Title

stata.com

Quadrature( ) — Numerical integration

Description

The `Quadrature()` class approximates the integral $\int_a^b f(x)\,dx$ by adaptive quadrature, where $f(x)$ is a real-valued function, and a and b are lower and upper limits.

Syntax

With the `Quadrature()` class, you can approximate an integral in as few as four steps—create an instance of the class with `Quadrature()`, specify the evaluator function with `setEvaluator()`, set the limits with `setLimits()`, and perform the computation with `integrate()`. Example 1 demonstrates this basic procedure.

The full syntax allows you to further define the integration problem and to obtain additional results. Syntax is presented under the following headings:

Step 1: Initialization

q = `Quadrature()`

Step 2: Definition of integration problem

void	*q*.setEvaluator(*pointer(real function) scalar fcn*)
void	*q*.setLimits(*real rowvector limits*)
void	*q*.setTechnique(*string scalar technique*)
void	*q*.setMaxiter(*real scalar maxiter*)
void	*q*.setAbstol(*real scalar abstol*)
void	*q*.setReltol(*real scalar reltol*)
void	*q*.setArgument(*real scalar i*, *arg*)
void	*q*.setTrace(*string scalar trace*)
pointer(real function) scalar	*q*.getEvaluator()
real rowvector	*q*.getLimits()
string scalar	*q*.getTechnique()
real scalar	*q*.getMaxiter()
real scalar	*q*.getAbstol()
real scalar	*q*.getReltol()
pointer scalar	*q*.getArgument(*real scalar i*)
real scalar	*q*.getNarguments()
string scalar	*q*.getTrace()

Step 3: Perform integration

real scalar	*q*.integrate()

Step 4: Display or obtain results

real scalar	*q*.value()
real scalar	*q*.iterations()
real scalar	*q*.converged()
real scalar	*q*.errorcode()
string scalar	*q*.errortext()
real scalar	*q*.returncode()

Utility function for use in all steps

void *q*.query()

Definition of q

A variable of type Quadrature is called an instance of the Quadrature() class. *q* is an instance of Quadrature(), a vector of instances, or a matrix of instances. If you are working interactively, you can create an instance of Quadrature() by typing

```
q = Quadrature()
```

For a row vector of *n* Quadrature() instances, type

```
q = Quadrature(n)
```

For an $m \times n$ matrix of Quadrature() instances, type

```
q = Quadrature(m, n)
```

In a function, you would declare one instance of the Quadrature() class *q* as a scalar.

```
void myfunc()
{
    class Quadrature scalar   q
    q = Quadrature()
    ...
}
```

Within a function, you can declare *q* as a row vector of *n* instances by typing

```
void myfunc()
{
    class Quadrature rowvector   q
    q = Quadrature(n)
    ...
}
```

For an $m \times n$ matrix of instances, type

```
void myfunc()
{
    class Quadrature matrix   q
    q = Quadrature(m, n)
    ...
}
```

Functions defining the integration problem

At a minimum, you need to tell the Quadrature() class about the function you wish to integrate and the limits of integration. You can also specify the technique used to compute the quadrature, the maximum number of iterations allowed, the convergence criteria, the arguments to be passed to the evaluator, and whether or not to print computation details.

Each pair of functions includes a *q*.set function that specifies an integration setting and a *q*.get function that returns the current setting.

q.setEvaluator( ) and q.getEvaluator( )

q.setEvaluator(*fcn*) sets a pointer to the evaluator function, which has to be called before any calculation. Additionally, the evaluator has a special format. Namely, it has at least one *real scalar* argument (first argument, corresponding to x) and returns a *real scalar* value, $f(x)$.

q.getEvaluator() returns a pointer to the evaluator (*NULL* if not specified).

q.setLimits( ) and q.getLimits( )

q.setLimits(*limits*) sets the integration limits as a two-dimensional rowvector. The limits can be finite or infinite. The lower limit must be less than or equal to the upper limit. Using a missing value as the lower limit indicates $-\infty$, and using a missing value as the upper limit indicates ∞.

q.getLimits() returns the integration limits (empty vector if not specified).

q.setTechnique( ) and q.getTechnique( )

q.setTechnique(*technique*) specifies the technique used to compute the quadrature.

technique specified in setTechnique() can be

technique	Description
"gauss"	adaptive Gauss–Kronrod method; the default
"simpson"	adaptive Simpson method

q.getTechnique() returns the current technique.

q.setMaxiter( ) and q.getMaxiter( )

q.setMaxiter(*maxiter*) specifies the maximum number of iterations, which must be an integer greater than 0. The default value of *maxiter* is 16000.

q.getMaxiter() returns the current maximum number of iterations.

q.setAbstol( ), q.getAbstol( ), q.setReltol( ), and q.getReltol( )

q.setAbstol(*abstol*) and *q*.setReltol(*reltol*) specify the convergence criteria with absolute and relative tolerances, which must be greater than 0. The default values of *abstol* and *reltol* are 1e–10 and 1e–8, respectively.

