Latexify-py

Alexander D. Cai
Harvard College
alexcai@college.harvard.edu

Abstract

We present the adzcai/latexify-py Python package for converting Python source code to LaTeX pseudocode. adzcai/latexify-py is intended for authors of technical texts who wish to provide a readable LaTeX pseudocode description of an algorithm alongside a Python reference implementation. adzcai/latexify-py is implemented using the Python ast package. It extends a previous implementation at https://github.com/google/latexify_py by focusing on modularity and improved user experience. The source code is available at https://github.com/adzcai/latexify-py.

1 Introduction

Programming courses often use a textbook as a reference. Such textbooks often describe algorithms using **pseudocode**, which ensures that the algorithm is understandable to readers regardless of their programming language background. Using LaTeX to write this pseudocode also enables the use of mathematical notation, which often provides a much more concise and familiar presentation that is implementation-agnostic. Another opposing paradigm is the **literate programming** paradigm, as represented by the use of Jupyter notebooks or R Markdown files, in which the algorithms are implemented using a specific programming language. These have the benefit of being *executable*, but force the author to choose language-specific or, often, framework-specific implementation details. adzcai/latexify-py supports translation from a low-level Python implementation of an algorithm to a high-level pseudocode description of that algorithm. (Note that this has the opposite direction as most compilers, which take in high-level code and compile it down to low-level machine instructions.) While Python, as a high-level programming language, is already highly readable, supporting translation to pseudocode still offers the benefits outlined above. This allows authors of pedagogical programming texts the benefits of both worlds. The most relevant previous work in this direction is the project google/latexify_py on GitHub, which has 7.3k stars at the time of writing. This project builds on top of google/latexify_py with a focus on extensibility and user experience. Our main contributions are as follows:

1. **Plugin system** 2.2. We rewrite google/latexify_py in terms of a plugin system that allows users to extend the package by implementing their own translation rules using the Python ast package.

2. **Identifiers** 2.3. We enable users to replace Python identifiers with arbitrary LaTeX strings and generalize replacement rules.

We also implement various minor improvements and bugfixes to google/latexify_py. All of our contributions come with unit tests implemented with pytest.

Preprint. Under review.

2 Methods

In this section, we discuss the key features of `adzcai/latexify-py` and its improvements upon `google/latexify_py`.

2.1 Overall procedure

Python exposes the builtin `ast` package for enabling programmers to parse and transform the abstract syntax tree (AST) corresponding to a given string of Python source code. It provides the `ast.parse` command for parsing a Python code string into its AST representation. It also provides the two classes `ast.NodeTransformer` and `ast.NodeVisitor` for recursively traversing the AST. `adzcai/latexify-py` consists mostly of subclasses of these two classes. The translation from Python to LaTeX takes place in two stages:

1. **AST transformation**. The first stage transforms the AST itself using subclasses of Python's `ast.NodeTransformer`. For example, the `AugAssignReplacer` class of `google/latexify_py` replaces Python's **augmented assignment operations** (e.g. `x += 1`) with the explicit assignment `x = x + 1`.
2. **AST translation**. The second stage recursively translates the AST into LaTeX code using subclasses of Python's `ast.NodeVisitor`. For example, our `IdentifierConverter` class converts Python identifiers into the corresponding LaTeX strings.

2.2 Plugin system

Our main contribution was rewriting the package in terms of an extensible **plugin system**. The original package consisted of a fixed set of program transformations. Users could opt in or out of specific features by providing keyword arguments to the public API. For example, the `use_math_symbols` keyword argument could be set to true to replace a variable name corresponding to a Greek letter with the corresponding Greek letter:

```
def greek_demo(alpha, beta, Omega):
    return alpha * beta + Omega

with_symbols = latexify.function(greek_demo, use_math_symbols=True)
without_symbols = latexify.function(greek_demo, use_math_symbols=True)
```

Here, `with_symbols` results in

$$\mathrm{greek\_demo}(\alpha, \beta, \Omega) = \alpha\beta + \Omega$$

while `without_symbols` results in

$$\mathrm{greek\_demo}(\mathrm{alpha}, \mathrm{beta}, \mathrm{Omega}) = \mathrm{alpha} \cdot \mathrm{beta} + \mathrm{Omega}.$$

This system does not enable the user to customize handling of specific AST nodes. This violates the principle that frameworks should have **escape hatches** (e.g. Rust's `unsafe` or React's `useRef`) that enable the user to circumvent the framework's design choices.

