Wolfram *Mathematica*® Tutorial Collection

ADVANCED NUMERICAL INTEGRATION IN *MATHEMATICA*

For use with Wolfram *Mathematica*® 7.0 and later.

For the latest updates and corrections to this manual:
visit reference.wolfram.com

For information on additional copies of this documentation:
visit the Customer Service website at www.wolfram.com/services/customerservice
or email Customer Service at info@wolfram.com

Comments on this manual are welcomed at:
comments@wolfram.com

Printed in the United States of America.

15 14 13 12 11 10 9 8 7 6 5 4 3 2

Wolfram Research, Inc. and Wolfram Media, Inc. ("Wolfram") make no representations, express, statutory, or implied, with respect to the Software (or any aspect thereof), including, without limitation, any implied warranties of merchantability, interoperability, or fitness for a particular purpose, all of which are expressly disclaimed. Wolfram does not warrant that the functions of the Software will meet your requirements or that the operation of the Software will be uninterrupted or error free. As such, Wolfram does not recommend the use of the software described in this document for applications in which errors or omissions could threaten life, injury or significant loss.

Contents

NIntegrate Introduction

Overview

The *Mathematica* function `NIntegrate` is a general numerical integrator. It can handle a wide range of one-dimensional and multidimensional integrals.

`NIntegrate`$[f[x_1,x_2,\ldots,x_n],\{x_1,a_1,b_1\},\{x_2,a_2,b_2\},\ldots,\{x_n,a_n,b_n\}]$	
	find a numerical integral for the function f over the region $[a_1,\ b_2]\times[a_2,\ b_2]\times\ldots\times[a_n,\ b_n]$

Finding a numerical integral of a function over a region.

In general, `NIntegrate` estimates the integral through sampling of the integrand value over the integration region. The various numerical integration methods prescribe the initial sampling steps and how the sampling evolves.

`NIntegrate` uses algorithms called "integration strategies" that attempt to compute integral estimates that satisfy user-specified precision or accuracy goals. The integration strategies use "integration rules" that compute integral estimates using weighted sums.

This numerically computes the integral $\int_0^1 \frac{1}{\sqrt{x}}\,dx$.

In[25]:= **NIntegrate**$\left[\frac{1}{\sqrt{x}},\{x,0,1\}\right]$

Out[25]= `2.`

`NIntegrate` uses symbolic preprocessing that simplifies integrals with piecewise functions and even or odd functions. Part of the symbolic preprocessing is the detection of one-dimensional oscillatory integrals of the types `NIntegrate` can handle efficiently.

This integrates a piecewise function over the interval $[0,\ 2]$.

In[26]:= **NIntegrate**$\left[\frac{1}{\sqrt{\text{Abs}[x-1]}},\ \{x,\ 0,\ 2\}\right]$

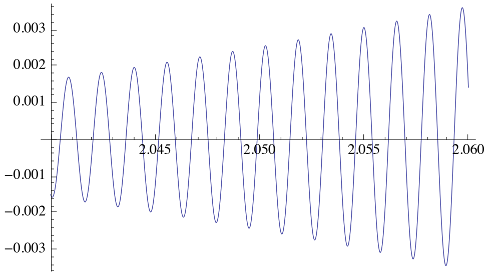
Out[26]= `4.`

This integrates a highly oscillatory function over the interval $[2, 3]$.

```
In[27]:= NIntegrate[(x - 2)^2 Sin[4000 x], {x, 2, 3}]
Out[27]= -0.000158625
```

This is a plot of the previous oscillatory integrand over $\frac{1}{50}$ of the integration region.

```
In[28]:= Plot[(x - 2)^2 Sin[4000 x], {x, 2 + 2/50, 2 + 3/50}]
Out[28]=
```



This integrates a piecewise combination of a piecewise function and an oscillatory function.

```
In[29]:= NIntegrate[
          Piecewise[{{1/√Abs[x - 1], x < 2}, {(x - 2)^2 Sin[4000 x], 2 < x < 3}}], {x, 0, 3}]
Out[29]= 3.99984
```

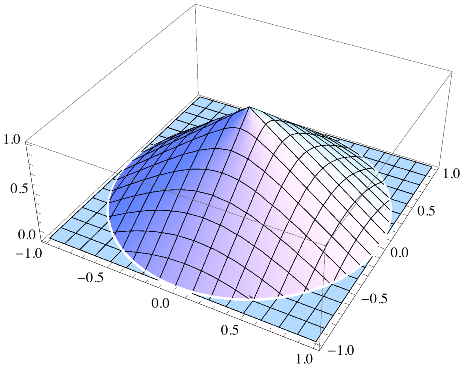
`NIntegrate` oscillatory algorithms are only for one-dimensional integrals. The oscillatory algorithms for finite regions are different from the oscillatory algorithms for infinite regions.

One-dimensional numerical integration is much simpler, and better understood, than multidimensional numerical integration. This is the reason a distinction between the two is made. All `NIntegrate` strategies except the oscillatory strategies can be used for multidimensional integration.

Here is a two-dimensional function: a cone with base in the square $[-1, 1] \times [-1, 1]$.

```
In[30]:= Plot3D[Boole[x^2 + y^2 < 1] * (1 - √(x^2 + y^2)), {x, -1, 1}, {y, -1, 1}, PlotRange → All]
```

Out[30]=



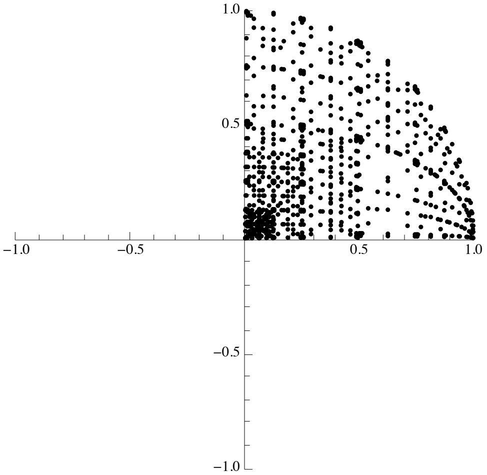
Here is the integral of the cone function.

```
In[7]:= NIntegrate[Boole[x^2 + y^2 < 1] * (1 - √(x^2 + y^2)), {x, -1, 1}, {y, -1, 1}]

Out[7]= 1.0472
```

Here are the sampling points used by `NIntegrate`. Note that the sampling points are only in a quarter of the integration region.

```
In[8]:= Graphics[{PointSize[0.01],
          Point[Reap[NIntegrate[Boole[x^2 + y^2 < 1] * (1 - √(x^2 + y^2)), {x, -1, 1},
               {y, -1, 1}, EvaluationMonitor :→ Sow[{x, y}]]][[2, 1]]]},
         Axes -> True, PlotRange → {{-1, 1}, {-1, 1}}]
```

Out[8]=



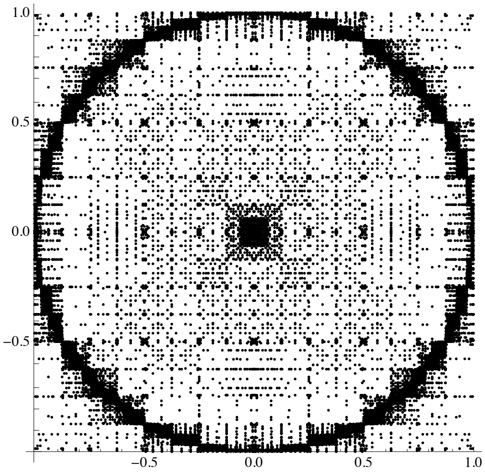
Here are the sampling points used by `NIntegrate` without symbolic preprocessing. (The reason that `NIntegrate` gives the `slwcon` message is because no symbolic preprocessing is applied.) Note that the sampling points are in the whole integration region and that they are denser around the circumference of the cone base and around the cone apex.

In[9]:=

```
Graphics[{PointSize[0.005],
  Point[Reap[NIntegrate[Boole[x^2 + y^2 < 1] * (1 - √(x^2 + y^2)), {x, -1, 1}, {y, -1, 1},
      Method -> {Automatic, "SymbolicProcessing" → 0}, EvaluationMonitor :→
       Sow[{x, y}]]][[2, 1]]]}, Axes → True, AxesOrigin → {-1, -1}]
```

```
NIntegrate::slwcon :
  Numerical integration converging too slowly; suspect one of the following: singularity, value
    of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫
```

Out[9]=



`NIntegrate` has several ways to deal with singular integrands. The deterministic adaptive strategies `"GlobalAdaptive"` and `"LocalAdaptive"` use singularity handling techniques (based on variable transformations) to speed up the convergence of the integration process. The strategy `"DoubleExponential"` employs trapezoidal quadrature with a special variable transformation on the integrand. This rule-transformation combination achieves optimal convergence for integrands analytic on an open set in the complex plane containing the interval of integration. The strategy `"DuffyCoordinates"` simplifies or eliminates certain types of singularities in multidimensional integrals.

Here is a one-dimensional integration with singularity handling.

In[2]:=

```
NIntegrate[1/√x, {x, 0, 1}, PrecisionGoal → 10] // Timing
```

Out[2]=

```
{0.006999, 2.}
```

Without singularity handling the previous integral is computed more slowly.

```
In[3]:=  NIntegrate[1/Sqrt[x], {x, 0, 1}, Method → {"GlobalAdaptive", "SingularityDepth" → ∞},
           MaxRecursion → 100, PrecisionGoal → 10] // Timing

Out[3]=  {0.008999, 2.}
```

For multidimensional integrands that have certain spherical symmetry the strategy `"DuffyCoordinates"` converges quite fast.

Here is a `"DuffyCoordinates"` integration.

```
In[12]:=  NIntegrate[1/Sqrt[x^2 + y^2 + z^2], {x, 0, 1},
            {y, 0, 1}, {z, 0, 1}, Method → "DuffyCoordinates"] // Timing

Out[12]=  {0.031, 1.19004}
```

Here is a computation of the previous integral with the default settings; it is approximately 5 times slower.

```
In[13]:=  NIntegrate[1/Sqrt[x^2 + y^2 + z^2], {x, 0, 1}, {y, 0, 1}, {z, 0, 1}] // Timing

Out[13]=  {0.203, 1.19004}
```

The `"Trapezoidal"` strategy gives optimal convergence for analytic periodic integrands when the integration interval is exactly one period.

Here is a calculation of an integral computed with the trapezoidal strategy. The result is compared with the exact value. The result computed with `"Trapezoidal"` is obtained faster and it is more precise than the one with default `NIntegrate` settings.

```
In[5]:=  exact = Integrate[Cos[20 x]^4, {x, 0, 2 π/20}]

Out[5]=  3 π/80
```

```
In[37]:=  resTrap = NIntegrate[Cos[20 x]^4, {x, 0, 2 π/20}, PrecisionGoal → 150,
               WorkingPrecision → 200, Method → "Trapezoidal"]; // Timing

Out[37]=  {0.015, Null}
```

```
In[40]:=  Abs[exact - resTrap]

Out[40]=  0. × 10^-201
```

Here is a (slower) computation of the same integral but with the default `Method` settings for `NIntegrate`.

```
In[38]:= resDef = NIntegrate[Cos[20 x]^4, {x, 0, 2 π/20},
             PrecisionGoal → 150, WorkingPrecision → 200]; // Timing

Out[38]= {0.219, Null}
```

```
In[39]:= Abs[exact - resDef]

Out[39]= 0.×10^-201
```

For multidimensional integrals, or in cases when only a rough integral estimate is needed, Monte Carlo methods are useful. `NIntegrate` has both crude and adaptive Monte Carlo and quasi Monte Carlo strategies.

Here is a multidimensional integral done quickly with a Monte Carlo algorithm.

```
In[19]:= X = Array[x, 30];
         NIntegrate[1/(Total@X), Evaluate[Sequence @@ Map[{#, 0, 1} &, X]],
          Method → "AdaptiveMonteCarlo", PrecisionGoal → 3]

Out[20]= 0.0674103
```

Design

Features

The principal features of the `NIntegrate` framework are:

- Code reuse (common code base)
- Object orientation (method property specification and communication)
- Data hiding
- Separation of method initialization phase and runtime computation
- Hierarchical and reentrant numerical methods
- Type- and precision-dynamic methods
- User extensibility and prototyping through plug-in capabilities
- Specialized data structures

Strategies, Rules, and Preprocessors

`NIntegrate` strategies can be divided into two general groups: deterministic and Monte Carlo. Each group can be divided further into adaptive, nonadaptive, and specialized strategies. Adaptive strategies try to improve the integral estimate by concentrating their efforts around the problematic areas. Non-adaptive strategies try to improve the integral estimate just by increasing the number of sampling points in the integration region. Specialized strategies are made for certain types of integrals (e.g., a product of an oscillatory and a non-oscillatory function).

Strategies	*Deterministic*	*Monte Carlo*
adaptive	"GlobalAdaptive" "LocalAdaptive"	"AdaptiveMonteCarlo" "AdaptiveQuasiMonteCarlo"
nonadaptive	"DoubleExponential" "Trapezoidal"	"MonteCarlo"
specialized	"DuffyCoordinates" "Oscillatory" "PrincipalValue"	

`NIntegrate` built-in integration strategies.

The strategies `"GlobalAdaptive"` and `"LocalAdaptive"` can have specifications of what integration rules to use.

Here is an example of `"GlobalAdaptive"` with an integration rule specification.

In[21]:=

$$\texttt{NIntegrate}\left[\frac{1}{\sqrt{x}}\,\texttt{Log}\left[\frac{1}{x}\right],\ \{x,\ 0,\ 1\},\ \texttt{Method} \to \{\texttt{"GlobalAdaptive"},\ \texttt{Method} \to \texttt{"ClenshawCurtisRule"}\}\right]$$

Out[21]=

```
4.
```

Both `"GlobalAdaptive"` and `"LocalAdaptive"` adaptive strategies can be used with one-dimensional and multidimensional integration rules.

	rules

one-dimensional	`"BooleRule"`
	`"ClenshawCurtisRule"`
	`"GaussBerntsenEspelidRule"`
	`"GaussKronrodRule"`
	`"LobattoKronrodRule"`
	`"LobattoPeanoRule"`
	`"MultiPanelRule"`
	`"NewtonCotesRule"`
	`"PattersonRule"`
	`"SimpsonThreeEightsRule"`
	`"TrapezoidalRule"`

multidimensional	`"CartesianRule"` `"MultiDimensionalRule"`

Built-in integration rules that can be used by `"GlobalAdaptive"` and `"LocalAdaptive"`.

The capabilities of all strategies are extended through integral preprocessing. The preprocessors can be seen as strategies that delegate integration to other strategies (preprocessors included).

Here is an example of the preprocessing of an integrand which is even with respect to each of its variables.

In[22]:=

$$\texttt{NIntegrate}\left[\texttt{Boole}\left[x^2+y^2<1\right]*\left(1-\sqrt{x^2+y^2}\right),\ \{x,-1,1\},\ \{y,-1,1\},\ \texttt{Method -> \{"EvenOddSubdivision", Method} \to \texttt{"LocalAdaptive"\}}\right]$$

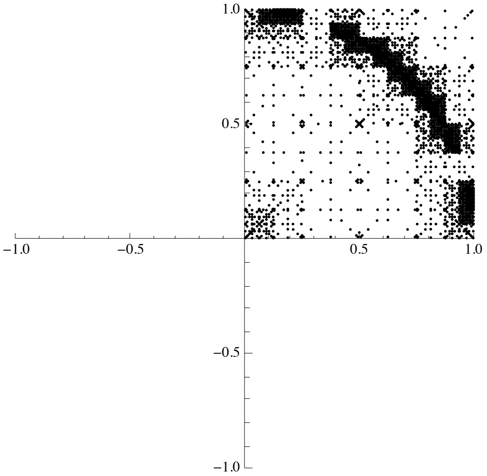
Out[22]=

```
1.0472
```

Here are the sampling points of the previous integration. If no preprocessing had been done, the plot would have been in the region $[-1, 1] \times [-1, 1]$ with a symmetry along both the x axis and the y axis.

```
In[23]:= Graphics[{PointSize[0.005],
           Point[Reap[NIntegrate[Boole[x^2 + y^2 < 1] * (1 - Sqrt[x^2 + y^2]), {x, -1, 1},
                 {y, -1, 1}, Method -> {"EvenOddSubdivision", Method → "LocalAdaptive"},
                 EvaluationMonitor :→ Sow[{x, y}]]][[2, 1]]]},
          Axes → True, PlotRange → {{-1, 1}, {-1, 1}}]
```

Out[23]=



preprocessors
"SymbolicPiecewiseSubdivision"
"EvenOddSubdivision"
"OscillatorySelection"
"UnitCubeRescaling"

NIntegrate preprocessors.

User Extensibility

Built-in methods can be used as building blocks for the efficient construction of special-purpose integrators. User-defined integration rules and strategies can also be added.

NIntegrate Integration Strategies

Introduction

An integration strategy is an algorithm that attempts to compute integral estimates that satisfy user-specified precision or accuracy goals.

An integration strategy prescribes how to manage and create new elements of a set of disjoint subregions of the initial integral region. Each subregion might have its own integrand and integration rule associated with it. The integral estimate is the sum of the integral estimates of all subregions. Integration strategies use integration rules to compute the subregion integral estimates. An integration rule samples the integrand with a set of points, called abscissas (or sampling points).

To improve an integral estimate the integrand should be sampled at additional points. There are two principal approaches: (i) adaptive strategies try to identify the problematic integration areas and concentrate the computational efforts (i.e., sampling points) on them; (ii) non-adaptive strategies increase the number of sampling points over the whole region in order to compute a higher degree integration rule estimate that reuses the integrand evaluations of the former integral estimate.

Both approaches can use symbolic preprocessing and variable transformation or sequence summation acceleration to achieve faster convergence.

In the following integration, the symbolic piecewise preprocessor in `NIntegrate` recognizes the integrand as a piecewise function, and the integration is done over regions for which $x \geq 1$ with the integrand $\frac{1}{\sqrt{x-1}}$ and regions for which $x \leq 1$ with the integrand $\frac{1}{\sqrt{1-x}}$.

In[31]:=

$$\mathbf{NIntegrate}\left[\frac{\mathbf{1}}{\sqrt{\mathbf{Abs}[\mathbf{x}-\mathbf{1}]}},\ \{\mathbf{x},\ \mathbf{0},\ \mathbf{2}\}\right]$$

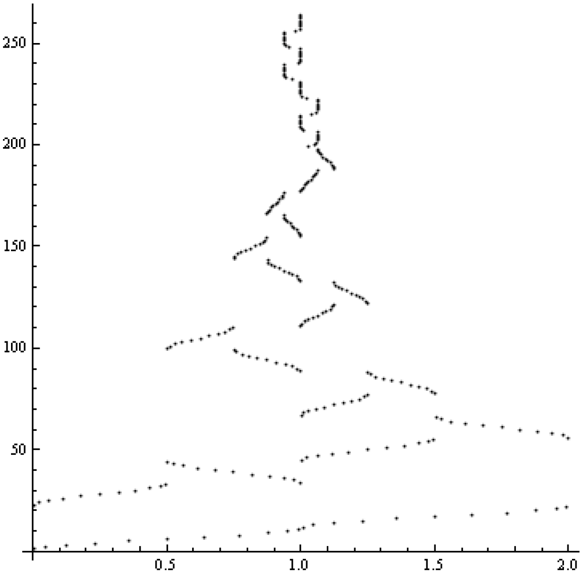
Out[31]=

```
4.
```

Here is a plot of all sampling points used in the integration. The integrand is sampled at the x coordinates in the order of the y coordinates (in the plot). It can be seen that the sampling points are concentrated near the singularity point 1. The patterns formed by the sampling points at the upper part of the plot differ from the patterns of the lower part of the plot because a singularity handler is applied.

```
In[10]:= points =
           Reap[NIntegrate[1/Sqrt[Abs[x - 1]], {x, 0, 2}, EvaluationMonitor :> Sow[x]]][[2, 1]];
         Graphics[{PointSize[0.006],
           Point /@ N@Transpose[{points, Range[Length[points]]}]},
          PlotRange → All, AspectRatio -> 1, Axes -> True]
```

Out[11]=



The section "Adaptive Strategies" gives a general description of the adaptive strategies. The default (main) strategy of `NIntegrate` is global adaptive, which is explained in the section "Global Adaptive Strategy". Complementary to it is the local adaptive strategy, which is explained in the section "Local Adaptive Strategy". Both adaptive strategies use singularity handling mechanisms, which are explained in the section "Singularity Handling".

The Monte Carlo strategies are explained in the sections "Crude Monte Carlo and Quasi Monte Carlo Strategies" and "Global Adaptive Monte Carlo and Quasi Monte Carlo Strategies".

The strategies `NIntegrate` uses for special types of integrals (or integrands) are explained in the corresponding sections: "Duffy's coordinates strategy", "Oscillatory strategies", and "Cauchy principal value integration".

Here is a table with links to descriptions of built-in integration strategies of `NIntegrate`.

strategies	deterministic	Monte Carlo
adaptive	`"GlobalAdaptive"` `"LocalAdaptive"`	`"AdaptiveMonteCarlo"` `"AdaptiveQuasiMonteCarlo"`
non-adaptive	`"DoubleExponential"` `"Trapezoidal"`	`"MonteCarlo"`

specialized	`"DuffyCoordinates"`
	`"Oscillatory"`
	`"PrincipalValue"`	

Adaptive Strategies

Adaptive strategies try to concentrate computational efforts where the integrand is discontinuous or has some other kind of singularity. Adaptive strategies differ by the way they partition the integration region into disjoint subregions. The integral estimates of each subregion contribute to the total integral estimate.

The basic assumption for the adaptive strategies is that for given integration rule R and integrand f, if an integration region V is partitioned into, say, two disjoint subregions V_1 and V_2, $V = V_1 \bigcup V_2$, $V_1 \bigcap V_2 = 0$, then the sum of the integral estimates of R over V_1 and V_2 is closer to the actual integral $\int_V f \, dx$. In other words,

$$\left| \int_V f \, dx - R_V(f) \right| > \left| \int_V f \, dx - R_{V_1}(f) + R_{V_2}(f) \right|, \tag{1}$$

and (1) will imply that the sum of the error estimates for $R_{V_1}(f)$ and $R_{V_2}(f)$ is smaller than the error estimate of $R_V(f)$.

Hence an adaptive strategy has these components [MalcSimp75]:

(i) an integration rule to compute the integral and error estimates over a region;

(ii) a method for deciding which elements of a set of regions $\{V_i\}_{i=1}^{n}$ to partition/subdivide;

(iii) stopping criteria for deciding when to terminate the adaptive strategy algorithm.

Global Adaptive Strategy

A global adaptive strategy reaches the required precision and accuracy goals of the integral estimate by recursive bisection of the subregion with the largest error estimate into two halves, and computes integral and error estimates for each half.

The global adaptive algorithm for `NIntegrate` is specified with the `Method` option value `"GlobalAdaptive"`.

```
In[32]:= NIntegrate[1 / Sqrt[x], {x, 0, 1}, Method -> "GlobalAdaptive"]

Out[32]= 2.
```

option name	*default value*	
`Method`	`Automatic`	integration rule used to compute integral and error estimates over each subregion
`"SingularityDepth"`	`Automatic`	number of recursive bisections before applying a singularity handler
`"SingularityHandler"`	`Automatic`	singularity handler
`"SymbolicProcessing"`	`Automatic`	number of seconds to do symbolic preprocessing

`"GlobalAdaptive"` options.

`"GlobalAdaptive"` is the default integration strategy of `NIntegrate`. It is used for both one-dimensional and multidimensional integration. `"GlobalAdaptive"` works with both Cartesian product rules and fully symmetric multidimensional rules.

`"GlobalAdaptive"` uses a data structure called a "heap" to keep the set of regions partially sorted, with the largest error region being at the top of the heap. In the main loop of the algorithm the largest error region is bisected in the dimension that is estimated to be responsible for most of its error.

It can be said that the algorithm produces the leaves of a binary tree, the nodes of which are the regions. The children of a node/region are its subregions obtained after bisection.

After a bisection of a region and the subsequent integration over the new (sub)regions, new global integral and global error estimates are computed, which are sums of the integral and error estimates of all regions that are leaves of the binary tree.

Each region has a record of how many bisections are made per dimension in order to produce it. When a region has been produced through too many bisections a singularity flattening algorithm is applied to it; see Singularity Handling.

`"GlobalAdaptive"` stops if the following expression is true:

$$\texttt{globalError} \leq \texttt{globalIntegral}\; 10^{-\texttt{pg}} \bigvee \texttt{globalError} \leq 10^{-\texttt{ag}}, \qquad (2)$$

where `pg` and `ag` are precision and accuracy goals.

The strategy also stops when the number of recursive bisections of a region exceeds a certain number (see MinRecursion and MaxRecursion), or when the global integration error oscillates too much (see "MaxErrorIncreases").

Theoretical and practical evidence show that the global adaptive strategies have in general better performance than the local adaptive strategies [MalcSimp75][KrUeb98].

MinRecursion and MaxRecursion

The minimal and maximal depths of the recursive bisections are given by the values of the options `MinRecursion` and `MaxRecursion`.

If for any subregion the number of bisections in any of the dimensions is greater than `MaxRecursion` then the integration by `"GlobalAdaptive"` stops.

Setting `MinRecursion` to a positive integer forces recursive bisection of the integration regions before the integrand is ever evaluated. This can be done to ensure that a narrow spike in the integrand is not missed. (See Tricking the error estimator.)

For multidimensional integration an effort is made to bisect in each dimension for each level of recursion in `MinRecursion`.

"MaxErrorIncreases"

Since (1) is expected to hold in `"GlobalAdaptive"` the global error is expected to decrease after the bisection of the largest error region and the integration over its new parts. In other words the global error is expected to be more or less monotonically decreasing with respect to the number of integration steps.

The global error might oscillate due to phase errors of the integration rules. Still, the global error is assumed at some point to start decreasing monotonically.

Below are listed cases in which this assumption might become false.

(i) The actual integral is zero.

Zero integral.

```
In[3]:= NIntegrate[Sin[x], {x, 0, 2 π}, MaxRecursion -> 100]
Out[3]= 0.
```

(ii) The specified working precision is not dense enough for the specified precision goal.

The working precision is not dense enough.

```
In[33]:= NIntegrate[1 / Sqrt[x], {x, 0, 1},
           MaxRecursion → 100, PrecisionGoal -> 17] // InputForm

           NIntegrate::slwcon :
             Numerical integration converging too slowly; suspect one of the following: singularity, value
                 of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

           NIntegrate::eincr :
             The global error of the strategy GlobalAdaptive has increased more than 400 times. The global error is
                 expected to decrease monotonically after a number of integrand evaluations. Suspect one
                 of the following: the difference between the values of PrecisionGoal and WorkingPrecision
                 is too small; the integrand is highly oscillatory or it is not a (piecewise) smooth function;
                 or the true value of the integral is 0. Increasing the value of the GlobalAdaptive option
                 MaxErrorIncreases might lead to a convergent numerical integration. NIntegrate obtained
                 2.0000000000000018` and 2.1241892251243344`*^-16 for the integral and error estimates. ≫
Out[33]//InputForm= 2.0000000000000018
```

(iii) The integration is badly conditioned [KrUeb98]. For example, the reason might be that the integrand is defined by complicated expressions or in terms of approximate solutions of mathematical problems (such as differential equations or nonlinear algebraic equations).

The strategy `"GlobalAdaptive"` keeps track of the number of times the total error estimate has not decreased after the bisection of the region with the largest error estimate. When that number becomes bigger than the value of the `"GlobalAdaptive"` option `"MaxErrorIncreases"`, the integration stops with a message (`NIntegrate::eincr`).

The default value of `"MaxErrorIncreases"` is 400 for one-dimensional integrals and 2000 for multidimensional integrals.

The following integration invokes the message `NIntegrate::eincr`, with the default value of `"MaxErrorIncreases"`.

```
In[1]:= NIntegrate[Sin[x^2 + x], {x, 0, 80 Pi}, Method → "GlobalAdaptive", MaxRecursion → 100]

          NIntegrate::slwcon :
            Numerical integration converging too slowly; suspect one of the following: singularity, value
                of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

          NIntegrate::eincr :
            The global error of the strategy GlobalAdaptive has increased more than 400 times. The global error is
                expected to decrease monotonically after a number of integrand evaluations. Suspect one
                of the following: the difference between the values of PrecisionGoal and WorkingPrecision
                is too small; the integrand is highly oscillatory or it is not a (piecewise) smooth function;
                or the true value of the integral is 0. Increasing the value of the GlobalAdaptive option
                MaxErrorIncreases might lead to a convergent numerical integration. NIntegrate obtained
                2.972314689667426` and 9.140875003915308` for the integral and error estimates. ≫

Out[1]= 0.
```

Increasing `"MaxErrorIncreases"` silences the `NIntegrate::eincr` message.

```
In[2]:= res = NIntegrate[Sin[x^2 + x], {x, 0, 80 Pi},
          Method → {"GlobalAdaptive", "MaxErrorIncreases" → 10 000}, MaxRecursion → 20]

          NIntegrate::slwcon :
            Numerical integration converging too slowly; suspect one of the following: singularity, value
                of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

Out[2]= 0.533246
```

The result compares well with the exact value.

```
In[3]:= exact = Integrate[Sin[x^2 + x], {x, 0, 80 Pi}];
        Abs[res - exact]

Out[4]= 6.84008×10^-13
```

Example Implementation of a Global Adaptive Strategy

This computes Gauss-Kronrod abscissas, weights, and error weights.

```
In[15]:= {absc, weights, errweights} =
           NIntegrate`GaussKronrodRuleData[5, MachinePrecision];
```

This is a definition of a function that applies the integration rule with abscissas and weights computed to the function f over the interval $\{a, b\}$.

```
In[16]:= IRuleEstimate[f_, {a_, b_}] :=
          Module[{integral, error},
            {integral, error} = (b - a) Total@MapThread[{f[#1] #2, f[#1] #3} &,
                {Rescale[absc, {0, 1}, {a, b}], weights, errweights}];
            {integral, Abs[error]}
          ]
```

This is a definition of a simple global adaptive algorithm that finds the integral of the function f over the interval $\{aArg, bArg\}$ with relative error tol.

```
In[17]:= IStrategyGlobalAdaptive[f_, {aArg_, bArg_}, tol_] :=
           Module[{t, integral, error, regions, r1, r2, a = aArg, b = bArg, c},

             {integral, error} = IRuleEstimate[f, {a, b}];
             (* boundaries, integral, error *)
             regions = {{{a, b}, integral, error}};

             While[error >= tol * integral,
               (* splitting of the region with the largest error *)
                                                  a + b
               {a, b} = regions[[1, 1]]; c = ─────;
                                                    2

               (* integration of the left region *)
               {integral, error} = IRuleEstimate[f, {a, c}];
               r1 = {{a, c}, integral, error};

               (* integration of the right region *)
               {integral, error} = IRuleEstimate[f, {c, b}];
               r2 = {{c, b}, integral, error};

               (* sort the regions: the largest error one is the first *)
               regions = Join[{r1, r2}, Rest[regions]];
               regions = Sort[regions, #1[[3]] > #2[[3]] &];

         (* global integral and error *)
               {integral, error} = Total[Map[Rest[#1] &, regions]];

             ];

             integral
           ];
```

This defines an integrand.

```
In[18]:= f[x_] := 1 / Sqrt[x]
```

The global adaptive strategy defined earlier gives the following result.

```
In[19]:= res = IStrategyGlobalAdaptive[f, {0, 1}, 10^-8]
Out[19]= 2.
```

Here is the exact result.

```
In[20]:= exact = Integrate[f[x], {x, 0, 1}]
Out[20]= 2
```

The relative error is within the prescribed tolerance.

```
In[21]:= Abs[res - exact] / exact
Out[21]= 2.63409×10^-9
```

Local Adaptive Strategy

In order to reach the required precision and accuracy goals of the integral estimate, a local adaptive strategy recursively partitions the subregion into smaller disjoint subregions and computes integral and error estimates for each of them.

The local adaptive algorithm for `NIntegrate` is specified with the `Method` option value `"LocalAdaptive"`.

```
In[5]:= NIntegrate[1 / Sqrt[x], {x, 0, 1}, Method -> "LocalAdaptive"]

Out[5]= 2.
```

option name	*default value*	
`Method`	`Automatic`	integration rule used to compute integral and error estimates over the subregions
`"SingularityDepth"`	`Automatic`	number of recursive bisections before applying a singularity handler
`"SingularityHandler"`	`Automatic`	singularity handler
`"Partitioning"`	`Automatic`	how to partition the regions in order to improve their integral estimate
`"InitialEstimateRelaxation"`	`True`	attempt to adjust the magnitude of the initial integral estimate in order to avoid unnecessary computation
`"SymbolicProcessing"`	`Automatic`	number of seconds to do symbolic preprocessing

`"LocalAdaptive"` options.

Like `"GlobalAdaptive"`, `"LocalAdaptive"` can be used for both one-dimensional and multidimensional integration. `"LocalAdaptive"` works with both Cartesian product rules and fully symmetric multidimensional rules.

The `"LocalAdaptive"` strategy has an initialization routine and a Recursive Routine (RR). RR produces the leaves of a tree, the nodes of which are regions. The children of a node/region are subregions obtained by its partition. RR takes a region as an argument and returns an integral estimate for it.

RR uses an integration rule to compute integral and error estimates of the region argument. If the error estimate is too big, RR calls itself on the region's disjoint subregions obtained by partition. The sum of the integral estimates returned from these recursive calls becomes the region's integral estimate.

RR makes the decision to continue the recursion knowing only the integral and error estimates of the region at which it is executed. (This is why the strategy is called "local adaptive.")

The initialization routine computes an initial estimation of the integral over the initial regions. This initial integral estimate is used in the stopping criteria of RR: if the error of a region is significant compared to the initial integral estimate then that region is partitioned into disjoint regions and RR is called on them; if the error is insignificant the recursion stops.

The error estimate of a region, `regionError`, is considered insignificant if

$$\texttt{initialIntegral} + \texttt{regionError} == \texttt{initialIntegral}. \tag{3}$$

The stopping criteria (3) will compute the integral to the working precision. Since you want to compute the integral estimate to user-specified precision and accuracy goals, the following stopping criteria is used:

$$\begin{aligned}&\texttt{integralEst} = \texttt{Min}\left[\texttt{initialIntegral}\;10^{-\texttt{pg}}\big/\texttt{eps},\; 10^{-\texttt{ag}}\big/\texttt{eps}\right];\\ &\texttt{integralEst} + \texttt{regionError} == \texttt{integralEst},\end{aligned} \tag{4}$$

where `eps` is the smallest number such that $1 + \texttt{eps} \neq 1$ at the working precision, and `pg` and `ag` are the user-specified precision and accuracy goals.

The recursive routine of `"LocalAdaptive"` stops the recursion if:

1. there are no numbers of the specified working precision between region's boundaries;
2. the maximum recursion level is reached;
3. the error of the region is insignificant, i.e., the criteria (4) is true.

MinRecursion and MaxRecursion

The options `MinRecursion` and `MaxRecursion` for `"LocalAdaptive"` have the same meaning and functionality as they do for `"GlobalAdaptive"`. See MinRecursion and MaxRecursion.

"InitialEstimateRelaxation"

After the first recursion is finished a better integral estimate, I_2, will be available. That better estimate is compared to the two integral estimates, I_1 and I_{1e} that the integration rule has used to give the integral estimate (I_1) and the error estimate ($|I_1 - I_{1e}|$) for the initial step. If

$$\rho = \frac{|I_2 - I_1|}{|I_2 - I_{1e}|} < 1,$$

then the integral estimate `integralEst` in (4) can be increased—that is, the condition (4) is relaxed—with the formula

```
integralEst = integralEst / ρ,
```

since $\rho < 1$ means that the rule's integral estimate is more accurate than what the rule's error estimate predicts.

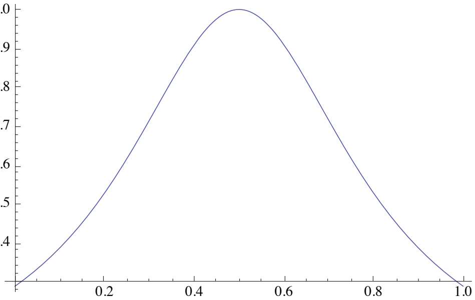
"Partitioning"

`"LocalAdaptive"` has the option `"Partitioning"` to specify how to partition the regions that do not satisfy (4). For one-dimensional integrals, if `"Partitioning"` is set to `Automatic`, `"LocalAdaptive"` partitions a region between the sampling points of the (rescaled) integration rule. In this way, if the integration rule is of closed type, every integration value can be reused. If `"Partitioning"` is given a list of integers $\{p_1, p_2, \ldots, p_n\}$ with length n that equals the number of integral variables, each dimension i of the integration region is divided into p_i equal parts. If `"Partitioning"` is given an integer p, all dimensions are divided into p equal parts.

Consider the following function.

```
In[4]:= Plot[1/(10 (1/2 - x)^2 + 1), {x, 0, 1}]
```

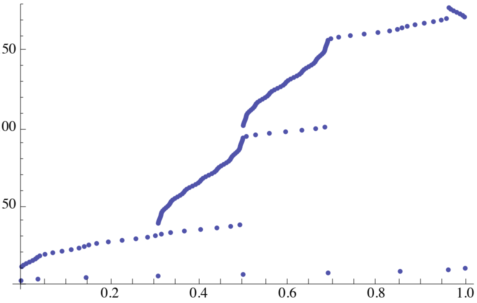
Out[4]=



These are the sampling points used by "`LocalAdaptive`" with its automatic region partitioning. It can be seen that the sampling points of each recursion level are between the sampling points of the previous recursion level.

```
In[1]:= sampledPoints = Reap[NIntegrate[1/(10 (1/2 - x)^2 + 1), {x, 0, 1},
            Method → {"LocalAdaptive"}, EvaluationMonitor :→ Sow[x]]][[2, 1]];
        ListPlot[Transpose[{sampledPoints, Range[Length[sampledPoints]]}]]
```

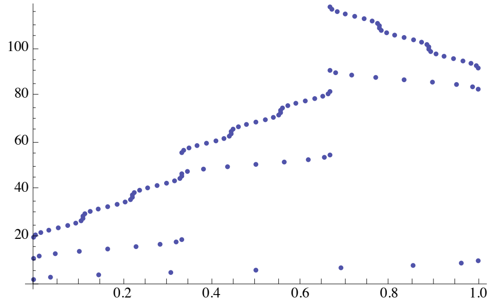
Out[2]=



These are the sampling points used by "`LocalAdaptive`" integration which partitions the regions with large error into three subregions. The patterns formed clearly show the three next recursion level subregions of each region of the first and second recursion levels.

In[5]:=
```
sampledPoints = Reap[NIntegrate[1/(10 (1/2 - x)^2 + 1),
     {x, 0, 1}, Method → {"LocalAdaptive", "Partitioning" → 3},
     EvaluationMonitor :> Sow[x]]][[2, 1]];
ListPlot[Transpose[{sampledPoints, Range[Length[sampledPoints]]}]]
```

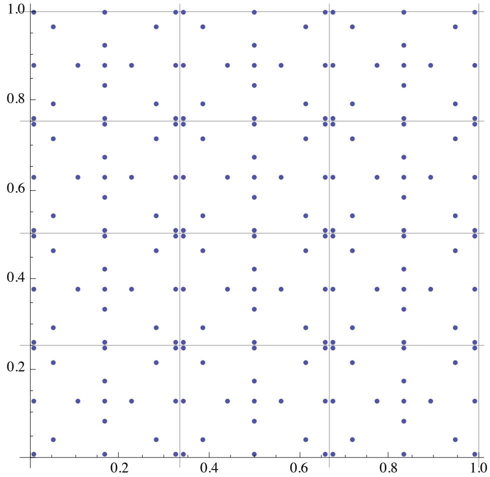
Out[6]=



Multidimensional example of using the "Partitioning" option. To make the plot, the sampling points of the first region to be integrated, $[0, 1]\times[0, 1]$, are removed.

In[7]:=
```
sampledPoints =
  Reap[NIntegrate[(x + y)^6, {x, 0, 1}, {y, 0, 1}, Method → {"LocalAdaptive",
       "Partitioning" → {3, 4}}, EvaluationMonitor :> Sow[{x, y}]]][[2, 1]];
sampledPoints = Partition[sampledPoints, Length[sampledPoints] / (3 * 4 + 1)];
sampledPoints = Flatten[Rest[sampledPoints], 1];
ListPlot[sampledPoints, AspectRatio → 1, GridLines -> {Range[3] / 3, Range[4] / 4}]
```

Out[10]=



Reuse of Integrand Values

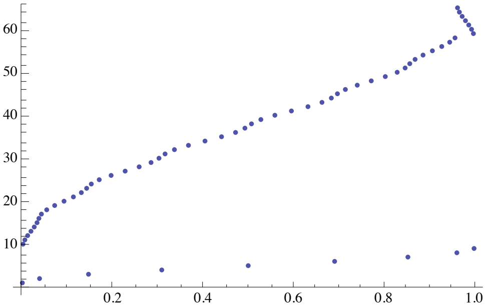
With its default partitioning settings for one-dimensional integrals "LocalAdaptive" reuses the integrand values at the end points of the sub-intervals that have integral and error estimates that do not satisfy (4).

Sampling points of the integration of $\int_0^1 x^6 \, dx$ by "LocalAdaptive". The variable rulePoints determines the number of points in the integration rule used by "LocalAdaptive".

```
In[13]:= rulePoints = 5;
         sampledPoints =
           Reap[NIntegrate[x^6, {x, 0, 1}, Method → {"LocalAdaptive", "SymbolicProcessing" →
                 0, Method → {"ClenshawCurtisRule", "Points" → rulePoints},
               "SingularityHandler" → None}, EvaluationMonitor ⧴ Sow[x]]][[2, 1]];
         Length[sampledPoints]
         ListPlot[Transpose[{sampledPoints, Range[Length[sampledPoints]]}]]
```

```
Out[15]= 65
```

Out[16]=



The percent of reused points in the integration.

```
In[17]:= totalRulePoints = 2 rulePoints - 1;
         totalPoints = (totalRulePoints - 1) totalRulePoints + totalRulePoints;
          totalPoints - Length[sampledPoints]
         ───────────────────────────────────── // N
                     totalPoints
```

```
Out[19]= 0.197531
```

Example Implementation of a Local Adaptive Strategy

This computes Clenshaw-Curtis abscissas, weights, and error weights.

```
In[33]:= {absc, weights, errweights} =
           NIntegrate`ClenshawCurtisRuleData[6, MachinePrecision];
```

This is a definition of a function that applies the integration rule, with the abscissas and weights computed in the previous example, to the function f over the interval $\{a, b\}$.

```
In[34]:= IRuleEstimate[f_, {a_, b_}] :=
          Module[{integral, error, scaledAbsc},
           scaledAbsc = Rescale[absc, {0, 1}, {a, b}];
           {integral, error} = (b - a)
             Total@MapThread[{f[#1] #2, f[#1] #3} &, {scaledAbsc, weights, errweights}];
           {integral, Abs[error], scaledAbsc}
          ]
```

This defines a simple local adaptive algorithm that finds the integral of the function f over the interval $\{aArg, bArg\}$ with relative error tol.

```
In[35]:= LocalAdaptiveRecurrence[f_, {a_, b_}, integralEst_] :=
           Module[{regions, integral, error, scaledAbsc},

            {integral, error, scaledAbsc} = IRuleEstimate[f, {a, b}];

            If[N[integralEst + error] == N[integralEst],
             (* Stopping criteria is satisfied *)
             integral,
             (* ELSE call itself recursively *)
             regions = Partition[scaledAbsc, 2, 1];
             Total[LocalAdaptiveRecurrence[f, #1, integralEst] & /@ regions]
            ]
           ];

         IStrategyLocalAdaptive[f_, {aArg_, bArg_}, tol_] :=
           Module[{integral, error, a = aArg, b = bArg, d = 1, dummy},

            If[a > b, {a, b} = {b, a}; d = -1];

            (* initial integral estimate *)
            {integral, error, dummy} = IRuleEstimate[f, {a, b}];

            d * LocalAdaptiveRecurrence[f, {a, b}, d * integral * tol / $MachineEpsilon]
           ];
```

This defines a function.

```
In[37]:= f[x_] := Sqrt[x] * Sin[x]
```

The local adaptive strategy gives the result.

```
In[38]:= res = IStrategyLocalAdaptive[f, {0, 8 π}, 10^-8]
Out[38]= -4.38857
```

This is the exact result.

```
In[39]:= exact = Integrate[f[x], {x, 0, 8 π}]
```

Out[39]= $\sqrt{\frac{\pi}{2}}\ (-4+\text{FresnelC}[4])$

The relative error is within the prescribed tolerance.

```
In[40]:= Abs[res - exact] / exact
```

Out[40]= -2.03056×10^{-11}

"GlobalAdaptive" versus "LocalAdaptive"

In general the global adaptive strategy has better performance than the local adaptive one. In some cases though the local adaptive strategy is more robust and/or gives better performance.

There are two main differences between `"GlobalAdaptive"` and `"LocalAdaptive"`:

1. `"GlobalAdaptive"` stops when the sum of the errors of all regions satisfies the precision goal, while `"LocalAdaptive"` stops when the error of each region is small enough compared to an estimate of the integral.

2. To improve the integral estimate `"GlobalAdaptive"` bisects the region with largest error, while `"LocalAdaptive"` partitions all regions the error for which is not small enough.

For multidimensional integrals `"GlobalAdaptive"` is much faster because `"LocalAdaptive"` does partitioning along each axis, so the number of regions can explode combinatorically.

Why and how global adaptive strategy is faster for one-dimensional smooth integrands is proved (and explained) in [MalcSimp75].

When `"LocalAdaptive"` is faster and performs better than `"GlobalAdaptive"`, it is because the precision-goal-stopping criteria and partitioning strategy of `"LocalAdaptive"` are more suited for the integrand's nature. Another factor is the ability of `"LocalAdaptive"` to reuse the integral values of all points already sampled. `"GlobalAdaptive"` has the ability to reuse very few integral values (at most 3 per rule application, 0 for the default one-dimensional rule, the Gauss-Kronrod rule).

The following subsection demonstrates the performance differences between `"GlobalAdaptive"` and `"LocalAdaptive"`.

"GlobalAdaptive" Is Generally Better than "LocalAdaptive"

The table that follows, with timing ratios and numbers of integrand evaluations, demonstrates that `"GlobalAdaptive"` is better than `"LocalAdaptive"` for the most common cases. All integrals considered are one-dimensional over $[0, 1]$, because (i) for multidimensional integrals the performance differences should be expected to deepen, since `"LocalAdaptive"` partitions the regions along each axis, and (ii) one-dimensional integrals over different ranges can be rescaled to be over $[0, 1]$.

