Solving PDEs using Spectral Methods in the Chebyshev basis by example of the Heat Equation

Special Topic on Approximation of Functions

Candidate Number: 12345

Abstract

This work shall attempt to numerically solve the heat equation $u_t = \alpha u_{xx}$ with Dirichlet boundary conditions over the domain $[-1,1] \times [0,T]$ by representing the spatial component as a *chebfun* (Chebyshev series) and moving on in time by the Forward Euler numerical scheme.

Our Goal: Numerically obtain the solution $u(x,T)$ of

$$\begin{cases} \frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} & u : [-1,1] \times [0,T] \mapsto \mathbb{R},\ T \in \mathbb{R}^+,\ \alpha \in \mathbb{R}^+ \\ u(x_j, 0) = u_0(x_j) & \forall x_j \in \tilde{X}_N,\ N \in \mathbb{N},\ N > 1,\ u_0 : [-1,1] \mapsto \mathbb{R} \\ u(b,t) = u_0(b) & \forall b \in \{-1,1\},\ \forall t \in (0,T]\,. \end{cases}$$

The implementation, centered around what we will refer to as **TschebFun**, including three major algorithms `TschebFun::interpolantThrough()`, `TschebFun::evaluateOn()` and `TschebFun::derivative()`, is done manually in C++, extended to work as a Python module and for demonstration, even features a high-level graphical interface to play with. Finally, we will compare the numerical results with the output of *Chebfun*'s high-level `pde15s()`.

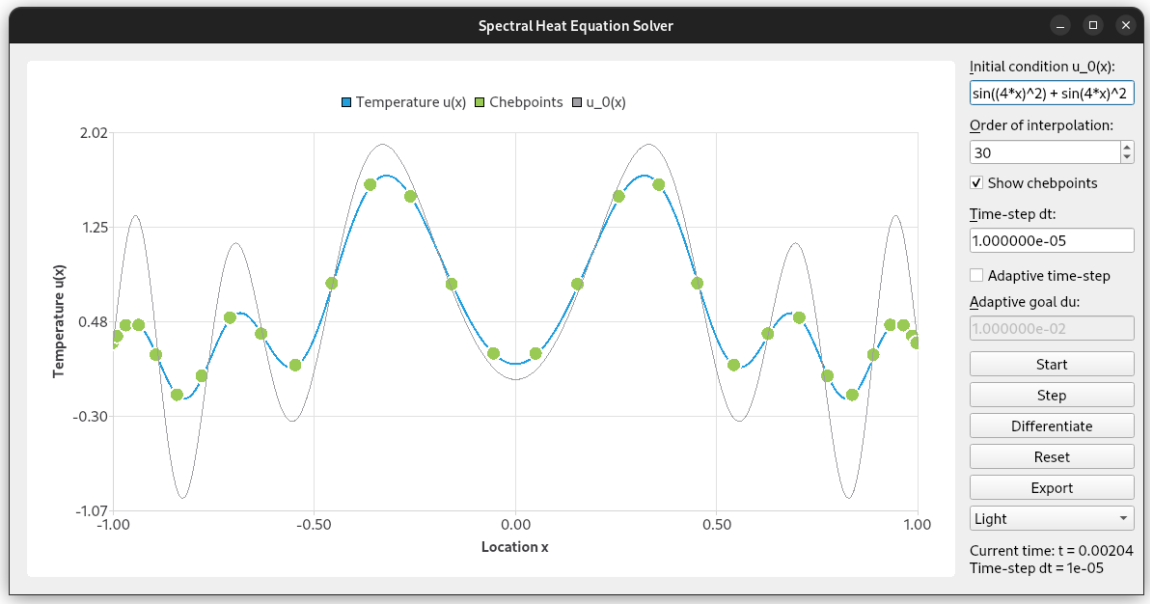

Figure 1: Screenshot of the graphical user interface. After entering an initial expression $u_0(x)$, depicted in grey, the simulation will run upon pressing 'Start'. The solution at time t, depicted in blue, is represented as a Chebyshev series of degree 29.

1 Motivation

Partial differential equations are notoriously hard to solve. One more possible approach to make way in this important class of problems is by the technique of spectral methods, incidentally closely related to finite element methods. The key idea is to perform the problem solution by representation of the occurring functions in a certain basis. For non-periodic problem settings, Chebyshev series are a fantastic choice.

Classical approaches to solving the heat equation are:

- Using the separation Ansatz

$$u(x,t) = \underline{X}(x)\underline{T}(t), \quad \underline{X} : [-1,1] \mapsto \mathbb{R}, \quad \underline{T} : [0,T] \mapsto \mathbb{R}$$

  one obtains a first- and second-order constant-coefficient ordinary differential equation in t and x, respectively. Of course, this is only applicable to suitable initial conditions.

- By Fourier-analysis: Similar to our approach below, or maybe even simpler: differentiation simply multiplies the Fourier coefficients by $\mathrm{i}k$, with k the wave number. This is a spectral method!

- Numerically, a finite difference scheme such as

$$\frac{U_j^{(n+1)} - U_j^{(n)}}{\Delta t} = \alpha \frac{U_{j+1}^{(n)} - 2U_j^{(n)} + U_{j-1}^{(n)}}{(\Delta x)^2}$$

  can be used to solve the problem.

Our method focusses on a combination of a spectral method (for the spatial component) and finite difference approximation (for the temporal component).

The first step is to interpolate through the initial data $\{(x_j, u_0(x_j)) \mid x_j \in X_N\}$, iteratively modify the resulting coefficients according to the partial differential equation and finally obtain a series representation of the solution after time T.

2 Chebyshev Interpolation

Let $\mathbb{N}$ denote the nonnegative integers, so $0 \in \mathbb{N}$.

2.1 Definition: Chebyshev polynomial

Chebyshev[1] polynomials $T_k : \mathbb{R} \mapsto \mathbb{R}$ are functions satisfying

$$T_k(x) = T_k(\cos\theta) := \cos(k\theta) = \frac{1}{2}(z^k + z^{-k})$$
$$z := \mathrm{e}^{i\theta}, \quad x := \mathfrak{Re}(z) = \cos(\theta) = \frac{1}{2}(z + z^{-1})$$

for degree $k \in \mathbb{N}$. Then, $T_0(x) = 1$, $T_1(x) = x$, $T_2(x) = 2x^2 - 1$, and so on.

