

User Manual:

Python code for generating C code for piecewise Chebyshev approximation

JOACHIM WUTTKE, Forschungszentrum Jülich GmbH, Germany

ALEXANDER KLEINSORGE, Technische Hochschule Wildau, Germany

This user guide documents Python and C software that supports the piecewise polynomial approximation of real-valued functions as described in our article “Algorithm 1xxx: Code generation for piecewise Chebyshev approximation”.

1 Introduction

The free and open-source Python package *ppapp* (piecewise polynomial approximation) contains Python and C code for piecewise polynomial function approximation. This manual addresses the approximation of a real-valued function of a real variable as described in our article “Algorithm 1xxx: Code generation for piecewise Chebyshev approximation” [1]. The software is released under the GNU General Public License Version 3 or higher; other licensing is negotiable.

The package is available through three distribution channels:

- the **project repository** at the GitLab instance of Forschungszentrum Jülich [2], with project root in directory `R/`;
- the **PyPI package** *ppapp* [3], installable via `pip install ppapp`;
- the **archival zip package** in the Collected Algorithms of the ACM (CALGO) [4], which contains release 1.1.0 as described in [1], with project root in directory `src/`.

The software may be further improved if new ideas arise. In particular, the future top-level directory `C/` shall provide the tile-wise Taylor approximation to complex functions in complex domains [5].

Section 2 describes the code generator that produces C source files containing tables of polynomial coefficients that approximate a specific function f . Section 3 describes how the generated C code is used.

2 Code generator

2.1 Install and run

The generator code is written in the programming language Python3. It depends on the arbitrary-precision library *Arb* that is part of *FLINT*, Fast Library for Number Theory [6]. A Python wrapper of FLINT is available from pypi.org as package *python-flint* [7].

The software can be installed from PyPI with:

```
pip install ppapp
```

This automatically installs the *python-flint* dependency and provides the command:

```
ppapp
```

Alternatively, when working from source, run the *ppapp* module from the root directory indicated in Sect 1:

```
cd <ppapp root directory>
python -m ppapp <mode> <arguments>
```

Authors' Contact Information: [Joachim Wuttke](mailto:j.wuttke@fz-juelich.de), j.wuttke@fz-juelich.de, Forschungszentrum Jülich GmbH, Jülich Centre for Neutron Science at MLZ, Lichtenbergstraße 1, 85748 Garching, Germany; [Alexander Kleinsorge](mailto:alkl9873@th-wildau.de), alkl9873@th-wildau.de, Technische Hochschule Wildau, Studiengang Telematik, Hochschulring 1, 15745 Wildau, Germany.

Running ppapp without arguments prints a summary of available modes:

No mode given

Usage:

```
ppapp i <f_module>          - run initial tests from my_testcases
ppapp v <f_module> <x>       - print function value f(x)
ppapp n <f_module> <M> <Nmax> <E> - print N_min(M',E), up to given Nmax
ppapp e <f_module> <M> <N>   - print maximum relative error, in units of epsilon
ppapp c <f_module> <M> <N> [<E>] - print plain table of Chebyshev coefficients c_n
ppapp p <f_module> <M> <N> [<E>] - print plain table of economized coefficients p_m
ppapp s <f_module> <M> <N> [<E>] - print C source defining economized coefficients p_m
ppapp t <f_module> <M> <Nx0> <E> - print C source defining test cases
ppapp r <f_module> <M> <N> <n>   - print deviation in units of epsilon, for <n> points
ppapp H <f_module> <M> <N> <n>   - print deviation histogram, for <n> random points
```

where

```
<f_module> - path to function definition file (e.g., 'mydir/f_imwofx.py')
<M>         - integer M >= 0 specifies 2^M subdomains per octave
<N>         - integer N >= 1 is the polynomial degree
<E>         - double E > 0 is the maximum relative error, in units of epsilon=2^-53
<Nx0>       - number of extra (non-Chebyshev) octaves on each side of the Chebyshev range
<n>         - integer n >= 1 is the total number of test points
```

Installation info:

```
- Invoked via:    /G/sw/ppapp/R/ppapp/__main__.py
- Running from:   source
- Package directory: /G/sw/ppapp/R/ppapp, with subdirectories:
  - docs/         user manual (PDF)
  - demo_functions/ example high-precision function definitions
```

