Mind the Gap (Under the Curve)

A Hobbyist's Summary on Quadrature with Applications to Option Pricing

Author: Gene Boo | Date: Jun 2023, Sep 2025

Introduction

Numerical integration, or quadrature, is a cornerstone of computational mathematics. It's the art of approximating definite integrals—those elegant expressions of accumulated change—using finite, well-chosen sums. In finance, especially in the realm of option pricing, quadrature methods offer a compelling alternative to closed-form formulas (when they exist) and Monte Carlo simulations (when brute force meets high dimensions). Quadrature is where mathematical elegance meets computational pragmatism.

The term **"quadrature"** comes from the Latin *quadratus*, meaning "square." Historically, it referred to the classical geometric problem of *squaring a figure*—constructing a square with the same area as a given circle or other shape using only a straightedge and compass. This was a central challenge in ancient Greek mathematics, famously leading to the proof that exact squaring of the circle is impossible (since π is transcendental).

Extension to Integration Over time, the meaning of quadrature broadened to denote area computation in general. Since integration in calculus is fundamentally about finding the area under a curve, the term "quadrature" became synonymous with numerical integration:

$$\text{Quadrature:} \int_a^b f(x)\,dx \approx \sum_{i=1}^{n} w_i\, f(x_i),$$

where w_i and x_i are weights and nodes in a quadrature rule (e.g., Simpson's rule, Gaussian quadrature).

```
Quadrature transforms integrals into weighted sums:
```

$$I = \int_a^b f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

```
Different rules pick nodes $x_i$ and weights $w_i$ to exploit smoothness, singularities,
symmetry, or infinite domains. The smarter the rule, the fewer points you need—and
the more accurate your result.
```

***Think of quadrature** as placing a few clever sensors across a landscape to estimate its total brightness. You don't need to measure every pixel—just the right ones. It's like tasting a dish and knowing the recipe: strategic sampling beats brute force. Below are some simple to complex integration problems that many practitioners use quadrature for:*

Rank	Domain	Integral	Name / Description
1	Finance	$C = e^{-rT} \int_0^\infty \max(s-K,0)\, f_{S_T}(s)\, ds$	Risk-neutral expectation for a European call
2	Logistics	$\mathbb{E}[C(Q)] = \int_0^\infty (c_h(x-Q)_+ + c_b(Q-x)_+)\, f_D(x)\, dx$	Newsvendor expected cost integral
3	Astrophysics	$D_L(z) = (1+z)\frac{c}{H_0}\int_0^z \frac{dz'}{E(z')}$	Luminosity distance in FLRW cosmology
4	Physics	$S(\alpha) = \int_0^\infty \cos(\alpha x^2)\, dx$	Fresnel cosine integral
5	Chemistry	$\iint \frac{\phi_a(\mathbf{r}_1)\phi_b(\mathbf{r}_1)\,\phi_c(\mathbf{r}_2)\phi_d(\mathbf{r}_2)}{\lVert\mathbf{r}_1 - \mathbf{r}_2\rVert}\, d\mathbf{r}_1\, d\mathbf{r}_2$	Two-electron repulsion integral (ERI)
6	Machine Learning	$Z(\theta) = \int_{\mathbb{R}^d} e^{-E(x;\theta)}\, dx$	Partition function of an energy-based model
7	Neural Networks	$p(y^* \mid x^*, \mathcal{D}) = \int p(y^* \mid x^*, w)\, p(w \mid \mathcal{D})\, dw$	Bayesian neural network predictive distribution

But quadrature isn't just a numerical trick—it's a philosophy. It asks: "What do I know about this function? Where does it change? Where does it hide its secrets?" And then it answers with precision, elegance, and sometimes, a bit of magic. Most importantly, with well implemented classes or helper functions, it preserves the original outlook of the integral within the code itself and this is the case for most computer languages (apart from say, assembly language).

In this guide, we'll journey from the basics of integration to the cutting edge of financial modeling. We'll explore trapezoids and Chebyshev nodes, adaptive zoom-ins and Gaussian elegance. We'll see how quadrature powers option pricing—from Black-Scholes to Carr-Madan, from stop-loss premiums to strike-aware formulations. Whether you're a curious teen, a hobbyist coder, or a budding quant, this article is your launchpad into the world where math meets money.

Why it matters: Quadrature lets us solve problems that are too messy for formulas and too slow for brute force. It's the bridge between theory and practice, between elegance and efficiency.

Fundamental Quadrature Methods

Trapezoid Rule

```
The simplest of them all, but wait - don't throw it out! It may not serve well
compared to other methods for direct integration, but it is most useful for
post-processing Fast Fourier Transform (FFT) results. We present to you the
trapezoid rule, which approximates integrals using linear interpolation to link the
nodes:
```

$$\int_a^b f(x)\,dx \approx \frac{b-a}{2}\,[f(a)+f(b)]$$

```
In the context of FFT-based methods, the trapezoid rule has a unique advantage: the
```

FFT naturally produces function samples at equally spaced points over a periodic domain. The trapezoid rule is *exact* for integrating any trigonometric polynomial whose highest frequency is below the Nyquist limit — precisely the type of output the FFT delivers. This means that for smooth, periodic functions reconstructed from FFT coefficients, the trapezoid rule achieves spectral (exponentially fast) convergence without the complexity of higher-order schemes.

Other quadrature rules, such as Simpson's or Gaussian quadrature, require either non-uniform nodes or additional function evaluations at midpoints, which are not directly available from the FFT grid without interpolation. Interpolation would add computational cost and potential aliasing errors, negating their theoretical accuracy advantage. The trapezoid rule, by contrast, uses the FFT's existing uniform grid directly, making $\mathbf{O}(N)$ to apply after an $\mathbf{O}(N \log N)$ transform, with no extra sampling.

In short, when your data is periodic and uniformly sampled — as it is after an FFT — the trapezoid rule is not just "good enough," it is *optimal* in both accuracy and efficiency.

```
# Composite Trapezoid Rule Pseudocode
function trapezoid(f, a, b, n):
    h = (b - a) / n
    sum = 0.5 * (f(a) + f(b))
    for i from 1 to n-1:
        x = a + i * h
        sum += f(x)
    return h * sum
```

***Imagine measuring** a curved shape by breaking it into many small trapezoids instead of rectangles. It's like using slanted roof pieces instead of flat blocks to cover a curved surface - you get a better fit with the same number of pieces!*

Simpson's Rule

Simpson's rule uses quadratic interpolation for better accuracy:

$$\int_a^b f(x)\,dx \approx \frac{b-a}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right]$$

It is a special case of the *Newton–Cotes formulas*, which approximate the integrand by an n-degree polynomial passing through $n+1$ equally spaced points, then integrate that polynomial exactly.

Beyond Quadratics: Higher-Degree Newton–Cotes Rules

By increasing the degree of the interpolating polynomial, we can create more accurate rules (for smooth functions) without refining the grid:

- **Trapezoid Rule** — degree 1 (linear interpolation)
- **Simpson's Rule** — degree 2 (quadratic interpolation)
- **Simpson's 3/8 Rule** — degree 3 (cubic interpolation):

  $$\int_a^b f(x)\,dx \approx \frac{3h}{8}[f(x_0)+3f(x_1)+3f(x_2)+f(x_3)]$$

  where $h=\frac{b-a}{3}$.
- **Boole's Rule** — degree 4 (quartic interpolation):

  $$\int_a^b f(x)\,dx \approx \frac{2h}{45}[7f(x_0)+32f(x_1)+12f(x_2)+32f(x_3)+7f(x_4)]$$

  where $h=\frac{b-a}{4}$.

These higher-order rules can integrate polynomials of degree up to n exactly, but they may suffer from *Runge's phenomenon* and numerical instability if n is too large over wide intervals. In practice, they are often applied in **composite form** — breaking the domain into smaller subintervals and applying the low-degree rule repeatedly.

Think of it like this: *Simpson's rule is like fitting a smooth arch between three points. If you add more points, you can fit fancier curves — cubic, quartic, and beyond — which hug the function more closely. But if you try to fit too fancy a curve over too wide a stretch, it can wiggle wildly between points. That's why we often use these rules in small chunks and piece them together. The concept sounds similar to splines, but in splines there are gradient constraints between pieces in place to ensure smoothness.*

Beyond Polynomials: Spline-Based Numerical & Analytical Hybrid Integration

Spline integration uses **piecewise polynomial functions** to approximate a function and then integrates the spline instead of the original function. This approach gives smoothness and numerical stability, especially when the function's behavior varies across intervals. It is a hybrid as the integration of spline curves can be done analytically. The integral is computed by summing the area under each spline segment:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n}\int_{x_{i-1}}^{x_i} S_i(x)\,dx$$

where $S_i(x)$ is the spline polynomial on the i-th interval. **Runge's Phenomenon:** When using high-degree polynomials over wide intervals, the interpolation may oscillate wildly between points, especially near the edges. This leads to poor approximations and instability. Spline methods mitigate this by using low-degree polynomials locally.

- **Linear Spline Integration** — piecewise degree 1 (equivalent to trapezoid rule): On each interval $I_i=[x_i,x_{i+1}]$ with $h_i=x_{i+1}-x_i$,

  $$S_i(x)=y_i+d_i\,(x-x_i),\quad d_i=\frac{y_{i+1}-y_i}{h_i}.$$

The exact integral of this segment is the trapezoid rule:

$$\int_{x_i}^{x_{i+1}} S_i(x)\,dx \;=\; \frac{h_i}{2}\,(y_i + y_{i+1}).$$

- **Cubic Spline Integration** — piecewise degree 3 with C^2 continuity:
  Let M_i denote the second derivative at node x_i (e.g., a *natural* spline uses $M_0 = M_n = 0$). On $I_i = [x_i, x_{i+1}]$,

  $$\begin{aligned} S_i(x) = {} & \frac{M_i}{6h_i}(x_{i+1}-x)^3 + \frac{M_{i+1}}{6h_i}(x-x_i)^3 \\ & + \left(y_i - \frac{M_i h_i^2}{6}\right)\frac{x_{i+1}-x}{h_i} + \left(y_{i+1} - \frac{M_{i+1} h_i^2}{6}\right)\frac{x-x_i}{h_i}. \end{aligned}$$

  Each S_i is a cubic polynomial, so $\int_{x_i}^{x_{i+1}} S_i(x)\,dx$ is obtained analytically by integrating the above terms.

- **Hermite Spline Integration** — piecewise cubic with derivative constraints (cubic Hermite):
  Using values $y_i = f(x_i)$ and slopes $m_i = f'(x_i)$, with $t = \frac{x-x_i}{h_i}$ on I_i,

  $$\begin{aligned} & S_i(x) = y_i\,h_{00}(t) + h_i m_i\,h_{10}(t) + y_{i+1}\,h_{01}(t) + h_i m_{i+1}\,h_{11}(t), \\ & h_{00}(t) = 2t^3 - 3t^2 + 1, \quad h_{10}(t) = t^3 - 2t^2 + t, \\ & h_{01}(t) = -2t^3 + 3t^2, \quad h_{11}(t) = t^3 - t^2. \end{aligned}$$

  This form is smooth and integrates exactly per segment by integrating the basis polynomials $h_{rs}(t)$.

- **PCHIP (Piecewise Cubic Hermite Interpolating Polynomial)** — monotone, shape-preserving cubic Hermite:
  PCHIP is the Hermite spline above with slopes m_i chosen to preserve local monotonicity and avoid overshoot. Let $h_k = x_{k+1} - x_k$ and

  $$d_k = \frac{y_{k+1} - y_k}{h_k}.$$

  For interior nodes $k = 1, \ldots, n-1$,

  $$m_k = \begin{cases} 0, & \text{if } d_{k-1}\,d_k \le 0, \backslash[6pt] \dfrac{w_1 + w_2}{\frac{w_1}{d_{k-1}} + \frac{w_2}{d_k}}, & \text{otherwise,} \end{cases}$$

  where $w_1 = 2h_k + h_{k-1}$ and $w_2 = h_k + 2h_{k-1}$. (End slopes use one-sided, shape-preserving formulas.) Integration then follows the Hermite segment integral.

- **B-Spline Integration** — basis-spline representation with local control:
  Represent the fitted spline as

  $$S(x) = \sum_j c_j\,N_{j,p}(x),$$

  where $N_{j,p}$ are B-spline basis functions of degree p defined by the Cox–de Boor recursion with knot vector $\{t_k\}$: \[N_{j,0}(x)=\begin{cases}1,&t_j\le x

- **Mixed Splines** — adaptively choose the spline per region:
  Partition $[a,b]$ into subregions $\{R_s\}$. In smooth regions use a cubic spline for C^2 accuracy; near kinks, steps, or strongly monotone stretches use PCHIP to prevent overshoot:

$$\int_a^b f(x)\,dx \;\approx\; \sum_s \int_{R_s} S^{(s)}(x)\,dx, \qquad S^{(s)}(x) \in \{\text{cubic}, \text{PCHIP}, \text{linear}\},$$

enforcing at least C^0 continuity (and optionally C^1) at region boundaries.

The composite spline integral is the sum of segment integrals:

$$\int_a^b f(x)\,dx \;\approx\; \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} S_i(x)\,dx.$$

Runge's Phenomenon: High-degree single-interval polynomials may oscillate near the edges, producing unstable approximations. Splines mitigate this by using low-degree polynomials locally with continuity constraints.

Real-World Examples

- **Finance (Yield Curves):** Integrate discount factors or forward rates. PCHIP is often preferred to avoid spurious arbitrage (overshoots) in sparsely sampled maturities.
- **Engineering (Stress–Strain Energy):** Integrate smooth cubic splines of experimental stress–strain data to compute specific energy absorption.
- **Robotics & Motion Planning:** B-splines model trajectories; integrating speed along a spline yields time/energy estimates with local control over segments.
- **Medical/Imaging:** Integrate dose–response or intensity profiles reconstructed with splines to get total dose/flux while preserving monotonic sections with PCHIP.
- **Computer Graphics/CAD:** B-splines (and NURBS) provide precise curve/surface control; integrating along parameterized splines gives arc length or area.

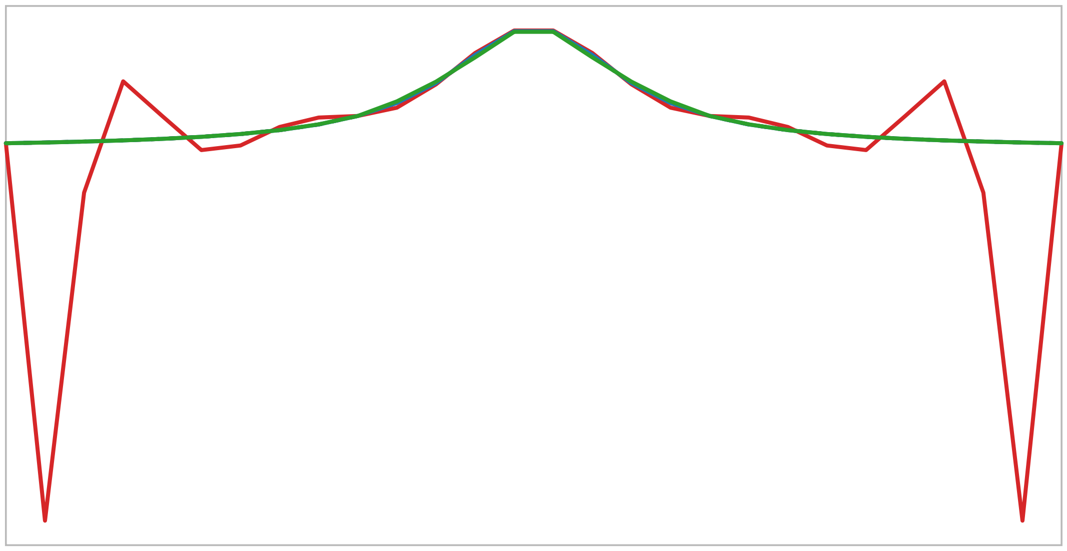
```
# Spline-Based Numerical & Analytical Hybrid Integration
// Inputs: x[0..n], y[0..n]  (monotone x), method ∈ { "linear", "cubic", "pchip",
"bspline" }
// Optional params: bc, knots, degree, quad_points, region_selector (for "mixed")
function spline_integrate(x, y, method, params):

    if method == "mixed":
        regions = params.region_selector(x, y)   // e.g., segment by
smoothness/monotonicity
        total = 0
        for R in regions:                        // R = {idx_start, idx_end, methodR}
            total += spline_integrate(x[R], y[R], methodR, params_for(methodR))
        return total

    // 1) Fit the chosen spline
    if method == "linear":
        S = fit_linear_spline(x, y)              // store per-interval slope d_i
    else if method == "cubic":
        S = fit_cubic_spline(x, y, bc=params.bc) // solve tri-diagonal for second
derivs M_i
    else if method == "pchip":
        S = fit_pchip(x, y)                      // compute monotone slopes m_i
(Fritsch-Carlson)
    else if method == "bspline":
        (t, c, p) = bspline_fit(x, y, degree=params.degree, knots=params.knots)

    // 2) Integrate
    total = 0
    if method in {"linear","cubic","pchip"}:
        for i in 1..n:
            a = x[i-1]; b = x[i]
            // retrieve polynomial coefficients for S_i(x) on [a,b]
            coeffs = local_polynomial(S, i)      // e.g., a0 + a1*(x-a) + a2*(x-a)^2
+ a3*(x-a)^3
            total += poly_segment_integral(coeffs, a, b) // analytic antiderivative
        return total

    else if method == "bspline":
        // integrate span-by-span on knot intervals where basis is low-degree and
sparse
        for k in valid_knot_spans(t):
            a = t[k]; b = t[k+1]
            if b ≤ a: continue
            // Quadrature (robust for arbitrary knots). p+3 Gauss points is usually
enough.
            total += gauss_legendre(
                        f(x) = sum_j c[j]*B_spline_basis(j, p, x, t),
                        a, b, m = params.quad_points or (p + 3))
        return total
```

Think of it like this: *Instead of stretching one big curve over all your data (which can wiggle too much), spline integration fits lots of small, smooth curves between pairs of points, but also ensures smoothness at the joints for non-linear ones. Hermite splines even use the slope at each point to make the fit smarter. Then, you add up the area under each little curve to get the total. It's like laying flexible tiles over a bumpy surface — much more accurate and stable than one big sheet!*

Visual Comparison of Interpolation Methods

The diagram below shows how different interpolation methods behave. Notice how the high-degree polynomial (red) oscillates near the edges — this is Runge's phenomenon. Cubic and Hermite splines (blue and green) follow the true function (dashed black but hidden by the splines) more smoothly.

Interpolation Methods on Runge's Function



Romberg Integration

Romberg integration is a systematic way to refine trapezoidal rule estimates by applying **Richardson extrapolation** — a technique that uses results at different step sizes to cancel out leading error terms and accelerate convergence.

The trapezoidal rule has an error term proportional to h^2 for smooth functions, where h is the step size. If we compute the trapezoid estimate with h and with $h/2$, we can combine them to eliminate the h^2 term, leaving an error of order h^4.

Romberg's recursive scheme:

$$R(k,0) = \text{Trapezoid estimate with } 2^k \text{ subintervals}$$

$$R(k,m) = R(k,m-1) + \frac{R(k,m-1) - R(k-1,m-1)}{4^m - 1}$$

Here:

- k = refinement level (more subintervals)
- m = extrapolation level (higher-order correction)
- 4^m appears because halving h reduces the error term by a factor of 2^{2m} for even-order methods

Two-Step Example

Suppose we want $\int_0^1 e^{-x^2} dx$:

1. Compute $R(0,0)$ using 1 trapezoid $(h=1)$ → say 0.68394
2. Compute $R(1,0)$ using 2 trapezoids $(h=0.5)$ → say 0.73137
3. Apply Richardson extrapolation for $m=1$:

$$R(1,1) = R(1,0) + \frac{R(1,0) - R(0,0)}{4^1 - 1}$$

$$R(1,1) = 0.73137 + \frac{0.73137 - 0.68394}{3} \approx 0.74737$$

This is already much closer to the true value ≈ 0.74682.

Three-Step Example

Continuing:

1. Compute $R(2,0)$ with 4 trapezoids ($h = 0.25$) $\to$ 0.74298
2. Extrapolate with $m = 1$:

$$R(2,1) = 0.74298 + \frac{0.74298 - 0.73137}{3} \approx 0.74692$$

3. Extrapolate again with $m = 2$ using $R(2,1)$ and $R(1,1)$:

$$R(2,2) = 0.74692 + \frac{0.74692 - 0.74737}{15} \approx 0.74689$$

Now we're way more accurate after just 3 trapezoid refinements.

Analogy 1 – The Coastline Trick: *Imagine measuring the length of a coastline with a big ruler — you miss all the little bays and inlets. Then you measure again with a smaller ruler — you catch more detail, but still miss the tiniest wiggles. If you know how the error shrinks when you halve the ruler size, you can combine the two measurements to cancel most of the bias and get a much better estimate without going to an absurdly tiny ruler.*

Analogy 2 – The Driving Speed Limit Sign: *Suppose you're driving toward a road sign with small text under the speed limit. At 100m away you can sort of guess the letters, but they're fuzzy. At 50m they're clearer, but still not perfect. If you notice how much clearer things get when halving the distance, you can predict what the sign says even before you're right next to it.*

Advanced Quadrature Techniques

Gaussian Quadrature

Gaussian quadrature chooses both the *nodes* x_i and *weights* w_i so that the formula

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

is exact for all polynomials up to degree $2n - 1$ (the maximum possible for n points). The nodes are the roots of an orthogonal polynomial associated with a weight function $w(x)$ on the interval $[a, b]$, and the weights are derived from the orthogonality conditions. This optimal placement means fewer points are needed for the same accuracy compared to equally spaced rules.

***Think of Gaussian quadrature** as sending surveyors to the most informative spots in a landscape. Instead of spacing them evenly, you place them exactly where they'll capture the most detail about the terrain. The math behind it guarantees that if your landscape is made of smooth hills (polynomials), you'll measure it perfectly with surprisingly few surveyors.*

Gauss–Legendre

For general integrals on $[-1, 1]$ with weight function $w(x) = 1$:

$$\int_{-1}^{1} f(x)\, dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

where x_i are the roots of the Legendre polynomial $P_n(x)$. This is the most common Gaussian quadrature and is easily rescaled to any finite interval $[a, b]$.

Gauss–Hermite

For integrals with weight e^{-x^2} on $(-\infty, \infty)$:

$$\int_{-\infty}^{\infty} f(x)e^{-x^2}\, dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

where x_i are the roots of the Hermite polynomial $H_n(x)$. This is ideal for problems involving the normal distribution, such as option pricing under Black-Scholes.