The above relations between x, z and θ reveal fundamental connections between three famous basis sets: Chebyshev, Laurent and Fourier.

There is a handy method of explicitly writing out the Chebyshev polynomials in x, namely by a three-step recursion with $T_0(x) = \cos(0) = 1$ and $T_1(x) = \cos(\theta) = x$.

2.1 Theorem: Chebyshev recursion formula

The Chebyshev polyomials satisfy the three-term recurrence relation

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

Proof. For $k > 1$, we have

$$\begin{aligned} 2xT_k(x) - T_{k-1}(x) &= 2x\frac{1}{2}(z^k + z^{-k}) - \frac{1}{2}(z^{k-1} + z^{-(k-1)}) \\ &= 2\frac{1}{2}(z + z^{-1})\frac{1}{2}(z^k + z^{-k}) - \frac{1}{2}(z^{k-1} + z^{-k+1}) \\ &= \frac{1}{2}(z^{k+1} + z^{k-1} + z^{-k+1} + z^{-k-1}) - \frac{1}{2}(z^{k-1} + z^{-k+1}) \\ &= \frac{1}{2}(z^{k+1} + z^{-(k+1)}) = T_{k+1}(x) \end{aligned}$$

□

[1] named after Pafnuty Lvovich Chebyshev, alternatively transliterated as Tchebycheff, Tchebyshev (French) or Tschebyschow (German)

2.1 An Orthogonal Basis

The Chebyshev polynomials also satisfy an *orthogonality relation*,

$$\langle T_m, T_n\rangle := \int_{-1}^{1} T_m(x)T_n(x)\frac{1}{\sqrt{1-x^2}}\mathrm{d}x = \int_{\pi}^{0} \cos(m\theta)\cos(n\theta)\frac{-\sin(\theta)}{\sqrt{1-\cos^2(\theta)}}\mathrm{d}\theta\,,$$

which becomes, with the fitting substitution $x = \cos(\theta)$ and $\mathrm{d}x = -\sin(\theta)\mathrm{d}\theta$,

$$\begin{aligned}\langle T_m, T_n\rangle &= \int_0^{\pi} T_m(\cos\theta)T_n(\cos\theta)\frac{\sin\theta}{\sin\theta}\mathrm{d}\theta = \int_0^{\pi}\cos(m\theta)\cos(n\theta)\mathrm{d}\theta \\ &= \frac{1}{2}\int_0^{\pi}\Big(\underbrace{\cos((m+n)\theta)}_{=\cos(2m\theta)\text{ for }m=n} + \underbrace{\cos((m-n)\theta)}_{=1\text{ for }m=n}\Big)\mathrm{d}\theta\end{aligned}$$

along with the knowledge that $\int_0^{\pi}\cos(k\theta)\mathrm{d}\theta = k^{-1}\left[\sin(k\theta)\right]_0^{\pi} = 0$ for $k \in \mathbb{Z}\backslash\{0\}$,

$$\langle T_m, T_n\rangle = \int_0^{\pi} T_m(\cos\theta)T_n(\cos\theta)\mathrm{d}\theta = \begin{cases} 0 & \text{for } m \neq n \\ \pi/2 & \text{for } m = n \neq 0 \\ \pi & \text{for } m = n = 0 \end{cases}$$

which can be effectively utilised to define a function space $(\mathbb{T}, +, \cdot)$ in the *orthogonal* basis of Chebyshev polynomials $\mathcal{T} := \{T_k\}_{k\in\mathbb{N}}$. Note that the operation $\langle\cdot,\cdot\rangle$ satisfies all axioms of an authentic inner product (linearity, etc.) over a function space due to the linearity of the integral.

In the following proceedings, we will restrict our view on functions over the interval $[-1, 1] \subset \mathbb{R}$. Any (real) Lipschitz-continuous function $f \in \mathcal{C}_L$, where $\mathcal{C}_L := \{g : [-1,1] \mapsto \mathbb{R} \mid \exists L \text{ s.t. } \forall x_1, x_2 \in \mathbb{R},\ |g(x_1) - g(x_2)| \leq L \cdot |x_1 - x_2|\}$ can be represented in the Chebyshev basis $\mathcal{T}$, as Lipschitz continuity is a sufficient condition for absolute and uniform convergence of the corresponding series representation

$$f(x) = \sum_{k=0}^{\infty} a_k T_k(x), \quad a_k \in \mathbb{R}, \quad k \in \mathbb{N}\,.$$

Utilising orthogonality, for any $f \in \mathcal{C}_L$, we find coefficients $a_l \in \mathbb{R}$ by 'right-multiplying' the equation $f = \sum_{k=0}^{\infty} a_k T_k$ with any one of the Chebyshev polynomials T_l.

$$\begin{aligned}\langle f, T_l\rangle &= \langle\sum_{k=0}^{\infty} a_k T_k, T_l\rangle = \int_0^{\pi}\sum_{k=0}^{\infty} a_k T_k(\cos\theta)\cdot T_l(\cos\theta)\mathrm{d}\theta \\ &= \sum_{k=0}^{\infty} a_k\langle T_k, T_l\rangle \quad \text{by linearity of the inner product} \\ &= \begin{cases} a_0\pi & \text{for } l = 0 \\ a_l\pi/2 & \text{for } l \neq 0 \end{cases} \quad \text{by orthogonality of the polynomials}\end{aligned}$$

which can easily be rearranged to give explicit relations for a_0 and a_k, summarised in the below theorem.

2.2 Theorem: Chebyshev series coefficient formula

For any $f \in \mathcal{C}_L$, one can obtain the Chebyshev series coefficients a_k, $k \in \mathbb{N}$ as

$$a_0 = \frac{1}{\pi}\langle f, T_0\rangle = \frac{1}{\pi}\int_0^\pi f(\cos\theta)\mathrm{d}\theta$$
$$a_k = \frac{2}{\pi}\langle f, T_k\rangle = \frac{2}{\pi}\int_0^\pi f(\cos\theta)\cos(k\theta)\mathrm{d}\theta, \quad k \neq 0\,.$$

Proof. As given in the discussion above. A different approach for the derivation of the explicit coefficient integrals can be found in Trefethen 2019 along with a complex analysis styled proof. □

Dealing with a numerical problem, we shall then approximate the above two integrals by the rectangular integral rule.

