# ttapp User Manual: Python code for generating C code for tilewise Taylor approximation

Miriam Moreno Sierra, Joachim Wuttke✉

Forschungszentrum Jülich GmbH, Jülich Centre for Neutron Science at MLZ,
Lichtenbergstraße 1, 85748 Garching, Germany

✉ j.wuttke@fz-juelich.de

Version 1.2.1, 18oct26

**Abstract**

This manual documents the Python software *ttapp* that supports the tilewise Taylor approximation of a given analytic function $f(z)$ in the complex domain, as described in the article "Tilewise Taylor approximation of an analytic function with near machine precision" [13].

## 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. This manual explains how to use package *ttapp*, and provides further details, especially on software engineering aspects of the code-generating code and the target code, thereby complementing the foundational paper [13].

The software is released under the GNU General Public License Version 3 or higher; other licensing is negotiable. The software package is available through three distribution channels:

- the **project repository** at the GitLab instance of Forschungszentrum Jülich [12];
- the **PyPI package** `ttapp` [2], installable via `pip install ttapp`;
- the **archival zip package** deposited in Zenodo [12].

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 explains how to install *ttapp* and how to supply the function to be approximated. Section 3 explains how to run the generator and its diagnostic modes. Section 4 maps the algorithms of the paper to the code, and describes the elementary set cover algorithms that mode `c` offers as alternatives to the exact solvers. Section 5 describes the generated C files and their use in *libcerf*.

## 2 Getting started

### 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`** [1] binds the C library FLINT [4], whose ball arithmetic [7] 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`** [3] (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`** [9] 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 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` aborts with an error.

## 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, run the package as a module 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 at least 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 representation 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_repr_err:       float #representation error, relative to machine epsilon
```

The generator calls `fn_vector` with $N$ up to 45 beyond the number of Taylor terms kept; the extra terms serve the truncation error bound. Of the series tail, 40 terms are summed explicitly, and the remainder is bounded by the optional function