Here are the definitions of some functions, precision goals, number of integrations, and the integration rule. The variable `integrationRule` can be changed in order to compare the profiling runs with the same integration rule. The last function is derived from $e^{-x}\sin(x)$ by the variable change $x \to -1 + \frac{1}{1-x}$ that transforms $[0, 1)$ into $[0, \infty)$.

In[70]:=

$$\texttt{funcs} = \left\{\sqrt{x},\ \frac{1}{\sqrt{x}},\ \frac{\texttt{Sin}[200\,x]}{\sqrt{x}},\ \texttt{Log}[x],\ x^{26},\ \frac{1}{10^4\left(\frac{1}{2}-x\right)^2+1},\ -\frac{e^{1-\frac{1}{1-x}}\,\texttt{Sin}\left[1-\frac{1}{1-x}\right]}{(1-x)^2}\right\};$$

```
precs = {6, 8, 10, 12, 14};
n = 10; (* number of integrations to determine the timing *)
integrationRule = Automatic;
```

```
In[74]:= FRangesToCube[{{x, 0, ∞}}]
```

Out[74]=

$$\left\{\left\{x \to -1 + \frac{1}{1-x}\right\},\ \frac{1}{(1-x)^2}\right\}$$

Exact integral values. All integrals are over $[0, 1]$.

```
In[75]:= exactvals = Integrate[#, {x, 0, 1}] & /@ funcs;
```

"`GlobalAdaptive`" timings.

```
In[76]:= gatimings =
           Map[First@Timing[Do[NIntegrate[#[[1]], {x, 0, 1}, PrecisionGoal → #[[2]],
                 Method → {"GlobalAdaptive", "SymbolicProcessing" → 0,
                   Method → integrationRule, "SingularityHandler" → None},
                 MaxRecursion → 200], {n}]] &, Outer[List, funcs, precs, 1], {2}];
```

"`LocalAdaptive`" timings.

```
In[77]:= latimings =
           Map[First@Timing[Do[NIntegrate[#[[1]], {x, 0, 1}, PrecisionGoal → #[[2]],
                 Method → {"LocalAdaptive", "SymbolicProcessing" → 0,
                   Method → integrationRule, "SingularityHandler" → None},
                 MaxRecursion → 200], {n}]] &, Outer[List, funcs, precs, 1], {2}];
```

"`GlobalAdaptive`" function evaluations.

```
In[78]:= ganfevals =
           Map[(k = 0; res = NIntegrate[#[[1, 1]], {x, 0, 1}, PrecisionGoal → #[[2]], Method →
                 {"GlobalAdaptive", "SymbolicProcessing" → 0, Method → integrationRule,
                  "SingularityHandler" → None}, MaxRecursion → 200,
                EvaluationMonitor :→ k++]; {k, Abs[res - #[[1, 2]]] / Abs[#[[1, 2]]]}) &,
            Outer[List, Transpose[{funcs, exactvals}], precs, 1], {2}];
```

"`LocalAdaptive`" function evaluations.

```
In[79]:= lanfevals =
           Map[(k = 0; res = NIntegrate[#[[1, 1]], {x, 0, 1}, PrecisionGoal → #[[2]], Method →
                 {"LocalAdaptive", "SymbolicProcessing" → 0, Method → integrationRule,
                  "SingularityHandler" → None}, MaxRecursion → 200,
                EvaluationMonitor :→ k++]; {k, Abs[res - #[[1, 2]]] / Abs[#[[1, 2]]]}) &,
            Outer[List, Transpose[{funcs, exactvals}], precs, 1], {2}];
```

Table with the timing ratios and integrand evaluations.

```
In[80]:= Grid[Prepend[Transpose@{funcs, ColumnForm /@ Table[precs, {Length[funcs]}],
             ColumnForm /@ (latimings / gatimings), ColumnForm[First /@ #] & /@ ganfevals,
             ColumnForm[First /@ #] & /@ lanfevals},
           Style[#, "SmallText"] & /@ {"functions", "precision goals",
              "LocalAdaptive timings / GlobalAdaptive timings", "GlobalAdaptive\nfunction\nevaluations",
              "LocalAdaptive\nfunction\nevaluations"}], Frame → All]
```

Out[80]=

functions	precision goals	$\frac{\text{LocalAdaptive timings}}{\text{GlobalAdaptive timings}}$	GlobalAdaptive function evaluations	LocalAdaptive function evaluations
$\sqrt{x}$	6 8 10 12 14	0.916655 1.28947 0.916675 1.79999 2.30768	165 253 407 649 1023	121 289 569 1017 1969
$\frac{1}{\sqrt{x}}$	6 8 10 12 14	1.18185 1.55555 1.94999 2.25424 2.8324	715 1045 1683 2651 4125	568 1184 2416 4376 8632
$\frac{\mathrm{Sin}[200\,x]}{\sqrt{x}}$	6 8 10 12 14	2.45784 2.95364 3.97817 4.95302 5.92579	1595 3047 4807 6237 11 913	3032 7064 14 736 24 144 53 768
Log[x]	6 8 10 12 14	20.5385 22.3438 19.8297 9.18835 6.21505	341 495 781 1243 1925	9080 9080 9080 9080 9080
x^{26}	6 8 10 12 14	2.56254 5.76456 5.09996 5.60002 9.80017	77 121 165 297 407	177 737 1353 2137 2697
$\frac{1}{1+10\,000\left(\frac{1}{2}-x\right)^2}$	6 8 10 12 14	1.74996 1.72977 2.17773 3.35385 4.8646	297 495 649 1089 1705	513 737 1297 3201 5329
$-\left(e^{1-\frac{1}{1-x}}\,\mathrm{Sin}\left[1-\frac{1}{1-x}\right]\right)\Big/\,(1-x)^2$	6 8 10 12 14	1.80001 2.24996 4.51424 4.1915 7.31431	165 231 363 583 1001	288 512 1184 1632 4376

Table with the errors of the integrations. Both "GlobalAdaptive" and "LocalAdaptive" reach the required precision goals.

In[81]:=
```
Grid[Prepend[Transpose@{funcs, ColumnForm /@ Table[precs, {Length[funcs]}],
    ColumnForm[#[[2]] & /@ #] & /@ ganfevals,
    ColumnForm[#[[2]] & /@ #] & /@ lanfevals}, Style[#, "SmallText"] & /@
   {"functions", "precision goals", "GlobalAdaptive\nrelative errors",
    "LocalAdaptive\nrelative errors"}], Frame → All]
```

Out[81]=

functions	precision goals	GlobalAdaptive relative errors	LocalAdaptive relative errors
$\sqrt{x}$	6 8 10 12 14	3.59747×10^{-8} 5.62106×10^{-10} 3.10635×10^{-12} 1.81521×10^{-14} 9.99201×10^{-16}	1.82143×10^{-8} 1.35204×10^{-10} 1.00403×10^{-12} 7.49401×10^{-15} 1.66533×10^{-16}
$\frac{1}{\sqrt{x}}$	6 8 10 12 14	4.76822×10^{-7} 2.63409×10^{-9} 1.02884×10^{-11} 5.56222×10^{-14} 1.11022×10^{-15}	2.16468×10^{-7} 1.60735×10^{-9} 1.19349×10^{-11} 4.52083×10^{-13} 3.21965×10^{-15}
$\frac{\text{Sin}[200\,x]}{\sqrt{x}}$	6 8 10 12 14	1.35856×10^{-8} 7.50499×10^{-11} 1.46188×10^{-13} 6.11798×10^{-15} 8.04997×10^{-16}	1.55299×10^{-9} 1.13674×10^{-11} 8.51687×10^{-14} 1.288×10^{-15} 1.77099×10^{-15}
Log[x]	6 8 10 12 14	9.6888×10^{-8} 7.56936×10^{-10} 5.91283×10^{-12} 2.23155×10^{-14} 6.66134×10^{-16}	0. 0. 0. 0. 0.
x^{26}	6 8 10 12 14	5.80785×10^{-15} 5.6205×10^{-15} 1.31145×10^{-15} 1.31145×10^{-15} 1.31145×10^{-15}	1.90696×10^{-11} 4.68375×10^{-14} 0. 0. 0.
$\frac{1}{1+10\,000\left(\frac{1}{2}-x\right)^2}$	6 8 10 12 14	1.78976×10^{-15} 1.90162×10^{-15} 1.34232×10^{-15} 1.34232×10^{-15} 1.45418×10^{-15}	5.1618×10^{-11} 7.94206×10^{-14} 2.2372×10^{-16} 1.1186×10^{-16} 2.2372×10^{-16}
$-\frac{e^{1-\frac{1}{1-x}}\,\text{Sin}\left[1-\frac{1}{1-x}\right]}{(1-x)^2}$	6 8 10 12 14	1.24141×10^{-10} 1.77636×10^{-15} 1.77636×10^{-15} 1.77636×10^{-15} 1.11022×10^{-15}	1.47526×10^{-12} 1.83453×10^{-12} 1.8463×10^{-12} 0. 0.

Singularity Handling

The adaptive strategies of `NIntegrate` speed up their convergence through variable transformations at the integration region boundaries and user-specified singular points or manifolds. The adaptive strategies also ignore the integrand evaluation results at singular points.

Singularity specification is discussed in "User-specified Singularities".

Multidimensional singularity handling with variable transformations should be used with caution; see "IMT Multidimensional Singularity Handling". Coordinate change for a multidimensional integral can simplify or eliminate singularities; see "Duffy's Coordinates for Multidimensional Singularity Handling".

For details about how `NIntegrate` ignores singularities see "Ignoring the Singularity".

The computation of Cauchy principal value integrals is described in "Cauchy Principal Value Integration".

User-Specified Singularities

Point Singularities

If it is known where the singularities occur, they can be specified in the ranges of integration, or through the option `Exclusions`.

Here is an example of an integral that has two singular points at $\frac{\pi}{6}$ and $\frac{\pi}{3}$.

In[58]:= $$\mathbf{NIntegrate}\left[\frac{1}{\sqrt{\left(x-\frac{\pi}{6}\right)}}\,\frac{1}{\sqrt{\left(x-\frac{\pi}{3}\right)}},\ \left\{x,\,0,\,\frac{\pi}{6},\,\frac{\pi}{3},\,\frac{\pi}{2}\right\}\right]$$

Out[58]= $3.87444\times10^{-8}-3.14159\,i$

Here is an example of a two-dimensional integral with a singular point at $(1,1)$.

In[59]:= `NIntegrate[Log[(1 - x)^2 + (1 - y)^2], {x, 0, 1, 2}, {y, 0, 1, 2}]`

Out[59]= `-2.94423`

Here is an example of an integral that has two singular points at $\frac{\pi}{6}$ and $\frac{\pi}{3}$ specified with the `Exclusions` option.

In[60]:=

$$\texttt{NIntegrate}\left[\frac{1}{\sqrt{\left(x-\frac{\pi}{6}\right)}}\ \frac{1}{\sqrt{\left(x-\frac{\pi}{3}\right)}},\ \left\{x,\ 0,\ \frac{\pi}{2}\right\},\ \texttt{Exclusions}\to\left\{\frac{\pi}{6},\ \frac{\pi}{3}\right\}\right]$$

Out[60]= $3.87444\times10^{-8} - 3.14159\,i$

Here is an example of a two-dimensional integral with a singular point at $(1, 1)$ specified with the `Exclusions` option.

```
In[61]:= NIntegrate[Log[(1 - x)^2 + (1 - y)^2], {x, 0, 2}, {y, 0, 2}, Exclusions → {{1, 1}}]
Out[61]= -2.94423
```

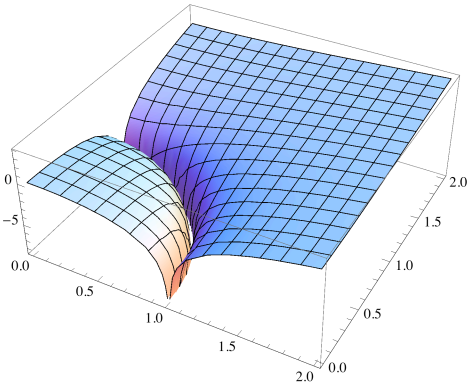
Curve, Surface, and Hypersurface Singularities

Singularities over curves, surfaces, or hypersurfaces in general can be specified through the option `Exclusions` with their equations. Such singularities, generally, cannot be specified using variable ranges.

This two-dimensional function is singular along the curve $x^2 + y^2 = 1$.

```
In[62]:= Plot3D[Log[(1 - (x^2 + y^2))^2], {x, 0, 2}, {y, 0, 2}, Exclusions → x^2 + y^2 == 1]
```

Out[62]=



Using `Exclusions` the integral is quickly calculated.

```
In[12]:= NIntegrate[Log[(1 - (x^2 + y^2))^2],
           {x, 0, 2}, {y, 0, 2}, Exclusions → x^2 + y^2 == 1] // Timing
Out[12]= {0.33295, 1.28132}
```

`NIntegrate` will reach convergence much more slowly if no singularity specification is given.

```
In[35]:= NIntegrate[Log[(1 - (x^2 + y^2))^2], {x, 0, 2}, {y, 0, 2}] // Timing

    NIntegrate::slwcon :
      Numerical integration converging too slowly; suspect one of the following: singularity, value
          of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

    NIntegrate::eincr :
      The global error of the strategy GlobalAdaptive has increased more than 2000 times. The global error
          is expected to decrease monotonically after a number of integrand evaluations. Suspect
          one of the following: the working precision is insufficient for the specified precision
          goal; the integrand is highly oscillatory or it is not a (piecewise) smooth function; or
          the true value of the integral is 0. Increasing the value of the GlobalAdaptive option
          MaxErrorIncreases might lead to a convergent numerical integration. NIntegrate obtained
          1.2814579311938816` and 0.0003423677128377028` for the integral and error estimates. ≫

Out[35]= {1.43478, 1.28146}
```

Here is an example of a case in which a singular curve can be specified with the variable ranges. If $x \in [0, 2]$ and $y \in [0, 2]$ this would not be possible—see the following example.

```
In[10]:= NIntegrate[Log[(1 - (x^2 + y^2))^2], {x, 0, 1}, {y, 0, √(1 - x^2), 1}]

Out[10]= -2.33614
```

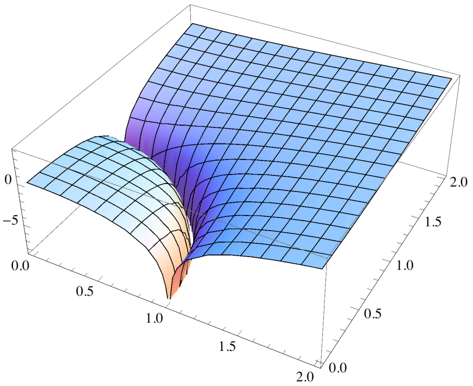
Example Implementation of Curve, Surface, and Hypersurface Singularity Handling

If the curve, surface, or hypersurface on which the singularities occur is known in implicit form (i.e., it can be described as a single equation) the function `Boole` can be used to form integration regions that have the singular curves, surfaces, or hypersurfaces as boundaries.

This two-dimensional function has singular points along the curve $x^2 + y^2 = 1$.

```
In[66]:= Plot3D[Log[(1 - (x^2 + y^2))^2], {x, 0, 2}, {y, 0, 2}, Exclusions → x^2 + y^2 == 1]
```

Out[66]=



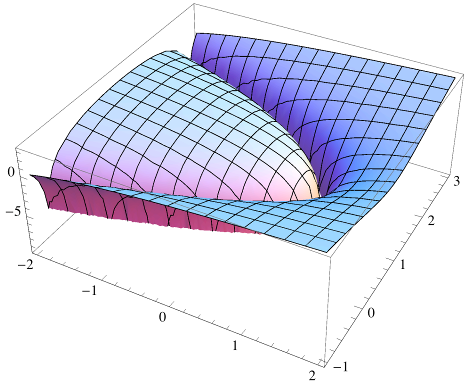
Using `Boole` the integral is calculated quickly.

```
In[9]:=  NIntegrate[Log[(1 - (x^2 + y^2))^2] * Boole[x^2 + y^2 < 1], {x, 0, 2}, {y, 0, 2}] +
           NIntegrate[Log[(1 - (x^2 + y^2))^2] * Boole[x^2 + y^2 > 1],
            {x, 0, 2}, {y, 0, 2}] // Timing
Out[9]= {0.295955, 1.28132}
```

This two-dimensional function has singular points along the curve $x + (1 - y)^2 = 1$.

```
In[68]:= Plot3D[Log[(1 - (x + (1 - y)^2))^2], {x, -2, 2}, {y, -1, 3}, Exclusions → x + (1 - y)^2 == 1]
```

Out[68]=



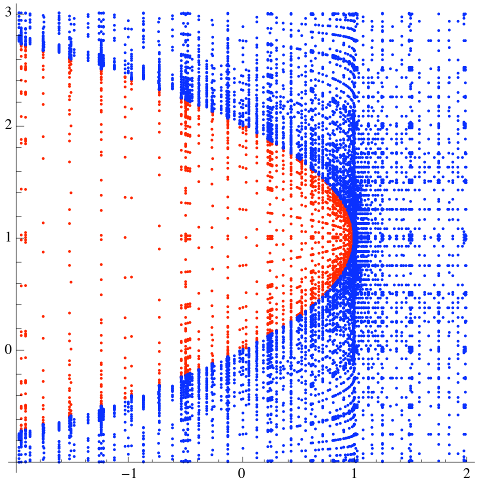
Again, using `Boole` the integral is calculated quickly.

```
In[8]:=  NIntegrate[Log[(1 - (x + (1 - y)^2))^2] * Boole[x + (1 - y)^2 < 1],
            {x, -2, 2}, {y, -1, 3}, PrecisionGoal → 4] +
           NIntegrate[Log[(1 - (x + (1 - y)^2))^2] * Boole[x + (1 - y)^2 > 1],
            {x, -2, 2}, {y, -1, 3}, PrecisionGoal → 4] // Timing
Out[8]= {0.432933, -2.22243}
```

This is how the sampling points of the integration look.

```
In[6]:= gr1 = {Red, Point /@ N@
            Reap[NIntegrate[Log[(1 - (x + (1 - y) ^ 2)) ^ 2] * Boole[x + (1 - y) ^ 2 < 1], {x, -2, 2},
               {y, -1, 3}, PrecisionGoal → 4, EvaluationMonitor ⧴ Sow[{x, y}]]][[2, 1]]};
        gr2 = {Blue, Point /@ N@Reap[NIntegrate[Log[(1 - (x + (1 - y) ^ 2)) ^ 2] *
                Boole[x + (1 - y) ^ 2 > 1], {x, -2, 2}, {y, -1, 3},
               PrecisionGoal → 4, EvaluationMonitor ⧴ Sow[{x, y}]]][[2, 1]]};
        Graphics[{PointSize[0.006], gr1, gr2}, Axes → True, AxesOrigin → {-2, -1}]
```

Out[7]=



Here is a function that takes a singular curve, surface, or hypersurface specification and uses the function `Boole` to make integration regions that have the singularities on their boundaries.

```
In[1]:= SingularManifoldNIntegrate[f_, ranges___, Equal[eq_, n_?NumericQ], opts___] :=
         NIntegrate[f * Boole[eq < n], ranges, opts] +
          NIntegrate[f * Boole[eq > n], ranges, opts]
```

This defines a three-dimensional function.

```
In[2]:= f[x_, y_, z_] := Log[((1 - (x + (1 - y) ^ 2 + (1 - z) ^ 2))) ^ 2];
```

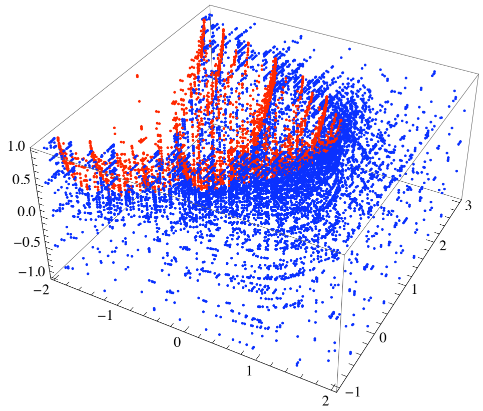
Here is the integral of a three-dimensional function with singular points along the surface $x + (1-y)^2 + (1-z)^2 = 1$.

```
In[3]:= SingularManifoldNIntegrate[f[x, y, z], {x, -2, 2}, {y, -1, 3},
         {z, -1, 1}, x + (1 - y) ^ 2 + (1 - z) ^ 2 == 1, PrecisionGoal → 3]

Out[3]= 21.7471 - 4.892636912996955 × 10^-339 ⅈ
```

These are the sampling points of the integration.

```
In[4]:= gr1 = {Red,
          Point[Re[#]] & /@ Reap[NIntegrate[f[x, y, z] * Boole[x + (1 - y) ^ 2 + (1 - z) ^ 2 < 1],
              {x, -2, 2}, {y, -1, 3}, {z, -1, 1}, PrecisionGoal → 3,
              EvaluationMonitor :> Sow[{x, y, z}]]][[2, 1]]};
        gr2 = {Blue, Point[Re[#]] & /@ Reap[NIntegrate[f[x, y, z] *
               Boole[x + (1 - y) ^ 2 + (1 - z) ^ 2 > 1], {x, -2, 2}, {y, -1, 3}, {z, -1, 1},
              PrecisionGoal → 3, EvaluationMonitor :> Sow[{x, y, z}]]][[2, 1]]};
        Graphics3D[{PointSize[0.006], gr1, gr2}, Axes -> True]
```

Out[5]=



"SingularityHandler" and "SingularityDepth"

Adaptive strategies improve the integral estimate by region bisection. If an adaptive strategy subregion is obtained by the number of bisections specified by the option "SingularityDepth", it is decided that subregion has a singularity. Then the integration over that subregion is done with the singularity handler specified by "SingularityHandler".

option name	*default value*	
"SingularityDepth"	Automatic	number of recursive bisections before applying a singularity handler
"SingularityHandler"	Automatic	singularity handler

"GlobalAdaptive" and "LocalAdaptive" singularity handling options.

If there is an integrable singularity at the boundary of a given region of integration, bisection could easily recur to MaxRecursion before convergence occurs. To deal with these situations the adaptive strategies of NIntegrate use variable transformations (IMT, "DoubleExponential", SidiSin) to speed up the integration convergence, or a region transformation (Duffy's coordinates) that relaxes the order of the singularity. The theoretical background of the variable transformation singularity handlers is given by the Euler-Maclaurin formula [DavRab84].

Use of the IMT Variable Transformation

The IMT variable transformation is the variable transformation in a quadrature method proposed by Iri, Moriguti, and Takasawa, called in the literature the IMT rule [DavRab84][IriMorTak70]. The IMT rule is based upon the idea of transforming the independent variable in such a way that all derivatives of the new integrand vanish at the end points of the integration interval. A trapezoidal rule is then applied to the new integrand, and under proper conditions high accuracy of the result might be attained [IriMorTak70][Mori74].

Here is a numerical integration that uses the IMT variable transformation for singularity handling.

```
In[13]:= NIntegrate[      1      , {x, 0, 1}, Method →
                    ───────────
                    Sqrt[1 - x]
           {"GlobalAdaptive", "SingularityHandler" → {IMT, "TuningParameters" → {10, 2}}}]

Out[13]= 2.
```

option name	default value	
"TuningParameters"	10	a pair of numbers $\{a, p\}$ that are the tuning parameters in the IMT transformation formula $a\, e^{1-\frac{1}{t^p}}$; if only a number a is given, it is interpreted as $\{a, 1\}$

IMT singularity handler option.

Adaptive strategies of NIntegrate employ only the transformation of the IMT rule. With the decision that a region might have a singularity, the IMT transformation is applied to its integrand. The integration continues, though not with a trapezoidal rule, but with the same integration rule used before the transformation. (Singularity handling with "DoubleExponential" switches to a trapezoidal integration rule.)

Also, adaptive strategies of `NIntegrate` use a variant of the original IMT transformation, with the transformed integrand vanishing only at one of the ends.

The IMT transformation $\varphi_{a,p}(t) : (0, 1] \to (0, 1]$, $a > 0$, $p > 0$, is defined.

```
In[14]:= φ[a_, p_, t_] := a Exp[1 - 1/t^p];
         φ[t_] := φ[1, 1, t]
```

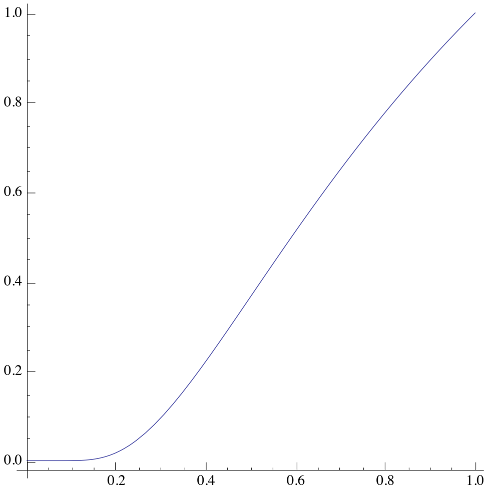
The parameters a and p are called tuning parameters [MurIri82].

The limit of the derivative of the IMT transformation is 0.

```
In[16]:= Limit[D[φ[a, p, t], t], t → 0, Assumptions → {a > 0, p > 0}]
Out[16]= 0
```

Here is the plot of the IMT transformation.

```
In[17]:= Plot[φ[t], {t, 0, 1}, AxesOrigin -> {0, -0.02},
          PlotRange → All, AspectRatio → Automatic]
Out[17]=
```

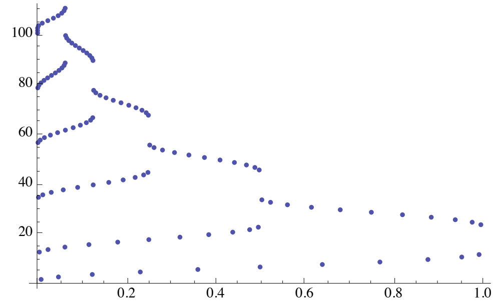


From the graph above follows that the transformed sampling points are much denser around 0. This means that if the integrand is singular at 0 it will be sampled more effectively, since a larger part of the integration rule sampling points will contribute large integrand values to the integration rule's integral estimate.

Since for any given working precision the numbers around 0 are much denser than the numbers around 1, after a region bisection the adaptive strategies of `NIntegrate` reverse the bisection variable of the subregion that has the right end of the bisected interval. This can be seen from the following plot.

```
In[18]:= pnts = Reap[NIntegrate[1/√x, {x, 0, 1}, Method →
              {"GlobalAdaptive", "SingularityHandler" → {IMT, "TuningParameters" → 1}},
             PrecisionGoal → 2, EvaluationMonitor ⧴ Sow[x]]]⟦2, 1⟧;
         ListPlot[Transpose[{pnts, Range[Length[pnts]]}]]
```

Out[19]=



No other singularity handler is applied to the subregions of a region to which the IMT variable transformation has been applied.

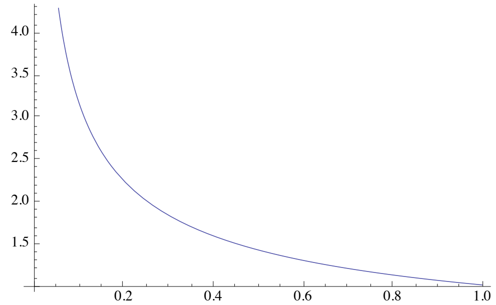
IMT Transformation by Example

Consider the function $\frac{1}{\sqrt{x}}$ over $(0, 1]$ that has a singularity at 0.

```
In[29]:= f[x_] := 1 / Sqrt[x]
```

```
In[30]:= Plot[f[x], {x, 0, 1}]
```

Out[30]=



Assume the integration is done with "`GlobalAdaptive`", with singularity handler IMT and singularity depth 4. After four bisections "`GlobalAdaptive`" will have a region with boundaries $\{0, 1/16\}$ that contains the singular end point. For that region the IMT variable transformation will change its boundaries to $\{0, 1\}$ and its integrand to the following.

```
In[31]:= {a, b} = {0, 1 / 16};
         f[Rescale[φ[t], {0, 1}, {a, b}]] D[Rescale[φ[t], {0, 1}, {a, b}], t]
```

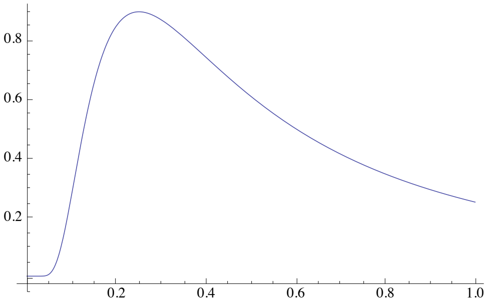
Out[32]=

$$\frac{\sqrt{e^{1-\frac{1}{t}}}}{4\,t^2}$$

Here is the plot of the new integrand.

```
In[33]:= {a, b} = {0, 1 / 16};
         Plot[f[Rescale[φ[t], {0, 1}, {a, b}]] D[Rescale[φ[t], {0, 1}, {a, b}], t] //
           Evaluate, {t, 0, 1}, AxesOrigin -> {0, -0.02}, PlotRange → All]
```

Out[34]=



The singularity is smashed!

Some of the sampling points, though, become too close to the singular end, and therefore special care should be taken for sampling points that coincide with the singular point because of the IMT transformation. `NIntegrate` ignores evaluations at singular points; see "Ignoring the Singularity".

For example, consider the sampling points and weights of the Gauss-Kronrod rule.

```
In[35]:= {absc, weights, errweight} = NIntegrate`GaussKronrodRuleData[5, MachinePrecision];
```

The Gauss-Kronrod sampling points for the region $\{0, 1/16\}$ and the derivatives of the rescaling follow.

```
In[36]:= abscGK = Rescale[#1, {0, 1}, {0, 1/16}] & /@ absc

Out[36]= {0.000497332, 0.00293188, 0.00768229, 0.0144228, 0.0225115,
          0.03125, 0.0399885, 0.0480772, 0.0548177, 0.0595681, 0.0620027}
```

```
In[37]:= derivativesGK = D[Rescale[t, {0, 1}, {0, 1/16}], t] /. t → # & /@ absc

Out[37]= {1/16, 1/16, 1/16, 1/16, 1/16, 1/16, 1/16, 1/16, 1/16, 1/16, 1/16}
```

Here is the integral estimate.

```
In[38]:= (f[abscGK] derivativesGK).weights

Out[38]= 0.484375
```

With the IMT transformation, these are the sampling points and derivatives.

```
In[39]:= abscGKIMT = Rescale[φ[#1], {0, 1}, {0, 1/16}] & /@
           NIntegrate`GaussKronrodRuleData[5, MachinePrecision][[1]]

Out[39]= {4.48942×10^-56, 9.37893×10^-11, 0.0000497657, 0.00222946, 0.0105784,
          0.0229925, 0.0355954, 0.0463014, 0.0543271, 0.0594983, 0.0620007}
```

```
In[40]:= derivativesGKIMT = D[Rescale[φ[t], {0, 1}, {0, 1/16}], t] /. t → # & /@
           NIntegrate`GaussKronrodRuleData[5, MachinePrecision][[1]]

Out[40]= {7.09017×10^-52, 4.26208×10^-8, 0.00329389, 0.0418657, 0.0815397,
          0.0919699, 0.0869529, 0.0782486, 0.0706212, 0.0654993, 0.0629993}
```

Here is the integral estimate with the IMT transformation.

```
In[41]:= (f[abscGKIMT] derivativesGKIMT).weights

Out[41]= 0.500562
```

The estimate calculated with the IMT variable transformation is much closer to the exact value.

```
In[42]:= Integrate[1/√x, {x, 0, 1/16}]

Out[42]= 1/2
```

Use of Double-Exponential Quadrature

When adaptive strategies use the IMT variable transformation they do not change the integration rule on the IMT-transformed regions. In contrast to this you can use both a variable transformation and a different integration rule on the regions considered to have singularity. (This is more in the spirit of the IMT rule [DavRab84].) This is exactly what happens when double-exponential quadrature is used—double-exponential quadrature uses the trapezoidal rule.

NIntegrate can use double-exponential quadrature for singularity handling only for one-dimensional integration.

Here is a numerical integration that uses double-exponential quadrature for singularity handling.

```
                          1
In[103]:= NIntegrate[―――――――――――, {x, 0, 1},
                      Sqrt[1 - x]
           Method → {"GlobalAdaptive", "SingularityHandler" → "DoubleExponential"}]
Out[103]= 2.
```

IMT versus "DoubleExponential" versus No Singularity Handling for One-Dimensional Integrals

Both singularity handlers (IMT and "DoubleExponential") are applied to regions that are obtained through too many bisections (as specified by "SingularityDepth").

The main difference between them is that IMT does not change the integration rule used to compute integral estimates on the region it is applied to—IMT is only a variable transformation. On the other hand, "DoubleExponential" uses both variable transformation and a different integration rule—the trapezoidal rule—to compute integral estimates on the region it is applied to. In other words, the singularity handler "DoubleExponential" delegates the integration to the double-exponential quadrature as described in Double-Exponential Strategy.

As a consequence, a region to which the IMT singularity handler is applied is still going to be subject to bisection by the adaptive integration strategy. Therefore, until the precision goal is reached the integrand evaluations done before the last bisection will be thrown away. On the other hand, a region to which the "DoubleExponential" singularity handler is applied will not be bisected. The trapezoidal rule quadrature used by "DoubleExponential" will compute integral estimates over the region with an increasing number of sampling points at each step, completely reusing the integrand evaluations of the sampling points from the previous steps.

So, if the integrand is "very" analytic (i.e., no rapid or sudden changes of the integrand and its derivatives wrt the integration variable) over the regions with end point singularity, the "DoubleExponential" singularity handler is going to be much faster than the IMT singularity handler. In the cases where the integrand is not analytic in the region given to the `"DoubleExponential"` singularity handler, or the double transformation of the integrand converges too slowly, it is better to switch to the `IMT` singularity handler. This is done if the option `"SingularityHandler"` is set to `Automatic`.

Following are tables that compare the `IMT`, `"DoubleExponential"`, and `Automatic` singularity handlers applied at different depths of bisection.

This loads a package that defines the profiling function `NIntegrateProfile` that gives the number of sampling points and the time needed by a numerical integration command.

```
In[17]:= Needs["Integration`NIntegrateUtilities`"];
```

Table for a "very" analytical integrand $\frac{1}{\sqrt{x}}$ that the `"DoubleExponential"` singularity handler easily computes.

```
In[34]:= exact = 2;
         tbl = ((t = {"IntegralEstimate", "Evaluations", "Timing"} /.
                      NIntegrateProfile[NIntegrate[1/Sqrt[x], {x, 0, 1}, Method → {"GlobalAdaptive",
                           "SingularityHandler" → #1[[1]], "SingularityDepth" → #1[[2]],
                           "SymbolicProcessing" → 0}, MaxRecursion → 100]];
                    {#1[[2]], Abs[t[[1, 1]] - exact], t[[2]], t[[3]]}) &) /@
              {{"IMT", Infinity}, {"IMT", 1}, {"DoubleExponential", 1},
               {"IMT", 4}, {"DoubleExponential", 4},
               {Automatic, 4}};
         TableForm[tbl, TableHeadings → Map[Style[#, FontFamily → Times, FontSize → 11] &,
              {{"No singularity handling", "IMT", "DoubleExponential",
                "IMT", "DoubleEponential", "Automatic"},
               {"SingularityDepth", ColumnForm[{"Difference from", "the exact integral"}],
                ColumnForm[{"Number of function", "evaluations"}], "Time (s)"}}, {-1}]]
```

```
                                              SingularityDepth   Difference from      Number of function   Time (s)
                                                                 the exact integral   evaluations
                        No singularity handling  ∞                  9.53644×10^-7        715                 0.0044994
                        IMT                      1                  1.06581×10^-14       88                  0.0025996
Out[36]//TableForm=     DoubleExponential        1                  3.10862×10^-15       65                  0.0020997
                        IMT                      4                  6.21725×10^-15       154                 0.0028996
                        DoubleEponential         4                  3.10862×10^-15       132                 0.0024996
                        Automatic                4                  3.10862×10^-15       132                 0.0022996
```

Table for an integrand, $\frac{70}{10^4\left(x-\frac{1}{32}\right)^2+\frac{1}{16}}$, that does not have a singularity and has a nearly discontinuous derivative (i.e., it is not "very" analytical). The `Automatic` singularity handler starts with `"DoubleExponential"` and then switches to `IMT`.

```
In[37]:= f[x_] := 70/(10^4 (x - 1/32)^2 + 1/16);
         exact = Integrate[f[x], {x, 0, 1}];
         tbl = ((t = {"IntegralEstimate", "Evaluations", "Timing"} /.
                  NIntegrateProfile[NIntegrate[f[x], {x, 0, 1}, Method → {"GlobalAdaptive",
                      "SingularityHandler" → #1[[1]], "SingularityDepth" → #1[[2]],
                      "SymbolicProcessing" → 0}, MaxRecursion → 100, PrecisionGoal → 8]];
                {#1[[2]], Abs[t[[1, 1]] - exact], t[[2]], t[[3]]}) &) /@
              {{"IMT", Infinity}, {"IMT", 1}, {"DoubleExponential", 1},
               {"IMT", 4}, {"DoubleExponential", 4},
               {Automatic, 4}};
         TableForm[tbl, TableHeadings → Map[Style[#, FontFamily → Times, FontSize → 11] &,
              {{"No singularity handling", "IMT", "DoubleExponential",
                "IMT", "DoubleEponential", "Automatic"},
               {"SingularityDepth", ColumnForm[{"Difference", "from the exact integral"}],
                ColumnForm[{"Number of function", "evaluations"}], "Time (s)"}}, {-1}]]
```

```
                                       | SingularityDepth  Difference               Number of function  Time (s)
                                       |                   from the exact integral  evaluations
                -----------------------+---------------------------------------------------------------------------
                No singularity handling| ∞                 1.95399×10^-14           495                 0.0038994
                IMT                    | 1                 1.42109×10^-14           528                 0.006699
Out[40]//TableForm= DoubleExponential  | 1                 7.10543×10^-15           3240                0.0436934
                IMT                    | 4                 2.4869×10^-14            594                 0.006899
                DoubleEponential       | 4                 7.10543×10^-15           950                 0.012998
                Automatic              | 4                 1.77636×10^-14           552                 0.0069989
```

A table for an integrand, $\frac{x+\frac{1}{-1+\mathrm{Log}[x]}}{x\,\mathrm{Log}[x]}$, for which the `Automatic` singularity handler starts with `"DoubleExponential"` and then switches to `IMT`.

```
In[41]:= f[x_] := (x + 1/(-1 + Log[x]))/(x Log[x]);
         exact = Integrate[f[x], {x, 0, 1}];
         tbl = ((t = {"IntegralEstimate", "Evaluations", "Timing"} /.
                  NIntegrateProfile[NIntegrate[f[x], {x, 0, 1}, Method → {"GlobalAdaptive",
                      "SingularityHandler" → #1[[1]], "SingularityDepth" → #1[[2]],
                      "SymbolicProcessing" → 0}, MaxRecursion → 3000, PrecisionGoal → 6]];
                {#1[[2]], Abs[t[[1, 1]] - exact], t[[2]], t[[3]]}) &) /@
              {{"IMT", Infinity}, {"IMT", 1}, {"DoubleExponential", 1},
               {"IMT", 4}, {"DoubleExponential", 4},
               {Automatic, 4}};
         TableForm[tbl, TableHeadings → Map[Style[#, FontFamily → Times, FontSize → 11] &,
              {{"No singularity handling", "IMT", "DoubleExponential",
                "IMT", "DoubleEponential", "Automatic"},
               {"SingularityDepth", ColumnForm[{"Difference from", "the exact integral"}],
                ColumnForm[{"Number of function", "evaluations"}], "Time (s)"}}, {-1}]]
```

```
                                       | SingularityDepth  Difference from     Number of function  Time (s)
                                       |                   the exact integral  evaluations
                -----------------------+----------------------------------------------------------------------
                No singularity handling| ∞                 0.000555531         56 925              2.26286
                IMT                    | 1                 4.58522×10^-14      88                  0.0027996
Out[44]//TableForm= DoubleExponential  | 1                 7.00532×10^-10      131                 0.012998
                IMT                    | 4                 7.88258×10^-15      132                 0.0028996
                DoubleEponential       | 4                 7.00528×10^-10      197                 0.0165974
                Automatic              | 4                 1.95931×10^-10      182                 0.0044993
```

IMT Multidimensional Singularity Handling

When used for multidimensional integrals, the IMT singularity handler speeds up the integration process only when the singularity is along one of the axes. When the singularity is at a corner of the integration region, using IMT is counterproductive. The function `NIntegrateProfile` defined earlier is used in the following examples.

The number of integrand evaluations and timings for an integrand that has a singularity only along the *x* axis. The default (automatic) singularity handler chooses to apply IMT to regions obtained after the default (four) bisections.

```
In[19]:= NIntegrateProfile@NIntegrate[1/Sqrt[x] + y, {x, 0, 1}, {y, 0, 1}]

Out[19]= {IntegralEstimate → 2.500000004270092, Evaluations → 442, Timing → 0.025}
```

The number of integrand evaluations and timings for an integrand that has a singularity only along the *x* axis with no singularity handler application.

```
In[20]:= NIntegrateProfile@NIntegrate[1/Sqrt[x] + y, {x, 0, 1}, {y, 0, 1},
           Method → {"GlobalAdaptive", "SingularityHandler" → None}, MaxRecursion → 30]

Out[20]= {IntegralEstimate → 2.4999994380778543, Evaluations → 1445, Timing → 0.0231}
```

The number of integrand evaluations and timings for an integrand that has a singularity at a corner of the integration region. The default (automatic) singularity handler chooses to apply the singularity handler `DuffyCoordinates` to regions obtained after the default (four) bisections.

```
In[21]:= NIntegrateProfile@NIntegrate[1/Sqrt[x^2 + y^2], {x, 0, 1}, {y, 0, 1}]

Out[21]= {IntegralEstimate → 1.7627471522176814, Evaluations → 2006, Timing → 0.038}
```

The number of integrand evaluations and timings for an integrand that has a singularity at a corner of the integration region. IMT is applied to regions obtained after the default (four) bisections.

```
In[22]:= NIntegrateProfile@NIntegrate[1/Sqrt[x^2 + y^2], {x, 0, 1}, {y, 0, 1},
           Method → {"GlobalAdaptive", "SingularityHandler" → "IMT"}, MaxRecursion → 30]

         NIntegrate::slwcon :
           Numerical integration converging too slowly; suspect one of the following: singularity, value
             of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫
```

```
NIntegrate::slwcon :
  Numerical integration converging too slowly; suspect one of the following: singularity, value
     of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

NIntegrate::slwcon :
  Numerical integration converging too slowly; suspect one of the following: singularity, value
     of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

General::stop : Further output of NIntegrate::slwcon will be suppressed during this calculation. ≫
```

```
Out[22]= {IntegralEstimate → 1.762747132592934, Evaluations → 7004, Timing → 0.0941}
```

The number of integrand evaluations and timings for an integrand that has a singularity at a corner of the integration region with no singularity handler application.

```
In[23]:= NIntegrateProfile@NIntegrate[1/Sqrt[x^2 + y^2], {x, 0, 1}, {y, 0, 1},
           Method → {"GlobalAdaptive", "SingularityHandler" → None}, MaxRecursion → 30]

Out[23]= {IntegralEstimate → 1.7627469943973395, Evaluations → 3791, Timing → 0.0451}
```

Duffy's Coordinates for Multidimensional Singularity Handling

Duffy's coordinates is a technique that transforms an integrand over a square, cube, or hypercube with a singular point in one of the corners into an integrand with a singularity over a line, which might be easier to integrate.

The following integration uses Duffy's coordinates.

```
In[63]:= NIntegrate[1/√(x^2 + y^2), {y, 0, 1}, {x, 0, 1},
           Method → {"GlobalAdaptive", "SingularityHandler" → "DuffyCoordinates"}] // Timing

Out[63]= {0.017997, 1.76275}
```

The following integration does not use Duffy's coordinates.

```
In[62]:= NIntegrate[1/√(x^2 + y^2), {y, 0, 1}, {x, 0, 1},
           Method → {"GlobalAdaptive", "SingularityHandler" -> None},
           MaxRecursion -> 20] // Timing

Out[62]= {0.038994, 1.76275}
```

The `NIntegrate` strategies `"GlobalAdaptive"` and `"LocalAdaptive"` apply the Duffy's coordinates technique only at the corners of the integration region.

When the singularity of a multidimensional integral occurs at a point, the coupling of the variables will make the singularity variable transformations used in one-dimensional integration counterproductive. A variable transformation that has a geometrical nature, proposed by Duffy in [Duffy82], makes a change of variables that replaces a point singularity at a corner of the integration region with a "softer" one on a plane.

If d is the dimension of integration and $r = x_1{}^2 + x_2{}^2 + \ldots + x_d{}^2$, then Duffy's coordinates is a suitable technique for singularities of the following type (see again [Duffy82]):

1. r^α, $r^\alpha \ln r$, $\alpha > -d$;
2. $x_1{}^{\alpha_1}\, x_2{}^{\alpha_2} \ldots \mathrm{x}_d{}^{\alpha_d}\, r^\beta$, $\alpha_i > -1,\ i \in [1, d]$, $\sum \alpha_i + \beta > -d$;
3. $\left(c_1\, x_1{}^\beta + c_2\, x_2{}^\beta + \ldots + c_d\, x_d{}^\beta\right)^\alpha$, $\beta > 0$, $\alpha\, \beta > -d$, $c_i > 0,\ i \in [1, d]$.

For example, consider the integral

$$\int_0^1 \int_0^x \frac{1}{\sqrt{4\,x^2 + y^2}}\, dx\, dy.$$

If the integration region $(0, 1] \times (0, x]$ is changed to $(0, 1] \times (0, 1]$ with the rule $y \to x\, y$, the Jacobian of which is x, the integral becomes

$$\int_0^1 \int_0^x \frac{1}{\sqrt{4\,x^2 + y^2}}\, dx\, dy \Leftrightarrow \int_0^1 \int_0^1 \frac{x}{\sqrt{4\,x^2 + (x\,y)^2}}\, dx\, dy \Leftrightarrow \int_0^1 \int_0^1 \frac{1}{\sqrt{y^2 + 4}}\, dx\, dy. \tag{5}$$

The last integral has no singularities at all!

