

Numerical Analysis in Boost.Math

Nick Thompson

“Numerical analysis is the study of algorithms for solving the problems of continuous mathematics”

-- Lloyd N. Trefethen

Topics in Numerical Analysis

- ☞ Numerical Integration/Differentiation
- ☞ Interpolation
- ☞ Signal denoising and compression
- ☞ Root finding/optimization
- ☞ ODE/PDE solvers
- ☞ Numerical linear algebra (GMRES/CG/QR)

Boost has tools for many of these

. . . but we will discuss only new features of Boost.Math, landing in 1.66

First, a thanks

John Maddock maintains [Boost.Math](#).

This is a heroic effort, and Boost.Math would die without him.

The features I added would have been stillborn without John's help.

“It’s always been ‘We can’t do it that way. It would be too slow.’ You know what’s slow? Spending all day trying to figure out why it doesn’t work. That’s slow. That’s the slowest thing I know.”

-- Paul Phillips

Goals of Boost.Math

- 1) Accuracy
- 2) Stability
- 3) Optimal asymptotic complexity
- 4) Speed

Unfortunate fact

Every numerical algorithm an undergrad can understand breaks in production

Examples

- ☒ Lagrange interpolation is ill-conditioned
- ☒ Gaussian elimination is ill-conditioned w/o pivoting
- ☒ Cubic spline interpolation is ill-conditioned
- ☒ Numerical differentiation by finite differences is ill-conditioned
- ☒ Jacobi iteration too slow
- ☒ Euler's method for ODEs too slow
- ☒ Newton-Cotes quadrature too slow

Reliability Model

HPC calcs: 100 nodes, 64 threads/node, 2 week compute => $\sim 10^{10}$ seconds of compute

Assuming each function call (at the top of the callstack) takes $1\mu\text{s}$, then there are $\sim 10^{16}$ function calls in the average HPC task

Reliability Model

If any of these $n \approx 10^{16}$ calls segfaults, leaks/uses too much memory, or leaks an abstraction, the whole job dies, or the result is worthless.

Let p denote the probability of failure for each function call.
Assuming independence

$$P(\text{Job succeeds}) = \binom{n}{0} (p)^{n-n} (1-p)^n = (1-p)^n$$

Reliability Model

On gcloud, this example computation costs ~\$75,000.
What is your tolerance for job failure?

Assume 1% can fail. Then we must have

$$p_{\text{fail}} \leq 1 - \exp(\log(.99)/10^{16}) \approx 10^{-18}$$

Reliability Model

HPC codes cannot segfault, leak memory, or break abstractions, except with vanishing probability.

If the algorithms in boost.math weren't advanced, they'd be useless.

Interpolation

Given a function known at points, you want to know its value at points between the known points.

Examples:

- ☞ Create a graph from a list of data points.
- ☞ Find values between points of a numerical solution to an ODE
- ☞ "Create" data between sensors

Two types of interpolation

Equispaced: Data is taken every millisecond

Irregular: Data is known wherever a sensor is placed

Equispaced Interpolation has a bad
reputation

because complex analytic basis functions + equispaced data =
catastrophic ill-conditioning