```
def tail_bound(x: float, y: float, Np: int, tau: float,
               WN: list[Coeff]) -> float:
    """Bound on the Taylor tail sum_{n>=Np} |f_n(x+iy)| tau^n."""
    ...
```

which `wofz.py` derives from a recurrence inequality for the Taylor coefficients of $w$ [13, App. A.4]. Without `tail_bound`, the generator falls back to a heuristic estimate: ten explicit terms, accepted if the next ten add less than 0.1 %.

In the provided implementation, `feval` uses *python-flint* (Arb) [1] 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 `ttapp/demo_function/wofz.py` that implements the Faddeeva function

$$w(z) = \exp(-z^2)\,\mathrm{erfc}(-iz) \tag{1}$$

introduced in [13, Sect. 1.3]. The arbitrary-precision computation of $w(z)$ is straightforward: `feval_acb` evaluates the right-hand side in ball arithmetic, and `fn_vector` obtains the coefficients $w_n = w^{(n)}(z)/n!$ from the recurrence $w_n = -\frac{2}{n}\,(z\,w_{n-1} + w_{n-2})$ for $n \geq 2$, seeded by $w_0 = w(z)$ and $w_1 = -2zw_0 + 2i/\sqrt{\pi}$, which follows from the differential equation of $w$. The domain `my_domain` is the first quadrant up to $|z| = 7$, less the Maclaurin disc $|z|^2 < 0.042$ that the target code serves by a series at the origin; beyond $|z| = 7$, *libcerf* turns to an asymptotic expansion. Candidate expansion centers are sought in the extended domain $\mathcal{D}^+$, the first quadrant including the bounding axes, again up to $|z| < 7$. The accuracy of the high-precision reference is not a concern, since the ball arithmetic of Arb tracks all errors. 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 `ttapp/demo_function/wofz.py`, and name it on the command line, after the mode letter. The package ships three more such modules: `exp_neg_z2.py` for $\exp(-z^2)$, and `cerf.py`, `cerfc.py` for the complex error functions, which serve mode `t` only. However, in contrast to the companion software *ppapp* [10] for real functions of a real argument [11], *ttapp* is not yet fully generic [13, Sect. 1.4]: its function-independent code relies on specific properties of $w(z)$, in particular on the monotonic decrease of $|w(z)|$ in the first quadrant [13, App. A.3], which is used to bound the relative error. Applying *ttapp* to another function $f(z)$ therefore requires some adaptation of that code.

Further attributes of the module, specified in `ttapp/function_interface.py`:

**`feval_acb(x, y)`** $f(x + iy)$ in the ball arithmetic of *python-flint*. Mode `t` takes its reference values from it, and modes `e` and `H` use it, where present, to resolve errors below one $\epsilon$.

**`fn_base(x, y, N)`, `fn_vector_from(base, x, y)`** a faster route to the coefficient vectors of mode `d`: `fn_base` precomputes what is needed at a lattice point, and `fn_vector_from`

returns `fn_vector(x, y, N)` for each of the recentred centers near it. `wofz.py` obtains $w$ at such a center by a Taylor shift from the high-precision coefficients at the lattice point, with a rigorous bound on the omitted terms, instead of evaluating erfc anew; this makes mode `d` about 1.3 times faster, with identical output. In addition, `wofz.py` computes the representation error of a coefficient only when it is read, which the rounding bound does for the first $N$ coefficients alone; together, mode `d` runs about 1.7 times faster than before.

**`guarantee_at(x, y)`** the theoretical relative error bound of the target implementation at $z = x + iy$, in units of $\epsilon$; required by mode `t`.

**`my_domain`** a dictionary with the outer radius `R`, the Maclaurin radius `r_mac`, and the radius `R_centers` of the disc that holds the candidate expansion centers, which may exceed `R` (the extended domain $\mathcal{D}^+$ of Ref. [13]); modes `d`, `c`, `t` take their domain from it, mode `H` its radii. Missing keys default to $R = 7$ and `R_centers` $= R$; `r_mac` defaults to 0 in modes `d`, `c`, `t` and to 0.23 in mode `H`.

**`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.

# 3 Usage

*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`, `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.

## 3.1 Mode d: Generate expansion center candidates

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

For each lattice point of the extended domain $\mathcal{D}^+$, i.e. each symmetry point $(i_x, i_y)\, a/2$ of the tiling with $i_x, i_y \geq 0$ and $|z| <$ `R_centers`, determines the maximal squared diameter $d^2$ of the covering ball and the optimal expansion center. The error bound of a ball is divided by the minimum of $|f|$ over those of its tiles that belong to the domain; tiles outside the domain, which need no coverage, do not constrain the ball.

*Parameters.*

**`N_Taylor`** Integer $N \geq 1$. Number of Taylor terms at which the balls are bounded; mode `c` later reduces it per tile to the minimal $N_k \leq N$ that still meets the tolerance. $N$ must be large enough that every tile fits into some ball.

`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 $3.3 \times 10^{-16}$.

`a` Tile edge, to be given as fraction `p/q` where $q$ is a power of 2; it is reduced to lowest terms. Then every tile corner and every lattice point is an exact double, and the target code finds the tile column of $x \geq 0$ exactly, as $\lfloor \lfloor qx \rfloor / p \rfloor$. The target code needs $\sqrt{2}\, a < r_{\mathrm{mac}}$, so that the Maclaurin disc covers the tile at the origin; for *libcerf*, `37/256`. Smaller values give smaller tiles and more expansion centers, reducing the required number of Taylor terms 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 $\geq 0$. Size of the process pool over which the lattice points are distributed; `0` asks for one worker per core. The result does not depend on it.

*Progress.* The lattice points are handed to the workers in a shuffled order, fixed by a seed. After every percent of them, a line on `stderr` reports the fraction done, the elapsed time, the remaining time, and the expected time of completion. The estimate extrapolates linearly in the fraction done; the shuffling makes it unbiased, whereas in column order the cost per point, which differs between axis points and others and depends on $|z|$, would bias it. Its relative error shrinks roughly as one over the square root of the number of points done.

*Output.* The candidates are written to the file `<func_module>_candidates.dat` (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> <a> <M_offsets> <n_workers>
# N_Taylor = <N_Taylor>
# delta = <delta>
# M_recenter = <M_offsets>
# a = <a>
# R_centers = <R_centers>
```

followed by a note on the data format. This header is consumed by mode `c` and propagated into every generated C source file. Following the header, the output contains one block per lattice column. Each block starts with the $x$ value, followed by one line per lattice point $(x, y)$:

```
y d2 cx cy
```

where `d2` is the squared diameter of the covering ball in units of $a^2$, the quantity $\kappa$ of [13, Eq. (4.10)], or 0 where no ball meets the tolerance, and `cx`, `cy` are the coordinates of the (possibly recentered) expansion center in hexadecimal float format (Section 5.2). The computation is parallelized using `multiprocessing.Pool`.