Now consider the integral

$$\int_0^1 \int_0^1 \frac{1}{\sqrt{4\,x^2 + y^2}}\, dx\, dy, \tag{6}$$

which is equivalent to the sum

$$\int_0^1 \int_0^x \frac{1}{\sqrt{4\,x^2 + y^2}}\, dx\, dy + \int_0^1 \int_x^1 \frac{1}{\sqrt{4\,x^2 + y^2}}\, dx\, dy.$$

The first integral of that sum is transformed as in (5); for the second one, though, the change of $(0, 1] \times (1, x]$ into $(0, 1] \times (0, 1]$ by $y \to x + (1 - x)\, y$ has the Jacobian $1 - x$, which will not bring the desired cancellation of terms. Fortunately, a change of the order of integration:

$$\int_0^1\int_x^1 \frac{1}{\sqrt{4\,x^2+y^2}}\,dx\,dy \Leftrightarrow \int_0^1\int_0^y \frac{1}{\sqrt{4\,x^2+y^2}}\,dy\,dx,$$

makes the second integral amenable for the transformation in (5):

$$\int_0^1\int_0^y \frac{1}{\sqrt{4\,x^2+y^2}}\,dy\,dx \Leftrightarrow \int_0^1\int_0^x \frac{1}{\sqrt{4\,y^2+x^2}}\,dx\,dy \Leftrightarrow \int_0^1\int_0^1 \frac{x}{\sqrt{4\,(x\,y)^2+x^2}}\,dx\,dy \Leftrightarrow \int_0^1\int_0^1 \frac{1}{\sqrt{1+4\,y^2}}\,dx\,dy. \qquad (7)$$

(In the second integral in the equation (3) the variables were permuted, which is not necessary to prove the mathematical equivalence, but it is faster when computing the integrals.)

So the integral (6) can be rewritten as an integral with no singularities:

$$\int_0^1\int_0^1 \frac{1}{\sqrt{4\,x^2+y^2}}\,dx\,dy \Leftrightarrow \int_0^1\int_0^1 \frac{1}{\sqrt{y^2+4}} + \frac{1}{\sqrt{1+4\,y^2}}\,dx\,dy.$$

If the integration variables were not permuted in (7), the integral (6) is going to be rewritten as

$$\int_0^1\int_0^1 \frac{1}{\sqrt{4\,x^2+y^2}}\,dx\,dy \Leftrightarrow \int_0^1\int_0^1 \frac{1}{\sqrt{y^2+4}} + \frac{1}{\sqrt{1+4\,x^2}}\,dx\,dy.$$

That is a more complicated integral, as its integrand is not simple along both axes. Subsequently it is harder to compute than the former one.

Here is the number of sampling points for the simpler integral.

```
                                1             1
In[58]:= Reap[NIntegrate[ ----------- + ------------ , {x, 0, 1}, {y, 0, 1},
                           √ y² + 4      √ 1 + 4 y²
             PrecisionGoal → 8, EvaluationMonitor :> Sow[{x, y}]]][[2, 1]] // Length

Out[58]= 187
```

Here is the number of sampling points for the more complicated integral.

```
                                1             1
In[59]:= Reap[NIntegrate[ ----------- + ------------ , {x, 0, 1}, {y, 0, 1},
                           √ y² + 4      √ 1 + 4 x²
             PrecisionGoal → 8, EvaluationMonitor :> Sow[{x, y}]]][[2, 1]] // Length

Out[59]= 323
```

NIntegrate uses a generalization to arbitrary dimension of the technique in the example above. (In [Duffy82] third dimension is described only.) An example implementation together with the generalization description are given below.

Here is a table that compares the different singularity handlings for $\int_0^1\int_0^1 \frac{1}{\sqrt{x^2+y^2}}\,dx\,dy$. (The profiling function NIntegrateProfile defined earlier is used.)

In[74]:=

```
exact = Integrate[1 / Sqrt[x^2 + y^2], {x, 0, 1}, {y, 0, 1}];
tbl = ((t = {"IntegralEstimate", "Evaluations", "Timing"} /. NIntegrateProfile[
          NIntegrate[1 / Sqrt[x^2 + y^2], {x, 0, 1}, {y, 0, 1}, Method →
            {"GlobalAdaptive", "SingularityHandler" → #1[[1]], "SingularityDepth" →
              #1[[2]], "SymbolicProcessing" → 0}, MaxRecursion → 12]];
       {#1[[2]], Abs[t[[1, 1]] - exact], t[[2]], t[[3]]}) &) /@
   {{None, Infinity}, {"IMT", 1}, {"IMT", 4},
    {"DuffyCoordinates", 4},
    {"DuffyCoordinates", 1}};
TableForm[tbl, TableHeadings → Map[Style[#, FontFamily → Times, FontSize → 11] &,
    {{"No singularity handling", "IMT", "IMT",
      "DuffyCoordinates", "DuffyCoordinates"}, {"SingularityDepth",
      ColumnForm[{"Difference", "from the", "nexact integral"}],
      ColumnForm[{"Number of", "function", "evaluations"}], "Time (s)"}}, {-1}]]
```

Out[75]=

	SingularityDepth	Difference from the nexact integral	Number of function evaluations	Time (s)
No singularity handling	∞	1.79642×10^{-7}	3791	0.0338948
IMT	1	1.48099×10^{-8}	10557	0.145778
IMT	4	4.14462×10^{-8}	7004	0.0960854
DuffyCoordinates	4	6.50582×10^{-8}	1394	0.0150977
DuffyCoordinates	1	1.09823×10^{-8}	629	0.0095986

Duffy's Coordinates Strategy

When Duffy's coordinates are applicable, a numerical integration result is obtained faster if Duffy's coordinate change is made before the actual integration begins. Making the transformation beforehand, though, requires knowledge at which corner(s) of the integration region the singularities occur. The "DuffyCoordinates" strategy in NIntegrate facilitates such pre-integration transformation.

Here is an example with an integrand that has singularities at two different corners of its integration region.

In[84]:= $$\mathtt{NIntegrate}\left[\frac{1}{\sqrt{x^2+y^2}}+\frac{1}{\sqrt{x+(1-y)}},\ \{x,0,1\},\ \{y,0,1\},\right.$$

```
Method → {"DuffyCoordinates", "Corners" → {{0, 0}, {0, 1}}}]
```

Out[84]= 2.86732

option name	*default value*	

`Method`	`{"GlobalAdaptive",`
	`"SingularityDepth"->∞}`
		the strategy with which the integration will be made after applying Duffy's coordinates transformation

`"Corners"`	`All`	a vector or a list of vectors that specify the corner(s) to apply the Duffy's coordinates tranformation to; the elements of the vectors are either 0 or 1; each vector length equals the dimension of the integral

`"DuffyCoordinates"` options.

The first thing `"DuffyCoordinates"` does is to rescale the integral into one that is over the unit hypercube (or square, or cube). If only one corner is specified `"DuffyCoordinates"` applies Duffy's coordinates transformation as described earlier. If more than one corner is specified, the unit hypercube of the previous step is partitioned into disjoint cubes with side length of one-half. Consider the integrals over these disjoint cubes. Duffy's coordinates transformation is applied to the ones that have a vertex that is specified to be singular. The rest are transformed into integrals over the unit cube. Since all integrals at this point have an integration region that is the unit cube, they are summated, and that sum is given to `NIntegrate` with a `Method` option that is the same as the one given to `"DuffyCoordinates"`.

The actual integrand used by `"DuffyCoordinates"` can be obtained through ``NIntegrate`DuffyCoordinatesIntegrand``, which has the same arguments as `NIntegrate`.

Here is an example for the "`DuffyCoordinates`" integrand of a three-dimensional function that is singular at one of the corners of the integration region.

In[78]:=
$$\texttt{NIntegrate`DuffyCoordinatesIntegrand}\left[\frac{1}{\sqrt{x^3+(1-y)^3+z^3}}, \{x,0,1\}, \{y,0,1\}, \{z,0,1\}, \texttt{Method}\to\{\text{"DuffyCoordinates"}, \text{"Corners"}\to\{0,1,0\}\}\right] // \texttt{Simplify}[\#, \texttt{Assumptions}\to\{0\le x\le 1, 0\le y\le 1, 0\le z\le 1\}]\ \&$$

Out[78]=
$$3\sqrt{\frac{x}{1+y^3+z^3}}$$

Here is an example for the "`DuffyCoordinates`" integrand for a two-dimensional function that is singular at two of the corners of the integration region.

In[79]:=
$$\texttt{NIntegrate`DuffyCoordinatesIntegrand}\left[\frac{1}{\sqrt{x^2+y^2}}\ \frac{1}{\sqrt{x^2+(1-y)^2}}, \{x,0,1\}, \{y,0,1\}, \texttt{Method}\to\{\text{"DuffyCoordinates"}, \text{"Corners"}\to\{\{0,0\},\{0,1\}\}\}\right] // \texttt{Simplify}[\#, \texttt{Assumptions}\to\{0\le x\le 1, 0\le y\le 1\}]\ \&$$

Out[79]=
$$\frac{1}{\sqrt{\left(1+2x+x^2+y^2\right)\left(5+2x+x^2-4y+y^2\right)}}+\frac{1}{\sqrt{\left(2+2x+x^2-2y+y^2\right)\left(2+2x+x^2+2y+y^2\right)}}+\frac{2}{\sqrt{\left(1+y^2\right)\left(4-4x+x^2\left(1+y^2\right)\right)}}+\frac{2}{\sqrt{\left(1+y^2\right)\left(4-4xy+x^2\left(1+y^2\right)\right)}}$$

"`DuffyCoordinates`" might considerably improve speed for the types of integrands described in "Duffy's Coordinates for Multidimensional Singularity Handling".

Integration with "`DuffyCoordinates`".

In[80]:=
$$\texttt{NIntegrate}\left[\frac{1}{\sqrt{x^2+y^2+z^2}}+\frac{1}{\sqrt{x^2+y^2+(1-z)^2}}, \{x,0,1\}, \{y,0,1\}, \{z,0,1\}, \texttt{Method}\to\{\text{"DuffyCoordinates"}, \text{"Corners"}\to\{\{0,0,0\},\{0,0,1\}\}\}\right] // \texttt{Timing}$$

Out[80]=
```
{0.022997, 2.38008}
```

Integration with the default `NIntegrate` options settings which is much slower than the previous one.

In[81]:=
$$\texttt{NIntegrate}\left[\frac{1}{\sqrt{x^2+y^2+z^2}}+\frac{1}{\sqrt{x^2+y^2+(1-z)^2}}, \{x,0,1\}, \{y,0,1\}, \{z,0,1\}\right] // \texttt{Timing}$$

Out[81]=
```
{0.25296, 2.38008}
```

Here is another example of a speedup by "DuffyCoordinates".

```
                             1
In[82]:=  NIntegrate[――――――――――――――――――, {x, 0, 1}, {y, 0, 1},
                      √ x + Sin[1 - y]
            Method → {"DuffyCoordinates", "Corners" → {0, 1}}] // Timing

Out[82]=  {0.010999, 1.12142}
```

Integration with the default NIntegrate options settings which is much slower than the previous one.

```
                             1
In[83]:=  NIntegrate[――――――――――――――――――, {x, 0, 1}, {y, 0, 1}] // Timing
                      √ x + Sin[1 - y]

Out[83]=  {0.035994, 1.12142}
```

Duffy's Coordinates Generalization and Example Implementation

See "Duffy's Coordinates for Multidimensional Singularity Handling" for the theory of Duffy's coordinates.

The implementation is based on the following two theorems.

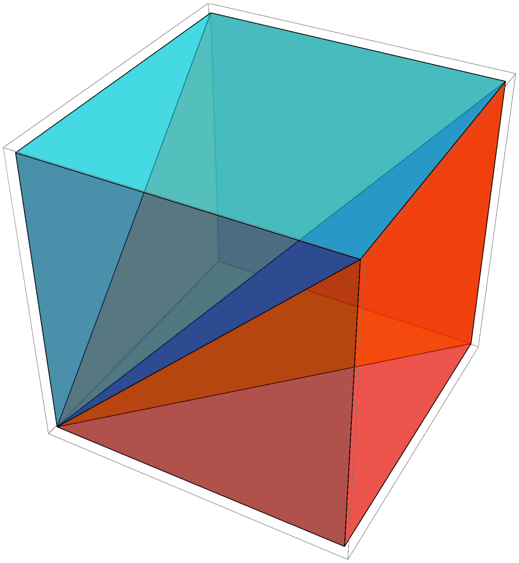
Theorem 1: A d-dimensional cube can be divided into d disjoint geometrically equivalent d-dimensional pyramids (with bases $(d-1)$-dimensional cubes) and with coinciding apexes.

Proof: Assume the side length of the cube is 1, the cube has a vertex at the origin, and the coordinates of all other vertexes are 1 or 0. Consider the $(d-1)$-dimensional cube walls $w_s = \{c_1, \ldots, c_{s-1}, 1, c_{s+1}, \ldots, c_d\}$, where $c_i \in [0, 1]$. Their number is exactly d, and the origin does not belong to them. Each of the w_s walls can form a pyramid with the origin. This proves the theorem.

Here is a plot that illustrates the theorem in 3D.

```
In[89]:= grx = GraphicsComplex[{{0, 0, 0}, {1, 0, 0}, {1, 0, 1}, {1, 1, 1}, {1, 1, 0}},
            {Polygon[{1, 2, 3}], Polygon[{1, 3, 4}], Polygon[{1, 4, 5}],
             Polygon[{1, 5, 2}], Polygon[{2, 3, 4, 5}]}];
         gry = MapAt[Map[RotateLeft[#] &, #] &, grx, {1}];
         grz = MapAt[Map[RotateRight[#] &, #] &, grx, {1}];
         Graphics3D[{Opacity[0.5], Red, grx, Cyan, gry, Yellow, grz}]
```

Out[92]=



If the d axes are denoted $x_1, x_2, \ldots, x_d$ the pyramid formed with the wall $w_1 = \{1, c_2, \ldots, c_d\}$ can be described as $0 \le x_1 \le 1,\ 0 \le x_i \le x_1,\ i \in \{2, \ldots, d\}$. Let σ^i denote the permutation derived after rotating $\{1, \ldots, d\}$ cyclically i times to the left (i.e., applying i times `RotateLeft` to $\{1, \ldots, d\}$). Then the following theorem holds:

Theorem 2: For any integral over the unit cube the following equalities hold:

$$\int_0^1 \int_0^1 \ldots \int_0^1 f(x_1, \ldots, x_d)\, dx_1 \ldots dx_d = \int_0^1 \int_0^{x_1} \ldots \int_0^{x_1} \sum_{i=0}^{d-1} f\left(x_{\sigma^i{}_1}, \ldots, x_{\sigma^i{}_d}\right) dx_{\sigma^i{}_1} \ldots dx_{\sigma^i{}_d} =$$

$$\int_0^1 \int_0^1 \ldots \int_0^1 x_1{}^{d-1} \sum_{i=0}^{d} f\left(x_1\, x_{\sigma^i{}_1}, \ldots, x_{\sigma^i{}_{i+1}}, \ldots, x_1\, x_{\sigma^i{}_d}\right) dx_{\sigma^i{}_1} \ldots dx_{\sigma^i{}_d}.$$

Proof: The first equality follows from Theorem 1. The second equality is just a change of variables that transforms a pyramid to a cube.

Here is a function that gives the rules and the Jacobian for the transformation of a hypercube with a specified side into a region.

```
In[93]:= FRangesToCube[ranges_, cubeSides : {{_, _} ...}] :=
           Module[{t, t1, jac, vars, rules = {}},
             vars = First /@ ranges;
             t = MapThread[(t1 = Rescale[#1[[1]], #2, {#1[[2]], #1[[3]]} /. rules];
                  AppendTo[rules, #1[[1]] → t1]; t1) &, {ranges, cubeSides}];
             jac = Times @@ MapThread[D[#1, #2] &, {t, vars}];
             {rules, jac}
            ] /; Length[ranges] == Length[cubeSides];
         FRangesToCube[ranges_, cubeSide : {_, _}] :=
           FRangesToCube[ranges, Table[cubeSide, {Length[ranges]}]];
         FRangesToCube[ranges_] := FRangesToCube[ranges, {0, 1}];
```

Here is an example of unit-square to infinite region rescaling.

```
In[96]:= FRangesToCube[{{x, 0, 8}, {y, x, ∞}}]
```

```
                                   1            8
Out[96]= {{x → 8 x, y → -1 + 8 x + ─────}, ──────────}
                                 1 - y     (1 - y)^2
```

Here is a function that computes the integrals obtained by the Duffy's coordinates technique when the singularity is at the origin.

```
In[97]:= DuffyCoordinatesAtOrigin[F_, ranges___] :=
           DuffyCoordinatesBounds[F, First /@ {ranges}, Transpose[Rest /@ {ranges}]];
         DuffyCoordinatesBounds[F_, vars_, bounds_] :=
           Module[{rules, jac, newF, rots, res, range},
            {rules, jac} = FRangesToCube[Transpose[Prepend[bounds, vars]]];
            newF = (F /. rules) * jac;
            rots = NestList[RotateLeft[#1] &, vars, Length[vars] - 1];
            res = Prepend[Map[newF /. Thread[vars -> #1] &, Rest[rots]], newF];
            range = Join[{{vars[[1]], 0, 1}}, Map[{#, 0, vars[[1]]} &, Rest[vars]]];
            {rules, jac} = FRangesToCube[range];
            {(Total[res] /. rules) * jac, Sequence @@ ( {#1, 0, 1} &  /@ vars)}
           ];
```

Here is a function that computes the integrals obtained by the Duffy's coordinates technique for a specified corner of the hypercube where the singularity occurs.

```
In[99]:= DuffyCoordinates[F_, ranges___] :=
           DuffyCoordinates[F, ranges, Table[0, {Length[{ranges}]}]];
         DuffyCoordinates[F_, rangesSeq__, corner_? (VectorQ[#1, IntegerQ] &)] :=
           Module[{factor, ranges = {rangesSeq}, newrange, t},
            factor = 1;
            newrange = {};
            MapIndexed[(
               t = ranges⟦#2⟦1⟧⟧;
               If[#1 == 0,
                newrange = Append[newrange, t],
                newrange = Append[newrange, {t⟦1⟧, t⟦3⟧, t⟦2⟧}]; factor = -factor]) &,
             corner];
            DuffyCoordinatesAtOrigin[factor * F, Sequence @@ newrange]
           ];
```

Here is a symbolic example.

```
In[101]:= DuffyCoordinates[F[x, y], {x, 0, 1}, {y, 0, 1}]

Out[101]= {x (F[x, x y] + F[x y, x]), {x, 0, 1}, {y, 0, 1}}
```

Here is another symbolic example.

```
In[102]:=  DuffyCoordinates[F[x, y, z], {x, 0, 1}, {y, 0, 1}, {z, 0, 1}]

Out[102]=  {x^2 (F[x, x y, x z] + F[x y, x z, x] + F[x z, x, x y]), {x, 0, 1}, {y, 0, 1}, {z, 0, 1}}
```

Here is a computational example.

```
                                             1
In[103]:=  NIntegrate @@ DuffyCoordinates[ ─────────────── , {x, 0, 4}, {y, 0, 3}, {z, 0, 2}]
                                          √(x² + y² + z²)

Out[103]=  9.52813
```

Using Duffy's coordinates is much faster than using no singularity handling (see the next example).

```
In[108]:=  res = NINT @@
                                    1
              DuffyCoordinates[ ───────────────────── , {x, 0, 4}, {y, 0, 3}, {z, 0, 2}, {0, 1, 0}];
                                √(x² + (3 - y)² + z²)
           res = Hold[Evaluate[res]] /. NINT → NIntegrate;
           Timing @@ res
Out[110]=  {0.009998, 9.52813}
```

Integration using no singularity handling.

```
                                   1
In[111]:=  Timing @NIntegrate[ ─────────────── , {x, 0, 4}, {y, 0, 3},
                               √(x² + y² + z²)

             {z, 0, 2}, Method → {"GlobalAdaptive", "SingularityHandler" → None}]

Out[111]=  {0.180971, 9.52813}
```

Of course, the internal implementation of `NIntegrate` gives similar performance results.

```
                                     1
In[107]:=  Timing @NIntegrate[ ───────────────────── , {x, 0, 4}, {y, 0, 3}, {z, 0, 2},
                               √(x² + (3 - y)² + z²)

             Method → {"DuffyCoordinates", "Corners" → {0, 1, 0}, "SymbolicProcessing" → 0}]

Out[107]=  {0.011998, 9.52813}
```

Ignoring the Singularity

Another way of handling a singularity is to ignore it. `NIntegrate` ignores sampling points that coincide with a singular point.

Consider the following integral that has a singular point at 1.

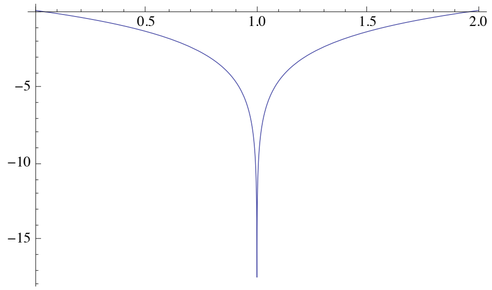
$$\int_0^2 \log\left((1-x)^2\right) \, dx.$$

The integrand goes to $-\infty$ when the integration variable is close to 1.

Here is a plot of the integrand.

```
In[118]:= Plot[Log[(1 - x)^2], {x, 0, 2}, PlotRange → All]
```

Out[118]=



`NIntegrate` gives a result that is close to the exact one.

In[114]:=

$$\texttt{exact} = \int_0^2 \texttt{Log}\left[(1-\texttt{x})^2\right] d\texttt{x};$$

$$\frac{\texttt{exact} - \texttt{NIntegrate}\left[\texttt{Log}\left[(1-\texttt{x})^2\right], \{\texttt{x}, 0, 2\}\right]}{\texttt{exact}}$$

```
Out[115]= 0.0000124017
```

Convergence is achieved when `MaxRecursion` is increased.

```
In[45]:= NIntegrate[Log[(1 - x)^2], {x, 0, 2}, Method → "GlobalAdaptive", MaxRecursion → 100]

           NIntegrate::slwcon :
             Numerical integration converging too slowly; suspect one of the following: singularity, value
                 of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

Out[45]= -4.
```

With its default options `NIntegrate` has a sampling point at 1, as can be seen from the following.

Check that `NIntegrate` has 1 as a sampling point.

```
In[119]:= InputForm /@ Select[#, 0.9 < # < 1.01 &] &@
            Reap[NIntegrate[x, {x, 0, 2}, EvaluationMonitor :→ Sow[x]]][[2, 1]]
Out[119]= {1.}
```

But for `NIntegrate[Log[(1 - x)^2], {x, 0, 2}]` the evaluation monitor has not picked a sampling point that is 1.

Sampling points that belong to the interval $\left[1 - 10^6, 1 + 10^6\right]$.

```
In[120]:=  InputForm /@ Select[#, 0.99999 < # < 1.00001 &] &@Reap[
              NIntegrate[Log[(1 - x)^2], {x, 0, 2}, EvaluationMonitor :→ Sow[x]]][[2, 1]]
Out[120]=  {}
```

In other words, the singularity at 1 is ignored. Ignoring the singularity is equivalent to having an integrand that is zero at the singular sampling point.

Note that the integral is easily integrated if the singular point is specified in the variable range. Following are the numbers of sampling points and timings for `NIntegrate` with the singular and nonsingular range specifications.

Integration with the singular point specified.

```
In[123]:=  {Reap[NIntegrate[Log[(1 - x)^2], {x, 0, 1, 2}, EvaluationMonitor :> Sow[x]]][[2, 1]] //
             Length, Timing[NIntegrate[Log[(1 - x)^2], {x, 0, 1, 2}]][[1]]}
Out[123]=  {260, 0.005999}
```

Integration by ignoring the singularity.

```
In[122]:=  {Reap[NIntegrate[Log[(1 - x)^2], {x, 0, 2},
                 MaxRecursion -> 20, EvaluationMonitor :> Sow[x]]][[2, 1]] // Length,
            Timing[NIntegrate[Log[(1 - x)^2], {x, 0, 2}, MaxRecursion -> 20]][[1]]}
Out[122]=  {670, 0.008998}
```

A more interesting example of ignoring the singularity is using Bessel functions in the denominator of the integrand.

Integral with several (five) integrable singularities.

```
                                  1
In[124]:=  NIntegrate[ ------------------------- , {x, 1, 20}, MaxRecursion → 1000] // InputForm
                        Sqrt[Abs[BesselJ[2, x]]]
Out[124]//InputForm=  59.539197071142375
```

The result can be checked using `NIntegrate` with singular range specification with the zeros of `BesselJ[2, x]` (see `BesselJZero`).

Integration with the Bessel zeros specified as singular points.

```
In[125]:= NIntegrate[              1              ,
                     Sqrt[Abs[BesselJ[2, x]]]
            {x, 1, 5.135622301840683`, 8.417244140399848`, 11.619841172149059`,
             14.79595178235126`, 17.959819494987826`, 20}, PrecisionGoal → 8] // InputForm

Out[125]//InputForm= 59.53926944377681
```

Needless to say, the last integration required the calculation of the `BesselJ` zeros. The former one "just integrates" without any integrand analysis.

Ignoring the singularity may not work with oscillating integrands.

For example, these two integrals are equivalent.

```
                      1                                   1      1
In[126]:= Integrate[ ― Sin[x], {x, 1, ∞}] == Integrate[ ― Sin[ ― ], {x, 0, 1}]
                      x                                   x      x

Out[126]= True
```

`NIntegrate` can do the first one.

```
                       1
In[127]:= NIntegrate[ ― Sin[x], {x, 1, ∞}]
                       x

Out[127]= 0.624713
```

`NIntegrate` cannot do the second one.

```
                       1      1
In[128]:= NIntegrate[ ― Sin[ ― ], {x, 0, 1}, Method → "GlobalAdaptive", MaxRecursion → 100]
                       x      x

Out[128]= 0.×10^1
```

However, if the integrand is monotonic in a neighborhood of its singularity, or more precisely, if it can be majorized by a monotonic integrable function, it can be shown that by ignoring the singularity, convergence will be reached.

For theoretical justification and practical recommendations of ignoring the singularity see [DavRab65IS] and [DavRab84].

Automatic Singularity Handling

One-Dimensional Integration

When the option "SingularityHandler" is set to Automatic for a one-dimensional integral, "DoubleExponential" is used on regions that are obtained by "SingularityDepth" number of partitionings. As explained earlier, this region will not be partitioned further as long as the "DoubleExponential" singularity handler works over it. If the error estimate computed by "DoubleExponential" does not evolve in a way predicted by the theory of the double-exponential quadrature, the singularity handling for this region is switched to IMT.

As explained in "Convergence Rate", the following dependency of the error is expected with respect to the number of double-exponential sampling points:

$$e^{-\frac{c\,n}{\log n}},$$

where c is a positive constant. Consider the relative errors E_m and E_n of two consecutive double-exponential quadrature calculations, made with m and n number of sampling points respectively, for which $m < n$. Assuming $E_m < 1$, $E_n < 1$, and $E_m > E_n$ it should be expected that

$$\frac{E_m}{E_n} \geq \frac{e^{-\frac{c\,m}{\log m}}}{e^{-\frac{c\,n}{\log n}}} \Leftrightarrow \tag{8}$$

$$\frac{\log E_m}{\log E_n} \leq \frac{-\frac{c\,m}{\log m}}{-\frac{c\,n}{\log n}} = \frac{m \log n}{n \log m}. \tag{9}$$

The switch from "DoubleExponential" to IMT happens when:

(i) the region error estimate is larger than the absolute value of the region integral estimate (hence the relative error is not smaller than 1);

(ii) the inequality (2) is not true in two different instances;

(iii) the integrand values calculated with the double-exponential transformation do not decay fast enough.

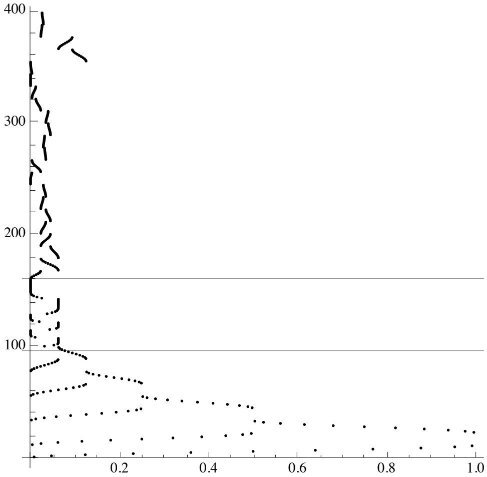
Here is an example of a switch from "DoubleExponential" to IMT singularity handling. On the plot the integrand is sampled at the x coordinates in the order of the y coordinates. The patterns of the sampling points over $\left[0, \frac{1}{16}\right]$ show the change from Gaussian quadrature ($y \in [0, 97]$) to double-exponential quadrature ($y \in [98, 160]$), which later is replaced by Gaussian quadrature using the IMT variable transformation ($y \in [160, 400]$).

In[143]:=
```
k = 0;
```

$$f[x\_] := \frac{70}{10^4 \left(x - \frac{1}{32}\right)^2 + \frac{1}{16}};$$

```
gr =
  Reap[NIntegrate[f[x], {x, 0, 1}, EvaluationMonitor :→ Sow[Point[{N[x], k++}]]]][[
    2, 1]]; Graphics[{PointSize[0.006], gr}, AspectRatio → 1,
  Axes → True, PlotRange → All, GridLines → {None, {97, 160}}]
```

Out[145]=



Multidimensional Integration

When the option "SingularityHandler" is set to Automatic for a multidimensional integral, both "DuffyCoordinates" and IMT are used.

A region needs to meet the following conditions in order for "DuffyCoordinates" to be applied:

- the region is obtained by "SingularityDepth" number of bisections (or partitionings) along each axis;
- the region is a corner of one of the initial integration regions (the specified integration region can be partitioned into integration regions by piecewise handling or by user-specified singularities).

A region needs to meet the following conditions in order for `IMT` to be applied:

- the region is obtained with by `"SingularityDepth"` number of bisections (or partitionings) along predominantly one axis;
- the region is not a corner region and it is on a side of one of the initial integration regions.

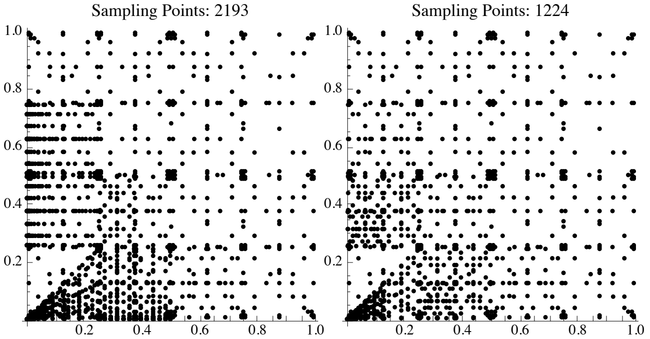
In other words, `IMT` is applied to regions that are derived through `"SingularityDepth"` number of partitionings but do not satisfy the conditions of the `"DuffyCoordinates"` automatic application.

`IMT` is effective if the singularity is along one of the axes. Using `IMT` for point singularities can be counterproductive.

Sampling points of two-dimensional integration, $\int_0^1\int_0^1 \frac{1}{\sqrt{x+y}}\,dy\,dx$, with `Automatic` (left) and `"DuffyCoordinates"` (right) singularity handling. It can be seen that the automatic singularity handling uses almost two times more points than `"DuffyCoordinates"`. To illustrate the effect of the singularity handlers they are applied after two bisections.

```
In[133]:= pointsAutomatic = Reap[NIntegrate[    1     , {x, 0, 1}, {y, 0, 1}, Method →
                                            ─────────
                                            √(x + y)
              {"GlobalAdaptive", "SingularityDepth" → 2, "SingularityHandler" → Automatic},
            EvaluationMonitor :→ Sow[{x, y}]]][[2, 1]]; pointsDuffy =
                            1
           Reap[NIntegrate[─────────, {x, 0, 1}, {y, 0, 1}, Method → {"GlobalAdaptive",
                           √(x + y)
              "SingularityDepth" → 2, "SingularityHandler" → "DuffyCoordinates"},
            EvaluationMonitor :→ Sow[{x, y}]]][[2, 1]];
          Row[{Graphics[{PointSize[0.015], Point /@ pointsAutomatic},
             Axes -> True, ImageSize → 200,
             PlotLabel → "Sampling\ Points:\ " <> ToString[Length[pointsAutomatic]]],
            Graphics[{PointSize[0.015], Point /@ pointsDuffy}, Axes -> True, ImageSize →
              200, PlotLabel → "Sampling\ Points:\ " <> ToString[Length[pointsDuffy]]]}]
```

Out[134]=



Timings for the integral, $\int_0^1 \int_0^1 \frac{1}{\sqrt{x+y}} \, dy \, dx$, with singularity handlers `Automatic`, `"DuffyCoordinates"`, and `IMT` and with no singularity handling. The integral has a point singularity at $\{0, 0\}$.

```
In[47]:= TableForm[{#, Timing[NIntegrate[1/√(x + y), {x, 0, 1}, {y, 0, 1},
              Method → {"GlobalAdaptive", "SingularityHandler" → #}]][[1]]} & /@
           {Automatic, "DuffyCoordinates", "IMT", None}, TableHeadings →
           {None, {StyleForm[ColumnForm[{"Singularity", "handler"}], FontFamily → Times],
             StyleForm["Time (s)", FontFamily → Times]}}]

           NIntegrate::slwcon :
             Numerical integration converging too slowly; suspect one of the following: singularity, value
                of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

                        Singularity          Time (s)
                        handler
                        ___________________________________
Out[47]//TableForm=     Automatic            0.023997
                        DuffyCoordinates     0.015997
                        IMT                  0.032995
                        None                 0.032995
```

Timings for the integral, $\int_0^1 \int_0^1 \left(x^{10} + \frac{1}{\sqrt{y}} \right) dy \, dx$, singular along y axis with singularity handlers `Automatic`, `"DuffyCoordinates"`, and `IMT` and with no singularity handling.

```
In[46]:= TableForm[
          {#, Timing[NIntegrate[1/√y + x^10, {x, 0, 1}, {y, 0, 1}, Method → {"GlobalAdaptive",
                "SingularityHandler" → #}, MaxRecursion → 20]][[1]]} & /@
           {Automatic, "DuffyCoordinates", "IMT", None}, TableHeadings →
           {None, {StyleForm[ColumnForm[{"Singularity", "handler"}], FontFamily → Times],
             StyleForm["Time (s)", FontFamily → Times]}}]

                        Singularity          Time (s)
                        handler
                        ___________________________________
Out[46]//TableForm=     Automatic            0.021997
                        DuffyCoordinates     0.038994
                        IMT                  0.023996
                        None                 0.035995
```

Cauchy Principal Value Integration

To evaluate the Cauchy principal value of an integral, `NIntegrate` uses the strategy `PrincipalValue`.

Cauchy principal value integration with singular point at 2.

In[153]:= $\mathbf{NIntegrate}\left[\frac{\sqrt{\mathbf{x}}}{\mathbf{x-2}}, \{\mathbf{x}, 0, 2, 5\}, \mathbf{Method} \to \text{"}\mathbf{PrincipalValue}\text{"}\right]$

```
Out[153]= 2.36355
```

In `NIntegrate`, `PrincipalValue` uses the strategy specified by its `Method` option to work directly on those regions where there is no difficulty and by pairing values symmetrically about the specified singularities in order to take advantage of the cancellation of the positive and negative values.

option name	*default value*	
`Method`	`Automatic`	method specification used to compute estimates over subregions
`SingularPointIntegrationRadius`	`Automatic`	a number ϵ_1 or a list of numbers $\{\epsilon_1, \epsilon_2, \ldots, \epsilon_n\}$ that correspond to the singular points $b_1, b_2, \ldots, b_n$ in the range specification; with each pair (b_i, ϵ_i) an integral of the form $\int_0^\epsilon (f(b+t) + f(b-t))\, dt$ is formed

`"PrincipalValue"` options.

Thus the specification

```
NIntegrate[f[x], {x, a, b, c}, Method -> {"PrincipalValue",
   Method -> methodspec, "SingularPointIntegrationRadius" -> ϵ}]
```

is evaluated as

$$\int_a^{b-\epsilon} f(x)\, dx + \int_0^{\epsilon} (f(b+t) + f(b-t))\, dt + \int_{b+\epsilon}^{c} f(x)\, dx,$$

where each of the integrals is evaluated using `NIntegrate` with `Method` -> *methodspec*. If ϵ is not given explicitly, a value is chosen based upon the differences $b-a$ and $c-b$. The option `SingularPointIntegrationRadius` can take a list of numbers that equals the number of singular points. For the derivation of the formula see [DavRab84].

This finds the Cauchy principal value of $\int_{-1/2}^{1} \frac{1}{x+x^2}\, dx$.

In[14]:= $$\mathbf{NIntegrate}\left[\frac{\mathbf{1}}{\mathbf{x+x^2}}, \left\{\mathbf{x}, -\frac{\mathbf{1}}{\mathbf{2}}, \mathbf{0}, \mathbf{1}\right\}, \mathbf{Method} \to \mathbf{PrincipalValue}\right]$$

Out[14]=
```
-0.6931471805596523
```

Here is the Cauchy principal value of $\int_{-2}^{1} \frac{1}{x+x^2}\, dx$. Note that there are two singularities that need to be specified.

In[114]:=
```
NIntegrate[1/(x+x^2), {x, -2, -1, 0, 1},Method->PrincipalValue]
```

Out[114]=
```
-1.38629
```

The singular points can be specified using the `Exclusions` option.

In[30]:=
```
NIntegrate[1 / (x + x^2), {x, -2, 1}, Method -> PrincipalValue, Exclusions → {-1, 0}]
```

Out[30]=
```
-1.38629
```

This checks the value. The result would be 0 if everything were done exactly.

In[31]:=
```
% + 2Log[2]
```

Out[31]= 7.59615×10^{-13}

It should be noted that the singularities must be located exactly. Since the algorithm pairs together the points on both sides of the singularity, if the singularity is slightly mislocated the cancellation will not be sufficiently good near the pole and the result can be significantly in error if `NIntegrate` converges at all.

Sampling Points Visualization

Consider the calculation of the principal value of

$$\int_0^2 \frac{1}{\log(x)}\, dx.$$

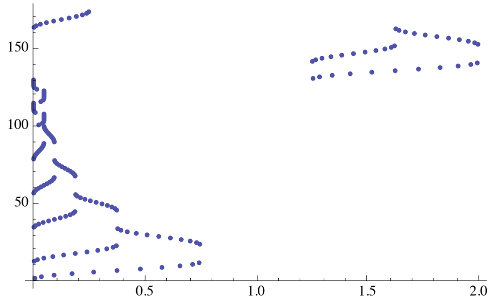
The following examples show two ways of visualizing the sampling points. The first shows the sampling points used. Since the integrand is modified in order to do the principal value integration, it might be desired to see the points at which the original integrand is evaluated. This is shown on the second example.

Here are sampling points used by `NIntegrate`. There are no points over the interval $\left[\frac{3}{4}, \frac{5}{4}\right]$, because of the `PrincipalValue` integration $\int_0^{1-1/4} \frac{1}{\log(x)}\, dx + \int_0^{1/4}\left(\frac{1}{\log(1+t)} + \frac{1}{\log(1-t)}\right) dt + \int_{1+1/4}^{2} \frac{1}{\log(x)}\, dx$, and there are sampling points over $\left[0, \frac{1}{4}\right]$.

In[154]:=

```
k = 0;
tbl = Reap[NIntegrate[1 / Log[x], {x, 0, 1, 2},
      Method → {"PrincipalValue", "SingularPointIntegrationRadius" → 1 / 4},
      EvaluationMonitor :→ Sow[{x, ++k}]]][[2, 1]];
ListPlot[tbl, PlotRange -> All]
```

Out[156]=



This defines a function which accumulates the argument values given to the integrand.

In[1]:=

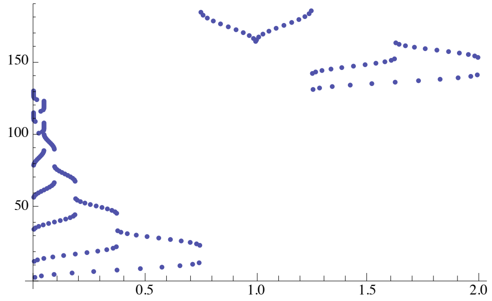
```
Clear[f]; f[x_?NumericQ] := (AppendTo[tbl, {x, ++k}]; 1 / Log[x]);
```

Here are the points at which the integrand has been evaluated. Note the symmetric pattern over the interval $\left[\frac{3}{4}, \frac{5}{4}\right]$.

In[166]:=

```
k = 0; tbl = {};
NIntegrate[f[x], {x, 0, 1, 2},
  Method -> {"PrincipalValue", "SingularPointIntegrationRadius" → 1 / 4}];
ListPlot[tbl, PlotRange -> All]
```

Out[168]=



Double-Exponential Strategy

The double-exponential quadrature consists of applying the trapezoidal rule after a variable transformation. The double-exponential quadrature was proposed by Mori and Takahasi in 1974 and it was inspired by the so-called IMT rule and TANH rule. The transformation is given the name "double-exponential" since its derivative decreases double-exponentially when the integrand's variable reaches the ends of the integration region.

The double-exponential algorithm for `NIntegrate` is specified with the `Method` option value `"DoubleExponential"`.

```
In[169]:= NIntegrate[1 / Sqrt[x], {x, 0, 1}, Method -> "DoubleExponential"]
Out[169]= 2.
```

option name	*default value*	
`"ExtraPrecision"`	`50`	maximum extra precision to be used internally
`"SymbolicProcessing"`	`Automatic`	number of seconds to do symbolic preprocessing

`"DoubleExponential"` options.

The double-exponential strategy can be used for one-dimensional and multidimensional integration. When applied to multidimensional integrals it uses the Cartesian product of the trapezoidal rule.

A double-exponential transformation $\phi(t)$ transforms the integral

$$\int_a^b f(t)\, dx \tag{10}$$

into

$$\int_{-\infty}^{+\infty} f(\phi(t))\, \phi'(t)\, dx, \tag{11}$$

where (a, b) can be finite, half-infinite ($b = \infty$), or infinite ($a = -\infty$, $b = \infty$). The integrand $f(x)$ must be analytic in (a, b) and might have singularity at one or both of the end points.

The transformed integrand decreases double-exponentially, that is, $f(\phi(t))\, \phi'(t) \approx \exp(-c \exp(|t|))$ as $|t| \to \pm\infty$.

The function $\phi(t)$ is analytic in $(-\infty, \infty)$. It is known that for an integral like (11) of an analytic integrand the trapezoidal rule is an optimal rule [Mori74].

The transformations used for the different types of integration regions are:

$$\int_a^b f(x)\,dx \Longrightarrow x = \frac{a+b}{2} + \frac{1}{2}(b-a)\tanh\left(\frac{1}{2}\pi\sinh(x)\right), \tag{12}$$

$$\int_a^\infty f(x)\,dx \Longrightarrow x = a + e^{\frac{1}{2}\pi\sinh(x)},$$

$$\int_{-\infty}^\infty f(x)\,dx \Longrightarrow x = \sinh\left(\frac{1}{2}\pi\sinh(x)\right),$$

where a and b are finite numbers.

The trapezoidal rule is applied to (11):

$$\mathrm{DE}(h) = h\sum_{i=-\infty}^{\infty} f(\phi(i\,h))\,\phi'(i\,h) \tag{13}$$

The terms in (13) decay double-exponentially for large enough $|i|$. Therefore the summation in (13) is cut off at the terms that are too small to contribute to the total sum. (A criterion similar to (3) for the local adaptive strategy is used. See also the following double-exponential example implementation.)

The strategy `"DoubleExponential"` employs the double-exponential quadrature.

The `"DoubleExponential"` strategy works best for analytic integrands; see "Comparison of Double-Exponential and Gaussian Quadrature".

`"DoubleExponential"` uses the Cartesian product of double-exponential quadratures for multidimensional integrals.

Cartesian double-exponential quadrature.

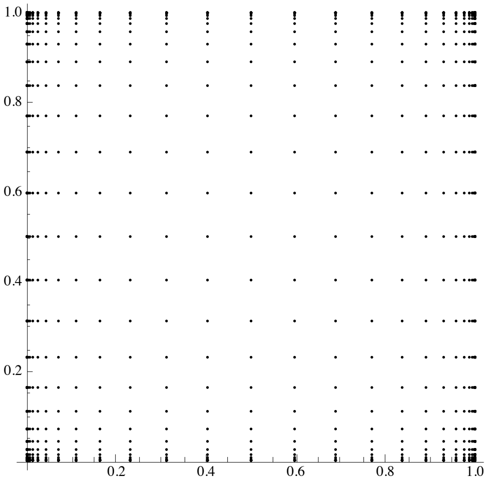
```
In[48]:= NIntegrate[1/Sqrt[x + y], {x, 0, 1}, {y, 0, 1},
          Method → "DoubleExponential", MaxRecursion → 200]
Out[48]= 1.10457
```

As with the other Cartesian product rules, if "DoubleExponential" is used for dimensions higher than three, it might be very slow due to combinatorial explosion.

The following plot illustrates the Cartesian product character of the "DoubleExponential" multidimensional integration.

```
In[49]:= tbl = Reap[
              NIntegrate[Sqrt[x] Sqrt[y], {x, 0, 1}, {y, 0, 1}, Method → "DoubleExponential",
                MaxRecursion → 200, EvaluationMonitor :→ Sow[{x, y}]]][[2, 1]];
         Graphics[{PointSize[0.005], Point /@ N[tbl]}, Axes → True]
```

Out[50]=



Double-exponential quadrature can be used for singularity handling in adaptive strategies; see "Singularity Handling".

MinRecursion and MaxRecursion

The option MinRecursion has the same meaning and functionality as it does for "GlobalAdaptive" and "LocalAdaptive" described in "MinRecursion and MaxRecursion". MaxRecursion for "DoubleExponential" restricts how many times the trapezoidal quadrature estimates are improved; see "Example Implementation of Double-Exponential Quadrature".

Comparison of Double-Exponential and Gaussian Quadrature

The "DoubleExponential" strategy works best for analytic integrands. For example, the following integral is done by "DoubleExponential" three times faster than the Gaussian quadrature (using a global adaptive algorithm).