The absolute tolerance gives an upper bound for the approximate measure of the absolute difference between the computed solution and the exact solution, while the relative tolerance gives an upper bound for the approximate measure of the relative difference between the computed and the exact solution.

q.getAbstol() and *q*.getReltol() return the current absolute and relative tolerances, respectively.

q.setArgument( ), q.getArgument( ), and q.getNarguments( )

q.setArgument(*i*, *arg*) specifies *arg* as the *i*th extra argument of the evaluator, where *i* is an integer between 1 and 9. Here *arg* can be anything. If *i* is greater than the current number of extra arguments, then the number of extra arguments will be increased to *i*.

q.getArgument(*i*) returns a pointer to the *i*th extra argument of the evaluator (*NULL* if not specified).

q.getNarguments() returns the current number of extra arguments.

q.setTrace( ) and q.getTrace( )

q.setTrace() sets whether or not to print out computation details. *trace* specified in setTrace() can be "on" or "off". The default value is "off".

q.getTrace() returns the current trace status.

Performing integration

q.integrate( )

q.integrate() computes the numerical integration, that is, the approximation of the integral of the evaluator from the lower limit to the upper limit. *q*.integrate() returns the computed quadrature value.

Functions for obtaining results

After performing integration, the functions below provide results including value of the integral, number of iterations, whether convergence was achieved, error messages, and return codes.

q.value( )

q.value() returns the computed quadrature value; it returns a missing value if not yet computed.

q.iterations( )

q.iterations() returns the number of iterations; it returns 0 if not yet computed.

q.converged( )

q.converged() returns 1 if converged and 0 if not.

q.errorcode( ), q.errortext( ), and q.returncode( )

q.errorcode() returns the error code generated during the computation; it returns 0 if no error is found.

q.errortext() returns an error message corresponding to the error code generated during the computation; it returns an empty string if no error is found.

q.returncode() returns the Stata return code corresponding to the error code generated during the computation.

The error codes and the corresponding Stata return codes are as follows:

Error code	Return code	Error text
1	111	must specify an evaluator function to compute numerical integration using `setEvaluator()`
2	111	must specify lower and upper integration limits as a rowvector with 2 columns using `setLimits()`
3	111	You specified extra argument *n* but did not specify extra argument *i*. When you specify extra argument *n*, you must also specify all extra arguments less than *n*.
4	111	code distributed with Stata has been changed so that a required subroutine cannot be found
5	416	evaluator function returned a missing value at one or more quadrature points
6	430	subintervals cannot be further divided to achieve the required accuracy
7	430	maximum number of iterations has been reached

Here *n* will be replaced by an actual number in the message, and *i* will be replaced by an actual number less than *n* in the message.

Utility function

At any stage of the integration problem, you can obtain a report of all settings and results currently stored in a class `Quadrature()` instance.

q.query( )

q`.query()` with no return value displays information stored in the class.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Introduction

The `Quadrature()` class is a Mata class for numerical integration.

`Quadrature()` uses adaptive quadrature to approximate the integral $\int_a^b f(x)\,dx$, where $f(x)$ is a real-valued function and a and b are lower and upper limits. `Quadrature()` approximates integrals for functions defined using an evaluator program.

For an introduction to class programming in Mata, see [M-2] **class**.

Examples

To approximate an integral, you first use `Quadrature()` to get an instance of the class. At a minimum, you must also use `setEvaluator()` to specify the evaluator function, `setLimits()` to specify the limits, and `integrate()` to perform the computations. In the examples below, we demonstrate both basic and more-advanced use of the `Quadrature()` class.

