Basic

derivative

Returns the slope or instantaneous rate of change of a function f at the independent variable value of x.

```
derivative(f, x, h=pow(10, -9))
```

Parameters:

- **f** : *function*
  - A mathematical function to take the derivative of.
- **x** : *int or float*
  - The independent variable value at which to take the derivative of the function.
- **h :** *float, optional*
  - The very small change in the independent variable's value. By default, it is equal to 0.000000001 or 10^{-9}.

Let's see an example:

```
from math import sin
from Basic import derivative

def f(x):
    return math.sin(x)

# We'll make print the derivative value at x=2:
```

```
val = derivative(f, 2)

print(val)
```

integral

Computes the area bounded by the function f, the axis of the independent variable, and two independent variable values of a and b. It computes the integral via Simpson's rule.

```
integral(f, a, b, n=pow(10, 6))
```

Parameters:

- **f** : *function*
  - A mathematical function to take the integral of.
- **a** : *int or float*
  - The left bound of the integral.
- **b** : *int or float*
  - The right bound of the integral.
- **n** : *int, optional*
  - The number of subintervals to use in the integration process. By default, it is equal to a million or 1000000.

Let's see an example:

```
from math import cos

from Basic import integral

def f(x):
```

```
    return math.cos(x)

# We'll set the bounds to a=0 and b=pi and then print the result:
val = integral(f, a, b)

print(val)
```

Vector

Vector

A class of a type of mathematical expression often used in linear algebra and kinematics (the physics of motion). It has multiple values combined together like a one-column or one-row matrix. Vectors usually either 2D (having an x and y component) or 3D (having an x, y, and z component) but in this module the user/developer is allowed to make 4D vectors. 4D vectors have an x component, a y component, a z component, and a w component.

class Vector(x, y, z=None, w=None)

Parameters:

- **x** : *int or float*
  - The first component of a vector.
- **y** : *int or float*
  - The second component of a vector.
- **z** : *int or float, optional*
  - The third component of a vector that is option and only needed for 3D and 4D vectors. By default, its value is None.
- **w** : *int or float, optional*
  - The third component of a vector that is option and only needed for 4D vectors. By default, its value is None.

v_add

Returns a new vector in which each of its components are the sums of the corresponding components of two vectors.

`self.v_add(other)`

Parameters:

- **self** : *Vector*
  - The first vector.
- **other** : *Vector*
  - The second vector.

v_subtract

Returns a new vector in which each of its components are the differences of the corresponding components of two vectors.

`self.v_subtract(other)`

Parameters:

- **self** : *Vector*
  - The first vector.
- **other** : *Vector*
  - The second vector.

dot_product

Multiplies the corresponding components of two vectors together and then sums all the products up.

`self.dot_product(other)`

Parameters:

- **self** : *Vector*
    - The first vector.
- **other** : *Vector*
    - The second vector.

cross_product

Cross-multiplying the components of two vectors as though they're matrices. This function only works when done on two vectors that are both 3D and neither of them should have any less nor more dimensions than 3.

```
self.cross_product(other)
```

Parameters:

- **self** : *Vector*
    - The first vector.
- **other** : *Vector*
    - The second vector.

magnitude

Returns the magnitude of a vector. This function works for 2D, 3D and 4D vectors.

```
self.magnitude()
```

Parameters:

- **self** : *Vector*
    - The vector that you're trying to find the magnitude of.

angle2D

Returns the angle, in radians, of a vector counterclockwise from the unit vector $\hat{\imath}$ $(+x)$. In later versions of Calcify, there will be functions for 3D and 4D variables.

```
self.angle2D()
```

Parameters:

- **self** : *Vector*
    - The vector that you're trying to find the 2D angle of.

Multivariable

partial_derivative_x

Computes the partial derivative of a multivariable scalar function, f, with respect to the first variable which is usually x.

```
partial_derivative_x(f, x, y, h=pow(10, -9), z=None)
```

Parameters:

- **f** : *function*
  - A mathematical function to take the partial derivative of.
- **x** : *int or float*
  - One of the independent variable values at which to take the partial derivative of the function.
- **y** : *int or float*
  - One of the independent variable values at which to take the partial derivative of the function.
- **h :** *float, optional*
  - The very small change in the independent variable's value. By default, it is equal to 0.000000001 or 10^{-9}.
- **z** : *int or float, optional*
  - One of the independent variable values at which to take the partial derivative of the function. It is equal to None by default for functions of only 2 independent variables but can be changed to any real number for functions of 3 independent variables.