Integration with "DoubleExponential".

```
In[215]:= NIntegrate[Log[1/x]/x^(1/4), {x, 0, 1}, PrecisionGoal → 10,
            Method → {"DoubleExponential", "SymbolicProcessing" → 0}] // Timing

Out[215]= {0.001999, 1.77778}
```

Integration with Gauss quadrature. (The default strategy of NIntegrate, "GlobalAdaptive" uses by default a Gauss-Kronrod integration rule with 5 Gaussian points and 6 Kronrod points.)

```
In[51]:= NIntegrate[Log[1/x]/x^(1/4), {x, 0, 1}, PrecisionGoal → 10, MaxRecursion → 100, Method →
             {"GlobalAdaptive", "SingularityDepth" → ∞, "SymbolicProcessing" → 0}] // Timing

Out[51]= {0.008998, 1.77778}
```

Since "DoubleExponential" converges double-exponentially with respect to the number of evaluation points, increasing the precision goal slightly increases the work done by "DoubleExponential". This is illustrated for two integrals, $\int_0^1 \frac{1}{\sqrt{x}}\,dx$ and $\int_0^1 e^{20(x-1)}\sin(256\,x)\,dx$. Each table entry shows the error and number of evaluations.

Double-exponential quadrature and Gaussian quadrature for $\int_0^1 \frac{1}{\sqrt{x}}\,dx$. Increasing the precision goal does not change the number of sampling points used by "DoubleExponential".

```
In[217]:= methods = {"DoubleExponential", "GlobalAdaptive"};
          pgoals = Range[5, 15, 2];
          TableForm[
            Outer[(k = 0; res = NIntegrate[1/Sqrt[x], {x, 0, 1}, Method → #1, PrecisionGoal → #2,
                   MaxRecursion → 20, EvaluationMonitor :> k++]; {Abs[res - 2] / 2, k}) &,
              methods, pgoals] // Transpose, TableHeadings → {pgoals, methods}, TableDepth → 2]

              | DoubleExponential        GlobalAdaptive
          ------------------------------------------------------------
           5  | {1.55431×10^-15, 33}    {1.55431×10^-15, 132}
           7  | {0., 64}                {1.55431×10^-15, 132}
Out[219]=  9  | {0., 64}                {8.88178×10^-16, 229}
          11  | {0., 64}                {8.88178×10^-16, 273}
          13  | {0., 64}                {8.88178×10^-16, 405}
          15  | {0., 123}               {8.88178×10^-16, 640}
```

Double-exponential quadrature and Gaussian quadrature for $\int_0^1 e^{20\,(x-1)}\sin(256\,x)\,dx$. Increasing the precision goal does not change the number of sampling points used by "DoubleExponential". (The integrations are done without symbolic preprocessing.)

```
In[220]:= methods = {"DoubleExponential", "GlobalAdaptive"};
          pgoals = Range[6, 10, 2];
          TableForm[
           Outer[(k = 0; res = NIntegrate[Exp[20 (x - 1)] Sin[256 x], {x, 0, 1}, Method →
                  {#1, "SymbolicProcessing" → 0}, PrecisionGoal → #2, MaxRecursion → 20,
                 EvaluationMonitor :> k++]; {Abs[res - 2] / 2, k}) &, methods, pgoals, 1] //
            Transpose, TableHeadings → {pgoals, methods}, TableDepth → 2]

             DoubleExponential  GlobalAdaptive
Out[222]= 6  {1.00007, 758}     {1.00007, 1454}
          8  {1.00007, 758}     {1.00007, 2357}
          10 {1.00007, 758}     {1.00007, 3369}
```

On the other hand, for non-analytic integrands "DoubleExponential" is quite slow, and a global adaptive algorithm using Gaussian quadrature can resolve the singularities easily.

"DoubleExponential" needs more than 10000 integrand evaluations to compute this integral with a non-analytic integrand.

```
In[52]:= k = 0;
         {NIntegrate[Abs[Sin[3 * x]], {x, 0, π},
           Method → {"DoubleExponential", "SymbolicProcessing" → 0},
           MaxRecursion → 10, EvaluationMonitor :> k++], k}

Out[53]= {2., 10185}
```

Gaussian quadrature is much faster for the integral.

```
In[54]:= k = 0; {NIntegrate[Abs[Sin[3 * x]], {x, 0, π},
           Method → {"GlobalAdaptive", "SymbolicProcessing" → 0},
           MaxRecursion → 10, EvaluationMonitor :> k++], k}

           NIntegrate::slwcon :
             Numerical integration converging too slowly; suspect one of the following: singularity, value
                of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

Out[54]= {2., 385}
```

Further, "DoubleExponential" might be slowed down by integrands that have nearly discontinuous derivatives, that is, integrands that are not "very" analytical.

Here is an example with a not "very" analytical integrand.

In[226]:=

$$\texttt{NIntegrate}\left[\frac{1}{16\left(x-\frac{\pi}{4}\right)^2+\frac{1}{16}},\ \{x,0,1\},\ \texttt{Method} \to \texttt{"DoubleExponential"}\right] \texttt{ // Timing}$$

```
Out[226]= {0.011998, 2.77878}
```

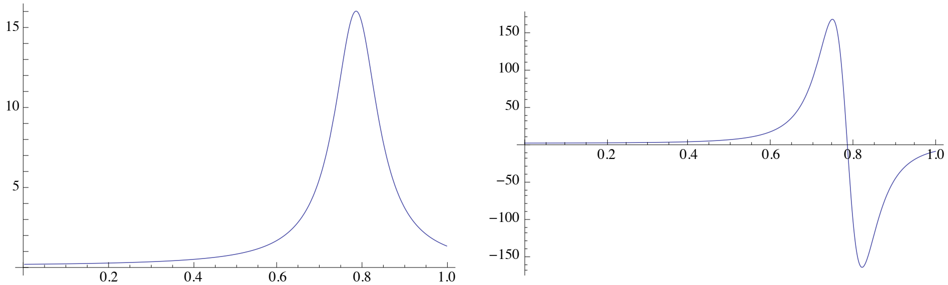
Again, Gaussian quadrature is much faster.

```
In[227]:= NIntegrate[1/(16 (x - π/4)^2 + 1/16), {x, 0, 1}, Method → "GlobalAdaptive"] // Timing

Out[227]= {0.005999, 2.77878}
```

Here are the plots of the integrand $\frac{1}{16\left(x-\frac{\pi}{4}\right)^2+\frac{1}{16}}$ and its derivative.

```
In[228]:= Block[{gr, gr1},
            gr = Plot[1/(16 (x - π/4)^2 + 1/16), {x, 0, 1}, PlotRange → All];
            gr1 = Plot[D[1/(16 (x - π/4)^2 + 1/16), x] // Evaluate, {x, 0, 1}, PlotRange → All];
            GraphicsArray[{gr, gr1}]
          ]
```

Out[228]=



Convergence Rate

This section demonstrates that the asymptotic error of the double-exponential quadrature in terms of the number n of evaluation points used is

$$e^{-\frac{c\,n}{\log(n)}}, \tag{14}$$

where c is a positive constant.

This defines a double-exponential integration function that an returns integral estimate and the number of points used.

```
In[229]:= DERuleEstimate[f_, {a_, b_}, h_, wprec_: MachinePrecision] :=
            Block[{$MaxExtraPrecision = 50 000, ϕ, F, i, j, temp, s1, s2},
             ϕ[t_] := Rescale[1/2 (Tanh[1/2 π Sinh[t]] + 1), {0, 1}, {a, b}];
             F[t_] := Evaluate[f[ϕ[t]] * D[ϕ[t], t]];
             i = 1;
             s1 = FixedPoint[(temp = F[i * h]; i++; N[N[temp, 3 * wprec] + #1, wprec]) &, 0];
             j = -1;
             s2 = FixedPoint[(temp = F[j * h]; j--; N[N[temp, 3 * wprec] + #1, wprec]) &, 0];
             {i - j + 1, h (s1 + F[0] + s2)}
            ];
```

This defines a function.

```
In[230]:= f[x_] := 1/x^(1/4) Log[1/x]
```

This is the exact integral.

```
In[231]:= exact = Integrate[f[x], {x, 0, 1}]

Out[231]= 16/9
```

This finds the errors and number of evaluation points for a range of step sizes of the trapezoidal rule.

```
In[232]:= {a, b} = {0, 1};
          wprec = 30;
          range = Table[1/i, {i, 2, 7}];
          range = Join[range, Mean /@ Partition[range, 2, 1]];
          range = Sort[range, Greater];
          err = Map[DERuleEstimate[f, {a, b}, #1, wprec] &, range];
          err = Map[{#1⟦1⟧, Abs[exact - #1⟦2⟧]} &, err]; (* errors *)
          logErr = Map[{#1⟦1⟧, Log[#1⟦2⟧]} &, err]; (* logarithm of the errors *)
          points = First /@ err;
```

This fits $\frac{x}{\text{Log}[x]}$ through the logarithms of the errors; see (14).

```
In[239]:= p[x_] := Evaluate[Fit[logErr, {1, x/Log[x]}, x]]
```

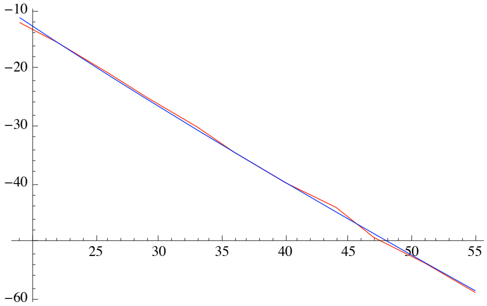
Here is the fitted function. The summation term 30.48 is just a translation parameter.

```
In[240]:= p[x]

Out[240]= 30.48 - (6.497 x)/Log[x]
```

You see that the errors fit the model (14):

```
In[241]:= ListLinePlot[{logErr, {#1, p[#1]} & /@ points},
           PlotRange -> All, PlotStyle -> {{Red}, {Blue}}]
```

Out[241]=



Example Implementation of Double-Exponential Quadrature

Following is an example implementation of the double-exponential quadrature with the finite region variable transformation (transformation (12) earlier).

This is a definition of a function that applies the trapezoidal rule to a transformed integrand. The function uses (13) and it is made to reuse integral estimates computed with a twice larger step.

```
In[173]:= IRuleEstimate[F_, h_, oldSum_: None] :=
            Block[{$MaxExtraPrecision = 50 000, step, i, temp, s1, s2},
             If[oldSum === None, step = 1, step = 2];
             i = 1;
             s1 = FixedPoint[(temp = F[i * h]; i += step; N[N[temp, 60] + #1]) &, 0];
             i = -1;
             s2 = FixedPoint[(temp = F[i * h]; i -= step; N[N[temp, 60] + #1]) &, 0];
                                                                   oldSum
             If[oldSum === None, h (s1 + F[0] + s2), h (s1 + s2) + ──────]
                                                                     2
            ];
```

This is a definition of a simple double-exponential strategy, which finds the integral of the function f over the finite interval $\{a, b\}$ with relative error tol.

```
In[189]:= Options[IStrategyDoubleExp] = {"MaxRecursion" → 7};
          IStrategyDoubleExp[f_, {a_, b_}, tol_, opts___] :=
            Module[{ϕ, F, h, t, temp, k = 0, maxrec},
             maxrec = "MaxRecursion" /. {opts} /. Options[IStrategyDoubleExp];
                                         1        1
             ϕ[t_] := Evaluate[Rescale[ ─ (Tanh[ ─ π Sinh[t]] + 1), {0, 1}, {a, b}]];
                                         2        2
             F[t_] := Evaluate[f[ϕ[t]] * D[ϕ[t], t]];
             h = 1;
             NestWhile[((temp = IRuleEstimate[F, h /= 2, #1]) && k++ < maxrec) &,
              IRuleEstimate[F, h, None], (Abs[#1] * tol <= Abs[#1 - #2]) &, 2];
             temp
            ];
```

This defines a function that is singular at 0.

In[194]:= $\texttt{f[x\_] :=} \dfrac{1}{\sqrt[4]{x}}$

Here is the integral estimate from the double-exponential strategy.

In[195]:= IStrategyDoubleExp[f, {0, 1}, 10^{-8}] // InputForm

```
Out[195]//InputForm= 1.3333333333333333
```

Here is the exact result.

```
In[177]:= Integrate[f[x], {x, 0, 1}] // N // InputForm

Out[177]//InputForm= 1.3333333333333333
```

The two results are the same.

This defines an oscillating function.

```
In[178]:= f[x_] := Cos[64 * Sin[x]]
```

Here is the integral estimate given by the double-exponential strategy.

In[179]:= res = IStrategyDoubleExp[f, {0, π}, 10^{-8}]; res // InputForm

```
Out[179]//InputForm= 0.29088010217372606
```

Here is the exact result.

```
In[180]:= exact = Integrate[f[x], {x, 0, π}]

Out[180]= π BesselJ[0, 64]
```

Here is the exact result in machine precision.

```
In[181]:= exact // N // InputForm

Out[181]//InputForm= 0.2908801021737257
```

The relative error is within the prescribed tolerance.

```
In[182]:= Abs[res - exact] / exact
```

Out[182]= 1.33587×10^{-15}

"Trapezoidal" Strategy

The "Trapezoidal" strategy gives optimal convergence for analytic periodic integrands when the integration interval is exactly one period.

option name	default value	
"ExtraPrecision"	50	maximum extra precision to be used internally
"SymbolicProcessing"	Automatic	number of seconds to do symbolic preprocessing

"Trapezoidal" options.

"Trapezoidal" takes the same options as "DoubleExponential". If the integration ranges are infinite or semi-infinite, "Trapezoidal" becomes "DoubleExponential".

For theoretical background, examples, and explanations of periodic functions integration (with trapezoidal quadrature) see [Weideman2002].

```
                        1
In[109]:= NIntegrate[ ————, {x, 0, 1}, Method → {"Trapezoidal", "SymbolicProcessing" → 0}]
                       √x

          NIntegrate::ncvi :
            NIntegrate failed to converge to prescribed accuracy after 9 iterated refinements in x in the
                region {{0., 1.}}. NIntegrate obtained 1.9771819583163235` and
                0.009451548754043415` for the integral and error estimates.

Out[109]= 1.97718
```

Here is a table that shows the number of sampling points for different values of the parameter t used by "GlobalAdaptive" and "Trapezoidal" respectively for the integral $\int_0^\pi \frac{\cos(t\sin(x)-k\,x)}{\pi}\,dx$, $k = 1$.

```
In[33]:= k = 1;
         tab =
           Table[{t, ({"IntegralEstimate", "Evaluations", "Timing"} /. NIntegrateProfile[
                          1
                NIntegrate[— Cos[t Sin[x] - k x], {x, 0, π}, Method → "GaussKronrodRule"],
                          π
                1])[[2]], ({"IntegralEstimate", "Evaluations", "Timing"} /.
                                                 1
               NIntegrateProfile[NIntegrate[— Cos[t Sin[x] - k x], {x, 0, π},
                                                 π
                 Method -> "Trapezoidal"], 1])[[2]]}, {t, 8, 80, 4}];
         TableForm[tab, TableHeadings → {None, {t, "GlobalAdaptive", "Trapezoidal"}}]
```

```
                     t   GlobalAdaptive  Trapezoidal
                     8   143             33
                     12  209             33
                     16  275             65
                     20  399             65
                     24  457             65
                     28  591             65
                     32  743             65
                     36  743             65
                     40  741             65
Out[35]//TableForm=  44  809             129
                     48  1007            129
                     52  941             129
                     56  963             129
                     60  1095            129
                     64  1121            129
                     68  1095            129
                     72  1137            129
                     76  1338            129
                     80  1227            129
```

Example Implementation

This function makes a trapezoidal quadrature integral estimate with specified points.

```
In[242]:= TrapStep[f_, {a_, b_}, n_?IntegerQ] :=
            Module[{h, absc, is},
                     b - a
              h = -------;
                     n - 1
              absc = Table[i, {i, a, b, h}];
              is = h * Total[MapAt[# / 2 &, f /@ absc, {{1}, {-1}}]];
              {is, ∞, n}
             ];
```

This function improves a trapezoidal quadrature integral estimate using sampling points between the old ones.

```
In[257]:= TrapStep[f_, {a_, b_}, {oldEstimate_, oldError_, oldn_}] :=
            Module[{n, h, absc, is},
              n = 2 oldn - 1;
                     b - a
              h = -------;
                     n - 1
              absc = Table[i, {i, a + h, b - h, 2 h}];
              is = h * Total[f /@ absc] + oldEstimate / 2;
              {is, Abs[is - oldEstimate], n}
             ];
```

This function is an interface to the preceding one.

```
In[272]:= Options[TrapezoidalIntegration] = {"MaxRecursion" → 7};
          TrapezoidalIntegration[f_, {a_, b_}, tol_, opts___] :=
           Block[{maxrec, k = 0, temp},
            maxrec = "MaxRecursion" /. {opts} /. Options[TrapezoidalIntegration];
            NestWhile[((temp = TrapStep[f, {a, b}, #]) && k++ < maxrec) &,
              TrapStep[f, {a, b}, 5], #[[2]] > tol &][[1]];
            temp[[1]]
           ]
```

Here is a definition of a (Bessel) function.

```
                    1
In[269]:= f[x_] := — Cos[80 Sin[x] - x]
                    π
```

Here is the trapezoidal quadrature estimate.

```
In[274]:= res = TrapezoidalIntegration[f, {0, π}, 10^-5] // N
Out[274]= -0.0560573
```

Here is the exact value.

```
In[278]:= exact = Integrate[f[x], {x, 0, π}]
Out[278]= BesselJ[1, 80]
```

The relative error is within the prescribed tolerance.

```
In[279]:= Abs[res - exact] / exact
Out[279]= -0.572732
```

Oscillatory Strategies

The oscillatory strategies of `NIntegrate` are are for one-dimensional integrals. Generally in quadrature, the algorithms for finite region integrals are different from the algorithms for infinite regions. `NIntegrate` uses Chebyshev expansions of the integrand and the global adaptive integration strategy for finite region oscillatory integrals. For infinite oscillatory integrals `NIntegrate` uses either a modification of the double-exponential algorithm or sequence summation acceleration over the sequence of integrals with regions between the integrand's zeros.

Here is an example that uses both algorithms.

```
            ⎡ Cos[2000 x]
            ⎢ ───────────          0 < x < 2
            ⎢     √x
In[13]:= NIntegrate⎢ Sin[20 x]
            ⎢ ─────────            x < 0      , {x, -∞, ∞}⎦
            ⎢   √(x²)
            ⎣ BesselY[2, x³] / x   x > 2

Out[13]= -1.5496
```

NIntegrate automatically detects oscillatory (one-dimensional) integrands, and automatically decides which algorithm to use according to the integrand's range.

The integrals detected as being of oscillatory type have the form

$$\int_a^b k(x)\, f(x)\, d x,$$

in which the oscillating kernel $k(x)$ is of the form:

1. $\sin(\omega\, x^p + c)$, $\cos(\omega\, x^p + c)$, $e^{i\,\omega\, x^p}$ for (a, b) finite;
2. $\sin(\omega\, x^p + c)$, $\cos(\omega\, x^p + c)$, $e^{i\,\omega\, x^p}$, $J_\nu(\omega\, x^p + c)$, $Y_\nu(\omega\, x^p + c)$, $H_\nu{}^{(1)}(\omega\, x^p + c)$, $H_\nu{}^{(2)}(\omega\, x^p + c)$, $j_\nu(\omega\, x^p + c)$, or $y_\nu(\omega\, x^p + c)$ for (a, b) infinite or semi-infinite.

In these oscillating kernel forms ω, c and ν are real constants, and p is a positive integer.

Finite Region Oscillatory Integration

Modified Clenshaw-Curtis quadrature ([PiesBrand75][PiesBrand84]) is for finite region one-dimensional integrals of the form

$$\int_a^b \sin\,(\omega\, x^p + c)\, f(x)\, d x,\ \int_a^b \cos(\omega\, x^p + c)\, f(x)\, d x,\ \text{ or } \int_a^b \exp(i\,\omega\, x^p + c)\, f(x)\, d x, \tag{15}$$

where a, b, k, c, p are finite real numbers.

The modified Clenshaw-Curtis quadrature rule approximates $f(x)$ with a single polynomial through Chebyshev polynomials expansion. This leads to simplified computations because of the orthogonality of the Chebyshev polynomials with sine and cosine functions. The modified Clenshaw-Curtis quadrature rule is used with the strategy "GlobalAdaptive". For smooth $f(x)$ the modified Clenshaw-Curtis quadrature is usually superior [KrUeb98] to other approaches for oscillatory integration (as Filon's quadrature and multi-panel integration between the zeros of the integrand).

Modified Clenshaw-Curtis quadrature is quite good for highly oscillating integrals of the form (15). For example, modified Clenshaw-Curtis quadrature uses less than a hundred integrand evaluations for both $\frac{\sin(200\,x)}{x^2}$ and $\frac{\sin(20\,000\,x)}{x^2}$.

Number of integrand evaluations for modified Clenshaw-Curtis quadrature for slowly oscillating kernel.

In[1]:= $\mathtt{k\ =\ 0;\ NIntegrate}\left[\frac{\mathtt{Sin[200\ x]}}{\mathtt{x}^2},\ \left\{\mathtt{x},\ \frac{2}{10},\ 2\right\},\ \mathtt{EvaluationMonitor\ :>\ k{+}{+}}\right]\mathtt{;\ k}$

Out[1]= `78`

Timing and integral estimates for modified Clenshaw-Curtis quadrature for slowly oscillating kernel.

In[3]:= $\mathtt{NIntegrate}\left[\frac{\mathtt{Sin[200\ x]}}{\mathtt{x}^2},\ \left\{\mathtt{x},\ \frac{2}{10},\ 2\right\}\right]\ \mathtt{//\ Timing}$

Out[3]= `{0.17, -0.0777739}`

Number of integrand evaluations for modified Clenshaw-Curtis quadrature for highly oscillating kernel.

In[5]:= $\mathtt{k\ =\ 0;\ NIntegrate}\left[\frac{\mathtt{Sin[20\,000\ x]}}{\mathtt{x}^2},\ \left\{\mathtt{x},\ \frac{2}{10},\ 2\right\},\ \mathtt{EvaluationMonitor\ :>\ k{+}{+}}\right]\mathtt{;\ k}$

Out[5]= `78`

Timing and integral estimates for modified Clenshaw-Curtis quadrature for highly oscillating kernel.

In[6]:= $\mathtt{NIntegrate}\left[\frac{\mathtt{Sin[20\,000\ x]}}{\mathtt{x}^2},\ \left\{\mathtt{x},\ \frac{2}{10},\ 2\right\}\right]\ \mathtt{//\ Timing}$

Out[6]= `{0.111, -0.000916893}`

On the other hand, without symbolic preprocessing, the default `NIntegrate` method—`"GlobalAdaptive"` strategy with a Gauss-Kronrod rule—uses thousands of evaluations for $\frac{\sin(200\,x)}{x^2}$, and it cannot integrate $\frac{\sin(20\,000\,x)}{x^2}$.

Number of integrand evaluations for Gaussian quadrature for slowly oscillating kernel.

```
In[7]:= k = 0; NIntegrate[Sin[200 x]/x^2, {x, 2/10, 2},
          Method → {Automatic, "SymbolicProcessing" → 0}, EvaluationMonitor :> k++]; k

Out[7]= 2656
```

Timing and integral estimates for Gaussian quadrature for slowly oscillating kernel.

```
In[8]:= NIntegrate[Sin[200 x]/x^2, {x, 2/10, 2},
           Method → {Automatic, "SymbolicProcessing" → 0}] // Timing

Out[8]= {0.2, -0.0777739}
```

Number of integrand evaluations for Gaussian quadrature for highly oscillating kernel.

```
In[9]:= k = 0; NIntegrate[Sin[20 000 x]/x^2, {x, 2/10, 2},
          Method → {Automatic, "SymbolicProcessing" → 0}, EvaluationMonitor :> k++]; k

      NIntegrate::slwcon :
        Numerical integration converging too slowly; suspect one of the following: singularity, value
              of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

      NIntegrate::ncvb :
        NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near
              {x} = {0.330106}. NIntegrate obtained -0.0905744 and
              0.42924020409664687` for the integral and error estimates. ≫

Out[9]= 1290
```

Timing and integral estimates for Gaussian quadrature for highly oscillating kernel.

```
In[10]:= NIntegrate[Sin[20 000 x]/x^2, {x, 2/10, 2},
            Method → {Automatic, "SymbolicProcessing" → 0}] // Timing

      NIntegrate::slwcon :
        Numerical integration converging too slowly; suspect one of the following: singularity, value
              of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

      NIntegrate::ncvb :
        NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near
              {x} = {0.330106}. NIntegrate obtained -0.0905744 and
              0.42924020409664687` for the integral and error estimates. ≫

Out[10]= {0.391, 0.×10^-1}
```

Extrapolating Oscillatory Strategy

The `NIntegrate` strategy `"ExtrapolatingOscillatory"` is is for oscillating integrals in infinite one-dimensional regions. The strategy uses sequence convergence acceleration for the sum of the sequence that consists of each of the integrals with regions between two consecutive zeros of the integrand.

Here is an example of an integration using `"ExtrapolatingOscillatory"`.

```
                                        1
In[294]:=  NIntegrate[Sin[200 x^2 + 5] ─────────,
                                    (x + 1)^2
             {x, 0, ∞}, Method → "ExtrapolatingOscillatory"]
Out[294]=  -0.0309721
```

option name	default value	
`Method`	`GlobalAdaptive`	integration strategy used to integrate between the zeros and which will be used if `ExtrapolatingOscillatory` fails
`"SymbolicProcessing"`	`Automatic`	number of seconds to do symbolic processing

Consider the integral

$$\int_a^\infty k(x)\, f(x)\, dx, \tag{16}$$

where the function $k(x)$ is the oscillating kernel and the function $f(x)$ is smooth. Let z_i be the zeros of $k(x)$ enumerated from the lower (finite) integration bound, that is, the inequality $a \le z_1 < z_2 < \ldots < z_i < \ldots$ holds. If the integral (16) converges then the sequence

$$\int_a^{z_1} k(x)\, f(x)\, dx,\ \int_a^{z_2} k(x)\, f(x)\, dx,\ \ldots,\ \int_a^{z_i} k(x)\, f(x)\, dx,\ \ldots \tag{17}$$

converges too. The elements of the sequence (17) are the partial sums of the sequence

$$\int_a^{z_1} k(x)\, f(x)\, dx,\ \int_{z_1}^{z_2} k(x)\, f(x)\, dx,\ \ldots,\ \int_{z_{i-1}}^{z_i} k(x)\, f(x)\, dx,\ \ldots. \tag{18}$$

Often a good estimate of the limit of the sequence (17) can be computed with relatively few elements of it through some convergence acceleration technique.

The `"Oscillatory"` strategy uses `NSum` with Wynn's extrapolation method for the integrals in (18). Each integral in (18) is calculated by `NIntegrate` without oscillatory methods.

The `"Oscillatory"` strategy applies its algorithm to oscillating kernels $k(x)$ in (16) that are of the form $\sin(\omega x^p + c)$, $\cos(\omega x^p + c)$, $J_\nu(\omega x^p + c)$, $Y_\nu(\omega x^p + c)$, $H_\nu^{(1)}(\omega x^p + c)$, $H_\nu^{(2)}(\omega x^p + c)$, $j_\nu(\omega x^p + c)$, or $y_\nu(\omega x^p + c)$, where ω, c, p, and ν are real constants.

Example Implementation

The following example implementation illustrates how the `"Oscillatory"` strategy works.

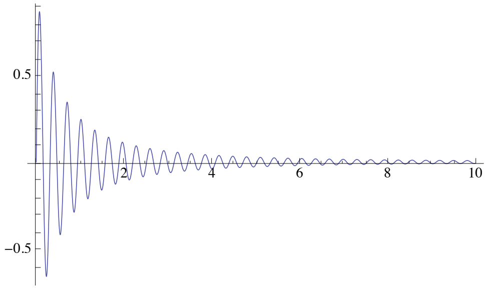
Here is a definition of an oscillation function that will be integrated in the interval $[0, \infty)$. The zeros of the oscillating function $\sin(\omega x)$ are $i \frac{1}{\omega} \pi,\ i \in \mathbb{N}$.

```
In[1]:=  Clear[ω, k, f];
         ω = 20;
         k[x_] := Sin[ω x];
                      1
         f[x_] := ----------- ;
                  (x + 1) ^ 2
```

Here is a plot of the oscillatory function in the interval $[0, 10]$.

```
In[89]:=  Plot[k[x] f[x], {x, 0, 10}, PlotPoints -> 1000, PlotRange -> All]
```

Out[89]=



This is a definition of a function that integrates between two consequent zeros. The zeros of the oscillating function $k(x) = \sin(\omega x)$ are $i \frac{1}{\omega} \pi,\ i \in \mathbb{N}$.

```
                                                    1                1
In[5]:=  psum[i_?NumberQ] := NIntegrate[k[x] f[x], {x, i — π, (i + 1) — π}]
                                                    ω                ω
```

Here is the integral estimate computed by sequence convergence acceleration (extrapolation).

```
In[6]:=  res = NSum[psum[i], {i, 0, ∞},
           Method → "AlternatingSigns", "VerifyConvergence" -> False]
Out[6]= 0.0492841
```

Here is the exact integral value.

```
In[7]:=  exact = Integrate[k[x] f[x], {x, 0, ∞}] // N
Out[7]= 0.0492841
```

The integral estimate is very close to the exact value.

In[8]:= $\dfrac{\texttt{Abs[exact - res]}}{\texttt{Abs[exact]}}$

Out[8]= 2.25444×10^{-7}

Here is another check using the "ExtrapolatingOscillatory" strategy.

```
In[94]:=  resEO = NIntegrate[Sin[ω x] f[x], {x, 0, ∞}, Method → "ExtrapolatingOscillatory"]
Out[94]= 0.0492841
```

The integral estimate by "ExtrapolatingOscillatory" is very close to the exact value.

In[95]:= $\dfrac{\texttt{Abs[exact - resEO]}}{\texttt{Abs[exact]}}$

Out[95]= 2.23802×10^{-7}

Double-Exponential Oscillatory Integration

The strategy "DoubleExponentialOscillatory" is for slowly decaying oscillatory integrals over one-dimensional infinite regions that have integrands of the form $\sin(\omega x^p + c) f(x)$, $\cos(\omega x^p + c) f(x)$, or $e^{i\omega x^p} f(x)$, where x is the integration variable, and ω, c, p are constants.

Integration with "DoubleExponentialOscillatory".

```
In[2]:=  NIntegrate[Sin[2 * x] * (1 / x^2), {x, 1, ∞},
          Method → {"DoubleExponentialOscillatory", "SymbolicProcessing" → 0}]
Out[2]= 0.0633358
```

option name	default value	
`Method`	`None`	integration strategy which will be used if `"DoubleExponentialOscillatory"` fails
`"TuningParameters"`	`Automatic`	tuning parameters of the error estimation
`"SymbolicProcessing"`	`Automatic`	number of seconds to do symbolic processing

Options of `"DoubleExponentialOscillatory"`.

`"DoubleExponentialOscillatory"` is based on the strategy `"DoubleExponential"`, but instead of using a transformation that reaches double-exponentially the ends of the integration interval `"DoubleExponentialOscillatory"` uses a transformation that reaches double-exponentially the zeros of $\sin(\omega x^p + c)$ and $\cos(\omega x^p + c)$. The theoretical foundations and properties of the algorithm are explained in [OouraMori91], [OouraMori99], [MoriOoura93]. The implementation of `"DoubleExponentialOscillatory"` uses the formulas and the integrator design in [OouraMori99].

The algorithm of `"DoubleExponentialOscillatory"` will be explained using the sine integral

$$I_s = \int_0^\infty f(x) \sin(\omega x)\, dx. \tag{19}$$

Consider the following transformation

$$x = \frac{M \phi(t)}{\omega},\ \phi(t) = \frac{t}{1 - e^{-2t - \beta(e^t - 1) - \alpha(1 - e^{-t})}}, \tag{20}$$

where α and β are constants satisfying

$$\beta = O(1),\ \alpha = o\left(\frac{1}{\sqrt{M \log M}}\right), 0 \le \alpha \le \beta \le 1.$$

The parameters α and β are chosen to satisfy

$$\alpha = \beta \Big/ \sqrt{1 + \frac{M \log(M+1)}{4\pi}}\ , \beta = \frac{1}{4} \tag{21}$$

(taken from [OouraMori99]).

Transformation (20) is applied to (19) to obtain

$$I_s = \int_0^\infty f\,(M\,(\phi\,t))\sin\,(M\,(\phi\,t))\,M\,(\phi'\,t)/\omega\,dt. \qquad (22)$$

Note that ω disappeared in the sine term. The trapezoidal formula with equal mesh size h applied to (22) gives

$$\mathrm{DEO}(I_s,\,h) = M\,h\sum_{n=-\infty}^{\infty} f\,(M\,(\phi\,(n\,h)/\omega)\sin\,(M\,\phi\,(n\,h))\,\phi'\,(n\,h)/\omega,$$

which is approximated with the truncated series sum

$$\mathrm{DEO}(I_s,\,h,\,N) = M\,h\sum_{n=-N_-}^{N_+} f\,(M\,(\phi\,(n\,h)/\omega)\sin\,(M\,\phi\,(n\,h))\,\phi'\,(n\,h)/\omega,\;\; N = N_- + N_+ + 1. \qquad (23)$$

M and h are chosen to satisfy

$$M\,h = \pi.$$

The integrand decays double-exponentially at large negative n as can be seen from (20). While the double-exponential transformation, (12) in the section "Double-Exponential Strategy", also makes the integrand decay double-exponentially at large positive t, the transformation (20) does not decay the integrand at large positive t. Instead it makes the sampling points approach double-exponentially to the zeros of $\sin(\omega\,x)$ at large positive t. Moreover

$$\sin(M\,\phi(n\,h)) \simeq \sin(M\,n\,h) = \sin(n\,\pi) = 0.$$

As is explained in [OouraMori99], since $\sin\,(\omega\,x)$ is linear near any of its zeros, the integrand decreases double-exponentially as x approaches a zero of $\sin\,(\omega\,x)$. This is the sense in which (23) is considered a double-exponential formula.

The relative error is assumed to satisfy

$$\varepsilon_M = \frac{|I_s - \mathrm{DEO}(I_s,h,N)|}{|I_s|} \simeq e^{-\frac{A}{h}} \simeq e^{-\frac{A\,M}{\pi}}. \qquad (24)$$

In [OouraMori99] the suggested value for A is 5.

Since the $\mathrm{DEO}(I_s, h, N)$ formulas cannot be made progressive, `"DoubleExponentialOscillatory"` (as proposed in [OouraMori99]) does between 2 and 4 integration estimates with different h. If the desired relative error is ε the integration steps are the following:

1. Choose $M = M_1$ such that

$$M_1 = -\frac{\pi \log\left(\sqrt{\varepsilon}\right)}{A},$$

and compute (23) with $M = M_1$. Let the result be I_{M_1}.

2. Next, set $M_2 = 2\,M_1$, and compute (23) with $M = M_2$. Let the result be I_{M_2}. The relative error of the first integration step $\varepsilon_{M_1} = \frac{\left|I_s - I_{M_1}\right|}{|I_s|}$ is assumed to be $\varepsilon_{M_1} \simeq \frac{\left|I_{M_2} - I_{M_1}\right|}{\left|I_{M_2}\right|}$. From (24) $\varepsilon_{M_2} \simeq {\varepsilon_{M_1}}^2$, and therefore, if

$$\varepsilon \geq \sigma \left(\frac{\left|I_{M_2} - I_{M_1}\right|}{\left|I_{M_2}\right|} \right)^2 \tag{25}$$

is satisfied, where σ is a robustness factor (by default 10) `"DoubleExponentialOscillatory"` exits with result I_{M_2}.

3. If (25) does not hold, compute

$$M_3 = M_2 \frac{\log \varepsilon}{\log\left(\sigma \left(\frac{\left|I_{M_2} - I_{M_1}\right|}{\left|I_{M_2}\right|} \right)^2 \right)}$$

and compute (23) with $M = M_3$. If

$$\varepsilon \geq \sigma \left(\frac{\left|I_{M_3} - I_{M_2}\right|}{\left|I_{M_3}\right|} \right)^{M_3/M_2} \tag{26}$$

`"DoubleExponentialOscillatory"` exits with result I_{M_3}.

4. If (26) does not hold, compute

$$M_4 = M_3 \frac{\log \varepsilon}{\log\left(\sigma\left(\frac{\left|I_{M_3} - I_{M_2}\right|}{\left|I_{M_3}\right|}\right)^{M_3/M_2}\right)}$$

and compute (23) with $M = M_4$. Let the result be I_{M_4}. If

$$\varepsilon \geq \sigma\left(\frac{\left|I_{M_4} - I_{M_3}\right|}{\left|I_{M_4}\right|}\right)^{M_4/M_3} \qquad (27)$$

does not hold, "DoubleExponentialOscillatory" issues the message NIntegrate::deoncon. If the value of the "DoubleExponentialOscillatory" method option is None, then I_{M_4} is returned. Otherwise "DoubleExponentialOscillatory" will return the result of NIntegrate called with the "DoubleExponentialOscillatory" method option.

For the cosine integral

$$I_c = \int_0^\infty f(x) \cos(\omega x)\, dx, \qquad (28)$$

the transformation corresponding to (20) is

$$x = M\, \phi\left(t - \frac{\pi}{2M}\right) \Big/ \omega.$$

Generalized Integrals

Here is the symbolic computation of the regularized divergent integral $\int_0^\infty \log(x)\sin(x)\, dx$.

```
In[110]:= exact =
            Limit[Integrate[Exp[-c x] Log[x] Sin[x], {x, 0, ∞}, Assumptions → c > 0], c → 0]

Out[110]= -EulerGamma
```

"DoubleExponentialOscillatory" computes the nonregularized integral above in a generalized sense.

```
In[111]:= NIntegrate[Log[x] Sin[x], {x, 0, ∞}] - exact

Out[111]= 4.89975×10^-12
```

More about the properties of "DoubleExponentialOscillatory" for divergent Fourier type integrals can found in [MoriOoura93].

Non-algebraic Multiplicand

Symbolic integration of an oscillatory integral.

In[116]:=
$$\texttt{exact = Integrate}\left[\texttt{Sin[20 x] Cos[18 x]}\,\frac{1}{\sqrt{\texttt{x + 1}}},\ \texttt{\{x, 0, }\infty\texttt{\}}\right]$$

Out[116]=
$$\frac{1}{12}\left(3\,\pi\left(\texttt{BesselJ}\left[-\frac{1}{2},\,2\right]+\texttt{BesselJ}\left[-\frac{1}{2},\,38\right]-\texttt{BesselJ}\left[\frac{1}{2},\,2\right]-\texttt{BesselJ}\left[\frac{1}{2},\,38\right]\right)+\right.$$
$$\left.16\left(19\,\texttt{HypergeometricPFQ}\left[\{1\},\,\left\{\frac{5}{4},\,\frac{7}{4}\right\},\,-361\right]+\texttt{HypergeometricPFQ}\left[\{1\},\,\left\{\frac{5}{4},\,\frac{7}{4}\right\},\,-1\right]\right)\right)$$

If the oscillatory kernel is multiplied by a nonalgebraic function, `"DoubleExponentialOscillatory"` still gives a good result.

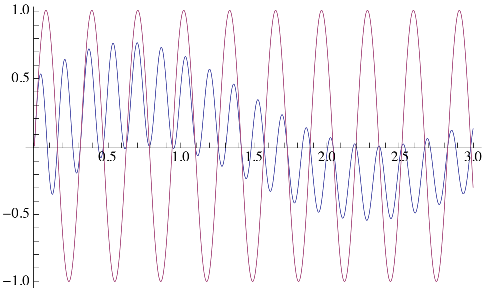
In[117]:=
$$\texttt{NIntegrate}\left[\texttt{Sin[20 x] Cos[18 x]}\,\frac{1}{\sqrt{\texttt{x + 1}}},\ \texttt{\{x, 0, }\infty\texttt{\}, PrecisionGoal}\to\texttt{10}\right]-\texttt{exact}$$

Out[117]= -1.92081×10^{-9}

Plots of the integrand and its oscillatory kernel.

In[119]:=
$$\texttt{Plot}\left[\left\{\texttt{Sin[20 x] Cos[18 x]}\,\frac{1}{\sqrt{\texttt{x + 1}}},\ \texttt{Sin[20 x]}\right\},\ \texttt{\{x, 0, 3\}}\right]$$

Out[119]=



Crude Monte Carlo and Quasi Monte Carlo Strategies

The crude Monte Carlo algorithm estimates a given integral by averaging integrand values over uniformly distributed random points in the integral's region. The number of points is incremented until the estimated standard deviation is small enough to satisfy the specified precision or accuracy goals. A Monte Carlo algorithm is called a quasi Monte Carlo algorithm if it uses equidistributed, deterministically generated sequences of points instead of uniformly distributed random points.

Here is a crude Monte Carlo integration.

```
In[3]:=  NIntegrate[ⅇ^-(x^4+y^4), {x, -2, 2}, {y, -2, 2}, Method → "MonteCarlo"]
Out[3]=  3.29043
```

Here is a crude quasi Monte Carlo integration.

```
In[4]:=  NIntegrate[ⅇ^-(x^4+y^4), {x, -2, 2}, {y, -2, 2}, Method → "QuasiMonteCarlo"]
Out[4]=  3.28632
```

option name	*default value*	
Method	"MonteCarloRule"	Monte Carlo rule specification
MaxPoints	50 000	maximum number of sampling points
"RandomSeed"	Automatic	a seed to reset the random generator
"Partitioning"	1	partitioning of the integration region along each axis
"SymbolicProcessing"	0	number of seconds to do symbolic preprocessing

"MonteCarlo" options.

option name	*default value*	
MaxPoints	50 000	maximum number of sampling points
"Partitioning"	1	partitioning of the integration region along each axis
"SymbolicProcessing"	0	number of seconds to do symbolic preprocessing

"QuasiMonteCarlo" options.

In Monte Carlo methods [KrUeb98] the d-dimensional integral $\int_V f(x)\, dx$ is interpreted as the following expected (mean) value:

$$\int_V f(x)\, dx = \mathrm{vol}(V) \int \frac{1}{\mathrm{vol}(V)} \texttt{Boole}\,(x \in V)\, f(x)\, dx = \mathrm{vol}(V)\, E(f), \tag{29}$$

where $E(f)$ is the mean value (the expectation) of the function f interpreted as a random variable, with respect to the uniform distribution on V, that is, the distribution with probability density $\mathrm{vol}\,(V)^{-1}\,\texttt{Boole}\,(x \in V)$. $\texttt{Boole}\,(x \in V)$ is denotes the characteristic function of the region V, while $\mathrm{vol}(V)$ denotes the volume of V.

The crude Monte Carlo estimate is made with the integration rule `"MonteCarloRule"`. The formulas for the integral and error estimation are given in the section "MonteCarloRule" in the tutorial "NIntegrate Integration Rules".

Consider the integral

$$\int_{\Omega} f(x)\, dx.$$

If the original integration region Ω is partitioned into the set of disjoint subregions $\{\Omega_i\}_{i=1}^{m}$, $\Omega = \bigcup_{i=1}^{m} \Omega_i$, then the integral estimate is

$$\sum_{i=1}^{m} \text{MC}(f, n_i),$$

and integration error is

$$\sum_{i=1}^{m} \text{SD}(f, n_i).$$

The number of sampling points used on each subregion generally can be different, but in the Monte Carlo algorithms all n_i are equal ($n_1 = n_2 = \ldots = n_m$).

The partitioning $\Omega = \bigcup_{i=1}^{m} \Omega_i$ is called stratification, and each Ω_i is called strata. Stratification can be used to improve crude Monte Carlo estimations. (The adaptive Monte Carlo algorithm uses recursive stratification.)

AccuracyGoal and PrecisionGoal

The default values for `AccuracyGoal` and `PrecisionGoal` are `Infinity` and 2 respectively when `NIntegrate`'s Monte Carlo algorithms are used.

MaxPoints

The option `MaxPoints` specifies what is the maximum number of (pseudo) random sampling points to be used to compute the Monte Carlo estimate of an integral.

Here is an example in which the maximum number of sampling points is reached and `NIntegrate` stops with a message.

```
                       1                                                         3
In[261]:= NIntegrate[ ——— , {x, 0.01, 1}, Method → {"MonteCarlo", "MaxPoints" → 10 }]
                      √ x

        NIntegrate::maxp: The integral failed to converge after 1100 integrand evaluations. NIntegrate obtained
              1.768394116870677` and 0.03357978772002253` for the integral and error estimates.

Out[261]= 1.76839
```

"RandomSeed"

The value of the option `"RandomSeed"` is used to seed the random generator used to make the sampling integration points. In that respect the use `"RandomSeed"` in Monte Carlo method is similar to the use of `SeedRandom` and `RandomReal`.

By using `"RandomSeed"` the results of a Monte Carlo integration can be reproduced. The results of the following two runs are identical.

Here is a Monte Carlo integration that uses `"RandomSeed"`.

```
                      1
In[56]:= NIntegrate[ ——— , {x, 0.01, 1},
                     √ x
           Method → {"MonteCarlo", "RandomSeed" → 12}] // InputForm

Out[56]//InputForm= 1.7828815270494558
```

This Monte Carlo integration gives the same number.

```
                      1
In[57]:= NIntegrate[ ——— , {x, 0.01, 1},
                     √ x
           Method → {"MonteCarlo", "RandomSeed" → 12}] // InputForm

Out[57]//InputForm= 1.7828815270494558
```

The following shows the first 20 points used in the Monte Carlo integrations.

```
In[65]:= pnts =
           Reap[NIntegrate[1/√x, {x, 0.01, 1}, Method → {"MonteCarlo", "RandomSeed" → 12},
               EvaluationMonitor :→ Sow[x]]]⟦2, 1⟧;
         Take[
          pnts,
          20]
Out[66]= {0.149394, 0.0460797, 0.526197, 0.402254, 0.249858, 0.155351,
          0.75201, 0.447633, 0.826597, 0.899822, 0.672286, 0.322249, 0.737047,
          0.162606, 0.53339, 0.12339, 0.36747, 0.095921, 0.83827, 0.16102}
```

The points coincide with the points made using `SeedRandom` and `Random`.

```
In[67]:= SeedRandom[12]; RandomReal[{0.01, 1}, 20]
Out[67]= {0.149394, 0.0460797, 0.526197, 0.402254, 0.249858, 0.155351,
          0.75201, 0.447633, 0.826597, 0.899822, 0.672286, 0.322249, 0.737047,
          0.162606, 0.53339, 0.12339, 0.36747, 0.095921, 0.83827, 0.16102}
```

Stratified Crude Monte Carlo Integration

In stratified sampling Monte Carlo integration you break the region into several subregions and apply the crude Monte Carlo estimate on each subregion separately.

