

ChebPy

Computing with Functions via Chebyshev Approximation in Python

The ChebPy Developers

Version 0.10.0

Abstract

ChebPy is a Python library for numerical computing with *functions* rather than numbers. A smooth function is replaced by a polynomial interpolant through Chebyshev points, accurate to close to machine precision, and thereafter differentiation, integration, rootfinding, convolution and ordinary arithmetic are carried out on that surrogate. The result is a system in which `f + g`, `f.diff()` and `f.roots()` mean what a mathematician expects them to mean. This paper describes the mathematical foundation (Section 2), the numerical kernels that implement it (Section 3), the two-hierarchy object model that organises them (Section 4), the extensions that carry the idea beyond smooth functions on bounded intervals — Fourier technology for periodic functions, support truncation for infinite intervals and endpoint-clustering maps for branch singularities (Section 5) — and the higher-level applications built on top (Section 6). ChebPy is a Python descendant of the MATLAB package Chebfun, and follows its algorithmic choices closely while departing from them where a different design suits the host language better.

Contents

1	Introduction	2
2	Chebyshev approximation	2
2.1	Chebyshev polynomials and series	2
2.2	Why the coefficients decay	3
2.3	Interpolation, and why it is nearly as good	3
3	The numerical kernels	4
3.1	The discrete transform	4
3.2	Deciding where to stop: <code>standard_chop</code>	4
3.3	Adaptive construction	5
3.4	Evaluation	5
3.5	Calculus	5
3.6	Multiplication	6
3.7	Rootfinding	6
3.8	Legendre conversion and convolution	7
4	Architecture	7
4.1	Two hierarchies meeting at a seam	7
4.2	Import layering	8
4.3	The piecewise container	8
4.4	The public surface	8
4.5	Interoperability	9

5	Beyond smooth functions on bounded intervals	9
5.1	Periodic functions: <code>Trigtech</code>	9
5.2	Infinite intervals: <code>CompactFun</code>	9
5.3	Endpoint singularities: <code>Singfun</code>	10
6	Applications	10
6.1	Quasimatrices	10
6.2	Gaussian process regression	10
7	Implementation and validation	11
8	Conclusion	11

1 Introduction

Numerical software usually asks the user to discretise first and compute afterwards: choose a grid, evaluate, and then apply a quadrature rule or a finite-difference stencil to the resulting array. The discretisation is the user’s responsibility, and its accuracy is the user’s problem.

ChebPy inverts that arrangement. The object of computation is the function itself. When a user writes

```
from chebpy import chebfun
import numpy as np

f = chebfun(lambda x: np.sin(10 * x) * np.exp(-x**2), [-3, 3])
f.sum() # definite integral over [-3, 3]
f.diff() # derivative, again a chebfun
f.roots() # every root in [-3, 3]
(f * f).sum() # integral of the square
```

the constructor samples `f` at Chebyshev points, adaptively increasing the sample count until the Chebyshev coefficients have decayed to the level of rounding error, and stores the truncated coefficient vector. Every subsequent operation is exact on that polynomial, and therefore accurate to roughly machine precision on the original function. The user chose no grid and no step size.

This is the design of Chebfun [13, 6], the MATLAB system developed at Oxford since 2004. ChebPy brings it to Python, built on NumPy, and is distributed on PyPI under the name `chebfun` while being imported as `chebpy`. The library was begun by Mark Richardson and is released under the BSD-3-Clause licence.

The central claim that makes any of this practical is one of approximation theory: for functions with even modest smoothness, polynomial interpolation in Chebyshev points converges fast, and for analytic functions it converges geometrically. Section 2 states this precisely; the rest of the paper is about turning it into software.

2 Chebyshev approximation

2.1 Chebyshev polynomials and series

The Chebyshev polynomial of the first kind of degree k is defined on $[-1, 1]$ by

$$T_k(x) = \cos(k \arccos x), \quad x \in [-1, 1], \quad (1)$$

equivalently by the three-term recurrence

$$T_0(x) = 1, \quad T_1(x) = x, \quad T_{k+1}(x) = 2xT_k(x) - T_{k-1}(x). \quad (2)$$

Under the substitution $x = \cos \theta$, (1) becomes $T_k(\cos \theta) = \cos k\theta$: a Chebyshev series in x is a cosine series in θ . Nearly every algorithm in this paper is, at bottom, an exploitation of that identity — it is what allows the fast Fourier transform to do the work.