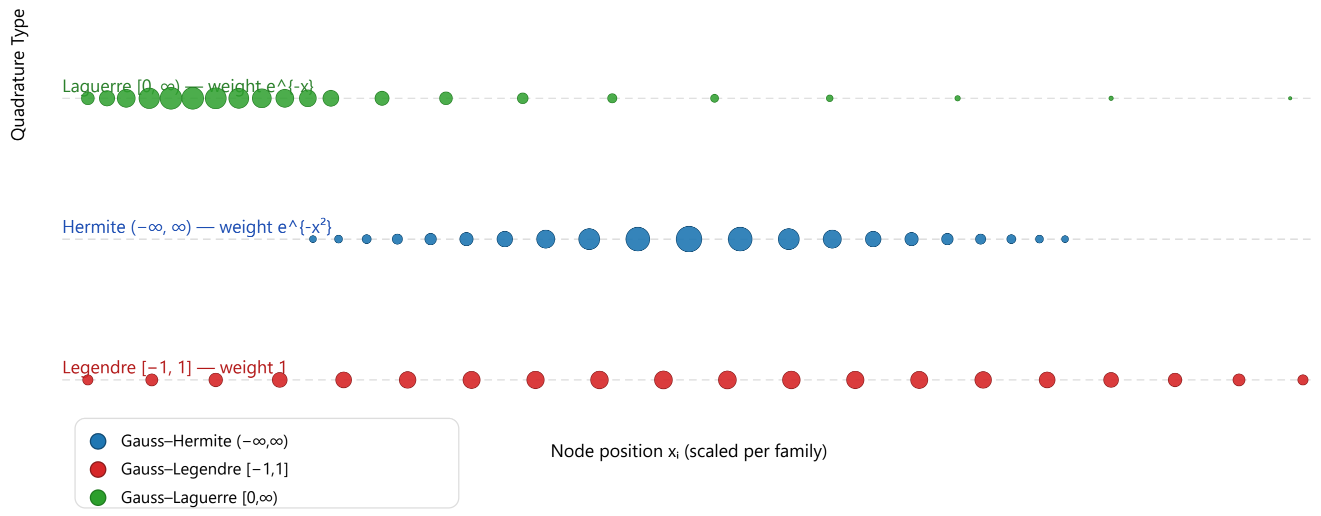
Gauss–Laguerre

For integrals with weight e^{-x} on $[0, \infty)$:

$$\int_{0}^{\infty} f(x)e^{-x}\, dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

where x_i are the roots of the Laguerre polynomial $L_n(x)$. Useful for exponential decay problems, such as radioactive decay models or certain Laplace transforms, and it is often used to integrate Lognormal problems - which we encounter a lot in GBM option pricing.

Gauss Node–Weight Comparison (N=50)



Node positions by family (lanes) with marker size ∝ weight. (Illustrative; exact nodes/weights depend on N and scaling.)

Beyond Legendre, Hermite, Laguerre...

Gaussian quadrature is a *family* of rules, each tied to a specific orthogonal polynomial and weight function:

- **Gauss–Chebyshev** (first and second kind) — for weights $(1-x^2)^{-1/2}$ and $(1-x^2)^{1/2}$ on $[-1,1]$, excellent for trigonometric integrals.
- **Gauss–Jacobi** — generalizes Legendre and Chebyshev with weight $(1-x)^{\alpha}(1+x)^{\beta}$, $\alpha,\beta > -1$.
- **Gauss–Gegenbauer** — for ultraspherical weights $(1-x^2)^{\lambda-1/2}$.
- **Gauss–Radau** — fixes one endpoint as a node; exact for degree $2n-2$.
- **Gauss–Lobatto** — fixes both endpoints; exact for degree $2n-3$.
- **Gauss–Kronrod** — extends an n-point Gauss–Legendre rule to $2n+1$ points, nesting the original nodes for adaptive error estimation.

Each variant is tuned to a specific weight function or integration constraint, allowing you to exploit known structure in the integrand.

***Think of these variants** as different "dial settings" on the same precision instrument. If you know your function has certain quirks — like it's heavier near the edges, or it lives on an infinite domain — you pick the Gaussian rule that's already tuned for that shape. That way, you get more accuracy with fewer points, just by matching the tool to the job.*

Gauss–Chebyshev

Gauss–Chebyshev quadrature is tailored for integrals with the weight function $w(x) = \frac{1}{\sqrt{1-x^2}}$ on $[-1,1]$:

$$\int_{-1}^{1} \frac{f(x)}{\sqrt{1-x^2}}\,dx \approx \frac{\pi}{n}\sum_{i=1}^{n} f\left(\cos\frac{2i-1}{2n}\pi\right)$$

The nodes are simply $x_i = \cos\frac{2i-1}{2n}\pi$ and all weights are equal to π/n. This rule is exact for polynomials of degree up to $2n-1$ multiplied by the weight $w(x)$. It is especially efficient for integrals that can be transformed into this form, such as many trigonometric integrals.

Think of Gauss–Chebyshev *as a method that already "knows" your function lives on a circle. It places its measuring points at equally spaced angles around that circle, which makes it perfect for problems involving sines, cosines, or anything periodic in disguise.*

Gauss–Jacobi

Gauss-Jacobi quadrature generalizes Gauss-Legendre and Gauss-Chebyshev by using the weight function $w(x) = (1-x)^\alpha(1+x)^\beta$, with $\alpha, \beta > -1$, on $[-1,1]$:

$$\int_{-1}^{1} (1-x)^\alpha (1+x)^\beta f(x)\, dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

The nodes x_i are the roots of the Jacobi polynomial $P_n^{(\alpha,\beta)}(x)$, and the weights w_i are computed from orthogonality conditions. By tuning α and β, you can emphasize accuracy near one or both endpoints — useful for integrands with endpoint singularities or sharp features.

Think of Gauss–Jacobi *as a "custom-fit" quadrature. If your function has tricky behavior near one or both ends of the interval, you can tell Gauss–Jacobi to send more surveyors there. It's like assigning extra inspectors to the weak spots in a bridge.*

Gauss–Lobatto

Gauss-Lobatto quadrature is a variant of Gaussian quadrature that *includes both endpoints* of the interval $[-1,1]$ as nodes. It is exact for polynomials of degree up to $2n-3$ (slightly less than Gauss-Legendre for the same n), but is valuable when endpoint values are known or required:

$$\int_{-1}^{1} f(x)\, dx \approx w_1 f(-1) + \sum_{i=2}^{n-1} w_i f(x_i) + w_n f(1)$$

The interior nodes are the roots of $P'_{n-1}(x)$, the derivative of the Legendre polynomial of degree $n-1$. Gauss-Lobatto is popular in spectral methods and finite element analysis, where boundary values play a key role.

Think of Gauss–Lobatto *as a measuring crew that insists on checking the gates at both ends of a fence as well as the posts in between. It's perfect when you care about what's happening right at the boundaries, not just in the middle.*

Adaptive Quadrature

Adaptive methods allocate work where the integrand demands it. They compare a *coarse* estimate on an interval with a *refined* estimate obtained by splitting the interval; the difference acts as an **error estimator**. Subintervals whose error exceeds tolerance are split recursively until local targets are met, then results are aggregated.

Simpson's rule (1D):

$$S[a,b] \;=\; \frac{b-a}{6}\Big(f(a)+4f(\tfrac{a+b}{2})+f(b)\Big), \quad E_S[a,b] \;=\; -\frac{(b-a)^5}{2880}\,f^{(4)}(\xi)$$

The adaptive estimator uses

$$\Delta_S \;=\; S[a,b]-\big(S[a,c]+S[c,b]\big) \;\approx\; \tfrac{1}{15}\;S[a,b]-S[a,c]-S[c,b]\big|$$

as a proxy for the local truncation error (with Richardson correction).

Gauss–Kronrod (GK 7–15): A 7-point Gauss rule is *embedded* in a 15-point Kronrod rule on the same nodes plus extras. Let I_G be the Gauss estimate and I_K the Kronrod estimate on a subinterval. An effective error indicator is

$$\Delta_{GK} \;=\; I_K - I_G \;,$$

with robust variants that scale by norms of f to avoid under/over-estimation. Subdivide where Δ exceeds a local tolerance.

```
# Adaptive Simpson (recursive, with Richardson correction and depth cap)

function asr(f, a, b, fa, fm, fb, S, tol, depth, max_depth):
        c   = 0.5*(a + b)
        m1  = 0.5*(a + c)
        m2  = 0.5*(c + b)
        fm1 = f(m1)
        fm2 = f(m2)
        Sl  = (c - a)/6 * (fa + 4*fm1 + fm)
        Sr  = (b - c)/6 * (fm + 4*fm2 + fb)
        S2  = Sl + Sr
        err = |S2 - S|

        if (err ≤ 15*tol) or (depth ≥ max_depth):
                # Richardson correction improves order to O(h^5)
                return S2 + (S2 - S)/15

        # Recurse on halves with halved tolerances
        left  = asr(f, a, c, fa, fm1, fm, Sl, tol/2, depth+1, max_depth)
        right = asr(f, c, b, fm, fm2, fb, Sr, tol/2, depth+1, max_depth)
        return left + right

function adaptive_simpson(f, a, b, tol, max_depth=20):
        fa = f(a); fb = f(b); fm = f(0.5*(a+b))
        S  = (b - a)/6 * (fa + 4*fm + fb)
        return asr(f, a, b, fa, fm, fb, S, tol, 0, max_depth)
```

```
# Adaptive Gauss-Kronrod 15(7) (iterative, stack-based)

function gk15_eval(f, a, b):
        # standard abscissae x_k in [0,1] (map to [-1,1] then to [a,b])
        # and weights w_g (Gauss 7) and w_k (Kronrod 15)
        # returns (I_K, I_G, err_indic, f_norm)
        ...
        return (IK, IG, |IK - IG|, norm_est)

function adaptive_gk(f, a, b, epsabs, epsrel, max_depth, limit_nodes):
        stack = [(a, b, 0)]
        total = 0
        used_nodes = 0
        while stack not empty:
                (u, v, depth) = stack.pop()
                (IK, IG, err, fnorm) = gk15_eval(f, u, v)
                used_nodes += 15
                tol_loc = max(epsabs, epsrel*abs(IK))
                if (err ≤ tol_loc) or (depth ≥ max_depth) or (used_nodes ≥
limit_nodes):
                        total += IK
                else:
                        m = 0.5*(u + v)
                        stack.push((m, v, depth+1))
                        stack.push((u, m, depth+1))
        return total
```

> ***Adaptive quadrature** is a smart spotlight. It shines brighter (uses more points) where the function is complicated and dims (saves points) where it's smooth. Instead of wasting effort evenly, it pays attention and spends effort where it matters.*

Worked mini-examples

1) Endpoint singularity (integrable): $\int_0^1 x^{-1/2}\,dx = 2$

Why adapt? Near $x = 0$, $f(x)$ spikes. Adaptive methods place many subintervals near 0 and few near 1, achieving high accuracy quickly.

$$\int_0^1 x^{-1/2}\,dx = \left[2x^{1/2}\right]_0^1 = 2.$$

2) Highly oscillatory: $\int_0^1 \frac{\sin(50x)}{x}\,dx$

Why adapt? Oscillations demand more points where the wavelength is short. Adaptive splitting concentrates effort near regions of rapid variation; elsewhere, larger panels suffice.

3) Localized spikes: $\int_{-3}^{3} e^{-x^2}\,dx = \sqrt{\pi}$

Why adapt? Most contribution comes from $|x| \lesssim 2$. Adaptive schemes refine where e^{-x^2} is large and coarsen in the tails.

Design details that matter

- **Tolerance policy:** Use absolute/relative blend $\mathrm{tol}_{\mathrm{loc}} = \max(\epsilon_{\mathrm{abs}}, \epsilon_{\mathrm{rel}} \cdot |I_{\mathrm{loc}}|)$.
- **Depth & size caps:** Set `max_depth` and minimum panel width to prevent infinite recursion on pathological f.
- **Reuse evaluations:** Cache and pass *endpoint/midpoint* values down the recursion to avoid recomputing f.
- **Robustness:** If `NaN`/`Inf` encountered, split aggressively or transform variables (e.g., endpoint maps for algebraic singularities).
- **Performance:** Prefer iterative stacks over deep recursion in Python to avoid overhead; vectorize batched point evaluations where possible.

Python reference implementations

```
# Adaptive Simpson (robust, recursive with depth cap)
from __future__ import annotations
import math
from typing import Callable

def adaptive_simpson(f: Callable[[float], float],
                                         a: float, b: float,
                                         epsabs: float = 1e-10,
                                         epsrel: float = 1e-8,
                                         max_depth: int = 20) -> float:
        """Adaptive Simpson integration on [a,b]."""

        def S(fa, fm, fb, a, b):
                return (b - a) * (fa + 4*fm + fb) / 6.0

        def recurse(a, b, fa, fm, fb, Sab, tol, depth):
                c   = 0.5*(a + b)
                m1  = 0.5*(a + c)
                m2  = 0.5*(c + b)
                fm1 = f(m1)
                fm2 = f(m2)
                Sl  = S(fa, fm1, fm, a, c)
                Sr  = S(fm, fm2, fb, c, b)
                S2  = Sl + Sr
                err = abs(S2 - Sab)
                if (err <= 15*tol) or (depth >= max_depth):
                        # Richardson correction
                        return S2 + (S2 - Sab)/15.0
                # Split tolerance
                return (recurse(a, c, fa, fm1, fm, Sl, 0.5*tol, depth+1) +
                                recurse(c, b, fm, fm2, fb, Sr, 0.5*tol, depth+1))

        fa = f(a); fb = f(b); fm = f(0.5*(a + b))
        Sab = S(fa, fm, fb, a, b)
        # Global tolerance target for the full interval
        tol0 = max(epsabs, epsrel*abs(Sab))
        return recurse(a, b, fa, fm, fb, Sab, tol0, 0)
```

```
# Adaptive Gauss–Kronrod 15(7) (iterative, QUADPACK-style)
from __future__ import annotations
import math
from typing import Callable, List, Tuple

# Kronrod 15 abscissae (nonnegative) and weights; Gauss-7 weights embedded
_xgk = [0.9914553711208126, 0.9491079123427585, 0.8648644233597691,
                0.7415311855993945, 0.5860872354676911, 0.4058451513773972,
                0.2077849550078985, 0.0]
_wg  = [0.1294849661688697, 0.2797053914892767,
                0.3818300505051189, 0.4179591836734694]  # Gauss 7 (nonnegative)
_wk  = [0.02293532201052922, 0.06309209262997855, 0.10479001032225019,
                0.14065325971552592, 0.16900472663926790, 0.19035057806478541,
                0.20443294007529889, 0.20948214108472783]  # Kronrod 15 (nonnegative)

def _gk15_eval(f: Callable[[float], float], a: float, b: float) ->
Tuple[float,float,float,float]:
        """Evaluate GK15 on [a,b]. Return (IK, IG, err_est, fnorm)."""
        c = 0.5*(a + b)
        h = 0.5*(b - a)
        # function values at positive nodes and symmetry
        fk_sum = 0.0
        fg_sum = 0.0
        absf_sum = 0.0

        # center
        fc = f(c)
        fk_sum += _wk[-1] * fc
        absf_sum += _wk[-1] * abs(fc)

        # loop over the 7 positive Kronrod abscissae
        for j in range(7):
                x = _xgk[j]
                w_k = _wk[j]
                xh = h * x
                f1 = f(c - xh)
                f2 = f(c + xh)
                fk_sum += w_k * (f1 + f2)
                absf_sum += w_k * (abs(f1) + abs(f2))

                # Gauss nodes are a subset for j = 1,3,5,7 in xgk order:
                # their indices in our loop correspond to 1,3,5,6 (because of 0-
based)
        # Gauss contribution (explicitly pick corresponding nodes)
        gauss_idx = [1, 3, 5]  # indexes into _xgk (0-based) for Gauss positive nodes
        for gi, wg in zip(gauss_idx, _wg[:-1]):
                x = _xgk[gi]
                xh = h * x
                f1 = f(c - xh)
                f2 = f(c + xh)
                fg_sum += wg * (f1 + f2)
        # Actually for Gauss-7, the center x=0 is a node with weight 0.417959…,
covered by _wg[-1].

        IK = h * fk_sum
        IG = h * (_wg[-1]*fc + fg_sum - _wg[-1]*fc)  # fg_sum already accounts; keep
clarity
        IG = h * (fg_sum + _wg[-1] * fc)

        # Error indicator and a mild norm-based safeguard
        err = abs(IK - IG)
        fnorm = h * absf_sum
        return IK, IG, err, fnorm

def quadgk(f: Callable[[float], float],
                  a: float, b: float,
                  epsabs: float = 1e-10,
                  epsrel: float = 1e-8,
                  max_depth: int = 20,
                  limit: int = 10_000) -> float:
        """Adaptive Gauss-Kronrod 15(7) on [a,b]."""
        # Work stack: (a, b, depth, last_IK, last_err) -- IK,err cached optional
        stack: List[Tuple[float,float,int]] = [(a, b, 0)]
        total = 0.0
        used = 0

        while stack:
                u, v, d = stack.pop()
                IK, IG, err, fnorm = _gk15_eval(f, u, v)
                used += 15
                tol = max(epsabs, epsrel * abs(IK))
                # Mild rescaling of err using norm (QUADPACK-like guard)
                if fnorm > 0:
                        err = min(err, 200 * math.ulp(1.0) * fnorm + err)

                if (err <= tol) or (d >= max_depth) or (used >= limit):
                        total += IK
                else:
                        m = 0.5*(u + v)
                        stack.append((m, v, d+1))
                        stack.append((u, m, d+1))
        return total
```

Quick sanity checks

```
# 1) Endpoint singularity (integrable): ∫_0^1 x^{-1/2} dx = 2
f1 = lambda x: 1.0/math.sqrt(x) if x > 0 else 0.0
print(adaptive_simpson(f1, 0.0, 1.0, 1e-10, 1e-10))
print(quadgk(f1, 0.0, 1.0, 1e-10, 1e-10))

# 2) Oscillatory: ∫_0^1 sin(50x)/x dx  (well-defined as Si(50))
f2 = lambda x: math.sin(50*x)/x if x ≠ 0 else 50.0
print(adaptive_simpson(f2, 0.0, 1.0, 1e-8, 1e-8))
print(quadgk(f2, 0.0, 1.0, 1e-8, 1e-8))

# 3) Gaussian bell: ∫_{-3}^{3} e^{-x^2} dx ≈ √π erf(3) ≈ 1.77241 …
f3 = lambda x: math.exp(-x*x)
print(adaptive_simpson(f3, -3.0, 3.0, 1e-12, 1e-10))
print(quadgk(f3, -3.0, 3.0, 1e-12, 1e-10))
```

Where each shines:

- **Adaptive Simpson:** Excellent for smooth, low-cost integrands; very few points per panel; simple, accurate error control.
- **Adaptive Gauss–Kronrod:** More function calls per panel but higher-order accuracy and a robust embedded error estimator; the go-to general-purpose choice.

***Adaptive quadrature** is like a smart photographer who takes more pictures of the interesting parts of a scene and fewer of the boring parts. It focuses computational effort where the function is most "active" or changing rapidly.*

Clenshaw–Curtis and Tanh–Sinh Quadrature

Clenshaw–Curtis quadrature approximates

$$I = \int_{-1}^{1} f(x)\,dx$$

by interpolating $f(x)$ at *Chebyshev–Lobatto nodes*

$$x_k = \cos\left(\frac{k\pi}{n}\right), \quad k = 0, 1, \ldots, n.$$

These nodes cluster near ± 1, mitigating Runge's phenomenon and capturing endpoint behaviour efficiently. The interpolant is expressed as a cosine series via the discrete cosine transform (DCT), and the integral is computed exactly for each cosine term. The resulting weights w_k can be precomputed, and because the nodes are *nested* (e.g., $n = 2^m$), previously computed function values can be reused in adaptive schemes.

The quadrature formula is:

$$I_n \;=\; \sum_{k=0}^{n} w_k\, f(x_k),$$

where w_k are derived from the cosine coefficients a_j of the interpolant:

$$a_j = \frac{2}{n}\sum_{k=0}^{n} f(x_k)\cos\left(\frac{jk\pi}{n}\right), \quad j = 0, \dots, n.$$

Integration term-by-term yields:

$$w_k = \frac{2}{n}\left[1 - \sum_{\substack{j=1\\ j\text{ even}}}^{n-1} \frac{2}{1-j^2}\cos\left(\frac{jk\pi}{n}\right)\right].$$

When to use: Smooth functions on finite intervals, especially when endpoint behaviour matters or when you want to reuse samples for increasing n. It is competitive with Gauss–Legendre in accuracy but cheaper to implement with FFT-based weight generation.

Tanh–Sinh quadrature (also called *double-exponential* or DE quadrature) excels for integrals with endpoint singularities or infinite derivatives at the boundaries. It maps $[-1,1]$ to $(-\infty,\infty)$ via:

$$x = \tanh\!\Big(\frac{\pi}{2}\sinh t\Big),$$

which causes the integrand to decay *double-exponentially* as $|t| \to \infty$. The transformed integral becomes:

$$I = \int_{-\infty}^{\infty} f\Big(\tanh\!\Big(\frac{\pi}{2}\sinh t\Big)\Big)\cdot \frac{\frac{\pi}{2}\cosh t}{\cosh^2\!\big(\frac{\pi}{2}\sinh t\big)}\,dt.$$

The Jacobian factor

$$\phi'(t) = \frac{\frac{\pi}{2}\cosh t}{\cosh^2\!\big(\frac{\pi}{2}\sinh t\big)}$$

decays so rapidly that the trapezoidal rule in t converges at an astonishing rate — often doubling the number of correct digits with each halving of the step size.

When to use: Integrals with algebraic or logarithmic endpoint singularities, or functions analytic in a strip around the real axis. Also effective for Cauchy principal value integrals after suitable symmetrisation.

Clenshaw–Curtis *is like taking measurements at points that naturally "bunch up" near the edges of your interval, where functions often misbehave. You then fit a smooth curve through those points using only cosines — which are easy to integrate — and add up the contributions.*

Tanh–Sinh *is like stretching the ends of your interval to infinity in a clever way, so any nasty spikes at the edges get squashed flat. Once flattened, you can march along with evenly spaced steps and still capture all the detail, because* *the transformation makes the function fade away super-fast.*

Why they are powerful

- **Clenshaw–Curtis:**
    - Nodes are *nested* — reuse old function values when increasing resolution.
    - FFT/DCT-based weight computation is $O(n \log n)$ vs $O(n^2)$ for Gauss–Legendre.
    - Excellent for smooth functions; competitive accuracy with Gauss rules.
- **Tanh–Sinh:**
    - Handles endpoint singularities without special-case code.
    - Double-exponential decay after transformation → extremely fast convergence.
    - Simple uniform-step trapezoidal rule in transformed space.

Mini-examples

Clenshaw–Curtis: $\int_{-1}^{1} \frac{1}{1+25x^2} dx$

Runge's function has steep changes near the ends. Chebyshev nodes cluster there, capturing the shape efficiently.