2.2 Numerical Computation of Coefficients

As computers rarely allow us to store infinitely many coefficients a_k, we will work with the truncated Chebyshev series

$$f_N(x) = \sum_{k=0}^{N-1} a_k T_k(x), \quad k \in \{0, ..., N-1\}, \quad N \in \mathbb{N}, N > 1 \tag{1}$$

which approximates the function f with a degree $N-1$ polynomial up to an error

$$f(x) - f_N(x) = \sum_{k=0}^{\infty} a_k T_k(x) - \sum_{k=0}^{N-1} a_k T_k(x) = \sum_{k=N}^{\infty} a_k T_k(x)\,.$$

2.2 Definition: Truncated Chebyshev series function space

Let $(\mathbb{T}_N, +, \cdot)$ denote the ring of truncated Chebyshev series up to order $N \in \mathbb{N}$, $N > 1$, introduced as a subspace of the polynomials of degree up to $N-1$, so $\mathbb{T}_N \subseteq \mathcal{P}_{N-1}$ with $\mathbb{T}_N := \mathrm{span}\{T_k \mid k = 0, ..., N-1\}$, or explicitly stated,

$$\mathbb{T}_N = \left\{ f_N : [-1,1] \mapsto \mathbb{R},\ f_N(x) = \sum_{k=0}^{N-1} a_k T_k(x) \;\middle|\; a_k \in \mathbb{R},\ k = 0, ..., N-1 \right\},$$

inheriting (pointwise) addition and multiplication from $(\mathcal{P}_{N-1}, +, \cdot)$.

Three methods to numerically compute the coefficients a_k of any truncated series $f_N \in \mathbb{T}_N$ approximating an $f \in \mathcal{C}_L$ are:

1. **Coefficient Integral Approximation**. The integrals can be approximated numerically using one of many available *approximation rules*.

   2.3 Theorem: Rectangular integral rule

   For a function $f : [a, b] \mapsto \mathbb{R}$, its integral can be approximated by

   $$\int_a^b f(x)\mathrm{d}x = \lim_{N\to\infty} \frac{b-a}{N} \sum_{j=0}^{N-1} f(x_j), \quad x_j := a + \frac{b-a}{N} j\,.$$

   We will look at an actual implementation of this in the next subsection.

2. **Using the Discrete Cosine Transform**. Recognise the structure of the above integral (Theorem 2.2) for $k \neq 0$ as a cosine transform of the function $f \circ \cos$. Let $g(t) = f(\cos(t))$ for $t \in [0, \pi]$ and 0 otherwise.

   2.3 Definition: Cosine Transform

   $$\hat{g}(\omega) := \int_{-\infty}^{\infty} g(t)\cos(\omega t)\mathrm{d}t, \quad g : \mathbb{R} \mapsto \mathbb{R}, \quad \hat{g} : \mathbb{R} \mapsto \mathbb{R}$$

   2.4 Definition: DCT-II: Discrete Cosine Transform - type II

   For $y_j := g(\theta_j)$, where $\theta_j \in \Theta_N$, the discrete $\hat{y}_k$ are obtained by

   $$\hat{y}_k := \sum_{j=0}^{N-1} y_j \cos\left[\tfrac{\pi}{N}\left(n + \frac{1}{2}\right)k\right] \qquad \text{for } \; k = 0,\ \ldots\ N-1\,.$$

   Most significantly, this approach via the Discrete Cosine Transform can be sped up by means of the *Fast Fourier Transform* (Cooley and Tukey 1965).

3. **Barycentric Interpolation Formula**. Numerically speaking, a significant improvement to these two approaches can be made by using the *Barycentric interpolation formula in Chebyshev points* (Trefethen 2019). Given more time, one should implement this feature in TschebFun as well.

The first two methods demonstrate why it is sensible to choose to sample a function in *chebpoints* (Definition 2.5, also confer Figure 2) instead of equispaced points. A much more fundamental numerical insight relating to the choice of points is the *Runge phenomenon* (Runge 1901).

2.5 Definition: Chebyshev points

From the equispaced points

$$\Theta_N := \left\{ \theta_j := \frac{j\pi}{N-1} \;\middle|\; j = 0, ..., N-1 \right\},$$

we can further define the Chebyshev points as the corresponding $\cos(\theta_j)$,

$$X_N := \{ x_j := \cos(\theta_j) \mid \theta_j \in \Theta_N \}.$$

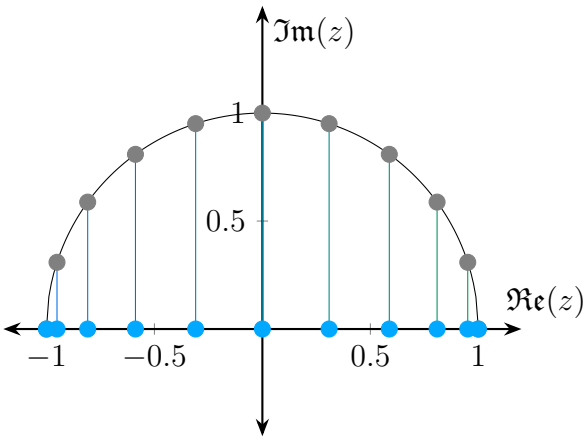

Figure 2: The Chebyshev points $\{x_j = \cos(\theta_j)\}$ are projections of the equispaced points $\{\theta_j\}$ on the unit circle onto the x-axis.

2.3 A Numerical Issue

Wanting to interpolate through N points, namely through $\{(x_j, f(x_j)) \mid x_j \in X_N\}$, the corresponding truncated Chebyshev series will have N coefficients. To approximate the coefficient integral, we use Theorem 2.3 without the limit.

Even approximately speaking, the rectangular integral rule in application to Theorem 2.2 does, in some situations, not approach the correct coefficients. We now consider three different variations of the rectangular rule integral approximation, see why none of them are optimal in this situation and look at a possible resolution to this numerical issue, inspired by the DCT (cf. Definition 2.4).

Intuitively, the problem is related to the endpoints of the interval $[0, \pi]$, the corresponding areas under the curve we aim to integrate and symmetry. In the following, consider the (near-) simplest case $N = 2$ and $f(x) = x$. The analytical solution is $a_0 = 0$ and $a_1 = 1$.

