

User Manual:

Python code for generating C code for tile-wise Taylor approximation

MIRIAM MORENO SIERRA, Forschungszentrum Jülich GmbH, Germany

JOACHIM WUTTKE, Forschungszentrum Jülich GmbH, Germany

This user guide documents Python software that supports the tile-wise Taylor approximation of a given analytic function $f(z)$ in the complex domain, as described in the article “Code generation for computing an analytic function with near machine precision on square tiles, with application to the Faddeeva function” [1].

1 Introduction

The free and open-source package *ttapp* contains Python code that generates C code that uses tabulated Taylor series on square tiles to approximate an analytic function in a finite complex domain. The implemented algorithms are described in a paper [1]. This manual explains how to use package *ttapp*.

The software is released under the GNU General Public License Version 3 or higher; other licensing is negotiable. It is available from two locations:

- the **project repository** at the GitLab instance of Forschungszentrum Jülich [2];
- archival copy deposited in Zenodo, <https://doi.org/10.5281/zenodo.22277192>.

The library *libcerf*, for which the tables are generated, ships no copy of *ttapp*; its file `dev/ttapp/README` points to the repository and to this manual.

Section 2 describes the code generator that produces the C source files and its parameters. The generated C source files are described with mode `c` in Section 2.4.

2 Code generator

2.1 Dependencies

The generator code is written in the programming language Python 3, version 3.10 or later, and rests on four packages:

```
pip install python-flint numpy scipy ortools
```

python-flint [3] binds the C library FLINT [4], whose ball arithmetic yields every high-precision value we use: the Taylor coefficients of $f(z)$, and the reference values against which an implementation is measured. Mind the name: the package is `python-flint`, *not* `flint`, which on PyPI is an unrelated package; the module it installs is nonetheless imported as `flint`.

numpy, scipy [5] (SciPy 1.9 or later) provide the solver HiGHS for mixed-integer linear programming, called as `scipy.optimize.milp`, which is cover algorithm `p`.

ortools [6] provides the CP-SAT solver, which is cover algorithm `s`.

Modes `e` and `H` need one further ingredient that is not a Python package: the implementation of f whose accuracy they measure. They take it from the function module (Section 2.3), which offers it as the attribute `my_impl_callable`; where the module gets it from is the module’s own affair, and *ttapp* itself loads no compiled library. The example

Corresponding author: Joachim Wuttke, j.wuttke@fz-juelich.de, Forschungszentrum Jülich GmbH, Jülich Centre for Neutron Science at MLZ, Lichtenbergstraße 1, 85748 Garching, Germany.

module `wofz.py` loads *libcerf*, the library these tables are generated for, through `ctypes`: from the path in the environment variable `LIBCERF_PATH` if that is set, otherwise from the system library search. It records the file it loaded as the attribute `my_impl_library`, which the two modes name in the headers of their output, so that measurements against different builds of the library remain distinguishable. If the library is nowhere to be found, `my_impl_callable` stays `None` and the two modes have nothing to measure: mode `e` says so and does nothing, mode `H` stops with an error message.

2.2 Install

The software can be installed as a Python package with:

```
pip install ttapp
```

This automatically installs the dependencies and provides the command:

```
ttapp <arguments>
```

To uninstall, simply do

```
pip uninstall ttapp
```

Alternatively, when working from source call the installation command

```
pip install -e .
```

from the project root directory; it will use metadata from `pyproject.toml`. Run and uninstall commands are the same as before.

When working from source, one can also run `ttapp` without installing. In this case, call the module main program `ttapp/ttapp.py` from the project root:

```
python3 -m ttapp <mode> <arguments>
```

Running `ttapp` without arguments prints a summary of available modes.

2.3 Function module

To make productive use of `ttapp`, one needs to provide an application-specific function definition file.