A basic example

▷ Example 1: Approximate an integral

We want to approximate $\int_0^\pi \sin(x)\,dx$ using `Quadrature()`. We first define an evaluator function `f()` as a wrapper for the built-in `sin()` function:

```
: real scalar f(real scalar x) {
>         return(sin(x))
> }
```

We need this wrapper because we must put the address of the evaluator function into an instance of `Quadrature()` and we are not able to get the address of a built-in function.

Having defined the evaluator function, we follow the four steps that are required each time we use the `Quadrature()` class. First, we create an instance `q` of the `Quadrature()` class:

```
: q = Quadrature()
```

Second, we use `setEvaluator()` to put a pointer to the evaluator function `f()` into `q`.

```
: q.setEvaluator(&f())
```

Third, we use `setLimits()` to specify the lower and upper limits.

```
: q.setLimits((0, pi()))
```

Fourth, we use `integrate()` to compute the approximation.

```
: q.integrate()
  2
```

We find that $\int_0^\pi \sin(x)\,dx = 2$. ◁

Integrals with infinite limits

We often need to evaluate integrals where one or both limits are infinite.

▷ Example 2: Approximate an integral from a finite point to ∞

To calculate the numerical integral of $\int_0^\infty \exp(-x)\,dx$, we use the `q` instance of `Quadrature()` from the previous example, and we initialize a new evaluator function `f1()` and new limits:

```
: real scalar f1(real scalar x) {
>         return(exp(-x))
> }
: q.setEvaluator(&f1())
: q.setLimits((0, .))
```

We specify `.` to set the upper limit to ∞.

Now we can compute the approximation:

```
: q.integrate()
  .9999999997
```

◁

▷ Example 3: Approximate an integral from $-\infty$ to ∞

To calculate the numerical integral of $\int_{-\infty}^{\infty} \exp(-x^2)\,dx$, we continue with `q` from the previous example, and we initialize a new evaluator function `f2()` and new limits:

```
: real scalar f2(real scalar x) {
>         return(exp(-x*x))
> }
: q.setEvaluator(&f2())
: q.setLimits((., .))
```

We specify `.` for both the upper and the lower limits. This sets the lower limit to $-\infty$ and the upper limit to ∞.

Now we can compute the approximation:

```
: q.integrate()
  1.772453852
```

◁

Passing arguments to the evaluator function

Often, statistical and mathematical functions that we wish to integrate need additional arguments. For instance, we may want to integrate with respect to one variable while setting other variables in the function to specific values. We can do this by passing arguments to the function evaluator.

▷ Example 4: Add an extra argument to the evaluator function

We calculate the numerical integral of $\int_0^1 (x^2 + z^2)\,dx$, where $z = 5$. We continue with the `q` instance of `Quadrature()` from the previous example, and we initialize a new evaluator function and limits:

```
: real scalar f3(real scalar x, real scalar z) {
>         return(x*x + z*z)
> }
: q.setEvaluator(&f3())
: q.setLimits((0, 1))
```

We use `setArgument()` to put the value of the extra argument `z` into `q`. Typing

```
: q.setArgument(1, 5)
```

specifies that the first extra argument has a value of 5. There is only one extra argument, so we are now ready to compute the approximation.

```
: q.integrate()
  25.33333333
```

◁

Singular points and setting tolerances

Problematic points of a function are known as "singular points". There are two common types of singular points. In the first type, x_0 is a singular point because $f(x_0)$ is ∞ or $-\infty$. In the second type, $f(x_0)$ is not continuous at x_0.

The methods implemented in the `Quadrature()` class are robust to singular points. Sometimes a function with singular points might be better approximated by specifying a smaller convergence tolerance, as in the example below.

▷ Example 5: Approximate an integral with a singular point

To calculate the numerical integral of $\int_0^1 \log(x)\,dx$, we define `q2` as a new instance of the `Quadrature()` class, and we initialize the new evaluator function and limits:

```
: q2 = Quadrature()
: real scalar f4(real scalar x) {
>         return(log(x))
> }
: q2.setEvaluator(&f4())
: q2.setLimits((0,1))
```

The `log()` has a singular point at `0`. We can compute the approximation by using the default absolute tolerance of 1e–10 and default relative tolerance of 1e–8.

```
: q2.integrate()
  -1.000000004
```

This is close to the value of -1 that we expect. However, we can obtain a more accurate result if we set stricter absolute and relative tolerances.

```
: q2.setAbstol(1e-15)
: q2.setReltol(1e-12)
: q2.integrate()
  -1
```

The absolute tolerance is an upper bound of the estimated absolute difference between the computed approximation and the exact solution. The relative tolerance is an upper bound of the estimated relative difference between the computed approximation and the exact solution. With these smaller tolerances, we now obtain a value of -1.

