7] Junzi Sun, Jacco Hoekstra, and Joost Ellerbroek. "OpenAP: An open-source aircraft performance model for air transportation studies and simulations". In: *Aerospace* 7.8 (July 2020), p. 104. DOI: 10.3390/aerospace7080104.

[8] Xavier Olive. "traffic, a toolbox for processing and analysing air traffic data". In: *Journal of Open Source Software* 4.39 (2019). DOI: 10.21105/joss.01518.