The function to be approximated is provided through a dedicated Python module, which specifies the interface between the generic approximation machinery and the target function $f(z)$. It exports two functions:

```
def feval(x: float, y: float) -> complex:
    """High-precision reference value of f(z) at z = x + iy."""
    ...

def fn_vector(x: float, y: float, N: int = 53) -> list[Coeff]:
    """Return N Taylor coefficients with rounding error estimates."""
    ...
```

Each element of the returned list is a `Coeff` dataclass with fields:

```
@dataclass
class Coeff:
    fn:          complex #double-precision value of f^(n)(z)/n!
    abs:         float  #|f^(n)(z)/n!|
    rel_rounding_err: float #rounding error relative to machine epsilon
```

In the provided implementation, `feval` uses *python-flint* (Arb) [3] for a high-precision (103-bit) evaluation, while `fn_vector` uses *python-flint* at 343-bit precision to compute Taylor coefficients before rounding them to double precision and tracking the introduced error.

As an example, the package includes `demo_function/wofz.py` that implements the Faddeeva function

$$w(z) = \exp(-z^2) \operatorname{erfc}(-iz) \quad (1)$$

introduced in [1, Sect. 1.3]. The arbitrary-precision computation of $w(z)$ is straightforward because Arb supports the Faddeeva function through its built-in method `acb_hypgeom_w`. For the current application, the Taylor expansion tiling is focused on the domain $|z| < 7$, where the function's behavior requires piecewise approximation. The accuracy of the high-precision reference is not a concern due to the intrinsic error tracking and interval arithmetic of the Arb library. To support any other complex function $f(z)$, one needs to write a new module exposing `feval` and `fn_vector`, following the model provided by `demo_function/wofz.py`, and name it on the command line, after the mode letter. However, in contrast to the companion software *ppapp* for real functions of a real argument [7], *ttapp* is not yet fully generic: its function-independent code relies on specific properties of $w(z)$, in particular on the monotonic decrease of $|w(z)|$ in the first quadrant, which is used to bound the relative error [1, Sect. 1.5]. Applying *ttapp* to another function $f(z)$ therefore requires some adaptation of that code. Mode `t` additionally requires a function `guarantee_at(x, y)` that returns the theoretical relative error bound of the target implementation at $z = x + iy$, in units of ϵ . Further optional attributes of the module, specified in `ttapp/function_interface.py`:

- my_domain** a dictionary with the outer radius R , the Maclaurin radius r_{mac} , the lattice spacing a , and the symmetry flags x_{sym} , y_{sym} ; modes `H` and `t` take their radii from it.
- my_testcases** a list of tuples $(x, y, f(x + iy), \text{tolerance})$ for self-validation of the module.
- my_impl_callable** the implementation under test, a callable $(x, y) \mapsto f(x + iy)$ in double precision, measured by modes `e` and `H`.
- my_impl_library** the file that implementation was loaded from; the two modes name it in their output headers.

2.4 Run modes

ttapp operates in different modes, selected by a letter as the first command-line argument and followed by the name of the function module:

```
ttapp <mode> <func_module> <arguments>
```

Called with a mode but no further argument, or with an argument list that does not fit the mode, *ttapp* prints what that mode expects. Arguments that several modes have in common keep one order throughout: `N_Taylor`, `delta`, `d2`, `inverse_a`, `M_offsets` come first, in that order, then the arguments specific to the mode, and `n_workers` last wherever a mode is parallelized. The two primary modes, `d` and `c`, form the main pipeline and are run in sequence to produce the C source files. Mode `t` generates C test code for the same domain. The remaining modes are diagnostic utilities. Unless stated otherwise, output is written to `stdout`; use redirection to save it in a file. Modes that write files write them into the current working directory, under names formed from the name of the function module, and create no subdirectory. Run *ttapp* out of source, therefore, from the directory that is to hold the results: the products of a `d` run and of the `c` runs based on it then lie side by side.

Mode `d`: Compute expansion center candidates.

```
ttapp d <func_module> <N_Taylor> <delta> <inverse_a> <M_offsets> <n_workers>
```

For each grid point in the domain, determines the maximal squared diameter d^2 of the covering ball and the optimal expansion center.

Parameters.

N_Taylor Integer $N \geq 1$. Upper bound on the Taylor truncation order. The script searches for the minimal $N_k \leq N$ that satisfies the error bound at each grid point; N must be large enough that such an N_k exists everywhere.

delta Float $\delta > 0$. Prescribed upper bound on the relative error, in units of machine epsilon $\epsilon = 2^{-53}$. A value of 3 corresponds to about 3.3×10^{-16} .

inverse_a Float $1/a > 0$. Inverse of the tile side length a . Larger values give smaller tiles and more expansion centers, reducing the required truncation order at the cost of more storage.

