Numerical Methods in Boost.Math

Nick Thompson, bandgap.io, Idaho State

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 1.66/1.67/1.68

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

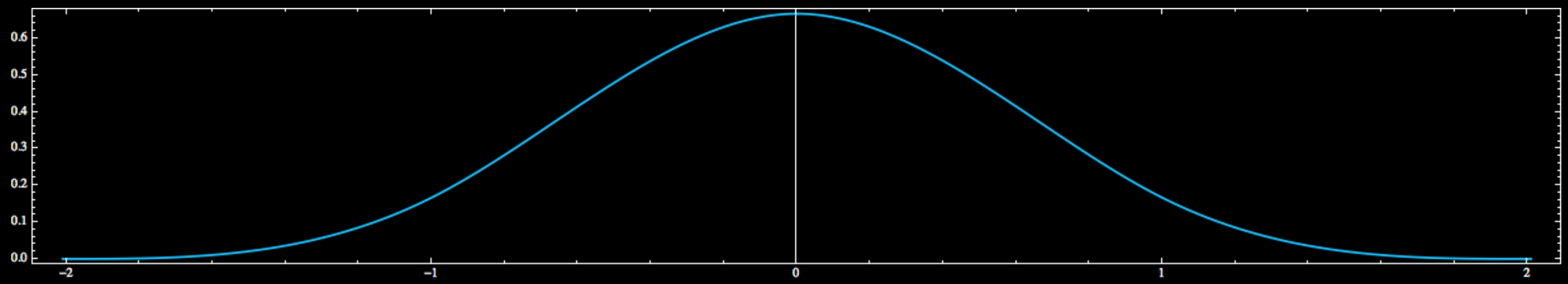
Start with a box function

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

and smooth with integration

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

$B_3(x)$



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

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

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

Irregular interpolation via barycentric_rational

Given a list of unequally spaced points $x_0 < x_1 < \cdots < x_{n-1}$ and a list of values $f(x_0), f(x_1), \ldots, f(x_{n-1})$
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

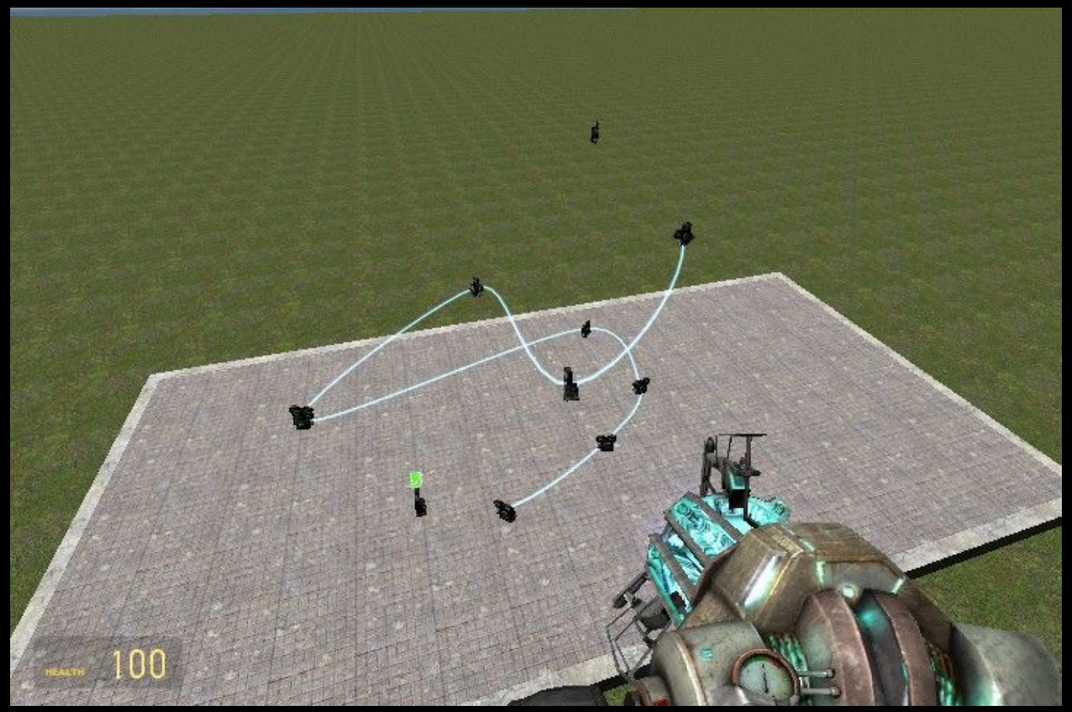
If $h := \max_i (x_{i+1} - x_i)$ then

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

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

Catmull-Rom (lands in 1.68)