From the expected (mean) value formula, Equation (29) at the beginning of Crude Monte Carlo and Quasi Monte Carlo Strategies, you have

$$E(f) = \frac{1}{\mathrm{vol}(V)} \int_V f(x)\, dx. \tag{30}$$

Let the region V be bisected into two half-regions, V_1 and V_2. $E_i(f)$ is the expectation of f on V_i, and $\mathrm{Var}_i(f)$ is the variance of f on V_i. From the theorem [PrFlTeuk92]

$$\mathrm{Var}(f) = \frac{1}{4}(E_1(f) - E_2(f))^2 + \frac{1}{2}(\mathrm{Var}_1(f) + \mathrm{Var}_2(f)) \tag{31}$$

you can see that the stratified sampling gives a variance that is never larger than the crude Monte Carlo sampling variance.

There are two ways to specify strata for the `"MonteCarlo"` strategy. One is to specify "singular" points in the variable range specifications, the other is to use the method sub-option `"Partitioning"`.

Stratified crude Monte Carlo integration using variable ranges specifications.

```
In[124]:= NIntegrate[x^2 + y^2, {x, 0, 1 / 3, 2 / 3, 1},
            {y, 0, 1 / 3, 2 / 4, 3 / 4, 1}, Method → "MonteCarlo"]
Out[124]= 0.666398
```

Stratified crude Monte Carlo integration using the sub-option "`Partitioning`".

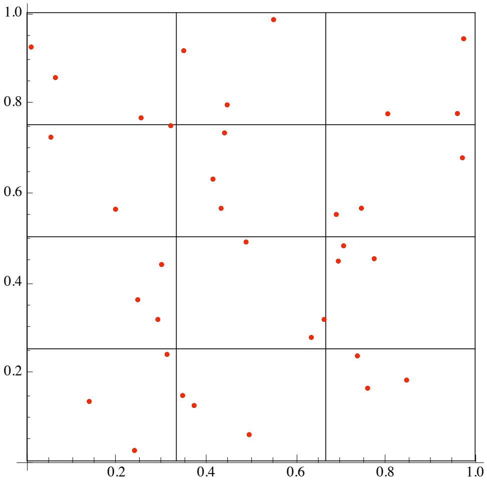
```
In[123]:= NIntegrate[x^2 + y^2, {x, 0,  1}, {y, 0, 1},
           Method → {"MonteCarlo", "Partitioning" → {3, 4}}]
Out[123]= 0.671852
```

If "`Partitioning`" is given a list of integers, $\{p_1, p_2, \ldots, p_n\}$ with length n that equals the number of integral variables, each dimension i of the integration region is divided into p_i equal parts. If "`Partitioning`" is given an integer p, all dimensions are divided into p equal parts.

This graph demonstrates the stratified sampling specified with "`Partitioning`". Each cell contains 3 points, as specified by the "`MonteCarloRule`" option "`Points`".

```
In[95]:= parts = {3, 4};
         t = Reap[NIntegrate[1, {x, 0, 1}, {y, 0, 1}, Method → {"MonteCarlo",
                  "Partitioning" → parts, Method → {"MonteCarloRule", "Points" → 3}},
               EvaluationMonitor :→ Sow[{x, y}]]]⟦2, 1⟧;
         grX = (Line[{{#1, 0}, {#1, 1}}] &) /@ Table[i, {i, 0, 1, 1/parts⟦1⟧}];
         grY = (Line[{{0, #1}, {1, #1}}] &) /@ Table[i, {i, 0, 1, 1/parts⟦2⟧}];
         grLP = Point /@ t;
         Graphics[{grLP, grX, grY, Red, grLP}, Axes -> True]
```

Out[100]=

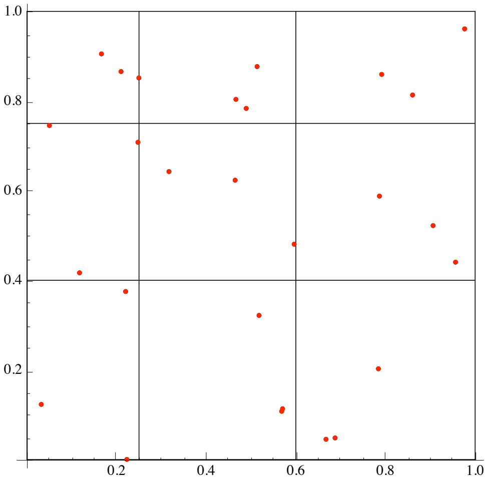


Stratified Monte Carlo sampling can be specified if the integration variable ranges are given with intermediate singular points.

Stratified Monte Carlo sampling through specification of intermediate singular points.

```
In[18]:= ranges = {{x, 0, 1/4, 3/5, 1}, {y, 0, 2/5, 3/4, 1}};
         t = Reap[NIntegrate[1, Evaluate[Sequence @@ ranges],
               Method → {"MonteCarlo", Method → {"MonteCarloRule", "Points" → 3}},
               EvaluationMonitor :→ Sow[{x, y}]]][[2, 1]];
         grX = Line[{{#1, 0}, {#1, 1}}] & /@ Rest@ranges[[1]];
         grY = Line[{{0, #1}, {1, #1}}] & /@ Rest@ranges[[2]];
         grLP = Point /@ t;
         Graphics[{grLP, grX, grY, Red, grLP}, Axes -> True]
```

Out[23]=



Stratified sampling improves the efficiency of the crude Monte Carlo estimation: if the number of strata is s, the standard deviation of the stratified Monte Carlo estimation is s times less of the standard deviation of the crude Monte Carlo estimation. (See the following example.)

The following benchmark shows that stratification speeds up the convergence.

```
In[120]:= n = 10; res =
            Timing[Do[NIntegrate[(ⅇ^x - 1)/(ⅇ - 1), {x, 0, 1}, Method → {"MonteCarlo", "Partitioning" →
                     #1, "MaxPoints" → 10^6}, PrecisionGoal → 2], {n}]][[1]] / n & /@ Range[4];
          ColumnForm[
            res]
Out[121]= 0.0114982
          0.0039994
          0.0025996
          0.0020997
```

Convergence Speedup of the Stratified Monte Carlo Integration

The following example confirms that if the number of strata is s, the standard deviation of the stratified Monte Carlo estimation is s times less than the standard deviation of the crude Monte Carlo estimation.

Here is a stratified integration definition based on the expected (mean) value formula (29).

```
In[122]:= MonteCarloEstimate[f_, strata_, n_] :=
  ({#1⟦1⟧, √(#1⟦2⟧)} &)[Total[({Mean[f /@ #1]/strata, Variance[f /@ #1]/((strata^2 n)/strata)} &) /@
     Table[Random[Real, {(i - 1)/strata, i/strata}], {i, strata}, {n/strata}]]]
```

Consider this integral.

```
In[123]:= f[x_] := (ⅇ^x - 1)/(ⅇ - 1)
          N[∫_0^1 f[x] ⅆx]

Out[124]= 0.418023
```

Here the integral above is approximated with 1000 points for the number of strata running from 1 to 40.

```
In[125]:= t = Table[MonteCarloEstimate[f, i, 1000], {i, 1, 40}];
```

These are the ratios between the standard deviations and the nonstratified, crude Monte Carlo estimation.

```
In[126]:= ratios = Transpose[t]⟦2⟧ / Transpose[t]⟦2, 1⟧;
```

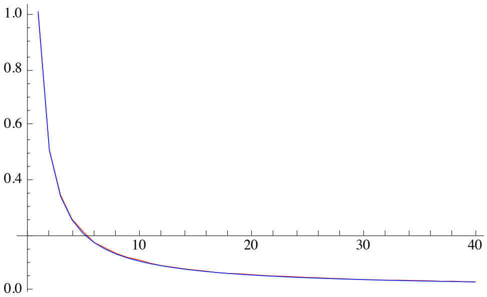
Note that $ratios[\![i]\!]$ is the ratio for the Monte Carlo estimation with i number of strata. This allows you to try a least squares fit of the function $\frac{1}{x}$ to $ratios$.

```
In[127]:= p[x_] := Evaluate@Fit[ratios, {1/x}, x]
          p[x]

Out[128]= 1.0075
          ------
            x
```

The fitting of $\frac{1}{x}$ shows a coefficient very close to 1, which is a confirmation of the rule of thumb that s number of strata give s-times faster convergence. This is the plot of the *ratios* and the $\frac{1}{x}$ least squares fit.

```
In[130]:= ListLinePlot[{ratios, p /@ Range[Length[ratios]]},
           PlotRange -> All, PlotStyle -> {{Red}, {Blue}}]
```

Out[130]=



Global Adaptive Monte Carlo and Quasi Monte Carlo Strategies

The global adaptive Monte Carlo and quasi Monte Carlo strategies recursively bisect the subregion with the largest variance estimate into two halves, and compute integral and variance estimates for each half.

Here is an example of adaptive Monte Carlo integration.

```
In[1]:= NIntegrate[ⅇ^-(x^4+y^4), {x, -π, π}, {y, -π, π}, Method → "AdaptiveMonteCarlo"]
Out[1]= 3.2531
```

option name	*default value*	
`Method`	`MonteCarloRule`	`MonteCarloRule` specification
`"Partitioning"`	`Automatic`	initial partitioning of the integration region along each axis
`"BisectionDithering"`	0	offset from the middle of the region side that is parallel to the bisection axis
`"MaxPoints"`	`Automatic`	maximum number of (pseudo-)random sampling points to be used
`"RandomSeed"`	`Automatic`	random seed used to generate the (pseudo-)random sampling points

Adaptive (quasi) Monte Carlo uses crude (quasi) Monte Carlo estimation rule on each subregion.

The process of subregion bisection and subsequent bi-integration is expected to reduce the global variance, and it is referred to as recursive stratified sampling. It is motivated by a theorem that states that if a region is partitioned into disjoint subregions the random variable variance over the region is greater than or equal to the sum of the random variable variances over each subregion. (See "Stratified Monte Carlo Integration" in the section "Crude Monte Carlo and Quasi Monte Carlo Strategies".)

The global adaptive Monte Carlo strategy `"AdaptiveMonteCarlo"` is similar to `"GlobalAdaptive"`. There are some important differences though.

1. `"AdaptiveMonteCarlo"` does not use singularity flattening, and does not have detectors for slow convergence and noisy integration.
2. `"AdaptiveMonteCarlo"` chooses randomly the bisection dimension. To avoid irregular separation of different coordinates a dimension recurs only if other dimensions have been chosen for bisection.
3. `"AdaptiveMonteCarlo"` can be tuned to bisect the subregions away from the middle. More at "BisectionDithering".

MinRecursion and MaxRecursion

The options `MinRecursion` and `MaxRecursion` for the adaptive Monte Carlo methods have the same meaning and functionality as they do for `"GlobalAdaptive"`. See "MinRecursion and MaxRecursion".

"Partitioning"

The option `"Partitioning"` of `"AdaptiveMonteCarlo"` provides initial stratification of the integration. It has the same meaning and functionality as `"Partitioning"` of the strategy `"MonteCarlo"`.

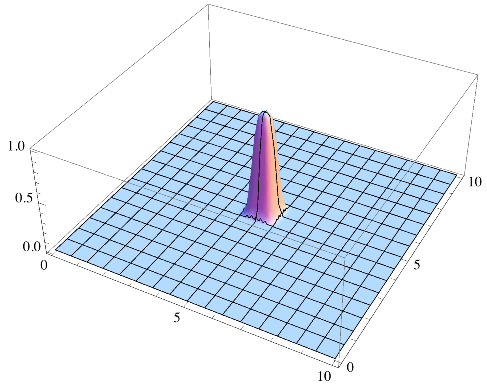
"BisectionDithering"

When the integrand has some special symmetry that puts significant parts of it in the middle of the region, it is better if the bisection is done slightly away from the middle. The value of the option `"BisectionDithering"` -> *dith* specifies that the splitting fraction of the region's splitting dimension side should be at $\frac{1}{2} \pm dith$ instead of $\frac{1}{2}$. The sign of *dith* is changed in a random manner. The default value given to `"BisectionDithering"` is $\frac{1}{10}$. The allowed values for *dith* are reals in the interval $\left[-\frac{1}{4}, \frac{1}{4}\right]$.

Consider the function.

```
In[195]:= f[x_, y_] := ⅇ^(- 30 ((x-5)^4+(y-5)^4));
          Plot3D[f[x, y], {x, 0, 10}, {y, 0, 10}, PlotPoints → 30, PlotRange → All]
```

Out[196]=



Consider the integral.

```
In[197]:= Integrate[f[x, y], {x, 0, 10}, {y, 0, 10}]
          % // N
```

Out[197]=

$$\frac{\left(-4\,\text{Gamma}\left[\frac{5}{4}\right]+\text{Gamma}\left[\frac{1}{4},\,18\,750\right]\right)^2}{4\sqrt{30}}$$

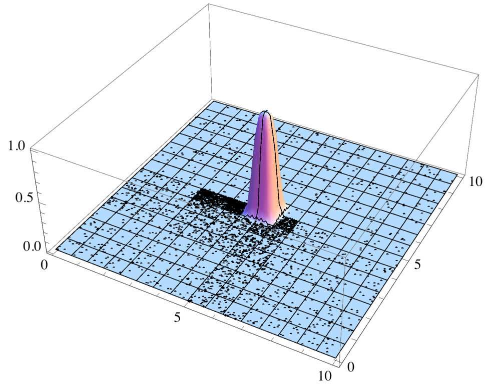
```
Out[198]= 0.599987
```

The integral is seriously underestimated if no bisection dithering is used i.e., "BisectionDithering" is given 0.

```
In[199]:= Mean@Table[NIntegrate[f[x, y], {x, 0, 10}, {y, 0, 10},
             Method → {"AdaptiveMonteCarlo", "BisectionDithering" → 0}], {20}]

Out[199]= 0.40383
```

The following picture shows why the integral is underestimated. The black points are the integration sampling points. It can be seen that half of the peak of the integrand is undersampled.

```
In[204]:= t = Reap[NIntegrate[f[x, y], {x, 0, 10}, {y, 0, 10},
               Method → {"AdaptiveMonteCarlo", "BisectionDithering" → 0, "RandomSeed" → 10},
               PrecisionGoal → 2, EvaluationMonitor ⧴ Sow[{x, y, 0}]]];
          Print["Integral value ", t⟦1⟧]
          cp = Plot3D[f[x, y], {x, 0, 10}, {y, 0, 10}, PlotPoints → 30, PlotRange → All];
          Graphics3D[{cp⟦1⟧, PointSize[0.006], Point /@ t⟦2, 1⟧},
           BoxRatios → {1, 1, 0.4}, PlotRange → All, Axes -> True]

   Integral value 0.292876

Out[207]=
```



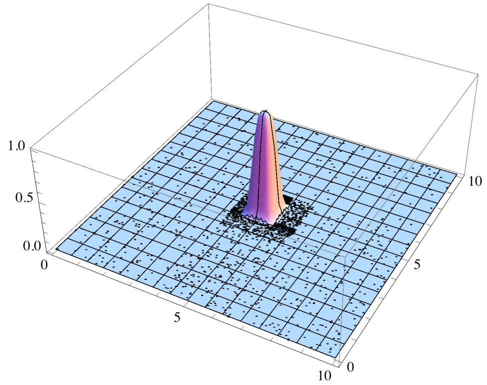
Specifying bisection dithering of $10\,\%$ gives a satisfactory estimation.

```
In[212]:= Mean@Table[NIntegrate[f[x, y], {x, 0, 10}, {y, 0, 10},
             Method → {"AdaptiveMonteCarlo", "BisectionDithering" -> 1/10}], {30}]

Out[212]= 0.596772
```

It can be seen on this plot, that the peak of the integrand is sampled better.

```
In[213]:= t = Reap[NIntegrate[f[x, y], {x, 0, 10}, {y, 0, 10},
              Method → {"AdaptiveMonteCarlo", "BisectionDithering" → 1/10, RandomSeed → 10},
              PrecisionGoal → 2, EvaluationMonitor :→ Sow[{x, y, 0}]]];
          Print["Integral value ", t⟦1⟧]
          cp = Plot3D[f[x, y], {x, 0, 10}, {y, 0, 10}, PlotPoints → 30, PlotRange → All];
          Graphics3D[{cp⟦1⟧, PointSize[0.006], Point /@ t⟦2, 1⟧},
           BoxRatios → {1, 1, 0.4}, PlotRange → All, Axes -> True]

   Integral value 0.610217
```

Out[216]=



Choice of Bisection Axis

For multidimensional integrals the adaptive Monte Carlo algorithm chooses the splitting axis of an integration region in two ways: (i) by random selection or (ii) by minimizing the variance of the integral estimates of each half. The axis selection is a responsibility of the "MonteCarloRule".

Example: Comparison with Crude Monte Carlo

Generally, the "AdaptiveMonteCarlo" strategy has greater performance than "MonteCarlo". This is demonstrated with the examples in this subsection.

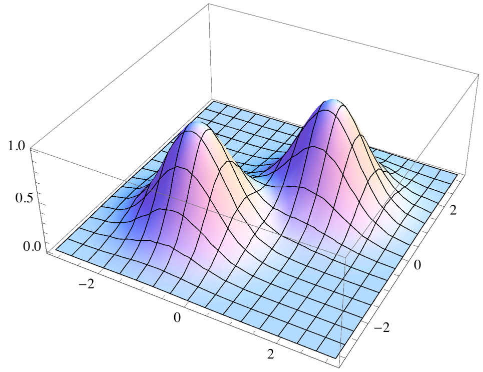
Consider the function.

```
In[217]:=  f[x_, y_] := ⅇ^-((x+1)^2+(y+1)^2) + ⅇ^-((x-1)^2+(y-1)^2)
```

This is a plot of the function.

```
In[218]:=  Plot3D[f[x, y], {x, -π, π}, {y, -π, π}]
```

Out[218]=



It can be seen from the following profiling that "AdaptiveMonteCarlo" uses nearly three times fewer sampling points than the crude "MonteCarlo" strategy.

These are the sampling points and timing for "MonteCarlo".

```
In[219]:=  {k = 0;
            (resMC = NIntegrate[f[x, y], {x, -π, π}, {y, -π, π}, Method → "MonteCarlo",
                  PrecisionGoal → 2, EvaluationMonitor :→ k++]) // Timing // First, k}
Out[219]=  {0.689894 Second, 22 500}
```

These are the sampling points and timing for "AdaptiveMonteCarlo".

```
In[220]:=  {k = 0;
            (resAMC =
                NIntegrate[f[x, y], {x, -π, π}, {y, -π, π}, Method → "AdaptiveMonteCarlo",
                  PrecisionGoal → 2, EvaluationMonitor :→ k++]) // Timing // First, k}
Out[220]=  {0.180972 Second, 5300}
```

This is the exact result.

```
In[221]:=  exact = Integrate[f[x, y], {x, -π, π}, {y, -π, π}]

           1
Out[221]=  — π (-Erf[1 - π] + Erf[1 + π]) (Erf[-1 + π] + Erf[1 + π])
           2
```

Here is the timing for 100 integrations with "MonteCarlo".

```
In[222]:= tblMC = Table[NIntegrate[f[x, y], {x, -π, π}, {y, -π, π},
                Method → "MonteCarlo", PrecisionGoal → 2], {100}]; // Timing
Out[222]= {11.8842 Second, Null}
```

The "MonteCarlo" integration compares well with the exact result. The numbers below show the error of the mean of the integral estimates, the mean of the relative errors of the integral estimates, and the variance of the integral estimates.

```
                                          tblMC - exact      (tblMC - exact).(tblMC - exact)
In[223]:= {Abs[Mean[tblMC] - exact], Mean[Abs[-------------]], -------------------------------}
                                              exact                  Length[tblMC]
Out[223]= {0.00137993, 0.00813663, 0.00430569}
```

Here is the timing for 100 integrations with "AdaptiveMonteCarlo", which is several times faster than "MonteCarlo" integrations.

```
In[233]:= tblAMC = Table[NIntegrate[f[x, y], {x, -π, π}, {y, -π, π},
                Method → "AdaptiveMonteCarlo", PrecisionGoal → 2], {100}]; // Timing
Out[233]= {4.21336 Second, Null}
```

The "AdaptiveMonteCarlo" integration result compares well with the exact result. The numbers below show the error of the mean of the integral estimates, the mean of the relative errors of the integral estimates, and the variance of the integral estimates.

```
In[234]:= {Abs[Mean[tblAMC] - exact],
               tblAMC - exact     (tblAMC - exact).(tblAMC - exact)
  Mean[Abs[--------------]], ---------------------------------}
                 exact                 Length[tblAMC]
Out[234]= {0.0129984, 0.00742212, 0.00366479}
```

"MultiPeriodic"

The strategy "MultiPeriodic" transforms all integrals into integrals over the unit cube and periodizes the integrands to be one-periodic with respect to each integration variable. Different periodizing functions (or none) can be applied to different variables. "MultiPeriodic" works for integrals with dimension less than or equal to twelve. If "MultiPeriodic" is given, integrals with higher dimension the "MonteCarlo" strategy is used.

Here is an example of integration with "MultiPeriodic".

```
In[2]:= NIntegrate[ⅇ^-(x1^4+x2^4+x3^4), {x1, -π, π},
         {x2, -π, π}, {x3, -π, π}, Method → "MultiPeriodic"]
Out[2]= 5.95735
```

option name	default value	
"Transformation"	SidiSin	periodizing transformation applied to the integrand
"MinPoints"	0	minimal number of sampling points
"MaxPoints"	10^5	maximum number of sampling points
"SymbolicProcessing"	Automatic	number of seconds to be used for symbolic preprocessing

"MultiPeriodic" can be seen as a multidimensional generalization of the strategy "Trapezoidal". It can also be seen as a quasi Monte Carlo method.

"MultiPeriodic" uses lattice integration rules; see [SloanJoe94] [KrUeb98].

Here integration lattice in $\mathbb{R}^d$, $d \in \mathbb{N}$, is understood to be a discrete subset of $\mathbb{R}^d$ which is closed under addition and subtraction, and which contains $\mathbb{Z}^d$. A lattice integration rule [SloanJoe94] is a rule of the form

$$Q\, f(x) = \frac{1}{N} \sum_{i=1}^{N} f(x_i),$$

where $\{x_1, x_2, \ldots, x_N\}$ are all the points of an integration lattice contained in $[0, 1)^n$.

If "MultiPeriodic" is called on, a d-dimensional integral option "Transformation" takes a list of one-argument functions $\{f_1, f_2, \ldots, f_d\}$ that is used to transform the corresponding variables. If "Transformation" is given a list with length l smaller than d, then the last function, f_l, is used for the last $d - l$ integration variables. If "Transformation" is given a function, that function will be used to transform all the variables.

Let d be the dimension of the integral. If $d = 1$ "MultiPeriodic" calls "Trapezoidal" after applying the periodizing transformation. For dimensions higher than 12 "MonteCarlo" is called without applying periodizing transformations. "MultiPeriodic" uses the so-called 2^d copy rules for $2 \leq d \leq 12$. For each $2 \leq d \leq 12$ "MultiPeriodic" has a set of copy rules that are used to compute a sequence of integral estimates. The rules with a smaller number of points are used first. If the error estimate of a rule satisfies the precision goal, or if the difference of two integral estimates in the sequence satisfies the precision goal, the integration stops.

Number of points for the 2^d copy rules in the rule sets for different dimensions.

```
In[3]:= tbl = (First /@ # &) /@ Rest[NIntegrate`MultiPeriodicDump`copyrules];
        tbl = MapIndexed[#1 * 2^(#2[[1]] + 1) &, tbl];
        mlen = Max[Length /@ tbl];
        tbl = Map[Join[#, Table["", {mlen - Length[#]}]] &, tbl];
        Style[TableForm[Transpose[tbl],
          TableHeadings → {Automatic, Range[2, Length[tbl] + 1]}], Small]
```

Out[7]=

	2	3	4	5	6	7	8	9	10	11	12
1	4996	4952	5008	5024	5056	5248	4864	5632	5120	6144	12 288
2	10 012	9992	9904	10 016	10 048	10 112	10 496	9728	11 264	10 240	20 480
3	20 012	20 024	19 984	19 808	20 032	20 096	20 224	20 992	19 456	22 528	45 056
4	40 028	40 024	40 048	39 968	39 616	40 064	40 192	40 448	41 984	38 912	77 824
5	80 044	80 056	80 048	80 096	79 936	79 232	80 128	80 384	80 896	83 968	167 936
6	160 036	160 088	160 112	160 096	160 192	159 872	158 464	160 256	160 768	161 792	323 584
7	320 084	320 072	320 176	320 224	320 192	320 384	319 744	316 928	320 512	321 536	643 072
8					640 448	640 384	640 768	639 488	633 856	641 024	1 282 048
9						1 280 896	1 280 768	1 281 536	1 278 976	1 267 712	2 535 424
10							2 561 792	2 561 536	2 563 072	2 557 952	5 115 904
11								5 123 584	5 123 072	5 126 144	10 252 288
12									10 247 168	10 246 144	20 492 288
13										20 494 336	40 988 672

Comparison with "MultiDimensionalRule"

Generally "MultiPeriodic" is slower than "GlobalAdaptive" using "MultiDimensionalRule". For computing high-dimension integrals with lower precision, "MultiPeriodic" might give results faster.

This defines the function of eight arguments.

```
In[8]:= f[x1_, x2_, x3_, x4_, x5_, x6_, x7_, x8_] :=
         1 / (1 + 0.9671190054385935` x1 + 0.21216802639809276` x2 +
             0.682779542171783` x3 + 0.32962509624641606` x4 + 0.5549215440908636` x5 +
             0.7907543870000786` x6 + 0.8580353669569777` x7 + 0.4796298578498076` x8)^9
```

Timing in seconds for computing $\int_0^1 \dots \int_0^1 f[x_1, \dots, x_8]\, dx_1 \dots dx_8$ using "MultiPeriodic" and "GlobalAdaptive" with "MultiDimensionalRule".

In[11]:=

```
tbl = Table[{"IntegralEstimate", "Evaluations", "Timing"} /.
     NIntegrateProfile[NIntegrate[f[x1, x2, x3, x4, x5, x6, x7, x8], {x1, 0, 1},
        {x2, 0, 1}, {x3, 0, 1}, {x4, 0, 1}, {x5, 0, 1}, {x6, 0, 1}, {x7, 0, 1},
        {x8, 0, 1}, Method → meth, MaxPoints → 10^8, PrecisionGoal → pg], 1], {pg, 1, 4},
    {meth, {"MultiPeriodic", {"MultiDimensionalRule", "Generators" → 5},
      {"MultiDimensionalRule", "Generators" → 9}}}];
TableForm[Map[#[[3]] &, tbl, {2}], TableHeadings →
  Map[Style[#, FontFamily → Times, FontSize → 11] &,
   {{"Precision goal → 1", "Precision goal → 2",
     "Precision goal → 3", "Precision goal → 4"}, {"MultiPeriodic",
     ColumnForm[{"MultiDimensionalRule", "with 5 generators"}], ColumnForm[
      {"MultiDimensionalRule", "with 9 generators"}]}}, {-1}], TableSpacing → 3]
```

Out[12]//TableForm=

```
                   | MultiPeriodic  MultiDimensionalRule  MultiDimensionalRule
                   |                with 5 generators     with 9 generators
-------------------+-----------------------------------------------------------
Precision goal → 1 | 0.12798        0.033995              0.356945
Precision goal → 2 | 0.418936       0.862869              11.6892
Precision goal → 3 | 8.09177        19.523                17.2124
Precision goal → 4 | 33.3839        476.796               18.9691
```

Number of integrand evaluations for computing $\int_0^1 \dots \int_0^1 f[x_1, \dots, x_8]\, dx_1 \dots dx_8$ using "MultiPeriodic" and "GlobalAdaptive" with "MultiDimensionalRule".

In[13]:=

```
TableForm[Map[#[[2]] &, tbl, {2}],
 TableHeadings → Map[Style[#, FontFamily → Times, FontSize → 11] &,
   {{"Precision goal → 1", "Precision goal → 2",
     "Precision goal → 3", "Precision goal → 4"}, {"MultiPeriodic",
     ColumnForm[{"MultiDimensionalRule", "with 5 generators"}], ColumnForm[
      {"MultiDimensionalRule", "with 9 generators"}]}}, {-1}], TableSpacing → 3]
```

Out[13]//TableForm=

```
                   | MultiPeriodic  MultiDimensionalRule  MultiDimensionalRule
                   |                with 5 generators     with 9 generators
-------------------+-----------------------------------------------------------
Precision goal → 1 | 4864           2807                  20 995
Precision goal → 2 | 15 360         68 571                922 675
Precision goal → 3 | 314 368        1 634 877             1 302 795
Precision goal → 4 | 1 274 880      37 678 361            1 358 045
```

Preprocessors

The capabilities of all strategies are extended through symbolic preprocessing of the integrals. The preprocessors can be seen as strategies that delegate integration to other strategies (preprocessors included).

"SymbolicPiecewiseSubdivision"

`"SymbolicPiecewiseSubdivision"` is a preprocessor that divides an integral with a piecewise integrand into integrals with disjoint integration regions on each of which the integrand is not piecewise.

option name	*default value*	
`Method`	`Automatic`	integration strategy or preprocessor to which the integration will be passed
`"ExpandSpecialPiecewise"`	`True`	which piecewise functions should be expanded
`TimeConstraint`	`5`	the maximum number of seconds for which the piecewise subdivision will be attempted
`"MaxPiecewiseCases"`	`100`	the maximum number of subregions the piecewise preprocessor can return
`"SymbolicProcessing"`	`Automatic`	number of seconds to do symbolic preprocessing

Options of `"SymbolicPiecewiseSubdivision"`.

As was mentioned at the beginning of the tutorial, `NIntegrate` is able to integrate simultaneously integrals with disjoint domains each having a different integrand. Hence, after the preprocessing with `"SymbolicPiecewiseSubdivision"` the integration continues in the same way as if, say, `NIntegrate` were given ranges with singularity specifications (which can be seen as specifying integrals with disjoint domains with the same integrand). For example, the strategy `"GlobalAdaptive"` tries to improve the integral estimate of the region with the largest error through bisection, and will choose that largest error region regardless of which integrand it corresponds to.

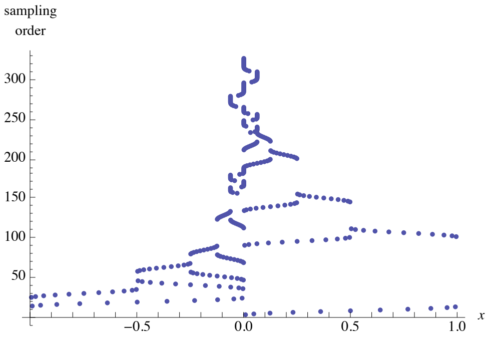
Below are the sampling points for the numerical estimation of the integral

$$\int_{-1}^{1} \begin{cases} \frac{2}{\sqrt{\sin(-x)}} & x < 0 \\ \frac{1}{\sqrt{x}} & x \geq 0 \end{cases} \, dx$$

On the plot, the integrand is sampled at the x coordinates in the order of the `ord` coordinates. It can be seen that "`GlobalAdaptive`" alternates sampling for the piece $\frac{2}{\sqrt{\sin(-x)}}, x < 0$ with sampling for the piece $\frac{1}{\sqrt{x}}, x \geq 0$.

```
In[12]:= pnts = Reap[NIntegrate[Piecewise[{{2/√Sin[-x], x < 0}, {1/√x, x ≥ 0}}],
            {x, -1, 1}, PrecisionGoal → 8, EvaluationMonitor :→ Sow[x]]][[2, 1]];
         ListPlot[Transpose[{pnts, Range[Length[pnts]]}], PlotRange → All,
          AxesOrigin → {-1, 0}, AxesLabel → {x, "sampling\norder"}]
```

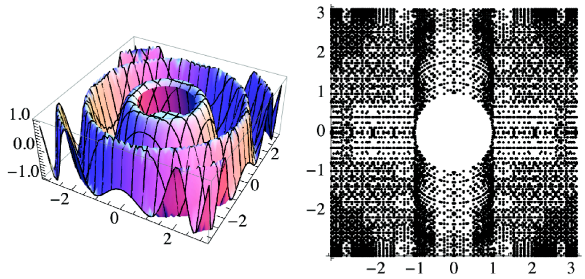
Out[13]=



Here are the sampling points for the numerical estimation of the integral $\int_{-\pi}^{\pi}\int_{-\pi}^{\pi}$`Boole` $[x^2 + y^2 > 1]\ \sin\ (x^2 + y^2)\ dy\, dx$. The integrand is plotted on the left, the sampling points are plotted on the right. The integral has been partitioned into $\int_{-\pi}^{-1}\int_{-\pi}^{\pi}\sin(x^2 + y^2)\,dy\,dx +$ $\int_{-1}^{1}\int_{-\pi}^{-\sqrt{1-x^2}}\sin(x^2 + y^2)\,dy\,dx + \int_{-1}^{1}\int_{\sqrt{1-x^2}}^{\pi}\sin(x^2 + y^2)\,dy\,dx + \int_{1}^{\pi}\int_{-\pi}^{\pi}\sin(x^2 + y^2)\,dy\,dx$, that is why the sampling points form a different pattern for $-1 \leq x \leq 1$.

```
In[14]:= gr = Plot3D[Boole[x² + y² > 1] Sin[x² + y²], {x, - π, π}, {y, -π, π}];
         grSP =
           Point /@ Reap[NIntegrate[Boole[x² + y² > 1] Sin[x² + y²], {x, - π, π}, {y, -π, π},
                Method → {"SymbolicPiecewiseSubdivision", Method → "GlobalAdaptive"},
                PrecisionGoal → 3, EvaluationMonitor :→ Sow[{x, y}]]][[2, 1]];
         grSP = Graphics[{PointSize[0.005], grSP}, Axes → True, AxesOrigin → {- π, - π}];
         GraphicsArray[{gr, grSP}]
```

In[17]:=



"ExpandSpecialPiecewise"

In some cases it is preferable to do piecewise expansion only over certain piecewise functions. In these case the option "ExpandSpecialPiecewise" can be given a list of functions to do the piecewise expansion with.

This Monte Carlo integral is done faster with piecewise expansion only over Boole.

```
In[18]:= f[x_, y_] :=
           Boole[x^2 + 2 y^2 < 1] Abs[x^2 + y^3 - 2] Abs[-x^2 + y^2 + 1] Abs[x^2 - 3 y^2 + x] 1/Sqrt[x^2 + y^2 + 10];
         NIntegrate[f[x, y], {x, -1, 1}, {y, -1, 1},
           Method → {"SymbolicPiecewiseSubdivision",
             "ExpandSpecialPiecewise" → {Boole}, Method → "MonteCarlo"}] // Timing
Out[19]= {0.108984, 0.634721}
```

Here is a Monte Carlo integration with piecewise expansion over both Boole and Abs.

```
In[20]:= NIntegrate[f[x, y], {x, -1, 1}, {y, -1, 1},
           Method → {"SymbolicPiecewiseSubdivision", Method → "MonteCarlo"}] // Timing
Out[20]= {0.19197, 0.625164}
```

"EvenOddSubdivision"

"EvenOddSubdivision" is a preprocessor that reduces the integration region if the region is symmetric around the origin and the integrand is determined to be even or odd. The convergence of odd integrals is verified by default.

option name	*default value*	
Method	Automatic	integration strategy or preprocessor to which the integration will be passed
VerifyConvergence	Automatic	should the convergence be verified if an odd integral is detected
"SymbolicProcessing"	Automatic	number of seconds to do symbolic preprocessing

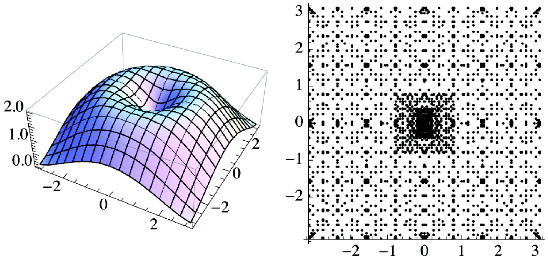
Options of "EvenOddSubdivision".

When the integrand is an even function and the integration region is symmetric around the origin, the integral can be computed by integrating only on some part of the integration region and multiplying with a corresponding factor.

Here is a plot of an even function and the sampling points without any preprocessing.

```
In[21]:= gr = Plot3D[1 + Sin[√(x^2 + y^2)], {x, -π, π}, {y, -π, π}];
         grSP = Point[
            Reap[NIntegrate[1 + Sin[√(x^2 + y^2)], {x, -π, π}, {y, -π, π}, Method → {Automatic,
                 "SymbolicProcessing" → 0}, EvaluationMonitor :→ Sow[{x, y}]]]⟦2, 1⟧];
         grSP = Graphics[{PointSize[0.01], grSP}, Axes → True, AxesOrigin → {-π, -π}];
         GraphicsGrid[{{gr, grSP}}]
```

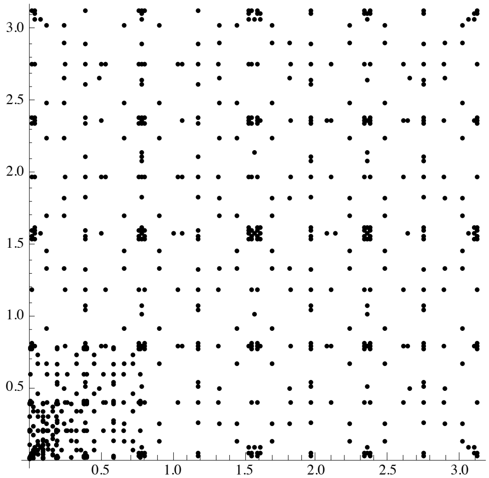
Out[24]=



These are the sampling points used by `NIntegrate` after `"EvenOddSubdivision"` has been applied. Note that the sampling points are only in the region $[0, \pi] \times [0, \pi]$.

```
In[25]:= Graphics[{PointSize[0.01],
           Point /@ Reap[NIntegrate[1 + Sin[√(x^2 + y^2)], {x, -π, π}, {y, -π, π},
                 EvaluationMonitor :→ Sow[{x, y}]]][[2, 1]]}, Axes → True]
```

Out[25]=



Transformation Theorem

The preprocessor `"EvenOddSubdivision"` is based on the following theorem.

Theorem: Given the d-dimensional integral

$$\int_{a_0}^{b_0} \cdots \int_{a_i(x_1,\ldots,x_{i-1})}^{b_i(x_1,\ldots,x_{i-1})} \cdots \int_{a_n(x_1,\ldots,x_{d-1})}^{b_n(x_1,\ldots,x_{d-1})} f(x_1, \ldots, x_d)\, dx_1 \ldots dx_d,$$

assume that for some $i \in \{1, 2, \ldots, d\}$ these equalities hold:

a) $a_i(x_1, \ldots, x_{i-1}) = -b_i(x_1, \ldots, x_{i-1})$,

b) for all $j > i,\ j \in \{1, 2, \ldots, d\}$:

$$a_j(x_1, \ldots, x_i, \ldots, x_{j-1}) = a_j(x_1, \ldots, -x_i, \ldots, x_{j-1}),$$
$$b_j(x_1, \ldots, x_i, \ldots, x_{j-1}) = b_j(x_1, \ldots, -x_i, \ldots, x_{j-1}).$$

In other words the range of x_i is symmetric around the origin, and the boundaries of the variables $x_j,\ j > i$ are even functions wrt x_i.

Then:

a) the integral is equivalent to

$$2 \int_{a_0}^{b_0} \cdots \int_{0}^{b_i(x_1,\ldots,x_i)} \cdots \int_{a_d(x_1,\ldots,x_{d-1})}^{b_d(x_1,\ldots,x_{d-1})} f(x_1, \ldots, x_d)\, dx_1 \ldots dx_d$$

if the integrand is even wrt x_i, that is,

$$f(x_1, \ldots, x_i, \ldots, x_d) = f(x_1, \ldots, -x_i, \ldots, x_d);$$

b) the integral is equivalent to 0, if the integrand is odd wrt x_i, that is,

$$f(x_1, \ldots, x_i, \ldots, x_d) = -f(x_1, \ldots, -x_i, \ldots, x_d).$$

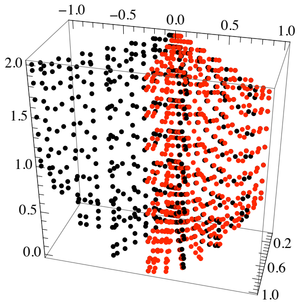
Note that the theorem above can be applied several times over an integral.

To illustrate the theorem consider the integral $\int_0^1 \int_{-x}^{x} \int_2^{y^2} x\, dz\, dy\, dx$. It is symmetric along y, and the integrand and the bounds of z are even functions wrt y.

Here is a plot of the sampling points without the application of "EvenOddSubdivision" (black) and with "EvenOddSubdivision" applied (red).

```
In[26]:= grEven = Point /@ Reap[NIntegrate[x, {x, 0, 1}, {y, -x, x}, {z, 2, y^2},
                Method → {"SymbolicPreprocessing", "UnitCubeRescaling" → False,
                  Method -> {"LobattoKronrodRule", "GaussPoints" → 5}},
                EvaluationMonitor ⧴ Sow[{x, y, z}]]]][[2, 1]];
         gr = Point /@ Reap[NIntegrate[x, {x, 0, 1}, {y, -x, x}, {z, 2, y^2},
                Method → {"LobattoKronrodRule", "GaussPoints" → 5, "SymbolicProcessing" → 0},
                EvaluationMonitor ⧴ Sow[{x, y, z}]]]][[2, 1]];
         Graphics3D[{gr, Red, grEven}, PlotRange → All, Axes → True,
          ViewPoint -> {2.813, 0.765, 1.718}]
```

Out[28]=

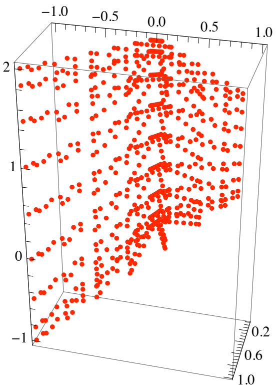


If the bounds of z are not even functions wrt y then the symmetry along y is broken. For example, the integral $\int_0^1 \int_{-x}^x \int_2^y x\,dz\,dy\,dx$ has no symmetry NIntegrate can exploit.

Here is a plot of the sampling points with "EvenOddSubdivision" applied (red). The region has no symmetry along y.

```
In[29]:= grEven = Point /@ Reap[NIntegrate[x, {x, 0, 1}, {y, -x, x}, {z, 2, y},
                Method → {"SymbolicPreprocessing", "UnitCubeRescaling" → False,
                  Method -> {"LobattoKronrodRule", "GaussPoints" → 5}},
                EvaluationMonitor ⧴ Sow[{x, y, z}]]]][[2, 1]];
         Graphics3D[{Red, grEven}, PlotRange → All, Axes → True,
          ViewPoint -> {2.813, 0.765, 1.718}]
```

Out[30]=



"VerifyConvergence"

Consider the following divergent integral $\int_{-\infty}^{\infty} x\, dx$. NIntegrate detects it as an odd function over a symmetric domain and tries to integrate $\int_{0}^{\infty} x\, dx$ (that is, check the convergence of $\int_{0}^{\infty} x\, dx$). Since no convergence was reached as is indicated by the ncvb message, NIntegrate gives the message oidiv that the integral might be divergent.

```
In[31]:= NIntegrate[x, {x, -∞, ∞}]

Out[31]= 0.
```

If the option VerifyConvergence is set to False no convergence verification—and hence no integrand evaluation—will be done after the integral is found to be odd.

```
In[32]:= NIntegrate[x, {x, -∞, ∞},
          Method → {"EvenOddSubdivision", "VerifyConvergence" → False}]

Out[32]= 0.
```

"OscillatorySelection"

"OscillatorySelection" is a preprocessor that selects specialized algorithms for efficient evaluation of one-dimensional oscillating integrals, the integrands of which are products of a trigonometric or Bessel function and a non-oscillating or a much slower oscillating function.

option name	*default value*	
"BesselInfiniteRangeMethod"	"ExtrapolatingOscillatory"	specialized integration algorithm for infinite region integrals with Bessel functions
"FourierFiniteRangeMethod"	Automatic	specialized integration algorithm for Fourier integrals over finite ranges
"FourierInfiniteRangeMethod"	{"DoubleExponentialOscillatory", Method->"ExtrapolatingOscillatory"}	specialized integration algorithm for Fourier integrals over infinite regions
Method	"GlobalAdaptive"	integration strategy or preprocessor to which the integration will be passed
"TermwiseOscillatory"	False	if the value of this option is True then the algorithm is selected for each term in a sum of oscillatory functions
"SymbolicProcessing"	Automatic	number of seconds to do symbolic processing

Options of "OscillatorySelection".

"OscillatorySelection" is used by default.

In[33]:= $$\texttt{NIntegrate}\left[\frac{\texttt{Sin[20 x]}}{\texttt{x + 1}}, \{\texttt{x}, 0, \infty\}\right]$$

```
Out[33]= 0.049757
```

Without the "OscillatorySelection" preprocessor NIntegrate does not reach convergence with its default option settings.

In[34]:= $$\texttt{NIntegrate}\left[\frac{\texttt{Sin[20 x]}}{\texttt{x + 1}}, \{\texttt{x}, 0, \infty\}, \texttt{Method} \to \{\texttt{Automatic}, \texttt{"SymbolicProcessing"} \to 0\}\right]$$

Out[34]= $0. \times 10^{2}$

The preprocessor "OscillatorySelection" is designed to work with the internal output of the "SymbolicPiecewiseSubdivision" preprocessor. "OscillatorySelection" itself partitions oscillatory integrals that include the origin or have oscillatory kernels that need to be expanded or transformed into forms for which the oscillatory algorithms are designed.

Here is a piecewise function integration in which all methods of "OscillatorySelection" are used. For this integral the preprocessor "SymbolicPiecewiseSubdivision" divides the integral into four different integrals; for each of these integrals "OscillatorySelection" selects an appropriate algorithm.

In[1]:= $$\texttt{NIntegrate}\left[\begin{cases} \frac{\texttt{BesselJ[3,-x]}}{\sqrt{\texttt{-x}}} & \texttt{x} < 0 \\ \frac{\texttt{Cos[200 x]}}{\sqrt{\texttt{x}}} & 0 < \texttt{x} < 20 \\ \frac{\texttt{Sin[2 x+3]}}{\texttt{x}^2} & \texttt{x} > 30 \\ \frac{1}{\texttt{Log[x]}} & \texttt{True} \end{cases}, \{\texttt{x}, -\infty, \infty\}\right]$$

```
Out[1]= 3.77933
```

The following table shows the names of the "OscillatorySelection" options used to specify the algorithms for each sub-interval in the integral above.