A Lipschitz-continuous f on $[-1, 1]$ has a unique absolutely and uniformly convergent Chebyshev expansion

$$f(x) = \sum_{k=0}^{\infty} a_k T_k(x), \quad a_k = \frac{2 - \delta_{k0}}{\pi} \int_{-1}^1 \frac{f(x) T_k(x)}{\sqrt{1-x^2}} dx. \quad (3)$$

2.2 Why the coefficients decay

The speed at which $a_k \rightarrow 0$ governs everything. Let E_ρ denote the *Bernstein ellipse*: the image of the circle $|z| = \rho > 1$ under the Joukowski map $z \mapsto (z + z^{-1})/2$, an ellipse with foci ± 1 and semi-axis sum ρ .

Theorem 1 (Geometric convergence; see [12], Ch. 8). *If f is analytic in E_ρ and satisfies $|f| \leq M$ there, then*

$$|a_k| \leq 2M\rho^{-k}, \quad \|f - f_n\|_\infty \leq \frac{2M\rho^{-n}}{\rho - 1},$$

where f_n is the degree- n truncation of (3).

Theorem 2 (Algebraic convergence; see [12], Ch. 7). *If f has $\nu \geq 1$ derivatives on $[-1, 1]$ with $f^{(\nu)}$ of bounded variation V , then $|a_k| = O(k^{-\nu-1})$ and $\|f - f_n\|_\infty = O(n^{-\nu})$.*

So an entire function is captured to machine precision by a few dozen coefficients; a C^∞ function by rather more; a function with a branch point on $[-1, 1]$ by far too many — which is precisely the case handled by the splitting and singularity machinery of Sections 4.3 and 5.3.

2.3 Interpolation, and why it is nearly as good

ChebPy does not compute the integrals in (3). It interpolates. The *Chebyshev points of the second kind* are

$$x_j = \cos\left(\frac{(n-1-j)\pi}{n-1}\right), \quad j = 0, \dots, n-1, \quad (4)$$

which are the extrema of T_{n-1} together with the endpoints ± 1 , ordered left to right so that $x_0 = -1$ and $x_{n-1} = 1$. In ChebPy these are produced by `algorithms.chebpts2(n)`.

Let p_{n-1} be the unique polynomial of degree $\leq n-1$ interpolating f at these points, with discrete Chebyshev coefficients c_k . The relationship between c_k and the true a_k is *aliasing*: on the grid (4), T_m and $T_{2(n-1)\pm m}$ are indistinguishable, so

$$c_k = a_k + a_{2(n-1)-k} + a_{2(n-1)+k} + a_{4(n-1)-k} + \dots \quad (5)$$

Under the hypotheses of Theorem 1 the aliasing error is of the same order as the truncation error itself. The practical consequence is the one that matters: *interpolation is within a small factor of the best polynomial approximation*, so ChebPy can use function samples — cheap — and lose almost nothing.

Two classical facts complete the picture. Interpolation in these points is stable: the Lebesgue constant grows only like $\frac{2}{\pi} \log n$. And it underlies Clenshaw–Curtis quadrature, so the integration routine of Section 3.5 inherits the same accuracy as the representation.

3 The numerical kernels

This section describes the algorithms in `chebpy.algorithms` and `chebpy.chebtech`, which together constitute the computational core. Table 1 summarises their costs.