- With Θ_N as in Definition 2.5, consider the issue that

$$a_1 = \frac{2}{\pi}\int_0^\pi \cos^2(\theta)\mathrm{d}\theta \approx \frac{2}{N}\sum_{j=0}^{N-1}\cos^2\left(\frac{j\pi}{N-1}\right) = \frac{2}{2}\left(\cos^2(0)+\cos^2(\pi)\right) = 2 \neq 1\,.$$

- Instead sampling at $\left\{\cos\left(\frac{j\pi}{N}\right) \;\middle|\; j = 0, ..., N-1\right\}$,

$$a_0 = \frac{1}{\pi}\int_0^\pi \cos(\theta)\mathrm{d}\theta \approx \frac{1}{N}\sum_{j=0}^{N-1}\cos\left(\frac{j\pi}{N}\right) = \frac{1}{2}\left(\cos(0)+\cos(\pi/2)\right) = \frac{1}{2} \neq 0\,.$$

- Even for a creative $\left\{\cos\left(\frac{j\pi}{N+1}\right) \;\middle|\; j = 1, ..., N\right\}$,

$$a_1 \approx \frac{2}{N}\sum_{j=1}^{N}\cos^2\left(\frac{j\pi}{N+1}\right) = \frac{2}{2}\left(\cos^2(\pi/3)+\cos^2(2\pi/3)\right) = \frac{1}{2} \neq 1\,.$$

Of course, the above approximations for a_0 and a_1 are exactly what they claim to be, approximations. And indeed, for larger and larger N, they will converge to the correct coefficients according to Theorem 2.3. However, we are not only interested in cases where $N \gg 1$ but also a smaller number of interpolation points.

Similarly, as a small extension to the rectangular integral rule, one can consider the trapezoidal rule which fixes the issue with $N = 2$, but has similar accuracy issues for larger N.

So we cannot use normal chebpoints for this interpolation method. One way to fix this issue is to use *modified chebpoints* (Definition 2.6) as in the DCT-II, leveraging half-way points (Benjamin et al. 2010).

2.6 Definition: Modified Chebyshev points

$$\tilde{\Theta}_N := \left\{\theta_j := \frac{(j+\frac{1}{2})\pi}{N-1} \;\middle|\; j = 0, ..., N-1\right\},$$
$$\tilde{X}_N := \{x_j := \cos(\theta_j) \mid \theta_j \in \Theta_N\}\,.$$

Our implementation, `TschebFun::interpolantThrough()`, uses these modified chebpoints to circumvent the above issues. In short, the half-way point integral approximation is more accurate in this case and, for small N, gives the exact coefficients as expected. More details can be found in Benjamin et al. 2010.

$$a_k = \frac{2}{N}\sum_{j=0}^{N-1} f(x_j)\cos(k\theta_j), \quad \theta_j \in \tilde{\Theta}_N, \quad x_j = \cos(\theta_j), \quad k > 0\,. \tag{2}$$

2.4 The Interpolation Algorithm

Implementing the rectangular integral rule, TschebFun begins by computing a_0 as the sum of all y, normalised by $\frac{1}{N}$. We compute the remaining coefficients in a for-loop, directly according to Equation (2). The input, $\boldsymbol{y} \in \mathbb{R}^N$ is a vector of function values sampled at the modified Chebyshev points $\tilde{X}_N$.

```
TschebFun TschebFun::interpolantThrough(Vector y) {
  int order = y.size(), degree = order - 1;
  Vector j = modifiedEquipoints(order);
  Vector coeffs = xt::zeros_like(y); // as many coefficients as data points
  coeffs[0] = xt::sum(y)() / order;
  for (size_t k = 1; k < order; k++)
    coeffs[k] = (2.0 / order) * xt::sum(y * xt::cos(j * k))();
  return TschebFun(coeffs);
}
```

Note that the above algorithm can also be expressed as the dot-product of a matrix, a Pseudo-Vandermonde matrix $V \in \mathbb{R}^{N\times N}$ in the Chebyshev basis, with the vector $\boldsymbol{y}$.

$$\boldsymbol{a} = V\boldsymbol{y} = \frac{1}{N}\begin{pmatrix} T_0(x_0) & T_0(x_1) & \cdots & T_0(x_N) \\ 2T_1(x_0) & 2T_1(x_1) & \cdots & 2T_1(x_N) \\ \vdots & \vdots & \ddots & \vdots \\ 2T_N(x_0) & 2T_N(x_1) & \cdots & 2T_N(x_N) \end{pmatrix}\begin{pmatrix} f(x_0) \\ f(x_1) \\ \vdots \\ f(x_N) \end{pmatrix}, \quad x_j \in \tilde{X}_N\,.$$

3 The Spectral Method

Now that we have the most important tool in our belt, Chebyshev interpolation, we can proceed to solve our original problem as given on page 1.

3.1 Forward Euler

As mentioned earlier, we will approximate the time-derivative using the Forward Euler numerical scheme. The numerical solution u of our differential equation then fulfills

$$\frac{\partial u^{(t)}}{\partial t} \approx \frac{u^{(t+\Delta t)} - u^{(t)}}{\Delta t} \quad \Leftrightarrow \quad u^{(t+\Delta t)} \approx u^{(t)} + \Delta t\frac{\partial u^{(t)}}{\partial t} = u^{(t)} + \alpha\Delta t\frac{\partial^2 u^{(t)}}{\partial x^2}\,,$$

which provides us with an explicit relation for the next time-level $U^{(t+\Delta t)}$ given the previous one. The numerical accuracy of this scheme is only $\mathcal{O}(\Delta t)$. Approaches to improve this would be by using, for instance, linear multistep methods.

So in our case, assuming the existence of a differentiation operator $\mathcal{D}_N : \mathbb{T}_N \mapsto \mathbb{T}_N$, the solution function $u^{(t)} \in \mathbb{T}_N$ at time t evolves as follows:

$$u^{(t+\Delta t)} = u^{(t)} + \alpha \Delta t \cdot \mathcal{D}_N^2 u^{(t)} \,.$$

Adaptive time-steps An optional optimisation to Forward-Euler that was implemented in HeatFun as well, is an adaptive time-step that prevents coefficient explosion by adjusting the time-step Δt after each iteration by

$$(\Delta t)_{\text{next}} = \left(\frac{\delta}{\|\boldsymbol{a}^{(t+\Delta t)} - \boldsymbol{a}^{(t)}\|_1} \right)^{\frac{1}{5}} \cdot \Delta t \,,$$

with 'goal' parameter $\delta \in \mathbb{R}$. The effect of this is simple: if the change (as measured by the 1-norm $\|\cdot\|_1$) between two iterations is larger than δ, the time step will be reduced. Otherwise, increased. δ can be adjusted in the graphical user interface upon activation of the optimisation.