$x \in (-\infty, 0]$	"BesselInfiniteRangeMethod"
$x \in [0, 20]$	"FourierFiniteRangeMethod"
$x \in [30, \infty)$	"FourierInfiniteRangeMethod"
$x \in [20, 30]$	Method

In this example "DoubleExponentialOscillatory" is called twice. "DoubleExponentialOscillatory" is a special algorithm for Fourier integrals, and the formula $e^{2 i x^2} = \cos(2 x^2) + i \sin(2 x^2)$ makes the integrand a sum of two Fourier integrands.

```
In[35]:= NIntegrate[Exp[2 ⅈ x^2]/(x + 1), {x, 0, ∞}] // InputForm

Out[35]//InputForm= 0.39934219109623426 + 0.2791805912092563*I
```

To demonstrate that "OscillatorySelection" has used the formula $e^{2 i x^2} = \cos(2 x^2) + i \sin(2 x^2)$, here is the integral above split "by hand." The result is identical with the last result.

```
In[36]:= NIntegrate[Cos[2 x^2]/(x + 1), {x, 0, ∞}] + ⅈ NIntegrate[Sin[2 x^2]/(x + 1), {x, 0, ∞}] // InputForm

Out[36]//InputForm= 0.39934219109623426 + 0.2791805912092563*I
```

The value Automatic for the option "FourierFiniteRangeMethod" means that if the integration strategy specified with the option Method is one of "GlobalAdaptive" or "LocalAdaptive" then that strategy will be used for the finite range Fourier integration, otherwise "GlobalAdaptive" will be used.

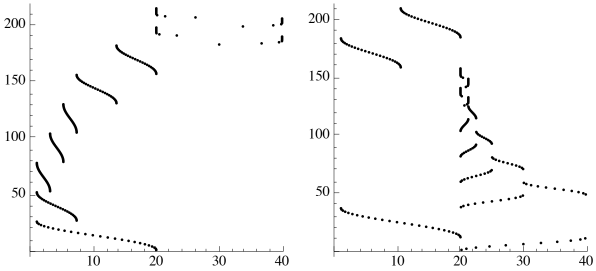
Here is a piecewise function integration that uses "DoubleExponential" strategy for the non-oscillatory integral and "LocalAdaptive" for the finite range oscillatory integral.

```
In[37]:= NIntegrate[{ Cos[200 x]/x^6     0 < x < 20
                      1/Sqrt[x-20]       True        ,
           {x, 1, 40}, Method → {"SymbolicPiecewiseSubdivision",
             Method → {"OscillatorySelection", Method → "DoubleExponential",
                "FourierFiniteRangeMethod" → {"LocalAdaptive", "Partitioning" → 3}}}]

Out[37]= 8.94871
```

These are the sampling points of the preceding integration and integral but with default option settings. The pattern between $[0, 20]$ on the left picture is typical for the local adaptive quadrature—the recursive partitioning into three parts can be seen (because of the option "Partitioning" -> 3 given to "LocalAdaptive"). The pattern over $[0, 20]$ on the right picture comes from "GlobalAdaptive". The pattern between $[20, 40]$ on the first picture is typical for the double-exponential quadrature. The same pattern can be seen on the second picture between $[20, 21 + 1/4]$ since "GlobalAdaptive" uses by default the "DoubleExponential" singularity handler.

```
In[38]:= k = 0; pointsDELA = Reap[NIntegrate[⎧ Cos[200 x]/x^6   0 < x < 20
                                              ⎩ 1/√(x-20)        True        ,
            {x, 1, 40}, Method → {"SymbolicPiecewiseSubdivision",
              Method → {"OscillatorySelection", Method → "DoubleExponential",
                "FourierFiniteRangeMethod" → {"LocalAdaptive", "Partitioning" → 3},
                "FourierInfiniteRangeMethod" → "ExtrapolatingOscillatory"}},
            EvaluationMonitor :→ Sow[{x, k++}]]]⟦2, 1⟧;
         k = 0; points = Reap[NIntegrate[⎧ Cos[200 x]/x^6   0 < x < 20
                                        ⎩ 1/√(x-20)        True        , {x, 1, 40},
            EvaluationMonitor :→ Sow[{x, k++}]]]⟦2, 1⟧;
         grDELA = Graphics[{PointSize[0.01], Point /@ pointsDELA},
            AspectRatio -> 1, Axes -> True,
          PlotRange -> {{0, 40}, All}];
         gr = Graphics[{PointSize[0.01], Point /@ points},
            AspectRatio -> 1, Axes -> True, PlotRange -> {{0, 40}, All}];
         GraphicsGrid[{{grDELA, gr}}]
```

Out[42]=



If the application of a particular oscillatory method is desired for a particular type of oscillatory integrals, either the corresponding options of "OscillatorySelection" should be changed, or the Method option in NIntegrate should be used without the preprocessor "OscillatorySelection".

Here is a piecewise function integration that uses "ExtrapolatingOscillatory" for any of the infinite range oscillatory integrals.

```
In[10]:= NIntegrate[⎧ BesselJ[3,-x]/√(-x)   x < 0
                    ⎪ Cos[200 x]/√x         0 < x < 20
                    ⎨                                    , {x, -∞, ∞},
                    ⎪ Sin[2 x+3]/x^2        x > 30
                    ⎩ 1/Log[x]              True
          Method → {"SymbolicPiecewiseSubdivision", Method → {"OscillatorySelection",
              "FourierInfiniteRangeMethod" → "ExtrapolatingOscillatory"}}]

Out[10]= 3.77933
```

If "ExtrapolatingOscillatory" is given as the method, "OscillatorySelection" uses it for infinite range oscillatory integration.

In[1]:= $$\texttt{NIntegrate}\left[\frac{\texttt{Sin}\left[2\,x^{3}+3\right]}{x^{2}},\ \{x,\ 1,\ \infty\},\ \texttt{Method}\to\texttt{"ExtrapolatingOscillatory"}\right]\ \texttt{// Timing}$$

```
Out[1]= {0.137979, -0.0206489}
```

The integration above is faster with the default options of NIntegrate. For this integral "OscillatorySelection", which is applied by default, uses "DoubleExponentialOscillatory".

In[2]:= $$\texttt{NIntegrate}\left[\frac{\texttt{Sin}\left[2\,x^{3}+3\right]}{x^{2}},\ \{x,\ 1,\ \infty\}\right]\ \texttt{// Timing}$$

```
Out[2]= {0.010998, -0.0206489}
```

Working with Sums of Oscillating Terms

In many cases it is useful to apply the oscillatory algorithms to integrands that are sums of oscillating functions. That is, each term of such integrands is a product of an oscillating function and a less oscillating one. (See "Oscillatory Strategies" for the forms recognized as oscillatory by NIntegrate.)

Here is an example of integration that applies the specialized oscillatory NIntegrate algorithms to each term of the integrand.

In[4]:= $$\texttt{NIntegrate}\left[\frac{\texttt{Cos}[100\,x]+\texttt{Sin}[18\,x]+\texttt{Cos}[12\,x]}{\sqrt{x}},\ \{x,\ 1,\ \infty\},\right.$$
$$\left.\texttt{Method}\to\{\texttt{"OscillatorySelection"},\ \texttt{"TermwiseOscillatory"}\to\texttt{True}\}\right]\ \texttt{// Timing}$$

```
Out[4]= {0.067989, 0.0879161}
```

By default this option is set to False, and the integral cannot be computed.

In[5]:= $$\texttt{NIntegrate}\left[\frac{\texttt{Cos}[100\,x]+\texttt{Sin}[18\,x]+\texttt{Cos}[12\,x]}{\sqrt{x}},\ \{x,\ 1,\ \infty\},\right.$$
$$\left.\texttt{Method}\to\{\texttt{"OscillatorySelection"},\ \texttt{"TermwiseOscillatory"}\to\texttt{False}\}\right]\ \texttt{// Timing}$$

Out[5]= $\{0.039994,\ 4.83416\times10^{123}\}$

The option is "TermwiseOscillatory" is set by default to False since splitting the integrals can lead in some cases to divergent results.

Here is a convergent integral. If it is split into two integrals each will be divergent.

In[6]:= $$\mathbf{Integrate}\left[\frac{\mathbf{Cos}[\mathbf{x}]^2}{\mathbf{x}^2}-\frac{\mathbf{Cos}[2\,\mathbf{x}]}{\mathbf{x}^2},\ \{\mathbf{x},\ 0,\ \infty\}\right]\ //\ \mathbf{N}$$

```
Out[6]= 1.5708
```

If "`TermwiseOscillatory`" `->` `True` is used the result is some big number (and lots of messages).

In[4]:= $$\mathbf{NIntegrate}\left[\frac{\mathbf{Cos}[\mathbf{x}]^2}{\mathbf{x}^2}-\frac{\mathbf{Cos}[2\,\mathbf{x}]}{\mathbf{x}^2},\ \{\mathbf{x},\ 0,\ \infty\},\ \mathbf{Method}\to\{\text{"OscillatorySelection"},\ \text{"TermwiseOscillatory"}\to\mathbf{True}\}\right]$$

```
NIntegrate::slwcon:
  Numerical integration converging too slowly; suspect one of the following: singularity, value
      of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

NIntegrate::ncvb:
  NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near
      {x} = {9.61429×10^-225}. NIntegrate obtained
      1.09277555296995444002380282417575577402166042202745166400395762181`65.95458⋱
        9770191*^27949 and
      1.09277555296995444002380282417575577402166042202745166400395762181`65.95458⋱
        9770191*^27949 for the integral and error estimates. ≫

General::ovfl: Overflow occurred in computation. ≫

General::unfl: Underflow occurred in computation. ≫

General::unfl: Underflow occurred in computation. ≫

NIntegrate::ncvb:
  NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near {x} =
      {8.1179984417887779478631707510323755968490403425701045019222812619 6×10^-76}.
      NIntegrate obtained -5.03035×10^76 and
      4.9601122390425185`*^76 for the integral and error estimates. ≫

Out[4]= 1.092775552969954×10^27949
```

If "`TermwiseOscillatory`" `->` `False` is used the result is closer to the exact one.

```
In[7]:= NIntegrate[Cos[x]^2 / x^2 - Cos[2 x] / x^2, {x, 0, Infinity},
         Method → {"OscillatorySelection", "TermwiseOscillatory" → False}]

NIntegrate::slwcon:
  Numerical integration converging too slowly; suspect one of the following: singularity, value
      of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

NIntegrate::ncvb:
  NIntegrate failed to converge to prescribed accuracy after 9 recursive bisections in x near
      {x} = {132.64}. NIntegrate obtained 1.57093011608408 7` and
      0.000748285430212249` for the integral and error estimates. ≫

Out[7]= 1.57093
```

"UnitCubeRescaling"

"UnitCubeRescaling" is a preprocessor that transforms the integration region into a unit cube or hypercube. The variables of the original integrand are replaced the result is multiplied by the Jacobian of the transformation.

option name	default value	
"FunctionalRangesOnly"	True	what ranges should be transformed to the unit cube
	"GlobalAdaptive"	integration strategy or preprocessor to which the integration will be passed
"SymbolicProcessing"	Automatic	number of seconds to do symbolic processing

Options of "UnitCubeRescaling".

This uses unit cube rescaling and it is faster than the computation that follows.

```
In[10]:= NIntegrate[Sin[x^2 + y^2] / √(x^2 + y^2), {x, 0, 5},
           {y, 0, √x}, Method → "UnitCubeRescaling"] // Timing
Out[10]= {0.221967, 0.596359}
```

This integration does not use unit cube rescaling. It is done approximately three times slower than the previous one.

```
In[11]:= NIntegrate[Sin[x^2 + y^2] / √(x^2 + y^2), {x, 0, 5}, {y, 0, √x},
           Method → {Automatic, "SymbolicProcessing" → 0}] // Timing
Out[11]= {0.570913, 0.596359}
```

"UnitCubeRescaling" transforms the integral

$$\int_{a_1}^{b_1}\int_{a_2(x_1)}^{b_2(x_1)}\cdots\int_{a_d(x_1,\ldots,x_{d-1})}^{b_d(x_1,\ldots,x_{d-1})} f(x_1,\ldots,x_d)\,dx_1\ldots dx_d \tag{32}$$

into an integral over the hypercube $[0,1]^d$. Assuming that a_1 and b_1 are finite and a_i, b_i, $i=2,\ldots,d$ are piecewise continuous functions the transformation used by "UnitCubeRescaling" is

$$x_i = a_i(\hat{x}_1,\ldots,\hat{x}_{i-1}) + \hat{x}_i\big(b_i(\hat{x}_1,\ldots,\hat{x}_{i-1}) - a_i(\hat{x}_1,\ldots,\hat{x}_{i-1})\big),\ i=1,\ldots,d. \tag{33}$$

The Jacobian of this transformation is

$$J(\hat{x}_1, \ldots, \hat{x}_d) = \prod_{i=1}^{d} \left(b_i(\hat{x}_1, \ldots, \hat{x}_d) - a_i(\hat{x}_1, \ldots, \hat{x}_d)\right). \qquad (34)$$

If for the i^{th} axis one or both of a_i and b_i are infinite, then the formula for x_i in (33) is a non-affine transformation that maps $[0, 1]$ into $\left[a_i(\hat{x}_1, \ldots, \hat{x}_{i-1}), b_i(\hat{x}_1, \ldots, \hat{x}_{i-1})\right]$. NIntegrate uses the following transformations:

$$x = a + \frac{1}{1 - \hat{x}} - 1,\ x \in [a, \infty),$$

$$x = 1 + b - \frac{1}{\hat{x}}, x \in [-\infty, b),$$

$$x = -\frac{1}{-1 + \hat{x}} - \frac{1}{\hat{x}}, x \in (-\infty, \infty),$$

where $\hat{x} \in [0, 1]$.

Applying "UnitCubeRescaling" makes the integrand more complicated if the integration region boundaries are constants (finite or infinite). Since NIntegrate has efficient affine and infinite internal variable transformations the integration process would become slower. If some of the integration region boundaries are functions, applying "UnitCubeRescaling" would make the integration faster since the computations that involve the integration variables are done only when the integrand is evaluated. Because of these performance considerations "UnitCubeRescaling" has the option "FunctionRangesOnly". If "FunctionRangesOnly" is set to True the rescaling is applied only to multidimensional functional ranges.

This integration uses unit cube rescaling.

```
In[12]:= NIntegrate[Exp[-1 / 10 (x + y)] x^2, {x, 0, ∞}, {y, 0, ∞},
           Method → {"UnitCubeRescaling", "FunctionalRangesOnly" → False}] // Timing
Out[12]= {0.483926, 20 000.}
```

This integration does not use unit cube rescaling. It is done approximately two times faster than the previous one.

```
In[13]:= NIntegrate[Exp[-1 / 10 (x + y)] x^2, {x, 0, ∞}, {y, 0, ∞},
           Method → {"UnitCubeRescaling", "FunctionalRangesOnly" → True}] // Timing
Out[13]= {0.184972, 20 000.}
```

Example Implementation

The transformation process used by "UnitCubeRescaling" is the same as the following one implemented by the function FRangesToCube (also defined in "Duffy's Coordinates Generalization and Example Implementation").

This function provides the transformation (33) and its Jacobian (34) for a list of integration ranges and a list of rectangular parallelepiped sides or a hypercube side.

```
In[14]:= FRangesToCube[ranges_, cubeSides : {{_, _} ...}] :=
           Module[{t, t1, jac, vars, rules = {}},
             vars = First /@ ranges;
             t = MapThread[(t1 = Rescale[#1[[1]], #2, {#1[[2]], #1[[3]]} /. rules];
                  AppendTo[rules, #1[[1]] → t1]; t1) &, {ranges, cubeSides}];
             jac = Times @@ MapThread[D[#1, #2] &, {t, vars}];
             {rules, jac}
            ] /; Length[ranges] == Length[cubeSides];
         FRangesToCube[ranges_, cubeSide : {_, _}] :=
           FRangesToCube[ranges, Table[cubeSide, {Length[ranges]}]];
         FRangesToCube[ranges_] := FRangesToCube[ranges, {0, 1}];
```

Each transformation of the transformation (33) can be done with Rescale.

```
In[17]:= Rescale[x, {0, 1}, {a, b}]

Out[17]= a + (-a + b) x
```

Note that for given axis i the transformation rules already derived for axes $1, \ldots, i-1$ need to be applied to the original boundaries before the rescaling of boundaries along the i^{th} axis.

The transformation rules and the Jacobian for $[0, 1]\times[0, 1]\to [0, 1]\times[a(x), b(x)]$.

```
In[18]:= {transRules, jacobian} = FRangesToCube[{{x, 0, 1}, {y, a[x], b[x]}}];
         transRules
         jacobian

Out[19]= {x → x, y → a[x] + y (-a[x] + b[x])}

Out[20]= -a[x] + b[x]
```

Application of the transformation to a function.

```
In[21]:= (F[x, y] /. transRules) jacobian

Out[21]= (-a[x] + b[x]) F[x, a[x] + y (-a[x] + b[x])]
```

The transformation rules and the Jacobian for $[0,1]\times[0,1]\to[0,\infty]\times[a(x),b(x)]$.

```
In[22]:= {transRules, jacobian} = FRangesToCube[{{x, 0, ∞}, {y, a[x], b[x]}}];
         transRules
         jacobian
```

```
                    1                 1                     1                 1
Out[23]= {x → -1 + -----, y → a[-1 + -----] + y (-a[-1 + -----] + b[-1 + -----])}
                   1 - x             1 - x                 1 - x             1 - x

               1              1
         -a[-1 + ---] + b[-1 + ---]
                 1-x           1-x
Out[24]= --------------------------
                 (1 - x)^2
```

The transformation rules and the Jacobian for $[0,1]\times[0,1]\to[a_0,b_0]\times[a_1(x),b_1(x)]$.

```
In[25]:= {transRules, jacobian} = FRangesToCube[{{x, a0, b0}, {y, a1[x], b1[x]}}];
         transRules
         jacobian
```

```
Out[26]= {x → a0 + x (-a0 + b0),
          y → a1[a0 + x (-a0 + b0)] + y (-a1[a0 + x (-a0 + b0)] + b1[a0 + x (-a0 + b0)])}

Out[27]= (-a0 + b0) (-a1[a0 + x (-a0 + b0)] + b1[a0 + x (-a0 + b0)])
```

"SymbolicPreprocessing"

"SymbolicPreprocessing" is a composite preprocessor made to simplify the switching on and off of the other preprocessors.

option name	default value	
Method	Automatic	integration strategy or preprocessor to which the integration will be passed
"SymbolicPiecewiseSubdivision"	True	piecewise subdivision
"EvenOddSubdivision"	True	even-odd subdivision
"OscillatorySelection"	True	detection of products with an oscillatory function
"UnitCubeRescaling"	Automatic	rescaling to the unit hypercube
"SymbolicProcessing"	Automatic	number of seconds to do symbolic processing

"SymbolicPreprocessing" options.

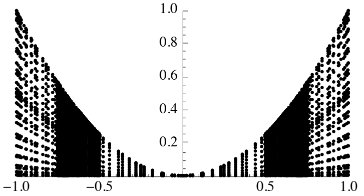
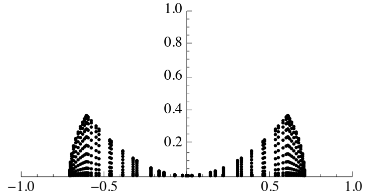
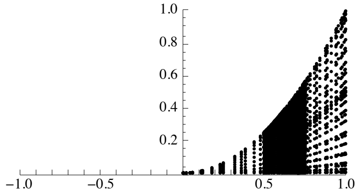
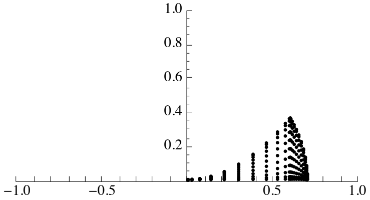
When "UnitCubeRescaling" is set to Automatic it is applied only to multidimensional functional ranges.

Here is an example of the integration of $\int_{-1}^{1}\int_{0}^{x^2}\frac{\text{Boole}\left[x^2+y^2<\frac{1}{2}\right]}{\sqrt{x^2+y^2}}\,dy\,dx$ with different combinations of preprocessor application.

In[30]:=

```
grarr = Map[Graphics[{PointSize[0.01],
                                    Boole[x^2 + y^2 < 1 / 2]
        Point /@ Reap[NIntegrate[─────────────────────────, {x, -1, 1}, {y, 0, x²},
                                         √(x² + y²)
            Method → {"SymbolicPreprocessing", "EvenOddSubdivision" → #[[1]],
              "SymbolicPiecewiseSubdivision" → #[[2]], Method → {"GlobalAdaptive",
                Method → "GaussKronrodRule", "SingularityDepth" → ∞}},
            PrecisionGoal → 3, EvaluationMonitor :→ Sow[{x, y}]]][[2, 1]]},
      PlotRange → {{-1, 1}, {0, 1}}, Axes -> True] &,
    Outer[List, {False, True}, {False, True}],
    {-2}];
Grid[Join[{{"", SpanFromLeft, "SymbolicPiecewiseSubdivision", SpanFromLeft},
    {SpanFromAbove, SpanFromBoth, False, True}},
   {Join[{"EvenOddSubdivision", False}, grarr⟦1⟧]},
   {Join[{SpanFromAbove, True}, grarr⟦2⟧]}], Dividers → All]
```

Out[44]=

		SymbolicPiecewiseSubdivision	
		False	True
EvenOddSubdivision	False		
	True		

Examples and Applications

Closed-Contour Integrals

This function calculates the mass of a closed contour given in polar coordinates parametrization.

```
In[42]:= ClosedContourIntegral[fexpr_,
            {x_, xpareq_}, {y_, ypareq_}, {θ_, 0, 2 π}, opts___] :=
          NIntegrate[fexpr √(x^2 + y^2) /. {x → xpareq, y → ypareq}, {θ, 0, 2 π},
            Evaluate[Sequence @@ Append[{opts}, Method → "Trapezoidal"]]]
```

This is circumference of the ellipse with radii 2 and 3 using `Integrate`.

```
In[43]:= {a, b} = {2, 3};
         exact = Integrate[√(a^2 Cos[θ]^2 + b^2 Sin[θ]^2), {θ, 0, 2 π}]

Out[44]= 8 EllipticE[-5/4]
```

Here is the circumference approximation of the ellipse with radii 2 and 3 using the same function.

```
In[45]:= ep = ClosedContourIntegral[1, {x, a Cos[θ]}, {y, b Sin[θ]}, {θ, 0, 2 π}]

Out[45]= 15.8654
```

The approximation compares quite well with the exact value.

```
In[46]:= Abs[exact - ep]

Out[46]= 9.14824×10^-13
```

Fourier Series Calculation

This is a *Mathematica* function that calculates a truncated Fourier series approximation of a function.

```
In[83]:= FourierAnalysis[f_, {x_, xmin_, xmax_}, nterms_,
            integrator_: (NIntegrate[##, Method → "GlobalAdaptive", MaxRecursion → 30] &)] :=
          Block[{a, b, funcTerms}, a = 2/(xmax - xmin)
               Table[integrator[Cos[(2 π)/(xmax - xmin) j x] f, {x, xmin, xmax}], {j, 0, nterms}];
            b = 2/(xmax - xmin) Table[integrator[Sin[(2 π)/(xmax - xmin) j x] f, {x, xmin, xmax}],
                {j, 1, nterms}];
            funcTerms = a[[1]]/2 + Total[Table[Cos[(2 π)/(xmax - xmin) j x] a[[j + 1]] +
                  Sin[(2 π)/(xmax - xmin) j x] b[[j]], {j, 1, nterms}]];
            funcTerms
          ];
```

Fourier approximation of $x^3 + x^2$ over $[-2, 2]$ using `Integrate`.

```
In[84]:= func = FourierAnalysis[x^3 + x^2, {x, -2, 2}, 12, Integrate]
```

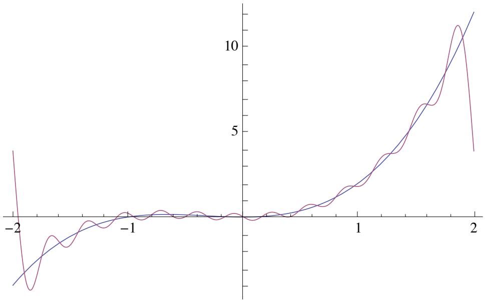
Out[84]=

$$\frac{4}{3}-\frac{16\,\mathrm{Cos}\left[\frac{\pi x}{2}\right]}{\pi^2}+\frac{4\,\mathrm{Cos}[\pi x]}{\pi^2}-\frac{16\,\mathrm{Cos}\left[\frac{3\pi x}{2}\right]}{9\pi^2}+\frac{\mathrm{Cos}[2\pi x]}{\pi^2}-\frac{16\,\mathrm{Cos}\left[\frac{5\pi x}{2}\right]}{25\pi^2}+\frac{4\,\mathrm{Cos}[3\pi x]}{9\pi^2}-$$
$$\frac{16\,\mathrm{Cos}\left[\frac{7\pi x}{2}\right]}{49\pi^2}+\frac{\mathrm{Cos}[4\pi x]}{4\pi^2}-\frac{16\,\mathrm{Cos}\left[\frac{9\pi x}{2}\right]}{81\pi^2}+\frac{4\,\mathrm{Cos}[5\pi x]}{25\pi^2}-\frac{16\,\mathrm{Cos}\left[\frac{11\pi x}{2}\right]}{121\pi^2}+\frac{\mathrm{Cos}[6\pi x]}{9\pi^2}+$$
$$\frac{16\left(-6+\pi^2\right)\mathrm{Sin}\left[\frac{\pi x}{2}\right]}{\pi^3}+\frac{\left(24-16\pi^2\right)\mathrm{Sin}[\pi x]}{2\pi^3}+\frac{16\left(-2+3\pi^2\right)\mathrm{Sin}\left[\frac{3\pi x}{2}\right]}{9\pi^3}+\frac{\left(3-8\pi^2\right)\mathrm{Sin}[2\pi x]}{2\pi^3}+$$
$$\frac{16\left(-6+25\pi^2\right)\mathrm{Sin}\left[\frac{5\pi x}{2}\right]}{125\pi^3}+\frac{\left(8-48\pi^2\right)\mathrm{Sin}[3\pi x]}{18\pi^3}+\frac{16\left(-6+49\pi^2\right)\mathrm{Sin}\left[\frac{7\pi x}{2}\right]}{343\pi^3}+\frac{1}{2}\left(\frac{3}{8\pi^3}-\frac{4}{\pi}\right)\mathrm{Sin}[4\pi x]+$$
$$\frac{16\left(-2+27\pi^2\right)\mathrm{Sin}\left[\frac{9\pi x}{2}\right]}{243\pi^3}-\frac{4\left(-3+50\pi^2\right)\mathrm{Sin}[5\pi x]}{125\pi^3}+\frac{16\left(-6+121\pi^2\right)\mathrm{Sin}\left[\frac{11\pi x}{2}\right]}{1331\pi^3}+\frac{\left(1-24\pi^2\right)\mathrm{Sin}[6\pi x]}{18\pi^3}$$

This a plot of $x^3 + x^2$ and the Fourier series approximation.

```
In[85]:= Plot[{Tooltip[x^3 + x^2, "Original\nfunction"],
           Tooltip[func, "Fourier\napproximation"]}, {x, -2, 2}, PlotRange → All]
```

Out[85]=



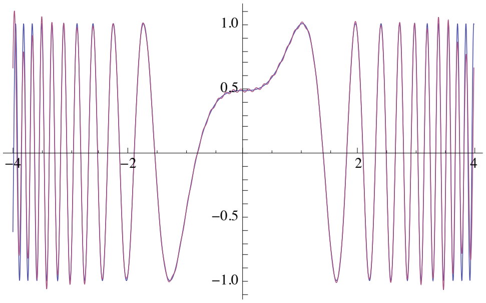
This calculates a 60-term Fourier approximation of `Sin`$[x^3 + \frac{1}{2}]$ over $[-4, 4]$ using `NIntegrate`. If `Integrate` is used the calculation will be very slow.

```
In[86]:= func = FourierAnalysis[Sin[x^3 + 1/2], {x, -4, 4}, 60]; // Timing

Out[86]= {11.2887, Null}
```

This a plot of `Sin`$[x^3 + \frac{1}{2}]$ and the Fourier series approximation.

```
In[87]:= Plot[{Tooltip[Sin[x^3 + 1/2], "Original\nfunction"],
           Tooltip[func, "Fourier\napproximation"]}, {x, -4, 4}, PlotRange → All]
```

Out[87]=



NIntegrate Integration Rules

Introduction

An integration rule computes an estimate of an integral over a region using a weighted sum. In the context of `NIntegrate` usage, an integration rule object provides both an integral estimate and an error estimate as a measure of the integral estimate's accuracy.

An integration rule samples the integrand with a set of points. These points are called sampling points. In the literature these are usually called abscissas. Corresponding to each sampling point x_i there is a weight number w_i. An integration rule estimates the integral $\int_a^b f(x)\,dx$ with the weighted sum $\sum w_i\, f(x_i)$. An integration rule is a functional, that is, it maps functions over the interval $[a, b]$ (or a more general region) into real numbers.

If a rule is applied over the region V this will be denoted as $R^V(f)$, where f is the integrand.

The sampling points of the rules considered below are chosen to compute estimates for integrals either over the interval $[0, 1]$, or the unit cube $[0, 1]^d$, or the "centered" unit cube $\left[-\frac{1}{2}, \frac{1}{2}\right]^d$, where d is the dimension of the integral. So if V is one of these regions, $R(f)$ will be used instead of $R^V(f)$. When these rules are applied to other regions, their abscissas and estimates need to be scaled accordingly.

The integration rule R is said to be exact for the function f if $R^{[a,b]}(f) = \int_a^b f(x)\,dx$.

The application of an integration rule R to a function f will be referred as an integration of f, for example, "when f is integrated by R, we get $R(f)$."

A one-dimensional integration rule is said to be of degree n if it integrates exactly all polynomials of degree n or less, and will fail to do so for at least one polynomial of degree $n+1$.

A multidimensional integration rule is said to be of degree n if it integrates exactly all monomials of degree n or less, and will fail to do so for at least one monomial of degree $n+1$, that is, the rule is exact for all monomials of the form $\prod_{i=1}^d x_i^{\alpha_i}$, where d is the dimension, $\alpha_i \geq 0$, and $\sum_{i=1}^d \alpha_i \leq n$.

A null rule of degree m will integrate to zero all monomials of degree $\leq m$ and will fail to do so for at least one monomial of degree $m+1$. Each null rule may be thought of as the difference between a basic integration rule and an appropriate integration rule of a lower degree.

If the set of sampling points of a rule R_1 of degree n contains the set of sampling points of a rule R_2 of a lower degree m, that is, $n > m$, then R_2 is said to be embedded in R_1. This will be denoted as $R_2 \subset R_1$.

An integration rule of degree n that is a member of a family of rules with a common derivation and properties but different degrees will be denoted as $R(f, n)$, where R might be chosen to identify the family. (For example, trapezoidal rule of degree 4 might be referred to as $T(f, 4)$.)

If each rule in a family is embedded in another rule in the same family, then the rules of that family are called progressive. (For any given $m \in \mathbb{N}$ there exists $n \in \mathbb{N}$, $n > m$, for which $R(f, m) \subset R(f, n)$).

An integration rule is of open type if the integrand is not evaluated at the end points of the interval. It is of closed type if it uses integrand evaluations at the interval end points.

An `NIntegrate` integration rule object has one integration rule for the integral estimate and one or several null rules for the error estimate. The sampling points of the integration rule and the null rules coincide. It should be clear from the context whether "integration rule" or "rule" would mean an `NIntegrate` integration rule object, or an integration rule in the usual mathematical sense.

Integration Rule Specification

All integration rules described below, except `"MonteCarloRule"`, are to be used by the adaptive strategies of `NIntegrate`. In `NIntegrate`, all Monte Carlo strategies, crude and adaptive, use `"MonteCarloRule"`. Changing the integration rule component of an integration strategy will make a different integration algorithm.

The way to specify what integration rule the adaptive strategies in `NIntegrate` (see "Global Adaptive Strategy" and "Local Adaptive Strategy") should use is through a `Method` suboption.

Here is an example of using an integration rule with a strategy (`"GlobalAdaptive"`).

```
In[1]:= NIntegrate[1 / Sqrt[x], {x, 0, 1},
          Method → {"GlobalAdaptive", Method → "ClenshawCurtisRule"}] // InputForm
Out[1]//InputForm= 1.9999999999193905
```

Here is an example of using the same integration rule as in the example above through a different strategy ("LocalAdaptive").

```
In[2]:= NIntegrate[1 / Sqrt[x], {x, 0, 1},
          Method → {"LocalAdaptive", Method → "ClenshawCurtisRule"}] // InputForm
Out[2]//InputForm= 1.9999999976742142
```

If NIntegrate is given a method option that has only an integration rule specification other than "MonteCarloRule", then that rule is used with the "GlobalAdaptive" strategy. The two inputs below are equivalent.

For this integration only integration rule is specified.

```
In[3]:= NIntegrate[1 / Sqrt[x], {x, 0, 1}, Method → "LobattoKronrodRule"] // InputForm
Out[3]//InputForm= 2.0000000000019873
```

For this integration an integration strategy and an integration rule are specified.

```
In[4]:= NIntegrate[1 / Sqrt[x], {x, 0, 1},
          Method → {"GlobalAdaptive", Method → "LobattoKronrodRule"}] // InputForm
Out[4]//InputForm= 2.0000000000019873
```

Similarly for "MonteCarloRule", the adaptive Monte Carlo strategy is going to be used when the following two equivalent commands are executed.

For this Monte Carlo integration only the "MonteCarloRule" is specified.

```
In[5]:= NIntegrate[1 / Sqrt[x], {x, 0, 1}, Method → "MonteCarloRule"] // InputForm
Out[5]//InputForm= 1.9923900530424228
```

For this Monte Carlo integration a Monte Carlo integration strategy and "MonteCarloRule" are specified.

```
In[6]:= NIntegrate[1 / Sqrt[x], {x, 0, 1},
          Method → {"AdaptiveMonteCarlo", Method → "MonteCarloRule"}] // InputForm
Out[6]//InputForm= 1.9745771611582486
```

"TrapezoidalRule"

The trapezoidal rule for integral estimation is one of the simplest and oldest rules (possibly used by the Babylonians and certainly by the ancient Greek mathematicians):

$$(b-a)\,\frac{f(a)+f(b)}{2} \approx \int_a^b f(x)\,dx. \tag{35}$$

The compounded trapezoidal rule is a Riemann sum of the form

$$T(f, n) = \frac{1}{2} h f(a) + h \sum_{i=1}^{n-1} f(a + h i) + \frac{1}{2} h f(b) \approx \int_a^b f(x) \, dx, \tag{36}$$

where $h = \frac{b-a}{n-1}$.

If the Method option is given the value "TrapezoidalRule", the compounded trapezoidal rule is used to estimate each subinterval formed by the integration strategy.

A "TrapezoidalRule" integration:

```
In[7]:= NIntegrate[x + 5, {x, 0, 7}, Method -> "TrapezoidalRule"]
Out[7]= 59.5
```

option name	default value	
"Points"	5	number of coarse trapezoidal points
"RombergQuadrature"	True	should Romberg quadrature be used or not
"SymbolicProcessing"	Automatic	number of seconds to do symbolic preprocessing

"TrapezoidalRule" options.

The trapezoidal rule and its compounded (multipanel) extension are not very accurate. (The compounded trapezoidal rule is exact for linear functions and converges at least as fast as n^{-2}, if the integrand has continuous second derivative [DavRab84].) The accuracy of the multipanel trapezoidal rule can be increased using the "Romberg quadrature".

Since the abscissas of $T(f, n)$ are a subset of $T(f, 2n - 1)$, the difference $|T(f, 2n - 1) - T(f, n)|$, can be taken to be an error estimate of the integral estimate $T(f, 2n - 1)$, and can be computed without extra integrand evaluations.

The option "Points" $->$ k can be used to specify how many coarse points are used. The total number of points used by "TrapezoidalRule" is $2k - 1$.

This verifies that the sampling points are as in (36).

```
In[8]:= k = 4;
        Reap@NIntegrate[x + 5, {x, 1, 7},
          Method -> {"TrapezoidalRule", "Points" → k, "RombergQuadrature" → False},
          EvaluationMonitor :> Sow[x]]
Out[9]= {54., {{1., 2., 3., 4., 5., 6., 7.}}}
```

"TrapezoidalRule" can be used for multidimensional integrals too.

Here is a multidimensional integration with "TrapezoidalRule". The exact result is $\int_0^1 \int_0^1 (x^2 + y)\, dy\, dx = 5/6 = 0.8333333\ldots.$

```
In[10]:= NIntegrate[x^2 + y, {x, 0, 1}, {y, 0, 1}, Method -> "TrapezoidalRule"]
Out[10]= 0.833333
```

Remark: NIntegrate has both a trapezoidal rule and a trapezoidal strategy; see "Trapezoidal" Strategy in the tutorial Integration Strategies. All internally implemented integration rules of NIntegrate have the suffix -Rule. So "TrapezoidalRule" is used to specify the trapezoidal integration rule, and "Trapezoidal" is used to specify the trapezoidal strategy.

Romberg Quadrature

The idea of the Romberg quadrature is to use a linear combination of $T(f, n)$ and $T(f, 2n-1)$ that eliminates the same order terms of truncation approximation errors of $T(f, n)$ and $T(f, 2n-1)$.

From the Euler-Maclaurin formula [DavRab84] we have

$$\int_a^b f(x)\, dx = \frac{1}{2} h f(a) + h \sum_{i=1}^{n-1} f(a + h i) + \frac{1}{2} h f(b) - \frac{1}{12} h^2 (f'(b) - f'(a)) + \frac{1}{720} (b-a) h^4 f^4[\xi],$$

where

$$h = \frac{b-a}{n-1},\ a < \xi < b.$$

Hence we can write

$$\int_a^b f(x)\, dx = T(f, n) + A h^2 + O(h^4),$$

$$\int_a^b f(x)\, dx = T(f, 2n-1) + A \left(\frac{h}{2}\right)^2 + O(h^4).$$

The h^2 terms of the equations above can be eliminated if the first equation is subtracted from the second equation four times. The result is

$$\int_a^b f(x)\,dx = \frac{4\,T(f, 2\,n-1) - T(f, n)}{3} + O\left(h^4\right).$$

This example shows that a trapezoidal rule using the Romberg quadrature gives better performance than the standard trapezoidal rule. Also, the result of the former is closer to the exact result, $\int_0^1 \sqrt{x}\,dx = \frac{2}{3} = 0.6666666\,\ldots$.

```
In[11]:= NIntegrate[Sqrt[x], {x, 0, 1},
            Method → {"GlobalAdaptive", Method → {"TrapezoidalRule",
                 "Points" → 5, "RombergQuadrature" → True}, "SingularityDepth" → ∞},
            MaxRecursion → 100, PrecisionGoal → 8] // InputForm // Timing

Out[11]= {0.06399, 0.6666666666571325}
```

Here is an integration with a trapezoidal rule that does not use Romberg quadrature.

```
In[10]:= NIntegrate[Sqrt[x], {x, 0, 1},
            Method → {"GlobalAdaptive", Method → {"TrapezoidalRule", "Points" → 5,
                 "RombergQuadrature" → False}, "SingularityDepth" → ∞},
            MaxRecursion → 100, PrecisionGoal → 8] // InputForm // Timing

Out[10]= {0.109983, 0.6666666644416138}
```

"TrapezoidalRule" Sampling Points and Weights

The following calculates the trapezoidal sampling points, weights, and error weights for a given precision.

```
In[3]:= n = 5; precision = MachinePrecision;
        {absc, weights, errweights} = NIntegrate`TrapezoidalRuleData[n, precision]

Out[4]= {{0., 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1.},
         {0.0625, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.125, 0.0625},
         {-0.0625, 0.125, -0.125, 0.125, -0.125, 0.125, -0.125, 0.125, -0.0625}}
```

Here is how the Romberg quadrature weights and error weights can be derived.

```
In[5]:= rombergAbsc = absc;
        lowOrderWeights = - (errweights - weights);
                          4 weights - lowOrderWeights
        rombergWeights = ─────────────────────────────;
                                       3
        rombergErrorWeights = rombergWeights - weights;
        {rombergAbsc, rombergWeights, rombergErrorWeights}

Out[9]= {{0., 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1.},
         {0.0416667, 0.166667, 0.0833333, 0.166667, 0.0833333, 0.166667, 0.0833333, 0.166667, 0.0416667},
         {-0.0208333, 0.0416667, -0.0416667, 0.0416667,
          -0.0416667, 0.0416667, -0.0416667, 0.0416667, -0.0208333}}
```

"NewtonCotesRule"

Newton-Cotes integration formulas are formulas of interpolatory type with sampling points that are equally spaced.

The Newton-Cotes quadrature for `NIntegrate` can be specified with the `Method` option value `"NewtonCotesRule"`.

```
In[20]:= NIntegrate[1 / Sqrt[x], {x, 0, 1}, Method → "NewtonCotesRule"]
Out[20]= 2.
```

option name	default value	
"Points"	3	number of coarse Newton-Cotes points
"Type"	Closed	type of the Newton-Cotes rule
"SymbolicProcessing"	Automatic	number of seconds to do symbolic preprocessing

"NewtonCotesRule" options.

Let the interval of integration, $[a, b]$, be divided into $n-1$ subintervals of equal length by the points

$$a,\, a+h,\, a+2\,h,\, \ldots,\, a+(n-1)\,h = b,\;\; h = \frac{b-a}{n-1}.$$

Then the integration formula of interpolatory type is given by

$$\int_a^b f(x)\,dx \approx \frac{b-a}{n-1}\sum_{k=0}^{n-1} B_{n-1,k}\, f(a+h\,k),$$

where

$$B_{n-1,k} = \frac{n-1}{b-a}\int_a^b \frac{w(x)}{(-a-k\,h+x)\,w'(a+h\,k)}\,dx,$$

with

$$w(x) = (x-a)\,(x-a-h)\ldots(x-a-(n-1)\,h).$$

When n is large, the Newton-Cotes n-point coefficients are large and are of mixed sign.

```
In[21]:= NIntegrate`NewtonCotesRuleData[25, MachinePrecision][[2]]

Out[21]= {0.00421169, 0.0712002, -0.499965, 5.17028, -43.2178, 306.528, -1854.44, 9697.73, -44 332.4,
          178 882., -642 291., 2.0662×10^6, -5.98934×10^6, 1.57199×10^7, -3.75117×10^7, 8.16646×10^7,
          -1.62678×10^8, 2.97256×10^8, -4.99278×10^8, 7.72171×10^8, -1.10118×10^9, 1.44964×10^9,
          -1.76314×10^9, 1.98245×10^9, -2.06138×10^9, 1.98245×10^9, -1.76314×10^9, 1.44964×10^9,
          -1.10118×10^9, 7.72171×10^8, -4.99278×10^8, 2.97256×10^8, -1.62678×10^8, 8.16646×10^7,
          -3.75117×10^7, 1.57199×10^7, -5.98934×10^6, 2.0662×10^6, -642 291., 178 882., -44 332.4,
          9697.73, -1854.44, 306.528, -43.2178, 5.17028, -0.499965, 0.0712002, 0.00421169}
```

Since this may lead to large losses of significance by cancellation, a high-order Newton-Cotes rule must be used with caution.

"NewtonCotesRule" Sampling Points and Weights

The following calculates the Newton-Cotes sampling points, weights, and error weights for a given precision.

```
In[22]:= n = 5; precision = MachinePrecision;
         {absc, weights, errweights} = NIntegrate`NewtonCotesRuleData[n, precision]

Out[23]= {{0., 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1.},
          {0.0348854, 0.20769, -0.0327337, 0.370229, -0.160141, 0.370229, -0.0327337, 0.20769, 0.0348854},
          {-0.0428924, 0.20769, -0.388289, 0.370229, -0.293474, 0.370229, -0.388289, 0.20769, -0.0428924}}
```

"GaussBerntsenEspelidRule"

Gaussian quadrature uses optimal sampling points (through polynomial interpolation) to form a weighted sum of the integrand values over these points. On a subset of these sampling points a lower order quadrature rule can be made. The difference between the two rules can be used to estimate the error. Berntsen and Espelid derived error estimation rules by removing the central point of Gaussian rules with odd number of sampling points.

The Gaussian quadrature for `NIntegrate` can be specified with the `Method` option value `"GaussBerntsenEspelidRule"`.

```
In[24]:= NIntegrate[1 / Sqrt[x], {x, 0, 1}, Method → "GaussBerntsenEspelidRule"]

Out[24]= 2.
```

option name	*default value*	
`"Points"`	`Automatic`	number of Gauss points
`"SymbolicProcessing"`	`Automatic`	number of seconds to do symbolic preprocessing

`"GaussBerntsenEspelidRule"` options.

A Gaussian rule $G(f, n)$ of n points for integrand f is exact for polynomials of degree $2n-1$ (i.e., $G(f, n) = \int_a^b f(x)\, dx$ if $f(x)$ is a polynomial of degree $\leq 2n-1$).

Gaussian rules are of open type since the integrand is not evaluated at the end points of the interval. (Lobatto rules, Clenshaw-Curtis rules, and the trapezoidal rule are of closed type since they use integrand evaluations at the interval end points.)

This defines the divided differences functional [Ehrich2000]

$$\operatorname{dvd}(t_1, t_2, \ldots, t_{s+1})[f] = \sum_{\nu=1}^{s+1} \left(\prod_{\substack{\mu=1 \\ \mu \neq \nu}}^{s+1} \left(t_\nu - t_\mu\right)^{-1} \right) f(t_\nu), \; 0 \leq t_1 < t_2 < \ldots < t_{s+1} \leq 1.$$

For the Gaussian rule $G(f, 2n+1)$, with sampling points $x_1, x_2, \ldots, x_{2n+1}$, Berntsen and Espelid have derived the following error estimate functional (see [Ehrich2000])

$$E(G(f, 2m+1)) = (-1)^n \frac{2^{2n+1}\, n!^2\, (2n)!}{(4n+1)!} \operatorname{dvd}(x_1, x_2, \ldots, x_{2n+1})[f].$$

(The original formula in [Ehrich2000] is for sampling points in $[-1, 1]$. The formula above is for sampling points in $[0, 1]$.)