Operation	Method	Cost
Values $\rightarrow$ coefficients	DCT-I via FFT	$O(n \log n)$
Coefficients $\rightarrow$ values	IDCT-I via FFT	$O(n \log n)$
Truncation length	<code>standard_chop</code>	$O(n)$
Construction	adaptive doubling	$O(n \log n)$
Evaluation at m points	Clenshaw / barycentric	$O(mn)$
Differentiation	coefficient recurrence	$O(n)$
Indefinite integration	coefficient recurrence	$O(n)$
Definite integration	Clenshaw–Curtis weights	$O(n)$
Multiplication	FFT coefficient convolution	$O(n \log n)$
Rootfinding	colleague eigenvalues + subdivision	$O(n^3)$ per block
Chebyshev $\leftrightarrow$ Legendre	three-term recurrence	$O(n^2)$

Table 1: Cost of the principal kernels, for a representation of length n .

3.1 The discrete transform

Converting between the n samples $v_j = f(x_j)$ and the n discrete coefficients c_k is a discrete cosine transform of type I. ChebPy realises it with NumPy’s complex FFT by even extension. In `algorithms.vals2coeffs2`, the sample vector is mirrored to length $2n - 2$,

$$\tilde{v} = (v_{n-1}, v_{n-2}, \dots, v_0, v_1, v_2, \dots, v_{n-2}),$$

an inverse FFT is applied, the first n entries are retained, and the interior entries $c_1, \dots, c_{n-2}$ are doubled. The inverse map, `coeffs2vals2`, reverses these steps with a forward FFT. Both routines detect purely real or purely imaginary input and take the real part accordingly, so a real function never acquires a spurious imaginary component through the transform.

3.2 Deciding where to stop: `standard_chop`

Given a coefficient vector believed to have converged, how many entries are genuinely significant? Truncating too early loses accuracy; truncating too late stores rounding noise and slows every later operation. ChebPy implements the algorithm of Aurentz and Trefethen [2] in `algorithms.standard_chop`, which proceeds in three steps.

1. **Envelope.** Form the monotonically non-increasing envelope of $|c_k|$ by taking a running maximum from the tail backwards, normalised so that its first entry is 1.
2. **Plateau.** Scan for the first index j at which the envelope stops decreasing meaningfully — the point where genuine decay gives way to the flat noise floor. The test compares the envelope at j with its value at $\lceil 1.25j + 5 \rceil$ against a tolerance-dependent ratio $r = 3(1 - \log e_1 / \log \varepsilon)$.
3. **Refinement.** Within the plateau, minimise $\log_{10}(\text{envelope})$ plus a linear ramp that biases the choice leftwards, and cut there.

Vectors shorter than 17 entries are never chopped: there is not enough evidence of decay to justify it. The tolerance $\varepsilon^{7/6}$ appears in step 3 as a floor, deliberately slightly looser than ε itself.

3.3 Adaptive construction

`algorithms.adaptive` drives the constructor. It samples on grids of $n = 2^k + 1$ points for $k = 4, 5, \dots$ — that is, 17, 33, 65, $\dots$ — and at each stage transforms to coefficients and applies `standard_chop`. If the chop length is strictly less than the vector length, the coefficients have resolved the function and the loop exits with the truncated vector. Doubling is used so that the total cost is dominated by the final, successful transform.

Two guards matter. If the largest sampled value falls below the working tolerance, the function is declared indistinguishable from zero and represented by the single coefficient 0; this avoids the degenerate case in which a vanishing vertical scale makes every relative test meaningless. And if k reaches `prefs.maxpow2` (default 16, i.e. 65 537 points) without resolution, ChebPy emits a `UserWarning` naming the class that failed and returns the unresolved coefficients rather than raising — a non-converged answer with a warning being more useful, in interactive work, than no answer.

The working tolerance is $\varepsilon_{\text{mach}} \cdot \max(h, 1)$, where h is a horizontal scale passed in by the caller, so that a function on a wide interval is not held to an unattainably tight absolute standard.

3.4 Evaluation

Two evaluation schemes coexist, and `Chebtech.__call__` selects between them by keyword (`how="clenshaw"` by default).

Clenshaw’s algorithm. `algorithms.clenshaw` sums the series directly through the backward recurrence associated with (2),

$$b_k = a_k + 2x b_{k+1} - b_{k+2}, \quad b_n = b_{n+1} = 0, \quad p(x) = a_0 + x b_1 - b_2. \quad (6)$$

The implementation unrolls the loop two steps at a time, which halves the number of vector temporaries NumPy must allocate. The cost is $O(n)$ per evaluation point with no precomputation.