◁

Displaying settings and results at each stage

Each instance of the class contains lots of information about the problem at hand. You can use the member function `query()` to display this information. Several other member functions display specific pieces of information. The example below illustrates how to use these functions.

▷ Example 6: Display information about the integration problem

To calculate the numerical integral of $\int_{-1}^1 |x|\,dx$, we define the class instance `q3`.

```
: q3 = Quadrature()
```

We can show all the default values by using `query()` before we perform any initialization or computation.

```
: q3.query()
Settings for Quadrature ─────────────────────────────────────────────────────
Version:                                    1.00

Evaluator
    Function:                               unknown

User-defined arguments:                     <none>

Integration limits
    Limits:                                 unknown

Quadrature technique:                       gauss

Trace:                                      off

Convergence
    Maximum iterations:                     16000
    Absolute tolerance:                     1.0000e-10
    Relative tolerance:                     1.0000e-08

Current status
    Quadrature value:                       .
    Converged:                              no

Note: The evaluator function has not been specified.
Note: Integration limits have not been specified.
```

Now we initialize the evaluator function and limits.

```
: real scalar f5(real scalar x) {
>         return(abs(x))
> }

: q3.setEvaluator(&f5())

: q3.setLimits((-1, 1))
```

We then compute the approximation.

```
: q3.integrate()
  1
```

We can set the trace to "on" to see the computation details.

```
: q3.setTrace("on")

: q3.integrate()

Quadrature trace:
    Iteration      Current value            Current error estimate
    0               1.017294655              2.57082e-02
    1                         1              4.18554e-14
  1
```

We turn off the trace by typing

```
: q3.setTrace("off")
```

For illustrative purposes, we switch the technique to the adaptive Simpson's method.

```
: q3.setTechnique("simpson")
```

The default method of the adaptive Gauss–Kronrod quadrature (`gauss`) is almost always better than Simpson's method. Simpson's method is a benchmark method included for completeness.

Our default maximum number of iterations is 16,000, and Stata will print out a warning message when the maximum number of iterations is reached. For illustrative purposes, we set the maximum to 2 and compute the approximation.

```
: q3.setMaxiter(2)
: q3.integrate()
Warning: Convergence not achieved; maximum number of iterations has been
         reached.
  .
```

The integration could not be performed using Simpson's method and allowing only 2 iterations, so we see a warning message and receive a missing value (`.`) as the value of the integral.

We switch back to a maximum of 16,000 iterations and compute the approximation.

```
: q3.setMaxiter(16000)
: q3.integrate()
  1
```

Now we can check the result after computation.

```
: q3.value()
  1
```

We can also check the number of iterations.

```
: q3.iterations()
  46
```

No error was found, so the error code and error message are

```
: q3.errorcode()
  0
: q3.errortext()
```

We can show all the values by using `query()` after the computation:

```
: q3.query()
Settings for Quadrature ------------------------------------------------------
Version:                                1.00
Evaluator
    Function:                           f5()
User-defined arguments:                 <none>
Integration limits
    Limits
        1:                              -1
        2:                              1
Quadrature technique:                   simpson
Trace:                                  off
Convergence
    Maximum iterations:                 16000
    Absolute tolerance:                 1.0000e-10
    Relative tolerance:                 1.0000e-08
Current status
    Quadrature value:                   1
    Converged:                          yes
    Iterations:                         46
```

◁

Vectors and matrices of integrals

If you have more than one integral to approximate, you can create a vector or matrix of instances of the `Quadrature()` class. Sometimes, for the convenience of organizing variables, you will find it more convenient to create a vector or matrix of instances than to create separate instances.

