DerivDiscreteDiff( ) — Finite difference coefficients for discrete numerical derivatives+

+This function is part of StataNow.

Description

The `DerivDiscreteDiff()` class computes the coefficients for a real, discrete numerical derivative of a given degree and accuracy using finite difference approximation.

Syntax

Syntax is presented under the following headings:

Step 1: Initialization

q = `DerivDiscreteDiff()`

Step 2: Definition of inputs and parameters

void	q`.setInput(`*real vector x*`)`
void	q`.setDegree(`*real scalar degree*`)`
void	q`.setAccuracy(`*real scalar acc*`)`
void	q`.setUseEqual(`*real scalar useeq*`)`
void	q`.setTol(`*real scalar tol*`)`
real colvector	q`.getInput()`
real scalar	q`.getDegree()`
real scalar	q`.getAccuracy()`
real scalar	q`.getUseEqual()`
real scalar	q`.getTol()`

Step 3: Perform computation

void	q`.solve()`

Step 4: Display or obtain results

real matrix	q`.getPoints()`
real matrix	q`.getCoefficients()`
real scalar	q`.converged()`
real scalar	q`.errorcode()`
string scalar	q`.errortext()`
real scalar	q`.returncode()`

Definition of q

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

```
q = DerivDiscreteDiff()
```

For a row vector of n `DerivDiscreteDiff()` instances, type

```
q = DerivDiscreteDiff(n)
```

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

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

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

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

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

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

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

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

Functions defining the inputs and parameters

At a minimum, you need to tell the `DerivDiscreteDiff()` class about the input vector, that is, over which points you are about to compute the derivative coefficients.

There is also a set of parameters that you may specify for the derivative, each of which has a default value if you do not set it.

Each pair of functions includes a `q.set*()` function that specifies a setting and a `q.get*()` function that returns the current setting.

q.setInput( ) and q.getInput( )

`q.setInput(x)` sets the input points. Note that *x* must be a sorted vector in ascending order with no duplicates.

`q.getInput()` returns the input points as a column vector (or an empty vector if the input points are not specified).

q.setDegree( ) and q.getDegree( )

`q.setDegree(degree)` sets the degree of the derivative. By default, the first-order derivative is set. *degree* must be a positive integer.

`q.getDegree()` returns the degree of the derivative.

q.setAccuracy( ) and q.getAccuracy( )

q`.setAccuracy(`*acc*`)` sets the computation accuracy of the derivative, that is, the order of the error. By default, *acc* is 2. *acc* must be a positive integer.

q`.getAccuracy()` returns the accuracy of the derivative.

q.setUseEqual( ) and q.getUseEqual( )

q`.setUseEqual(`*useeq*`)` sets an indicator for whether the input vector of points is equally spaced. A value of `1` (the default) means yes and `0` means no.

Note that the class will rely on the values set in this function and will not check whether the input vectors are equally spaced.

q`.getUseEqual()` returns the setting for whether to assume an equally spaced vector of points.

q.setTol( ) and q.getTol( )

q`.setTol(`*tol*`)` specifies the tolerance used for computation. The default value of *tol* is `1e-10`.

q`.getTol()` returns the currently specified tolerance.

Function for performing computation

q.solve( )

q`.solve()` invokes the computation process.

Functions for obtaining results

After computation, the functions below provide results, including points and their corresponding weighted coefficients, whether convergence was achieved or an error message or a return code was returned.

q.getPoints( )

q`.getPoints()` returns a matrix of stride indices. Each row of this matrix has a reference element, with index 0, corresponding to the point at which the derivative will be taken. The other values on the same row correspond to the negative or positive stride indices from the reference point.

q.getCoefficients( )

q`.getCoefficients()` returns the values of the weighted coefficients corresponding to the points indexed in *q*`.getPoints()`.

q.converged( )

q`.converged()` returns `1` if the computation 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	too few points are provided
2	430	the underlying linear system is unstable
3	111	dimension of the input is 0

Remarks and examples

Remarks are presented under the following headings:

Introduction

The DerivDiscreteDiff() class is a Mata class that computes the coefficients for a real discrete numerical derivative of a given degree and accuracy. Taylor expansion is used to approximate the derivative. The total number of points required for each derivative, including the reference point itself and some of its neighboring points, is based on the given degree and accuracy.

If *q*.setUseEqual() is set to 0, indicating the input points are not equally spaced, then each input point is used as a reference point in turn, and the corresponding coefficients are calculated at that point and its neighboring points. The number of coefficients computed for each reference point depends on the specified degree and accuracy. The total number of coefficient computations equals the number of input points.