partial_derivative_y

Computes the partial derivative of a multivariable scalar function, f, with respect to the second variable which is usually y.

```
partial_derivative_y(f, x, y, h=pow(10, -9), z=None)
```

Parameters:

- **f** : *function*
  - A mathematical function to take the partial derivative of.
- **x** : *int or float*
  - One of the independent variable values at which to take the partial derivative of the function.
- **y** : *int or float*
  - One of the independent variable values at which to take the partial derivative of the function.
- **h :** *float, optional*
  - The very small change in the independent variable's value. By default, it is equal to 0.000000001 or 10^{-9}.
- **z** : *int or float, optional*
  - One of the independent variable values at which to take the partial derivative of the function. It is equal to None by default for functions of only 2 independent variables but can be changed to any real number for functions of 3 independent variables.

partial_derivative_z

Computes the partial derivative of a multivariable scalar function, f, with respect to the third variable which is usually z. It only works for functions with 3 independent variables.

```
partial_derivative z (f, x, y, z, h=pow(10, -9))
```

Parameters:

- **f** : *function*

    - A mathematical function to take the partial derivative of.
- **x** : *int or float*
    - One of the independent variable values at which to take the partial derivative of the function.
- **y** : *int or float*
    - One of the independent variable values at which to take the partial derivative of the function.
- **z** : *int or float*
    - One of the independent variable values at which to take the partial derivative of the function. For the other partial derivative functions in this module, this input is optional, but in this specific function it is mandatory.
- **h :** *float, optional*
    - The very small change in the independent variable's value. By default, it is equal to 0.000000001 or 10^{-9}.

gradient

Returns a vector of the partial derivatives with respect to x and y (and an optional z variable) of a function of 2 or 3 independent variables, f.

```
gradient(f, x, y, z=None)
```

Parameters:

- **f** : *function*
    - A mathematical function to take the gradient of.
- **x** : *int or float*
    - One of the independent variable values at which to take the gradient of the function.
- **y** : *int or float*
    - One of the independent variable values at which to take the gradient of the function.
- **z** : *int or float, optional*
    - One of the independent variable values at which to take the partial derivative of the function. It is equal to None by default for functions of only 2

independent variables but can be changed to any real number for functions of 3 independent variables.

double_integral

Calculates the volume bounded by the surface of a function f, a and b on one of the independent variables, c and d on another one of the independent variables, and the x-y plane. Like the regular integral function in the Basic module, it uses Simpson's rule.

```
double_integral(f, a, b, c, d, n=pow(10, 3))
```

Parameters:

- **f** : *function*
  - A mathematical function to take the double integral of.
- **a** : *int or float*
  - The left bound of the integral on the x-axis.
- **b** : *int or float*
  - The right bound of the integral on the x-axis.
- **c** : *int or float*
  - The left bound of the integral on the y -axis.
- **d** : *int or float*
  - The right bound of the integral on the y-axis.
- **n** : *int, optional*
  - The number of subintervals to use in the integration process. By default, it is equal to a thousand or 1000.

triple_integral

Calculates the hyper-volume bounded by the hyper-surface of a function f, a and b on one of the independent variables, c and d on another one of the independent

variables, g and h on another one of the independent variables, and the x-y-z space, or hyper-plane. Like the regular integral function in the Basic module and the double_integral function in this module, it uses Simpson's rule.

triple_integral(f, a, b, c, d, g, h)

Parameters:

Parameters:

- **f** : *function*
  - A mathematical function to take the triple integral of.
- **a** : *int or float*
  - The left bound of the integral on the x-axis.
- **b** : *int or float*
  - The right bound of the integral on the x-axis.
- **c** : *int or float*
  - The left bound of the integral on the y -axis.
- **d** : *int or float*
  - The right bound of the integral on the y-axis.
- **g** : *int or float*
  - The left bound of the integral on the z-axis.
- **h** : *int or float*
  - The right bound of the integral on the z-axis.
- **n** : *int, optional*
  - The number of subintervals to use in the integration process. By default, it is equal to a hundred or 100.

directional_derivative

For a 2D, 3D, or 4D parametric function of t symbolized as f, the derivative at a particular direction is taken. The particular direction is notated with a vector.

directional_derivative(f, t, vector)

Parameters:

- **f** : *function*
    - A list of mathematical functions to take the directional derivative of. It can be 2D, 3D, or 4D.
- **t** : *int or float*
    - The independent variable value at which to take the directional derivative of the function.
- **vector :** *Vector*
    - The vector that determines the direction where with to take the derivative of the function.