Equispaced interpolation can be fast and stable

using basis functions of compact support

Building smooth basis functions of compact support

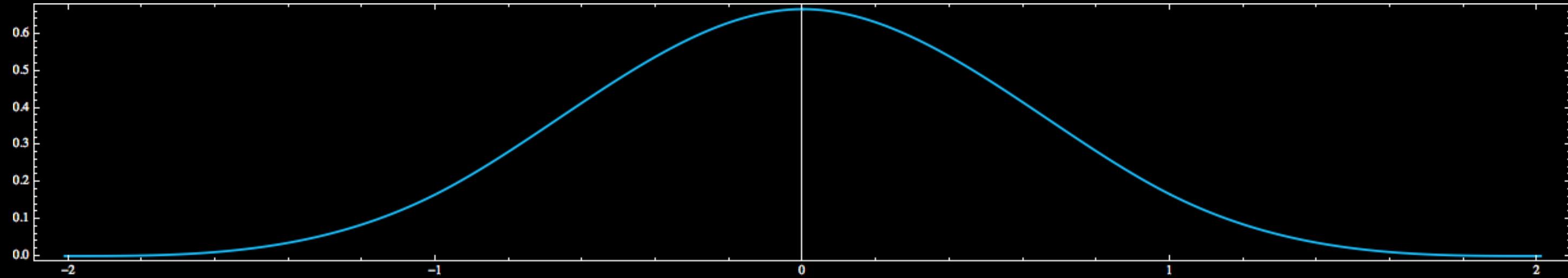
Start with a box function

$$B_0(x) := \begin{cases} 1 & |x| \leq 1/2 \\ 0 & |x| > 1/2 \end{cases}$$

and smooth with integration

$$B_{m+1}(x) := \int_{x-1/2}^{x+1/2} B_m(t) dt$$

$$B_3(x)$$



$$B_3 \in C_0^2([-2, 2])$$

Build an interpolant with B_3 using
translates and scalings

Define $B_{m,k}(x) := B_m\left(\frac{x - a - hk}{h}\right)$ and write

$$s(x) = \sum_{k=-1}^n \alpha_k B_{3,k}(x)$$

Satisfying the interpolation condition $s(x_j) = y_j$ requires solving a tridiagonal linear system.

Equispaced interpolation via cubic_b_spline

```
#include <boost/math/interpolators/cubic_b_spline.hpp>
using boost::math::cubic_b_spline;
std::vector<double> v(n);
// initialize v ...
double t0 = 0; // initial time
double h = 0.01; // spacing between points
cubic_b_spline<double> spline(f.begin(), f.end(), t0, h);
// Interpolate:
double y = spline(0.3);
// Interpolate derivative:
double yp = spline.prime(0.3);
```

cubic_b_spline complexity

$\mathcal{O}(n)$ constructor, $\mathcal{O}(1)$ evaluation

cubic_b_spline accuracy

$$f \in C^4([a, b]) \implies \|f - s\|_{\infty} \leq \frac{h^4}{16} \|f^{(4)}\|_{\infty}$$

Irregular interpolation via barycentric_rational

Given a list of unequally spaced points $x_0 < x_1 < \dots < x_{n-1}$ and a list of values $f(x_0), f(x_1), \dots, f(x_{n-1})$
we wish to construct a rational function r such that $r(x_j) = f(x_j)$

Rational functions

can be written as

$$r(x) = \frac{\sum_{j=0}^n a_j x^j}{\sum_{k=0}^m b_k x^k}$$

Interpolating with generic rational functions is unwise, as the denominator can be zero, leading to poles in the interpolation domain.

Rational interpolants

with $n = m$ can always be written in the barycentric form

$$r(x) = \frac{\sum_{k=0}^n \frac{w_k}{x-x_k} f(x_k)}{\sum_{k=0}^n \frac{w_k}{x-x_k}}$$

Barycentric weights

Different weights can be chosen to produce interpolants of varying character. For Boost's barycentric_rational, we have

$$w_k := (-1)^{k-d} \sum_{i \in J_k} \prod_{j=i, j \neq k}^{i+d} \frac{1}{|x_k - x_j|}$$

where

$$J_k := \{i : k - d \leq i \leq k \wedge 0 \leq i \leq n - d\}$$

Using barycentric_rational

```
#include <boost/math/interpolators/barycentric_rational.hpp>
using boost::math::barycentric_rational;
std::vector<double> x(n);
std::vector<double> y(n);
// initialize x, y . . .
int approximation_order = 3;
barycentric_rational<double> b(x.begin(), x.end(), y.begin(), approximation_order);
// Interpolate:
double y = b(12.7);
```

barycentric_rational complexity

$\mathcal{O}(n)$ constructor and $\mathcal{O}(n)$ evaluation.

barycentric_rational accuracy and stability

If $h := \max_i(x_{i+1} - x_i)$ then

$$f \in C^{d+2}[a, b] \implies \|f - r\| \leq h^{d+1}(b - a) \frac{\|f^{(d+2)}\|}{d + 2}$$

Special cases have been proven forward stable, but no general proof is known

Chebyshev polynomials

are defined by a three-term recurrence

$$T_0(x) := 1, \quad T_1(x) := x, \quad T_{n+1}(x) := 2xT_n(x) - T_{n-1}(x)$$

For $x \in [-1, 1]$, we can write $x = \cos(\theta)$ and we have the identity $T_n(\cos(\theta)) = \cos(n\theta)$.