3.2 Differentiation in the Spatial Direction

So now, our aim is to find an explicit relation between the coefficients of the derivative of a function $u^{(t)} \in \mathbb{T}_N$ and the original coefficients. As differentiation is a linear operation, which can be especially well understood from this very fact, there exists a matrix $D_N \in \mathbb{R}^{N-1 \times N}$ corresponding to the differentiation operation $\mathcal{D}_N$.

$$\text{For any } u \in \mathbb{T}_N,\ u(x) = \sum_{k=0}^{N-1} a_k T_k(x), \quad u' = \mathcal{D}_N u \ \Leftrightarrow\ \boldsymbol{a}' = D_N \boldsymbol{a} \,.$$

The explicit form of this differentiation matrix can be found in chapter 6 of Trefethen 2000. Our approach shall be a bit different, the algorithmic implementation can be made more space- and time-efficient by performing an iteration over the coefficients directly.

In TschebFun, it is implemented as follows, as adapted from Harris et al. 2020:

```
TschebFun TschebFun::derivative() {
  int n = coefficients.size();
  n = n - 1; // differentiation reduces the order by 1
  Vector coeffs = coefficients; // make a copy
  Vector derivative = xt::zeros<double>({n});
  for (size_t j = n; j > 2; j--) {
    derivative[j - 1] = (2 * j) * coeffs[j];
    coeffs[j - 2] += (j * coeffs[j]) / (j - 2);
```

```
  }
  if (n > 1)
    derivative[1] = 4 * coeffs[2];
  derivative[0] = coeffs[1];
  return TschebFun(derivative);
}
```

The next step is to make sure that the newly obtained $u^{(t+\Delta t)}$ fulfills the boundary conditions $u^{(t)}(-1) = l$ and $u^{(t)}(1) = r$ as well!

3.3 Enforcing Boundary Conditions

One way of forcing the boundary conditions, at least the first that came to mind when thinking of this issue, is to pin down the two highest-order coefficients, who are untouched by the differential equation (differentiating a polynomial twice reduces its degree by two), in the series representation after the iteration.

Let $l := u_0(-1)$, $r := u_0(1)$. Recognise that

$$T_k(-1) = T_k(\cos\pi) = \cos(k\pi) = (-1)^k$$
$$T_k(1) = T_k(\cos 0) = \cos(k0) = 1$$

which leads to

$$\begin{aligned} u(-1,t) = \sum_{k=0}^{N-1} a_k^{(t)} T_k(-1) &= \overbrace{\sum_{k=0}^{N-3} a_k^{(t)}(-1)^k}^{:=\Sigma_1} + (-1)^{N-2} a_{N-2} + (-1)^{N-1} a_{N-1} &= l \\ u(1,t) = \sum_{k=0}^{N-1} a_k^{(t)} T_k(1) &= \underbrace{\sum_{k=0}^{N-3} a_k^{(t)}}_{:=\Sigma_2} + a_{N-2} + a_{N-1} &= r \end{aligned}$$

By adding up the above two equations, one obtains

$$\Sigma_1 + \Sigma_2 + \underbrace{\left((-1)^{N-2}+1\right)}_{\in\{0,2\}} a_{N-2} + \underbrace{\left((-1)^{N-1}+1\right)}_{\in\{0,2\}} a_{N-1} = l + r \tag{3}$$

For N even: Equation (3) has one unkown $a_{N-2} = \frac{l+r-\Sigma_1-\Sigma_2}{2}$, $a_{N-1} = r - a_{N-2} - \Sigma_2$.

For N odd: Equation (3) has one unkown $a_{N-1} = \frac{l+r-\Sigma_1-\Sigma_2}{2}$, $a_{N-2} = r - a_{N-1} - \Sigma_2$.

We apply this coefficient modification once after each iteration to stay consistent with given boundary conditions. We are now ready for the main solver!

3.4 The Spectral Heat Equation Solver

Combining the above insights and algorithms, we finally arrive at the full solver:

Algorithm 1: Our final solving algorithm

```
input: u_0 ∈ ℝ^N, s.t. {u_0}_j = u_0(x_j) ∀x_j ∈ X̃_N, final time T ∈ ℝ^+.
output: u^(T) ∈ 𝕋_N.

let u^(0) = TschebFun::interpolantThrough(u_0).
for (t = 0; t ≤ T; t += Δt); do
  Behaviour according to the differential equation:
  set u^(t+Δt) = u^(t) + αΔt · D_N^2 u^(t) using TschebFun::derivative() twice.

  Enforce boundary conditions on u^(t+Δt):
  let Σ_1 = Σ_{k=0}^{N-3} a_k^(t+Δt) (-1)^k.
  let Σ_2 = Σ_{k=0}^{N-3} a_k^(t+Δt).
  if N even; do
    set a_{N-2}^(t+Δt) = (l+r-Σ_1-Σ_2)/2.
    set a_{N-1}^(t+Δt) = r - a_{N-2}^(t+Δt) - Σ_2.
  else if N odd; do
    set a_{N-1}^(t+Δt) = (l+r-Σ_1-Σ_2)/2.
    set a_{N-2}^(t+Δt) = r - a_{N-1}^(t+Δt) - Σ_2.
  end

  return u^(T), which can be evaluated using TschebFun::evaluateOn().
end
```

Perhaps simpler, more compact and including adaptive time-steps, the core part of the above algorithm is implemented in C++ as

```
void HeatSolver::iterate() {
  // base step, leaves two degrees of freedom a_{N}, a_{N-1}
  TschebFun previousU = currentU;
  currentU = previousU + previousU.derivative().derivative() * (dt * alpha);
  forceBoundaryConditions();
  totalTime += dt;

  if (optimisations.adaptiveDt) {
    double change = xt::sum(xt::abs(currentU.coefficients -
↪   previousU.coefficients))();
    dt *= pow(optimisations.adaptiveGoalDu / change, 0.2);
  }
}
```

4 Analysis, Comparison and Results

The implementation is complete and we are ready to look at our results. One more thing is missing though, given the coefficients $\boldsymbol{a}^{(T)}$ we also want to evaluate the Chebyshev series $u^{(T)} \in \mathbb{T}_N$ on some $x \subset [-1, 1]$!