### 3.2 Mode c: Compute cover, generate target code

```
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 indices, and writes four output files.

*Parameters.*

`candidate_file` Path to the output file produced by mode `d`. The parameters `N_Taylor`, `delta`, and `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`. HiGHS takes the covering conditions as linear inequality rows and searches by branch-and-cut over the linear-programming relaxation. 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. CP-SAT searches by lazy clause generation with conflict learning, supported by relaxation-based workers that run in parallel. 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 $z_o = -i$ 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 the reference point $z_o$. Favors compact covers near the origin.

The three heuristics are discussed in Section 4.2.

`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 $\geq 0$. 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.

*Axis reservation.* Whatever the algorithm, tiles in the narrow wedge $j_y \geq 17.5\, j_x$ along the imaginary axis are removed from the ranges of all centers off that axis, as long as some center on the axis ($i_x = 0$) covers them; such centers preserve the symmetry of $w$ on the axis [13, App. A.5]. Likewise, a tile that lies in several selected balls is assigned to a center on the

imaginary axis if there is one, otherwise to the center closest to the tile's farthest corner, which is the radius that enters the tile's error bound [13, Alg. 4].

*Verifications.* Before any file is written, two properties of the result are checked, and mode `c` stops with a diagnostic if either is violated. First, every tile must stay within the error budget $\delta$ with all $N$ Taylor terms, not merely with a reduced number. This cannot fail if the candidates of mode `d` are consistent with the cover, so a failure indicates a defect. 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 $a$.

*Output.* Prints progress to `stdout`: the parameters $a$ and $\delta$ read from the candidates file, 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 effect of the presolve, the dimensions of the integer linear program, its wall-clock solve time, and the solver status are printed. While CP-SAT runs, a line on `stderr` every minute gives the elapsed time, the best cover and the proven bound so far, and the time at which the time limit ends the run; the run ends earlier if it proves the cover optimal, which cannot be foreseen. The four output files are named after the function module (e.g. `auto_taylor_wofz_*`; below generically `auto_taylor_f_*`). Algorithm `s` may write them several times, the superseded files moving to `<name>.bak` as described above.

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

**File `auto_taylor_f_Nk.tab`** A table of the per-tile truncation index $N_k$. Laid out as a matrix with one line per tile column $j_x$, the entries running over the tile rows $j_y$ from the lowest row of the domain; entries $-2$ denote grid cells outside the computational domain, entries $-1$ tiles that carry no Taylor expansion, which the second verification above rules out.

**Files `auto_taylor_f_tiles.c`, `auto_taylor_f_coeffs.c`** Two C source files ready for inclusion in a target project; their layout is described in Section 5.1.

### 3.3 Mode v: Print error budget at one point

```
ttapp v <func_module> <N_Taylor> <d2> <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 a given number $N$ of Taylor terms, squared ball diameter $d^2$, and tile edge length $a$.

*Output.* After a few header lines that restate the arguments:

```
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.

### 3.4 Mode e: Generate table of worst relative errors per tile

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

For each tile in the region $0 \le x < X$, $-X \le y < X$, with $X = \lfloor 10/a \rfloor\, a \approx 10$, 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, nor on the shuffled order in which tiles are handed out; after every 5 % of them, a line on `stderr` reports progress and the expected time of completion, as in mode `d`. Output is written to `<func_module>_worst_per_tile.tab`; 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`.

### 3.5 Mode H: Print error histogram

```
ttapp H <func_module> <region> <range> <nbins> <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 $|z| < r_{\mathrm{mac}}$, `2` the Taylor annulus $r_{\mathrm{mac}} < |z| < R$, `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 `range` $\epsilon$ are counted in the last bin, which thus doubles as overflow bin; points with a non-finite relative error are discarded.