Tanh–Sinh: $\int_0^1 x^{-1/2}\, dx$

Square-root singularity at $x = 0$ is flattened by the DE transform, so the trapezoidal rule converges rapidly without special handling.

```
import numpy as np
from math import pi
from scipy.fft import dct as _dct

def clenshaw_curtis(f, a: float, b: float, n: int = 128) → float:
"""
Clenshaw–Curtis integration on [a,b] with n panels (n+1 Chebyshev–Lobatto nodes).
"""
if b == a:
        return 0.0

# Chebyshev–Lobatto nodes on [-1,1]
k = np.arange(n + 1)
x_cheb = np.cos(pi * k / n)

# Map to [a,b]
xm = 0.5 * (b + a) + 0.5 * (b - a) * x_cheb

# Sample f
y = np.array([f(x) for x in xm], dtype=float)

# DCT-I (unnormalized) → Chebyshev coefficients
c = _dct(y, type=1, norm=None)
a_cheb = c / n
```

```
a_cheb[0] *= 0.5
a_cheb[-1] *= 0.5

# Integrate cosine series on [-1,1]:
# ∫ f ≈ 2*a_0 + sum_{even m≥2} 2*a_m/(1 - m^2)
I_hat = 2.0 * a_cheb[0]
if n ≥ 2:
        m_even = np.arange(2, n + 1, 2)
        I_hat += np.sum(2.0 * a_cheb[m_even] / (1.0 - m_even**2))

# Scale to [a,b]
return 0.5 * (b - a) * I_hat


def tanh_sinh(f, a: float, b: float, epsabs: float = 1e-12, epsrel: float = 1e-12,
                  h0: float = 0.5, max_refinements: int = 12, max_terms: int =
10_000) -> float:
"""
Tanh-Sinh (double-exponential) quadrature on [a,b].
Handles endpoint singularities via x = (a+b)/2 + (b-a)/2 * tanh( (π/2) sinh t ).
"""
if b == a:
        return 0.0

def phi(t):
        u = 0.5 * pi * np.sinh(t)
        return 0.5 * (a + b) + 0.5 * (b - a) * np.tanh(u)

def phi_prime(t):
        u = 0.5 * pi * np.sinh(t)
        return 0.5 * (b - a) * (0.5 * pi * np.cosh(t)) / (np.cosh(u) ** 2)

def g(t):
        x = phi(t)
        return f(x) * phi_prime(t)

Ih_prev = None
h = h0

for _ in range(max_refinements + 1):
        # Trapezoidal on (-∞, ∞): h * [ g(0) + Σ_{k=1..K} (g(kh)+g(-kh)) ]
        S = g(0.0)
        k = 1
        tail_ok_runs = 0

        while k ≤ max_terms:
                t = k * h
                gp = g(t)
                gm = g(-t)
                S += gp + gm
                # Tail criterion: consecutive small contributions
                tail_contrib = h * (abs(gp) + abs(gm))
                tol_here = max(epsabs, epsrel * abs(h * S))
                if tail_contrib < 0.25 * tol_here:
                        tail_ok_runs += 1
                        if tail_ok_runs ≥ 3:
                                break
                else:
                        tail_ok_runs = 0

                k += 1

        Ih = h * S

        if Ih_prev is not None:
                tol_refine = max(epsabs, epsrel * abs(Ih))
                if abs(Ih - Ih_prev) ≤ tol_refine:
                        return Ih

        Ih_prev = Ih
        h *= 0.5  # refine step and repeat

return Ih_prev if Ih_prev is not None else 0.0
```

```
import math

# 1) Smooth (Clenshaw–Curtis): Runge-type on [-1,1]
f_smooth = lambda x: 1.0 / (1.0 + 25.0*x*x)
print("Clenshaw–Curtis ≈", clenshaw_curtis(f_smooth, -1.0, 1.0, n=256))

# 2) Endpoint singularity (Tanh–Sinh): ∫_0^1 x^{-1/2} dx = 2
f_sing = lambda x: 1.0 / math.sqrt(x) if x > 0 else 0.0
print("Tanh–Sinh (x^{-1/2}) ≈", tanh_sinh(f_sing, 0.0, 1.0, 1e-12, 1e-12))

# 3) Log singularity (Tanh–Sinh): ∫_0^1 log(x) dx = -1
f_log = lambda x: math.log(x) if x > 0 else 0.0
print("Tanh–Sinh (log) ≈", tanh_sinh(f_log, 0.0, 1.0, 1e-12, 1e-12))
```

Filon and Levin Quadrature for Oscillatory Integrals

Standard quadrature methods struggle with highly oscillatory integrals like:

$$\int_a^b f(x)\, e^{i\omega x}\, dx \quad \text{or} \quad \int_a^b f(x)\, \cos(\omega x)\, dx,$$

especially when ω is large. Filon and Levin quadrature are specialized techniques that exploit the structure of such integrals to achieve high accuracy without excessive sampling.

Filon Quadrature

Filon's method is designed for integrals of the form:

$$\int_a^b f(x)\,\cos(\omega x)\,dx \quad \text{or} \quad \int_a^b f(x)\,\sin(\omega x)\,dx,$$

where $f(x)$ is slowly varying and ω is large. The idea is to:

- Approximate $f(x)$ using a low-degree polynomial (typically quadratic).
- Integrate the product of this polynomial with the oscillatory function *analytically*.

This avoids the need to resolve every oscillation numerically and yields accurate results even for large ω.

Levin Quadrature

Levin's method handles more general oscillatory integrals:

$$\int_a^b f(x)\,e^{i\omega g(x)}\,dx,$$

where $g(x)$ is a known phase function and $f(x)$ is smooth. It works by:

- Rewriting the integral as a differential equation using integration by parts.
- Approximating the solution using basis functions (e.g., polynomials or splines).
- Solving a linear system to compute the integral efficiently.

Levin's method is especially powerful when $g(x)$ is nonlinear or when the oscillations are not uniform.

When to Use

- **Filon:** Best for integrals with known sinusoidal oscillations and smooth amplitude functions.
- **Levin:** Best for general oscillatory integrals with nonlinear phase or variable frequency.

```
# Filon Quadrature (simplified for cosine)
function filon_cos(f, a, b, omega):
    h = (b - a) / 2
    x0 = a
    x1 = (a + b) / 2
    x2 = b
    f0, f1, f2 = f(x0), f(x1), f(x2)

    theta = omega * h
    A = 2 * np.sin(theta) / theta
    B = 4 * np.sin(theta) / (theta**2)
    C = 2 * (np.sin(theta) - theta * np.cos(theta)) / (theta**3)
    return h * (A * f0 + B * f1 + C * f2)

# Levin Quadrature (conceptual)
function levin_integrate(f, g, a, b, omega):
    # Approximate f(x) using basis functions ϕ_j(x)
    # Solve for coefficients c_j such that:
    #   d/dx [c_j(x) * e^{iωg(x)}] ≈ f(x) * e^{iωg(x)}
    # Integrate analytically or numerically
    return sum_j c_j * ∫ ϕ_j(x) dx
```

***Imagine trying to measure a rapidly vibrating string** — if you use a ruler at fixed intervals, you'll miss the fine details. Filon's method says: "Let's fit a smooth curve to the string's shape and compute the area under the curve times the vibration." Levin's method says: "Let's understand how the vibration behaves and solve the math from that angle." Both are smarter than brute-force measuring every wiggle!*

Building Custom Quadrature Rules

The Golub-Welsch algorithm computes nodes and weights for Gaussian quadrature by solving an eigenvalue problem for the Jacobi matrix derived from orthogonal polynomial recurrence relations.

```
# Golub-Welsch Algorithm Summary
# 1) Get recurrence coefficients a_k, b_k for weight w(x)
# 2) Build Jacobi tridiagonal matrix J
# 3) Eigen-decompose J → eigenvalues x_i and eigenvectors v_i
# 4) Weights w_i = μ₀ * (v_i[0])²
```

Generalized Gaussian Quadrature (Golub–Welsch)

Gaussian quadrature rules can be generated generically from the three-term recurrence of orthogonal polynomials with respect to a weight $w(x)$ on a support $[a,b]$ (possibly infinite). The *Golub-Welsch* method constructs the symmetric tridiagonal **Jacobi matrix** from recurrence coefficients and obtains nodes as its eigenvalues and weights from the first components of its normalized eigenvectors:

J =

$$\begin{bmatrix} a_0 & \sqrt{\beta_1} & & \\ \sqrt{\beta_1} & a_1 & \sqrt{\beta_2} & \\ & \ddots & \ddots & \ddots \\ & & \sqrt{\beta_{n-1}} & a_{n-1} \end{bmatrix}, \quad x_i = \lambda_i(J), \quad w_i = \mu_0\,(v_{1i})^2$$

Here a_k and β_k come from the orthonormal polynomial recurrence for $w(x)$, $\mu_0 = \int w(x)\,dx$ is the zeroth moment, x_i are the nodes, and w_i are the weights. This single mechanism yields Gauss-Legendre, -Hermite, -Laguerre, -Jacobi, and more.

Plain-speak: *If you know how a function is "weighted" over an interval, you can build a tiny matrix whose eigenvalues tell you exactly where to sample, and how much each sample "counts." That's why Gaussian rules are so accurate with so few points.*

```
# Generalized Gaussian quadrature via Golub-Welsch
# Families: Jacobi (incl. Legendre/Chebyshev), Laguerre, Hermite
# Utilities: map to finite intervals, integrate with given nodes/weights

import numpy as np
from math import gamma, sqrt, pi

def golub_welsch(a: np.ndarray, b: np.ndarray, mu0: float):
        """
        Nodes/weights from orthonormal three-term recurrence:
          p_{k+1}(x) = (x - a_k) p_k(x) - β_k p_{k-1}(x),  β_k>0
        Inputs:
          a: (n,)   diagonal coeffs a_k
          b: (n-1,) positive coeffs β_k, k=1..n-1
          mu0: zeroth moment ∫ w(x) dx over the support
        Returns:
          x: (n,) nodes, eigenvalues of Jacobi matrix
          w: (n,) weights, mu0 * (first-eigenvector-components)^2
        """
        n = len(a)
        if len(b) ≠ n - 1:
                raise ValueError("b must have length n-1.")
        J = np.zeros((n, n), dtype=float)
        np.fill_diagonal(J, a)
        off = np.sqrt(b)
        np.fill_diagonal(J[1:], off)
        np.fill_diagonal(J[:, 1:], off)
        x, V = np.linalg.eigh(J)
        w = mu0 * (V[0, :] ** 2)
        return x, w
```

```
# ────────── Families ──────────

def gauss_jacobi(n: int, alpha: float, beta: float):
        """
        On [-1,1], weight (1-x)^alpha (1+x)^beta, alpha,beta>-1.
        Exactness: polynomials up to degree 2n-1 under this weight.
        """
        if alpha ≤ -1 or beta ≤ -1:
                raise ValueError("alpha and beta must be > -1.")
        k = np.arange(n, dtype=float)
        a = (beta**2 - alpha**2) / ((2*k + alpha + beta) * (2*k + alpha + beta +
2.0))
        km = np.arange(1, n, dtype=float)
        num = 4.0 * km * (km + alpha) * (km + beta) * (km + alpha + beta)
        den = (2*km + alpha + beta)**2 * (2*km + alpha + beta + 1.0) * (2*km + alpha
+ beta - 1.0)
        b = num / den
        mu0 = 2.0**(alpha + beta + 1.0) * gamma(alpha + 1.0) * gamma(beta + 1.0) /
gamma(alpha + beta + 2.0)
        return golub_welsch(a, b, mu0)

def gauss_legendre(n: int):
        """Legendre = Jacobi(alpha=0, beta=0) on [-1,1], weight 1."""
        return gauss_jacobi(n, 0.0, 0.0)

def gauss_hermite(n: int):
        """Hermite, weight exp(-x^2) on (-inf, inf)."""
        a = np.zeros(n, dtype=float)
        b = 0.5 * np.arange(1, n, dtype=float) if n > 1 else np.array([],
dtype=float)
        mu0 = sqrt(pi)  # ∫ exp(-x^2) dx
        return golub_welsch(a, b, mu0)

def gauss_laguerre(n: int, alpha: float = 0.0):
        """Laguerre, weight x^alpha * exp(-x) on [0, inf), alpha>-1."""
        if alpha ≤ -1:
                raise ValueError("alpha must be > -1.")
        a = 2.0 * np.arange(n, dtype=float) + alpha + 1.0
        b = np.arange(1, n, dtype=float) * (np.arange(1, n, dtype=float) + alpha) if
n > 1 else np.array([], dtype=float)
        mu0 = gamma(alpha + 1.0)
        return golub_welsch(a, b, mu0)

# ────────── Utilities ──────────

def map_to_interval(x: np.ndarray, w: np.ndarray, a: float, b: float):
        """Affine map from [-1,1] to [a,b]."""
        xm = 0.5 * (b - a) * x + 0.5 * (a + b)
        wm = 0.5 * (b - a) * w
        return xm, wm
```

```
def integrate(f, x: np.ndarray, w: np.ndarray):
        """Compute sum_i w_i f(x_i) for the given rule."""
        return np.dot(w, np.vectorize(f)(x))
```

Usage Examples

```
# 1) Legendre on [-1,1]: ∫ x^4 dx = 2/5, exact with n=3
x, w = gauss_legendre(3)
print(integrate(lambda t: t**4, x, w))  # → 0.4

# Map Legendre to [0,2] and integrate x^2
xg, wg = gauss_legendre(4)
xm, wm = map_to_interval(xg, wg, 0.0, 2.0)
print(np.dot(wm, xm**2))  # → 8/3 ≈ 2.666666 ...

# 2) Hermite: ∫_{-∞}^{∞} x^2 e^{-x^2} dx = √π / 2, exact with n=2
xh, wh = gauss_hermite(2)
print(integrate(lambda t: t**2, xh, wh))  # → sqrt(pi)/2

# 3) Laguerre (alpha=0): ∫_0^∞ x^2 e^{-x} dx = Γ(3) = 2, exact with n=2
xl, wl = gauss_laguerre(2, alpha=0.0)
print(integrate(lambda t: t**2, xl, wl))  # → 2.0

# 4) Jacobi as Chebyshev (first kind): α=β=-1/2 on [-1,1]
xj, wj = gauss_jacobi(5, alpha=-0.5, beta=-0.5)
print(integrate(lambda t: 1.0, xj, wj))  # → π
```

Notes & Best Practices:

- **Match the weight:** These rules integrate $f(x)\,w(x)$. For plain $\int f(x)\,dx$ on finite intervals, use Legendre and map to $[a,b]$.
- **Exactness:** Degree $2n-1$ (with respect to the weight). Increase n for non-polynomial f.
- **Stability:** Use orthonormal recurrences and symmetric tridiagonals (as here) for robust nodes/weights.
- **Performance:** Vectorize f and reuse nodes/weights across integrals; precompute per interval/model.

Generalized (Non-Gaussian) Quadrature Creation

Gaussian quadrature is optimal when the weight function $w(x)$ is one of the classical orthogonal polynomial families (Legendre, Hermite, Laguerre, Jacobi, etc.). But in many applications — from heavy-tailed finance models to skewed probability densities in physics — the natural weight function does not match these classical forms.

Generalized quadrature refers to constructing nodes x_i and weights w_i for *any* given weight function $w(x)$ on a domain Ω, so that:

$$\int_\Omega f(x)\,w(x)\,dx \approx \sum_{i=1}^{n} w_i\,f(x_i)$$

The process typically involves:

1. **Defining the weight function:** $w(x)$ may be a PDF, a physical kernel, or any known shape.
2. **Choosing a node strategy:** Quantile-based placement (via inverse CDF), clustering where $w(x)$ is large, or solving orthogonal polynomial recurrences for $w(x)$.
3. **Computing weights:** Integrating $w(x)$ over Voronoi cells around each node, or solving a moment-matching system so the rule is exact for a chosen basis (e.g., polynomials up to degree m).
4. **Normalizing:** If $w(x)$ is a PDF, ensure $\sum w_i = 1$.

This approach generalizes the Golub–Welsch idea: instead of using pre-tabulated recurrence coefficients for classical weights, you derive them numerically for your custom $w(x)$, or bypass them entirely with direct bin-integration.

Think of it like designing a custom fishing net: *Gaussian quadrature nets are pre-made for certain fish (functions) in certain waters (weight functions). But if you're fishing in a strange new sea — say, one full of heavy-tailed monsters or oddly-shaped schools — you weave your own net to match the catch. You decide where the knots (nodes) go and how big each mesh (weight) should be, so you scoop up the most important parts without wasting effort.*

Key Insight: Generalized quadrature lets you integrate efficiently under *any* weight function — from Student-t to Beta to exotic kernels — by matching the rule to the shape of the problem. This is especially powerful in finance, physics, and statistics, where the "natural" weight is rarely one of the textbook cases.

Example: Custom Quadrature for a Student-t Distribution

Sometimes the integrand's natural weight function is not one of the classical Gaussian families. In such cases, we can build a *custom quadrature rule* tuned to that weight. For example, suppose our integration is naturally weighted by the PDF of a Student-t distribution with ν degrees of freedom:

$$w(x) = f_{\text{Student-t},\nu}(x)$$

We can:

1. Select nodes as quantiles of the distribution (via its inverse CDF), avoiding extreme tails.
2. Compute weights by integrating the PDF over small bins around each node.
3. Normalise the weights so they sum to 1 (if w is a PDF).

This produces a quadrature rule that concentrates effort where the distribution has most of its mass, improving efficiency for heavy-tailed or skewed weights.

```
# Custom Quadrature for any distribution with PDF and PPF
function custom_quadrature(pdf, ppf, N, tail_cut):
        percentiles = linspace(tail_cut, 1 - tail_cut, N)
        nodes = ppf(percentiles)
        weights = zeros(N)
        for i in 0..N-1:
                left  = midpoint(nodes[i], nodes[i-1]) if i > 0 else -∞
                right = midpoint(nodes[i], nodes[i+1]) if i < N-1 else +∞
                weights[i] = integrate(pdf, left, right)
        weights /= sum(weights)
        return nodes, weights

# Example: Student-t with v=5
pdf = lambda x: t_pdf(x, nu=5)
ppf = lambda p: t_ppf(p, nu=5)
nodes, weights = custom_quadrature(pdf, ppf, N=50, tail_cut=0.05)
```

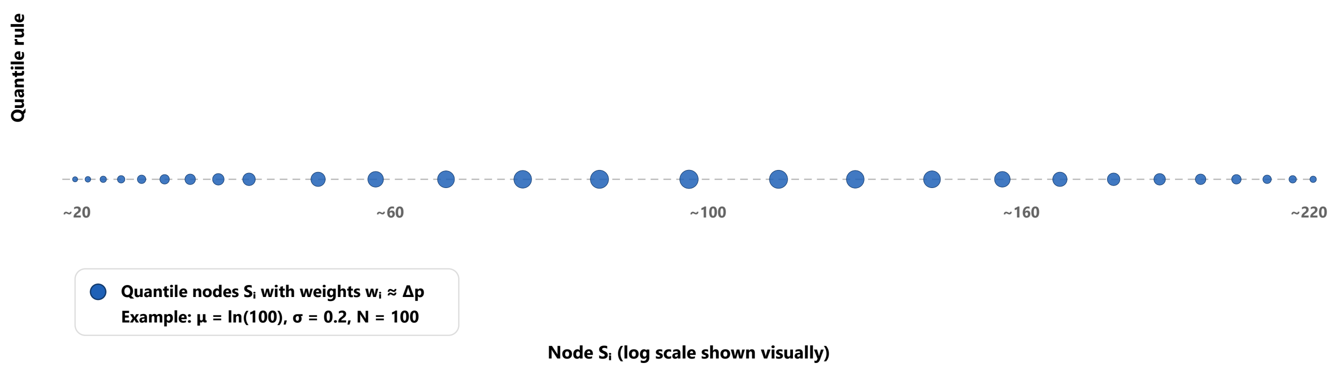
Think of it like taste-testing soup: If you know where the flavour is concentrated — the spice pockets, the chunky bits — you don't take random sips. You sample exactly from those spots and weigh them according to how much they contribute to the whole pot. That's what a custom quadrature rule does: it samples where the action is.

Key Insight: Custom quadrature rules let you integrate efficiently under *any* distribution — Student-t, Beta, Gamma, skew-normal — by matching the nodes and weights to the shape of the weight function.

Quantile-based Custom Quadrature (Lognormal)

```
# Goal: approximate E[f(S)] where S ~ Lognormal(μ, σ) without Monte Carlo.
# Idea: Use the quantile transform S = F^{-1}(p), p ~ Uniform(0,1).
# Then: E[f(S)] = ∫_0^1 f(F^{-1}(p)) dp  ≈  Σ w_i f(F^{-1}(p_i))

INPUT:
  μ, σ                 # log-space mean & vol (Black-Scholes: μ = ln(S0) + (r - q - ½σ²)T, σ = σ√T)
  N                    # number of nodes (e.g., 100)
  tail_cut ∈ (0,1)  # avoid extreme tails, e.g., 1e-4
```

```
CONSTRUCT NODES (uniform in probability):
  p_i = linspace(tail_cut, 1 - tail_cut, N)
  x_i = Φ^{-1}(p_i)                # standard normal quantile (any accurate approx)
  S_i = exp(μ + σ x_i)             # lognormal nodes

WEIGHTS (on p-space, sum to ~1):
  For i = 0..N-1:
    if i == 0:     w_i = (p_1 - p_0)/2
    elif i == N-1: w_i = (p_{N-1} - p_{N-2})/2
    else:          w_i = (p_{i+1} - p_{i-1})/2
  Normalize: w_i = w_i / Σ_j w_j

ESTIMATE EXPECTATION:
  E[f(S)] ≈ Σ_i w_i * f(S_i)

NOTES:
- This targets the mass where it lives (efficient for skewed/heavy tails).
- No PDF factor is needed because dp “is” the weight measure.
- Replace Lognormal by any distribution with PPF (quantile) available numerically.
```



Custom quantile rule for Lognormal: take evenly spaced probabilities p_i, map through the PPF to nodes S_i, and use Δp weights.

Creating your own quadrature rule is like designing a custom measuring tool for a specific shape. If you know something about what you're measuring (like where it's bumpy or flat), you can place your measurement points strategically to get accurate results with fewer measurements.

Key Insight: More knowledge about the function → fewer nodes needed. Clever node placement → higher accuracy with same computation.