Barycentric interpolation. `algorithms.bary` evaluates instead from the samples, using the second barycentric formula [3]

$$p(x) = \sum_{j=0}^{n-1} \frac{w_j}{x - x_j} f_j \bigg/ \sum_{j=0}^{n-1} \frac{w_j}{x - x_j}, \quad (7)$$

which for the points (4) has the remarkably simple weights $w_j = (-1)^j$, halved at the two endpoints (`algorithms.barywts2`). Formula (7) is numerically stable and $O(n)$ per point. ChebPy switches between two loop orderings — over evaluation points when there are few of them, over nodes when there are many — and repairs the removable singularities at $x = x_j$, where (7) produces 0/0, by substituting the exact nodal value.

3.5 Calculus

Because differentiation and integration act linearly on (3) and T_k satisfies simple recurrences, all three calculus operations are $O(n)$ manipulations of the coefficient vector — no quadrature nodes, no step size.

Definite integral. From $\int_{-1}^1 T_k(x) dx = 0$ for odd k and $2/(1 - k^2)$ for even k ,

$$\int_{-1}^1 f(x) dx = 2a_0 + \sum_{\substack{k \geq 2 \\ k \text{ even}}} \frac{2a_k}{1 - k^2}, \quad (8)$$

implemented in `Chebtech.sum` by zeroing the odd coefficients and contracting with the weight vector. This is Clenshaw–Curtis quadrature, expressed in coefficient space.

Indefinite integral. `Chebtech.cumsum` returns F with $F(-1) = 0$ from

$$b_k = \frac{a_{k-1} - a_{k+1}}{2k} \quad (k \geq 2), \quad b_1 = a_0 - \frac{1}{2}a_2, \quad b_0 = \sum_{k \geq 1} (-1)^{k+1} b_k, \quad (9)$$

the last line being exactly the constant that enforces the boundary condition, since $T_k(-1) = (-1)^k$.

Derivative. `Chebtech.diff` inverts (9). Writing $f' = \sum_k z_k T_k$,

$$z_{k-1} = z_{k+1} + 2k a_k, \quad k = n-1, n-2, \dots, 1, \quad (10)$$

with z_0 subsequently halved. The implementation evaluates the two parity classes as separate reversed cumulative sums, so the whole recurrence is two NumPy `cumsum` calls rather than a Python loop.

Note that (10) multiplies by k : differentiation amplifies the high-order coefficients, and repeated differentiation loses roughly one digit per application to a well-resolved function. This is intrinsic to the operation, not to the representation.

3.6 Multiplication

The product of two Chebyshev series is a convolution of their coefficients, which `coeffmult` evaluates in $O(n \log n)$. Each coefficient vector is extended to a symmetric sequence of length $2n - 2$ with the leading entry doubled; the two extensions are multiplied pointwise in Fourier space; the result is folded back and scaled by $1/4$. Since the product of two degree- n polynomials has degree $2n$, the operands are first prolonged to a common length so that the product is represented exactly rather than implicitly aliased.

3.7 Rootfinding

Finding all roots of a polynomial of degree several hundred is the one place where a naive approach fails outright, and ChebPy’s treatment in `algorithms.rootsunit` is correspondingly careful.

Colleague matrix. For a series $f = \sum_{k=0}^N a_k T_k$ with $a_N \neq 0$, the roots are the eigenvalues of the $N \times N$ *colleague matrix* [7, 4]

$$A = \begin{pmatrix} 0 & 1 & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ & \ddots & \ddots & \ddots & \\ & & \frac{1}{2} & 0 & \frac{1}{2} \\ & & & \frac{1}{2} & 0 \end{pmatrix} - \frac{1}{2a_N} e_N(a_0, a_1, \dots, a_{N-1}), \quad (11)$$

the Chebyshev-basis analogue of the companion matrix, formed as a symmetric tridiagonal part plus a rank-one correction in the last row. ChebPy passes A to `numpy.linalg.eigvals`.