**`nbins`** Integer. Number of histogram bins.

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

**`n_workers`** Integer $\geq 0$. 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 bin:

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

A warning is printed if the last bin 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.

### 3.6 Mode w: Print minimum $|f|$ around a lattice point

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

Computes and prints, as `wm=<value>` after a few header lines, the minimum of $|f(z)|$ over those tiles of the domain that lie within the ball of squared diameter $d^2$ about the lattice point $(i_x, i_y)$, each represented by its upper right corner, where $|w|$ is smallest in the first quadrant, as it decreases monotonically. This quantity appears in the denominator of the relative error bound of mode `d`. A diagnostic utility for inspecting individual values without running the full pipeline.

### 3.7 Mode t: Generate target test code

```
ttapp t <func_module> <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, the usage of the table and the function it defines, and the recipe and count of the test cases, 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 $r_{\mathrm{mac}} \le |z| < R$; 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 are the midpoints of the balls returned by `feval_acb`, rounded to double, and are written as hex floats (Section 5.2). 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/taylortest.c` in *libcerf* for an example.

### 3.8 Mode u: Run unit tests

```
ttapp u
```

Runs the Python unit tests shipped in `ttapp/tests` from anywhere, also for an installed package, through `unittest` discovery. From the repository root, the tests can also be run with