divergence

In mathematical terms, it uses the dot product between the partial derivative operations and the values of the components of a vector function. It only works on 3D vector functions in 3D space.

```
divergence(vector_function, x, y, z)
```

Parameters:

- **vector_function** : *Vector*
    - A vector composed of multiple functions as its components. You pretend it's a mathematical function to take the divergence of.
- **x** : *int or float*
    - One of the independent variable values at which to take the divergence of the function.
- **y** : *int or float*
    - One of the independent variable values at which to take the divergence of the function.
- **z** : *int or float*
    - One of the independent variable values at which to take the divergence of the function.

curl2D

In mathematical terms, it takes a cross multiplying of the partial derivative operations and the components of the vector function as though they're matrices. It only works on 2D vector functions in 2D space. It returns a scalar.

```
curl2D(vector_function, x, y)
```

Parameters:

- **vector_function** : *Vector*
  - A vector composed of multiple functions as its components. You pretend it's a mathematical function to take the curl of.
- **x** : *int or float*
  - One of the independent variable values at which to take the curl of the function.
- **y** : *int or float*
  - One of the independent variable values at which to take the curl of the function.

curl3D

In mathematical terms, it takes the cross product of the partial derivative operations and the components of the vector function as though they're 3D vectors. It only works on 3D vector functions in 3D space. It returns a vector.

```
curl3D
```

Parameters:

- **vector_function** : *Vector*

    - A vector composed of multiple functions as its components. You pretend it's a mathematical function to take the curl of.
- **x** : *int or float*
    - One of the independent variable values at which to take the curl of the function.
- **y** : *int or float*
    - One of the independent variable values at which to take the curl of the function.
- **z** : *int or float*
    - One of the independent variable values at which to take the curl of the function.

Polar

polar_to_cartesian

Returns a list of x and y components of a 2D parametric function derived from a polar function.

```
polar_to_cartesian(r, theta)
```

Parameters:

- **r** : *function*
  - A polar function used to symbolize the radius and to take the directional derivative of.
- **theta** : *int or float*
  - The independent angle variable.

cartesian_to_polar

Converts the cartesian coordinates of a parametric function into the radius coordinate with the angle coordinate represented by the parametric independent variable, t.

```
cartesian_to_polar(f, t)
```

Parameters:

- **f** : *function*
  - A list of mathematical functions to take the directional derivative of. It can only be a list of 2 functions to represent a 2D parametric function.
- **t** : *int or float*

- The independent variable value which also represents the angle variable.

polar_integral

Finds the area swept by the enclosure of the polar function r from angle alpha to angle beta.

```
polar_integral(r, alpha, beta)
```

Parameters:

- **r** : *function*
  - A polar function to take the integral of.
- **a** : *int or float*
  - The more clockwise angular bound of the integral.
- **b** : *int or float*
  - The more counterclockwise angular bound of the integral.

ODE

solve_first_order

Returns a numpy array of a solution of a first order differential equation. For Calcify 1.0.0, only the Euler method is currently available. Not only that, but there are no PDE functions in the Calcify 1.0.0 package but there will most likely be some in future versions.

```
solve_first_order(y, yp, x_range, n=po(10, 6), method="euler")
```

Parameters:

- **y** : *int or float*
  - The initial condition for the dependent variable of the solution.
- **yp** : *function*
  - The function of both the dependent and independent variables that represents the derivative of y with respect to x.
- **x_range** : *list*
  - A list of 2 independent variable values representing the bounds of the domain of the solution.
- **n** : *int, optional*
  - Represents the number of iterations for Euler's method and how many y values will be in the solution. By default, it is equal to a million or 1000000.

solve_second_order

Returns a numpy array of a solution of a second order differential equation. For Calcify 1.0.0, only the Euler method is currently available. Not only that, but there are no PDE functions in the Calcify 1.0.0 package but there will most likely be some in future versions.

solve_second_order(y, yp, ypp, x_range, n=pow(10, 6)

Parameters:

- **y** : *int or float*
  - The initial condition for the dependent variable of the solution.
- **yp** : *int or float*
  - The initial condition for the first derivative of y with respect to x.
- **ypp** : *function*
  - The function of the dependent and independent variables and the first derivative that represents the second derivative of y with respect to x.
- **x_range** : *list*
  - A list of 2 independent variable values representing the bounds of the domain of the solution.
- **n** : *int, optional*
  - Represents the number of iterations for Euler's method and how many y values will be in the solution. By default, it is equal to a million or 1000000.