This example shows the number of sampling points used by NIntegrate with various values of the "GaussBerntsenEspelidRule" option "Points".

```
In[25]:= Table[(k = 0; NIntegrate[x^(1/2), {x, 0, 1},
            Method → {"GaussBerntsenEspelidRule", "Points" → i},
            EvaluationMonitor :> k++]; k), {i, 2, 20}]
Out[25]= {164, 106, 110, 128, 146, 164, 182, 200, 218, 236, 225, 243, 261, 279, 231, 245, 259, 273, 287}
```

"GaussBerntsenEspelidRule" Sampling Points and Weights

The following calculates the Gaussian abscissas, weights, and Bernsen-Espelid error weights for a given number of coarse points and precision.

```
In[26]:= n = 5; precision = 20;
         {absc, weights, errweights} =
          NIntegrate`GaussBerntsenEspelidRuleData[n, precision]

Out[27]= {{0.010885670926971503598, 0.056468700115952350462,
           0.13492399721297533795, 0.24045193539659409204, 0.36522842202382751383,
           0.50000000000000000000, 0.63477157797617248617, 0.75954806460340590796,
           0.86507600278702466205, 0.94353129988404764954, 0.98911432907302849640},
          {0.027834283558086833242, 0.062790184732452312317, 0.093145105463867125713,
           0.11659688229599523996, 0.13140227225512333109, 0.13646254338895031536,
           0.13140227225512333109, 0.11659688229599523996, 0.093145105463867125713,
           0.062790184732452312317, 0.027834283558086833242},
          {-0.025580415424079299767, 0.085466250921751643670, -0.15407013862509290808,
           0.21562641393186216190, -0.25790465419339191313, 0.27292508677790063075, -0.25790465419339191313,
           0.21562641393186216190, -0.15407013862509290808, 0.085466250921751643670, -0.025580415424079299767}}
```

The Berntsen-Espelid error weights are implemented below.

This implements the divided differences.

```
In[28]:= polyd[vec_List, nu_] := (Times @@ (vec[[nu]] - Drop[vec, {nu}])) ^ (-1);
         dvdWeights[vec_List] :=
           dvdWeights[vec] = Table[polyd[vec, nu], {nu, 1, Length[vec]}];
```

This computes the abscissas and the weights of $G(f, 2n+1)$.

```
In[30]:= {absc, weights, errweights} = NIntegrate`GaussRuleData[2 n + 1, precision];
```

This computes the Berntsen-Espelid error weights.

In[31]:=

$$\frac{\left((-1)^{n}\sqrt{\pi}\;\mathbf{Gamma}[1+n]^{2}\right)}{\mathbf{Gamma}\left[\frac{3}{2}+2\,n\right]\,2^{2\,n}}\;\mathbf{dvdWeights}[\mathbf{absc}]$$

```
Out[31]= {-0.025580415424079299767, 0.085466250921751643670, -0.15407013862509290808,
          0.21562641393186216190, -0.25790465419339191313, 0.27292508677790063075, -0.25790465419339191313,
          0.21562641393186216190, -0.15407013862509290808, 0.085466250921751643670, -0.025580415424079299767}
```

"GaussKronrodRule"

Gaussian quadrature uses optimal sampling points (through polynomial interpolation) to form a weighted sum of the integrand values over these points. The Kronrod extension of a Gaussian rule adds new sampling points in between the Gaussian points and forms a higher-order rule that reuses the Gaussian rule integrand evaluations.

The Gauss-Kronrod quadrature for `NIntegrate` can be specified with the `Method` option value `"GaussKronrodRule"`.

```
In[32]:= NIntegrate[1 / Sqrt[x], {x, 0, 1}, Method → "GaussKronrodRule"]

Out[32]= 2.
```

option name	*default value*	
`"Points"`	`Automatic`	number of Gauss points that will be extended with Kronrod points
`"SymbolicProcessing"`	`Automatic`	number of seconds to do symbolic processing

`"GaussKronrodRule"` options.

A Gaussian rule $G(f, n)$ of n points for integrand f is exact for polynomials of degree $2n-1$, that is, $G(f, n) = \int_a^b f(x)\,dx$ if $f(x)$ is a polynomial of degree $\leq 2n-1$.

Gauss-Kronrod rules are of open type since the integrand is not evaluated at the end points of the interval.

The Kronrod extension $\mathrm{GK}(f, n)$ of a Gaussian rule with n points $G(f, n)$ adds $n+1$ points to $G(f, n)$ and the extended rule is exact for polynomials of degree $3n+1$ if n is even, or $3n+2$ if n is odd. The weights associated with a Gaussian rule change in its Kronrod extension.

Since the abscissas of $G(f, n)$ are a subset of $\mathrm{GK}(f, n)$, the difference $|\mathrm{GK}(f, n) - G(f, n)|$ can be taken to be an error estimate of the integral estimate $\mathrm{GK}(f, n)$, and can be computed without extra integrand evaluations.

This example shows the number of sampling points used by `NIntegrate` with various values of `"GaussKronrodRule"` option `"Points"`.

```
In[33]:= Table[
          (k = 0; NIntegrate[x^10, {x, 0, 1}, Method → {"GaussKronrodRule", "Points" → i},
            EvaluationMonitor :> k++]; k), {i, 2, 20}]

Out[33]= {284, 91, 63, 33, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41}
```

For an implementation description of Kronrod extensions of Gaussian rules, see [PiesBrand74].

"GaussKronrodRule" Sampling Points and Weights

The following calculates the Gauss-Kronrod abscissas, weights, and error weights for a given number of coarse points and precision.

```
In[34]:= n = 5; precision = 20;
         {absc, weights, errweights} = NIntegrate`GaussKronrodRuleData[n, precision]
Out[35]= {{0.00795731995257876775, 0.04691007703066800360,
           0.12291663671457538978, 0.23076534494715845448, 0.36018479341910840329,
           0.50000000000000000000, 0.63981520658089159671, 0.76923465505284154552,
           0.87708336328542461022, 0.95308992296933199640, 0.99204268004742123225},
          {0.02129101837554091643 2, 0.05761665831123669701, 0.09340039827824632873 4,
           0.12052016961432379335, 0.13642490095627946117, 0.14149370892874560 66,
           0.13642490095627946117, 0.12052016961432379335, 0.09340039827824632873 4,
           0.05761665831123669701, 0.02129101837554091643 2},
          {0.02129101837554091643 2, -0.06084678421685784675, 0.09340039827824632873 4,
           -0.11879416563535944067, 0.13642490095627946117, -0.14295073551569883784,
           0.13642490095627946117, -0.11879416563535944067, 0.09340039827824632873 4,
           -0.06084678421685784675, 0.02129101837554091643 2}}
```

The calculations below demonstrate the degree of the Gauss-Kronrod integration rule (see above).

This computes the degree of the Gauss-Kronrod integration rule.

```
In[36]:= p = If[OddQ[n], 3 * n + 2, 3 * n + 1]
Out[36]= 17
```

This defines a function.

```
In[37]:= f[x_] := x^p
```

The command below implements the integration rule weighted sums for the integral estimate, $\sum_{i=1}^{2n+1} w_i\, f(x_i)$, and the error estimate, $\sum_{i=1}^{2n+1} e_i\, f(x_i)$, where $\{x_i\}_{i=1}^{2n+1}$ are the abscissas, $\{w_i\}_{i=1}^{2n+1}$ are the weights, and $\{e_i\}_{i=1}^{2n+1}$ are the error weights.

These are the integral and error estimates for $\int_0^1 f(x)\, dx$ computed with the rule.

```
In[38]:= Total@MapThread[{f[#1] #2, f[#1] #3} &, {absc, weights, errweights}]
Out[38]= {0.0555555555555555556, 0.0004434409627672096}
```

The integral estimate coincides with the exact result.

```
In[39]:= N[Integrate[f[x], {x, 0, 1}], precision]
Out[39]= 0.055555555555555555556
```

The error estimate is not zero since the embedded Gauss rule is exact for polynomials of degree $\leq 2n-1$. If we integrate a polynomial of that degree, the error estimate becomes zero.

This defines a function.

```
In[40]:= f[x_] := x^(2 n-1)
```

These are the integral and error estimates for $\int_0^1 f(x)\,dx$ computed with the rule.

```
In[41]:= Total@MapThread[{f[#1] #2, f[#1] #3} &, {absc, weights, errweights}]

Out[41]= {0.1000000000000000000, 0.×10^-20}
```

Here is the exact result using `Integrate`.

```
In[42]:= N[Integrate[f[x], {x, 0, 1}], precision]

Out[42]= 0.10000000000000000000
```

"LobattoKronrodRule"

The Lobatto integration rule is a Gauss-type rule with preassigned abscissas. It uses the end points of the integration interval and optimal sampling points inside the interval to form a weighted sum of the integrand values over these points. The Kronrod extension of a Lobatto rule adds new sampling points in between the Lobatto rule points and forms a higher-order rule that reuses the Lobatto rule integrand evaluations.

`NIntegrate` uses the Kronrod extension of the Lobatto rule if the `Method` option is given the value `"LobattoKronrodRule"`.

```
In[43]:= NIntegrate[1 / Sqrt[x], {x, 0, 1}, Method → "LobattoKronrodRule"]

Out[43]= 2.
```

option name	*default value*	
`"Points"`	5	number of Gauss-Lobatto points that will be extended with Kronrod points
`"SymbolicProcessing"`	`Automatic`	number of seconds to do symbolic preprocessing

`"LobattoKronrodRule"` options.

A Lobatto rule $L(f, n)$ of n points for integrand f is exact for polynomials of degree $2n-3$, (i.e., $L(f, n) = \int_a^b f(x)\,dx$ if $f(x)$ is a polynomial of degree $\leq 2n-3$).

The Kronrod extension $\mathrm{LK}(f, n)$ of a Lobatto rule with n points $L(f, n)$ adds $n-1$ points to $L(f, n)$ and the extended rule is exact for polynomials of degree $3n-2$ if n is even, or $3n-3$ if n is odd. The weights associated with a Lobatto rule change in its Kronrod extension.

As with "GaussKronrodRule", the number of Gauss points is specified with the option "GaussPoints". If "LobattoKronrodRule" is invoked with "Points" $->$ n, the total number of rule points will be $2n-1$.

This example shows the number of sampling points used by NIntegrate with various values the of "LobattoKronrodRule" option "Points".

```
In[44]:= Table[
           (k = 0; NIntegrate[x^10, {x, 0, 1}, Method → {"LobattoKronrodRule", "Points" → i},
             EvaluationMonitor :> k++]; k), {i, 3, 20}]
Out[44]= {304, 91, 63, 33, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39}
```

Since the Lobatto rule is a closed rule, the integrand needs to be evaluated at the end points of the interval. If there is a singularity at these end points, NIntegrate will ignore it.

For an implementation description of Kronrod extensions of Lobatto rules, see [PiesBrand74].

"LobattoKronrodRule" Sampling Points and Weights

The following calculates the Lobatto-Kronrod abscissas, weights, and error weights for a given number of coarse points and precision.

```
In[45]:= n = 5; precision = 20;
         {absc, weights, errweights} = NIntegrate`LobattoKronrodRuleData[n, precision]
Out[46]= {{0.×10^-20, 0.05479723624366560671, 0.17267316464601143283,
           0.32950886704450351424, 0.50000000000000000000, 0.67049113295549648576,
           0.82732683535398856717, 0.94520276375633439329, 1.0000000000000000000},
          {0.01532186948853615520 3, 0.08963134977660367799 0, 0.14198938902406054911,
           0.16711686990820884179, 0.1718810436051815362, 0.16711686990820884179,
           0.14198938902406054911, 0.08963134977660367799 0, 0.01532186948853615520 3},
          {-0.03467813051146384479 7, 0.08963134977660367799 0, -0.13023283319816167312,
           0.16711686990820884179, -0.18367451195037401934, 0.16711686990820884179,
           -0.13023283319816167312, 0.08963134977660367799 0, -0.03467813051146384479 7}}
```

The calculations below demonstrate the degree of the Lobatto-Kronrod integration rule (see above).

This computes the degree of the Lobatto-Kronrod integration rule.

```
In[47]:= p = If[OddQ[n], 3 * n - 3, 3 * n - 2]
Out[47]= 12
```

This defines a function.

```
In[48]:= f[x_] := x^p
```

The command below implements the integration rule weighted sums for the integral estimate, $\sum_{i=1}^{2n-1} w_i f(x_i)$, and the error estimate, $\sum_{i=1}^{2n-1} e_i f(x_i)$, where $\{x_i\}_{i=1}^{2n-1}$ are the abscissas, $\{w_i\}_{i=1}^{2n-1}$ are the weights, and $\{e_i\}_{i=1}^{2n-1}$ are the error weights.

These are the integral and error estimates for $\int_0^1 f(x)\,dx$ computed with the rule.

```
In[49]:= Total@MapThread[{f[#1] #2, f[#1] #3} &, {absc, weights, errweights}]
Out[49]= {0.0769230769230769213, -0.0011566945263191618}
```

The preceding integral estimate coincides with the exact result.

```
In[50]:= N[Integrate[f[x], {x, 0, 1}], precision]
Out[50]= 0.076923076923076923077
```

The preceding error estimate is not zero since the embedded Lobatto rule is exact for polynomials of degree $\leq 2n-3$. If we integrate a polynomial of that degree, the error estimate becomes zero.

This defines a function.

```
In[51]:= f[x_] := x^(2 n-3)
```

These are the integral and error estimates for $\int_0^1 f(x)\,dx$ computed with the rule.

```
In[52]:= Total@MapThread[{f[#1] #2, f[#1] #3} &, {absc, weights, errweights}]
Out[52]= {0.124999999999999999964, -7.×10^-19}
```

The exact result using `Integrate`.

```
In[53]:= N[Integrate[f[x], {x, 0, 1}], precision]
Out[53]= 0.12500000000000000000
```

"ClenshawCurtisRule"

A Clenshaw-Curtis rule uses sampling points derived from the Chebyshev polynomial approximation of the integrand.

The Clenshaw-Curtis quadrature for `NIntegrate` can specified with the `Method` option value `"ClenshawCurtisRule"`.

```
In[54]:= NIntegrate[1 / Sqrt[x], {x, 0, 1}, Method → "ClenshawCurtisRule"]
Out[54]= 2.
```

option name	default value	
"Points"	5	number of coarse Clenshaw-Curtis points
"SymbolicProcessing"	Automatic	number of seconds to do symbolic processing

`"ClenshawCurtisRule"` options.

Theoretically a Clenshaw-Curtis rule with n sampling points is exact for polynomials of degree n or less. In practice, though, Clenshaw-Curtis rules achieve the accuracy of the Gaussian rules [Evans93][OHaraSmith68]. The error of the Clenshaw-Curtis formula is analyzed in [OHaraSmith68].

The sampling points of the classical Clenshaw-Curtis rule are zeros of Chebyshev polynomials. The sampling points of a practical Clenshaw-Curtis rule are chosen to be Chebyshev polynomial extremum points. The classical Clenshaw-Curtis rules are not progressive, but the practical Clenshaw-Curtis rules are [DavRab84][KrUeb98].

Let $\mathrm{PCC}(f, n)$ denote a practical Clenshaw-Curtis rule of n sampling points for the function f.

The progressive property means that the sampling points of $\mathrm{PCC}(f, n)$ are a subset of the sampling points of $\mathrm{PCC}(f, 2n-1)$. Hence the difference $|\mathrm{PCC}(f, 2n-1) - \mathrm{PCC}(f, n)|$ can be taken to be an error estimate of the integral estimate $\mathrm{PCC}(f, 2n-1)$, and can be computed without extra integrand evaluations.

The `NIntegrate` option `Method -> {"ClenshawCurtisRule", "Points" -> k}` uses a practical Clenshaw-Curtis rule with $2n-1$ points $\mathrm{PCC}(f, 2n-1)$.

```
In[55]:= NIntegrate[Sqrt[x], {x, 0, 1}, Method -> {"ClenshawCurtisRule", "Points" -> 10}]
Out[55]= 0.666667
```

This example shows the number of sampling points used by `NIntegrate` with various values of the `"ClenshawCurtisRule"` option `"Points"`.

```
In[56]:= Table[
           (k = 0; NIntegrate[x^10, {x, 0, 1}, Method → {"ClenshawCurtisRule", "Points" → i},
             EvaluationMonitor :> k++]; k), {i, 3, 20}]

Out[56]= {208, 226, 79, 83, 35, 41, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39}
```

"ClenshawCurtisRule" Sampling Points and Weights

Here are the sampling points and the weights of the Clenshaw-Curtis rule for a given coarse number of points and precision.

```
In[57]:= n = 5; precision = 20;
         {absc, weights, errweights} = NIntegrate`ClenshawCurtisRuleData[n, precision]

Out[58]= {{0.×10^-21, 0.03806023374435662194, 0.14644660940672623780,
            0.30865828381745511414, 0.50000000000000000000, 0.69134171618254488586,
            0.85355339059327376220, 0.96193976625564337806, 1.00000000000000000000},
           {0.00793650793650793651, 0.07310932460800907751, 0.13968253968253968254,
            0.18085892936024489075, 0.19682539682539682540, 0.18085892936024489075,
            0.13968253968253968254, 0.07310932460800907751, 0.00793650793650793651},
           {-0.02539682539682539683, 0.07310932460800907751, -0.12698412698412698413,
            0.18085892936024489075, -0.20317460317460317460, 0.18085892936024489075,
            -0.12698412698412698413, 0.07310932460800907751, -0.02539682539682539683}}
```

Here is another way to compute the sampling points of $\mathrm{PCC}(f, 2n-1)$.

```
In[59]:= nn = 2 n - 1;
         N[1/2 Table[Cos[π/(nn - 1) i], {i, nn - 1, 0, -1}] + 1/2, precision]

Out[60]= {0, 0.038060233744356621936, 0.14644660940672623780,
          0.30865828381745511414, 0.50000000000000000000, 0.69134171618254488586,
          0.85355339059327376220, 0.96193976625564337806, 1.0000000000000000000}
```

This defines a function.

```
In[61]:= f[x_] := x^(2 n-1)
```

These are the integral and error estimates for $\int_0^1 f(x)\,dx$ computed with the rule.

```
In[62]:= Total@MapThread[{f[#1] #2, f[#1] #3} &, {absc, weights, errweights}]

Out[62]= {0.10000000000000000000, 0.0017578125000000000}
```

The exact value by `Integrate`.

```
In[63]:= Integrate[f[x], {x, 0, 1}]

Out[63]= 1/10
```

"MultiPanelRule"

"MultiPanelRule" combines into one rule the applications of a one-dimensional integration rule over two or more adjacent intervals. An application of the original rule to any of the adjacent intervals is called a panel.

Here is an example of an integration with "MultiPanelRule".

```
In[64]:= NIntegrate[1 / Sqrt[x], {x, 0, 1},
          Method → {"MultiPanelRule", Method -> "GaussKronrodRule", "Panels" -> 3}]
Out[64]= 2.
```

option name	default value	
Method	"NewtonCotesRule"	integration rule specification that provides the abscissas, weights, and error weights for a single panel
"Panels"	5	number of panels
"SymbolicProcessing"	Automatic	number of seconds to do symbolic processing

"MultiPanelRule" options.

Let the unit interval $[0, 1]$ be partitioned into k sub-intervals with the points $0 = y_0 < y_1 < \ldots < y_k = 1$.

If we have the rule

$$R(f) = \sum_{i=1}^{n} w_i\, f(x_i) \approx \int_0^1 f(x)\, dx,$$

it can be transformed into a rule for the interval $[y_{j-1}, y_j]$,

$$\frac{1}{y_j - y_{j-1}} \sum_{i=1}^{n} w_i\, f\left(x_i \left(y_j - y_{j-1}\right) + y_{j-1}\right) \approx \int_{y_{j-1}}^{y_j} f(x)\, dx,\ j = 1, \ldots, k.$$

Let $x_{i\,j} = x_i \left(y_j - y_{j-1}\right) + y_{j-1}$, and $y_j - y_{j-1} = 1/k$, $j = 1, \ldots, k$. Then the k-panel integration rule based on $R(f)$ can be written explicitly as

$$k \times R(f) = \sum_{j=1}^{k} \frac{1}{y_j - y_{j-1}} \sum_{i=1}^{n} w_i\, f\left(x_{i\,j}\right) = \frac{1}{k} \sum_{j=1}^{k} \sum_{i=1}^{n} w_i\, f\left(x_{i\,j}\right).$$

If $R(f)$ is closed, that is, $R(f)$ has 0 and 1 as sampling points, then $x_{n\,j-1} = x_{1\,j}$, and the number of sampling points of $k \times R(f)$ can be reduced to $k\,(n-1)+1$. (This is done in the implementation of "MultiPanelRule".)

More about the theory of multi-panel rules, also referred to as compounded or composite rules, can be found in [KrUeb98] and [DavRab84].

"MultiPanelRule" Sampling Points and Weights

The sampling points and the weights of the "MultiPanelRule" can be obtained with this command.

```
In[65]:= npanels = 3;
         NIntegrate`MultiPanelRuleData[
          {"GaussKronrodRule", "Points" -> 2}, npanels, MachinePrecision]

Out[66]= {{0.0123633, 0.0704416, 0.166667, 0.262892, 0.32097, 0.345697, 0.403775,
           0.5, 0.596225, 0.654303, 0.67903, 0.737108, 0.833333, 0.929558, 0.987637},
          {0.0329966, 0.0818182, 0.103704, 0.0818182, 0.0329966, 0.0329966, 0.0818182, 0.103704,
           0.0818182, 0.0329966, 0.0329966, 0.0818182, 0.103704, 0.0818182, 0.0329966},
          {0.0329966, -0.0848485, 0.103704, -0.0848485, 0.0329966, 0.0329966, -0.0848485, 0.103704,
           -0.0848485, 0.0329966, 0.0329966, -0.0848485, 0.103704, -0.0848485, 0.0329966}}
```

Here are the abscissas and weights of a Gauss-Kronrod rule.

```
In[67]:= {absc, weights, errweights} = NIntegrate`GaussKronrodRuleData[2, MachinePrecision]

Out[67]= {{0.03709, 0.211325, 0.5, 0.788675, 0.96291},
          {0.0989899, 0.245455, 0.311111, 0.245455, 0.0989899},
          {0.0989899, -0.254545, 0.311111, -0.254545, 0.0989899}}
```

The multi-panel rule abscissas can be obtained using Rescale.

```
                                                                                  1
In[68]:= Join @@ Map[Rescale[absc, {0, 1}, #] &, Partition[Range[0, npanels] ─────────, 2, 1]]
                                                                               npanels

Out[68]= {0.0123633, 0.0704416, 0.166667, 0.262892, 0.32097, 0.345697, 0.403775,
          0.5, 0.596225, 0.654303, 0.67903, 0.737108, 0.833333, 0.929558, 0.987637}
```

This shows how to derive the multi-panel rule weights from the original weights.

```
            1
In[69]:= ───────── Join @@ Table[weights, {npanels}]
          npanels

Out[69]= {0.0329966, 0.0818182, 0.103704, 0.0818182, 0.0329966, 0.0329966, 0.0818182,
          0.103704, 0.0818182, 0.0329966, 0.0329966, 0.0818182, 0.103704, 0.0818182, 0.0329966}
```

"CartesianRule"

A d-dimensional Cartesian rule has sampling points that are a Cartesian product of the sampling points of d one-dimensional rules. The weight associated with a Cartesian rule sampling point is the product of the one-dimensional rule weights that correspond to its coordinates.

The Cartesian product integration for `NIntegrate` can be specified with the `Method` option value `"CartesianRule"`.

```
In[70]:= NIntegrate[1 / Sqrt[x + y + z], {x, 0, 1},
          {y, 0, 1}, {z, 0, 1}, Method -> "CartesianRule"]
Out[70]= 0.862877
```

option name	default value	
`Method`	`"GaussKronrodRule"`	a rule or a list of rules with which the Cartesian product rule will be formed
`"SymbolicProcessing"`	`Automatic`	number of seconds to do symbolic preprocessing

`"CartesianRule"` options.

For example, suppose we have the formulas:

$$\int_0^1 f_1(x)\,dx \approx \sum_{i=1}^{n_1} w_i^1\, f_1(\alpha_i^1),$$

$$\int_0^1 f_2(x)\,dx \approx \sum_{i=1}^{n_2} w_i^2\, f_2(\alpha_i^2),$$

$$\int_0^1 f_3(x)\,dx \approx \sum_{i=1}^{n_3} w_i^3\, f_3(\alpha_i^3).$$

that are exact for polynomials of degree d_1, d_2, and d_3, respectively. Then it is not difficult to see that the formula with $n_1 \times n_2 \times n_3$ points,

$$\int_0^1 \int_0^1 \int_0^1 f(x, y, z)\,dx\,dy\,dz \approx \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} \sum_{k=1}^{n_3} w_{1\,i}\, w_{2\,j}\, w_{3\,k}\, f(\alpha_{1\,i}, \alpha_{2\,j}, \alpha_{3\,k})$$

is exact for polynomials in $x_1,\ x_2,\ x_3$ of degree $\min(d_1, d_2, d_3)$. Note that the weight associated with the abscissa $\{\alpha_i^1, \alpha_i^2, \alpha_i^3\}$ is $w_i^1\, w_i^2\, w_i^3$.

The general Cartesian product formula for D one-dimensional rules the i of which has n_i sampling points $\{\alpha_j^i\}_{j=1}^{n_i}$ and weights $\{w_j^i\}_{j=1}^{n_i}$ is

$$\int_V f(x_1, \ldots, x_d)\, dx_1 \ldots dx_D \approx \sum_{i_1=1}^{n_1}\left(\ldots \sum_{i_D=1}^{n_D}\left(\prod_{k=1}^{D} w_{i_k}^k\right) f\left(\alpha_{i_1}^1, \ldots, \alpha_{i_D}^D\right)\right). \tag{37}$$

Clearly (1) can be written as

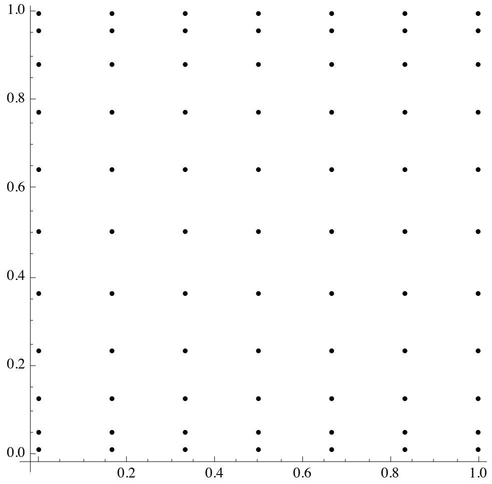
$$\int_V f(x)\, dx \approx \sum_{i_D=1}^{n} w_i\, f(\alpha_i), \tag{38}$$

where $n = \prod_{k=1}^{D} n_k$, and for each integer $k \in [1, n]$, $\alpha_k = \{\alpha_i^1, \ldots, \alpha_i^k\}$ and $w_k = \prod_{k=1}^{D} w_{i_k}^k$.

Here is a visualization of a Cartesian product rule integration. Along the x axis "TrapezoidalRule" is used; along the y axis "GaussKronrodRule" is used.

```
In[71]:= pnts = Reap[NIntegrate[x + y^9, {x, 0, 1}, {y, 0, 1}, Method ->
                {{"TrapezoidalRule", "Points" -> 4}, {"GaussKronrodRule", "Points" -> 5}},
              EvaluationMonitor :> Sow[{x, y}]]][[2, 1]];
         Graphics[Point /@ pnts, AspectRatio -> 1, Axes → True, AxesOrigin → {-0.02, -0.02}]
```

Out[72]=



Cartesian rules are applicable for relatively low dimensions (≤ 4), since for higher dimensions they are subject to "combinatorial explosion." For example, a five-dimensional Cartesian product of 5 identical one-dimensional rules each having 10 sampling points would have 10^5 sampling points.

`NIntegrate` uses Cartesian product rule if the integral is multidimensional and the `Method` option is given a one-dimensional rule or a list of one-dimensional rules.

Here is an example specifying Cartesian product rule integration with `GaussKronrodRule`.

```
In[73]:= NIntegrate[x + y, {x, 0, 1}, {y, 0, 1}, Method -> "GaussKronrodRule"]

Out[73]= 1.
```

Here is an example specifying Cartesian product rule integration with a list of one-dimensional integration rules.

```
In[74]:= NIntegrate[x + y, {x, 0, 1}, {y, 0, 1},
          Method -> {"LobattoKronrodRule", "GaussKronrodRule"}]

Out[74]= 1.
```

Another example specifying Cartesian product rule integration with a list of one-dimensional integration rules.

```
In[75]:= NIntegrate[x + y^3, {x, 0, 1}, {y, 0, 1}, Method ->
            {{"TrapezoidalRule", "Points" -> 8}, {"GaussKronrodRule", "GaussPoints" -> 12}}]

Out[75]= 0.75
```

More about Cartesian rules can be found in [Stroud71].

"CartesianRule" Sampling Points and Weights

The sampling points and the weights of the `"CartesianRule"` rule can be obtained with the command `NIntegrate`CartesianRuleData`.

```
In[76]:= crule = NIntegrate`CartesianRuleData[{{"GaussKronrodRule", "GaussPoints" → 2},
             {"TrapezoidalRule", "Points" → 2}}, MachinePrecision]

Out[76]= NIntegrate`CartesianRule[{{{0.03709, 0.211325, 0.5, 0.788675, 0.96291}, {0., 0.5, 1.}},
           {{0.0989899, 0.245455, 0.311111, 0.245455, 0.0989899}, {0.166667, 0.666667, 0.166667}},
           {{0.0989899, -0.254545, 0.311111, -0.254545, 0.0989899}, {-0.0833333, 0.166667, -0.0833333}}}]
```

`NIntegrate`CartesianRuleData` keeps the abscissas and the weights of each rule separated. Otherwise, as it can be seen from (38) the result might be too big for higher dimensions.

The results of NIntegrate`CartesianRuleData can be put into the form of (38) with this function.

```
In[77]:= productFunc = MapAt[Flatten[Outer[Times, Sequence @@ #]] &, #, {1, 3}] &@
               MapAt[Flatten[Outer[Times, Sequence @@ #]] &, #, {1, 2}] &@
             MapAt[Flatten[Outer[List, Sequence @@ #], Length[#] - 1] &, #, {1, 1}] &;
```

```
In[78]:= productFunc[crule]
```

```
Out[78]= NIntegrate`CartesianRule[
          {{{0.03709, 0.}, {0.03709, 0.5}, {0.03709, 1.}, {0.211325, 0.}, {0.211325, 0.5},
             {0.211325, 1.}, {0.5, 0.}, {0.5, 0.5}, {0.5, 1.}, {0.788675, 0.},
             {0.788675, 0.5}, {0.788675, 1.}, {0.96291, 0.}, {0.96291, 0.5}, {0.96291, 1.}},
           {0.0164983, 0.0659933, 0.0164983, 0.0409091, 0.163636, 0.0409091, 0.0518519, 0.207407,
            0.0518519, 0.0409091, 0.163636, 0.0409091, 0.0164983, 0.0659933, 0.0164983},
           {-0.00824916, 0.0164983, -0.00824916, 0.0212121, -0.0424242, 0.0212121, -0.0259259, 0.0518519,
            -0.0259259, 0.0212121, -0.0424242, 0.0212121, -0.00824916, 0.0164983, -0.00824916}}]
```

"MultiDimensionalRule"

A fully symmetric integration rule for the cube $\left[-\frac{1}{2}, \frac{1}{2}\right]^d$, $d \in \mathbb{N}$, $d > 1$ consists of sets of points with the following properties: (i) all points in a set can be generated by permutations and/or sign changes of the coordinates of any fixed point from that set; (ii) all points in a set have the same weight associated with them.

The fully symmetric multidimensional integration (fully symmetric cubature) for NIntegrate can be specified with the Method option value "MultiDimensionalRule".

```
In[79]:= NIntegrate[1 / Sqrt[x + y], {x, 0, 1}, {y, 0, 1}, Method → "MultiDimensionalRule"]
```

```
Out[79]= 1.10457
```

A set of points of a fully symmetric integration rule that satisfies the preceding properties is called an orbit. A point of an orbit, $\{x_1, x_2, \ldots, x_d\}$, for the coordinates of which the inequality $x_1 \geq x_2 \geq \ldots \geq x_d$ holds, is called a generator. (See [KrUeb98][GenzMalik83].)

option name	default value	
"Generators"	5	number of generators of the fully symmetric rule
"SymbolicProcessing"	Automatic	number of seconds to do symbolic preprocessing

"MultiDimensionalRule" options.

If an integration rule has K orbits denoted $\Omega_1, \Omega_2, \ldots, \Omega_K$, and the i^{th} of them, Ω_i, has a weight w_i associated with it, then the integral estimate is calculated with the formula

$$\int_{\left[-\frac{1}{2},\frac{1}{2}\right]^d} f(X)\, dX \approx \sum_{i=1}^{K} w_i \sum_{X_j \in \Omega_i} f\left(X_j\right).$$

A null rule of degree m will integrate to zero all monomials of degree $\leq m$ and will fail to do so for at least one monomial of degree $m+1$. Each null rule may be thought of as the difference between a basic integration rule and an appropriate integration of lower degree.

The "MultiDimensionalRule" object of NIntegrate is basically an interface to three different integration rule objects that combine an integration rule and one or several null rules. Their number of generators and orders are summarized in the table below. The rule objects with 6 and 9 generators use three null rules, each of which is a linear combination of two null rules. The null rule linear combinations are used in order to avoid phase errors. See [BerntEspGenz91] for more details about how the null rules are used.

Number of generators and orders of the fully symmetric rules of NIntegrate:

Number of Generators	Integration Rule Order	Order of Each of the Null Rules	Described in
5	7	5	[GenzMalik80]
6	7	5, 3, 1	[GenzMalik83] [BerntEspGenz91]
9	9	7, 5, 3	[GenzMalik83] [BerntEspGenz91]

This is the number of sampling points used by NIntegrate with its fully symmetric multidimensional integration rules for integrals of the form $\int_0^1 \int_0^1 (x^m + y^m)\, dy\, dx$, $m = 1, \ldots, 20$.

```
In[80]:= tbl = Table[Prepend[Table[
             (k = 0; NIntegrate[x^m + y^m, {x, 0, 1}, {y, 0, 1},
               Method -> {"MultiDimensionalRule", "Generators" -> gen},
               EvaluationMonitor :> k++]; k), {gen, {5, 6, 9}}], m], {m, 1, 20}];
         Grid[Join[{{"Monomial", "Number of generators", SpanFromLeft, SpanFromLeft},
             {"degree", "5", "6", "9"}}, tbl],
          Dividers -> {{False, True, False}, {False, False, True, False}},
          Alignment -> {Center}]
```

```
Monomial | Number of generators
 degree  |  5       6        9
---------+----------------------
    1    | 17      21       33
    2    | 17   426 615     33
    3    | 17   206 157     33
    4    | 17      21     21 417
```

Out[81]=

```
 5   |  17     21     39 897
 6   |  527    651      33
 7   | 1003    903      33
 8   | 1241   1281     231
 9   | 1445   1617     429
10   | 1717   1785     561
11   | 3145   3045     561
12   | 3689   3297     561
13   | 3825   3843     561
14   | 3825   3843     825
15   | 4063   3591     957
16   | 3893   2247    1089
17   | 3961   2205    1155
18   | 3995   3297    1155
19   | 4403   3255    1155
20   | 6035   4137    1155
```

"MultiDimensionalRule" Sampling Points and Weights

This subsection gives an example of a calculation of an integral estimate with a fully symmetric multidimensional rule.

Here is the parameter for the number of generators.

In[82]:=

```
numberOfGenerators = 9;
```

This function takes a generator point and creates its orbit.

In[83]:=

```
MakeOrbit[generator_] :=
  Module[{perms, signs, gperms, len = Length[generator]},
   perms = Permutations[Range[len]];
   signs = Flatten[Outer[List, Sequence @@ Table[{1, -1}, {len}]], len - 1];
   gperms = Map[Part[generator, #1] &, perms];
   Union[Flatten[Outer[Times, gperms, signs, 1], 1]]
  ];
```

The generators and weights for given number of generators.

In[84]:=

```
dimension = 2;
precision = MachinePrecision;
rdata =
  NIntegrate`MultiDimensionalRuleData[numberOfGenerators, precision, dimension];
generators = rdata[[1, 1]];
weights = rdata[[1, 2]];
```

This computes the orbit of each generator.

In[89]:=

```
orbits = MakeOrbit /@ generators;
```

This defines a function.

In[90]:=

```
Clear[f]
f[x_, y_] := x^3 * y^3
```

This applies the multidimensional rule.

```
In[92]:= Total@MapThread[Total[Map[f @@ (#1 + 1 / 2) &, #1] * #2] &, {orbits, weights}] //
          InputForm
Out[92]//InputForm= 0.06250000000000001
```

Here is the exact result.

```
In[93]:= Integrate[f[x, y], {x, 0, 1}, {y, 0, 1}] // N // InputForm
Out[93]//InputForm= 0.0625
```

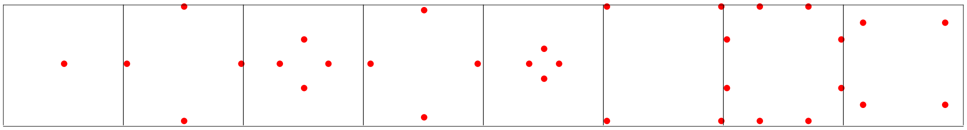
This makes graphics primitives for points of the orbits.

```
In[94]:= graphs = Graphics[{Red, AbsolutePointSize[4], Point /@ #1},
                Axes -> False, AspectRatio -> 1, Frame -> True, FrameTicks → None,
                PlotRange → {{-1, 1}, {-1, 1}} / 2, ImageSize → {75, 75}] & /@ orbits;
```

Here is how the different orbits look.

```
In[95]:= Row[graphs]
```

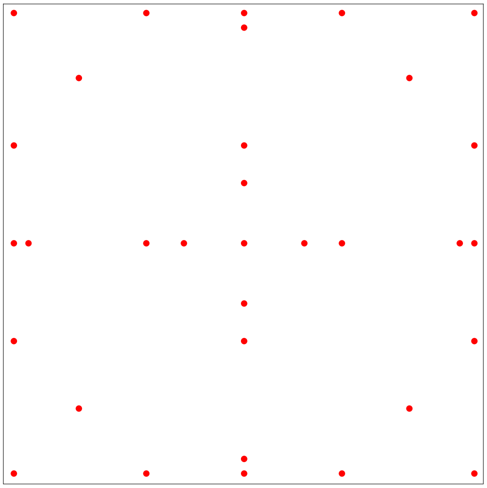
Out[95]=



Here are all rule points together.

```
In[96]:= Graphics[First /@ graphs, Frame -> True, FrameTicks → None]
```

Out[96]=



"MonteCarloRule"

A Monte Carlo rule estimates an integral by forming a uniformly weighted sum of integrand evaluations over random (quasi-random) sampling points.

Here is an example of using "MonteCarloRule" with 1000 sampling points.

```
In[97]:=  NIntegrate[(ⅇ^x - 1)/(ⅇ - 1), {x, 0, 1}, Method → {"MonteCarloRule", "Points" → 1000}]

Out[97]=  0.413394
```

option name	*default value*	
"Points"	100	number of sampling points
"PointGenerator"	Random	sampling points coordinates generator
"AxisSelector"	Automatic	selection algorithm of the splitting axis when global adaptive Monte Carlo integration is used
"SymbolicProcessing"	Automatic	number of seconds to do symbolic preprocessing

"MonteCarloRule" options.

In Monte Carlo methods [KrUeb98], the d-dimensional integral $\int_V f(x)\,dx$ is interpreted as the following expected (mean) value

$$\int_V f(x)\,dx = \mathrm{vol}(V) \int_{\mathbb{R}^d} \frac{1}{\mathrm{vol}(V)} \,\mathrm{Boole}(x \in V)\, f(x)\,dx = \mathrm{vol}(V)\, E(f), \tag{39}$$

where $E(f)$ is the mean value of the function f interpreted as a random variable, with respect to the uniform distribution on V, that is, the distribution with probability density $\mathrm{vol}(V)^{-1}\,\mathrm{Boole}(x \in V)$. $\mathrm{Boole}(x \in V)$ denotes the characteristic function of the region V; $\mathrm{vol}(V)$ denotes the volume of V.

The crude Monte Carlo estimate of the expected value $E(f)$ is obtained by taking n independent random vectors $\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_n \in \mathbb{R}^d$ with density $\mathrm{vol}(V)^{-1}\,\mathrm{Boole}(x \in V)$ (that is, the vectors are uniformly distributed on V), and making the estimate

$$\mathrm{MC}(f, n) = \frac{1}{n} \sum_{i=1}^{n} f(x_i). \tag{40}$$

Remark: The function $\mathrm{vol}(V)^{-1}\,\mathrm{Boole}(x \in V)$ is a valid probability density function because it is non-negative on the whole of $\mathbb{R}^d$ and $\int_{\mathbb{R}^d} \mathrm{vol}(V)^{-1}\,\mathrm{Boole}(x \in V)\,dx = 1$.

According to the strong law of large numbers, the convergence

$$\mathrm{MC}(f, n) \to \mu(f),\ n \to \infty,$$

happens with probability 1. The strong law of large numbers does not provide information for the error $\mathrm{MC}(f, n) - \int_V f(x)\,dx$, so a probabilistic estimate is used.

Let ϑ be defined as

$$\vartheta = \int_V f(x)\,dx.$$

Formula (40) is an unbiased estimator of ϑ(that is, the expectation of $\mathrm{MC}(f, n)$ for various sets of $\{x_i\}_{i=1}^n$ is ϑ) and its variance is

$$\frac{1}{n}\int_V (f(x) - \vartheta)^2\,dx = \frac{\mathrm{Var}(f)}{n},$$

where $\mathrm{Var}(f)$ denotes the variance of f, The standard error of $\mathrm{MC}(f, n)$ is thus $\frac{\sqrt{\mathrm{Var}(f)}}{\sqrt{n}}$.

In practice the $\mathrm{Var}(f)$ is not known, so it is estimated with the formula

$$s^2 = \frac{1}{n-1}\sum_{i=1}^n (f(x_i) - \mathrm{MC}(f, n))^2.$$

The standard error of $\mathrm{MC}(f, n)$ is then

$$\mathrm{SD}(f, n) = \frac{s}{\sqrt{n}} = \frac{1}{\sqrt{n\,(n-1)}}\sqrt{\textstyle\sum_{i=1}^n (f(x_i) - \mathrm{MC}(f, n))^2}\ . \tag{41}$$

The result of the Monte Carlo estimation can be written as $\mathrm{MC}(f, n) \pm \mathrm{SD}(f, n)$.

It can be seen from Equation (41) that the convergence rate of the crude Monte Carlo estimation does not depend on the dimension d of the integral, and if n sampling points are used then the convergence rate is $\sqrt{n}$.

The NIntegrate integration rule "MonteCarloRule" calculates the estimates $\mathrm{MC}(f, n)$ and $\mathrm{SD}(f, n)$.

The estimates can be improved incrementally. That is, if we have the estimates $\mathrm{MC}(f, n_0)$ and $\mathrm{SD}(f, n_0)$, and a new additional set of sample function values $\{f_1, f_2, \ldots, f_{n_1}\}$, then using (40) and (41) we have

$$\mathrm{MC}(f, n_0 + n_1) = \frac{1}{n_0 + n_1}\left(\mathrm{MC}(f, n_0)\, n_0 + \sum_{i=1}^{n_1} f_i\right),$$

$$\mathrm{SD}(f, n_0 + n_1) = \frac{1}{\sqrt{(n_0 + n_1)(n_0 + n_1 - 1)}}\left((n_0 - 1)\, n_0\, \mathrm{SD}(f, n_0)^2 + \sum_{i=1}^{n_1} (f_i - \mathrm{MC}(f, n_0 + n_1))^2\right)^{\frac{1}{2}}.$$

To compute the estimates $\mathrm{MC}(f, n_0 + n_1)$ and $\mathrm{SD}(f, n_0 + n_1)$, it is not necessary to know the random points used to compute the estimates $\mathrm{MC}(f, n_0)$ and $\mathrm{SD}(f, n_0)$.

"AxisSelector"

When used for multidimensional global adaptive integration, "MonteCarloRule" chooses the splitting axis of an integration subregion it is applied to in two ways: (i) by random selection or (ii) by minimizing the sum of the variances of the integral estimates of each half of the subregion, if the subregion is divided along that axis. The splitting axis is selected after the integral estimation.

The random axis selection is done in the following way. "MonteCarloRule" keeps a set of axes for selection, A. Initially A contains all axes. An element of A is randomly selected. The selected axis is excluded from A. After the next integral estimation, an axis is selected from A and excluded from it, and so forth. If A is empty, it is filled up with all axes.

The minimization of variance axis selection is done in the following way. During the integration over the region, a subset of the sampling points and their integrand values is stored. Then for each axis, the variances of the two subregions that the splitting along this axis will produce are estimated using the stored sampling point and corresponding integrand values. The axis for which the sum of these variances is minimal is chosen to be the splitting axis, since this would mean that if the region is split on that axis, the new integration error estimate will be minimal. If it happens that for some axis all stored points are clustered in one of the half-regions, then that axis is selected for splitting.

option value	
`Random`	random splitting axis election
`MinVariance` | `{MinVariance, "SubsampleFraction"->`*frac*`}`	splitting axis selection that minimizes the sum of variances of the new regions

`"AxisSelector"` options.

option name	*default value*	
`"SubsampleFraction"`	`1/10`	fraction of the sampling points used to determine the splitting axis

`MinVariance` option.

This is an example of using `"MonteCarloRule"`'s option `"AxisSelector"`.

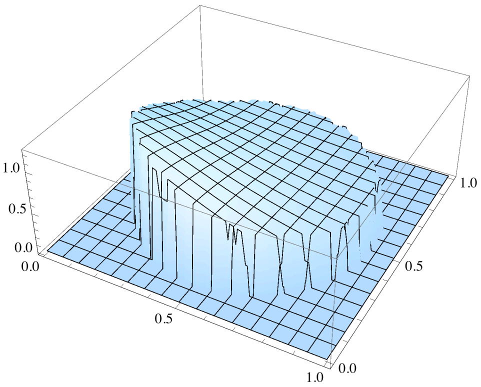
```
In[98]:= t = NIntegrate[Exp[-((x - 1 / 2)^2 + (y - 1 / 2)^2)], {x, 0, 1},
           {y, 0, 1}, Method → {"MonteCarloRule", "AxisSelector" → Random}]
Out[98]= 0.85354
```

In the examples below the two axis selection algorithms are compared. In general, the minimization of variance selection uses less number of sampling points. Nevertheless, using the minimization of variance axis selection slows down the application of `"MonteCarloRule"`. So for integrals for which both axis selection methods would result in the same number of sampling points, it is faster to use random axis selection. Also, using larger fraction sampling points to determine the splitting axis in minimization of variance selection makes the integration slower.