Subdivision. The eigenvalue solve is $O(N^3)$, so it is used only for $N \leq 50$. Above that threshold ChebPy bisects the interval, evaluates the series on each half by Clenshaw, re-expands each half by `vals2coeffs2`, and recurses. The split point is not 0 but

$$\sigma = -0.004849834917525,$$

a deliberately irrational-looking constant inherited from Chebfun. Splitting at the exact midpoint would repeatedly land breakpoints on symmetry axes of even or odd functions, where roots cluster and the two halves inherit the worst of the conditioning; an off-centre split avoids that resonance. The recursion doubles the tolerance at each level, since accumulated map compositions make the deep sub-intervals progressively less able to sustain the original accuracy.

Filtering and polishing. The eigenvalues of (11) are complex in general. Roots with $|\text{Im}| \geq h_{\text{tol}}$ are discarded, the survivors are taken as real, those outside $[-1 - h_{\text{tol}}, 1 + h_{\text{tol}}]$ are dropped, and the extreme two are clamped to the interval so that a root at an endpoint is reported exactly there. `algorithms.newtonroots` then applies Newton polishing, at most `prefs.maxiter` = 10 iterations, to recover the digits lost in the eigenvalue solve.

3.8 Legendre conversion and convolution

Convolution is not natural in the Chebyshev basis, but it is in the Legendre basis, where the Hale–Townsend algorithm [8] computes $h = f \star g$ in $O(n^2)$ via a recurrence on the Legendre coefficients. ChebPy therefore provides `cheb2leg` and `leg2cheb`, each an $O(n^2)$ three-term recurrence derived from (2) and from $(k+1)P_{k+1} = (2k+1)xP_k - kP_{k-1}$ respectively, built column by column so that the recurrence is applied to whole coefficient vectors at once.

`chebpy._convolution` then dispatches on the shape of the operands. Two single-piece funs of equal width take the fast Legendre path, whose output is a piecewise polynomial on $[-2, 2]$ split at the origin — the convolution of two functions supported on $[-1, 1]$ generically has a derivative discontinuity there, and representing it as two pieces keeps both smooth. Anything else falls back to building each output sub-interval adaptively by Gauss–Legendre quadrature.

4 Architecture

4.1 Two hierarchies meeting at a seam

The library separates *what the approximation is* from *where it lives*. Two runtime type hierarchies express that split and meet at `Classicfun`:

Hierarchy	Meaning
<code>Onefun</code> $\rightarrow$ <code>Smoothfun</code> $\rightarrow$ <code>{Chebtech, Trigtech}</code>	a function on the reference interval $[-1, 1]$
<code>Fun</code> $\rightarrow$ <code>Classicfun</code> $\rightarrow$ <code>{Bndfun, CompactFun, Singfun}</code>	that function placed on $[a, b]$

A `Chebtech` knows about Chebyshev coefficients and nothing about intervals. A `Bndfun` owns a `Chebtech` plus an affine map $[-1, 1] \rightarrow [a, b]$, and translates between them: evaluation composes with the forward map, `diff` divides by the constant Jacobian, `sum` multiplies by it. Because the Jacobian of an affine map is a constant, these translations are one-line adjustments — which is exactly why Section 5.3, where the map is *not* affine, has to override them.

Above both sits `Chebfun`, which is not a `Fun` at all but a *container* of `Fun` pieces.

4.2 Import layering

The package is organised into strict import layers; a lower layer never imports an upper one at module scope.

Layer	Modules
high-level API / applications	api, quasimatrix, gpr
piecewise container	chebfun (+ _convolution, _pointwise, _singular_construction, _ufuncs)
Classicfun pieces	bndfun, compactfun, singfun
interval-mapped base	classicfun
reference-interval technology	chebtech, trigtech
numerical kernels	algorithms, chebyshev
primitives	utilities, plotting, smoothfun
foundations	settings, exceptions, decorators, maps, fun, onefun

The foundation layer has no intra-package imports at all. **Chebfun**'s public methods are thin wrappers that delegate to the private implementation modules, which keeps the main class readable at roughly a thousand lines despite the breadth of the operator surface.

