DerivDiscretePartial( ) — Compute discrete numerical partial derivatives[+]

[+]This function is part of StataNow.

Description

The `DerivDiscretePartial()` class computes discrete numerical partial derivatives at a list of points.

Syntax

Syntax is presented under the following headings:

Step 1: Initialization

q = `DerivDiscretePartial()`

Step 2: Definition of inputs and parameters

void	*q*.setXValues(*pointer vector x*)
void	*q*.setYValues(*real vector data*, *real vector size*)
void	*q*.setDirection(*real vector direction*)
void	*q*.setDegree(*real vector degree*)
void	*q*.setAccuracy(*real vector acc*)
void	*q*.setUseEqual(*real vector useeq*)
void	*q*.setTol(*real scalar tol*)
pointer colvector	*q*.getXValues()
class NDMatrix scalar	*q*.getYValues()
real colvector	*q*.getDirection()
real colvector	*q*.getDegree()
real colvector	*q*.getAccuracy()
real colvector	*q*.getUseEqual()
real scalar	*q*.getTol()

Step 3: Perform computation

void	*q*.solve()

Step 4: Display or obtain results

class NDMatrix scalar	*q*.getDerivatives()
real scalar	*q*.converged()
real scalar	*q*.errorcode()
string scalar	*q*.errortext()
real scalar	*q*.returncode()

Definition of q

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

```
q = DerivDiscretePartial()
```

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

```
q = DerivDiscretePartial(n)
```

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

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

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

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

You can instead declare *q* as a row vector of *n* instances by typing

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

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

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

Functions defining inputs and parameters

At a minimum, you need to tell the `DerivDiscretePartial()` class about input vectors *x* and *y*.

Optionally, you may specify the directions, the degrees, and the accuracies of the derivatives. You may also specify whether the input vector *x* of points is equally spaced.

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

q.setXValues( ) and q.getXValues( )

q`.setXValues(`*x*`)` sets the *x* values as a vector of pointers to real values. Each element in *x* points to a real vector that corresponds to a list of points (that is, points on the support of the function to be differentiated) sorted in ascending order with no duplicates.

q`.getXValues()` returns the *x* values as a vector of pointers.

q.setYValues( ) and q.getYValues( )

q`.setYValues(`*data*`,` *size*`)` sets the multidimensional *y* values in *data* as a vector and the lengths for each of the dimensions in *size* as a vector as well. Internally, these inputs will be stored as an instance of the `NDMatrix()` class.

If the actual dimension of the input matrix is $n_1 \times n_2 \times \cdots \times n_d$, then the length of *data* is $\prod_{i=1}^{d} n_i$, and the value of *size* should be $(n_1, n_2, \ldots, n_d)$.

q.getYValues() returns the instance of the NDMatrix() class that stores the multidimensional data and its size.

q.setDirection( ) and q.getDirection( )

q.setDirection(*direction*) sets the directions of differentiation as a vector indicating along which dimensions the derivative is to be taken. The default value for *direction* is 1.

q.getDirection() returns a vector of derivative directions.

q.setDegree( ) and q.getDegree( )

q.setDegree(*degree*) sets the degrees of differentiation as a vector indicating what the orders of the derivative are. The default value for *degree* is 1.

q.getDegree() returns a vector of the derivative degrees.

q.setAccuracy( ) and q.getAccuracy( )

q.setAccuracy(*acc*) sets the accuracies for differentiation as a vector. The default value for *acc* is 2.

q.getAccuracy() returns a vector of the derivative accuracies.

q.setUseEqual( ) and q.getUseEqual( )

q.setUseEqual(*useeq*) sets a vector of indicators for whether the input vectors of points are 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 computation tolerance. The default value of *tol* is 1e-8. See more details in [M-1] **Tolerance**.

q.getTol() returns the currently specified tolerance.

Function for performing differentiation

q.solve( )

q.solve() computes the discrete numerical derivative based on the inputs.

Functions for obtaining results

After performing the differentiation, the functions below return the derivatives, report convergence, and provide details on any errors.

q.getDerivatives( )

q`.getDerivatives()` returns the computed derivative as an instance of class `NDMatrix()`.

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	430	there are errors while computing weights
2	430	dimensions do not conform
3	111	dimension of the input is 0
4	430	convergence is not achieved for some derivatives

Remarks and examples

Remarks are presented under the following headings:

Introduction

The `DerivDiscretePartial()` class is a Mata class for computing discrete numerical derivatives.

