## Title

stata.com

**Quadrature( ) —** Numerical integration

## Description

`Quadrature()` 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, respectively.

`QuadratureVec()` is functionally the same as `Quadrature()`, except that it handles a vector of integration problems more conveniently.

## Syntax

Mata's quadrature functions allow you to approximate integrals in as few as four steps—create an instance of the problem with `Quadrature()` or `QuadratureVec()`, specify the evaluator functions with `setEvaluator()`, set the limits with `setLimits()`, and perform the computation with `integrate()`.

`Quadrature()` is better suited for one integration problem, whereas `QuadratureVec()` solves a vector of integration problems more conveniently. Example 1 demonstrates how `Quadrature()` approximates one integral. Example 2 demonstrates how `QuadratureVec()` approximates a set of integrals.

The full syntax allows you to further define the integration problem and to obtain additional results. `Quadrature()` and `QuadratureVec()` have similar syntax, except for the following difference: the arguments and returned values of `Quadrature()` functions are typically scalars corresponding to one integration problem, whereas the arguments and returned values of `QuadratureVec()` functions are a column vector corresponding to a set of problems—that is, `QuadratureVec()` handles a column vector of functions, limits, integration techniques, maximum iterations, tolerances, etc. The $i$th element of the column vector corresponds to the $i$th integration problem.

Syntax is presented under the following headings (applicable to both `Quadrature()` and `QuadratureVec()`, unless specified otherwise):

## Step 1: Problem initialization

```
q = Quadrature()
```

or

```
q = QuadratureVec()
```

## Step 2: Problem definition

### Quadrature(): Definition of one 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()` |

### QuadratureVec(): Definition of a vector of integration problems

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

## Step 3: Perform integration

### Quadrature()

| | |
|---|---|
| *real scalar* | *q*.integrate() |

### QuadratureVec()

| | |
|---|---|
| *real colvector* | *q*.integrate() |

With Quadrature(), a scalar result is returned; with QuadratureVec(), a column vector of results is returned.

## Step 4: Display or obtain results

### Quadrature()

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

### QuadratureVec()

| | |
|---|---|
| *real colvector* | *q*.value() |
| *real colvector* | *q*.iterations() |
| *real colvector* | *q*.converged() |
| *real colvector* | *q*.errorcode() |
| *string colvector* | *q*.errortext() |
| *real colvector* | *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 in Mata.

You can create an instance q of Quadrature() by typing

```
q = Quadrature()
```

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

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

You can also create a vector or matrix of Quadrature() instances. 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)
```

However, to solve several instances of integration problems, you should use `QuadratureVec()`.

A variable of type `QuadratureVec` is an instance of the `QuadratureVec()` class in Mata. You can create an instance `q` of `QuadratureVec()` by typing