```
python3 -m pytest
```

The test suite covers the Taylor coefficients and their error bounds, the sorted diameters, parity classes and ULP stepping of mode `d`, the exact cover algorithms with their presolve and the command-line checks of mode `c`, the generated C files of the greedy cover 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.

## 4 Code generation details

### 4.1 Pipeline and modules

Code generation is a pipeline of two stages, run as the modes `d` and `c` of Section 3. Stage `d` works through the lattice of possible expansion centers and determines, for each of them, the largest ball within which the Taylor series meets the tolerance, and a recentered position that minimizes the rounding error. This is the costly stage: every candidate requires high-precision Taylor coefficients from the function module. Its result, the candidates file, is the interface to stage `c`, which selects a cover from the candidates, assigns each tile to one expansion, computes the truncation index per tile, and writes the C files. The cover algorithm and its time limit can thus be varied without recomputing the candidates.

Table 1 maps the algorithms of the paper [13] to the modules of the Python package. The diagnostic modes `v`, `w`, `e`, and `H` live in modules of their own: `runat.py`, `wminat.py`, `worst_per_tile.py`, and `mc_histogram.py`.

Table 1: Where the algorithms of the paper [13] are implemented. Names are `module.function`; nested functions are named after a colon.

| Paper | *ttapp* |
|---|---|
| Eq. (4.10), sequence of squared diameters | `bgrid_to_candidates.sorted_diameters` |
| Alg. 1, expansion centers and diameters (mode `d`) | `bgrid_to_candidates.process_point` |
| error bound $\eta$, Sect. 3 | `terms.truncation_error`, `terms.rounding_error` |
| minimum modulus in the denominator of $\eta$ | `terms.wmin` |
| Alg. 2, polyomino shapes | `grid.polyomino_pattern` |
| Alg. 3, set of ranges | `cover.initialize_ranges`, `grid.Grid.polyomino` |
| set cover problem, Sect. 4.5 | `cover.eliminate_dominated`, `cover.solve_exact`, `cover.solve_exact_cpsat` |
| elementary cover algorithms, Section 4.2 of this manual | inline in `cover.main` |
| Alg. 4, subdomains and truncation indices | `cover.main:assign`, `cover.main:truncation_orders` |
| Alg. 5, top level | `ttapp.main`, dispatching on the mode letter |
| headers of all generated files | `provenance` |
| test cases (mode `t`) | `testcases.main` |

## 4.2 Elementary set cover algorithms

The exact cover algorithms `p` and `s` of mode `c` solve the set cover problem by mixed-integer linear programming (MILP), as described in the paper [13]. For much larger problem instances, the MILP approach could prove impracticable. For such cases, *ttapp* also implements the following elementary algorithms, which construct reasonably small covers within fractions of a second. Being deterministic, they moreover yield identical covers across platforms and software versions, which we exploit in regression tests. Algorithm A1, the greedy algorithm, applies to any set cover problem; Algorithms A2 and A3 exploit the planar geometry of the range space.

We use the notation of the paper [13]. $\mathsf{T}$ is the set of tiles that cover the domain $\mathcal{D} \subset \mathbb{C}$. $\mathbf{X}$ is the set of candidate ranges; a range $\mathsf{R} \in \mathbf{X}$ is the set of tiles inside the ball of one candidate center, as listed in the candidates file. A cover $\mathbf{G} \subset \mathbf{X}$ is a set of ranges whose union is $\mathsf{T}$. $\mathsf{Y} \subset \mathsf{T}$ denotes the tiles not yet covered. Every tile belongs to at least one range, since mode `c` rejects a grid with a tile that no candidate covers (Section 3).

A decent approximation to the minimal cover is provided by the *greedy algorithm* [5, ch 35.3], summarized in Algorithm A1, that iteratively selects the range that covers the maximum number of yet uncovered tiles. Since every tile belongs to at least one range, each iteration removes at least one tile from $\mathsf{Y}$; therefore the while loops of Algorithms A1–A3 terminate after at most $|\mathsf{T}|$ iterations. The cover found by the greedy algorithm is at most a factor $H(m) \le 1 + \ln m$ larger than the optimal one, where $m := \max_{\mathsf{R} \in \mathbf{X}} |\mathsf{R}|$ is the maximal range size, and $H(m)$ the $m$-th harmonic number [5, thm 35.4]; a coarser bound is $1 + \ln |\mathsf{T}|$ [8, sect 10.1].

The following two variants of the greedy algorithm impose a geometric constraint or preference that enforces a more regular order in the choice of ranges. Greedy variants with modified selection or tie-breaking rules have a long record in set covering practice [6]. The approximation guarantee of the plain greedy algorithm does not carry over to the variants; their justification is purely empirical: for the Faddeeva function on the first-quadrant domain of the paper ($N = 23$, $\delta = 3$, $a = 1/7$, $M = 63$: 1928 tiles, 7623 candidate ranges), they yielded covers of 63 and 60 ranges, against 72 for the plain greedy algorithm, while the exact solver found a cover of 54 ranges.

Both variants give higher priority to tiles $\mathcal{T}$ with smaller distance

$$b(\mathcal{T}, z_\mathrm{o}) := |z_\mathcal{T} - z_\mathrm{o}| \tag{2}$$

**Algorithm A1** Compute covering by greedy algorithm.

**require** $\mathsf{T}, \mathbf{X}$
$\mathbf{G} \leftarrow \{\}$