If *q*.setUseEqual() is set to 1, indicating the input points are equally spaced, some of the interior points will have the same coefficients as long as they have enough neighboring points on both sides. Thus, unlike above where coefficients must be calculated for every input point, we need to compute the coefficients only at a small number of representative points. While the number of coefficients computed for each reference point still depends on the specified degree and accuracy, the total number of coefficient computations equals the number of unique positions required, which is again governed by the specified degree and accuracy.

For the computation of each coefficient, a system of linear equations with a Vandermode coefficient matrix is formed as in the underdetermined coefficients method, and the LU method is used to solve this linear system. The specified tolerance is for the LU method to determine whether a singular system is found. In that case, the system becomes unstable, and the results are not reliable as the specified degree and accuracy increase.

Examples

To find the coefficients, you first use `DerivDiscreteDiff()` to get an instance of the class. At a minimum, you must also use *q*`.setInput()` to specify the input points. In the examples below, we demonstrate both unequally spaced and equally spaced points.

▷ Example 1: Example with unequally spaced points

Consider the unequally spaced points $(-5, -2, 1, 2.5, 3)$.

We first define an instance of the `DerivDiscreteDiff()` class:

```
: q = DerivDiscreteDiff()
```

We then set the input points and specify that the points are not equally spaced:

```
: q.setInput((-5, -2, 1, 2.5, 3))
: q.setUseEqual(0)
```

Recall that, by default, the coefficients for the first-order derivative with second-order accuracy are computed. We type

```
: q.solve()
```

We check the points by typing

```
: q.getPoints()
         1    2    3
    +----------------+
  1 |    0    1    2 |
  2 |   -1    0    1 |
  3 |   -1    0    1 |
  4 |   -1    0    1 |
  5 |   -2   -1    0 |
    +----------------+
```

Here, for each row, the element equal to `0` is the reference point at which the derivative is taken. The number of rows equals the number of input points.

Positive values indicate the neighboring point indices that are on the right side of the reference point. For instance, `1` here means the point adjacent to the right of the reference point, `2` means the second point to the right of the reference point, and so on.

Negative values indicate the neighboring point indices that are on the left side of the reference point. For instance, `-1` here means the point adjacent to the left of the reference point, `-2` means the second point to the left of the reference point, and so on.

The corresponding coefficients can be found by typing

```
: q.getCoefficients()
                   1              2              3
    +----------------------------------------------+
  1 |            -.5    .6666666667   -.1666666667 |
  2 |   -.1666666667              0    .1666666667 |
  3 |   -.1111111111   -.3333333333    .4444444444 |
  4 |   -.1666666667   -1.333333333            1.5 |
  5 |    .1666666667   -2.666666667            2.5 |
    +----------------------------------------------+
```

For greater precision, we set the accuracy to a higher value and recompute the derivative:

```
: q.setAccuracy(4)

: q.solve()

: q.getPoints()
          1    2    3    4    5
    ┌──────────────────────────┐
  1 │     0    1    2    3    4 │
  2 │    -1    0    1    2    3 │
  3 │    -2   -1    0    1    2 │
  4 │    -3   -2   -1    0    1 │
  5 │    -4   -3   -2   -1    0 │
    └──────────────────────────┘

: q.getCoefficients()
                    1               2               3               4
    ┌─────────────────────────────────────────────────────────────────
  1 │  -.7583333333     1.777777778    -3.333333333     5.688888889
  2 │         -.0625   -.4222222222            1.25    -1.777777778
  3 │   .0083333333   -.0888888889    -.6666666667     1.422222222
  4 │       -.003125    .0277777778          -.3125    -.9777777778
  5 │   .0046296296   -.0395061728     .3703703704    -3.160493827
    └─────────────────────────────────────────────────────────────────

                    5
     ─────────────────┐
  1            -3.375 │
  2            1.0125 │
  3             -.675 │
  4          1.265625 │
  5             2.825 │
     ─────────────────┘
```

◁

▷ Example 2: Example with equally spaced points

Consider 10 equally spaced points between 1 and 2, inclusive.

```
: x = rangen(1, 2, 10)

: x
                  1
     ┌──────────────┐
   1 │            1 │
   2 │  1.111111111 │
   3 │  1.222222222 │
   4 │  1.333333333 │
   5 │  1.444444444 │
   6 │  1.555555556 │
   7 │  1.666666667 │
   8 │  1.777777778 │
   9 │  1.888888889 │
  10 │            2 │
     └──────────────┘
```