4.3 The piecewise container

A single polynomial cannot represent $|x|$ or a step, but a *piecewise* polynomial can. A **Chebfun** holds an ordered list of **Fun** pieces on abutting sub-intervals, together with breakpoint data recording the value at each junction. Operations distribute over the pieces: **sum** adds the piece integrals, **roots** concatenates the piece roots and (under **prefs.mergeroots**) merges duplicates arising at shared breakpoints, and a binary operation between two **Chebfun**s first refines both onto the union of their breakpoints so that the operands are piecewise-aligned.

Breakpoints therefore serve two purposes at once: they carry genuine discontinuities, and they subdivide regions where a single polynomial would need an impractical degree.

4.4 The public surface

Entry point	Purpose
chebfun(f, domain, n=None)	construct from a callable, adaptively or at fixed length
trigfun(f, domain)	construct with Fourier technology (Section 5.1)
equifun(values, domain)	construct from equispaced samples
pwc(domain, values)	piecewise-constant Chebfun
chebpts(n, domain)	Chebyshev points and barycentric weights
gpr(x, y, ...)	Gaussian process regression (Section 6.2)
Quasimatrix, polyfit	continuous linear algebra (Section 6.1)
UserPreferences	tolerances and algorithmic switches

Preferences are a context-managed singleton, so a block of code can tighten or loosen tolerances locally without disturbing global state:

```
from chebpy import UserPreferences

with UserPreferences() as prefs:
    prefs.eps = 1e-10 # looser tolerance, cheaper constructions
    g = chebfun(f, [0, 1]) # built at 1e-10
# previous preferences restored here
```

The defaults are $\varepsilon = \varepsilon_{\text{mach}}$, `maxpow2` = 16, `maxiter` = 10 Newton steps, and 2001 plotting samples.

4.5 Interoperability

Chebfun registers NumPy ufunc overloads (`chebpy._ufuncs`), so `np.sin(f)`, `np.exp(f)` and their relatives return Chebfuns constructed by composition rather than arrays of samples. The composed function is passed back through the adaptive constructor, so `sin(f)` carries its own resolved degree rather than inheriting f 's.

5 Beyond smooth functions on bounded intervals

Chebyshev interpolation resolves functions that are analytic on a neighbourhood of a bounded real interval. Three extensions widen that domain of applicability, each by changing the representation rather than the algorithms built on it.

5.1 Periodic functions: **Trigtech**

For a smooth *periodic* function, a Fourier series is the natural basis: it needs roughly $\pi/2$ times fewer degrees of freedom than a Chebyshev series of comparable accuracy, and it enforces periodicity exactly instead of approximately. **Trigtech** sits beside **Chebtech** under **Smoothfun**, exposing the identical **Onefun** interface.

Samples are taken at the n equispaced points $x_j = -1 + 2j/n$, and coefficients are stored in NumPy's native FFT ordering,

$$c_k = \frac{1}{n} \sum_{j=0}^{n-1} f(x_j) e^{-2\pi i j k / n} = (\text{numpy.fft.fft}(v)/n)_k, \quad (12)$$

with the constant term at index 0, positive frequencies next and negative frequencies wrapped to the end. Evaluation at arbitrary $x \in [-1, 1]$ uses the summation formula

$$f(x) = \sum_k c_k e^{i\pi\omega_k(x+1)}, \quad \omega_k = \text{fftfreq}(n) \cdot n. \quad (13)$$

Storing in FFT order rather than the human-readable DC-centred order means the transforms are direct NumPy calls with no shifting; a helper converts to centred order for display and coefficient plots.

5.2 Infinite intervals: **CompactFun**

MATLAB Chebfun handles $[a, \infty)$ and $(-\infty, \infty)$ with **@unbndfun**, mapping the infinite interval onto $[-1, 1]$ by a rational change of variables. ChebPy takes a different route, and the departure is deliberate.