In computer graphics, functions need to be interpolated at every pixel. Storing all this data is prohibitive, so we need to interpolate between stored points.



```cpp
#include <boost/math/interpolators/catmull_rom.hpp>
using boost::math::catmull_rom;

std::vector<Point> v(n);
// initialize v then pass it to interpolator:
catmull_rom<Point> cat(v.data(), v.size());

// Interpolate:
auto p = cat(0.01);
// Compute tangent:
auto tangent = cat.prime(0.01);
```

Catmull-Rom complexity

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

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

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

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 Chebyshev polynomials

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

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

Orthogonality

gives us a way of computing the coefficients of the expansion

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

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

Trapezoidal quadrature to recover Chebyshev series coefficients

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

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);
```

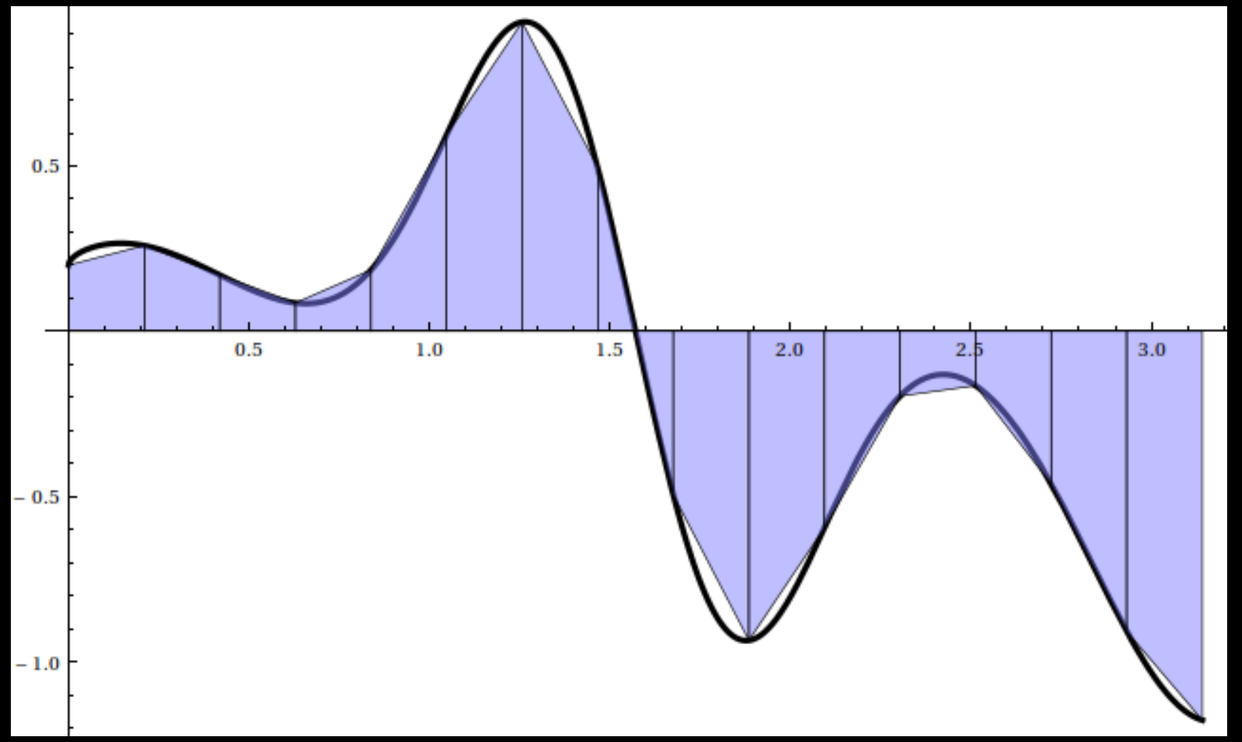
Gotchas with Boost.Math's Chebyshev transform

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)

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

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

Trapezoidal Rule

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

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

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

Euler-Maclaurin Expansion

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

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

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

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

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);
```

Contour integrals

Contour integrals and high-order numerical derivatives can be computed with trapezoidal quadrature, but the contour must be chosen by the user.

Periodic integrands

are a pretty big restriction for exponential convergence.