To compute the second-order derivative with an accuracy of 5, we define an instance of the `DerivDiscreteDiff()` class, set the input points and specify that the points are equally spaced, and then compute the solution:

```
: q = DerivDiscreteDiff()
: q.setInput(x)
: q.setDegree(2)
: q.setAccuracy(5)
: q.setUseEqual(1)
: q.solve()
```

We check the results by typing

```
: q.getPoints()
         1     2     3     4     5     6     7
    ┌──────────────────────────────────────────┐
  1 │    0     1     2     3     4     5     6  │
  2 │   -1     0     1     2     3     4     5  │
  3 │   -2    -1     0     1     2     3     4  │
  4 │   -3    -2    -1     0     1     2     3  │
  5 │   -4    -3    -2    -1     0     1     2  │
  6 │   -5    -4    -3    -2    -1     0     1  │
  7 │   -6    -5    -4    -3    -2    -1     0  │
    └──────────────────────────────────────────┘

: q.getCoefficients()
               1          2          3          4          5          6          7
    ┌──────────────────────────────────────────────────────────────────────────────┐
  1 │      365.4    -1409.4    2369.25      -2286     1336.5     -437.4      61.65  │
  2 │      61.65     -66.15    -114.75      211.5    -128.25      41.85      -5.85  │
  3 │      -5.85      102.6       -189         90       6.75       -5.4         .9  │
  4 │         .9     -12.15      121.5     -220.5      121.5     -12.15         .9  │
  5 │         .9       -5.4       6.75         90       -189      102.6      -5.85  │
  6 │      -5.85      41.85    -128.25      211.5    -114.75     -66.15      61.65  │
  7 │      61.65     -437.4     1336.5      -2286    2369.25    -1409.4      365.4  │
    └──────────────────────────────────────────────────────────────────────────────┘
```

Here we can see the number of rows returned by *q*.`getPoints()` equals the number of unique positions required, which is the sum of degree and accuracy.

◁

Conformability

`DerivDiscreteDiff():`

input:		
		void
output:		
	result:	1×1

`DerivDiscreteDiff(`*n*`):`

input:		
	n:	1×1
output:		
	result:	$1 \times n$

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

input:		
	m:	1×1
	n:	1×1
output:		
	result:	$m \times n$

q`.setInput(`*x*`):`

input:		
	x:	$1 \times N$ or $N \times 1$
output:		
	result:	*void*

q`.getInput():`

input:		
		void
output:		
	result:	$N \times 1$

q`.setDegree(`*degree*`):`

input:		
	degree:	1×1
output:		
	result:	*void*

q`.getDegree():`

input:		
		void
output:		
	result:	1×1

q`.setAccuracy(`acc`):`

input:

acc:	1×1

output:

result:	*void*

q`.getAccuracy():`

input:

void

output:

result:	1×1

q`.setUseEqual(`$useeq$`):`

input:

useeq:	1×1

output:

result:	*void*

q`.getUseEqual():`

input:

void

output:

result:	1×1

q`.setTol(`tol`):`

input:

tol:	1×1

output:

result:	*void*

q`.getTol():`

input:

void

output:

result:	1×1

q`.solve():`

input:

void

output:

result:	*void*

`q.getPoints()`:

input:

void

output:

result: *N1* × *N2*

`q.getCoefficients()`:

input:

void

output:

result: *N1* × *N2*

`q.converged()`:

input:

void

output:

result: 1 × 1

`q.errorcode()`:

input:

void

output:

result: 1 × 1

`q.errortext()`:

input:

void

output:

result: 1 × 1

`q.returncode()`:

input:

void

output:

result: 1 × 1

Diagnostics

`DerivDiscreteDiff()`, `q.set*()`, `q.get*()`, `q.solve()`, `q.converged()`, `q.errorcode()`, `q.errortext()`, and `q.returncode()` functions abort with an error message when used incorrectly.

Reference

Hassan, H. Z., A. A. Mohamad, and G. E. Atteia. 2012. An algorithm for the finite difference approximation of derivatives with arbitrary degree and order of accuracy. *Journal of Computational and Applied Mathemetics* 236: 2622–2631. https://doi.org/10.1016/j.cam.2011.12.019.

Also see

[M-5] **DerivDiscretePartial( )** — Compute discrete numerical partial derivatives$^+$

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