We implement an extensible **plugin system** to support user customization. The implementation is simple and modular. We create a `PluginList` class inheriting from `ast.NodeVisitor`. Its constructor takes in a list of `ast.NodeVisitor` instances. Its `visit` method, which takes in an AST node and outputs a LaTeX string, simply iterates through the list of `ast.NodeVisitors` and calls the first one that supports translation of the AST node.

The list of `ast.NodeVisitors` must in particular be subclasses of our `Plugin` class, which replaces all calls to `visit` with calls to the `visit` function of the stack that it is part of. This ensures that, for example, if a user wishes to override some `adzcai/latexify-py` functionality by providing their plugin, the translation will always check the user's plugin before falling back to the library implementation.

We also refactor the functionality of `google/latexify_py` in terms of the new plugin system. In particular, the `ExpressionCodegen` class of `google/latexify_py` had grown to include LaTeX translations for various `numpy` commands, which seemed odd to include in the base package since these depended on `numpy` while the actual expressions would not. As a result, `google/latexify_py` would convert expressions such as `np.linalg.inv(np.array([[1, 1], [-1, 1]]))` purely based on the identifier names, even if `numpy` was not installed in the environment. `adzcai/latexify-py` implements this functionality in a separate `NumpyPlugin` that can be optionally included. We provide a `default_stack` function that takes in a list of plugins and appends our `ExpressionCodegen` and `IdentifierConverter` core plugins, which provide the core translation functionality.

We have already found this plugin system to be highly effective in allowing users to customize code transformation with only basic knowledge of how the Python AST works. For example, `jaxtyping` (`https://docs.kidger.site/jaxtyping/`) is a Python package that supports type annotations of shapes of tensors (e.g. vectors and matrices). An $M \times N$ matrix A could be annotated as `A: Float[Array, "M N"]`. It would be impossible for the user to implement this behavior using `google/latexify_py`. Using `adzcai/latexify-py`, however, the user can write a simple rule-based `Plugin`:

```python
class JaxTypingPlugin(Plugin):
    def visit_Subscript(self, node: ast.Subscript):
        if (
            isinstance(node.value, ast.Name)
            and node.value.id in ("Float", "Int")
            and isinstance(node.slice, ast.Tuple)
            and len(elts := node.slice.elts) == 2
            and isinstance(elts[0], ast.Name)
            and elts[0].id == "Array"
            and isinstance(elts[1], ast.Constant)
            and isinstance(dim := elts[1].value, str)
        ):
            dim = r"␣\times␣".join(dim.strip().split())
            return r"\mathbb{R}^{" + dim + r"}"
        raise NotImplementedError

def f(A: Float[Array, "M␣N"]):
    return A
```

Then running `latexify.algorithmic(f, plugins=[JaxTypingPlugin()])` returns the reasonable result

```
**function** $f(A : \mathbb{R}^{M \times N})$
    **return** $A$
**end function**
```
.

We see that our plugin system gives users the ability to easily customize the package's behavior.

2.3 Identifiers

`google/latexify_py` only allowed replacing identifiers (e.g. variable names, argument names) with other valid Python identifiers (see `transformers.identifier_replacer.py`). That is, it was impossible to associate a given Python identifier with an arbitrary LaTeX expression. This was because the `IdentifierReplacer` class was implemented as a subclass of `ast.NodeTransformer`, which only supports replacing an AST node with another valid AST node. It also did not support replacement of nested identifiers such as `np.linalg.eigvals` with a single LaTeX expression. Doing so in the original implementation would require adding `np.linalg` to a list of prefixes that would be entirely removed in the transformation process.

In `adzcai/latexify-py`, we add a `IdentifierConverter` class that allows converting a Python identifier to an arbitrary LaTeX string. We keep the original `IdentifierReplacer`, though, since this can still be used to alias one identifier to another without having to re-implement the functionality. For example, if a user wanted to replace `my_matrix_inverse(A)` with A^{-1}, doing so would be challenging using the IdentifierReplacer, since the resulting text (the -1 superscript) does not remain in the same place. Rather, the user could replace `my_matrix_inverse` with `np.linalg.inv` to achieve this functionality.