2.2 Function argument

All commands require a function definition file as the second argument:

```
ppapp v /path/to/my_function.py 1.0
```

The function definition file specifies the interface between the generic approximation machinery and the specific target function that is to be approximated. It defines one function and two global objects:

```
def my_arb_f(X: arb, prec: int) -> arb:
    """Evaluates f(x) with given precision"""
    ...

my_domain: Tuple[float, float] = (a, b)
my_testcases: List[Tuple[float, float, float]] = [
    (x, f_expected, tolerance),
    ...
]
```

Function `my_arb_f` computes $f(x)$ in interval arithmetic with a precision of `prec` binary digits, using the python-flint wrapper for Arb [6]. The tuple `my_domain` contains the limits of the total domain $[a, b]$. The entries in the list `my_testcases` are triples $(x, f_{\text{expected}}(x), \text{tol})$. The test suite will fail unless for each test case, the function value $f(x)$, computed by our arbitrary-precision function `my_arb_f`, agrees with $f_{\text{expected}}(x)$ with a relative error not larger than `tol`.

As an example, the package includes `ppapp/demo_functions/imwofx.py` that implements the function

$$f(x) := \exp(-x^2) \operatorname{erfi}(x) \equiv \operatorname{Im} w(x) \quad (1)$$

introduced in [1, Sect 1.3]. The arbitrary-precision computation of f is straightforward because Arb supports `erfi(x)` as built-in method `arb.erfi()`. The domain is $[a, b] = [0.5, 12)$. The test cases allow a tolerance of 10^{-5} , i. e. they are meant to ensure the basic correctness of the high-precision implementation but not its accuracy. The latter is not a concern because of the intrinsic accuracy control of Arb. To support any other function f , one needs to write a new implementation file, based on the model provided by `ppapp/demo_functions/imwofx.py`.

A second example, `ppapp/demo_functions/polynomial.py`, implements the simple polynomial

$$f(x) := x^3 - x^2 + x - 1 \quad (2)$$

over the domain $[1.5, 4)$. This serves as a useful test case because the Chebyshev approximation of an exact degree-3 polynomial should yield only four significant coefficients (p_0 through p_3), with all higher-order coefficients being negligible (at the level of numerical noise).

2.3 Run modes

The program `ppapp` operates in different modes that are selected by a letter provided as first command-line argument. All output is written to `stdout`; use redirection to save it in a file.

Mode i: initial test of high-precision function.

```
ppapp i <f_module>
```

Tests the function implementation against the test cases defined in `my_testcases`. This verifies that the high-precision reference function is basically correct before using it for coefficient generation.

Mode f: compute function value.

```
ppapp v <f_module> <x>
```

Computes a single function value $f(x)$.

Mode n: compute minimal polynomial degree.

```
ppapp n <f_module> <M> <Nmax> <relerr>
```

Computes the minimal polynomial degree for which the relative error, in units of ϵ , is not larger than the given `relerr`. This mode has been used to produce Table 1 in [1].

Mode e: estimate error bound.

```
ppapp e <f_module> <M> <N>
```

Prints an upper bound for the total relative error in units of ϵ . Based on results from modes `n` and `e`, make your choice of M and N , as discussed in [1, Sect 4.4].

Modes c, p: Compute coefficient tables.

```
ppapp c <f_module> <M> <N> [<relerr>]
ppapp p <f_module> <M> <N> [<relerr>]
```

Prints a table of Chebyshev coefficients c_{ln} (mode c) or of economized polynomial coefficients p_{ln} (mode p). These modes have been used to produce Fig 3 of [1]. The optional `relerr` argument, in units of ϵ , activates tests that ensure that the relative error never exceeds this bound.

Mode s: Generate target C source code.

```
ppapp s <f_module> <M> <N> [<relerr>]
```

Prints C source code with arrays that hold the p_{ln} .

Mode t: Generate test C source code.

```
ppapp t <f_module> <M> <Nxo> <relerr>
```

Prints C source code with test cases, covering a range $a2^{-N_{xo}} \dots b2^{N_{xo}}$ that extends beyond the Chebyshev domain if $N_{xo} > 0$.