$\mathsf{Y} \leftarrow \mathsf{T}$ {tiles not yet covered}
**while** $\mathsf{Y} \neq \{\}$ **do**
    $\mathsf{R} \leftarrow \mathsf{R}' \in \mathbf{X}$ that maximizes $|\mathsf{Y} \cap \mathsf{R}'|$
    add $\mathsf{R}$ to $\mathbf{G}$
    $\mathsf{Y} \leftarrow \mathsf{Y} \setminus \mathsf{R}$
**end while**
**return** $\mathbf{G}$

**Algorithm A2** Constraint variant of greedy algorithm.

**require** $\mathsf{T}, \mathbf{X}$; $z_\mathrm{o} \in \mathbb{C}$
$\mathsf{T}' \leftarrow$ list of all $\mathcal{T} \in \mathsf{T}$, sorted by increasing $b(\mathcal{T}, z_\mathrm{o})$
$\mathbf{G} \leftarrow \{\}$
$\mathsf{Y} \leftarrow \mathsf{T}$ {tiles not yet covered}
**while** $\mathsf{Y} \neq \{\}$ **do**
    $\mathcal{T}_0 \leftarrow$ first tile of $\mathsf{T}'$ that is in $\mathsf{Y}$
    $\mathbf{X}' \leftarrow \{\mathsf{R} \in \mathbf{X} \mid \mathcal{T}_0 \in \mathsf{R}\}$
    $\mathsf{R} \leftarrow \mathsf{R}' \in \mathbf{X}'$ that maximizes $|\mathsf{Y} \cap \mathsf{R}'|$
    add $\mathsf{R}$ to $\mathbf{G}$
    $\mathsf{Y} \leftarrow \mathsf{Y} \setminus \mathsf{R}$
**end while**
**return** $\mathbf{G}$

**Algorithm A3** Distance-weighted variant of greedy algorithm.

**require** $\mathsf{T}, \mathbf{X}$; $z_\mathrm{o} \in \mathbb{C}$; $\xi \in \mathbb{R}$
$\beta_\mathcal{T} \leftarrow \exp(-\xi b(\mathcal{T}, z_\mathrm{o}))$ for all $\mathcal{T} \in \mathsf{T}$
$\mathbf{G} \leftarrow \{\}$
$\mathsf{Y} \leftarrow \mathsf{T}$ {tiles not yet covered}
**while** $\mathsf{Y} \neq \{\}$ **do**
    $\mathsf{R} \leftarrow \mathsf{R}' \in \mathbf{X}$ that maximizes $\sum_{\mathcal{T} \in \mathsf{Y} \cap \mathsf{R}'} \beta_\mathcal{T}$
    add $\mathsf{R}$ to $\mathbf{G}$
    $\mathsf{Y} \leftarrow \mathsf{Y} \setminus \mathsf{R}$
**end while**
**return** $\mathbf{G}$

from a reference point $z_o \notin \mathcal{D}$, where $z_\mathcal{T}$ is the lower left corner of tile $\mathcal{T}$, the point $(j_x a, j_y a)$ for the tile with indices $(j_x, j_y)$. In Algorithm A2, the *constraint* variant of the greedy algorithm, in each iteration we only consider ranges that cover the yet uncovered tile with smallest $b$. This greatly reduces the number of candidate ranges, and thereby makes this algorithm particularly fast. It transfers a classical one-dimensional idea to the plane: to cover points on a line by intervals, it is optimal to iteratively select the uncovered point with smallest coordinate and the interval that covers it and reaches farthest.

Algorithm A3, the *distance-weighted* variant of the greedy algorithm, is slower than the constraint variant, but tends to yield the smallest covers. Each tile $\mathcal{T}$ is assigned a weight $\beta_\mathcal{T} := \exp(-\xi b(\mathcal{T}, z_o))$, where $\xi$ is a tunable hyperparameter. In each iteration, we select the range that maximizes the sum of weights of newly covered tiles. Since every tile must be covered in the end, the weights do not change the optimization goal; they only steer the order of the greedy choices. The distance-weighted variant interpolates between the two other algorithms: For $\xi = 0$, all weights are equal, and Algorithm A3 reduces to Algorithm A1. For $\xi \to \infty$, the sum of weights is generically dominated by the term from the uncovered tile with smallest $b$, so that the chosen range is constrained as in Algorithm A2.

The reference point $z_o = -i$ and the hyperparameter $\xi = 5$ are hard-coded in module `cover.py`; the command line of mode `c` takes no argument for them. The ranges **X** handed to the algorithms already reflect the axis reservation described under mode `c` in Section 3.

# 5 Target code details

## 5.1 Generated files

Code generation involves heuristics and hyperparameters, so that one usually goes through several iterations before the target code is correct and satisfactory. To keep this cycle smooth, *ttapp* writes the tables of the target algorithm [13, Sect. 2.1] to source files of their own, separate from the hand-written code, which adds them during compilation, in C or C++ through the `#include` directive. Each file is generated in its entirety; nothing is to be added or edited by hand. It opens with a comment that records how it was generated: when, by which version of *ttapp*, and with which parameters.

Mode `c` (Section 3) writes two C source files for inclusion in the target project, `auto_taylor_f_tiles.c` and `auto_taylor_f_coeffs.c`, where `f` stands for the name of the function module. 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 <version>
//   - Source repository:
//       https://jugit.fz-juelich.de/mlz/app/ttapp
//   - Archived release:
//       https://doi.org/10.5281/zenodo.22277190
//   - Documentation:
//       ...
//
// Generator called as:
//   ttapp c <func_module> <candidate_file> <algo> <timeLimit> <n_workers>
// using candidates from:
//   ttapp d <func_module> <N_Taylor> <delta> <a> <M_offsets> <n_workers>
// Parameters:
```

```
//   N_Taylor = <N_Taylor> # number of Taylor terms
//   ...
// 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 candidates command, the same cover lines (the first reading `# cover algorithm <algo>: <its name>`), and a note on the data format. 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.

**`auto_taylor_f_tiles.c`** defines typed constants