4.1 Evaluation: The Clenshaw Algorithm

For this purpose, let us recall the recurrence relation we proved in the very beginning, Theorem 2.1. It turns out that it follows the more general Clenshaw recurrence form, whom the algorithm also borrows its name from (Press et al. 1987, pp. 172–178).

4.1 Theorem: Clenshaw recurrence relation

If $f(x) = \sum_{k=0}^{N} a_k F_k(x)$ and there exists a three-term recursion of the form

$$F_{n+1}(x) = \alpha(n, x)F_n(x) + \beta(n, x)F_{n-1}(x), \quad F_n : \mathbb{R} \mapsto \mathbb{R}, \quad \alpha, \beta \in \mathbb{R}\,,$$

then the value $f(x)$ can be computed by iteratively updating

$$y_k = \alpha(k, x)y_{k+1} + \beta(k + 1, x)y_{k+2} + a_k \quad \text{with start } y_{N+2} = y_{N+1} = 0\,,$$

to finally arrive at $f(x) = a_0F_0(x) + y_1F_1(x) + \beta(1, x)F_0(x)y_2$.

From Theorem 2.1 we identify $\alpha(n, x) = 2x$ and $\beta(n, x) = -1$ to arrive at the following special case for Chebyshev series:

```
Vector TschebFun::evaluateOn(Vector x) {
  Vector U_kp2;
  Vector U_kp1 = xt::zeros_like(x);
  Vector U_k = xt::ones_like(x) * coefficients[coefficients.size() - 1];
  for (int k = coefficients.size() - 2; k >= 0; k--) {
    U_kp2 = U_kp1;
    U_kp1 = U_k;
    U_k = 2.0 * x * U_kp1 - U_kp2 + coefficients[k];
  }
  return (U_k - U_kp2 + coefficients[0]) / 2.0;
}
```

Essentially, the idea is simple: Using the recurrence relation (Theorem 2.1) to our advantage and tracking the sum of $a_kT_k(x)$ on the way from $k = N - 2$ to $k = 0$, which is why we call it the backward Clenshaw recurrence. The above implementation can work with vector-valued $\boldsymbol{x}$.

4.2 Analysis: Interface to Python

Another exciting aspect of HeatFun is its extension to a compiled Python module using `pybind11`[2], which allows any user to leverage its functionality along with convenient Python & NumPy!

```python
import heatfun, numpy as np
u0 = lambda x: np.exp(-12 * x**2)
x_of_interest = np.linspace(-1.0, 1.0, 500)
cheb_x = heatfun.modifiedChebpoints(30)
solution = heatfun.solve(u0(cheb_x), 0.01, x_of_interest)
```

4.3 Comparison: Chebfun

In order to verify the numerical results, we compare our solution with the one that the amazing Chebfun provides by means of the following script (Trefethen, Birkisson and Driscoll 2018):

```
% example adapted from 'Exploring ODEs', page 282
pdefun = @(t, x, u) diff(u, 2);
bc.left = @(t, u) u;
bc.right = @(t, u) u;
opts = pdeset('plot', 'off');
[t, u] = pde15s(pdefun, [0 0.005 0.010], u0, bc, opts);

x = linspace(-1.0, 1.0, 500).';
all_outputs = u(x);
output = all_outputs(:, end);
dlmwrite(filename, output, 'precision', '%.16f')
```

Chebfun provides a lot more versatile functionality for ODEs, but also PDEs, than demonstrated here. Especially, it does not use Forward Euler to step through time, as HeatFun does.

[2] https://pybind11.readthedocs.io/

4.4 Results: On Target?

This comparison yields the following results for a few interesting functions:

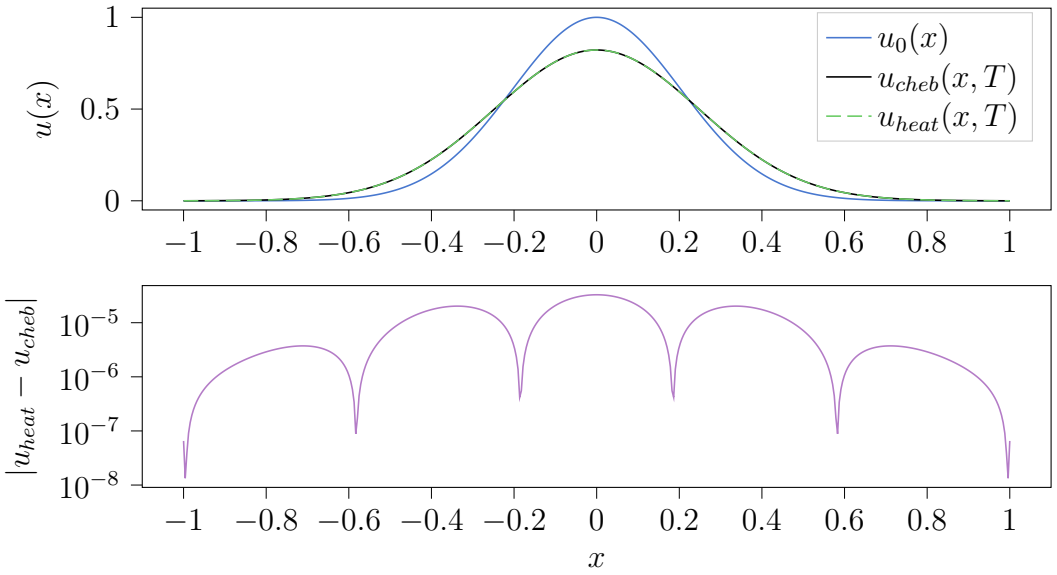

Figure 3: Comparison of HeatFun and Chebfun for solving the stated heat equation problem with $\alpha = 1$, $T = 0.01$, $N = 30$ and initial condition $u_0(x) = \mathrm{e}^{-12x^2}$. The squared error between the two functions, evaluated in 500 points, is 8.6×10^{-8}.