We acknowledge, however, that exposing both the `IdentifierConverter` and `IdentifierReplacer` classes is potentially confusing, and are curious about other possible solutions to this design issue.

2.4 Miscellaneous improvements

We also implement numerous minor improvements, including:

- `elif` **statements**. These are not explicitly represented in the Python AST and must be inferred from the code structure. Specifically, if the body of an `else` branch contains just a single `if` statement, these are merged into an `elif` statement in the outputted LaTeX.
- **Nested functions**. This is useful for defining sub-procedures within an algorithm.
- **File saving**. We add a `to_file` keyword argument that enables saving the resulting LaTeX code to a `.tex` file. This allows LaTeX code to be generated from a Python script or Jupyter notebook and automatically included into the source code of a LaTeX project.
- **Tests**. We implement tests using `pytest` for all of our contributions. We use the existing test suite to verify that our modifications are backwards-compatible with `google/latexify_py`.

This library is currently used extensively in another project by the author, a textbook for reinforcement learning (`https://rlbook.adzc.ai/`). Reinforcement learning algorithms can be quite complicated to describe and implement, and thus provides an excellent test bed for exploring limitations of `adzcai/latexify-py`. Below, we include an example algorithm including a nested function, plugin usage, vector shape annotations, and custom identifier replacements.

```python
def reinforce_estimator(
    env: gym.Env, pi, theta: Float[Array, " D"]
):
    tau = sample_trajectory(env, pi(theta))
    nabla_hat = jnp.zeros_like(theta)
    total_reward = sum(r for _s, _a, r in tau)
    for s, a, r in tau:
        def policy_log_likelihood(theta: Float[Array, " D"]) -> float:
            return log(pi(theta)(s, a))
        nabla_hat += jax.grad(policy_log_likelihood)(theta) * total_reward
    return nabla_hat

result = latexify.algorithmic(
    reinforce_estimator,
    use_math_symbols=True,
    plugins=[
        SumProdPlugin(),
        JaxTypingPlugin(),
    ],
    custom_identifiers={"jax.grad": r"\nabla"},
)
```

Then `result` gives the following LaTeX code:

```
function reinforce_estimator$(\text{env} : \text{gym.Env}, \pi, \theta : \mathbb{R}^D)$
    "Estimate the policy gradient using REINFORCE."
    $\tau \leftarrow \text{sample\_trajectory}(\text{env}, \pi(\theta))$
    $\widehat{\nabla} \leftarrow \text{jnp.zeros\_like}(\theta)$
    $\text{total\_reward} \leftarrow \sum_{(\_s, \_a, r) \in \tau} (r)$
    for $(s, a, r) \in \tau$ do
        function policy_log_likelihood$(\theta : \mathbb{R}^D)$
            return $\log \pi(\theta) \left(s, a\right)$
        end function
        $\widehat{\nabla} \leftarrow \widehat{\nabla} + \nabla(\text{policy\_log\_likelihood}) \left(\theta\right) \cdot \text{total\_reward}$
    end for
    return $\widehat{\nabla}$
end function
```

3 Conclusion

We implement a Python package, `adzcai/latexify-py`, that is able to convert Python source code into LaTeX pseudocode. This enables authors of pedagogical programming texts to simultaneously provide a relatively lower-level Python implementation as well as a higher-level pseudocode description that is able to use mathematical notation. Our main contribution is to rewrite an existing library, `google/latexify_py`, in terms of a plugin system that enables users to easily express their own conversions using the Python `ast` package.

There are many future directions yet to be implemented. One would be to make use of the `algpseudocode` LaTeX package's `\Require` and `\Ensure` functions to include documentation about the function arguments and return value. This would require parsing docstrings of various formats. Another step would be to implement a **probability** plugin that intelligently understands taking the expectation of a function over some distribution, sampling various random variables, etc. The plugin system greatly improves the process of implementing and testing extensions to the translation functionality. We hope that `adzcai/latexify-py` will enable authors to provide readable pseudocode descriptions of their Python algorithms.