A **CompactFun** distinguishes two intervals. The *logical* interval is what the user asked for and may have infinite endpoints. The *numerical support* is the finite region outside which the function differs from its asymptotic limit by less than a tolerance. The constructor discovers the latter by probing outwards from a finite anchor — the bounded endpoint for a semi-infinite interval, or the origin for the doubly-infinite case — doubling the radius up to `numsupp_max_width` = 10^6 over at most `numsupp_max_probes` = 30 probes, and detecting where the tail has settled. Inside that region an ordinary **Chebtech** is stored; outside it the function is reported as a scalar constant, `tail_left` or `tail_right` (default 0).

The advantage is that the stored object is an ordinary Chebyshev representation, so every kernel of Section 3 applies unchanged and no rational map degrades the conditioning. The

limitation is equally clear: the approach suits functions that decay to a constant, and integrals that diverge are detected and raise `DivergentIntegralError` rather than returning a meaningless finite number. For a finite logical interval, storage and logical intervals coincide and a `CompactFun` behaves exactly as a `Bndfun`.

5.3 Endpoint singularities: `Singfun`

A function such as $\sqrt{x}$ on $[0, 1]$ or $\sqrt{x(1-x)}$ is analytic on the open interval but has a branch point at an endpoint. Its Chebyshev coefficients decay only algebraically (Theorem 2), so the adaptive constructor runs to its iteration limit and returns a poor answer.

`Singfun` removes the singularity by a change of variable. It is a sibling of `Bndfun` whose only structural novelty is that the bijection between the storage variable $t \in [-1, 1]$ and the logical variable $x \in [a, b]$ is not affine but a *slit-strip map* of Adcock and Richardson [1], whose derivative vanishes super-algebraically at the clustered endpoint. For a left-clustered map with parameters (L, α) ,

$$s(y) = \frac{L}{2}(y - 1), \quad \gamma = \frac{\alpha}{\pi} \log(e^{\pi/\alpha} - 1), \quad u(s) = \frac{\alpha}{\pi} \log(1 + e^{\pi(s+\gamma)/\alpha}), \quad (14)$$

with $x = a + (b - a)u(s(y))$; the shift γ is chosen so that $u(0) = 1$. `SingleSlitMap` clusters at one endpoint, `DoubleSlitMap` at both.

The composition $f \circ m$ is then analytic in a Bernstein ellipse around $[-1, 1]$ and is resolved by ordinary Chebyshev interpolation to spectral accuracy. Everything that depends only on the map being a bijection — evaluation, rootfinding, the binary operators — is inherited from `Classicfun` unchanged. Only `sum`, `cumsum` and `diff` need bespoke implementations, because the constant-Jacobian shortcuts of Section 4 no longer hold.

The map does not quite cover $[a, b]$: the image of $[-1, 1]$ omits a small gap of relative width `gap_unit` at the clustered end, controlled by L . This is the price of the construction, and the parameters are exposed through `MapParams` so a user can trade coverage against conditioning.

6 Applications

6.1 Quasimatrices

A *quasimatrix* is an $\infty \times n$ matrix: n columns, each a Chebfun on a common domain, with rows indexed by a continuum. The inner product $\int_a^b f(x)g(x) dx$ replaces the discrete dot product, and with that substitution much of dense linear algebra carries over. ChebPy’s `Quasimatrix` implements QR by the Householder triangularisation of Trefethen [11], and from it the SVD, least-squares solves and polynomial fitting via `polyfit`.

The continuous QR is the useful primitive: applied to the quasimatrix whose columns are $1, x, x^2, \dots$, Gram–Schmidt would break down numerically at modest degree, whereas Householder triangularisation returns a numerically orthonormal basis — the Legendre polynomials, up to scaling — and does so stably.

6.2 Gaussian process regression

`chebpy.gpr` implements Gaussian process regression following Rasmussen and Williams [9] and the Chebfun `gpr` routine. Given observations (x_i, y_i) , it forms the posterior over functions and returns the posterior mean as a Chebfun, the variance as a Chebfun, and any requested posterior samples as a `Quasimatrix`.