M_offsets Integer $M \geq 0$. Number of neighboring floating-point representable centers to consider when recentering an expansion point. Recentering adjusts the nominal lattice point by up to M ULPs in each coordinate to minimize truncation and rounding error. Lattice points on a coordinate axis are clamped: they keep their vanishing coordinate exactly and are only shifted along the axis. Typical values are 7 (for quick tests) and 63 (for final production runs); $M = 0$ disables recentering. The cost grows as $(2M + 1)^2$ evaluations of the coefficient vector per grid point.

n_workers Integer ≥ 0 . Size of the process pool over which the grid points are distributed; 0 asks for one worker per core. The result does not depend on it.

Output. The candidates are written to the file `<func_module>_candidates.dat` in the current working directory (e.g. `wofz_candidates.dat`). The file starts with a header recording the `ttapp` version (the git commit when run from a checkout), the timestamp, the full invocation command, and all parameters:

```
# Created by ttapp <version> at <date and time>:
# ttapp d <func_module> <N_Taylor> <delta> <inverse_a> <M_offsets> <n_workers>
# N_Taylor = <N_Taylor>
# delta = <delta>
# M_recenter = <M_offsets>
# 1/a = <inverse_a>
```

This header is consumed by mode `c` and propagated into every generated C source file. Following the header, the output contains one block per x -column of the grid. Each block starts with the x value, followed by one line per grid point (x, y) :

```
y d2 cx cy
```

where $d2$ is the squared diameter of the covering ball, and cx , cy are the coordinates of the (possibly recentered) expansion center in hexadecimal float format. The computation is parallelized using `multiprocessing.Pool`.

Mode c: Cover algorithm.

```
ttapp c <func_module> <candidate_file> <algo> <timeLimit> <n_workers> # p, s
ttapp c <func_module> <candidate_file> <algo> # g, c, w
```

Reads the output of mode `d`, solves the covering problem to select expansion centers, computes per-tile truncation orders, and writes four output files.

Parameters.

candidate_file Path to the output file produced by mode `d`. The parameters `N_Taylor`, `delta`, and `inverse_a` are read from the file header and do not need to be repeated.

algo Letter. Selects the covering algorithm:

p — **linear programming** Solve the set cover problem exactly, as an integer linear program, using the HiGHS solver through `scipy.optimize.milp`. Before the solver is invoked, the problem is reduced by iterated dominance elimination (a standard set cover presolve, which removes tiles and candidate centers whose covering constraints are implied by others); this preserves the optimum and typically accelerates the solver considerably. Guarantees the minimum number of expansion centers, subject to a wall-clock limit (`timeLimit`). Prints the solve time and the solver status; if optimality is not proven within the limit, the best cover found is used and the remaining MIP gap is reported.

s — **CP-SAT** Solve the same problem, after the same dominance presolve, with the CP-SAT portfolio of Google OR-tools: one Boolean clause per tile, minimizing the number of selected centers. Requires the `orTools` package. Like **p** it is an anytime solver: on timeout the best cover found is used and the proven lower bound is reported. Its portfolio runs on `n_workers` threads. Every improvement the solver finds on its way is written out at once, under the ordinary file names. A smaller cover brings new files, the superseded ones moving to `<name>.bak`; an interrupted run thus leaves its best cover behind, and the one before it. A higher lower bound leaves the tables as they are and only restates the proven interval in their header, which is the sole progress a long run makes once it has found the optimal cover but not yet proven it optimal.

g — **greedy** At each step, select the expansion center covering the largest number of currently uncovered tiles. Tends to minimize the total number of centers.

c — **constraint** Iterate over tiles in order of increasing distance from the reference point $-i$ and, for each uncovered tile, choose the center covering the most additional uncovered tiles. May produce fewer centers near the origin at the cost of more centers overall.

w — **distance-weighted** At each step, select the center maximizing a weighted count of newly covered tiles, the weight decreasing exponentially with the distance from a reference point $-i$. Favors compact covers near the origin.

timeLimit Wall-clock limit, in seconds, granted to the solver; `0` grants it as much time as it needs to prove the optimum.

n_workers Integer ≥ 0 . Threads of the CP-SAT portfolio; `0` asks for one per core. Algorithm **p** accepts the argument but has no use for it, HiGHS being called single-threaded.