Chebyshev polynomials

are orthogonal over the interval $[-1, 1]$ with respect to the weight function $(1 - x^2)^{-1/2}$ so

$$\int_{-1}^1 \frac{T_n(x)T_m(x)}{\sqrt{1 - x^2}} dx = \frac{\pi}{2 - \delta_{m0}} \delta_{mn}$$

Among all polynomials

Chebyshev polynomials have minimal uniform norm deviation from zero on $[-1, 1]$, i.e., $\|T_n\|_{\infty, [-1, 1]} = 1$.

Chebyshev interpolation

Suppose a function f is expensive to evaluate.

By projecting it onto a span of Chebyshev polynomials, we can sacrifice a tiny bit of accuracy and gain massive increases in speed.

In addition, we gain a fast numerical integration scheme, as well as a stable numerical differentiation scheme.

Chebyshev series

Just as a Fourier series is an expansion in complex exponentials, a Chebyshev series is an expansion in Chebyshev polynomials

$$f(\theta) = \sum_{n \in \mathbb{Z}} c_n \exp(in\theta) \quad (\text{Fourier series})$$

$$g(x) = \sum_{n \in \mathbb{N}} a_n T_n(x) \quad (\text{Chebyshev series})$$

Orthogonality

gives us a way of computing the coefficients of the expansion

$$a_m = \frac{(2 - \delta_{m0})}{\pi} \int_0^\pi f(\cos(\theta)) \cos(m\theta) d\theta$$

The integrand is periodic, so trapezoidal quadrature is exponentially convergent.

Trapezoidal quadrature to recover Chebyshev series coefficients

$$\int_0^\pi f(\cos(\theta)) \cos(m\theta) d\theta \approx \frac{\pi}{n} \sum_{k=0}^{n-1} f(\cos((k + 1/2)\pi/n)) \cos(m(k + 1/2)\pi/n)$$

This is a DCT-II for $x_k = f(\cos((k + 1/2)\pi/n))$, and hence all coefficients can be recovered in $\mathcal{O}(n \log n)$ time.

Chebyshev transform in boost.math

```
#include <boost/math/special_functions/chebyshev_transform.hpp>

using boost::math::chebyshev_transform;
auto f = [] (double x) { return sin(x); };

chebyshev_transform<double> cheb(f, 0.0, M_PI);

// Interpolate:
double x = cheb(0.3);
// Integrate over [0, M_PI]
double I = cheb.integrate();
// Differentiate:
double xp = cheb.prime(0.3);
```

Gotchas with Boost.Math's Chebyshev transform

Since we require a DCT-II, we have a dependency on FFTW.

Pull requests to eliminate this dependency are welcome!

(This shouldn't be too bad: We only require a power of two DCT-II)