Let x_i be a vector of length n_i, and let px_i be a pointer to the vector x_i, where $i = 1, 2, \ldots, d$. Let $y = f(x_1, x_2, \ldots, x_d)$ be a multidimensional matrix representing some unknown function on $(x_1, x_2, \ldots, x_d)$.

With this class, we can take any partial or mixed derivative of y with respect to some x_i's.

At a minimum, you need to tell the `DerivDiscretePartial()` class about input vectors x and y. The argument for the method `q.setXValues()` should be a vector of pointers $(px_1, px_2, \ldots, px_d)$, where px_i points to x_i for $i = 1, 2, \ldots, d$. The arguments for the method `q.setYValues()` should be a vector *data* that represents y as a vector of length $\prod_{i=1}^{d} n_d$ and a vector *size* that equals $(n_1, n_2, \ldots, n_d)$.

For a vector of directions $(j_1, j_2, \ldots, j_k)$ and a vector of degrees $(m_1, m_2, \ldots, m_k)$, the mixed partial derivative of y is

$$\frac{\partial^{\sum_{i=1}^{k} m_i} y}{\partial^{m_1} x_{j_1} \partial^{m_2} x_{j_2} \ldots \partial^{m_k} x_{j_k}}$$

In this case, the argument for the method `q.setDirection()` should be a vector $(j_1, j_2, \ldots, j_k)$, and the argument for the method `q.setDegree()` should be a vector $(m_1, m_2, \ldots, m_k)$.

We may also set the accuracies $(acc_1, acc_2, \ldots, acc_k)$ for each part of the mixed partial derivatives using `q.setAccuracy()` and whether the input vectors are equally spaced $(useeq_1, useeq_2, \ldots, useeq_k)$ using `q.setUseEqual()`.

Note that if all the values of `q.setAccuracy()` are the same, then we can use a scalar instead of a vector as the argument. For example, if $acc := acc_1 = acc_2 = \cdots = acc_k$, then we can use *acc* as the argument for `q.setAccuracy()`. This is also true for `q.setDirection()`, `q.setDegree()`, and `q.setUseEqual()`.

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

Examples

To solve a discrete numerical derivative problem, you first use `DerivDiscretePartial()` to get an instance of the class. At a minimum, you must also use `q.setXValues()` and `q.setYValues()` to specify the input x's and corresponding y's.

▷ Example 1: Ordinary derivative

Consider the function $f(x) = \sin(x)$.

We first define an instance of the class:

```
: q = DerivDiscretePartial()
```

And then we set the input x's and corresponding y's. Remember that we also need to set the dimension of y.

```
: x = rangen(0, 1, 10)
: y = sin(x)
: q.setXValues(&x)
: q.setYValues(y, length(y))
```

Recall that the default direction is set to `1` and that the default degree of derivative is also set to `1`. We assume the input x is equally spaced. We then set the computation accuracy to `5` and compute the derivative:

```
: q.setAccuracy(5)
: q.solve()
```

We check the result by typing

```
: q.getDerivatives().getData()
                  1
    ┌──────────────┐
  1 │   .999999336 │
  2 │  .9938336501 │
  3 │  .9754100103 │
  4 │  .9449570257 │
  5 │  .9028497782 │
  6 │  .8496076999 │
  7 │  .7858874242 │
  8 │  .7124748124 │
  9 │  .6302746686 │
 10 │  .5403042494 │
    └──────────────┘
```

Now consider an unequally spaced input x:

```
: x = rangen(0, 0.5, 5) \ rangen(0.51, 1, 10)
: y = sin(x)
: q.setXValues(&x)
: q.setYValues(y, length(y))
```

We need to indicate the input is not equally spaced and then compute the derivative by typing

```
: q.setUseEqual(0)
: q.solve()
: q.getDerivatives().getData()
                  1
    ┌──────────────┐
  1 │  .9999989701 │
  2 │  .9921978752 │
  3 │   .968912322 │
  4 │  .9305077042 │
  5 │   .877582566 │
  6 │   .872744509 │
  7 │  .8448859261 │
  8 │   .814523552 │
  9 │  .7817473667 │
 10 │  .7466544989 │
 11 │  .7093489463 │
 12 │  .6699412626 │
 13 │  .6285482313 │
 14 │  .5852925003 │
 15 │  .5403023675 │
    └──────────────┘
```

◁

▷ Example 2: Partial derivative

Consider the function $f(x,y,z) = (x+y+z)^3 \exp(y^2)\log(z)$ and the problem of approximating $(\partial y)/(\partial x)$ and $(\partial^2 y)/(\partial y \partial z)$.