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

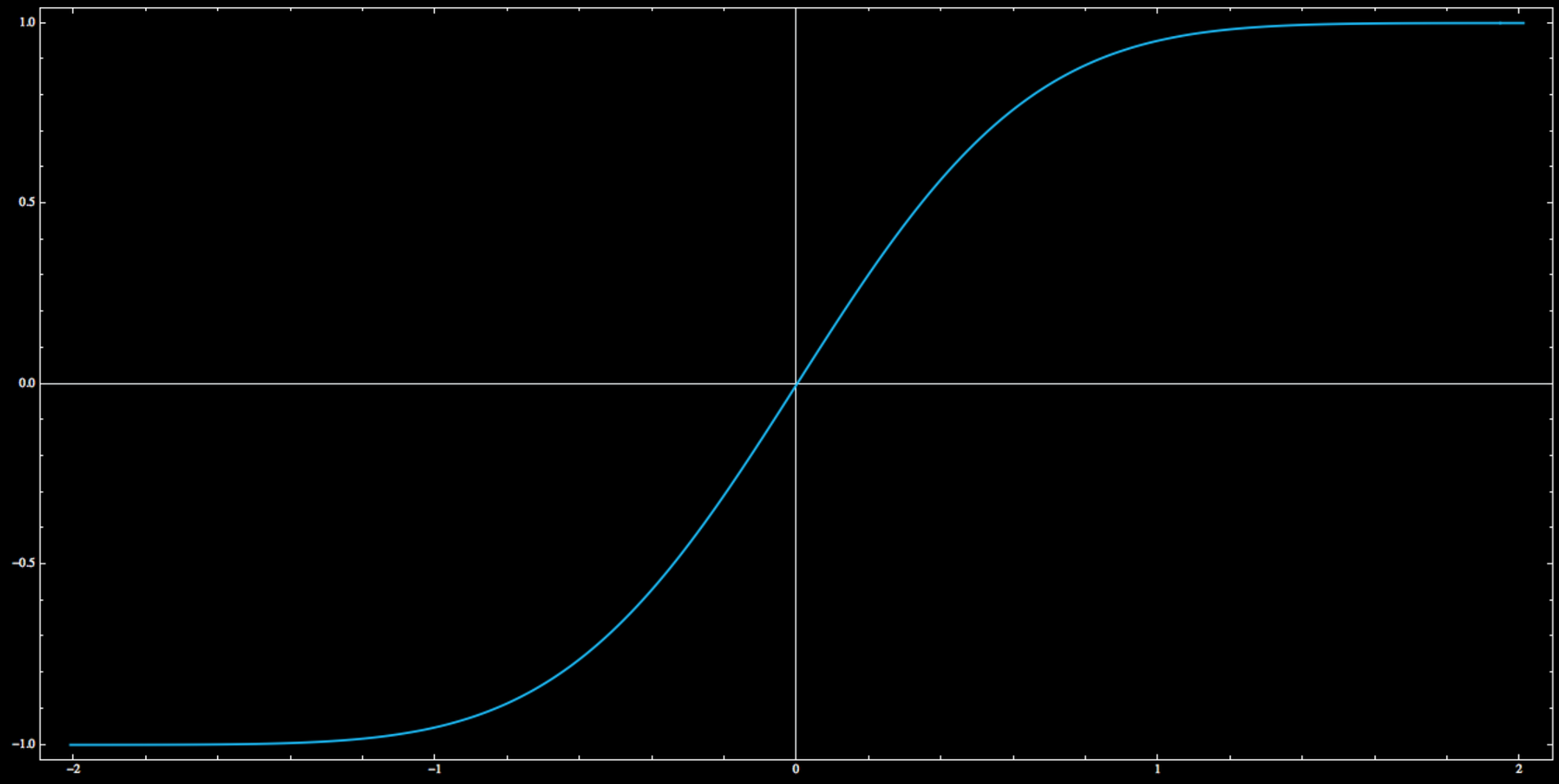
Variable transformations

Let $x = g(t)$. Then

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

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 $(g^{-1}(a), g^{-1}(b)) = (-\infty, \infty)$ . . .