Quadrature

is the numerical art of evaluating integrals

$$I_a^b[f] := \int_a^b f(x) \, dx \in \mathbb{R}$$

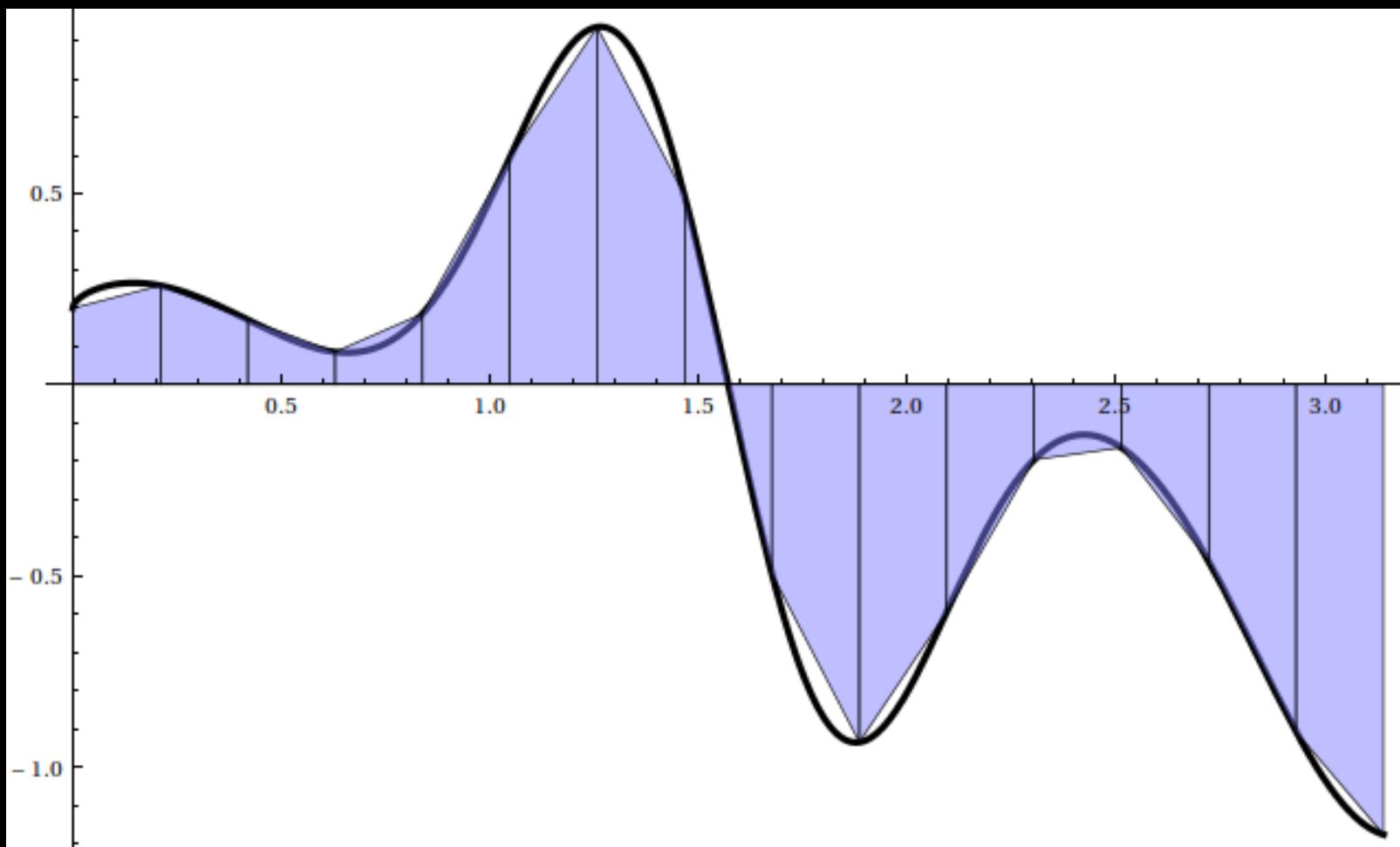
Quadrature

Most integrals do not have "analytic representations", such as

$$\int_{-\infty}^{\infty} e^{-x^2} \, dx = \sqrt{\pi}$$

so numerical quadrature much be used.

Trapezoidal Rule



(Image courtesy of [Jeremy Kun](#))

Trapezoidal Rule

Let $x_k := a + kh$ for $k \in \{0, \dots, n\}$. Then the trapezoidal rule is given by

$$\int_a^b f(x) dx \approx h \left[\frac{f(x_0) + f(x_n)}{2} + \sum_{k=1}^{n-1} f(x_k) \right] =: T_h[f]$$

Trapezoidal Rule

If $f \in C^2[a, b]$ then the error of the trapezoidal rule is

$$|I_a^b[f] - T_h[f]| \leq \frac{b-a}{2} h^2 \|f''\|_\infty$$

The $\mathcal{O}(h^2)$ error implies that this method is too slow.