The last two arguments are required by the exact algorithms **p** and **s**, which alone can run indefinitely and in parallel, and refused by the three heuristics, which can make use of neither extra time nor extra cores.

Verifications. Before any file is written, two properties of the result are asserted, and mode **c** fails with a diagnostic if either is violated. First, every tile must stay within the error budget δ at the full truncation order N , not merely at some reduced order. Second, every tile of the grid must carry an expansion center, since the target code dispatches to all of them; a tile that no candidate ball can cover calls for a larger N or a smaller a . Neither can fail if the candidates of mode **d** are consistent with the cover, so a failure indicates a defect rather than an unlucky parameter choice — except for the second check, which legitimately rejects a lattice too coarse for the target code, $a > 0.23/\sqrt{2}$, where the Maclaurin disk no longer covers the tile at the origin.

Output. Prints progress to stdout: the number of initially uncovered tiles, the number of available expansion centers, and the total number of centers selected. For p and s, additionally the dimensions of the integer linear program, its wall-clock solve time, and the solver status are printed. The four output files are prefixed with the function module name (e.g. `auto_taylor_wofz_*`; below generically `auto_taylor_f_*`) and are written to the current working directory. Whenever a run writes a second, differently sized cover, as algorithm s does, the files of its predecessor are moved to `<name>.bak` rather than overwritten; a rewrite that only restates the bound on the optimum replaces the files it belongs to.

File `auto_taylor_f_centers.tab` A table of the selected expansion centers. Each line contains the integer grid indices i_x i_y of a center on the lattice $(a/2) \mathbb{Z}^2$ of the symmetry points of the tiling, at position $(i_x, i_y) a/2$.

File `auto_taylor_f_Nk.tab` A table of the per-tile truncation order N_k . Laid out as a matrix indexed by (j_x, j_y) ; entries -2 denote grid cells outside the computational domain, entries -1 tiles that carry no Taylor expansion.

Files `auto_taylor_f_tiles.c`, `auto_taylor_f_coeffs.c` Two C source files ready for inclusion in a target project. Both files open with a self-documenting block header, which names the generator, the generation timestamp, the full commands used for both the c and d modes, the parameters, what the cover cost and is worth, and the usage of the definitions in the file:

```
// --- Begin of auto-generated code; do not edit ---
// clang-format off
//
// Generated
//   on <date>, <time>
//   by the tilewise Taylor approximation generator ttapp
//   ...
//
// Generator called as:
//   ttapp c <func_module> <candidate_file> <algo> <timeLimit> <n_workers>
// using candidates from:
//   ttapp d <func_module> <N_Taylor> <delta> <inverse_a> <M_offsets> <n_workers>
// Parameters:
//   N_Taylor = <N_Taylor> # Taylor truncation order
//   ...
// Cover:
//   algorithm <algo>: <its name>
//   <n> centers (polyominoes) in <t> s
//   proven bound in [<lower bound>, <n>]
//
// Usage within libcerf:
//   ...
```

The C code follows after a blank line, and the file closes with `// clang-format on` and an end marker. The two `.tab` files instead begin with the provenance stanza `# Created by ttapp <version> at <date and time>:` and the `ttapp` command, followed by the same command and cover lines. The time is the wall clock of the cover computation, the search for the centers with its presolve, not of the whole run. The last line appears for the exact algorithms only, and states where the unknown optimum has been narrowed down to: from the lower bound the solver has proven up to the cover it has found. Equal ends mean that the cover is proven minimal. The file `auto_taylor_f_tiles.c` then defines typed constants:

```
static const int   NTay      = <N>;    // Taylor truncation order
static const double delta    = <d>;    // relative error tolerance (units of eps = 2^-53)
static const double inverseA = <1/a>;  // 1/a, inverse of lattice constant a
static const int   nXcover   = <n>;    // number of tiles per axis
```

auto_taylor_f_tiles.c additionally defines:

```
alignas(64) static const signed char Tiles[2 * nXcover * nXcover];
```

This array stores both the cover index and the per-tile truncation order N_k for each tile, interleaved as `Tiles[2*(jx*nXcover+jy)]` and `Tiles[2*(jx*nXcover+jy)+1]`.