Returning Chebfuns rather than sampled arrays is the point. The posterior mean can immediately be differentiated, integrated or root-found; the credible band is a pair of Chebfuns whose crossings with a threshold are located by Section 3.7’s rootfinder rather than by scanning

a grid. A periodic kernel is available, in which case the posterior is built on Fourier technology (Section 5.1).

7 Implementation and validation

ChebPy targets Python 3.11–3.14 and depends only on NumPy and Matplotlib. Its development infrastructure is synchronised from a shared template, keeping the repository’s own source clearly separated from generated configuration.

The correctness bars are enforced mechanically rather than by convention. The test suite spans 23 modules covering every layer from the FFT kernels to the public API, and coverage is held at **100%**; genuinely unreachable defensive branches must carry an explicit `pragma` with a stated reason, which makes each exemption reviewable rather than invisible. Docstring coverage is likewise held at 100%, and the package type-checks cleanly under `mypy -strict` with two independent checkers in the continuous integration gate. Additional workflows run mutation testing, fuzzing, CodeQL analysis and dependency auditing.

For a library whose value proposition is “accurate to machine precision”, the significance of these bars is specific: a silent inaccuracy in a kernel such as `standard_chop` or the colleague matrix would not crash anything. It would return plausible numbers that are wrong in the last several digits. Full branch coverage of the numerical paths is the practical defence.

8 Conclusion

ChebPy makes functions into first-class objects by pairing one theorem with a handful of carefully chosen algorithms. The theorem is that Chebyshev interpolation converges geometrically for analytic functions and is near-optimal among polynomial approximations. The algorithms — the FFT-based transform, the `standard_chop` truncation test, the adaptive doubling constructor, Clenshaw and barycentric evaluation, the coefficient recurrences for calculus, and the colleague-matrix rootfinder with off-centre subdivision — turn it into a system where `f.roots()` returns every root and `f.sum()` returns the integral to fifteen digits, with no grid chosen by the user.

The architecture separates the reference-interval representation from its placement on a domain, which is what allows Fourier technology, infinite intervals and endpoint singularities to be added as new representations rather than as special cases threaded through the algorithms. The higher-level applications — quasimatrices and Gaussian process regression — then follow from the same principle applied once more: if functions are objects, then matrices of functions and distributions over functions are objects too.

References

- [1] B. Adcock and M. Richardson, *New exponential variable transform methods for functions with endpoint singularities*, arXiv:1305.2643, 2013.
- [2] J. L. Aurentz and L. N. Trefethen, *Chopping a Chebyshev series*, arXiv:1512.01803, 2015.
- [3] J.-P. Berrut and L. N. Trefethen, *Barycentric Lagrange interpolation*, SIAM Review, 46(3):501–517, 2004.
- [4] J. P. Boyd, *Computing zeros on a real interval through Chebyshev expansion and polynomial rootfinding*, SIAM Journal on Numerical Analysis, 40(5):1666–1682, 2002.

- [5] C. W. Clenshaw, *A note on the summation of Chebyshev series*, Mathematics of Computation, 9(51):118–120, 1955.
- [6] T. A. Driscoll, N. Hale and L. N. Trefethen (eds.), *Chebfun Guide*, Pafnuty Publications, Oxford, 2014.
- [7] I. J. Good, *The colleague matrix, a Chebyshev analogue of the companion matrix*, Quarterly Journal of Mathematics, 12(1):61–68, 1961.
- [8] N. Hale and A. Townsend, *An algorithm for the convolution of Legendre series*, SIAM Journal on Scientific Computing, 36(3):A1207–A1220, 2014.
- [9] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*, MIT Press, 2006.
- [10] L. N. Trefethen, *Spectral Methods in MATLAB*, SIAM, Philadelphia, 2000.
- [11] L. N. Trefethen, *Householder triangularization of a quasimatrix*, IMA Journal of Numerical Analysis, 30(4):887–897, 2010.
- [12] L. N. Trefethen, *Approximation Theory and Approximation Practice*, SIAM, Philadelphia, 2013.
- [13] The Chebfun Developers, *Chebfun — numerical computing with functions*, <http://www.chebfun.org/>.