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



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 quadrature

```
#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);
```

Gauss-Kronrod quadrature

For collocation/Nystrom methods, we can extract the nodes and weights via:

```
gauss_kronrod<double, 15> gk15;
auto& w = gk15.weights();
auto& a = gk15.abscissas();
```

Monte-Carlo Integration (lands in 1.67)

All previous quadrature methods are 1D! For integration in very large dimension, we use Monte-Carlo integration

Monte-Carlo Integration (1.67)

```cpp
#include <boost/math/quadrature/naive_monte_carlo.hpp>
// Define a function to integrate. Calls must be threadsafe
auto g = [](std::vector<double> const & x)
{
  return 1.0 / (1.0 - cos(x[0])*cos(x[1])*cos(x[2]));
};
// Bounds can be finite or infinite:
std::vector<std::pair<double, double>> bounds{{0, M_PI}, {0, M_PI}, {0, M_PI}};
double error_goal = 0.001
naive_monte_carlo<double, decltype(g)> mc(g, bounds, error_goal);

std::future<double> task = mc.integrate();
while (task.wait_for(std::chrono::seconds(1)) != std::future_status::ready) {
    display_progress(mc.progress());
    if (some_signal_heard()) {
        mc.cancel();
    }
}
double I = task.get();
```

Monte-Carlo integration

Naive Monte-Carlo integration is very robust.

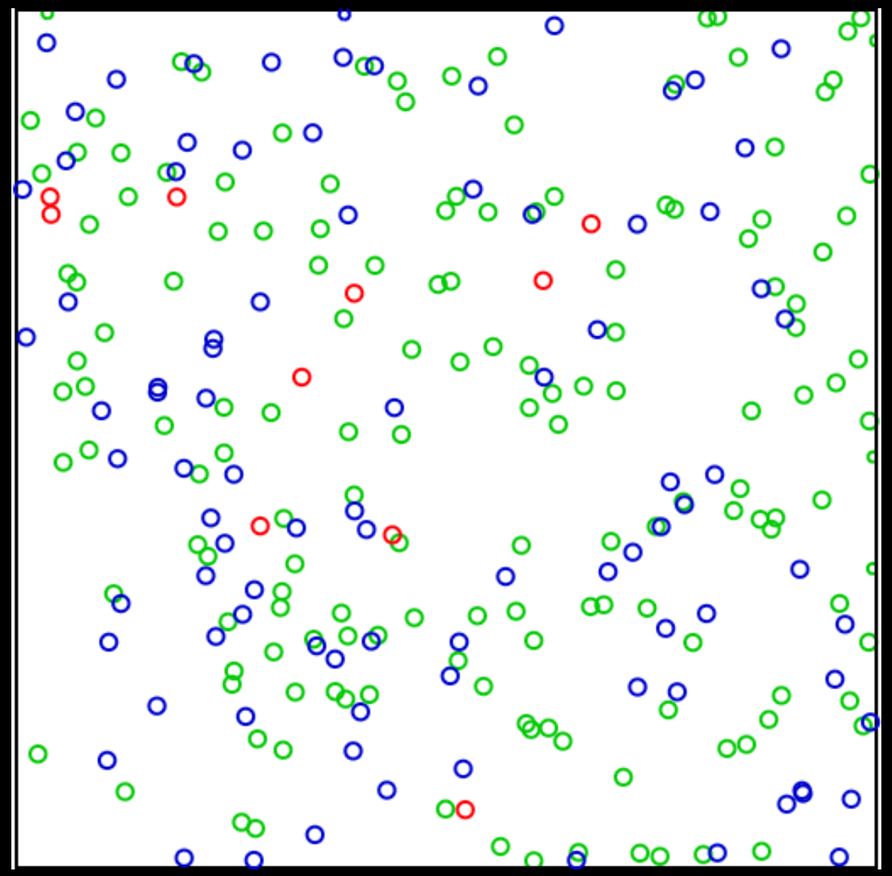
But for n function evaluations, we only get error of order $\sigma/\sqrt{n}$.