Consider the following function.

```
In[2]:= f[x_?NumberQ, y_?NumberQ] := Boole[Abs[(x - 1/2)^2 + (y - 1/2)^2] < 1/6] (ⅇ^(-3 (x^2+y^2)) + 1/2);
        Plot3D[f[x, y], {x, 0, 1}, {y, 0, 1}, PlotRange -> All, PlotPoints -> 20]
```

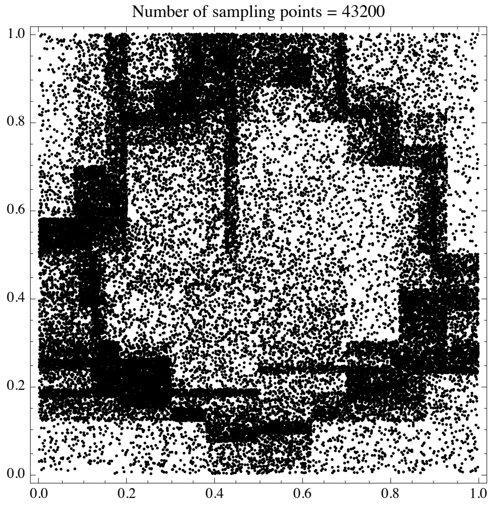
Out[3]=



These are the adaptive Monte Carlo integration sampling points for the function above with random choice of splitting axis.

```
In[4]:= t = Reap[NIntegrate[f[x, y], {x, 0, 1}, {y, 0, 1}, Method → {"AdaptiveMonteCarlo",
               Method → {"MonteCarloRule", "AxisSelector" → Random}}, MinRecursion → 1,
             PrecisionGoal → 2.8, EvaluationMonitor :> Sow[{x, y}]]][[2, 1]];
        Graphics[{PointSize[0.006], Point[t]}, AspectRatio -> 1, Frame -> True,
         PlotLabel -> "Number of sampling points = " <> ToString[Length[t]]]
```

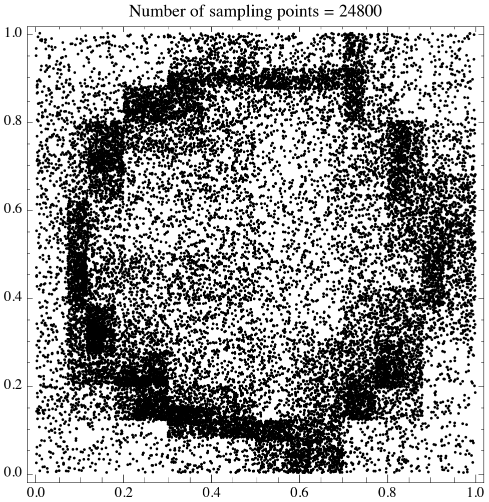
Out[5]=



These are the sampling points with choice of splitting axes that minimize the variance. Compared to the previous Monte Carlo integration, the sampling points of this one are more concentrated around the circle $(x-1/2)^2+(y-1/2)^2=1/6$, and their number is nearly twice as small.

```
In[6]:=  t = Reap[NIntegrate[f[x, y], {x, 0, 1}, {y, 0, 1},
             Method → {"AdaptiveMonteCarlo", Method → {"MonteCarloRule", "AxisSelector" →
                   {"MinVariance", "SubsampleFraction" → 1 / 3}}}, MinRecursion → 1,
             PrecisionGoal → 2.8, EvaluationMonitor :> Sow[{x, y}]]][[2, 1]];
         Graphics[{PointSize[0.006], Point[t]}, AspectRatio -> 1,
          Frame -> True,
          PlotLabel -> "Number of sampling points = " <> ToString[Length[t]]]
```

Out[6]=



Number of sampling points = 24800

Here is an adaptive Monte Carlo integration that uses random axis selection.

```
In[104]:=  Do[NIntegrate[$\frac{1}{\sqrt{x^2 + y^2}}$, {x, -1, 2}, {y, -1, 2},
              Method → {"AdaptiveMonteCarlo", Method → {"MonteCarloRule",
                    "Points" -> 500, "AxisSelector" → Random}}], {100}] // Timing

Out[104]=  {4.21036, Null}
```

Here is an adaptive Monte Carlo integration for the preceding integral that uses the minimization of variance axis selection and is slower than using random axis selection.

```
In[105]:=  Do[NIntegrate[$\frac{1}{\sqrt{x^2 + y^2}}$, {x, -1, 2}, {y, -1, 2}, Method → {"AdaptiveMonteCarlo",
                Method → {"MonteCarloRule", "Points" -> 500, "AxisSelector" →
                     {"MinVariance", "SubsampleFraction" → 0.3}}}], {100}] // Timing

Out[105]=  {4.20636, Null}
```

Using a larger fraction of stored points for the minimization of variance axis choice slows down the integration.

```
                          1
In[106]:= Do[NIntegrate[―――――――――――, {x, -1, 2}, {y, -1, 2}, Method → {"AdaptiveMonteCarlo",
                        Sqrt[x^2 + y^2]
            Method → {"MonteCarloRule", "Points" -> 500, "AxisSelector" →
               {"MinVariance", "SubsampleFraction" → 0.6}}}], {100}] // Timing

Out[106]= {5.08623, Null}
```

Comparisons of the Rules

All integration rules, except "MonteCarloRule", are to be used by adaptive strategies in NIntegrate. Changing the type and the number of points of the integration rule component for an integration strategy will make a different integration algorithm. In general these different integration algorithms will perform differently for different integrals. Naturally the following questions arise.

1. Is there a type of rule that is better than other types for any integral or for integrals of a certain type?

2. Given an integration strategy, what rules perform better with it? For what integrals?

3. Given an integral, an integration strategy, and an integration rule, what number of points in the rule will minimize the total number of sampling points used to reach an integral estimate that satisfies the precision goal?

For a given integral and integration strategy the integration rule which achieves a result that satisfies the precision goal with the smallest number of sampling points is called the best integration rule. There are several factors that determine the best integration rule.

1. In general the higher the degree of the rule the faster the integration will be for smooth integrands and for higher-precision goals. On the other hand, the rule degree might be too high for the integrand and hence too many sampling points might be used when the adaptive strategies work around, for example, the integrand's discontinuities.

2. The error estimation functional of a rule influences significantly the total amount of work by the integration strategy. Rules with a smaller number of points might lead (i) to a wrong result because of underestimation of the integral, or (ii) to applying too many sampling points because of overestimation of the integrand. (See "Examples of Pathological Behavior".) Further, the error estimation functional might be computed with one or several embedded null rules. In general, the larger the number of the null rules the better

the error estimation—fewer phase errors are expected. The number of the null rules and the weights assigned to them in the sum that computes the error estimate determines the sets of pathological integrals and integrals hard to compute for that rule. (Some of the multidimensional rules of `NIntegrate` use several embedded null rules to compute the error estimate. All of the one-dimensional integration rules of `NIntegrate` use only one null rule.)

3. Local adaptive strategies are more effective with closed rules that have their sampling points more uniformly distributed (for example, `"ClenshawCurtisRule"`) than with open rules (for example, `GaussKronrodRule`) and closed rules that have sampling points distributed in a non-uniform way (for example, `"LobattoKronrodRule"`).

4. The percent of points reused by the strategy might greatly determine what is the best rule. For one-dimensional integrals, `"LocalAdaptive"` reuses all points of the closed rules. `"GlobalAdaptive"` throws away almost all points of the regions that need improvement of their error estimate.

Number of Points in a Rule

This subsection demonstrates with examples that the higher the degree of the rule the faster the integration will be for smooth integrands and for higher-precision goals. It also shows examples in which the degree of the rule is too high for the integrand and hence too many sampling points are used when the adaptive strategies work around the integrand's discontinuities. All examples use Gaussian rules with Berntsen-Espelid error estimate.

Here is the error of a Gaussian rule in the interval $[a, b]$.

$$E[G(f, n)] = \frac{(b-a)^{2n+1}(n!)^4}{(2n+1)[(2n)!]^3} f^{(2n)}(\xi), \; a < \xi < b.$$

(See [DavRab84].)

Here is a function that calculates the error of a rule for the integral $\int_0^1 f(x)\,dx$, using the exact value computed by `Integrate` for comparison.

```
In[107]:= RuleError[f_, rule_String, prec_, pnts_?NumberQ] :=
            Block[{absc, weights, errweights},
             {absc, weights, errweights} =
              ToExpression["NIntegrate`" <> rule <> "Data"][pnts, prec];
             Abs[Total[MapThread[f[#1] #2 &, {absc, weights}]] - Integrate[f[x], {x, 0, 1}]]
            ];
```

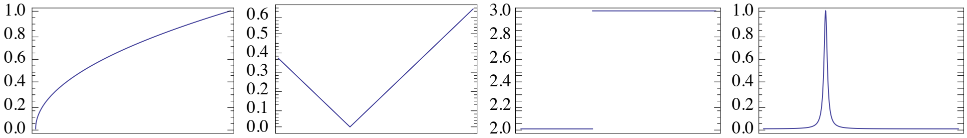
This defines a list of functions.

In[108]:=
$$\text{funcs} = \left\{\sqrt{x}\text{ , Abs}\left[x - \frac{1}{e}\right], \left[\begin{array}{ll} 2 & x \le \frac{1}{e} \\ 3 & x > \frac{1}{e}\end{array}\right., \frac{1}{10^4\left(\frac{1}{\pi}-x\right)^2+1}\right\};$$

Here are plots of the functions in the interval $[0, 1]$.

```
In[109]:= Row[Plot[#, {x, 0, 1}, PlotRange -> All, Frame -> True,
              FrameTicks -> {None, Automatic}, ImageSize → {120, 120}] & /@ funcs, "   "]
```

Out[109]=



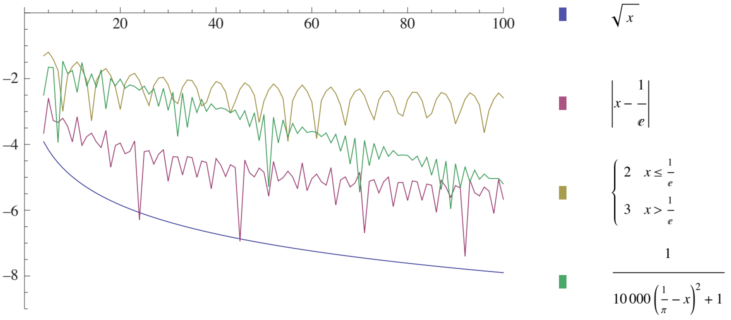
Here is the computation of the errors of "`GaussBerntsenEspelidRule`" for $\int_0^1 \sqrt{x}\,dx$, $\int_0^1 \left|x - \frac{1}{2}\right| dx$, $\int_0^1 \begin{cases} 2 & x \le \frac{1}{2} \\ 3 & x > \frac{1}{2}\end{cases} dx$, and $\int_0^1 \frac{1}{10^4\left(\frac{1}{3}-x\right)^2+1}\,dx$ for a range of points.

```
In[110]:= errors = Table[{pnts, RuleError[#, "GaussBerntsenEspelidRule", 30, pnts]},
               {pnts, 4, 100, 1}] & /@ Function /@ (Function[{f}, f /. x -> #] /@ funcs);
```

Here are plots of how the logarithm of the error decreases for each of the functions. It can be seen that the integral estimates of discontinuous functions and functions with discontinuous derivatives improve slowly when the number of points is increased.

```
In[111]:= gr = ListLinePlot[
             MapThread[Tooltip[{#[[1]], Log[10, #[[2]]]} & /@ #1, #2] &, {errors, funcs}],
             PlotRange -> {{0, 100}, {0, -9}}, AxesOrigin -> {0, 0}, ImageSize -> {300}];
          xc = 110;
          xcSq = 106;
          legend =
            {Text[funcs[[1]], {xc, -2}, {-1, 0}], Text[funcs[[2]], {xc, -4}, {-1, 0}],
             Text[funcs[[3]], {xc, -6}, {-1, 0}], Text[funcs[[4]], {xc, -8}, {-1, 0}]};
          legendSq = {Text["  ", {xcSq, -2}, {-1, 0}], Text["  ", {xcSq, -4}, {-1, 0}],
             Text["  ", {xcSq, -6}, {-1, 0}], Text["  ", {xcSq, -8}, {-1, 0}]};
          legendSq = MapThread[Append[#1, Background -> #2] &,
             {legendSq, Cases[gr, Hue[s__], ∞]}];
          Row[{gr, " ", Graphics[{ legend, legendSq},
             ImageSize -> {200, 200}, AspectRatio -> 5]}]
```

Out[117]=



Minimal Number of Sampling Points

Here is a function that finds the number of sampling points used in an integration.

```
In[118]:= Attributes[SamplingPoints] = {HoldFirst};
          SamplingPoints[expr_] :=
           Module[{k = 0, res},
            res = Hold[expr] /. HoldPattern[NIntegrate[s___]] ⧴
               NIntegrate[s, EvaluationMonitor ⧴ k++]; ReleaseHold[res]; k]
```

This finds the number of sampling points used for a range of precision goals and a range of integration rule coarse points.

```
In[19]:= tblga = Table[{pg, pnts, SamplingPoints[NIntegrate[#,
                {x, 0, 1}, Method -> {"GlobalAdaptive", "SymbolicProcessing" -> 0,
                  Method -> {"GaussBerntsenEspelidRule", "Points" -> pnts}},
                MaxRecursion -> 100, WorkingPrecision -> 35, PrecisionGoal -> pg]]},
            {pg, 4, 30}, {pnts, 4, 25}] & /@ funcs;
```

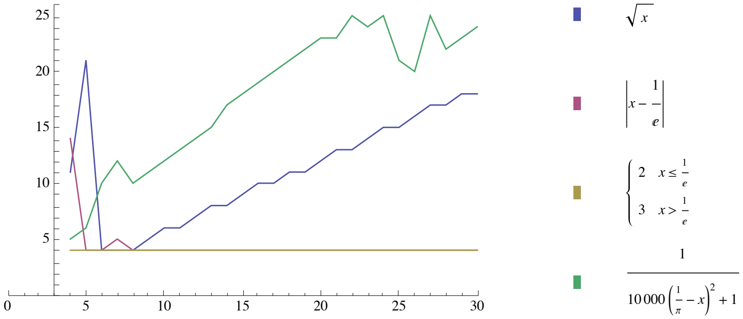
This finds the for each precision the minimum total number of sampling points. This way the number of coarse integration rule points used is also found.

```
In[121]:= minPnts = (#[[Position[#, Min[#[[3]] & /@ #]][[1, 1]]]] &[#] & /@ #) & /@ tblga;
```

This is a plot of the precision goal and the number of integration rule points with which the minimum number of total sampling points was used.

```
In[122]:= gr = ListLinePlot[(Drop[#, -1] & /@ #) & /@ minPnts, PlotRange -> {{0, 30}, {0, 26}},
             PlotStyle -> Thickness[0.003], AxesOrigin -> {3, 0}, ImageSize -> {300, 200}];
          xc = 110;
          xcSq = 106;
          legend =
            {Text[funcs[[1]], {xc, -2}, {-1, 0}], Text[funcs[[2]], {xc, -4}, {-1, 0}],
             Text[funcs[[3]], {xc, -6}, {-1, 0}], Text[funcs[[4]], {xc, -8}, {-1, 0}]};
          legendSq = {Text["  ", {xcSq, -2}, {-1, 0}], Text["  ", {xcSq, -4}, {-1, 0}],
             Text["  ", {xcSq, -6}, {-1, 0}], Text["  ", {xcSq, -8}, {-1, 0}]};
          legendSq = MapThread[Append[#1, Background -> #2] &,
             {legendSq, Cases[gr, Hue[s__], ∞]}];
          Row[{gr, " ", Graphics[{ legend, legendSq},
             ImageSize -> {200, 200}, AspectRatio -> 5]}]
```

Out[128]=



Rule Comparison

Here is a function that calculates the error of a rule for the integral $\int_0^1 f(x)\,dx$, using the exact value computed by `Integrate` for comparison.

```
In[129]:= RuleErrors[f_, rule_String, prec_, pnts_?NumberQ] :=
            Block[{absc, weights, errweights, exact},
             {absc, weights, errweights} =
              ToExpression["NIntegrate`" <> rule <> "Data"][pnts, prec];
             {Abs[Total[MapThread[f[#1] #2 &, {absc, weights}]] - Integrate[f[x], {x, 0, 1}]],
              Abs[Total[MapThread[f[#1] #2 &, {absc, errweights}]]]}
            ];
```

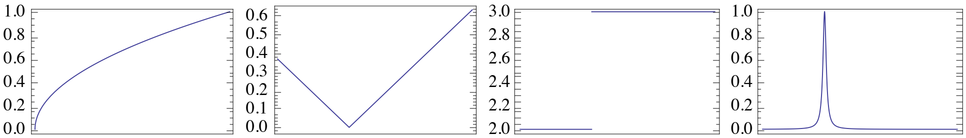
This defines a list of functions.

In[130]:=
$$\texttt{funcs} = \left\{\sqrt{x},\ \mathrm{Abs}\left[x - \frac{1}{e}\right],\ \left[\begin{array}{ll} 2 & x \le \frac{1}{e} \\ 3 & x > \frac{1}{e} \end{array}\right.,\ \frac{1}{10^4\left(\frac{1}{\pi} - x\right)^2 + 1}\right\};$$

Here are plots of the functions in the interval $[0, 1]$.

```
In[131]:= Row[Plot[#, {x, 0, 1}, PlotRange -> All, Frame -> True,
              FrameTicks -> {None, Automatic}, ImageSize -> {120, 120}] & /@ funcs, "   "]
```

Out[131]=



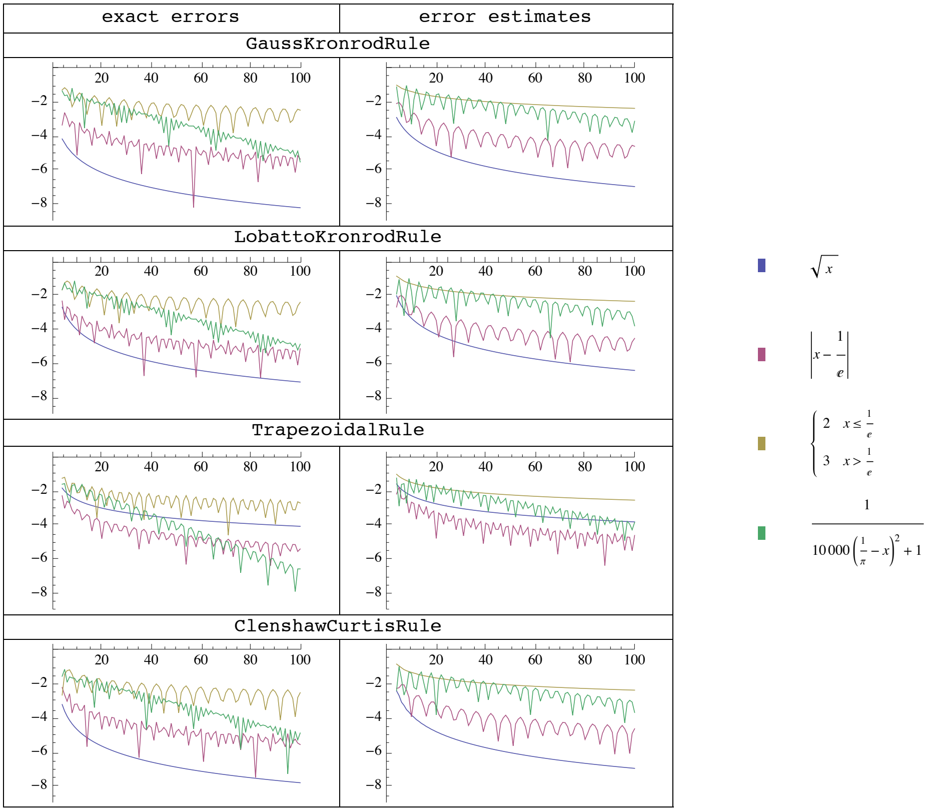
This is the computation of the errors of `"GaussKronrodRule"`, `"LobattoKronrodRule"`, `"TrapezoidalRule"`, and `"ClenshawCurtisRule"` for each of the integrals $\int_0^1 \sqrt{x}\,dx$, $\int_0^1 \left|x - \frac{1}{2}\right| dx$, $\int_0^1 \begin{cases} 2 & x \le \frac{1}{2} \\ 3 & x > \frac{1}{2} \end{cases} dx$, and $\int_0^1 \frac{1}{10^4\left(\frac{1}{3} - x\right)^2 + 1}\,dx$ for a range of points.

```
In[132]:= rules = {"GaussKronrodRule",
             "LobattoKronrodRule", "TrapezoidalRule", "ClenshawCurtisRule"};
          errors = Outer[Table[{pnts, RuleErrors[#2, #1, 30, pnts]}, {pnts, 4, 100, 1}] &,
             rules, Function /@ (Function[{f}, f /. x -> #] /@ funcs)];
          exactErrors = Map[#[[1]] &, errors, {-2}];
          ruleErrors = Map[#[[2]] &, errors, {-2}];
```

Here are plots of how the logarithms of the errors decrease for each rule and each function.

In[136]:=
```
Row[{Grid[Join[{{"exact errors", "error estimates"}},
    Flatten[Transpose[{{#, SpanFromLeft} & /@ rules, Transpose[Map[
          Function[{d},
           (gr = ListLinePlot[
               Map[{#[[1]], Log[10, #[[2]]]} & /@ # &, d], ImageSize -> {200, 100},
               PlotRange -> {{0, 100}, {0, -9}}, AxesOrigin -> {0, 0}];
xc = 110;
xcSq = 106;
legend = {Text[funcs[[1]], {xc, -1.5}, {-1, 0}],
                Text[funcs[[2]], {xc, -3.5}, {-1, 0}], Text[funcs[[3]],
                 {xc, -5.5}, {-1, 0}], Text[funcs[[4]], {xc, -7.5}, {-1, 0}]};
legendSq = {Text["   ", {xcSq, -1.5}, {-1, 0}], Text["   ", {xcSq, -3.5}, {-1, 0}],
                Text["   ", {xcSq, -5.5}, {-1, 0}],
                Text["   ", {xcSq, -7.5}, {-1, 0}]};
legendSq = MapThread[Append[#1, Background -> #2] &, {legendSq,
                 Cases[gr, Hue[s__], ∞]}];
gr)], {exactErrors, ruleErrors}, {2}]]}], 1]], Dividers -> All],
  Graphics[{legend, legendSq}, ImageSize -> {200, 200}, AspectRatio -> 5]}]
```

Out[136]=



Examples of Pathological Behavior

Tricking the Error Estimator

In this subsection an integral will be discussed which `NIntegrate` underestimates with its default settings since it fails to detect part of the integrand. The part is detected if the precision goal is increased.

The Wrong Estimation

Consider the following function.

```
In[13]:= f[x_] := Sech[10 * (x - 0.2)] ^ 2 + Sech[100 * (x - 0.4)] ^ 4 + Sech[1000 * (x - 0.6)] ^ 6
```

Here is its exact integral over $[0, 1]$.

```
In[138]:= exact = Integrate[f[x], {x, 0, 1}]
Out[138]= 0.210803
```

`NIntegrate` gives the estimate.

```
In[139]:= est = NIntegrate[f[x], {x, 0, 1}]
Out[139]= 0.209736
```

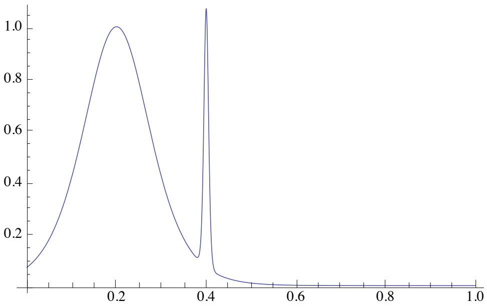
This is too inaccurate when compared to the exact value.

```
In[140]:= Abs[exact - est]
Out[140]= 0.00106667
```

Here is the plot of the function, which is also wrong.

```
In[141]:= Plot[f[x], {x, 0, 1}]
```

Out[141]=



Better Results

Better results can be achieved using the `NIntegrate` option `PrecisionGoal` and increasing the recursion depth.

```
In[17]:= NIntegrate[f[x], {x, 0, 1}, Method → "GlobalAdaptive",
          MaxRecursion → 20, PrecisionGoal → 12]

           NIntegrate::slwcon :
             Numerical integration converging too slowly; suspect one of the following: singularity, value
                  of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

Out[17]= 0.210803
```

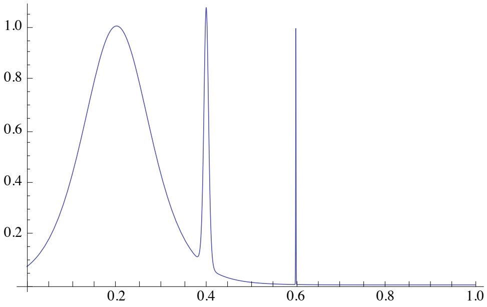
This is a table that finds the precision goal for which no good results are computed.

```
In[18]:= Table[{pg, NIntegrate[f[x], {x, 0, 1}, Method → "GlobalAdaptive",
             MaxRecursion → 20, PrecisionGoal → pg]}, {pg, 6, 12}]

           NIntegrate::slwcon :
             Numerical integration converging too slowly; suspect one of the following: singularity, value
                  of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

           NIntegrate::slwcon :
             Numerical integration converging too slowly; suspect one of the following: singularity, value
                  of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

           NIntegrate::slwcon :
             Numerical integration converging too slowly; suspect one of the following: singularity, value
                  of the integration is 0, highly oscillatory integrand, or WorkingPrecision too small. ≫

           General::stop : Further output of NIntegrate::slwcon will be suppressed during this calculation. ≫

Out[18]= {{6, 0.209736}, {7, 0.209736}, {8, 0.209736},
          {9, 0.210803}, {10, 0.210803}, {11, 0.210803}, {12, 0.210803}}
```

If the plot points are increased, the plot of the function looks different.

```
In[144]:= Plot[f[x], {x, 0, 1}, PlotPoints → 100]
```

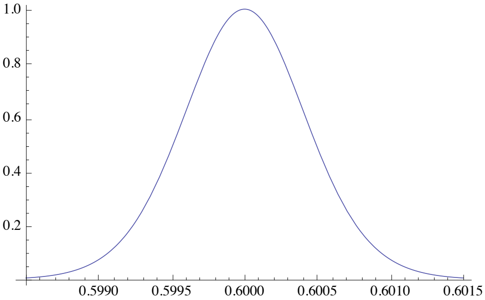
Out[144]=



Here is the zoomed plot of the spike that `Plot` is missing with the default options.

```
In[145]:=  eps = 0.0015; Plot[f[x], {x, 0.6 - eps, 0.6 + eps}]
```

Out[145]=



If this part of the function is integrated, the result fits the quantity that is "lost" (or "missed") by `NIntegrate` with the default option settings.

```
In[146]:=  NIntegrate[f[x], {x, 0.6 - eps, 0.6 + eps}]

Out[146]= 0.00106857
```

```
In[147]:=  Abs[exact - est]

Out[147]= 0.00106667
```

Why the Estimator Is Misled

These are the abscissas and weights of a Gauss-Kronrod rule used by default by `NIntegrate`.

```
In[146]:=  {absc, weights, errweights} =
             NIntegrate`GaussKronrodRuleData[5, MachinePrecision];
```

This defines a function for application of the rule.

```
In[147]:=  IRuleEstimate[f_, {a_, b_}] :=
            Module[{integral, error},
             {integral, error} = (b - a) Total@MapThread[{f[#1] #2, f[#1] #3} &,
                 {Rescale[absc, {0, 1}, {a, b}], weights, errweights}];
             {integral, Abs[error]}
            ]
```

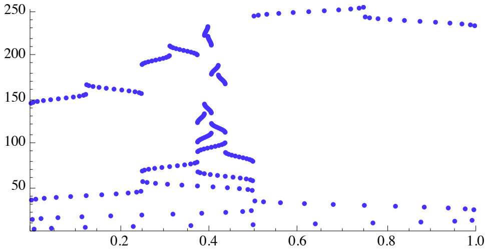
This finds the points at which the adaptive strategy samples the integrand.

```
In[148]:=  cTbl = Reap[NIntegrate[f[x], {x, 0, 1},
                 EvaluationMonitor :→ Sow[x]]][[2]] // Flatten;
```

This is a plot of the sampling points. The vertical axis is for the order at which the points have been used to evaluate the integrand.

```
In[149]:= ListPlot[Transpose[{cTbl, Range[1, Length[cTbl]]}], AspectRatio → 0.5,
           PlotRange → {{0, 1}, {0, Length[cTbl]}}, PlotStyle → {Hue[0.7]}]
```

Out[149]=



It can be seen on the preceding plot that `NIntegrate` does extensive computation around the top of the second spike near $x = 0.4$. `NIntegrate` does not do as much computation around the unintegrated spike near $x = 0.6$.

These are Gauss-Kronrod and Gauss abscissas in the last set of sampling points, which is over the region $[0.5, 0.75]$.

```
In[150]:= gk = Sort[Take[cTbl, -11]]
          g = Take[gk, {2, -2, 2}]

Out[150]= {0.501989, 0.511728, 0.530729, 0.557691, 0.590046,
           0.625, 0.659954, 0.692309, 0.719271, 0.738272, 0.748011}

Out[151]= {0.511728, 0.557691, 0.625, 0.692309, 0.738272}
```

Here the integrand is applied over the abscissas.

```
In[152]:= fgk = f /@ gk;
          fg = f /@ g;
```

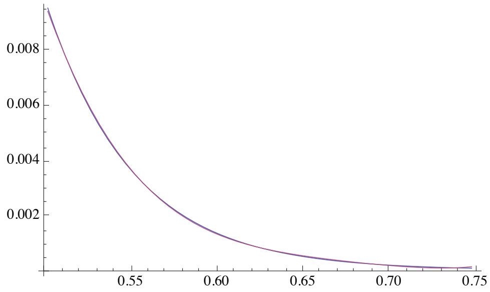
Here is a polynomial approximation of the integrand over the abscissas.

```
In[154]:= gkf[x_] := Evaluate[InterpolatingPolynomial[Transpose[{gk, fgk}], x]]
          gf[x_] := Evaluate[InterpolatingPolynomial[Transpose[{g, fg}], x]]
```

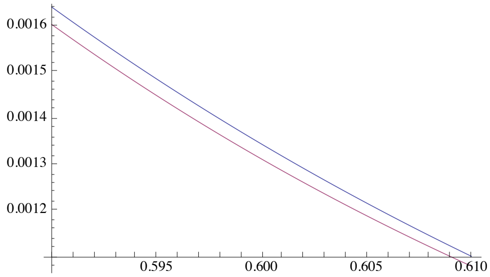
These plots show that the two polynomial approximations almost coincide over $x = 0.6$.

```
In[156]:= Plot[{gkf[x], gf[x]}, {x, Min[gk], Max[gk]}]
          eps = 0.01;
          Plot[{gkf[x], gf[x]}, {x, 0.6 - eps, 0.6 + eps}]
```

Out[156]=



Out[158]=



If the polynomials are integrated over the region where 0.6 is placed, the difference between them, which `NIntegrate` uses as an error estimate, is really small.

```
In[159]:= Integrate[gkf[x], {x, 0.5, 0.75}]
          Integrate[gf[x], {x, 0.5, 0.75}]
          % - %% // FullForm

Out[159]= 0.000491184

Out[160]= 0.000491184

Out[161]//FullForm= -3.6652469947995314`*^-10
```

Since the difference is the error estimate assigned for the region $[0.5, 0.75]$, with the default precision goal `NIntegrate` never picks it up for further integration refinement.

Phase Errors

In this subsection are discussed causes why integration rules might seriously underestimate or overestimate the actual error of their integral estimates. Similar discussion is given in [LynKag76].

This defines a function.

In[162]:=

$$\texttt{f[x\_, }\lambda\texttt{\_, }\mu\texttt{\_] :=} \frac{10^{-\mu}}{(x-\lambda)^2 + 10^{-2\mu}}$$

Consider the numerical and symbolic evaluations of the integral of `f[x, 0.415, 1.25]` over the region $[-1, 1]$.

```
In[163]:= num = NIntegrate[f[x, 0.415, 1.25], {x, -1, 1}, PrecisionGoal -> 2]

Out[163]= 1.72295
```

```
In[164]:= exact = Integrate[f[x, 0.415, 1.25], {x, -1, 1}]

Out[164]= 3.00604 + 0. i
```

They differ significantly. The precision goal requested is 2, but relative error is much higher than 10^{-2}.

```
In[165]:= Abs[num - exact] / Abs[exact]

Out[165]= 0.426837
```

(Note that `NIntegrate` gives correct results for higher-precision goals.)

Below is an explanation why this happens.

Let the integration rule R_2 be embedded in the rule R_1. Accidentally, the error estimate $\left|R_1{}^V[f] - R_2{}^V[f]\right|$ of the integral estimate $R_1{}^V[f]$, where $V = [-1, 1]$, can be too small compared with the exact error $\left|R_2{}^V[f] - \int_V f(x)\,dx\right|$.

To demonstrate this, consider the Gauss-Kronrod rule $\mathrm{GK}[f, 5]$ with 11 sampling points that has an embedded Gauss rule $G[f, 5]$ with 5 sampling points. (This is the rule used in the two integrations above.)

```
In[166]:= {absc, weights, errweights} =
            NIntegrate`GaussKronrodRuleData[5, MachinePrecision];
```

This defines a function that applies the rule.

```
In[167]:= IRuleEstimate[f_, {a_, b_}] :=
           Module[{integral, error},
            {integral, error} = (b - a) Total@MapThread[{f[#1] #2, f[#1] #3} &,
                {Rescale[absc, {0, 1}, {a, b}], weights, errweights}];
            {integral, Abs[error]}
           ]
```

This is the integral $\int_{-1}^{1} f[x, \lambda, \mu]\,dx$ of `f[x, λ, μ]` previously defined.

In[168]:= `exact = -ArcTan[10^μ (-1 + λ)] + ArcTan[10^μ (1 + λ)];`

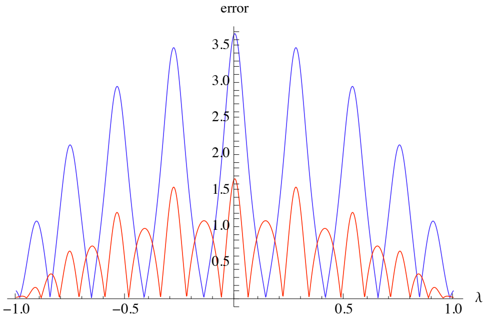
We can plot a graph with the estimated error of $\mathrm{GK}(f, 5)$ and the real error for different values of λ in $[-1, 1]$. That is, you plot $|\mathrm{GK}(f, 5) - G(f, 5)|$ and $\left|\mathrm{GK}(f, 5) - \int_{-1}^{1} f[x, \lambda, \mu]\, dx\right|$.

In[169]:=

```
Block[{λ, μ = 1.15, pnts = 1000, rres, errres, exactres, lambdas},
 (* the plot uses 1000 values λ *)
 lambdas = Table[λ, {λ, -1, 1, 2/(pnts - 1)}];
 (* this computes the integral and error esitmates over the λ's *)
 {rres, errres} = Transpose@
   Map[Function[{λ}, IRuleEstimate[f[#1, λ, Evaluate[μ]] &, {-1, 1}]], lambdas];

 (* this computes the exact integrals over the λ's *)
 exactres  = Map[exact /. λ -> #1 &, lambdas];

 (* this finds the number underestimating error estimates *)
 Print["Percent of underestimation: ",
  100 * Length[Select[errres - Abs[exactres - rres], #1 < 0 &]] / Length[lambdas] // N,
  "%", " "];

 (* the plots, blue is for |GK[f,5]-GK[f,5]|,
 red is for |GK[f,5]-∫_{-1}^{1} f[x,λ,μ]| *)
 ListLinePlot[{Transpose[{lambdas, errres}],
   Transpose[{lambdas, Abs[exactres - rres]}]}, PlotRange -> All,
  PlotStyle -> {{Hue[0.7]}, {Hue[0]}}, AxesLabel -> {λ, "error"}]
]
```

```
Percent of underestimation: 23.8%
```

Out[169]=



In the plot above, the blue graph is for the estimated error, $|\mathrm{GK}(f, 5) - G(f, 5)|$. The graph of the actual error $\left|\mathrm{GK}(f, 5) - \int_{-1}^{1} f[x, \lambda, \mu]\, dx\right|$ is red.

You can see that the value `0.415` of the parameter λ is very close to one of the $|\mathrm{GK}(f, 5) - G(f, 5)|$ local minimals.

A one-dimensional quadrature rule can be seen as the result of the integration of a polynomial that is fitted through the rule's abscissas and the integrand values over them. We can further try to see the actual fitting polynomials for the integration of `f[x, λ, μ]`.

```
In[170]:= Clear[FitPlots];
          FitPlots[f_, {a_, b_}, abscArg_] :=
            Module[{absc = Rescale[abscArg, {0, 1}, {a, b}]},
             (* this finds the interpolating polynomial
              through the Gauss abscissas and the values of f over them *)
             polyGauss[x_] := Evaluate[InterpolatingPolynomial[Transpose[
                  {Take[absc, {2, -2, 2}], f[#1] & /@ (Take[absc, {2, -2, 2}])}], x]];

             (* this finds the interpolating polynomial through the Gauss-
              Kronrod abscissas and the values of f over them *)
             polyGaussKronrod[x_] := Evaluate[InterpolatingPolynomial[
                 Transpose[{absc, f[#1] & /@ absc}], x]];

             (* plot of the Gauss interpolating points *)
             samplPointsGauss = Graphics[{GrayLevel[0], PointSize[0.02], Point /@
                  Transpose[{Take[absc, {2, -2, 2}], f[#1] & /@ Take[absc, {2, -2, 2}]}]}];

             (* plot of the Gauss-Kronrod interpolating points *)
             samplPointsGaussKronrod =
              Graphics[{Red, PointSize[0.012], Point /@ Transpose[{absc, f[#1] & /@ absc}]}];

             (* interpolating polynomials and f plots *)
             Block[{$DisplayFunction = Identity},
              funcPlots = Plot[{polyGauss[x], polyGaussKronrod[x], f[x]}, {x, a, b},
                  PlotRange -> All, PlotStyle -> {{Hue[0.7]}, {Hue[0.8]}, {Hue[0]}}];
             ];

             exact = Integrate[f[x], {x, a, b}];
             r1 = Integrate[polyGauss[x], {x, a, b}];
             r2 = Integrate[polyGaussKronrod[x], {x, a, b}];
             Print["estimated integral:" <> ToString@r2,
              "  exact integral:" <> ToString@Re@exact];
             Print["estimated error:" <> ToString@Abs[r1 - r2],
              "  actual error:" <> ToString@Abs[r2 - exact]];
             Show[{funcPlots, samplPointsGauss, samplPointsGaussKronrod}]
            ];
```

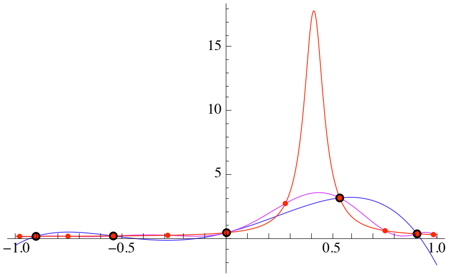
In the plots below the function `f[x, λ, μ]` is plotted in red, the Gauss polynomial is plotted in blue, the Gauss-Kronrod polynomial is plotted in violet, the Gauss sampling points are in black, and the Gauss-Kronrod sampling points are in red.

You can see that since the peak of `f[x, 0.415, 1.25]` falls approximately halfway between two abscissas, its approximation is an underestimate.

```
In[172]:= FitPlots[f[#1, 0.415, 1.25] &, {-1, 1}, absc]
```

```
estimated integral:1.72295  exact integral:3.00604

estimated error:0.0133177  actual error:1.28309
```

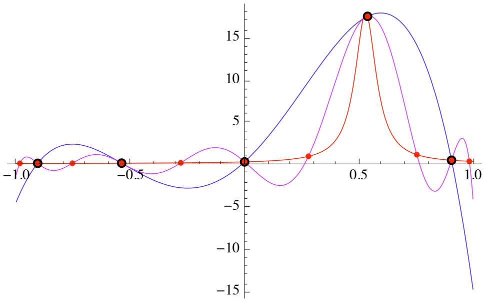
Out[172]=



Conversely, you can see that since the peak of `f[x, 0.53, 1.25]` falls approximately on one of the abscissas, its approximation is an overestimate.

```
In[173]:= FitPlots[f[#1, 0.53, 1.25] &, {-1, 1}, absc]

            estimated integral:4.77891  exact integral:2.98577

            estimated error:3.77834  actual error:1.79313
```

Out[173]=



Index of Technical Terms

Abscissas

Degree of a one-dimensional integration rule

Degree of a multidimensional integration rule

Exact rule

Embedded rule

Null rule

Product rule

Progressive rule

Sampling points

NIntegrate References

[BerntEspGenz91] Berntsen, J., T. O. Espelid, and A. Genz. "An Adaptive Algorithm for the Approximate Calculation of Multiple Integrals." *ACM Trans. Math. Softw*. 17, no. 4 (1991): 437-451. http://citeseer.ist.psu.edu/berntsen91adaptive.html

[BrezRedZag91] Brezinski, C. and M. Redivo Zaglia. *Extrapolation Methods*. North-Holland, 1991.

[CohRodVil99] Cohen, H., F. Rodriguez Villegas, and D. Zagier. "Convergence Acceleration of Alternating Series." *Experimental Mathematics* 9, no. 1 (2000): 3-12. http://www.expmath.org/restricted/9/9.1/cohen.ps

[DavRab65IS] Davis, P. J. and P. Rabinowitz. "Ignoring the Singularity in Approximate Integration." *J. SIAM: Series B, Numerical Analysis* 2, no. 3 (1965): 367-383.

[DavRab84] Davis, P. J. and P. Rabinowitz. *Methods of Numerical Integration*, 2nd ed. Academic Press, 1984.

[DeBruijn58] De Bruijn, N. G. *Asymptotic Methods in Analysis*. North-Holland, 1958.

[Duffy82] Duffy, M. G. "Quadrature over a Pyramid or Cube of Integrands with a Singularity at a Vertex." *J. SIAM Numer. Anal*. 19, no. 6 (1982).

[Ehrich2000] Ehrich, S. "Stopping Functionals for Gaussian Quadrature Formulas." *J. Comput. Appl. Math. Special issue on Numerical Analysis 2000, Vol. V: Quadrature and Orthogonal Polynomials* 127, no. 1-2 (2001): 153-171. http://citeseer.ist.psu.edu/ehrich00stopping.html

[Evans93] Evans, G. *Practical Numerical Integration*. Wiley, 1993.

[GenzMalik80] Genz, A. C. and A. A. Malik. "An Adaptive Algorithm for Numerical Integration over an N-dimensional Rectangular Region." *J. Comp. Appl. Math*. 6, no. 4 (1980): 295-302.

[GenzMalik83] Genz, A. C. and A. A. Malik. "An Imbedded Family of Fully Symmetric Numerical Integration Rules." *J. SIAM Numer. Anal.* 20, no. 3 (1983): 580-588.

[HammHand64] Hammersley, J. M. and D. C. Handscomb. *Monte Carlo Methods*. Chapman and Hall, 1964.

[IriMorTak70] Iri, M., S. Moriguti, and Y. Takasawa. "On a Certain Quadrature Formula." *Kokyuroku of the Res. Inst. for Math. Sci. Kyoto Univ*. 91 (1970): 82-118 (in Japanese). English translation in *J. Comp. Appl. Math.* 17, no. 1-2 (1987): 3-20.

[KrUeb98] Krommer, A. R. and C. W. Ueberhuber. *Computational Integration*. SIAM Publications, 1998.

[LynKag76] Lyness, J. N. and J. J. Kaganove. "Comments on the Nature of Automatic Quadrature Routines." *ACM Trans. Math. Software* 2, no. 1 (1976): 65-81.

[MalcSimp75] Malcolm, M. A. and R. B. Simpson. "Local versus Global Strategies for Adaptive Quadrature." *ACM Transactions on Mathematical Software* 1, no. 2 (1975): 129-146.

[Mori74] Mori, M. "On the Superiority of the Trapezoidal Rule for the Integration of Periodic Analytic Functions." *Memoirs of Numerical Mathematics* 1 (1974): 11-19.

[MoriOoura93] Ooura, T. and M. Mori. "Double Exponential Formulas for Fourier Type Integrals with a Divergent Integrand." In *Contributions in Numerical Mathematics*, *World Scientific Series in Applicable Analysis*, Vol. 2 301-308, 1993.

[MurIri82] Murota, K. and M. Iri. "Parameter Tuning and Repeated Application of the IMT-Type Transformation in Numerical Quadrature." *Numerische Mathematik* 38, no. 3 (1982): 347-363.

[OouraMori91] Ooura, T. and M. Mori. "A Double Exponential Formula for Oscillatory Functions over the Half Infinite Interval." *J. Comput. Appl. Math.* 38, no. 1-3 (1991): 353-360.

[OouraMori99] Ooura, T. and M. Mori. "A Robust Double Exponential Formula for Fourier Type Integrals." *J. Comput. Appl. Math.* 112, no. 1-2 (1999): 229-241.

[OHaraSmith68] O'Hara, H. and F. J. Smith. "Error Estimation in the Clenshaw-Curtis Quadrature Formula." *Comput. J.* 11 (1968): 213-219.

[PiesBrand74] Piessens, R. and M. Branders. "A Note on the Optimal Addition of Abscissas to Quadrature Formulas of Gauss and Lobatto Type." *Math. of Comput.* 28, no. 125 (1974): 135-139.

[PiesBrand75] Piessens, R. and M. Branders. "Algorithm 002. Computation of Oscillating Integrals." *J. Comput. Appl. Math.* 1 (1975): 153-164.

[PiesBrand84] Piessens, R. and M. Branders. "Computation of Fourier Transform Integrals Using Chebyshev Series Expansions." *Computing* 32, no. 2 (1984): 177-186.

[PrFlTeuk92] Press, W. H., B. P. Flannery, and S. A. Teukolsky. *Numerical Recipes in C*. Cambridge University Press, 1992.

[Rice75] Rice, J. R. "A Metalgorithm for Adaptive Quadrature." *J. Assoc. Comput. Mach*. 22, no. 1 (1975): 61-82.

[SkKeip93] Skeel, R. D. and J. B. Keiper. *Elementary Numerical Computing with Mathematica.* McGraw-Hill, Inc. (1993)

[SloanJoe94] Sloan, I. H. and S. Joe. *Lattice Methods for Multiple Integration*. Oxford University Press, 1994.

[Stroud71] Stroud, A. H. *Approximate Calculation of Multiple Integrals.* Prentice-Hall, 1971.

[Weideman2002] Weideman, J. A. C. "Numerical Integration of Periodic Functions: A Few Examples." *Amer. Math. Monthly* 109, no. 1 (2002): 21-36.