We first define an instance of the class and set the input x's and corresponding f's. Remember that we also need to set the dimension of f.

```
: q = DerivDiscretePartial()
: x = rangen(-1, 2, 10)
: y = rangen(1, 2, 30)
: z = rangen(1, 1.5, 25) \ rangen(1.51, 2, 25)
: q.setXValues((&x, &y, &z))
: f = J(0, 1, 0)
: dfdx = J(0, 1, 0)
: d2fdydz = J(0, 1, 0)
: for (i = 1; i <= length(x); i++) {
>         for (j = 1; j <= length(y); j++) {
>                 for (k = 1; k <= length(z); k++) {
>                     f = f \ (x[i]+y[j]+z[k])^3 * exp(y[j]^2) * log(z[k])
>                     dfdx = dfdx
>                       \ 3*(x[i]+y[j]+z[k])^2*exp(y[j]^2)*log(z[k])
>                     d2fdydz = d2fdydz
>                       \ 6*(x[i]+y[j]+z[k])^2*y[j]*exp(y[j]^2)*log(z[k])
>                       + 6*(x[i]+y[j]+z[k])*exp(y[j]^2)*log(z[k])
>                       + 2*(x[i]+y[j]+z[k])^3*y[j]*exp(y[j]^2)/z[k]
>                       + 3*(x[i]+y[j]+z[k])^2*exp(y[j]^2)/z[k]
>                 }
>         }
> }
: q.setYValues(f, (length(x), length(y), length(z)))
```

Here we also compute the exact solutions for $(\partial y)/(\partial x)$ and $(\partial^2 y)/(\partial y \partial z)$ so that we can compare them against our results.

To approximate $(\partial y)/(\partial x)$, recall that the default direction is set to `1` and that the default degree of differentiation is set to `1`. We assume the input x is equally spaced. We then set the computation accuracy to `4` and compute the derivative:

```
: q.setAccuracy(4)
: q.solve()
```

We check the relative difference between our result and the exact solution:

```
: mreldif(q.getDerivatives().getData(), dfdx)
  1.58247e-14
```

Now consider the mixed derivative $(\partial^2 y)/(\partial y \partial z)$. We need to specify that one of the inputs is not equally spaced before we compute the derivative:

```
: q.setDirection((2, 3))
: q.setAccuracy(6)
: q.setUseEqual((1, 0))
: q.solve()
```

We again check the relative difference between our result and the exact solution:

```
: mreldif(q.getDerivatives().getData(), d2fdydz)
  7.94318e-06
```

Conformability

`DerivDiscretePartial()`:

input:

void

output:

result: 1×1

`DerivDiscretePartial(`*n*`)`:

input:

n: 1×1

output:

result: $1 \times n$

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

input:

m: 1×1

n: 1×1

output:

result: $m \times n$

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

input:

x: $1 \times N$ or $N \times 1$

output:

result: *void*

q`.getXValues()`:

input:

void

output:

result: $N \times 1$

q`.setYValues(`*data*`,` *size*`)`:

input:

data: $1 \times M$ or $M \times 1$

size: $1 \times N$ or $N \times 1$

output:

result: *void*

q`.getYValues()`:

input:

void

output:

result: 1×1

q.setDirection(*direction*):

input:

direction: $1 \times N$ or $N \times 1$ or 1×1

output:

result: *void*

q.getDirection():

input:

void

output:

result: $N \times 1$ or 1×1

q.setDegree(*degree*):

input:

degree: $1 \times N$ or $N \times 1$ or 1×1

output:

result: *void*

q.getDegree():

input:

void

output:

result: $N \times 1$ or 1×1

q.setAccuracy(*acc*):

input:

acc: $1 \times N$ or $N \times 1$ or 1×1

output:

result: *void*

q.getAccuracy():

input:

void

output:

result: $N \times 1$ or 1×1

q.setUseEqual(*useeq*):

input:

useeq: $1 \times N$ or $N \times 1$ or 1×1

output:

result: *void*

q.getUseEqual():

input:

void

output:

result: $N \times 1$ or 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.getDerivatives():

input:

void

output:

result: 1×1

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

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

Also see

[M-5] **DerivDiscreteDiff( )** — Finite difference coefficients for discrete numerical derivatives+

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