Modern Quadrature Architectures

Contemporary quadrature implementations leverage parallel computing architectures:

- GPU acceleration for evaluating integrands at multiple nodes simultaneously

- Multi-core processing for dividing integration domains
- Distributed computing for high-dimensional integrals

***Modern quadrature** is like having a team of people measure a large field together instead of one person doing all the work. GPUs are like having hundreds of helpers working in parallel, making the process much faster for complex problems.*

GPU-Accelerated Quadrature with CUDA

The core idea is to map each node evaluation $f(x_i)$ to an independent GPU thread. This is ideal for embarrassingly parallel quadrature where nodes do not depend on each other. The GPU computes values in parallel; the host performs the weighted reduction.

```
# ═════════════════════════
# CUDA C++ (single GPU)
# ═════════════════════════

__device__ double f(double x) {
        // device-compatible integrand
        return …;
}

__global__ void eval_integrand(const double* __restrict__ nodes,
                                                           double* __restrict__ out,
                                                           int N) {
        int i = blockIdx.x * blockDim.x + threadIdx.x;
        if (i < N) out[i] = f(nodes[i]);
}

double quad_gpu(const double* nodes_h, const double* weights_h, int N) {
        // 1) Allocate on device
        double *nodes_d, *vals_d;
        cudaMalloc(&nodes_d, N * sizeof(double));
        cudaMalloc(&vals_d,  N * sizeof(double));

        // 2) H2D copy
        cudaMemcpy(nodes_d, nodes_h, N * sizeof(double), cudaMemcpyHostToDevice);

        // 3) Launch kernel
        int TPB = 256;
        int BPG = (N + TPB - 1) / TPB;
        eval_integrand<<>>(nodes_d, vals_d, N);

        // 4) D2H copy and weighted sum on host
        std::vector vals(N);
        cudaMemcpy(vals.data(), vals_d, N * sizeof(double), cudaMemcpyDeviceToHost);

        double sum = 0.0;
        for (int i = 0; i < N; ++i) sum += weights_h[i] * vals[i];

        cudaFree(nodes_d); cudaFree(vals_d);
        return sum;
}
```

```
# ═════════════════════════
# Python (Numba CUDA, single GPU)
# ═════════════════════════
from numba import cuda
import numpy as np

@cuda.jit(device=True)
def f(x):
        # device-compatible integrand
        return  ...

@cuda.jit
def eval_integrand(nodes, out):
        i = cuda.grid(1)
        if i < nodes.size:
                out[i] = f(nodes[i])

def quad_gpu(nodes, weights):
        N = len(nodes)
        d_nodes  = cuda.to_device(nodes)
        d_vals   = cuda.device_array(N, dtype=np.float64)

        TPB = 256
        BPG = (N + TPB - 1) // TPB
        eval_integrand[BPG, TPB](d_nodes, d_vals)

        vals = d_vals.copy_to_host()
        return np.dot(weights, vals)
```

Analogy: *Think of a giant kitchen with hundreds of chefs. Each chef cooks one dish (node) at the same time. You just plate the dishes (sum with weights) and serve the final result.*

Multi-GPU Quadrature for Huge Jobs

> When a single GPU is not enough (memory or throughput), split the node set into shards and assign each shard to a different GPU. Each GPU computes its partial weighted sum; the host aggregates the partials. Keep shards balanced to maximize utilization.

```
# ═══════════════════════
# CUDA C++ (multi-GPU, single node)
# ═══════════════════════

double quad_multi_gpu(const double* nodes, const double* weights,
                                          long long N, int num_gpus) {
        long long chunk = (N + num_gpus - 1) / num_gpus;
        std::vector partial(num_gpus, 0.0);

        parallel_for (int g = 0; g < num_gpus; ++g) {
                cudaSetDevice(g);
                long long start = g * chunk;
                long long end   = std::min(start + chunk, N);
                long long M     = std::max(0LL, end - start);
                if (M == 0) { partial[g] = 0.0; continue; }

                // Device alloc
                double *d_nodes, *d_vals;
                cudaMalloc(&d_nodes, M * sizeof(double));
                cudaMalloc(&d_vals,  M * sizeof(double));

                // Copy shard of nodes
                cudaMemcpy(d_nodes, nodes + start, M * sizeof(double),
cudaMemcpyHostToDevice);

                // Launch kernel
                int TPB = 256, BPG = (int)((M + TPB - 1) / TPB);
                eval_integrand<<>>(d_nodes, d_vals, (int)M);

                // Copy back and reduce on host
                std::vector vals(M);
                cudaMemcpy(vals.data(), d_vals, M * sizeof(double),
cudaMemcpyDeviceToHost);

                double s = 0.0;
                for (long long i = 0; i < M; ++i) s += weights[start + i] * vals[i];
                partial[g] = s;

                cudaFree(d_nodes); cudaFree(d_vals);
        }

        // Host aggregate
        double total = 0.0;
        for (int g = 0; g < num_gpus; ++g) total += partial[g];
        return total;
}
```

```
# ═════════════════════════
# Python (Numba CUDA, multi-GPU, single node)
# ═════════════════════════
from numba import cuda
import numpy as np

@cuda.jit
def eval_integrand(nodes, out):
        i = cuda.grid(1)
        if i < nodes.size:
                out[i] = f(nodes[i])

def quad_multi_gpu(nodes, weights, num_gpus):
        N = len(nodes)
        chunk = int(np.ceil(N / num_gpus))
        partials = []

        for g in range(num_gpus):
                cuda.select_device(g)
                start, end = g*chunk, min((g+1)*chunk, N)
                if end ≤ start: partials.append(0.0); continue

                nodes_g = nodes[start:end]
                weights_g = weights[start:end]

                d_nodes  = cuda.to_device(nodes_g)
                d_vals   = cuda.device_array(len(nodes_g), dtype=np.float64)

                TPB = 256
                BPG = (len(nodes_g) + TPB - 1) // TPB
                eval_integrand[BPG, TPB](d_nodes, d_vals)

                vals = d_vals.copy_to_host()
                partials.append(np.dot(weights_g, vals))

        return float(np.sum(partials))
```

Best practices:

- **Shard evenly:** Balance work to avoid idle GPUs.
- **Batch large jobs:** Process nodes in tiles that fit device memory.

- **Overlap transfers:** Use pinned memory, cudaMemcpyAsync, and streams to overlap copy/compute.
- **Stable reduction:** For very large N, do hierarchical Kahan or pairwise sums.

Let's Get Colossal: Multi-Node, Multi-GPU with MPI + CUDA

Scale across a cluster by assigning each Message Passing Interface (MPI) rank a GPU (1 rank ↔ 1 GPU). Partition the workload globally across ranks so that each GPU processes a distinct subset of data or quadrature nodes. Each rank computes a partial weighted sum locally on its GPU, and then participates in a collective *all-reduce* operation to combine these partial results into the final integral. For systems with multiple GPUs per node, use NCCL (NVIDIA Collective Communications Library) to perform fast intra-node reductions, followed by MPI for inter-node reductions across the cluster. This hierarchical strategy minimizes communication overhead and scales efficiently from a single node to large multi-node GPU clusters.

Imagine you have many calculators (GPUs), each held by a different person (MPI ranks) in a big team spread across different rooms (nodes). Each person is given a piece of the problem to solve, and they use their calculator to work out their share. When everyone is done, they all shout out their answers into a system that adds everything together (the all-reduce). If a room has multiple people with calculators, they first whisper to each other to combine their results quickly (NCCL within a node), and then one representative from the room shares the total with the rest of the team across all rooms (MPI across nodes). This way, the work gets done much faster than if only one person did all the calculations.

```
# ═══════════════════════════════════════════
# MPI + CUDA (C++-style design, 1 rank ↔ 1 GPU)
# ═══════════════════════════════════════════

int main(int argc, char** argv) {
        MPI_Init(&argc, &argv);
        int world_size, world_rank;
        MPI_Comm_size(MPI_COMM_WORLD, &world_size);
        MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);

        // Map rank to local GPU (e.g., via local_rank env)
        int local_rank = get_local_rank();           // implementation-specific
        cudaSetDevice(local_rank);

        // Global partition
        long long N = total_nodes();
        long long chunk = (N + world_size - 1) / world_size;
        long long start = world_rank * chunk;
        long long end   = std::min(start + chunk, N);
        long long M     = std::max(0LL, end - start);

        // Tile loop for out-of-core workloads
        double partial_sum = 0.0;
        for (long long off = 0; off < M; off += TILE) {
                long long m = std::min((long long)TILE, M - off);

                // Stage tile nodes/weights (H2D async with pinned host memory)
                cudaMemcpyAsync(d_nodes, h_nodes + start + off, m*sizeof(double),
H2D, stream);
                eval_integrand<<>>(d_nodes, d_vals, (int)m);
                cudaMemcpyAsync(h_vals, d_vals, m*sizeof(double), D2H, stream);
                cudaStreamSynchronize(stream);

                // Host-side weighted sum (or device-side reduction kernel)
                partial_sum += weighted_sum(h_vals, h_weights + start + off, m);
        }

        // Global reduction
        double total = 0.0;
        MPI_Allreduce(&partial_sum, &total, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);

        // Final result on all ranks
        if (world_rank == 0) print(total);

        MPI_Finalize();
}
```

```
# ═══════════════════════════════════════════
# Python (mpi4py + Numba CUDA), 1 rank ↔ 1 GPU
# ═══════════════════════════════════════════
from mpi4py import MPI
from numba import cuda
import numpy as np

@cuda.jit
def eval_integrand(nodes, out):
        i = cuda.grid(1)
        if i < nodes.size:
                out[i] = f(nodes[i])

def quad_mpi_cuda(all_nodes, all_weights):
        comm = MPI.COMM_WORLD
        size = comm.Get_size()
        rank = comm.Get_rank()

        # Rank→GPU mapping (e.g., modulo GPUs per node)
        gpu_id = rank % cuda.gpus.lst.size
        cuda.select_device(gpu_id)

        N = len(all_nodes)
        chunk = (N + size - 1) // size
        start, end = rank*chunk, min((rank+1)*chunk, N)

        partial = 0.0
        TILE = 1_000_000  # tune to memory
        for off in range(start, end, TILE):
                hi = min(off + TILE, end)
                nodes = all_nodes[off:hi]
                weights = all_weights[off:hi]

                d_nodes = cuda.to_device(nodes)
                d_vals  = cuda.device_array(nodes.size, dtype=np.float64)

                TPB = 256
                BPG = (nodes.size + TPB - 1)//TPB
                eval_integrand[BPG, TPB](d_nodes, d_vals)
                vals = d_vals.copy_to_host()

                partial += float(np.dot(weights, vals))

        total = comm.allreduce(partial, op=MPI.SUM)
        return total
```

Conceptual tips for Colossal Titans:

- **Ranks and GPUs:** Use one MPI (*Message Passing Interface*) rank per GPU (*Graphics Processing Unit*); bind ranks to GPUs deterministically (e.g., `local_rank`, which is the rank ID within a node).
- **Two-tier reduction:** Intra-node (NCCL = *NVIDIA Collective Communications Library* all-reduce) then inter-node (MPI Allreduce) for efficiency.
- **Overlap:** Use pinned host buffers (*page-locked host memory for faster transfer*) + multiple CUDA streams (*asynchronous command queues on the GPU*) to overlap H2D (*host-to-device*) / D2H (*device-to-host*) transfers with compute.
- **Chunking:** Tile nodes when they exceed device memory; pipeline tiles per stream to keep GPUs busy.
- **Numerical care:** Use pairwise summation (*tree-like addition order*) or Kahan summation (*compensated arithmetic to reduce round-off error*) for large reductions; validate against CPU (*Central Processing Unit*) baselines.
- **Load balance:** If f(x) cost varies across x, use work-stealing (*idle ranks grab extra work*) or dynamic shards (*divide workload adaptively*) to avoid stragglers.

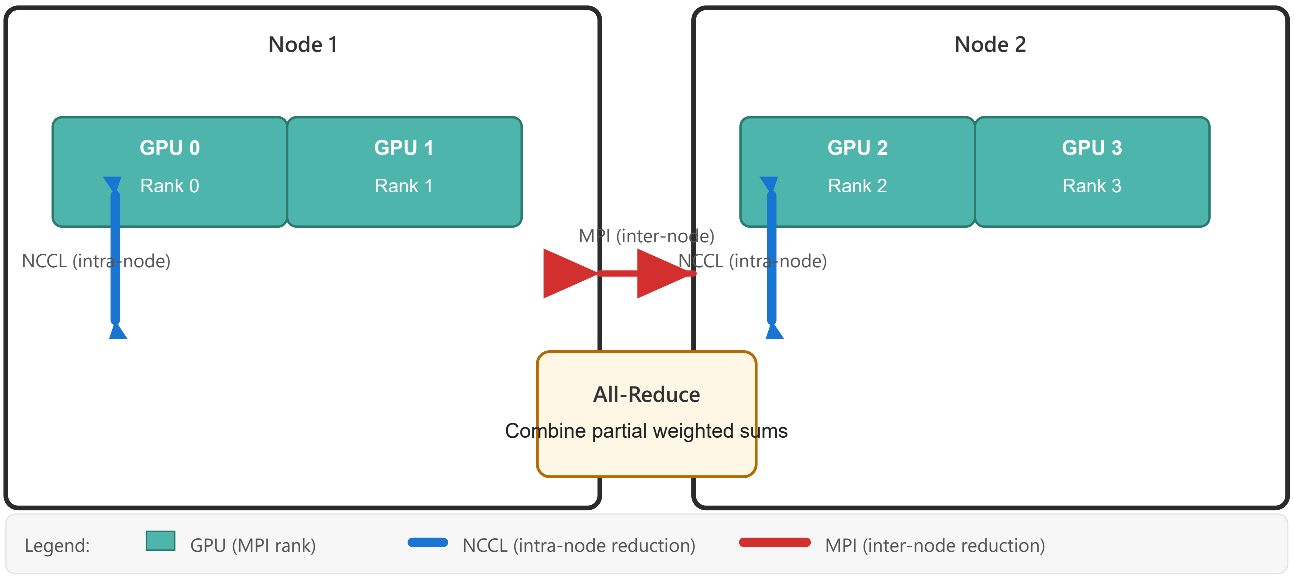
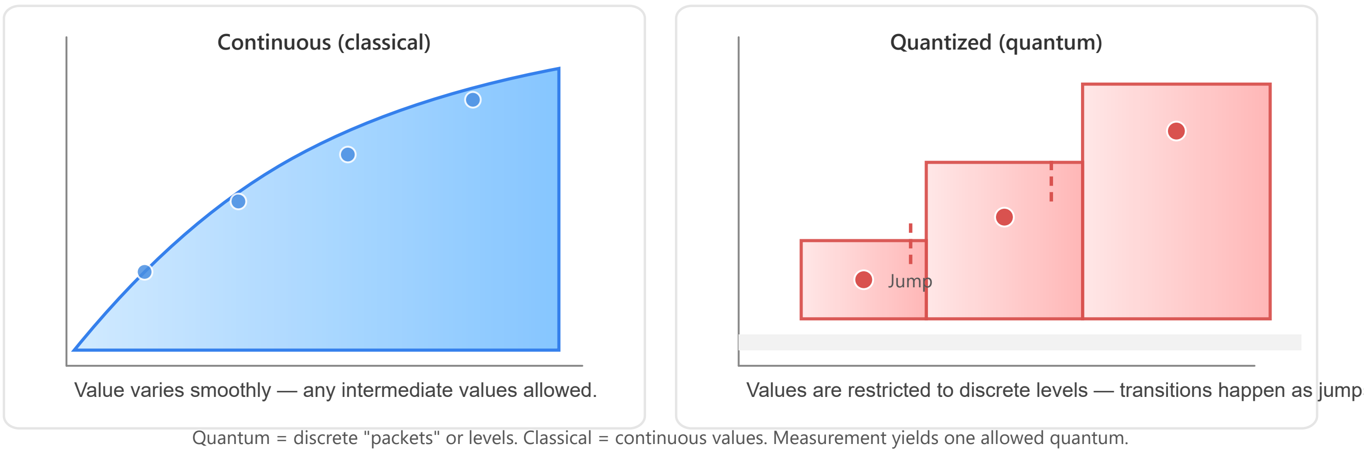


Diagram: GPUs are mapped 1:1 with MPI ranks; intra-node reductions use NCCL for speed and an inter-node MPI all-reduce completes the global aggregation. Drop this SVG into your HTML — it is scalable and editable.

> ***Analogy:*** *Imagine mapping an entire continent with many fleets of drones. Each ship (node) carries drones (GPUs), each drone scans a region (shard). Ships first merge their maps locally, then all ships merge their maps together into one seamless atlas.*

Past infinity, past certainty — into the (purely conceptual) quantum fold of Quadrature

> **Author's note:** This "Quantum Quadrature" section describes an imaginary concept — how quantum integration techniques *could* perhaps be applied to numerical quadrature in finance and other fields. While the underlying algorithms (e.g., quantum amplitude estimation) are real and actively researched, large-scale, production-ready implementations are not yet available on current quantum hardware. This is a projection of where the technology may head, rather than a description of tools you can use today.


Continuous (classical)
Value varies smoothly — any intermediate values allowed.
Quantized (quantum)
Jump
Values are restricted to discrete levels — transitions happen as jump

Quantum = discrete "packets" or levels. Classical = continuous values. Measurement yields one allowed quantum.

Illustration: left = continuous variable (any value allowed), right = quantized levels (discrete steps).

```
Classical quadrature rules — even when accelerated on GPUs or clusters — still
evaluate the integrand at a finite set of nodes, summing weighted function values to
```

approximate the integral. **Quantum quadrature** takes a fundamentally different approach: it encodes the integration problem into the amplitudes of a quantum state and applies *quantum amplitude estimation* (QAE) to extract the expectation value with *quadratically fewer* samples than classical Monte Carlo methods require. Instead of looping over nodes one by one, a quantum computer can, in principle, prepare a superposition over all nodes simultaneously, evaluate the integrand across that superposition, and use interference to read out the mean value.

In theory, this could make certain high-dimensional or computationally expensive integrals tractable, especially when the integrand can be efficiently encoded as a quantum oracle and the weight function can be prepared as a quantum state. However, large-scale, production-ready implementations of such "quantum quadrature" are not yet available on current hardware — this is a forward-looking concept based on existing quantum algorithms like QAE, rather than a tool you can run today.

In practical terms, *quantum amplitude estimation* works by preparing a quantum state whose amplitudes reflect the probability distribution of interest, then using controlled interference to estimate the average value of the integrand with far fewer oracle calls than classical sampling would require. The use of *superposition* means the quantum processor can, in a single logical step, represent and evolve all node evaluations at once — a form of parallelism fundamentally different from splitting work across classical cores or GPUs. This theoretical advantage is why quantum quadrature is attracting attention for problems like multi-dimensional Monte Carlo in finance or physics, where classical cost grows prohibitively. That said, today's quantum devices remain limited in qubit count, coherence time, and error rates, so these methods are still in the research and prototyping stage rather than in production deployment.

Term Explainers

Quantum Oracle: In quantum computing, an oracle is a special-purpose quantum circuit that encodes a function $f(x)$ into the state of your qubits. It's a "black box" you can query inside a quantum algorithm without knowing its internal details, but it must be *unitary* (reversible) so it can run forwards and backwards. For integration problems, the oracle is built to mark amplitudes or phases according to $f(x)$, so that algorithms like Quantum Amplitude Estimation can extract the average value efficiently.

Layman's view: *Think of it as a sealed vending machine that, when you press a button labelled x, instantly lights up the answer $f(x)$. In a quantum setting, you can press all the buttons at once in superposition, and the machine lights up all the answers in parallel.*

Qubits: A qubit (quantum bit) is the basic unit of quantum information. Unlike a classical bit, which is either 0 or 1, a qubit can be in a *superposition* of 0 and 1:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

where α and β are complex amplitudes with $|\alpha|^2 + |\beta|^2 = 1$. Multiple qubits can also be *entangled*, meaning their states are linked in ways that have no classical counterpart.

Layman's view: *If a classical bit is like a coin lying flat — heads (0) or tails (1) — a qubit is like a spinning coin that's both heads and tails until you catch it. And if you spin two coins together in just the right way, they can become entangled — flip one, and the other "knows" instantly, no matter how far apart they are.*

Given a weight function $w(x)$ (often a probability density) and an integrand $f(x)$, we can write:

$$I = \int_{\Omega} f(x)w(x)\,dx$$

The quantum procedure:

1. **State preparation:** Prepare a quantum state $|\psi_w\rangle = \sum_x \sqrt{w(x)}\,|x\rangle$ so that measuring yields samples from $w(x)$.
2. **Function encoding:** Implement $f(x)$ as a quantum oracle U_f that encodes $f(x)$ into an ancilla qubit's amplitude or phase.
3. **Amplitude estimation:** Use QAE to estimate the mean value of $f(x)$ over $w(x)$ with $O(1/\epsilon)$ complexity instead of $O(1/\epsilon^2)$.
4. **Result extraction:** The estimated amplitude corresponds to the integral value (up to known scaling factors).

This approach is, if sound, quite promising for high-dimensional integrals in finance, physics, and Bayesian inference, where classical methods struggle.

***Think of quantum quadrature** like tasting every spoonful of soup at once in a quantum superposition. Instead of sampling one spoonful at a time, you prepare a magical state that contains the flavour of the whole pot. Then, with a clever interference trick, you read out the average flavour with far fewer sips than any classical chef could manage.*

Pseudocode: Quantum Amplitude Estimation for Integration

```
# Quantum Quadrature via Amplitude Estimation (conceptual)

Given:
        w(x)  - weight function (PDF)
        f(x)  - integrand
        ε     - target error

1. Discretize domain Ω into M basis states |x_j⟩
2. Prepare |ψ_w⟩ = Σ_j sqrt(w(x_j)) |x_j⟩
   # State preparation circuit encodes w(x) into amplitudes

3. Define oracle U_f:
   |x_j⟩|0⟩ → |x_j⟩( sqrt(1 - f(x_j))|0⟩ + sqrt(f(x_j))|1⟩ )
   # Encodes f(x_j) into amplitude of ancilla qubit

4. Construct Grover-like operator Q from U_f and |ψ_w⟩

5. Apply Quantum Amplitude Estimation:
   - For k = 0..m-1:
           Apply controlled-Q^(2^k) to |ψ_w⟩
   - Perform inverse Quantum Fourier Transform on control register
   - Measure to obtain estimate â of amplitude a
```

```
6. Return I ≈ â  (scaled appropriately if w(x) not normalized)
```

Key Advantages

- **Quadratic speedup** in sample complexity over classical Monte Carlo.
- Natural fit for integrals expressed as expectations under a known distribution $w(x)$.
- Potential to handle very high-dimensional problems where classical quadrature is infeasible.

Challenges

- **State preparation:** Efficiently encoding arbitrary $w(x)$ into amplitudes is non-trivial.
- **Oracle construction:** $f(x)$ must be implemented as a quantum circuit with low depth.
- **Hardware limits:** Current 'Noisy Intermediate-Scale Quantum' (NISQ) devices have noise and qubit count constraints.
- **Discretization:** Continuous domains must be mapped to finite qubit registers.

Key Insight: Quantum quadrature doesn't replace classical methods today — but it points to a future where integrals that are currently intractable could be estimated efficiently by exploiting quantum parallelism and amplitude estimation. What's real and in active use right now is *Quantum Monte Carlo* — though the term means different things in different contexts. In physics and chemistry, it refers to powerful *classical* stochastic algorithms for simulating quantum systems. In quantum computing theory, it describes algorithms like *Quantum Amplitude Estimation* that, in principle, can accelerate Monte Carlo–style integration by a quadratic factor. The former is a mature, widely deployed tool; the latter is a 'maybe' approach awaiting hardware capable of running it at scale.