```
static const int    NTay     = <N>;   // number of Taylor terms
static const double delta    = <d>;   // relative error tolerance (units of eps = 2^-53)
static const int    aNum     = <p>;   // tile edge a = aNum/aDen
static const int    aDen     = <q>;   // tile column of x >= 0: (int)(aDen*x)/aNum
static const int    nXcover  = <n>;   // number of tiles per axis
```

and the tile table

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

whose dimension $2\,\texttt{nXcover}^2$ is written out as an integer, and whose element type becomes `short` when the cover has 128 centers or more. This array stores both the cover index and the per-tile truncation index $N_k$ for each tile, interleaved, one tile column after the other:

```
Tiles[2*(jx*nXcover+jy)]      // cover index of tile (jx, jy)
Tiles[2*(jx*nXcover+jy)+1]    // truncation index N_k of that tile
```

Cover index $-2$ marks a cell outside the domain.

**`auto_taylor_f_coeffs.c`** carries the same header, with a usage section that points to the tiles file for the parameters, and defines

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

where `nCenters` is the number of selected expansion centers; both factors are written as concrete integers. 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 $n = 0, 1, \ldots, N-1$, all in hexadecimal float format.

## 5.2 Hexadecimal output

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

```
0x1.ef90904c7eeeep-2, -0x1.9d5e2f887fe99p-15, ...
```

The letter `p` is followed by the base-2 exponent in decimal notation: `0x1.8p-13` represents $1.5{\cdot}2^{-13}$. This notation keeps the coefficient table short while representing every double-precision number exactly.

### 5.3 Alignment specifier

As in the companion software *ppapp* for piecewise Chebyshev approximation [10, 11], 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.

### 5.4 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/taylortest.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.

## References


[1] *Python Package Index. Project python-flint*, https://pypi.org/project/python-flint.

[2] *Python Package Index. Project ttapp*, https://pypi.org/project/ttapp.

[3] *SciPy. Fundamental algorithms for scientific computing in Python*, 2008–.

[4] *FLINT: Fast Library for Number Theory*, 2024, https://flintlib.org. Version 3.1.2.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, Mass., 3rd ed., 2009.

[6] T. Grossman and A. Wool, *Computational experience with approximation algorithms for the set covering problem*, European J. Oper. Res., 101 (1997), pp. 81–92.

[7] F. Johansson, *Arb: efficient arbitrary-precision midpoint-radius interval arithmetic*, IEEE Trans. Comput., 66 (2017), pp. 1281–1292, https://doi.org/10.1109/TC.2017.2690633.

[8] J. Matoušek, *Lectures on Discrete Geometry*, vol. 212 of Graduate Texts in Mathematics, Springer, New York, 2002.

[9] L. Perron and V. Furnon, *OR-Tools: the CP-SAT constraint solver (version 9.15)*, 2026, https://developers.google.com/optimization.

[10] J. Wuttke and A. Kleinsorge, *ppapp, Piecewise polynomial approximation generator*, 2025, https://jugit.fz-juelich.de/mlz/app/ppapp.

[11] J. Wuttke and A. Kleinsorge, *Algorithm 1062: Code Generation for Piecewise Chebyshev Approximation*, ACM Trans. Math. Software, 52 (2026), 13 (16 pages), https://doi.org/10.1145/3805698.

[12] J. Wuttke, *ttapp, Tilewise Taylor approximation generator*, 2026, https://jugit.fz-juelich.de/mlz/app/ttapp, https://doi.org/10.5281/zenodo.22277190.

[13] J. Wuttke, *Tilewise Taylor approximation of an analytic function with near machine precision*, submitted to Numerical Algorithms.