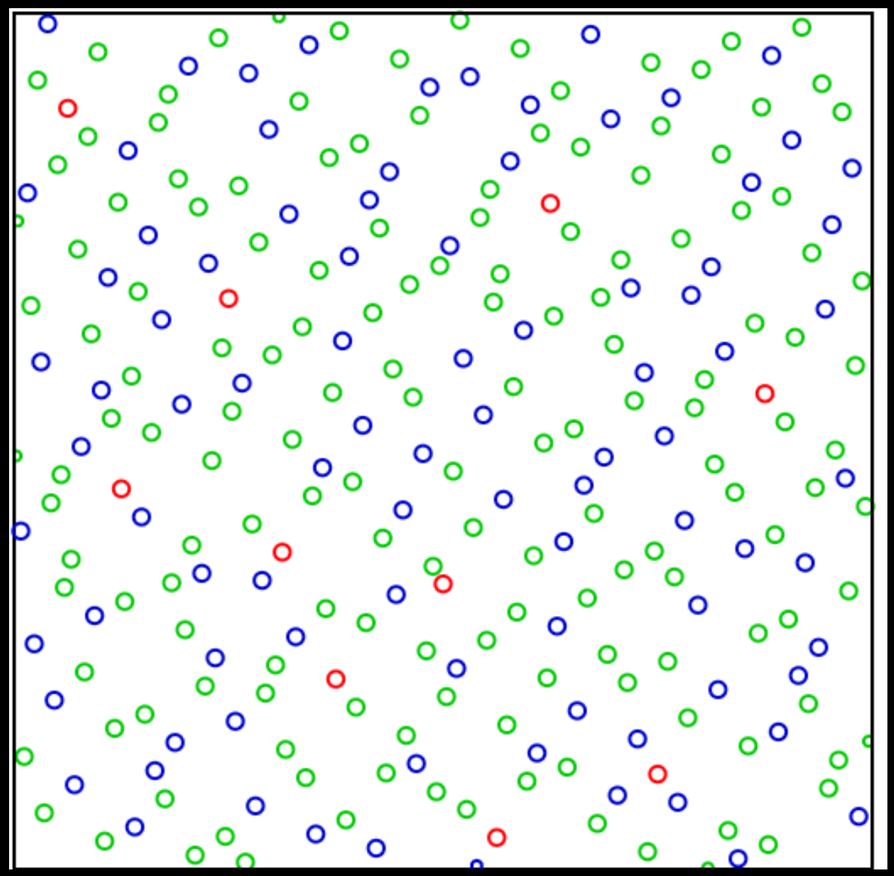
Recovering one more digit takes 100x the time!

Quasi-Monte Carlo integration (lands in 1.68)

uses a "sub"-random sequence for quadrature nodes to improve the convergence rate over quasi-Monte Carlo integration

Random vs Quasi-random sequences





Quasi-random sequences (lands in 1.67)

```
#include <boost/random/faure.hpp>
#include <boost/random/sobol.hpp>
#include <boost/random/niederreiter_base2.hpp>

boost::random::niederreiter_base2 gen(2000);
```

Randomized quasi-Monte Carlo integration (1.67)

Quadrature nodes taken from a randomized quasi-random sequence. Convergence rate improves over traditional Monte-Carlo integration to $\mathcal{O}\left(\frac{\log(n)^d}{\sqrt{t}n}\right)$, where t is the number of threads, d is the dimension, and n is the number of function evaluations each thread is able to perform.

Randomized quasi-Monte Carlo integration

```
#include <boost/math/quadrature/randomized_quasi_monte_carlo.hpp>
auto g = [](std::vector<double> const & x) {
  return 1/(1-cos[0]*cos[1]);
};

std::vector<std::pair<double, double>> bounds{{0, 1}, {0, 1}};
double error_goal = 0.0001;
randomized_quasi_monte_carlo<double, decltype(g)> mc(g, bounds, error_goal);

auto task = mc.integrate();
// Update error goal, view current error estimate, or cancel here.
double value = task.get();
```

Numerical differentiation

via the complex step derivative:

$$f'(x) \approx \Im f(x + ih)/h + \mathcal{O}(h^2)$$

```
#include <boost/math/tools/numerical_differentiation.hpp>

double x = 7.2;
double e_prime = complex_step_derivative(std::exp<std::complex<double>>, x);
```

Numerical differentiation

via finite differences

```
auto f = [](double x) { return std::exp(x); };
double x = 1.7;
double dfdx = finite_difference_derivative(f, x);
```

Note: No stepsize needed! Error is roughly $\sim \epsilon^{6/7}$.

Fourier transforms

Fourier transforms (as well as all oscillatory quadratures) have large relative condition number.

Sophisticated methods must be employed to do a numerical Fourier transform.

Oura's method (1.68)

Boost provides a double-exponential oscillatory quadrature designed by Takuya Oura.

```
auto f = [](Real x)->Real { return 1/x; };
Real omega = 1;
Real Is = oura_fourier_sin<decltype(f), Real>(f, omega);
```

This computes $\displaystyle\int_0^\infty \frac{\sin(\omega x)}{x}\,\mathrm{d}x$

(Dream) Crown jewel of release 1.68: RiskShrink signal denoising and wavelet compression!

Still in gestation, but should be able to denoise horrible signals reliably without user input.

Should be able to (lossy) compress non-random signals of length N to length (optimally) $\sqrt{N}$ and $N/100$ in general.

Should be able to turn dense matrices into sparse matrices with little loss of fidelity.

github.com/boostorg/math