Quantum coffee break: *I tried to make a quantum leap into another dimension... but ended up in the kitchen instead, with a cup noodle and coffee. I have no recollection of anything. Apparently my wavefunction collapsed towards the microwave. That said, let's resume with actual 'our'-worldly algorithmic applications and methods.*

Option Pricing with Quadrature Methods

Black-Scholes Model

The Black-Scholes model prices options under geometric Brownian motion:

$$C = S_0 N(d_1) - Ke^{-rT} N(d_2)$$

where

$$d_1 = \frac{\ln(S_0/K) + (r + \sigma^2/2)T}{\sigma\sqrt{T}}, \quad d_2 = d_1 - \sigma\sqrt{T}$$

Gauss-Hermite for Option Pricing

Using the identity for normal distribution:

$$C = e^{-rT} \int_{-\infty}^{\infty} \max(S_0 e^{(r-\sigma^2/2)T + \sigma\sqrt{T}x} - K, 0)\phi(x)dx$$

where $\phi(x)$ is the standard normal density. Gauss-Hermite quadrature approximates this as:

$$C \approx e^{-rT} \sum_{i=1}^{n} w_i \max(S_0 e^{(r-\sigma^2/2)T + \sigma\sqrt{T}x_i} - K, 0)$$

Strike-Aware Formulation

Rather than lazily integrating over the entire real line, we can more intelligently integrate from the strike price to infinity (the value domain):

$$C = e^{-rT} \int_{\ln(K/S_0)}^{\infty} (S_0 e^y - K) f(y) dy$$

where $f(y)$ is the log-normal density function.

```
# Strike-aware Gauss-Legendre Pseudocode
sigsqrt = sigma * sqrt(T)
mu = (r - q) * T + 0.5 * sigma^2 * T
z_star = -(ln(S0/K) + mu) / sigsqrt
a = z_star
b = z_star + 10 * sigsqrt  # Upper bound

x_gl, w_gl = gauss_legendre_nodes_weights(n)
sum = 0
for i in 1..n:
    z = 0.5*(b-a)*x_gl[i] + 0.5*(a+b)
    S = S0 * exp(mu + sigsqrt * z)
    payoff = max(S - K, 0)
    density = (1/sqrt(2*pi)) * exp(-0.5*z*z)
    sum += w_gl[i] * payoff * density
C = exp(-r*T) * 0.5*(b-a) * sum
```

***Strike-aware quadrature** is like knowing exactly where the treasure is buried and digging only in that area instead of searching the entire island. It's more efficient because you focus your efforts where it matters most. And for that we need an experienced scout or treasure hunter holding a Geiger Counter, not a radar.*

Carr-Madan Quadrature Formulation

The Carr-Madan approach prices calls by working in the Fourier domain with an exponentially *damped* payoff to ensure integrability. Let $k = \ln K$ and choose $\alpha > 0$ such that the damped call $e^{\alpha k} C(T, K)$ has a finite Fourier transform. Define the log-price characteristic function under the risk-neutral measure, $\varphi(u) = \mathbb{E}[e^{iu \ln S_T}]$. Then a common quadrature form is

$$C(T,K) = \frac{e^{-\alpha k}}{\pi} \int_0^\infty \Re\{e^{-iuk}\,\psi(u)\}\,du, \quad \psi(u) = \frac{e^{-rT}\,\varphi(u - i(\alpha+1))}{\alpha^2 + \alpha - u^2 + i(2\alpha+1)u}.$$

Practical details:

- **Damping:** Typical choices $\alpha \in [1, 2]$ stabilize the integral across many models (Black-Scholes, Heston, Variance-Gamma, etc.).
- **Oscillations:** Use the real-imaginary split $\Re\{e^{-iuk}\psi(u)\} = \cos(uk)\Re\psi(u) + \sin(uk)\Im\psi(u)$ to reduce cancellation and control error.
- **Truncation:** Replace $[0, \infty)$ with $[0, U_{\max}]$. Increase $U_{\max}$ until the tail is negligible or use a variable transform (e.g., tanh-sinh) to compress infinity.
- **Grid vs. FFT:** For a *surface* of strikes, discretize u on a uniform grid and use FFT. For *individual strikes*, a high-order quadrature (Clenshaw-Curtis, Gauss-Laguerre on a mapped domain, or adaptive GK15) is often simpler and more precise.
- **Puts:** Recover with put-call parity or use a separate damping regime (negative α).

```
# Carr-Madan by direct quadrature (single strike)

input: S0, r, q, T, K, alpha>0, charfun  # charfun(u) = E[e^{iu ln S_T}] under Q
k    = ln(K)
psi(u) = exp(-r*T) * charfun(u - i*(alpha + 1))
                  / (alpha*alpha + alpha - u*u + i*(2*alpha + 1)*u)

# real-imag split integrand for stability
integrand(u) = cos(u*k) * Re(psi(u)) + sin(u*k) * Im(psi(u))

# choose truncation and quadrature
```

```
Umax = choose_cutoff_by_tolerance()
I = adaptive_GaussKronrod( integrand, 0, Umax )  # or Clenshaw-Curtis on [0,Umax]

price = exp(-alpha*k) * I / pi
return price
```

```
# Carr-Madan by FFT (many strikes on a log-strike grid)

input: S0, r, q, T, alpha, N (power of two), du (u spacing)
u_j   = j * du, j=0..N-1
k0    = chosen_min_log_strike
dk    = 2*pi/(N*du)

# compute damped transform samples with Simpson or trapezoid weights
for j in 0..N-1:
  u = u_j
  psi = exp(-r*T) * charfun(u - i*(alpha + 1))
                / (alpha*alpha + alpha - u*u + i*(2*alpha + 1)*u)
  g[j] = psi * exp(-i*u*k0) * w_j     # w_j: integration weights (e.g., Simpson)

# inverse FFT-like recovery
G = FFT(g)
for m in 0..N-1:
  k_m   = k0 + m*dk
  C[m]  = exp(-alpha*k_m) * Re(G[m]) / pi

# map log-strikes k_m to strikes K_m = exp(k_m)
return { (K_m, C[m]) } across m
```

Intuition: *Instead of adding up payoffs in price space, Carr–Madan moves to the "frequency" world, where complex shapes become easier to handle. Damping acts like a gentle fade-out so the integral doesn't "ring" at infinity. For many strikes, the FFT turns thousands of prices into a single lightning-fast transform. The author has also shown in another article 'Fast Fourier Transform as an Engine for the Convolution of PDFs and Black-Scholes Option Pricing Using Python' that one could integrate the payoff and the PDF directly using FFT convolution, which will churn out thousands of option values for thousands of corresponding spot prices. These can be interpolated to a high accuracy for any spot price using a Hermite (PCHIP) spline.*

When to use: If you have a clean characteristic function (BS, Heston, Lévy models), Carr–Madan is ideal for fast, multi-strike pricing (FFT), or precise single-strike pricing (quadrature). Tune α, the truncation $U_{\max}$, and use a stable real–imag split.

Characteristic Functions
Black-Scholes (Log-Normal) model:

$$\varphi_{BS}(u;T) = \exp\Big(i\,u\big[\ln S_0 + (r - q - \tfrac{1}{2}\sigma^2)\,T\big] \\ - \tfrac{1}{2}\,\sigma^2\,u^2 T\Big)\,.$$

Heston (Stochastic Volatility) model:

$$\varphi_H(u;T) = \exp\Big\{i\,u\,\ln S_0 + i\,u\,(r-q)\,T + \frac{\kappa\theta}{\sigma^2}\Big[(\kappa - i\rho\sigma\,u - d)\,T - 2\,\ln\frac{1 - g\,e^{-dT}}{1-g}\Big] \\ + \frac{v_0}{\sigma^2}\,(\kappa - i\rho\sigma\,u - d)\,\frac{1-e^{-dT}}{1-g\,e^{-dT}}\Big\}, \backslash[0.6em]\text{where}\quad d \qquad = \sqrt{(\kappa - i\rho\sigma\,u)}$$

Variance-Gamma (Lévy) model:

$$\varphi_{VG}(u;T) = \exp\Big(i\,u\,\omega\,T \;-\; \frac{T}{\nu}\,\ln\!\big(1 - i\,\theta\,\nu\,u + \tfrac{1}{2}\,\sigma^2\,\nu\,u^2\big)\Big)\,, \backslash[0.6em]\omega \qquad = \frac{1}{\nu}\,\ln\!\big(1 - \theta\,\nu - \tfrac{1}{2}\,\sigma^2\,\nu\big)\,.$$

Stop-Loss Premium Formulation

The call can be seen as a discounted *stop-loss premium* (a tail expectation):

$$C \;=\; e^{-rT}\,\mathbb{E}\big[(S_T - K)^+\big] \;=\; e^{-rT}\int_K^\infty (s-K)\,f_{S_T}(s)\,ds.$$

A powerful reformulation integrates over *quantiles* rather than prices:

$$C \;=\; e^{-rT}\int_{F(K)}^{1} \big(F^{-1}(p) - K\big)\,dp,$$

where F is the CDF of S_T and F^{-1} its quantile (PPF). This is model-agnostic: any distribution with a PPF can be priced this way. It directly targets the region $p \in [F(K), 1]$ where the payoff is nonzero, and it avoids oscillations entirely.

In practice, when specifying F under the risk-neutral measure $\mathbb{Q}$, a *drift correction* is often required so that the discounted asset price is a martingale. This is typically achieved via a **Girsanov change of measure**, adjusting the drift term in the log-price process so that $\mathbb{E}_{\mathbb{Q}}[S_T] = S_0 e^{(r-q)T}$. For Lévy or other non-Gaussian models, this correction ensures the characteristic exponent $\psi(u)$ satisfies $\psi(-i) = r - q$. Note that this applies to all the option pricing models in this document.

If the PPF is not available in closed form but the *moment generating function* (MGF) $M_X(t) = \mathbb{E}[e^{tX}]$ of $X = \ln S_T$ is known, one can recover F or its tail probabilities via numerical Laplace/Fourier inversion of the MGF or characteristic function (FT of the PDF). This allows the same stop-loss/quantile integration framework to be applied to a wide class of distributions, including those defined only through their MGF.

- **Numerical stability:** The integrand is monotone in p; use adaptive Simpson or Gauss-Kronrod on $[F(K), 1]$.
- **Endpoint handling:** Near $p \to 1$, the integrand may be flat. Use an endpoint map (e.g., $p = 1 - e^{-y}$) if needed to expose exponential tail decay.
- **Heavy tails:** If the tail is heavy, use a tail cut $p_{\max} < 1$ with a bound for the remainder, or accelerate with a change of variables.
- **Links:** The integral is the (discounted) Tail Value-at-Risk (TVaR) of S_T shifted by K: $C = e^{-rT} \int_{F(K)}^{1} F^{-1}(p)\, dp - e^{-rT} K (1 - F(K))$.

```
# Stop-Loss premium by quantile integration (model-agnostic)

input: K, r, T, CDF F(s), quantile PPF Q(p) = F^{-1}(p)

pK = F(K)

integrand(p) = Q(p) - K

# adaptive integration on [pK, 1]
# consider change of variables to improve endpoint behavior:
#   p = 1 - exp(-y), y in [ -ln(1 - pK), ∞ )
#   dp = exp(-y) dy
#   ∫_{pK}^1 (Q(p) - K) dp = ∫_{y0}^{∞} ( Q(1 - e^{-y}) - K ) e^{-y} dy

if endpoint_flat:
  y0 = -ln(1 - pK)
  J  = adaptive_GaussKronrod( lambda y: (Q(1 - exp(-y)) - K)*exp(-y), y0, Ymax )
else:
  J  = adaptive_Simpson( integrand, pK, 1 )

price = exp(-r*T) * J
return price
```

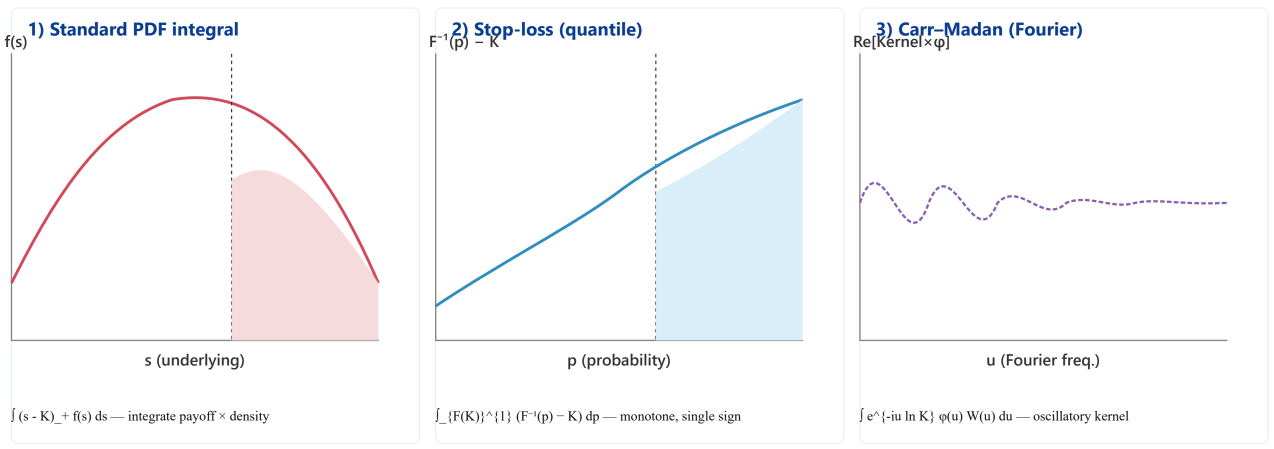
```
# Alternative: direct price-space integration with mapping to (0, ∞)
# substitute s = K + x, x in [0, ∞)
# C = e^{-rT} ∫_0^∞ x * f_S(K + x) dx
# map x = g(t) using tanh-sinh or x = exp(t) - 1 transforms for robust tails

g(t)    = tanh_sinh_map_to_infinity(t)     # e.g., x = tanh_sinh_transform(t)
g'(t)   = derivative_of_g(t)
integrand(t) = g(t) * f_S(K + g(t)) * g'(t)

price = e^{-rT} * trapezoid_over_R( integrand )  # DE tanh-sinh on t ∈ (-∞, ∞)
return price
```

Let's discuss why this is a superb formulation. The stop-loss premium formulation powerfully rewrites the call price as

$$C = e^{-rT} \int_{F(K)}^{1} \left(F^{-1}(p) - K\right) dp.$$

Because the quantile (PPF) $F^{-1}(p)$ is strictly increasing on $[0,1]$, the integrand $F^{-1}(p) - K$; is a monotonic, single-sign function on $[F(K),1]$. In contrast to integrating a bell-shaped density (with its peaks and tails) or an oscillatory Fourier kernel, this smooth, unidirectional curve:

- Eliminates oscillations and sign-changes, so uniform or adaptive quadrature converges with far fewer nodes.
- Delivers stable, predictable error estimates—no hidden extrema to confound local refinement.
- Restricts computation to the true payoff region $[F(K),1]$, avoiding wasted evaluations where the payoff is zero.
- Works with any model that provides an invertible CDF-quantile pair, from lognormal to heavy-tailed or skewed distributions.



Visualization: left = PDF × payoff; middle = stop-loss quantile; right = Carr-Madan (oscillatory)

***Intuition:** The stop-loss view says, "Only the part of the distribution above the strike matters." - just like the strike-aware Gauss-Legendre mentioned above. Sliding to quantiles lets you walk straight along that tail from "just at the strike" to "way above it," adding up how much the option is in-the-money at each probability level.*

When to use: If your model gives a *quantile function* (PPF) or an easy way to sample/invert the CDF, this method is trivial to implement, robust, and fast for single-strike pricing. It generalizes beyond lognormal to any distribution with a PPF and pairs naturally with your custom-quadrature toolkit on $[p_K, 1]$.

Practical guidance and examples

Choosing parameters and transforms

- **Carr–Madan:** Start with $\alpha \in [1, 2]$. For quadrature, increase $U_{\max}$ until the residual tail is below tolerance; for FFT, pick Δu and N to cover the log-strike span with desired resolution $\Delta k = 2\pi/(N\,\Delta u)$. Always use the real-imag split.
- **Stop-Loss:** Use adaptive Gauss-Kronrod on $[p_K, 1]$. If the integrand is too flat near 1, apply $p = 1 - e^{-y}$ and integrate over $y \in [y_0, \infty)$ with tanh-sinh or a halved-step trapezoid (double-exponential decay aids convergence).

Model examples (sketch)

- **Black-Scholes:** $\varphi(u) = \exp\big(ium - \frac{1}{2}\sigma_T^2 u^2\big)$, with $m = \ln S_0 + (r - q - \frac{1}{2}\sigma^2)T$, $\sigma_T = \sigma\sqrt{T}$. For Stop-Loss, $Q(p) = \exp\big(m + \sigma_T \Phi^{-1}(p)\big)$.
- **Heston:** closed-form $\varphi(u)$ available → Carr–Madan integrates cleanly; Stop-Loss is viable if you can numerically invert the CDF or build a monotone spline PPF from samples.
- **Lévy models:** characteristic functions are native; Carr–Madan is usually preferred for speed and stability; Stop-Loss works if a PPF or accurate CDF is available.

```
# Error control heuristics
# Carr-Madan tail test:
# Increase Umax until |∫_{Umax}^{Umax+Δ} integrand(u) du| < tol_tail
for trial in [U1, U2, U3]:
  tail_est = panel_quadrature(integrand, trial, trial + deltaU)
  if abs(tail_est) < tol_tail:
        Umax = trial; break

# Stop-Loss endpoint test:
# Refine adaptive integration until successive estimates differ < tol
I_prev = None
for refine in 1..R:
  I = adaptive_GK(integrand_on_quantiles, pK, 1, tol_refine)
  if I_prev is not None and abs(I - I_prev) < tol:
        break
  I_prev = I; tol_refine *= 0.5
```

Some Notes:

- **Monte Carlo:** A numerical integration technique based on random sampling of the domain. It converges slowly in one dimension ($O(N^{-1/2})$) but its convergence rate is independent of dimension, making it valuable for very high-dimensional problems such as portfolio risk or path-dependent option pricing.
- **Lebesgue integration:** A measure-theoretic framework that integrates by summing over *ranges of function values* rather than slices of the domain. It underpins modern probability theory and risk-neutral pricing. Monte Carlo methods are a natural numerical realisation of Lebesgue integration when expectations are taken under a probability measure. Deterministic quadrature rules like Gaussian quadrature can also be seen as Lebesgue-style in that they weight function values according to a measure (e.g. a weight function $w(x)$).
- **Riemann sums:** The classical definition of an integral as the limit of sums of $f(\xi_i)\,\Delta x_i$ over partitions of the domain. In numerical analysis, this viewpoint leads directly to *rectangular rules* (left, right, midpoint), and by refinement to *trapezoidal* and *Simpson's* rules. These are efficient for smooth, low-dimensional integrands but can be inefficient for oscillatory payoffs or high-dimensional finance problems, if implemented naively.

Black–Scholes KO (single observation) via custom quantile quadrature

This is an example just to show that using quadrature means in code, you are really just looking at the integral as it is penned on paper, any modifications to the payoff are direct modifications in the code, making everything very clear, concise, and easy to manage:

```
# Price: European up-and-out call with single barrier observation at T
# Payoff: (S_T - K)_+ * 1{ S_T < B }          # knock-out if S_T ≥ B
# Under risk-neutral BS: ln S_T ~ N(m, v),    m = ln S0 + (r - q - ½σ²)T,    v = σ²
T

INPUT:
  S0, K, B, T, r, q, σ
  N         # e.g., 200 quantile nodes
  tail_cut # e.g., 1e-4

SETUP (lognormal params):
  m = ln(S0) + (r - q - 0.5*σ^2)*T
  s = σ * sqrt(T)

NODES (probability space):
  p_i = linspace(tail_cut, 1 - tail_cut, N)
  z_i = Φ^{-1}(p_i)
  S_i = exp(m + s * z_i)

WEIGHTS (Δp):
  w_i from centered differences on p_i (normalize to 1)

PAYOFF FILTER:
  for each i: g_i = max(S_i - K, 0) * 1{ S_i < B }

PRICE:
  V ≈ exp(-r*T) * Σ_i w_i * g_i

NOTES:
- This is extremely stable: monotone payoff, no oscillations.
- For multi-observation barriers, integrate conditional survival per step or
switch to path methods.
```

Beyond Black-Scholes & Finance

Quadrature methods extend naturally to more complex models, where the option price is still an expectation under the risk-neutral measure but the underlying dynamics change the form of the characteristic function or density:

- **Stochastic volatility models (Heston, SABR):**
  Heston's log-price $X_t = \ln S_t$ has a closed-form characteristic function:

$$\begin{aligned}\varphi_H(u;T) = \exp\Bigg\{ & i\,u\,\ln S_0 + i\,u\,(r-q)\,T \\ & + \frac{\kappa\theta}{\sigma^2}\left[(\kappa - i\rho\sigma\,u - d)\,T - 2\,\ln\!\left(\frac{1-g\,e^{-dT}}{1-g}\right)\right] \\ & + \frac{v_0}{\sigma^2}\,\frac{\kappa - i\rho\sigma\,u - d}{1-g\,e^{-dT}}\,\left(1-e^{-dT}\right)\Bigg\},\end{aligned}$$

with

$$d = \sqrt{(\kappa - i\rho\sigma\, u)^2 + \sigma^2\,(u^2 + i\,u)}, \quad g = \frac{\kappa - i\rho\sigma\, u - d}{\kappa - i\rho\sigma\, u + d}.$$

Quadrature methods (Carr–Madan, COS, etc.) integrate this φ_H against the chosen payoff kernel.

- **Jump-diffusion processes (Merton, Kou):**
  Merton's model augments Black-Scholes with normally distributed jumps:

  $$\varphi_M(u;T) = \exp\Big\{iumT - \tfrac{1}{2}\sigma^2 u^2 T + \lambda T\left(e^{iu\mu_J - \frac{1}{2}\sigma_J^2 u^2} - 1\right)\Big\},$$

  where λ is jump intensity, μ_J and σ_J are jump mean and volatility. Kou's double-exponential jumps replace the Gaussian jump term with $\lambda T\left(\frac{p\eta_1}{\eta_1 - iu} + \frac{(1-p)\eta_2}{\eta_2 + iu} - 1\right)$.
- **Variance Gamma and Lévy processes:**
  Variance Gamma has characteristic function

  $$\varphi_{VG}(u;T) = \left(1 - i\theta\nu u + \tfrac{1}{2}\sigma^2\nu u^2\right)^{-T/\nu} \exp(iu\omega T),$$

  with drift correction $\omega = \frac{1}{\nu}\ln\left(1 - \theta\nu - \frac{1}{2}\sigma^2\nu\right)$. More generally, for a Lévy process with exponent $\psi(u)$, $\varphi(u;T) = \exp\big(T\,\psi(u)\big)$ and quadrature integrates this against the payoff transform.
- **Multi-asset options (tensor product quadrature, sparse grids):**
  For d underlyings with joint density $f(\mathbf{s})$, the price is

  $$V = e^{-rT}\int_{\mathbb{R}^d} \mathrm{Payoff}(\mathbf{s})\, f(\mathbf{s})\, d\mathbf{s}.$$

  Tensor-product Gaussian quadrature uses $\sum_{i_1=1}^{n_1}\cdots\sum_{i_d=1}^{n_d} w_{i_1}\cdots w_{i_d}\,\mathrm{Payoff}(s_{i_1},\ldots,s_{i_d})$, while sparse grids reduce the number of nodes from $O(n^d)$ to $O(n(\log n)^{d-1})$ for smooth payoffs.