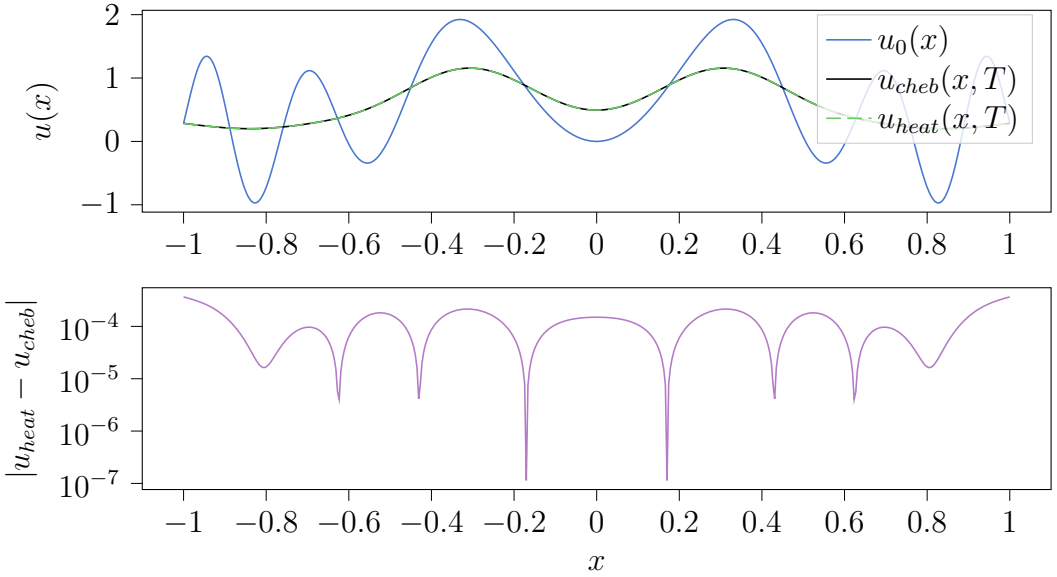

Figure 4: Comparison of HeatFun and Chebfun for solving the stated heat equation problem with $\alpha = 1$, $T = 0.01$, $N = 30$ and initial condition $u_0(x) = \sin((4x)^2) + \sin(4x)^2$. The squared error between the two functions, evaluated in 500 points, is 1.01×10^{-5}.

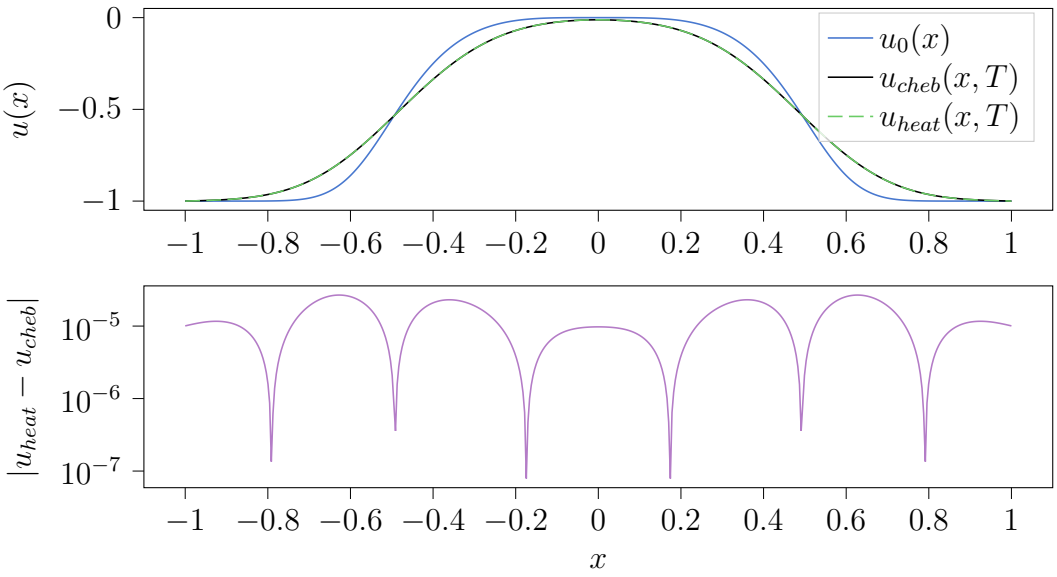

Figure 5: Comparison of HeatFun and Chebfun for solving the stated heat equation problem with $\alpha = 1$, $T = 0.01$, $N = 30$ and initial condition $u_0(x) = \tanh(-10x^4)$. The squared error between the two functions, evaluated in 500 points, is 1.06×10^{-7}.

5 Discussion and Outlook

Generally, the solver works pretty well and efficiently in our given setting! The interpolation procedure, although kept simple, produces coefficients that match those returned by Chebfun in MatLab up to 10 decimal places, at least for simple functions. Of course, the interpolation accuracy is crucial for the accuracy of the end result. If even $u_0 \in \mathbb{T}_N$, then the approximation is exact and the error is only inherited from Forward Euler, as the spatial derivative is exact (up to numerical round-off).

Some initial conditions, and especially high-degree Chebyshev series (i.e. high N) cause numerical issues with Forward Euler. This is because the derivative of a polynomial simply gains higher and higher pre-factors for growing degrees of the monomials. The effect is then amplified by the boundary conditions forced onto the highest-degree coefficients, sometimes leading to fairly high coefficients that will ruin our next iterate. The upside is that this issue can always be remedied by reducing the time-step (ultimately, this was the reason for adding the adaptive time-step optimisation).

Given more time, one should implement a linear multistep method instead of simple Forward Euler. Given even more time, it would be interesting to compare the three mentioned coefficient computation methods listed in Section 2.2 and look at the potential speed-up facilitated by the FFT. Following from there, another important

aspect would be to find a good way of quantifying the interpolation accuracy for *any* input using the method implemented in TschebFun, and analyse numerical stability in a detailed fashion.

References

Benjamin, Arthur T., Larry Ericksen, Pallavi Jayawant and Mark Shattuck (2010). 'Combinatorial trigonometry with Chebyshev polynomials'. In: *Journal of Statistical Planning and Inference* 140, pp. 2157–2160.

Cooley, James W. and John W. Tukey (1965). 'An algorithm for the machine calculation of complex Fourier series'. In: *Mathematics of Computation* 19, pp. 297–301. DOI: 10.2307/2003354.

Harris, Charles R. et al. (Sept. 2020). 'Array programming with NumPy'. In: *Nature* 585.7825, pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

Press, William H., Saul A. Teukolsky, William T. Vetterling and Brian P. Flannery (1987). *Numerical Recipes in FORTRAN - The Art of Scientific Computing*. 2nd ed.