auto_taylor_f_coeffs.c carries only a comment referring to `auto_taylor_f_tiles.c` for parameters, and additionally defines:

```
alignas(64) static const double TaylorCoeffs[2 * (NTay + 1) * nCenters];
```

where `nCenters` is the number of selected expansion centers, encoded as a concrete integer in the array dimension. For each expansion center, the array holds the real and imaginary parts of the center coordinates followed by the real and imaginary parts of the Taylor coefficients $f^{(n)}(c_k)/n!$ for $n = 0, 1, \dots, N-1$, all in hexadecimal float format.

Mode v: Evaluate f at one point.

```
ttapp v <func_module> <N_Taylor> <d2> <inverse_a> <x> <y>
```

Computes and prints the truncation error and rounding error of the Taylor approximation of $f(z)$ at the point $z = x+iy$, for given truncation order N , squared ball diameter d^2 , and lattice constant a .

Output format:

```
te=<truncation error> re=<rounding error>
```

Both errors are in units of machine epsilon. Useful for diagnosing the error at a specific point without running the full pipeline.

Mode e: Worst relative error per tile.

```
ttapp e <func_module> <inverse_a> <n_per_tile> <n_workers>
```

For each tile in the region $0 \leq x < 10$, $-10 \leq y < 10$, draws `n_per_tile` random points uniformly distributed in the tile and evaluates the relative error of the implementation under test against the high-precision reference. Records the worst-case error and the corresponding point. The x range is halved by the symmetry $f(-\bar{z}) = \overline{f(z)}$, whereas the y range covers both half planes, because an implementation typically reaches the lower one through a reflection formula and thus by a code path of its own. Tiles are distributed over a pool of `n_workers` processes, one per core if the argument is 0; each tile is seeded from its own indices, so that the result does not depend on that number. Output is written to `<func_module>_worst_per_tile.tab` in the current working directory; its header names the library measured, besides the provenance stanza and the run parameters. This mode validates that the double-precision implementation of $f(z)$ meets its accuracy specification, as a prerequisite for the statistical analysis in mode H.

Mode H: Error histogram.

```
ttapp H <func_module> <region> <range> <nchannels> <ndraws> <n_workers>
```

Draws a large number of random points in a specified region of the complex plane, evaluates the relative error of an $f(z)$ implementation against the high-precision reference, and produces a histogram. The computation is parallelized over multiple worker processes.

Parameters:

region Integer. Selects the sampling region, in the first quadrant throughout: 1 the Maclaurin disc $|z| < r_{\text{mac}}$, 2 the Taylor annulus $r_{\text{mac}} < |z| < R$, 3 the asymptotic annulus $R < |z| < 2R$, with the radii from the function module's `my_domain`. The output header spells the region out.

range Float. Histogram range in units of machine epsilon. Points with relative error exceeding $\text{range} \epsilon$ are counted in the overflow channel and flagged.

nchannels Integer. Number of histogram channels.

ndraws Float (interpreted as integer after division by the number of workers). Total number of random sample points.

n_workers Integer ≥ 0 . Size of the process pool; 0 asks for one worker per core. Each worker draws the same number of points from its own seeded generator, so this number is part of what the result depends on; the header of the output reports it, together with the total that was actually drawn.

Output. Prints the histogram to stdout, one line per channel:

```
<center in units of epsilon> <count>
```

A warning is printed if the overflow channel is non-empty. The data are preceded by a comment header that carries the provenance stanza, the library measured, the run parameters, and a description of the two columns, so that a redirected stream documents itself. In that header the region index is resolved into explicit geometry, and the number of draws is the number actually drawn, which falls short of `ndraws` whenever the per-worker count has to be truncated.

Mode w: Worst minimum $|f|$ at tile.

```
ttapp w <func_module> <d2> <inverse_a> <ix> <iy>
```

Computes and prints $f_{\min}$, the minimum of $|f(z)|$ over the lattice neighbors within squared distance d^2 of the half-integer lattice point (i_x, i_y) . This quantity appears in the denominator of the relative error bound. A diagnostic utility for inspecting individual $f_{\min}$ values without running the full pipeline.