Quadrature is like a Swiss Army knife *for quantitative finance—once you understand how to use it, you can price all sorts of financial instruments, not just simple options. It's particularly useful for options that depend on multiple assets or have unusual payoff structures.*

Multidimensional Integrals

For multidimensional integrals, tensor products of 1D rules can be used, but the number of points grows exponentially with dimension. Smolyak sparse grids dramatically reduce nodes while maintaining accuracy for smooth integrands.

Full tensor product quadrature. Given 1D rules on $[-1,1]$,

$$U_{n_j}^{(j)}[g] \;=\; \sum_{i=1}^{n_j} w_i^{(j)}\, g\big(x_i^{(j)}\big), \qquad j = 1,\ldots,d,$$

the d-dimensional integral on $\Omega = [-1,1]^d$ is approximated by the tensor product

$$I_d = \int_\Omega f(\mathbf{x})\, d\mathbf{x} \approx \sum_{i_1=1}^{n_1} \cdots \sum_{i_d=1}^{n_d} \Big(\prod_{j=1}^{d} w_{i_j}^{(j)} \Big)\, f(x_{i_1}^{(1)}, \dots, x_{i_d}^{(d)}),$$

with node count $N_{\text{tensor}} = \prod_{j=1}^d n_j = O(n^d)$ for $n_j \asymp n$ — the “curse of dimensionality.”

Smolyak sparse grids (anisotropic form). Let $\{U_\ell^{(j)}\}_{\ell \ge 1}$ be a nested sequence of 1D rules (e.g., Clenshaw–Curtis), and define hierarchical surpluses

$$\Delta_\ell^{(j)} = U_\ell^{(j)} - U_{\ell-1}^{(j)}, \qquad U_0^{(j)} \equiv 0.$$

The Smolyak operator of level q in d dimensions is

$$\mathcal{A}_q^{(d)}[f] = \sum_{\substack{\boldsymbol{\ell} \in \mathbb{N}^d \\ \|\boldsymbol{\ell}\|_1 \le q+d-1}} \Big(\Delta_{\ell_1}^{(1)} \otimes \cdots \otimes \Delta_{\ell_d}^{(d)} \Big)[f],$$

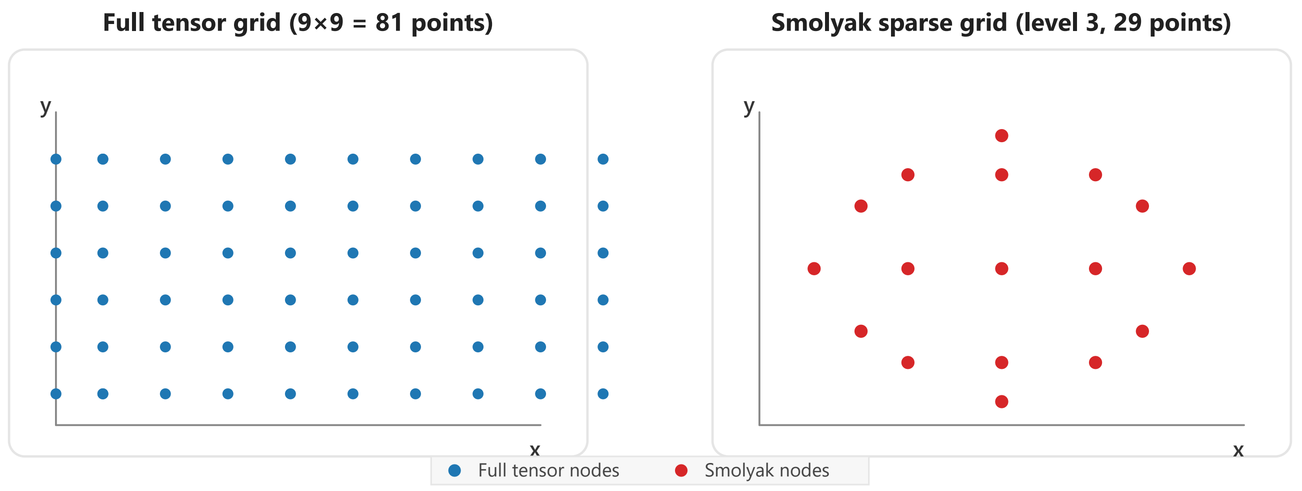
or, in an *anisotropic* setting with importance weights $\boldsymbol{\gamma} = (\gamma_1, \dots, \gamma_d)$,

$$\mathcal{A}_{q,\gamma}^{(d)}[f] = \sum_{\substack{\boldsymbol{\ell} \in \mathbb{N}^d \\ \sum_{j=1}^d \gamma_j(\ell_j - 1) \le q}} \Big(\Delta_{\ell_1}^{(1)} \otimes \cdots \otimes \Delta_{\ell_d}^{(d)} \Big)[f].$$

For nested rules with $n_\ell \approx 2^{\ell-1} + 1$, the total nodes satisfy $N_{\text{sparse}} = O(2^q\, q^{d-1})$, dramatically smaller than $O(2^{qd})$. For functions with bounded mixed derivatives (mixed Sobolev smoothness), the error decays near the 1D rate up to polylog factors in d.

***Why sparse grids matter:** A full grid in 10D is like trying to photograph every grain of sand on a beach. Smolyak takes smart snapshots along carefully chosen “slices” so you still see the big picture without counting every grain.*

Visual intuition: dense vs. sparse grids (2D)



Dense tensor grids explode as n^d. Smolyak sparse grids keep the 1D accuracy trend with far fewer points for smooth, mixed-regularity integrands.

Example integrands

- **Smooth mixed-derivative case:**

$$f(\mathbf{x}) = \exp\Big(-\sum_{j=1}^{d} a_j(x_j - c_j)^2\Big)\,\cos\Big(2\pi\sum_{j=1}^{d} b_j x_j\Big),\quad \mathbf{x}\in[-1,1]^d,$$

  with $a_j > 0$. Ideal for sparse grids (high mixed smoothness).

- **Basket call (via Gaussian map):**

$$V \;=\; e^{-rT}\,\mathbb{E}\Bigg[\Big(\sum_{j=1}^{d} w_j S_{0j} e^{\mu_j+\sigma_j Z_j} - K\Big)^{+}\Bigg],$$

  where $\mathbf{Z}\sim\mathcal{N}(0,\Sigma)$. Use Gauss-Hermite in dD or Cholesky to map $\mathbf{Z} = L\mathbf{y}$, $\mathbf{y}\sim\mathcal{N}(0,I)$ and then apply sparse grids on $\mathbf{y}$.

Pseudocode: full tensor product quadrature

```
# Full tensor quadrature (d dimensions)

input:
  - f: function R^d → R
  - rules[1..d]: 1D quadrature rules, each with nodes x_j[1..n_j], weights
w_j[1..n_j]

function tensor_integrate(f, rules):
        idx = [1]*d
        total = 0.0
        while True:
                # build point and weight
                x = [ rules[j].x[ idx[j] ] for j in 1..d ]
                w = 1.0
                for j in 1..d:
```

```
                w *= rules[j].w[ idx[j] ]
            total += w * f(x)

            # advance mixed-radix counter
            k = d
            while k ≥ 1:
                if idx[k] < rules[k].n:
                    idx[k] += 1
                    for m in k+1..d:
                        idx[m] = 1
                    break
                else:
                    k -= 1
            if k == 0:
                return total
```

Trade-off: Full tensor is simple and optimal for small d (e.g., $d \leq 3$) with cheap integrands. Cost grows as $O(n^d)$.

Pseudocode: Smolyak sparse grid (nested 1D rules)

```
# Smolyak sparse grid of level q with nested 1D rules

input:
  - f: function R^d → R
  - U_level(j, ell): returns 1D level-ell rule for dimension j:
          nodes x^{(j)}_{ell}[1..n_ell], weights w^{(j)}_{ell}[1..n_ell]
        with nesting: nodes at ell-1 are contained in ell
  - q: sparse grid level
  - d: dimension

function smolyak_integrate(f, U_level, q, d):
        total = 0.0

        # iterate over all multi-indices ell = (ell_1, … ,ell_d)
        for ell in all_multi_indices(d):
                if sum(ell) ≤ q + d - 1:
                        # build tensor product of surpluses Δ_{ell_j}
                        nodes_list = []
                        weights_list = []
                        for j in 1..d:
                                # current and previous level rules
                                nodes_curr, weights_curr = U_level(j, ell[j])
                                if ell[j] > 1:
                                        nodes_prev, weights_prev = U_level(j,
ell[j]-1)
                                else:
```

```
                                                   nodes_prev, weights_prev = [], []

                                        # surplus = current rule minus previous rule
(matching nodes)
                                        Δ_nodes, Δ_weights = hierarchical_surplus(nodes_curr,
weights_curr,

nodes_prev, weights_prev)
                                        nodes_list.append(Δ_nodes)
                                        weights_list.append(Δ_weights)

                                # tensor product loop over surplus nodes
                                for (pt, wt) in tensor_product(nodes_list, weights_list):
                                        total += wt * f(pt)

        return total
```

***Multidimensional quadrature** is like trying to measure the volume of a complex shape. A full grid approach would measure every single point, which is inefficient. Sparse grids are like taking strategic measurements at key locations to estimate the volume without all the work.*

In any multidimensional pricing or risk problem, the first step is to clearly identify the *integrand* — the function you are integrating over the joint distribution of your risk factors. Quants should always build a straightforward Monte Carlo version first: it serves as a "truth" benchmark or at least a ballpark sanity check before attempting more exotic quadrature or sparse grid methods. Once the integrand is understood, make every effort to reduce the effective dimension algebraically. Standard calculus tools apply:

- **Fubini's theorem:** If $f(x,y)$ is integrable on $A \times B$, then

$$\iint_{A\times B} f(x,y)\,dx\,dy \;=\; \int_A \left[\int_B f(x,y)\,dy\right] dx \;=\; \int_B \left[\int_A f(x,y)\,dx\right] dy.$$

  This enables *iterative integration* — integrate out one variable at a time when the inner integral is tractable. *Finance example 1 (Max of two):*

$$C = e^{-rT} \iint_{\mathbb{R}^2} \left(\max(s_1,s_2) - K\right)^+ f_{S_1}(s_1) f_{S_2}(s_2)\, ds_1 ds_2$$

$$C = e^{-rT} \int_0^{\infty} \left(\int_0^{s_1} (s_1 - K)^+ f_{S_2}(s_2)\, ds_2 + \int_{s_1}^{\infty} (s_2 - K)^+ f_{S_2}(s_2)\, ds_2 \right) f_{S_1}(s_1)\, ds_1$$

Finance example 2 (Independent factors): For separable $f(x,y) = g(x)h(y)$,

$$\iint g(x)h(y)\, dx\, dy \;=\; \left(\int g(x)\, dx\right)\left(\int h(y)\, dy\right).$$

> *Peel one layer at a time: fix one variable, finish the easy inside piece, then move on.*

```
# Fubini iterative integration (max-call under independence)
def price_max_call(f1, f2, K, r, T):
        def inner(s1):
                t1 = integrate(lambda s2: max(s1 - K, 0.0) * f2(s2), 0.0, s1)
                t2 = integrate(lambda s2: max(s2 - K, 0.0) * f2(s2), s1, inf)
                return (t1 + t2) * f1(s1)
        return math.exp(-r*T) * integrate(inner, 0.0, inf)
```

- **Change of variables:** Use $\mathbf{u} = T(\mathbf{x})$ to simplify domain/structure; adjust by the Jacobian.

$$\int_{\Omega} f(\mathbf{x})\, d\mathbf{x} \;=\; \int_{T(\Omega)} f(T^{-1}(\mathbf{u}))\;\; \det J_{T^{-1}}(\mathbf{u})\big|\, d\mathbf{u}.$$

Example 1 (Polar): For $\iint_{x^2+y^2 \le R^2} g(\sqrt{x^2+y^2})\, dx\, dy$,

$$\int_0^{2\pi} \int_0^{R} g(r)\, r\, dr\, d\theta \;=\; 2\pi \int_0^{R} g(r)\, r\, dr.$$

Example 2 (Finance, decorrelation): For $Z \sim \mathcal{N}(0, \Sigma)$, take Cholesky L s.t. $\Sigma = LL^{\top}$, set $Z = LY$ with $Y \sim \mathcal{N}(0, I)$.

> *Rotate your axes so the problem lines up with them; straight cuts beat diagonal cuts.*

```
# Decorrelate correlated normals via Cholesky
def decorrelated_samples(L, n_samples):
        for _ in range(n_samples):
                y = np.random.normal(size=L.shape[0])
                yield L @ y
```

- **Change of bounds:** Triangles and simplices often become rectangles under a clever map.

$$\iint_{\substack{x\geq 0,\, y\geq 0\\ x+y\leq 1}} h(x,y)\,dx\,dy \xrightarrow{u=x+y,\, v=x} \int_0^1\left(\int_0^u h(u-v,v)\,dv\right)du.$$

If h depends only on u, then $\int_0^u h(u)\,dv = u\,h(u)$ and the integral becomes 1D:

$$\int_0^1 u\,h(u)\,du.$$

> *Redraw a slanted fence into a neat rectangle; measuring becomes trivial.*

```
# Change of bounds mapping for triangle x≥0,y≥0,x+y≤1
def triangle_to_uv_integral(h_u):
        # if h(x,y) = H(x+y) only
        return integrate(lambda u: u * h_u(u), 0.0, 1.0)
```

- **Pre-solving with integration by parts (IBP):** Reduce difficult kernels before numerics.

$$I = \int_0^\infty e^{-ax}\sin(bx)\,dx \;=\; \frac{b}{a^2+b^2}.$$

(One IBP plus a standard Laplace integral.) *Finance:* In Laplace/Fourier pricing, IBP moves derivatives to exponential kernels, simplifying the residual integral.

> *Do the easy algebraic pruning first; then compute only what's left.*

```
# SymPy: verify a closed-form piece before numerics
import sympy as sp
x,a,b = sp.symbols('x a b', positive=True)
expr = sp.exp(-a*x)*sp.sin(b*x)
I = sp.integrate(expr, (x, 0, sp.oo))
print(sp.simplify(I))  # b/(a**2 + b**2)
```

- **Reduction to a low-dimensional statistic:** If the payoff depends only on $S = \sum_{j=1}^{d} w_j X_j$, transform to $(S, \text{orthogonal})$ and integrate out orthogonal parts. *Deelstra's comonotonic upper bound (Arithmetic basket):*

$$C = e^{-rT}\,\mathbb{E}\left[\left(\tfrac{1}{d}\sum_{j=1}^{d} S_{T,j} - K\right)^{+}\right]$$
$$\rightsquigarrow\ C_{\text{upper}} = e^{-rT}\int_0^1 \left(\tfrac{1}{d}\sum_{j=1}^{d} F_j^{-1}(p) - K\right)^{+} dp$$

> *Assume perfect lockstep across names — you get a safe overestimate that collapses to 1D.*

```
# Deelstra's comonotonic bound via quantile integration on [0,1]
def deelstra_upper_bound(F_inv_list, K, r, T, rule):
        def integrand(p):
                avg = sum(F_inv(p) for F_inv in F_inv_list) / len(F_inv_list)
                return max(avg - K, 0.0)
        total = 0.0
        for w, node in zip(rule.weights, rule.nodes):  # nodes in [0,1]
                total += w * integrand(node)
        return math.exp(-r*T) * total
```

- **Conditioning (Law of iterated expectations):** Collapse one variable via a conditional expectation.

$$\mathbb{E}[g(X,Y)] \;=\; \mathbb{E}\big[\,\mathbb{E}[g(X,Y)\mid X]\,\big].$$

Finance: If $Y \mid X = x$ is Gaussian and g is affine in Y, the inner expectation is analytic → 1D left.

> *Ask one question at a time; if the first answer reveals the second, you skip it.*

```
# Conditioning reduction: E[g(X,Y)] with Y|X Gaussian
def reduced_integrand(x):
        mu, sig = mu_sigma_given_x(x)
        return analytic_E_over_Y(mu, sig)   # closed form in Y
price = integrate(lambda x: reduced_integrand(x) * fX(x), x_lo, x_hi)
```

- **Marginalisation / separability:** If factors separate, multiply one-dimensional pieces.

$$\iint h(x)\,k(y)\,dx\,dy \;=\; \Big(\int h(x)\,dx\Big)\Big(\int k(y)\,dy\Big).$$

Finance: Independent term structures, product payoffs.

> *If chores are independent, split the list and finish each separately.*

- **Convolution structure (sum of factors):** If $Z = X + Y$ with independent X, Y,

$$f_Z(z) \;=\; \int f_X(x)\,f_Y(z-x)\,dx, \qquad \hat{f}_Z(u) \;=\; \hat{f}_X(u)\,\hat{f}_Y(u).$$

If the payoff depends only on Z, use 1D convolution or FFT (via characteristic functions). *Finance:* Sum of (approximate) lognormals; price via FFT over φ_Z.

> *Add the "shadows" (Fourier) where addition becomes multiplication; then return.*

```
# 1D convolution via FFT using characteristic functions
def price_via_fft(payoff_hat, phi_Z, u_grid):
        # payoff_hat(u): Fourier transform of payoff kernel
        # phi_Z(u): characteristic function of Z
        G = payoff_hat(u_grid) * phi_Z(u_grid)
        # inverse FFT to real space (schematic)
        return ifft(G).real
```

- **Orthogonal transforms (PCA/rotations):** Rotate to align one axis with the dominant direction; integrate orthogonal parts out.

$$\mathbf{x} = Q\mathbf{y}, \quad Q^\top Q = I, \qquad \int f(\mathbf{x})\, d\mathbf{x} \;=\; \int f(Q\mathbf{y})\, d\mathbf{y}.$$

  Finance: PCA for correlated Gaussian factors; keep first components, marginalise the rest.

  > *Turn the problem so the "action" sits on a single axis; the rest fades out.*

- **Symmetry and group actions:** If f is invariant under permutations/rotations, integrate over a fundamental region and multiply.

$$\int_\Omega f(\mathbf{x})\, d\mathbf{x} \;=\; |\mathcal{G}| \int_{\Omega/\mathcal{G}} f(\mathbf{x})\, d\mathbf{x}.$$

  Finance: Exchangeable assets in a symmetric basket.

  > *Compute one wedge of the pie, then multiply by the number of wedges.*

- **Indicator-function tricks (region reparametrisation):** Replace $\mathbf{1}_{\{X>Y\}}$ with a variable change $U = X - Y, V = Y$ so the indicator becomes $\mathbf{1}_{\{U>0\}}$ with simple bounds.

  > *Swap to difference+baseline; the inequality becomes a plain "greater than zero."*

- **Known moments (polynomial integrands):** Use moment formulas and avoid integration.

$$X \sim \mathcal{N}(0, \sigma^2): \quad \mathbb{E}[X^{2n}] = (2n-1)!!\, \sigma^{2n}, \qquad \mathbb{E}[X^{2n+1}] = 0.$$

  Finance: Polynomial approximations of payoffs; replace integrals by moments.

  > *If you know all the averages in advance, you can skip counting every case.*

- **Laplace / Fourier transforms (transform pricing):** Move to transform space, integrate there, invert.

$$\int f(x)\, g(x)\, dx \;=\; \frac{1}{2\pi} \int \hat{f}(u)\, \hat{g}(-u)\, du, \qquad \mathcal{L}\{f\}(s) = \int_0^\infty e^{-sx} f(x)\, dx.$$

  Finance: Carr–Madan/COS integrate against characteristic functions; often 1D.

> *Translate a hard sentence into a language where it's easy — then translate back.*

- **"p-trivial" variables (drop almost-sure constants):** If a variable is a.s. constant under the measure (P-trivial), it contributes nothing; remove that dimension.

> *If something never changes, don't waste time measuring it.*

- **CAS assist (SymPy/Mathematica) before quadrature:** Offload closed-form sub-integrals, simplifications, or transforms to a CAS, then apply numerics to the residual part.

```
# SymPy pipeline: pre-solve inner integral, then numeric outer
import sympy as sp
x,y,a = sp.symbols('x y a', positive=True)
inner = sp.integrate(sp.exp(-a*y) * sp.cos(x*y), (y, 0, sp.oo))  # closed form in y
# inner = a / (a**2 + x**2)
outer = sp.integrate(inner * sp.exp(-x**2), (x, -sp.oo, sp.oo))  # or hand to quadgk if hard
print(sp.simplify(inner))
```

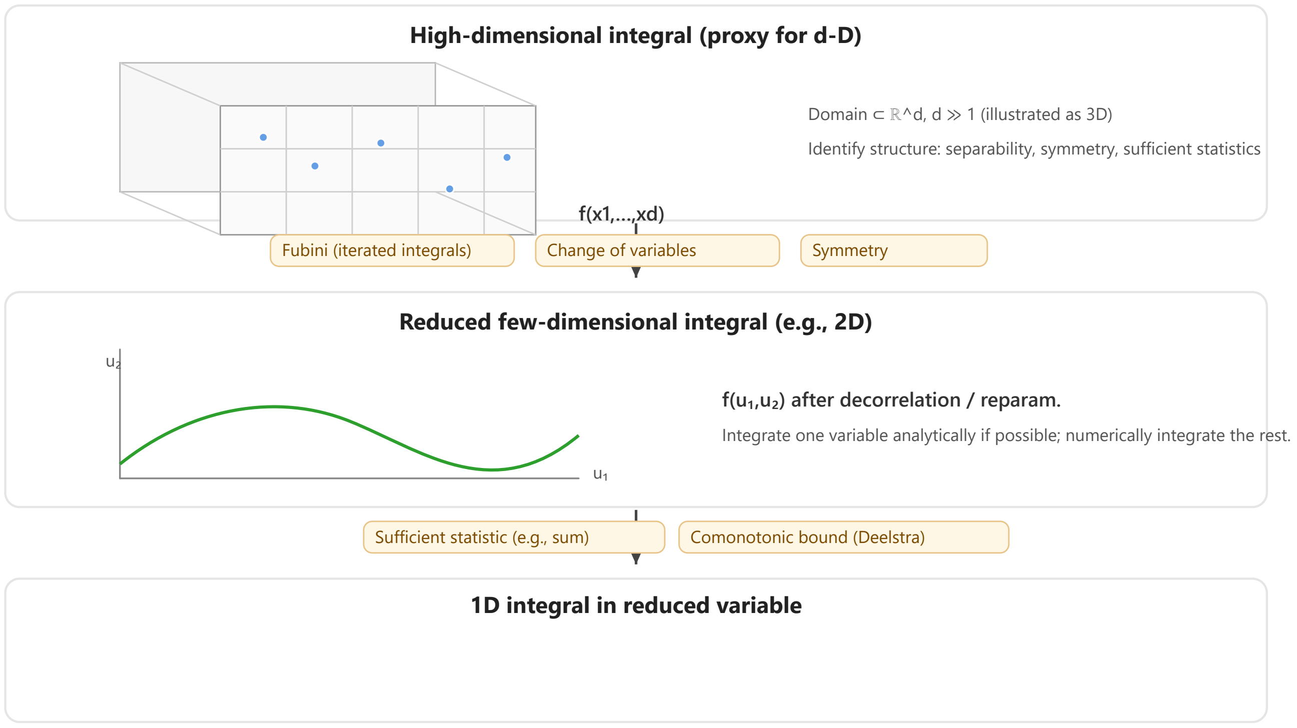
Pseudocode: Deelstra's comonotonic bound

```
def deelstra_upper_bound(F_inv_list, K, r, T, quad_rule):
  # F_inv_list: list of quantile functions F_j^{-1}(p)
  def integrand(p):
          avg = sum(F_inv(p) for F_inv in F_inv_list) / len(F_inv_list)
          return max(avg - K, 0.0)
  total = 0.0
  for w, node in zip(quad_rule.weights, quad_rule.nodes):
          total += w * integrand(node)
  return math.exp(-r*T) * total
```

Jargon note:

- **p-trivial:** In probability theory, an event is "p-trivial" if it has probability 0 or 1 under the measure P. In integration contexts, a "p-trivial" variable is one that is almost surely constant — integrating over it adds nothing, so you can drop that dimension entirely.