Euler-Maclaurin Expansion

Let $f \in C^m[a, b]$. Then

$$\int_a^{a+T} f(x) dx = T_h[f] - \sum_{j=1}^{\lceil m/2 \rceil} \frac{b_{2j} h^{2j}}{(2j)!} (f^{(2j-1)}(a+T) - f^{(2j-1)}(a)) + (-1)^m h^m \int_a^{a+T} B_m \left(\frac{x-a}{h} \right) f^{(m)}(x) dx$$

where b_{2j} are the Bernoulli numbers and B_m are the Bernoulli polynomials.

Euler-Maclaurin expansion

If $f \in C^m[a, a+T]$ is periodic, then

$$\int_a^{a+T} f(x) dx = T_h[f] + (-1)^m h^m \int_a^{a+T} B_m \left(\frac{x-a}{h} \right) f^{(m)}(x) dx$$

If f is smooth and periodic, then the trapezoidal rule converges faster than any power of h .

Trapezoidal quadrature in Boost.Math

```
#include <boost/math/quadrature/trapezoidal.hpp>
using boost::math::quadrature::trapezoidal;
auto f = [](double x) { return 1/(5 - 4*cos(x)); };
double I = trapezoidal(f, 0.0, 2*M_PI);
```

Periodic integrands

are a pretty big restriction for exponential convergence.

Could we make a variable transformation that allows exponential convergence for any C^∞ integrand?