```
        q = QuadratureVec()
```

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

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

## Functions defining the integration problem

At a minimum, you need to tell `Quadrature()` or `QuadratureVec()` about the functions you wish to integrate and the limits of integration. You can also specify the techniques used to compute the quadrature, the maximum number of iterations allowed, the convergence criteria, the arguments to be passed to the evaluators, and how to print computation details.

### q.setEvaluator( ) and q.getEvaluator( )

With `Quadrature()`, *q*`.setEvaluator(`*fcn*`)` sets a pointer to the evaluator function *fcn*. `setEvaluator()` has to be called before any calculation. An evaluator has a special format—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 function (*NULL* if not specified).

With `QuadratureVec()`, setting evaluation functions is done the same way, except that *q*`.setEvaluator()` sets a column vector of pointers to evaluator functions and *q*`.getEvaluator()` returns a column vector of pointers to the functions. If *fcn* is a pointer scalar, then the pointer to the evaluator specified is used for all integration problems.

### q.setLimits( ) and q.getLimits( )

With `Quadrature()`, *q*`.setLimits(`*limits*`)` sets the integration limits as a two-dimensional row vector that contains the lower and upper limit in that order. 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 $\infty$.

With `QuadratureVec()`, *q*`.setLimits(`*limits*`)` sets different limits for different integration problems, where *limits* is an $n \times 2$ matrix; that is, one row for each integration problem, where the first column contains lower limits and the second column contains upper limits. If *limits* is a two-dimensional row vector, then the pair of limits specified is used for all integration problems.

*q*`.getLimits()` returns the integration limits; if not specified, this will be an empty vector for `Quadrature()` and an empty matrix for `QuadratureVec()`.

## q.setTechnique( ) and q.getTechnique( )

With `Quadrature()`, *q*`.setTechnique(`*technique*`)` specifies the technique used to compute the quadrature. *technique* can be the following:

| *technique* | Description |
|---|---|
| `"gauss"` | adaptive Gauss–Kronrod method; the default |
| `"simpson"` | adaptive Simpson method |

With `QuadratureVec()`, *q*`.setTechnique(`*technique*`)` specifies different techniques for different integration problems, where *technique* is a column vector of the techniques described above. If *technique* is a scalar, then the technique specified is used for all integration problems.

*q*`.getTechnique()` returns the current technique.

## q.setMaxiter( ) and q.getMaxiter( )

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

With `QuadratureVec()`, *q*`.setMaxiter(`*maxiter*`)` specifies different maximum number of iterations for different integration problems, where *maxiter* is a column vector of maximums. If *maxiter* is a scalar, then the maximum specified is used for all integration problems.

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

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

With `Quadrature()`, *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 absolute and relative tolerances 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.

With `QuadratureVec()`, *q*`.setAbstol(`*abstol*`)` and *q*`.setReltol(`*reltol*`)` specify different tolerances for different integration problems, where *abstol* and *reltol* are column vectors of tolerances. If *abstol* or *reltol* is a scalar, then the tolerance specified is used for all integration problems.

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

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

With `Quadrature()`, the *i*th extra argument *arg* of the evaluator can be specified with *q*`.setArgument(`*i*`,` *arg*`)`, where *i* is an integer between 1 and 9. Here *arg* can be anything: a scalar, an expression, a matrix, or even a pointer to a function. 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).

Example 5 demonstrates how extra arguments are specified with `Quadrature()`.

With `QuadratureVec()`, extra arguments can be specified in the same way as with `Quadrature()`, except that the first argument is a column vector of problem identifiers (integers) for which an extra argument is set. That is, *q*`.setArgument(`*pid*`,` *i*`,` *arg*`)` will set the *i*th extra argument to *arg* for integration problems in column vector *pid*.

*q*`.getArgument(`*pid*`,` *i*`)` returns a pointer to the *i*th extra argument of the evaluator for problems *pid*.

Example 6 demonstrates how extra arguments are specified with `QuadratureVec()`.

With `Quadrature()`, the current number of extra arguments can be obtained with *q*`.getNarguments()`. *q*`.getNarguments()` is not defined for `QuadratureVec()`.

With `QuadratureVec()`, the number of integration problems can be obtained with *q*`.getDimension()`. *q*`.getDimension()` is not defined for `Quadrature()`.

### q.setTrace( ) and q.getTrace( )

With `Quadrature()`, *q*`.setTrace(`*trace*`)` sets whether to print out computation details. *trace* can be `"on"` or `"off"`. The default value is `"off"`.

*q*`.getTrace()` returns the current trace status.

With `QuadratureVec()`, `setTrace(`*trace*`)` and `getTrace()` work in the same way, except that a column vector of settings is input or returned. If *trace* is a scalar, then the trace specified is used for all integration problems.

## Performing integration

### q.integrate( )

With `Quadrature()`, *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.

With `QuadratureVec()`, *q*`.integrate()` does the same as with `Quadrature()`, except that it approximates a set of integrals and returns a column vector of computed quadrature values.

## Functions for obtaining results

After performing integration, the functions below provide results, including the value of the integrals, number of iterations, whether convergence was achieved, error messages, and return codes. The following describe the functions with `Quadrature()`; the functions work the same with `QuadratureVec()`, except that a column vector is returned instead of a scalar.

### q.value( )

*q*`.value()` returns the computed quadrature value or values; it returns missing 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 an 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 for both `Quadrature()` and `QuadratureVec()` are as follows:

| Error code | Return code | Error text |
|---|---|---|
| 1 | 111 | you must specify an evaluator function to compute numerical integration using `setEvaluator()` |
| 2 | 111 | you 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*[a] |
| 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 |

[a] *n* will be an actual number, and *i* will be a number less than *n*.

The following are applicable to `QuadratureVec()` only:

| Error code | Return code | Error text |
|---|---|---|
| 8 | 430 | no problem is defined in the class |
| 9 | 430 | error found for some problems in the class |
| 10 | 430 | you must specify at least one evaluator function to compute numerical integration using `setEvaluator()` |
| 11 | 430 | you must specify lower and upper integration limits as a matrix with 2 columns using `setLimits()` |

## Utility function

At any stage of solving the integration problem, you can obtain a report of all current settings and results with

### q.query( )

*q*.query() displays information stored in the class. It has no return value.

## Remarks and examples

stata.com

Remarks are presented under the following headings:

## Introduction

Quadrature() and QuadratureVec() are Mata classes for numerical integration. For an introduction to class programming in Mata, see [M-2] **class**.

Historically, quadrature means the process of determining area. The term is still used nowadays to refer to solutions in terms of integrals. Quadrature() and QuadratureVec() use the adaptive quadrature method to approximate integrals of the form $\int_a^b f(x)\,dx$, where $f(x)$ is a real-valued function and $a$ and $b$ are lower and upper limits, respectively.

## Examples

To approximate an integral, you first use Quadrature() or QuadratureVec() to create an instance of the class. At a minimum, you must also use setEvaluator() to specify the evaluator functions, setLimits() to specify the limits, and integrate() to perform the computations. In the examples below, we demonstrate both basic and more advanced uses of these Mata classes.

### A basic example of Quadrature()

▷ 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 use the address of the evaluator function in an instance of `Quadrature()`. Note that the addresses of built-in functions like `sin()` are not accessible.

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

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

Second, we use `setEvaluator()` to set a pointer to the evaluator function `f()`:

```
: 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 $\int_0^\pi \sin(x)\,dx = 2$. ◁

## A basic example of QuadratureVec()

### ▷ Example 2: Approximate a set of integrals

We want to approximate integrals $\int_1^2 2x \ \ dx$, $\int_0^\pi \sin(x)\,dx$, and $\int_0^1 \exp(x)\,dx$ using `QuadratureVec()`. We first define the corresponding evaluator functions as `f1()`, `f2()`, and `f3()`.

```
: real scalar f1(real scalar x) {
>         return(2*x)
> }
: real scalar f2(real scalar x) {
>         return(sin(x))
> }
: real scalar f3(real scalar x) {
>         return(exp(x))
> }
```

Having defined the evaluator functions, we follow the four steps that are required. First, we create an instance `q` of the `QuadratureVec()` class:

```
: q = QuadratureVec()
```

Second, we use `setEvaluator()` to point to the evaluator functions, defined as the column vector `evaluator`.

```
: evaluator = (&f1() \ &f2() \ &f3())
: q.setEvaluator(evaluator)
```

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

```
: limits = ((1, 2) \ (0,pi()) \ (0,1))
: q.setLimits(limits)
```

Fourth, we use `q.integrate()` to compute the approximations:

```
          : q.integrate()
                           1
              +-------------+
            1 |           3 |
            2 |           2 |
            3 | 1.718281828 |
              +-------------+
```

We find that $\int_1^2 2x\ dx = 3$, $\int_0^\pi \sin(x)\,dx = 2$, and $\int_0^1 \exp(x)\,dx = 1.718281828\ldots$. ◁

## Integrals with infinite limits

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

### ▷ Example 3: Approximate an integral from a finite point to $\infty$

To calculate the numerical integral of $\int_0^\infty \exp(-x)\,dx$, we define an evaluator function `f4()`, create an instance `q` of `Quadrature()`, set the evaluator, and set the limits:

```
          : real scalar f4(real scalar x) {
          >         return(exp(-x))
          > }
          : q = Quadrature()
          : q.setEvaluator(&f4())
          : q.setLimits((0, .))
```

We specify `.` to set the upper limit to $\infty$.

Now we can compute the approximation:

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

◁

### ▷ Example 4: Approximate an integral from $-\infty$ to $\infty$

To calculate the numerical integral of $\int_{-\infty}^{\infty} \exp(-x^2)\,dx$, we use commands similar to those in example 3, but we now define an evaluator function `f5()`:

```
          : real scalar f5(real scalar x) {
          >         return(exp(-x*x))
          > }
          : q = Quadrature()
          : q.setEvaluator(&f5())
          : q.setLimits((., .))
```

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

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 5: Add an extra argument to the evaluator function with Quadrature()

We want to calculate the numerical integral of $\int_0^1 (x^2 + z^2)\, dx$, where $z = 5$. The code is as follows:

```
: real scalar f6(real scalar x, real scalar z) {
>         return(x*x + z*z)
> }
: q = Quadrature()
: q.setEvaluator(&f6())
: q.setLimits((0, 1))
```