- **Comonotonic:** Random variables that move together perfectly — higher values of one always correspond to higher values of the others.


High-dimensional integral (proxy for d-D)
Domain ⊂ ℝ^d, d ≫ 1 (illustrated as 3D)
Identify structure: separability, symmetry, sufficient statistics
f(x1,...,xd)
Fubini (iterated integrals)
Change of variables
Symmetry
Reduced few-dimensional integral (e.g., 2D)
u₂
u₁
f(u₁,u₂) after decorrelation / reparam.
Integrate one variable analytically if possible; numerically integrate the rest.
Sufficient statistic (e.g., sum)
Comonotonic bound (Deelstra)
1D integral in reduced variable

Integrals over a Circle and Arbitrary Shapes

Circle / Disk (polar coordinates)

Idea (pizza-slice method): *A disk is easiest to sweep in radius and angle. Imagine slicing a pizza: for each angle θ, move outward from the center (radius r) and add up the contributions of the function. The little area of a thin "ring slice" is not just dr dθ—it's bigger the farther you are from the center. That stretch factor is the radius* ***r*** *(because a small angular step makes a bigger arc at larger radius). That's why the Jacobian is* ***r****.*

- ***Angle*** *tells you where you are around the circle.*
- ***Radius*** *tells you how far from the center you are.*
- ***Jacobian = r*** *scales the tiny area correctly.*

Fun fact: For smooth periodic functions, the simple uniform trapezoid rule in angle often converges spectacularly fast because it "wraps around" perfectly.

Let the disk be $D_R = \{(x,y) \in \mathbb{R}^2 : x^2 + y^2 \le R^2\}$. Use the polar map

$$x = r\cos\theta, \quad y = r\sin\theta, \quad r \in [0,R],\ \theta \in [0,2\pi).$$

The Jacobian matrix is

$$J(r,\theta) = \begin{bmatrix} \partial x/\partial r & \partial x/\partial\theta \\ \partial y/\partial r & \partial y/\partial\theta \end{bmatrix} = \begin{bmatrix} \cos\theta & -r\sin\theta \\ \sin\theta & r\cos\theta \end{bmatrix},$$

and $|\det J| = r$. Thus, for integrable f,

$$\iint_{D_R} f(x,y)\,dx\,dy = \int_0^{2\pi}\int_0^R f(r\cos\theta, r\sin\theta)\,r\,dr\,d\theta.$$

Change-of-variables & quadrature:

Map Gauss–Legendre nodes $\{\xi_k, w_k\}_{k=1}^{n_r}$ on $[-1,1]$ to $[0,R]$:

$$r_k = \tfrac{R}{2}(1+\xi_k), \qquad \tilde{w}_k = \tfrac{R}{2}\,w_k.$$

Use uniform angles $\theta_j = \frac{2\pi j}{n_\theta}$, $j=0,\dots,n_\theta-1$, with weights $w_{\theta j} = \frac{2\pi}{n_\theta}$ (trapezoid on a periodic function). Then

$$I \approx \sum_{j=0}^{n_\theta-1}\sum_{k=1}^{n_r} w_{\theta j}\,\tilde{w}_k\, f(r_k\cos\theta_j, r_k\sin\theta_j)\, r_k.$$

Notes: You can alternatively use radial rules designed for a weight r (e.g., Gauss–Jacobi with $\alpha=1,\beta=0$ after an affine map) so the factor r is “baked into” the weights.

```
# Goal: I = ∬_{x^2+y^2 ≤ R^2} f(x,y) dx dy
# Use polar change of variables with Jacobian r.

input: function f(x,y), radius R, integers n_r, n_theta

# 1) Angular nodes & weights (periodic trapezoid)
for j in 0..n_theta-1:
  theta[j] = 2*pi*j / n_theta
  w_theta[j] = 2*pi / n_theta

# 2) Radial nodes & weights: Gauss–Legendre on [0,R]
#    Map from [-1,1] to [0,R]; include Jacobian (R/2)
xi[], w[] = gauss_legendre(n_r)         # on [-1,1]
for k in 1..n_r:
  r[k] = 0.5*R*(1 + xi[k])
  w_r[k] = 0.5*R*w[k]                   # from mapping

# 3) Double sum (note the extra factor r[k] from Jacobian of polar)
I = 0
for j in 0..n_theta-1:
  for k in 1..n_r:
    x = r[k] * cos(theta[j])
    y = r[k] * sin(theta[j])
    I += w_theta[j] * w_r[k] * f(x, y) * r[k]

return I

# Tips:
# - Increase n_theta if f varies rapidly with angle.
# - Increase n_r if f varies rapidly with radius or near r=R.
# - For smooth periodic dependence on theta, trapezoid converges very fast.
```

Arbitrary shape $\Omega \subset \mathbb{R}^2$ (triangulate & sum)

Idea (tile with triangles): *Any wiggly shape can be covered by tiny, non-overlapping triangles—like a mosaic. Triangles are great because they're flat and simple. We compute the integral on each triangle (by mapping it to a standard "reference" triangle) and then add everything up.*

- ***Mesh*** *your shape into triangles.*
- ***Flatten*** *each triangle onto a simple reference triangle.*
- ***Sample & sum*** *with a small set of well-chosen points (Gaussian quadrature).*

Fun fact: The determinant of the affine map's Jacobian on a triangle is constant and equals ***2 × area of the triangle****.*

Let $\Omega \subset \mathbb{R}^2$ be a polygonal (or meshed) domain, partitioned into disjoint triangles:

$$\Omega = \bigcup_{e=1}^{E} T_e, \qquad T_e \cap T_{e'} = \varnothing \ (e \neq e').$$

For each triangle T_e with vertices v_1, v_2, v_3, define the affine map from the reference triangle

$$T_{\text{ref}} = \{(\xi, \eta) : \xi \geq 0,\ \eta \geq 0,\ \xi + \eta \leq 1\}$$

by

$$\phi_e(\xi, \eta) = v_1 + \xi(v_2 - v_1) + \eta(v_3 - v_1).$$

Its Jacobian matrix $J_e = [v_2 - v_1 \ \ v_3 - v_1]$ is constant on T_e, and $|\det J_e| = 2\,|T_e|$ where $|T_e|$ is the area of triangle T_e. Then

$$\int_{\Omega} f(x,y)\,dx\,dy = \sum_{e=1}^{E} \int_{T_{\text{ref}}} f\big(\phi_e(\xi,\eta)\big)\,|\det J_e|\,d\xi\,d\eta.$$

Reference-triangle quadrature rules: Nodes (ξ_q, η_q) and weights w_q are defined on T_{ref} whose area is $1/2$. A few standard choices:

- *Degree-1 (centroid):* One point at $(\frac{1}{3}, \frac{1}{3})$ with weight $w = \frac{1}{2}$.
- *Degree-2 (three-point):* Points $(\frac{1}{6}, \frac{1}{6}), (\frac{2}{3}, \frac{1}{6}), (\frac{1}{6}, \frac{2}{3})$, each with weight $w = \frac{1}{6}$ (weights sum to $1/2$).

Per-triangle approximation:

$$I_e \approx |\det J_e| \sum_{q=1}^{N_q} w_q\, f\big(\phi_e(\xi_q, \eta_q)\big), \qquad I \approx \sum_{e=1}^{E} I_e.$$

```
# Goal: I = ∬_Ω f(x,y) dx dy by triangulation and Gaussian quadrature.

input: function f(x,y), triangle list {T_e}, quadrature rule {(xi_q, eta_q, w_q)}
```

```
on T_ref

I = 0
for each triangle T_e with vertices v1=(x1,y1), v2=(x2,y2), v3=(x3,y3):
  # 1) Build affine map phi_e and Jacobian J_e
  e1 = v2 - v1
  e2 = v3 - v1
  J = [e1 | e2]                     # 2x2 matrix
  detJ = abs(det(J))                # constant on T_e

  # 2) Quadrature in reference coordinates
  I_e = 0
  for each (xi_q, eta_q, w_q):
    x_q = v1.x + xi_q*e1.x + eta_q*e2.x
    y_q = v1.y + xi_q*e1.y + eta_q*e2.y
    I_e += w_q * f(x_q, y_q)

  # 3) Scale by |detJ|
  I += detJ * I_e

return I

# Tips:
# - Use a finer mesh or higher-degree quadrature if f changes rapidly.
# - For curved boundaries, either refine triangles near the boundary or use curved
elements.
# - The determinant satisfies detJ = 2 * area(T_e).
```

Boundary integral (Green/Gauss)

***Idea (walk the fence, skip the lawn):** Instead of summing over the whole area, sometimes you can walk only along the boundary and get the same answer. This is perfect when what you care about is a "flux" through the boundary or the domain is easier to describe by its edges/curve.*

- *Turn a 2D area integral into a 1D boundary integral (when applicable).*
- *Great for polygons, splines, or piecewise smooth curves.*
- *Integrate along edges with standard 1D quadrature.*

Fun fact: The area of any simple polygon can be computed just by walking its boundary (the "shoelace formula").

Green's theorem (two equivalent forms, CCW orientation):

Circulation form:

$$\iint_\Omega \left(\frac{\partial Q}{\partial x} - \frac{\partial P}{\partial y}\right) dA = \oint_{\partial\Omega} P\,dx + Q\,dy.$$

Divergence / Gauss form in 2D:

$$\iint_\Omega \nabla\cdot\mathbf{F}\,dA = \oint_{\partial\Omega} \mathbf{F}\cdot\mathbf{n}\,ds.$$

To convert a given area integrand f:

- Choose P, Q so that $\partial Q/\partial x - \partial P/\partial y = f$, then integrate $P\,dx + Q\,dy$ along $\partial\Omega$; *or*
- Choose $\mathbf{F}$ so that $\nabla\cdot\mathbf{F} = f$, then integrate $\mathbf{F}\cdot\mathbf{n}$ along $\partial\Omega$.

Examples:

$$\text{Area}(\Omega) = \iint_\Omega 1\,dA = \oint_{\partial\Omega} \tfrac{1}{2}(x\,dy - y\,dx) = \iint_\Omega \nabla\cdot(x,0)\,dA = \oint_{\partial\Omega} (x,0)\cdot\mathbf{n}\,ds.$$

$$\iint_\Omega x\,dA = \oint_{\partial\Omega} \tfrac{1}{2}x^2\,dy \quad (\text{take } P = 0,\ Q = \tfrac{1}{2}x^2).$$

Parametric boundary & 1D quadrature: Let a boundary segment be parameterized by $\gamma(t) = (x(t), y(t))$, $t \in [a,b]$. Then

$$\int_\gamma P\,dx + Q\,dy = \int_a^b \big(P(\gamma(t))\,x'(t) + Q(\gamma(t))\,y'(t)\big)\,dt.$$

For a straight segment from $a = (x_0, y_0)$ to $b = (x_1, y_1)$, use $\gamma(t) = a + t(b-a)$, $t \in [0,1]$, so $x'(t) = x_1 - x_0$, $y'(t) = y_1 - y_0$. Apply Gauss-Legendre in 1D:

$$\int_0^1 g(t)\,dt \approx \sum_{j=1}^{n} w_j\,g(t_j),$$

hence on the segment

$$\int_{\text{seg}} P\,dx + Q\,dy \approx \sum_{j=1}^{n} w_j \Big[P(\gamma(t_j))\,(x_1 - x_0) + Q(\gamma(t_j))\,(y_1 - y_0)\Big].$$

Summing over all segments gives $\oint_{\partial\Omega}$.

Polygon area (shoelace) as a special case: For vertices (x_i, y_i), $i = 1, \ldots, N$, with $(x_{N+1}, y_{N+1}) = (x_1, y_1)$,

$$\text{Area} = \frac{1}{2}\sum_{i=1}^{N} \big(x_i y_{i+1} - x_{i+1} y_i\big).$$

```
# Goal: Convert ∬_Ω f dA to a boundary integral via Green, then do 1D quadrature.

input: polygon/spline boundary segments {gamma_e(t) on [0,1]}, choice of (P,Q) with
dQ/dx - dP/dy = f,
       1D Gauss-Legendre nodes {t_j, w_j}

I = 0
for each boundary segment gamma_e(t) = (x_e(t), y_e(t)), t in [0,1], oriented CCW:
  dxdt(t) = derivative of x_e at t
  dydt(t) = derivative of y_e at t
  I_e = 0
  for j in 1..n:
    tj = t_j
    xj = x_e(tj); yj = y_e(tj)
    I_e += w_j * ( P(xj,yj)*dxdt(tj) + Q(xj,yj)*dydt(tj) )

  I += I_e

return I

# Notes:
# - Ensure boundary orientation is counterclockwise for the sign to match Green's
theorem.
# - For piecewise-linear edges, dx/dt and dy/dt are constants per edge.
# - If using the divergence form, parameterize n and use F·n instead.
```

Quadrature in the Wild: From Physics to Finance

Quadrature is a universal tool for solving real-world problems across everything about math & science. Let's take a world tour of integration in action.

Physics: The Path Integral

In quantum mechanics, the probability amplitude for a particle to move from point A to point B is given by the path integral, which sums over all possible paths:

$$K(B, A) = \int \mathcal{D}x(t) e^{\frac{i}{\hbar} S[x(t)]}$$

where $S[x(t)]$ is the classical action for each path. This infinite-dimensional integral is notoriously difficult to compute, but in practice, physicists discretize time and use numerical quadrature on a high-dimensional space (via lattice methods). For simple potentials, Gaussian quadrature on each time slice can be efficient.

Think of it like this: *Imagine every possible wiggly path a photon could take from your flashlight to the wall. The path integral adds up the "quantumness" of each wild route. Quadrature helps us approximate this sum of all realities.*

Chemistry: The Electronic Structure Problem

The energy of a molecule is determined by solving the Schrödinger equation for electrons. This involves integrating the electron repulsion terms over molecular orbitals:

$$(ij|kl) = \iint \frac{\phi_i(\mathbf{r}_1)\phi_j(\mathbf{r}_1)\phi_k(\mathbf{r}_2)\phi_l(\mathbf{r}_2)}{|\mathbf{r}_1 - \mathbf{r}_2|} d\mathbf{r}_1 d\mathbf{r}_2$$

These 6-dimensional integrals (for each pair of electrons) are computed using Gaussian quadrature with atomic-specific grids. The choice of grid points and weights is critical for accuracy in codes like Gaussian or PySCF.

Think of it like this: *Calculating how electrons—those moody particles—avoid each other in a molecule is like predicting the drama at a dance party. Quadrature is the social algorithm that figures out who stands where to minimize awkward collisions.*

Biology: Pharmacokinetics

In drug modeling, the concentration of a drug in the body over time is described by integral equations. For example, the area under the curve (AUC) is a key metric:

$$\text{AUC} = \int_0^\infty C(t)dt$$

where $C(t)$ is the drug concentration. This integral is often computed using adaptive quadrature (like Gauss-Kronrod) from experimental data points, especially since $C(t)$ is exponential and decays rapidly.

Think of it like this: *How much medicine is in your system over time? Quadrature measures the total "dose hours" by adding up the concentration minute by minute, like counting the total rainfall from a storm.*

In drug modeling, the concentration of a drug in the body over time is described by integral equations. For example, the area under the curve (AUC) is a key metric:

$$\text{AUC} = \int_0^\infty C(t)dt$$

where $C(t)$ is the drug concentration. This integral is often computed using adaptive quadrature (like Gauss-Kronrod) from experimental data points, especially since $C(t)$ is exponential and decays rapidly.

Think of it like this: *How much medicine is in your system over time? Quadrature measures the total "dose hours" by adding up the concentration minute by minute, like counting the total rainfall from a storm.*

Maps & Geography: Geospatial Quadrature

To integrate fields on Earth (e.g., rainfall, irradiance) over a region, use the sphere or ellipsoid surface element

$$dA = R^2 \sin\theta \, d\theta \, d\phi$$

```
or solid angle $d\Omega$. Great circle domains, spherical polygons, or equal-area
pixelizations (e.g., HEALPix) all boil down to "sum values × areas."
```

```
# Average over region Γ on sphere (radius R):
```

$$\langle F\rangle = \frac{1}{\mathrm{Area}(\Gamma)}\iint_\Gamma F(\theta,\phi)\,dA, \quad dA = R^2\sin\theta\,d\theta\,d\phi$$

```
# Approaches: # (1) Equal-area grids (e.g., HEALPix) → sum F_i w_i # (2) Lebedev
nodes → high-order angular quadrature on S^2 # (3) Polygonal region: clip nodes to
Γ or triangulate the spherical polygon Given sample nodes $(\theta_i,\phi_i)$ with weights $w_i$:
```

$$I \approx \sum_{i\in\Gamma} w_i\,F(\theta_i,\phi_i), \qquad \mathrm{Area}(\Gamma)\approx\sum_{i\in\Gamma} w_i$$

```
(if weights integrate to 1 over the sphere)
```

***Why the** $\sin\theta$? Imagine Earth wearing "latitude belts." Belts near the equator are longer than those near the poles;* $\sin\theta$ *scales each belt's width to get the right area.*

Polygon areas on ellipsoid: Use spherical-excess or ellipsoidal algorithms; for raster data, sum cell means × cell areas. Quadrature is just "weighted summation."

DSP: Audio

Energy, filters, and spectral features: integrals of $|x(t)|^2$, filter kernels, or windowed transforms. Trapezoid on uniform samples is surprisingly powerful for periodic/FFT workflows.

```
# Short-time energy over a frame:
```

$$E \approx \Delta t\sum_n |x[n]|^2$$

```
# Filter output integral (continuous):
```

$$y(t_0) = \int h(\tau)\,x(t_0-\tau)\,d\tau \;\approx\; \sum_i w_i\,h(\tau_i)\,x(t_0-\tau_i)$$

***Fun fact:** Your ears handle ~*10^{12} *variation in sound power (≈120 dB). Energy integrals quantify "how loud" a clip is, not just how high the peaks are.*

Graphics: Rendering

Hemisphere sampling (lighting): integrate BRDF × incoming radiance over a hemisphere. Cosine-weighted quadrature matches the physics of Lambertian surfaces.

```
# Outgoing radiance:
```

$$L_o = \int_{\text{hemisphere}} f_r(\omega_i)\,L_i(\omega_i)\,\cos\theta_i\,d\omega_i$$

Choose nodes ω_i with weights w_i (cosine-weighted):

$$L_o \approx \sum_i w_i\, f_r(\omega_i)\, L_i(\omega_i)\, \cos\theta_i$$

Radiosity / form factors: double integrals over surfaces; discretize patches and apply Gauss rules per patch pair.

Picture it: *You're sampling the dome of light above each pixel. More samples near the normal (cosine-weighting) because those rays contribute most.*

Graphics: Quadrature for Animation and Quaternions

In computer graphics and animation, **quadrature** (numerical integration) is essential for computing smooth motion, blending rotations, and simulating physics. When rotations are represented by **quaternions**, integration ensures continuous and stable interpolation over time.

- **Quaternion Basics:** A quaternion $q = w + xi + yj + zk$ represents a rotation in 3D space without gimbal lock. Normalized quaternions satisfy:

$$\|q\| = \sqrt{w^2 + x^2 + y^2 + z^2} = 1.$$

- **Quaternion Differential Equation:** For angular velocity vector $\boldsymbol{\omega}(t)$, the quaternion derivative is:

$$\dot{q}(t) = \frac{1}{2}\,\Omega(\boldsymbol{\omega}(t))\,q(t),$$

  where

$$\Omega(\boldsymbol{\omega}) = \begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{bmatrix}.$$

- **Numerical Integration for Animation:** To update orientation over time:

$$q(t + \Delta t) \approx q(t) + \Delta t \cdot \dot{q}(t),$$

  then normalize q to maintain unit length. Higher-order quadrature (e.g., Runge-Kutta) improves accuracy:

$$q_{n+1} = q_n + \frac{\Delta t}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$

  where k_i are intermediate slopes from the quaternion ODE.

- **Spherical Linear Interpolation (SLERP):** For keyframe animation, interpolate between q_0 and q_1 on the unit sphere:

$$\mathrm{SLERP}(q_0, q_1; t) = \frac{\sin((1-t)\theta)}{\sin\theta} q_0 + \frac{\sin(t\theta)}{\sin\theta} q_1, \quad \theta = \arccos(q_0 \cdot q_1).$$

  This ensures constant angular velocity and smooth motion.

Real-World Applications

- **Character Animation:** Smooth blending of skeletal rotations using quaternion integration.

- **Camera Motion:** Interpolating camera orientation for cinematic paths.
- **Physics Engines:** Integrating angular velocity to update rigid body orientation.
- **VR/AR:** Stable head-tracking and orientation prediction using quaternion ODE integration.