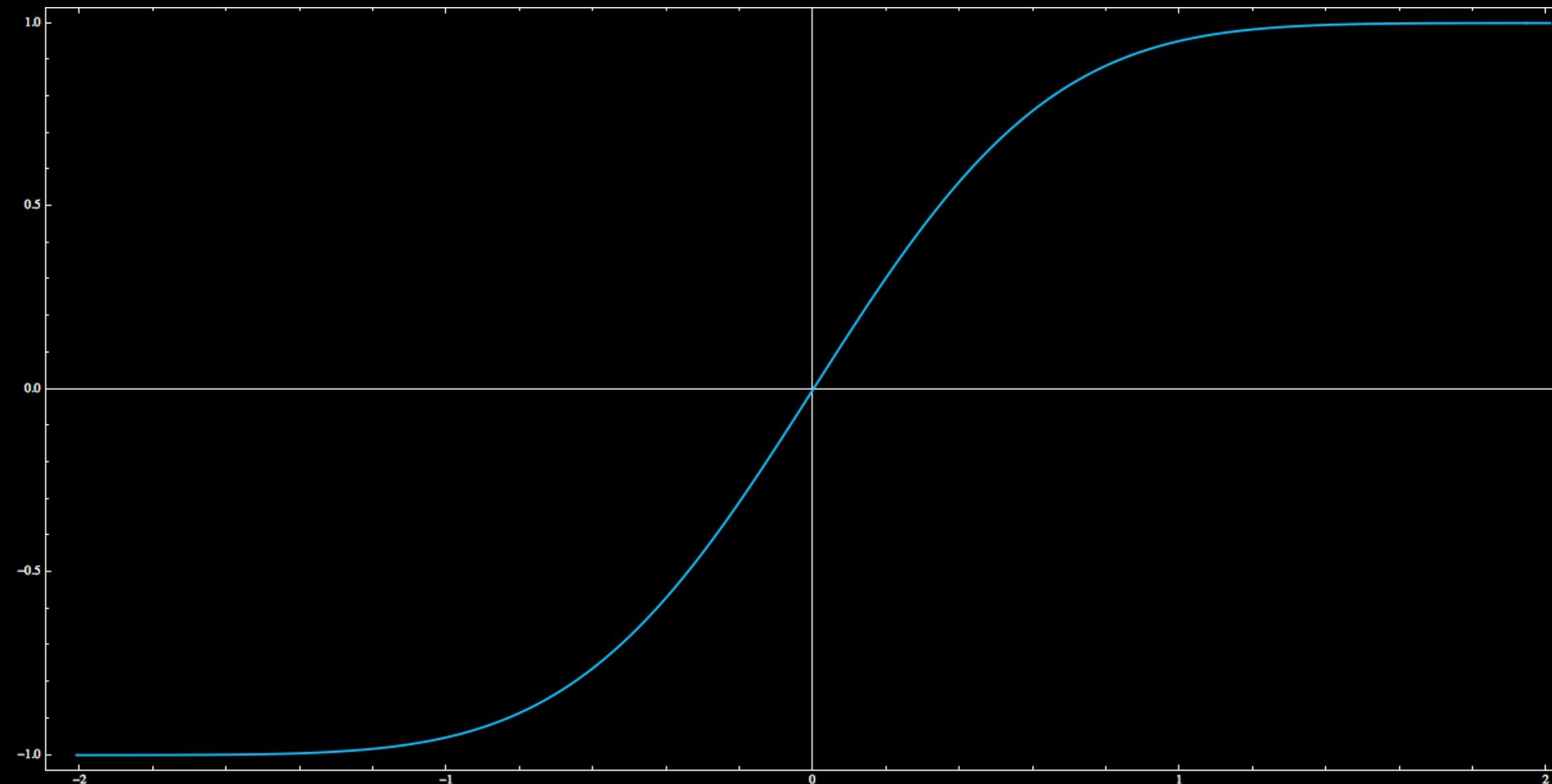
Variable transformations

Let $x = g(t)$. Then

$$\int_a^b f(x) \, dx = \int_{g^{-1}(a)}^{g^{-1}(b)} f(g(t))g'(t) \, dt$$

No one has developed a transform such that $f \circ g \cdot g'$ is periodic on the interval $(g^{-1}(a), g^{-1}(b))$, but it's easy enough the ensure that $(g^{-1}(a), g^{-1}(b)) = (-\infty, \infty) \dots$

$$g(t) := \tanh\left(\frac{\pi}{2}\sinh(t)\right)$$



Tanh-sinh quadrature

Applying the trapezoidal rule to $f \circ g \cdot g'$ with transform
 $g(t) := \tanh\left(\frac{\pi}{2}\sinh(t)\right)$ gives us tanh-sinh quadrature.

For analytic f , this scheme is exponentially convergent.

Tanh-sinh in boost.math

```
#include <boost/math/quadrature/tanh_sinh.hpp>
using boost::math::quadrature::tanh_sinh;
tanh_sinh<double> integrator;
auto f = [] (double x) -> double { return 1/(1 + pow(tan(x), 3)); };
double Q = integrator.integrate(f, (double) 0, half_pi<double>());
```

Gauss-Kronrod Quadrature

`tanh_sinh` is very aggressive about attacking singularities. Sometimes, roundoff error will cause `tanh_sinh` quadrature to evaluate your function at the singularity.

We can integrate functions singular at endpoints using Gauss-Kronrod quadrature.

Gauss-Kronrod quadrature

adds evaluation points to a Gaussian quadrature in such a way that an error estimate is produced along with the value of the integral.

Gauss-Kronrod

```
#include <boost/math/quadrature/gauss_kronrod.hpp>
using boost::math::quadrature::gauss_kronrod;
auto f = [&](double x) -> double { return 1/(1 + pow(tan(x), 3)); };
gauss_kronrod<double, 15> gk15;
double Q = gk15.integrate(f, 0.0, M_PI/2);
```

Send us pull requests! We need

- ☞ Wavelet transforms
- ☞ RiskShrink denoising
- ☞ Sparse grid quadrature
- ☞ Monte Carlo integration (Frank-Wolfe bayesian quadrature)
- ☞ Multivariate interpolation
- ☞ Low dimensional quadrature
- ☞ FFTs