Runge, Carl (1901). 'Über empirische Funktionen und die Interpolation zwischen äquidistanten Ordinaten'. In: *Zeitschrift für Mathematik und Physik* 46.224-243, p. 20.

Trefethen, Lloyd N. (2000). *Spectral Methods in MATLAB*. Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9780898719598. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898719598.

– (2019). *Approximation Theory and Approximation Practice, Extended Edition*. Philadelphia, PA: Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9781611975949. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611975949.

Trefethen, Lloyd N., Asgeir Birkisson and Tobin A. Driscoll (2018). 'Exploring ODEs'. In: Society for Industrial and Applied Mathematics. URL: http://people.maths.ox.ac.uk/trefethen/Exploring.pdf (visited on 24/12/2022).

A Some More Functions

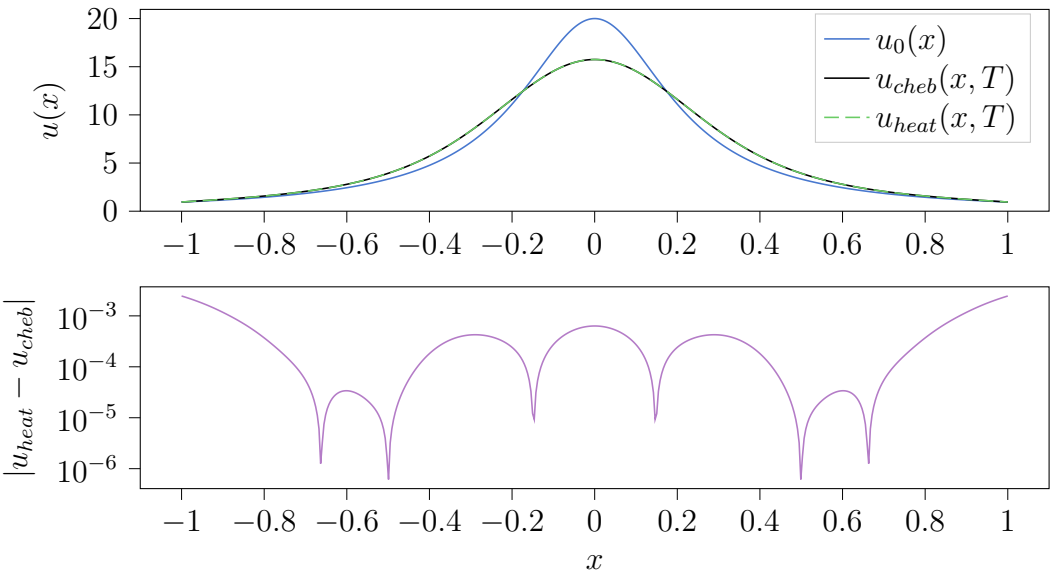

Figure 6: Comparison of HeatFun and Chebfun for solving the stated heat equation problem with $\alpha = 1$, $T = 0.01$, $N = 30$ and initial condition $u_0(x) = \tanh(-10x^4)$. The squared error between the two functions, evaluated in 500 points, is 2.31×10^{-4}.

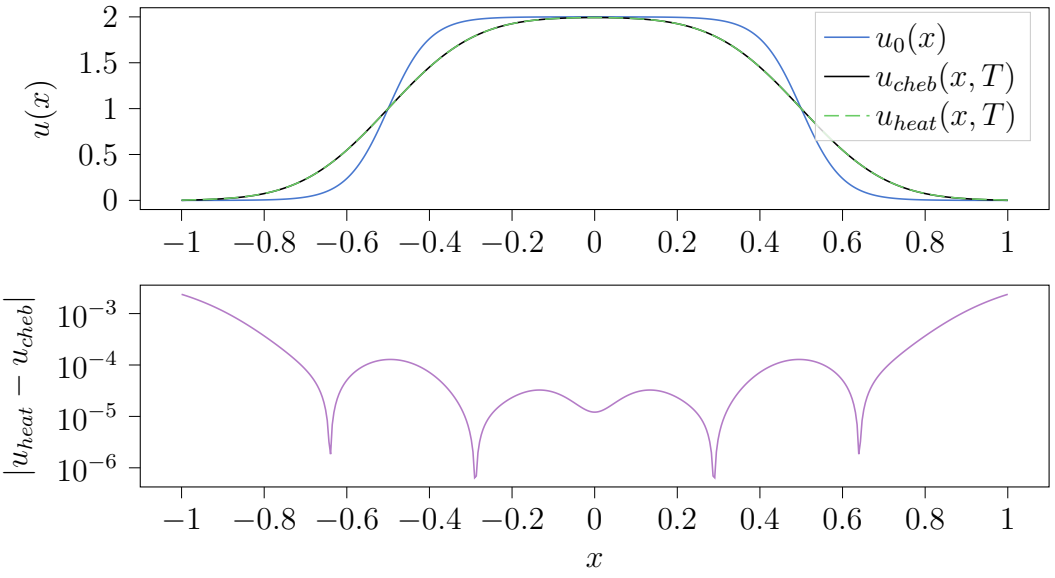

Figure 7: Comparison of HeatFun and Chebfun for solving the stated heat equation problem with $\alpha = 1$, $T = 0.01$, $N = 30$ and initial condition $u_0(x) = \tanh(-10x^4)$. The squared error between the two functions, evaluated in 500 points, is 1.9×10^{-4}.

B TschebFun's Class Definition

To make the capabilities and perhaps some internal details of TschebFun more clear, here is its header file.

```
#pragma once

#include <xtensor/xarray.hpp>
#include <xtensor/xindex_view.hpp>
#include <xtensor/xview.hpp>

typedef xt::xarray<double> Vector;

class TschebFun {
 public:
  xt::xarray<double> coefficients;

 public:
  TschebFun(Vector coeffs);
  size_t order() { return coefficients.size(); };
  size_t degree() { return coefficients.size() - 1; };
  Vector evaluateOn(Vector x);
  TschebFun derivative();
  static TschebFun interpolantThrough(Vector y);

  static Vector equipoints(size_t N);
  static Vector chebpoints(size_t N);
  static Vector modifiedEquipoints(size_t N);
  static Vector modifiedChebpoints(size_t N);

  TschebFun operator+(const TschebFun &other);
  TschebFun operator-(const TschebFun &other);
  TschebFun operator*(const double &factor);
};
```