Pseudocode: Quaternion Integration with RK4

```
// Inputs: q (unit quaternion), omega (angular velocity vector), dt (time step)
function integrate_quaternion(q, omega, dt):
    def dqdt(q, omega):
        Omega = [[0, -omega.x, -omega.y, -omega.z],
                 [omega.x, 0, omega.z, -omega.y],
                 [omega.y, -omega.z, 0, omega.x],
                 [omega.z, omega.y, -omega.x, 0]]
        return 0.5 * Omega * q

    k1 = dqdt(q, omega)
    k2 = dqdt(q + 0.5*dt*k1, omega)
    k3 = dqdt(q + 0.5*dt*k2, omega)
    k4 = dqdt(q + dt*k3, omega)

    q_new = q + (dt/6)*(k1 + 2*k2 + 2*k3 + k4)
    return normalize(q_new)
```

Video Games: Physics

Quadrature handles impulses, energies, occlusion integrals, and optimal control costs. Smooth gameplay often hides a lot of integrals under the hood.

```
# Controller cost:
```

$$J = \int_0^T \left(q\,x(t)^2 + r\,u(t)^2\right)\,dt \;\approx\; \sum_k w_k\left(q\,x(t_k)^2 + r\,u(t_k)^2\right)$$

Analogy: *It's like scoring how "wobbly" and "throttle-heavy" your car was over a lap. Integrate wobble and throttle over time to tune a smoother ride.*

Real-Estate: Architecture

Daylight/insolation on façades: integrate the sun path and visibility over time and angles. Shadowing splits the domain → adaptive rules shine (pun intended).

```
# Annual insolation at a point:
```

$$Q = \int_{\text{year}} \int_{\text{sun-visible}} I_{\text{sun}}(\alpha, \beta, t)\, \cos(\text{incidence})\, d\Omega\, dt \;\approx\; \sum_{t,\Omega} w_{t,\Omega}\, I_{\text{sun}}\, \cos(\text{incidence})$$

Rule of thumb: *A tiny change in panel tilt can integrate to big gains over a whole year. Quadrature helps find that sweet spot.*

Shipbuilding: Naval

Hydrostatics and resistance: integrate pressure or sectional areas along the hull. Traditional "curves of areas" are made for Simpson/Boole's rules.

```
# Displacement volume:
```

$$V = \int_0^L A(x)\, dx \;\approx\; \text{Simpson}(A(x), x \in [0, L])$$

Fun fact: *Long before GPUs, shipwrights used Simpson's rule on paper to estimate displacement from station drawings.*

Navigation / Logistics

Fuel/energy along a route: integrate consumption rate over time/distance. If grades or winds change sharply, adaptive quadrature saves calls.

```
# Fuel:
```

$$\text{Fuel} = \int \text{rate}(s(t), \text{grade}(t), \text{load})\, dt \;\approx\; \sum_k w_k\, \text{rate}(s_k, \text{grade}_k, \text{load})$$

Real world: *Two routes with the same length can burn very different fuel if one has hills. The integral remembers every slope.*

Remote Sensing: Crude Oil Detection

Oil films change spectral reflectance and polarization. We detect via band-integrated reflectance and angular models, then fuse evidence across pixels.

```
# Band-integrated reflectance in sensor band [λ1, λ2] with spectral response S(λ):
```

$$R_{\text{band}} \;=\; \frac{\displaystyle\int_{\lambda_1}^{\lambda_2} R(\lambda)\, S(\lambda)\, d\lambda}{\displaystyle\int_{\lambda_1}^{\lambda_2} S(\lambda)\, d\lambda}$$

# BRDF angular integration (sun-sensor geometry):

$$L_{\text{TOA}} \;\approx\; \iint_{\Omega_{\text{surf}}} f_r(\lambda, \omega_i \to \omega_o)\, E_{\text{sun}}(\lambda, \omega_i)\, \cos\theta_i\, d\omega_i$$

# Probabilistic detection (per pixel):

$$\log \mathcal{L}(\text{oil} \mid \mathbf{R}) \;=\; \sum_b \log p\big(R_b \;\; \mu_b^{\text{oil}}, \sigma_b^{\text{oil}}\big)$$

Why the rainbow sheen? *Thin films create interference—different wavelengths amplify/cancel at different thicknesses, changing the integral across the band the satellite sees.*

Crystal Science: Diffraction

Structure factors integrate electron density against complex exponentials. Powder diffraction averages orientations over the sphere—perfect for spherical quadrature.

# Structure factor at scattering vector q:

$$F(\mathbf{q}) = \sum_{j=1}^{N} f_j(q)\, e^{\,i\,\mathbf{q}\cdot\mathbf{r}_j}$$

# Debye scattering (pairwise distances r_{ij}):

$$I(q) \propto \sum_{i=1}^{N} \sum_{j=1}^{N} f_i(q) f_j(q)\, \frac{\sin(q\, r_{ij})}{q\, r_{ij}}$$

# Powder average (orientational integral on S^2):

$$I_{\text{powder}}(q) = \frac{1}{4\pi} \int_{S^2} F(R\,\mathbf{q})\;^2\, d\Omega \;\approx\; \sum_{\ell} w_\ell\, \big|F(R_\ell \mathbf{q})\big|^2$$

(Use Lebedev nodes R_ℓ, weights w_ℓ)

Think of it: *X-ray diffraction is the crystal's "barcode." The integral sums waves scattered by each atom; peaks appear where all waves add in sync.*

Lasers: Mode Overlap

Coupling efficiency is a normalized overlap of modes. Gaussian/Hermite rules excel for analytic beams; FE rules shine in complex waveguides.

$$\eta = \frac{\iint E_1(x,y)\,E_2^*(x,y)\,dx\,dy^{\;2}}{\left(\iint |E_1|^2\right)\left(\iint |E_2|^2\right)} \approx \sum_e \sum_q w_{e,q}\,(E_1\,E_2^*)$$

***Laser handshake:** Modes couple best when their "shapes" match. The integral is a compatibility score between the beams.*

Astrophysics

Distances, band fluxes, and lensing—all are integrals. Smooth 1D rules often suffice; angular pieces use spherical quadrature.

```
# Luminosity distance (flat ΛCDM; generalize with curvature if needed):
```

$$D_L(z) = (1+z)\,\frac{c}{H_0}\int_0^z \frac{dz'}{E(z')}, \quad E(z) = \sqrt{\Omega_m(1+z)^3 + \Omega_\Lambda + \Omega_r(1+z)^4}$$

```
# Band-integrated flux through a filter T(ν):
```

$$F_{\text{band}} = \frac{\displaystyle\int f_\nu(\nu)\,T(\nu)\,d\nu}{\displaystyle\int T(\nu)\,d\nu}$$

```
# (Sketch) Weak lensing potential (2D convolution integral):
```

$$\psi(\boldsymbol{\theta}) = \frac{1}{\pi}\iint \kappa(\boldsymbol{\theta}')\,\ln\,\boldsymbol{\theta} - \boldsymbol{\theta}'\big|\,d^2\theta'$$

***Cosmic road trip:** D_L integrates how the expansion stretches space between us and a galaxy. Filters then integrate a galaxy's light through the telescope's "color glasses."*

Finance: Local Volatility and Beyond

In finance, quadrature powers some of the most sophisticated models. Let's dive deeper.

Local Volatility (Dupire's Formula)

Dupire's formula calculates the local volatility surface from market option prices:

$$\sigma_{\text{local}}(S,t)^2 = \frac{\frac{\partial C}{\partial T} + (r-q)S\frac{\partial C}{\partial S} + qC}{\frac{1}{2}S^2\frac{\partial^2 C}{\partial S^2}}$$

The derivatives $\frac{\partial C}{\partial T}$, $\frac{\partial C}{\partial S}$, and $\frac{\partial^2 C}{\partial S^2}$ are computed from smoothed option price data using numerical differentiation. Quadrature is used implicitly when integrating the local volatility PDE to price options.

Think of it like this: *Local volatility is the "mood swings" of the market at every possible stock price and time. Quadrature helps us deduce these mood swings from the option prices traders are paying.*

SABR Model

The SABR model (Stochastic Alpha Beta Rho) describes the dynamics of forward rates with stochastic volatility:

$$dF = \alpha F^{\beta} dW_1, \quad d\alpha = \nu \alpha dW_2, \quad \mathbb{E}[dW_1 dW_2] = \rho dt$$

Option prices under SABR are computed using asymptotic expansions or by numerical integration of the probability density. For example, the option price can be written as:

$$C(K) = \int_0^{\infty} \int_0^{\infty} (F - K)^+ p(F, \alpha) dF d\alpha$$

where $p(F, \alpha)$ is the joint density. This 2D integral is efficiently computed with Gaussian quadrature or sparse grids.

Think of it like this: *SABR is like modeling a car's speed (the forward rate) and the driver's mood (volatility) simultaneously. Quadrature adds up all the possible outcomes of speed and mood swings to price the option.*

Heston Model

The Heston model we've seen earlier has a closed-form characteristic function, allowing option prices to be computed via Fourier quadrature (Carr-Madan). However, the integral can be oscillatory, and adaptive quadrature is often used to handle the oscillations efficiently.

Think of it like this: *The Heston model is the market's heartbeat—sometimes steady, sometimes erratic. Quadrature listens to the heartbeat's frequency components to price options.*

Stochastic Local Volatility (SLV)

SLV models combine local volatility and stochastic volatility to capture the best of both worlds. The option price is given by:

$$C = e^{-rT} \int_0^{\infty} \int_0^{\infty} (S - K)^+ p(S, v) dS dv$$

where $p(S, v)$ is the joint density of stock price and volatility. This 2D integral is computed using quadrature on a grid, often with adaptive methods in the volatility dimension.

Think of it like this: *SLV is like having a weather model that accounts for both local humidity (local vol) and global storm systems (stochastic vol). Quadrature is the supercomputer that adds up all the weather scenarios.*

FFT: The Quadrature in Disguise

The Fast Fourier Transform (FFT) is not just a fast algorithm; it's a powerful quadrature rule in disguise. Let's uncover the connection.

Chebyshev Nodes and Clenshaw-Curtis

Recall that Clenshaw-Curtis quadrature uses Chebyshev nodes:

$$x_k = \cos\left(\frac{k\pi}{n}\right), \quad k = 0, 1, \ldots, n$$

These nodes are the roots of Chebyshev polynomials and are clustered near the endpoints, which helps capture endpoint behavior. The weights for Clenshaw-Curtis can be computed via the FFT because the discrete cosine transform (DCT) is intimately related to the Chebyshev series.

Think of it like this: *Chebyshev nodes are like placing more sensors near the edges of a drum—where the skin is tightest and most sensitive. The FFT is the quick way to calculate the drum's sound from those sensors.*

FFT as a Quadrature Rule

The FFT itself can be viewed as a quadrature rule for integrating periodic functions. Consider integrating a periodic function $f(x)$ over $[0, 2\pi]$:

$$I = \int_0^{2\pi} f(x)dx$$

If we sample $f(x)$ at n equally spaced points $x_j = 2\pi j/n$, the trapezoidal rule gives:

$$I \approx \frac{2\pi}{n} \sum_{j=0}^{n-1} f(x_j)$$

But for periodic functions, the trapezoidal rule is exponentially accurate! The FFT can be used to compute this sum quickly, especially if $f(x)$ is smooth and periodic.

Think of it like this: *The FFT is like a revolving door that smoothly integrates people flowing in and out of a building. For periodic events (like rush hour), it's incredibly efficient.*

Convolution: The Integral that FFT Loves

Convolution is a fundamental operation in math and engineering:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

This integral appears everywhere: in signal processing (filtering), probability (sum of random variables), and finance (option pricing with stochastic processes). The convolution theorem states that the Fourier transform of a convolution is the product of the Fourier transforms:

$$\mathcal{F}\{f * g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\}$$

So, to compute convolution, we can FFT both functions, multiply, and then inverse FFT. This is often faster than direct numerical integration.

***Think of it like this:** Convolution is like blending two smoothies together. The FFT method is like using a super-blender that instantly combines their flavors, rather than stirring slowly with a spoon (quadrature).*

Quadrature for Convolution

But what if we want to use quadrature directly for convolution? We can discretize the integral:

$$(f * g)(t) \approx \sum_i w_i f(\tau_i) g(t - \tau_i)$$

This is a quadrature rule! However, for each t, we need to evaluate the sum, which is expensive. The FFT method computes this for all t simultaneously in $O(n \log n)$ time, whereas direct quadrature would be $O(n^2)$.

***Think of it like this:** Direct quadrature for convolution is like hand-mixing every possible combination of ingredients. The FFT is like using a mixer with many blades—it does all the combinations at once.*

Time is Money: Quadrature for Bermudan Options

Bermudan options can be exercised at specific dates before expiration. Pricing them requires solving a dynamic programming problem, and quadrature plays a key role.

The Dynamic Programming Equation

The value of a Bermudan option at time t_i is:

$$V(S, t_i) = \max\left(\text{payoff}(S), e^{-r\Delta t}\mathbb{E}[V(S_{t_{i+1}}, t_{i+1}) | S_{t_i} = S]\right)$$

The conditional expectation is an integral:

$$\mathbb{E}[V(S_{t_{i+1}})|S] = \int_0^\infty V(S', t_{i+1}) p(S'|S) dS'$$

where $p(S'|S)$ is the transition density. This integral is computed using quadrature at each time step.

***Think of it like this:** Pricing a Bermudan option is like deciding whether to take a gift now or wait for a potentially better gift later. Quadrature helps us calculate the value of waiting by adding up all future possibilities.*

Quadrature Methods

We can use Gaussian quadrature with points tailored to the transition density. For example, if the density is log-normal, Gauss-Laguerre quadrature is efficient. The process:

1. Discretize the asset price space into quadrature points $\{S_j\}$ at each exercise date.
2. At the final date, set $V(S,T) = \text{payoff}(S)$.
3. Move backwards in time: for each date, compute the continuation value at each S_j using quadrature:

$$C(S_j) = e^{-r\Delta t}\sum_k w_k V(S_k, t_{i+1}) p(S_k|S_j)$$

4. Set $V(S_j, t_i) = \max(\text{payoff}(S_j), C(S_j))$.

This is called the quadrature method for Bermudan options.

***Think of it like this:** It's like solving a maze backwards—starting from the end and working back to the beginning, using quadrature to shine a light on the best path at each step.*

FFT Acceleration

If the transition density has a closed-form characteristic function, the expectation can be written as a convolution. Then, FFT can be used to compute the continuation value for all asset prices simultaneously. This is faster than quadrature for large grids.

***Think of it like this:** FFT for Bermudan options is like using a megaphone to broadcast the value of waiting to all asset prices at once, rather than whispering to each one individually.*

The Grand Reduction: Taming a High-Dimensional Integral

Let's conjure a practical example from finance: pricing a basket option on multiple assets under a multivariate Black-Scholes model. We'll start with a high-dimensional integral and reduce it step by step.

The Problem

Consider a basket of d assets with prices $S_1, S_2, \ldots, S_d$ following correlated geometric Brownian motion. The basket call option payoff at time T is:

$$\text{payoff} = \max\left(\sum_{i=1}^{d} w_i S_i(T) - K, 0\right)$$

The risk-neutral price is:

$$C = e^{-rT} \int_{\mathbb{R}^d} \max\left(\sum_{i=1}^{d} w_i S_i(0) e^{(r-\frac{1}{2}\sigma_i^2)T+\sigma_i\sqrt{T}z_i} - K, 0\right) \phi_d(\mathbf{z}; \Sigma) d\mathbf{z}$$

where $\phi_d(\mathbf{z}; \Sigma)$ is the multivariate normal density with correlation matrix Σ. This is a d-dimensional integral.

Step 1: Change of Variables - Cholesky Decomposition

We decompose $\Sigma = LL^\top$, where L is lower triangular. Then, we set $\mathbf{z} = L\mathbf{y}$, where $\mathbf{y} \sim \mathcal{N}(0, I)$. The integral becomes:

$$C = e^{-rT} \int_{\mathbb{R}^d} \max\left(\sum_{i=1}^{d} w_i S_i(0) e^{(r-\frac{1}{2}\sigma_i^2)T+\sigma_i\sqrt{T}(L\mathbf{y})_i} - K, 0\right) \phi(y_1)\phi(y_2)\cdots\phi(y_d) d\mathbf{y}$$

Now the integrand is a product of independent normal densities.

Think of it like this: *We're untangling the correlated assets into independent factors, like separating intertwined threads.*

Step 2: Dimension Reduction via Conditioning

If the basket is large, we can use a conditioning variable. Let $X = \sum_{i=1}^{d} w_i S_i(T)$. We can write:

$$C = e^{-rT} \mathbb{E}[\max(X - K, 0)] = e^{-rT} \int_0^\infty \mathbb{P}(X > k) dk$$

But $\mathbb{P}(X > k)$ is still hard to compute. Alternatively, we can use a comonotonic approximation or principal component analysis (PCA) to reduce the dimension.

Step 3: PCA Reduction

Perform PCA on the correlation matrix Σ. The first few principal components explain most of the variance. Let P be the matrix of eigenvectors. Then, $\mathbf{z} = P\mathbf{u}$, where $\mathbf{u}$ are the principal components. We can truncate the sum after $m < d$ components:

$$z_i \approx \sum_{j=1}^{m} P_{ij} u_j$$

The integral becomes m-dimensional. For example, if $m = 2$, we have:

$$C \approx e^{-rT} \int_{\mathbb{R}^2} \max\left(\sum_{i=1}^{d} w_i S_i(0) e^{(r-\frac{1}{2}\sigma_i^2)T+\sigma_i\sqrt{T}\sum_{j=1}^{2} P_{ij}u_j} - K, 0\right) \phi(u_1)\phi(u_2)\,du_1 du_2$$

Think of it like this: *PCA is like finding the main actors in a play—the few that drive the plot. We focus on them and ignore the extras.*

Step 4: Numerical Integration

Now we have a 2D integral. We can compute it using Gaussian quadrature:

$$C \approx e^{-rT} \sum_{k=1}^{n_1} \sum_{l=1}^{n_2} w_k w_l \max\left(\sum_{i=1}^{d} w_i S_i(0) e^{(r-\frac{1}{2}\sigma_i^2)T+\sigma_i\sqrt{T}(P_{i1}u_k+P_{i2}u_l)} - K, 0\right)$$

where u_k and u_l are Gauss-Hermite nodes, and w_k, w_l are the corresponding weights.

Step 5: Validation with Monte Carlo

To validate, we set up a Monte Carlo simulation:

1. Generate N samples of $\mathbf{y} \sim \mathcal{N}(0, I_d)$.
2. Transform to $\mathbf{z} = L\mathbf{y}$.
3. Compute $S_i(T) = S_i(0)e^{(r-\frac{1}{2}\sigma_i^2)T+\sigma_i\sqrt{T}z_i}$ for each asset.
4. Compute the basket value $X = \sum_i w_i S_i(T)$.
5. Average the payoff: $C \approx e^{-rT}\frac{1}{N}\sum \max(X-K, 0)$.

We use variance reduction techniques like antithetic sampling to improve accuracy.

Think of it like this: *Monte Carlo is the party where we invite millions of random scenarios to see how they behave. Quadrature is the careful interview of a few representative scenarios.*

Step 6: Compare Results

For a basket with $d = 10$ assets, we might find that PCA with $m = 2$ components captures 95% of the variance. The quadrature method with $20 \times 20 = 400$ points might be faster and more accurate than Monte Carlo with $100,000$ paths.

Key Insight: Dimension reduction is like compressing a high-resolution image into a smaller file without losing important details. Quadrature then works efficiently on the compressed version.

Conclusion

Quadrature methods aren't just a set of dusty formulas from a numerical analysis textbook — they're a precision-engineered toolkit for turning hairy integrals into neat, well-behaved numbers. In financial mathematics, they let you move seamlessly from the humble trapezoidal rule to the Rolls-Royce of Gaussian quadrature, choosing the right ride for the terrain between speed and accuracy.

In option pricing, quadrature can be the Goldilocks method: not as assumption-bound as closed-form solutions, not as variance-hungry as Monte Carlo. Whether you're bending the problem into a strike-aware integral, flipping it into Fourier space, or re-casting it as a stop-loss premium, quadrature has a way of finding the sweet spot — integrating form and function, if you will.

Practical Implementation Tips:

- Test convergence with increasing node counts — don't just take it on faith.
- Choose transforms to expose natural weights — let the integrand show you where it wants to be sampled.
- Prefer nested rules for adaptivity — reuse points, save cycles, keep your grid from going off on a tangent.
- Vectorize integrand evaluations — because looping over points one-by-one is so last century.
- For per-strike precision, use quadrature; for whole surfaces, use FFT — know when to sum it up and when to transform.

Remember: *Quadrature is like hiring a team of elite surveyors who know exactly where to stand to measure the landscape. Monte Carlo is like sending a million tourists with tape measures and hoping the average works out. Both have their place, but when you need to "integrate" into the market quickly, quadrature can help you avoid going off on a random walk. Oh here's a joke - "Why did the quadrature rule break up with Monte Carlo? Because it wanted something more deterministic in its life!"*

So, as you head off to tackle your next pricing problem, keep your wits sharp and your weights positive. Don't be afraid to change variables if the view looks better from another coordinate system. And if anyone tells you integration is boring, just smile and say: "I've got *bounds* on that opinion." After all, in the grand sum of things, it's not just about getting the area under the curve — it's about making every point count.

References & Further Reading

1. Carr, P. & Madan, D. (1999). Option valuation using the fast Fourier transform. *Journal of Computational Finance.*
2. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P. *Numerical Recipes: The Art of Scientific Computing* (3rd ed.). Cambridge University Press. (The definitive handbook for numerical algorithms).
3. Golub, G. H., & Welsch, J. H. (1969). Calculation of Gauss quadrature rules. *Mathematics of Computation*, 23(106), 221–230. (The seminal Golub-Welsch paper).
4. Trefethen, L. N. (2008). Is Gauss quadrature better than Clenshaw–Curtis? *SIAM Review*, 50(1), 67–87. (A fantastic and surprising comparison).

5. Mori, M. (1974). Quadrature formulas obtained by variable transformation. *Publications of the Research Institute for Mathematical Sciences*, 9(3), 121–130. (The original Japanese publication on tanh-sinh quadrature).
6. Deelstra, G., Liinev, J., & Vanmaele, M. (2004). Pricing of arithmetic basket options by conditioning. *Insurance: Mathematics and Economics*, 34(1), 55–77.
7. Heston, S. L. (1993). A closed-form solution for options with stochastic volatility with applications to bond and currency options. *The Review of Financial Studies*, 6(2), 327–343.
8. Hagan, P. S., Kumar, D., Lesniewski, A. S., & Woodward, D. E. (2002). Managing smile risk. *Wilmott Magazine*, 1, 84–108. (The SABR model paper).
9. Heinrich, S. (2001). Quantum integration in Sobolev classes. *Journal of Complexity*, 17(1), 1-16. (Theoretical foundations of quantum numerical integration).
10. Rebentrost, P., Gupt, B., & Bromley, T. R. (2018). Quantum computational finance: Monte Carlo pricing of financial derivatives. *Physical Review A*, 98(2), 022321. (Application of amplitude estimation to finance).
11. Montanaro, A. (2015). Quantum speedup of Monte Carlo methods. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 471(2181), 20150301. (A key paper on the quadratic speedup).
12. Sanders, J., & Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Addison-Wesley Professional. (The beginner-friendly bible for CUDA).
13. Kirk, D. B., & Hwu, W. W. (2016). *Programming Massively Parallel Processors: A Hands-on Approach* (3rd ed.). Morgan Kaufmann. (The advanced, comprehensive guide to GPU computing).
14. NVIDIA Corporation. (2024). *CUDA C++ Programming Guide*. https://docs.nvidia.com/cuda/cuda-c-programming-guide/ (The ultimate official reference).
15. Gropp, W., Lusk, E., & Skjellum, A. (2014). *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (3rd ed.). MIT Press. (The standard textbook for MPI).
16. Message Passing Interface Forum. (2015). *MPI: A Message-Passing Interface Standard Version 3.1*. https://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf (The formal specification).
17. Giles, M. B. (2015). Multilevel Monte Carlo methods. *Acta Numerica*, 24, 259-328. (Not directly quadrature, but a crucial modern MC method for validation).
18. Gander, W., & Gautschi, W. (2000). Adaptive Quadrature—Revisited. *BIT Numerical Mathematics*, 40(1), 84–101. (An excellent review of robust adaptive methods).
19. Boyd, J. P. (2001). *Chebyshev and Fourier Spectral Methods* (2nd ed.). Dover. (The comprehensive work linking polynomials, FFT, and quadrature).