We use `setArgument()` to set 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
```

The value of the first extra argument can be obtained by typing

```
: *(q.getArgument(1))
  5
```

Note that `getArgument()` returns a pointer, which is why the monadic operator `*` is used. For more information on pointers in Mata, see [M-2] **pointers**.

◁

### ▷ Example 6: Add extra arguments to the evaluator functions with QuadratureVec()

We want to calculate the numerical integral of four functions, all with extra arguments. In the code below, the extra argument in `f7()` and `f8()` is the vector $y$; in `f9()`, it is the string $s$; and in `f10()`, it is a pointer to a function `f()`. We also define a function called `fcn()`.

```
: real scalar f7(real scalar x, real vector y) {
>         return(x + norm(y))
> }
: real scalar f8(real scalar x, real vector y) {
>         return(x*x + sum(y))
> }
: real scalar f9(real scalar x, string scalar s) {
>         return(sin(x) + strlen(s))
> }
: real scalar f10(real scalar x, pointer(function) scalar f) {
>         return(x + (*f)(0))
> }
: y = (1, 1.5, 10)
: s = "abc"
: real scalar fcn(real scalar x) {
>         return(-cos(x))
> }
```

We define a new instance S of QuadratureVec() and set the four evaluator functions and the limits from 0 to 1 for all integration problems.

```
: S = QuadratureVec()
: S.setEvaluator((&f7() \ &f8() \ &f9() \ &f10()))
: S.setLimits((0, 1))
```

Now we use setArgument() to set the extra arguments. For the first two problems, with evaluator functions f7() and f8(), extra argument 1 is set to vector $y$. For the third problem, which involves evaluator function f9(), extra argument 1 is set to string $s$. And, for the fourth integration problem, which involves function f10(), extra argument 1 is set to a pointer to the function fcn().

```
: S.setArgument((1 \ 2), 1, y)
: S.setArgument(3, 1, s)
: S.setArgument(4, 1, &fcn())
```

We can get the extra arguments as follows. We use getArgument() to get four pointers (one for each integration problem) for extra argument 1. To get the actual value of the extra argument, we use the monadic operator *. For more information on pointers in Mata, see [M-2] **pointers**.

```
: ptr = S.getArgument((1 \ 2 \ 3 \ 4), 1)
: (*ptr[1])
          1      2      3
    +----------------------+
  1 |     1    1.5     10  |
    +----------------------+

: (*ptr[2])
          1      2      3
    +----------------------+
  1 |     1    1.5     10  |
    +----------------------+

: (*ptr[3])
  abc
: (*(*ptr[4]))(0)
  -1
```

Now we can compute the approximations:

```
: S.integrate()
                  1
    +---------------+
  1 |  10.66120072  |
  2 |  12.83333333  |
  3 |  3.459697694  |
  4 |          -.5  |
    +---------------+
```

◁

## 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 $\infty$ or $-\infty$. In the second type, $f(x_0)$ is not continuous at $x_0$.

The methods implemented in the Quadrature() and QuadratureVec() classes 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 7: 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 f11(real scalar x) {
>         return(log(x))
> }
: q2.setEvaluator(&f11())
: q2.setLimits((0,1))
```

The log() function has a singular point at 0. We can compute the approximation by using the default absolute tolerance of 1e-10 and the 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 on the estimated absolute difference between the computed approximation and the exact solution. The relative tolerance is an upper bound on the estimated relative difference between the computed approximation and the exact solution.

◁

## Displaying settings and results at each stage

Each instance of the Quadrature() or QuadratureVec() class contains a lot of information about the problem at hand. You can use the utility function query() to display this information. Several other utility functions display specific pieces of information. The example below illustrates how to use these functions.

### ▷ Example 8: Display information about the integration problem

To calculate the numerical integral $\int_{-1}^1 |x|\, dx$, 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 f12(real scalar x) {
>         return(abs(x))
> }

: q3.setEvaluator(&f12())

: 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, adaptive Gauss–Kronrod quadrature (`gauss`), is almost always better than Simpson’s method (`simpson`). Simpson’s method is a benchmark method included for completeness.