Mode r: Tabulate representative deviations.

```
ppapp r <f_module> <M> <N> <n>
```

Computes and prints the relative deviation δf between the polynomial approximation and the high-precision reference function, in units of ϵ , for n points regularly spaced on a logarithmic scale. This mode has been used to produce Fig 4a in [1].

Mode H: Produce deviations histogram.

```
ppapp H <f_module> <M> <N> <n>
```

Computes and prints a histogram of relative deviations for n random-drawn points. This mode has been used to produce Fig 4b in [1].

2.4 Hexadecimal output

The output files are self-explaining thanks to initial comment lines. Let us explain just one detail: In the auto-generated C source files, floating-point numbers are written in hexadecimal format, like

```
0x1.ef90904c7eeeep-2, 0x1.461380c17af85p-8, -0x1.9d5e2f887fe99p-15, -0x1.35c476fb1ab4ap-24, ...
```

Note that the letter p is followed by the base 2 exponent in decimal notation, i.e. $0x1.8p-13$ is $1.5 \cdot 2^{-13}$.

2.5 Unit tests

The PyPI package includes a comprehensive test suite. When working with the repository, the tests can be run with:

```
python3 -m pytest tests/ -v
```

The test suite covers mathematical helper functions, power law analysis, error bounds, subdomain computation, output formatting, function modules, Chebyshev coefficient computation, polynomial approximation accuracy, and full pipeline integration.

3 Usage demonstrator

The C source code generated by ppapp can be used in C or C++ projects to compute function values $f(x)$. In a typical application, this code would be integrated with other code that evaluates f outside the intermediate domain considered here, using expansions for small and large x .

A C demonstration implementation is available in subdirectory ppapp/docs/target-C-code-demo. It shows how to use the auto-generated coefficients for efficient function evaluation.

The target algorithm for evaluating the piecewise polynomial approximation is also illustrated in the Python module ppapp.target_algorithm (included in the PyPI package). This plain Python implementation is for illustration purposes only. The ppapp project is designed to generate C code for optimized high-throughput computation. For production use from Python, the evaluation code should be implemented as a C extension module or at least use NumPy for vectorized operations. The Python script should not be used as a template for production code.

3.1 Alignment specifier

In order to minimize the number of cache loads, the auto-generated arrays that hold tables of polynomial coefficients must start at the beginning of a 64 bytes memory block [1, Sect 2.5]. This is achieved by defining these arrays as

```
alignas(64) static const double ppapp_Coeffs0[...] = { ... };
```

The specifier alignas is defined in the language standards C23 and C++11. For older versions of C it is not in the standard, but may be supported as a compiler extension. C11 has a specifier _Alignas. For even older language variants one would depend on compiler-specific attributes.

The Python code generator automatically inserts the correct alignment directives and pads coefficient tables with zeros when necessary to maintain proper alignment [1, Sect 2.5].

In the project repository, the C demonstration executable is built and run with the commands

```
cd <target C directory>
mkdir build
cd build
cmake ..
make
./test_computation
./run_computation
```

Program test_computation runs automated tests using test cases generated by ppapp mode t. It verifies that the polynomial approximation meets the specified error bounds. Program run_computation demonstrates the usage of the generated code by computing and printing function values at a set of sample points.

References

- [1] Wuttke J and Kleinsorge A *Algorithm 1xxx: Code generation for piecewise Chebyshev approximation*. submitted to ACM TOMS.
- [2] Wuttke J et al (2025) PPAPP, Piecewise Polynomial APPROXimation generator. <https://jugit.fz-juelich.de/mlz/ppapp> (2025).
- [3] Python Package Index. Project ppapp. <https://pypi.org/project/ppapp>.
- [4] Wuttke J and Kleinsorge A *Code generating code for piecewise Chebyshev approximation. Code snapshot and user manual. Collected Algorithms of the ACM, submitted*.
- [5] Wuttke J *Code generation for computing an analytical function with near machine precision on square tiles, with application to the Faddeeva function*. in preparation.
- [6] The FLINT team (2024) FLINT: Fast Library for Number Theory. <https://flintlib.org> (2024). Version 3.1.2.
- [7] Python Package Index. Project python-flint. <https://pypi.org/project/ppapp>.