▷ Example 7: A vector of integrals

To demonstrate, we use the `Quadrature()` class to approximate the three integrals $\int_0^1 \sqrt{x}\,dx$, $\int_1^2 \exp(x)\,dx$, and $\int_0^{0.5} (1/\sqrt{x})\,dx$. We begin by defining the three evaluator functions.

```
: real scalar f6(real scalar x) {
>         return(sqrt(x))
> }
: real scalar f7(real scalar x) {
>         return(exp(x))
> }
: real scalar f8(real scalar x) {
>         return(1/sqrt(x))
> }
```

Now we create `qv`, a vector of instances of the `Quadrature()` class.

```
: qv = Quadrature(3)
```

Next we set the evaluator function and the limits for each element in the vector of instances.

```
: qv[1].setEvaluator(&f6())
: qv[1].setLimits((0, 1))
: qv[2].setEvaluator(&f7())
: qv[2].setLimits((1, 2))
: qv[3].setEvaluator(&f8())
: qv[3].setLimits((0, 0.5))
```

We use `integrate()` to compute each of the approximations.

```
: qv[1].integrate()
  .6666666667
: qv[2].integrate()
  4.67077427
: qv[3].integrate()
  1.414213562
```

◁

Conformability

`Quadrature()`:

input:
void

output:
result: 1×1

`Quadrature(`*n*`)`:

input:
n : 1×1

output:
result: $1 \times n$

`Quadrature(`*m*`, `*n*`)`:

input:
m : 1×1
n : 1×1

output:
result: $m \times n$

`setEvaluator(`*fcn*`)`:

input:
fcn: 1×1

output:
result: *void*

`getEvaluator()`:

input:
void

output:
result: 1×1

`setLimits(`*limits*`)`:

input:
limits: 1×2

output:
result: *void*

`getLimits()`:

input:
void

output:
result: 1×2

`setTechnique(`*technique*`)`:

input:

technique: 1×1

output:

result: *void*

`getTechnique()`:

input:

void

output:

result: 1×1

`setMaxiter(`*maxiter*`)`:

input:

maxiter: 1×1

output:

result: *void*

`getMaxiter()`:

input:

void

output:

result: 1×1

`setAbstol(`*abstol*`)`:

input:

abstol: 1×1

output:

result: *void*

`getAbstol()`:

input:

void

output:

result: 1×1

`setReltol(`*reltol*`)`:

input:

reltol: 1×1

output:

result: *void*

`getReltol()`:

input:

void

output:

result: 1×1

`setArgument(`*i*`,` *arg*`)`:

input:

	i:	1×1
	arg:	*anything*

output:

	result:	*void*

`getArgument(`*i*`)`:

input:

	i:	1×1

output:

	result:	1×1

`getNarguments()`:

input:

		void

output:

	result:	1×1

`setTrace(`*trace*`)`:

input:

	trace:	1×1

output:

	result:	*void*

`getTrace()`:

input:

		void

output:

	result:	1×1

`integrate()`:

input:

		void

output:

	result:	1×1

`value()`:

input:

		void

output:

	result:	1×1

`iterations()`:

input:

		void

output:

	result:	1×1

`converged()`:
input:
void
output:
result: 1×1

`errorcode()`:
input:
void
output:
result: 1×1

`errortext()`:
input:
void
output:
result: 1×1

`returncode()`:
input:
void
output:
result: 1×1

`query()`:
input:
void
output:
void

Diagnostics

When used incorrectly, the following functions abort with error: `Quadrature()`, `set*()`, `get*()`, `value()`, `iterations()`, `converged`, `errorcode()`, `errortext()`, `returncode()`, and `query()`.

`integrate()` also aborts with error if it is used incorrectly. If `integrate()` runs into numerical difficulties, it returns a missing value and displays a warning message that includes some details about the problem encountered.

References

Davis, P. J., and P. Rabinowitz. 1984. *Methods of Numerical Integration*. 2nd ed. San Diego, CA: Academic Press.

Gander, W., and W. Gautschi. 2000. Adaptive quadrature—revisited. *BIT Numerical Mathematics* 40: 84–101. https://doi.org/10.1023/A:1022318402393.

Gonnet, P. 2009. Adaptive quadrature re-revisited. PhD thesis, ETH No. 18347. http://www.academia.edu/1976055/Adaptive_quadrature_re-revisited.

Lyness, J. N., and J. J. Kaganove. 1977. A technique for comparing automatic quadrature routines. *Computer Journal* 20: 170–177. https://doi.org/10.1093/comjnl/20.2.170.

Piessens, R., E. de Doncker-Kapenga, C. W. Überhuber, and D. K. Kahaner. 1980. *QUADPACK: A Subroutine Package for Automatic Integration*. Berlin: Springer.

Also see

[M-2] **class** — Object-oriented programming (classes)

[M-5] **deriv( )** — Numerical derivatives

[M-5] **moptimize( )** — Model optimization

[M-5] **optimize( )** — Function optimization