Mode t: Generate test C source code.

```
ttapp t <func_module> <inverse_a>
```

Writes `auto_test_taylor_<func_module>.c` to the current working directory. The file opens with a comment block that names the generator, the command line, the parameters, and the usage of the table and the function it defines, and it closes with an end marker; the C code in between is fenced by `clang-format off/on`. The grid consists of the tiles that intersect the domain $r_{\text{mac}} \leq |z| < R$; both radii are taken from the function module's `my_domain` (defaults 7 and 0, the latter meaning no Maclaurin hole, so that the small- $|z|$ code paths are exercised). Each tile contributes up to 20 test points: the tile center; the four corners; four points a random 1 to 16 ulps inward of the corners, per coordinate; one random point on each edge; and seven random points inside the tile; points outside the domain are omitted. Reference values come from the high-precision function, correctly rounded to double, and are written as hex floats (Sec. 2.5). All random generators are seeded from the tile indices, so that a rerun reproduces the file except for the timestamp in its header. Each test point carries its own error tolerance: the bound `guarantee_at(x, y)` on the

true relative error provided by the function module, plus the exact relative error of rounding the reference to double (the test compares against the rounded reference, so its rounding error adds to the measured deviation), rounded up to two decimals. For each function module f , the file defines the function `run_tests_f()`, which the embedding test program must complement by `test_one_f()` and `main()`; the suffix lets several such files share one test program, see file `test/taylor_test.c` in `libcerf` for an example.

Mode u: Run unit tests.

```
ttapp u
```

Runs the Python unit tests; equivalent to the command given in Sec. 2.6.

2.5 Hexadecimal output

Floating-point numbers in the auto-generated C source files are written in hexadecimal format, like

```
0x1.ef90904c7eeep-2, -0x1.9d5e2f887fe99p-15, ...
```

The letter p is followed by the base-2 exponent in decimal notation: `0x1.8p-13` represents $1.5 \cdot 2^{-13}$.

2.6 Unit tests

The test suite ships with the package, in `ttapp/tests`. From the repository root, the tests can be run with

```
python3 -m pytest
```

From anywhere, also for an installed package, `ttapp u` runs the same tests through `unittest` discovery. The test suite covers the Taylor coefficients and their error bounds, the sorted diameters, parity classes and ULP stepping of mode d, the exact and heuristic cover algorithms with their presolve and the command-line checks of mode c, the generated C files of mode c against reference files, the test-point recipe and file layout of mode t, and the header of mode H, including the record of the library measured.

3 Use in libcerf

The generated files are used in the library *libcerf*: `lib/w_of_z.c` includes the two tables `auto_taylor_wofz_tiles.c` and `auto_taylor_wofz_coeffs.c`, `test/taylor_test.c` runs the test cases of `test/auto_test_taylor_wofz.c`, and the executable `run_wofz`, built from `run/run_wofz.c`, evaluates $w(z)$ from the command line.

3.1 Alignment specifier

As in the companion software for piecewise Chebyshev approximation [7], the auto-generated arrays are declared with

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

to align them on 64-byte (cache-line) boundaries and minimize cache loads. The specifier `alignas` is defined in the language standards C23 and C++11. For C11 use `_Alignas`; for older variants, use compiler-specific attributes.

References

- [1] Wuttke J *Code generation for computing an analytical function with near machine precision on square tiles, with application to the Faddeeva function*. in preparation.
- [2] Wuttke J (2026) *ttapp*. Tile-wise Taylor approximation generator. <https://jugit.fz-juelich.de/mlz/app/ttapp>. Zenodo <https://doi.org/10.5281/zenodo.22277192>.
- [3] *Python Package Index*. Project *python-flint*. <https://pypi.org/project/python-flint>.

- [4] *FLINT: Fast Library for Number Theory*. <https://flintlib.org> (2024). Version 3.1.2.
- [5] *SciPy: Fundamental algorithms for scientific computing in Python* (2008–).
- [6] Perron L and Furnon V (2026) *OR-Tools: the CP-SAT constraint solver (version 9.15)*. <https://developers.google.com/optimization>.
- [7] Wuttke J and Kleinsorge A (2026) *Algorithm 1062: Code Generation for Piecewise Chebyshev Approximation*. *ACM T. Math. Software* **52**.