The default maximum number of iterations is 16,000, and a warning message is printed 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
```

We can check the result after computation.

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

We can also check the number of iterations needed to converge.

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

No error was found, so the error code is 0, and the error text is an empty string.

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

We can show all the information with `query()` after the computation.

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

Evaluator
    Function:                                f12()

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
```

◁

## Solving 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. However, we recommend using QuadratureVec() because the latter makes it more convenient to define and solve a vector of integration problems.

### ▷ Example 9: Solve a vector of integrals

In this example, we show how both Quadrature() and QuadratureVec() can be used to solve the same set of problems.

First, 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 f13(real scalar x) {
>         return(sqrt(x))
> }
: real scalar f14(real scalar x) {
>         return(exp(x))
> }
: real scalar f15(real scalar x) {
>         return(1/sqrt(x))
> }
```

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

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

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

```
: qv[1].setEvaluator(&f13())
: qv[1].setLimits((0, 1))
: qv[2].setEvaluator(&f14())
: qv[2].setLimits((1, 2))
: qv[3].setEvaluator(&f15())
: 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
```

QuadratureVec() provides a more concise way to define and solve such a set of integration problems.

We first create QV, an instance of the QuadratureVec() class with

```
: QV = QuadratureVec()
```

The evaluator functions have to be defined as above and then can be set with

```
: evaluator = (&f13() \ &f14() \ &f15())
: QV.setEvaluator(evaluator)
```

The limits can be set with

```
        : limits = ((0, 1) \ (1, 2) \ (0, 0.5))
        : QV.setLimits(limits)
```

We use `integrate()` to compute all the approximations as follows:

```
        : QV.integrate()
                          1
            +---------------+
          1 |   .6666666667 |
          2 |    4.67077427 |
          3 |   1.414213562 |
            +---------------+
```

◁

## Conformability

In the descriptions below, if the input or output dimensions differ for `Quadrature()` and `QuadratureVec()`, both dimensions are specified—first for `Quadrature()`, followed by "or", and then for `QuadratureVec()`.

`Quadrature()`:

*input*:

|  |  |
|---|---|
|  | *void* |

*output*:

|  |  |
|---|---|
| *result*: | $1 \times 1$ |

`Quadrature(`*n*`)`:

*input*:

|  |  |
|---|---|
| $n$ : | $1 \times 1$ |

*output*:

|  |  |
|---|---|
| *result*: | $1 \times n$ |

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

*input*:

|  |  |
|---|---|
| $m$ : | $1 \times 1$ |
| $n$ : | $1 \times 1$ |

*output*:

|  |  |
|---|---|
| *result*: | $m \times n$ |

`QuadratureVec()`:

*input*:

|  |  |
|---|---|
|  | *void* |

*output*:

|  |  |
|---|---|
| *result*: | $1 \times 1$ |

`setEvaluator(`*fcn*`)`:

*input*:

|  |  |
|---|---|
| *fcn*: | $1 \times 1$ or $n \times 1$ |

*output*:

|  |  |
|---|---|
| *result*: | *void* |

`getEvaluator()`:

*input*:

|  |  |
|---|---|
|  | *void* |

*output*:

|  |  |
|---|---|
| *result*: | $1 \times 1$ or $n \times 1$ |

`setLimits(`*limits*`)`:

*input*:

|  |  |
|---|---|
| *limits*: | $1 \times 2$ or $n \times 2$ |

*output*:

|  |  |
|---|---|
| *result*: | *void* |

`getLimits()`:

*input*:

|  |  |
|---|---|
|  | *void* |

*output*:

|  |  |
|---|---|
| *result*: | $1 \times 2$ or $n \times 2$ |

`setTechnique(`*technique*`)`:

*input*:

|  |  |
|---|---|
| *technique*: | $1 \times 1$ or $n \times 1$ |

*output*:

|  |  |
|---|---|
| *result*: | *void* |

`getTechnique()`:

*input*:

|  |  |
|---|---|
|  | *void* |

*output*:

|  |  |
|---|---|
| *result*: | $1 \times 1$ or $n \times 1$ |

`setMaxiter(`*maxiter*`)`:

*input*:

|  |  |
|---|---|
| *maxiter*: | $1 \times 1$ or $n \times 1$ |

*output*:

|  |  |
|---|---|
| *result*: | *void* |

`getMaxiter()`:

*input*:

|  |  |
|---|---|
|  | *void* |

*output*:

|  |  |
|---|---|
| *result*: | $1 \times 1$ or $n \times 1$ |

`setAbstol(`*abstol*`)`:

*input*:

| | |
|---|---|
| *abstol*: | $1 \times 1$ or $n \times 1$ |

*output*:

| | |
|---|---|
| *result*: | *void* |

`getAbstol()`:

*input*:

| | |
|---|---|
| | *void* |

*output*:

| | |
|---|---|
| *result*: | $1 \times 1$ or $n \times 1$ |

`setReltol(`*reltol*`)`:

*input*:

| | |
|---|---|
| *reltol*: | $1 \times 1$ or $n \times 1$ |

*output*:

| | |
|---|---|
| *result*: | *void* |

`getReltol()`:

*input*:

| | |
|---|---|
| | *void* |

*output*:

| | |
|---|---|
| *result*: | $1 \times 1$ or $n \times 1$ |

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

*input*:

| | |
|---|---|
| *i*: | $1 \times 1$ |
| *arg*: | *anything* |

*output*:

| | |
|---|---|
| *result*: | *void* |

`getArgument(`*i*`)` (`Quadrature()` only):

*input*:

| | |
|---|---|
| *i*: | $1 \times 1$ |

*output*:

| | |
|---|---|
| *result*: | $1 \times 1$ |

`setArgument(`*pid*`, `*i*`, `*arg*`)` (`QuadratureVec()` only)

*input*:

| | |
|---|---|
| *pid*: | $n \times 1$ |
| *i*: | $1 \times 1$ |
| *arg*: | *anything* |

*output*:

| | |
|---|---|
| *result*: | *void* |

`getArgument(`*pid*`, `*i*`)` (`QuadratureVec()` only):

*input*:

| | |
|---|---|
| *pid*: | $n \times 1$ |
| *i*: | $1 \times 1$ |

*output*:

| | |
|---|---|
| *result*: | $n \times 1$ |

`getNarguments()` (`Quadrature()` only):

| | | |
|---|---|---|
| *input*: | | |
| | | *void* |
| *output*: | | |
| | *result*: | $1 \times 1$ |

`getDimension()` (`QuadratureVec()` only):

| | | |
|---|---|---|
| *input*: | | |
| | | *void* |
| *output*: | | |
| | *result*: | $1 \times 1$ |

`setTrace(`*trace*`)`:

| | | |
|---|---|---|
| *input*: | | |
| | *trace*: | $1 \times 1$ or $n \times 1$ |
| *output*: | | |
| | *result*: | *void* |

`getTrace()`:

| | | |
|---|---|---|
| *input*: | | |
| | | *void* |
| *output*: | | |
| | *result*: | $1 \times 1$ or $n \times 1$ |

`integrate()`:

| | | |
|---|---|---|
| *input*: | | |
| | | *void* |
| *output*: | | |
| | *result*: | $1 \times 1$ or $n \times 1$ |

`value()`:

| | | |
|---|---|---|
| *input*: | | |
| | | *void* |
| *output*: | | |
| | *result*: | $1 \times 1$ or $n \times 1$ |

`iterations()`:

| | | |
|---|---|---|
| *input*: | | |
| | | *void* |
| *output*: | | |
| | *result*: | $1 \times 1$ or $n \times 1$ |

`converged()`:

| | | |
|---|---|---|
| *input*: | | |
| | | *void* |
| *output*: | | |
| | *result*: | $1 \times 1$ or $n \times 1$ |

`errorcode()`:

*input*:

|  |  |
|---|---|
|  | *void* |

*output*:

|  |  |
|---|---|
| *result*: | $1 \times 1$ or $n \times 1$ |

`errortext()`:

*input*:

|  |  |
|---|---|
|  | *void* |

*output*:

|  |  |
|---|---|
| *result*: | $1 \times 1$ or $n \times 1$ |

`returncode()`:

*input*:

|  |  |
|---|---|
|  | *void* |

*output*:

|  |  |
|---|---|
| *result*: | $1 \times 1$ or $n \times 1$ |

`query()`:

*input*:

|  |  |
|---|---|
|  | *void* |

*output*:

|  |  |
|---|---|
|  | *void* |

## Diagnostics

When used incorrectly, all functions abort with an error. If `integrate()` runs into numerical difficulties, it returns a missing value and displays a warning message with details about the problem encountered.

## References

Davis, P. J., and P. Rabinowitz. 1984. *Methods of Numerical Integration*. 2nd ed. San Diego: 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-2] **pointers** — Pointers

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

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

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

[M-4] **Mathematical** — Important mathematical functions