Examples

May 18, 2024

[ ]:
```python
import numerics as nm
```

[ ]:
```python
t = [0, 10, 20, 30, 35, 40]
v = [0, 95.05, 322.07, 912.03, 1227.05, 1615.37]

nm.poly_interpolation(t, v, 25, 1) # Interpolating at 25 with polynomial of␣
 ↪degree 1 (linear)
```

[ ]:
```
array([-857.85 ,   58.996])
```

[ ]:
```python
nm.poly_interpolation(t, v, 25, 2) #Quadratic interpolaation
```

[ ]:
```
array([-6.9753e+02,  4.5636e+01,  2.6720e-01])
```

[ ]:
```python
nm.poly_interpolation(t, v, 25, 3) #Cubic interpolation
```

[ ]:
```
array([-1.95627e+03,  1.86495e+02, -4.82770e+00,  5.99400e-02])
```

[ ]:
```python
nm.linear_spline_interpolation(t, v, 25) # Linear spline interpolation; Gives␣
 ↪coefficients of all the splines
```

[ ]:
```
array([[    0.   ,     9.505],
       [ -131.97 ,    22.702],
       [ -857.85 ,    58.996],
       [ -978.09 ,    63.004],
       [-1491.19 ,    77.664]])
```

[ ]:
```python
nm.quadratic_spline_interpolation(t, v, 25) #Quadratic spline interpolationl;␣
 ↪Gives coefficients of all the splines
```

[ ]:
```
array([[ 1.13686838e-15,  9.50500000e+00,  0.00000000e+00],
       [ 1.31970000e+00, -1.68890000e+01,  1.31970000e+02],
       [ 2.30970000e+00, -5.64890000e+01,  5.27970000e+02],
       [-3.81780000e+00,  3.11161000e+02, -4.98678000e+03],
       [ 6.74980000e+00, -4.28571000e+02,  7.95853000e+03]])
```

[ ]:
```python
nm.nddp(t, v, 25, 1) #Linear nddp interpolation at t = 25
```

[ ]:
```
617.05
```

[ ]:
```
nm.nddp(t, v, 25, 2) #Quadratic nddp interpolation at t = 25
```

[ ]:
```
610.37
```

[ ]:
```
nm.nddp(t, v, 25, 3) #Cubic nddp interpolation at t = 25
```

[ ]:
```
625.355
```

[ ]:
```
def func(x, y):
    return y

nm.euler_first(func, 0.5, 2, 1, 0) #First order euler solver with 05 step size;␣
 ↪x_stop=2, y(0)=1, x_start=0
```

[ ]:
```
array([1.    , 1.5   , 2.25  , 3.375 , 5.0625])
```

[ ]:
```
nm.rk_2(func, 0.5, 2, 1, 0, method='heun')#RK 2 integration using 0.5 step size␣
 ↪from 0 to 2 with y(0) = 1
#Using Heun method. Default is heun method
```

[ ]:
```
array([1.        , 1.625     , 2.640625  , 4.29101562, 6.97290039])
```

[ ]:
```
nm.rk_2(func, 0.5, 2, 1, 0, method='midpoint')#RK 2 integration using 0.5 step␣
 ↪size from 0 to 2 with y(0) = 1
#Using midpoint method
```

[ ]:
```
array([1.        , 1.625     , 2.640625  , 4.29101562, 6.97290039])
```

[ ]:
```
nm.rk_2(func, 0.5, 2, 1, 0, method='ralston') #RK 2 integration using 0.5 step␣
 ↪size from 0 to 2 with y(0) = 1
# Using ralston method
```

[ ]:
```
array([1.        , 1.625     , 2.640625  , 4.29101562, 6.97290039])
```

[ ]:
```
nm.rk_4(func, 0.5, 2, 1, 0) #RK 4 integration using 0.5 step size from 0 to 2␣
 ↪with y(0) = 1
```

[ ]:
```
array([1.        , 1.6484375 , 2.71734619, 4.47937536, 7.38397032])
```

[ ]:
```
x = [2, 4, 6, 8, 10]
y = [7, 11, 15, 19, 23]
nm.poly_regression(x, y, 1) #Polynomial regression using polynomial of degree 1␣
 ↪(Linear)
# Returns coefficients of linear polynomial
```

[ ]:
```
array([3., 2.])
```

[ ]:
```
x = [1, 2, 3, 4, 5]
y = [9, 24, 47, 78, 117]
nm.poly_regression(x, y, 2) #Quadratic regression
# Returns coefficients of quadratic polynomial
```

[ ]:
```
array([2., 3., 4.])
```

[ ]:
```
x = [0.2, 0.4, 0.6, 0.8, 1]
y = [0.5437, 1.4778, 4.0171, 10.9196, 29.6826]
nm.exp_regression(x, y) #exponential regreesion
#Returns coeffieients of exponential function
```

[ ]:
```
(0.20001214955547886, 4.999921092489269)
```

[ ]:
```
x = [0.2, 0.4, 0.6, 0.8, 1]
y = [0.02, 0.08, 0.18, 0.32, 0.5]
nm.pow_regression(x, y) #powerlaw regression
#Returns power law coefficients (Typo in Sir's note as they are given ulta)
```

[ ]:
```
(0.4999999999999998, 1.9999999999999996)
```

[ ]:
```
def func(x):
    return x*x*x-9
nm.bisection(func, 2, 3, 1.e-6, 1.e-2)
# Uses bisection method for root finding
#Interval starts at 2 and ends at 3
#Tollerance in x interval 10^(-6)
#Tollerance in y interval 10^(-2)
#Lower tollerance value taken
#Returns x value, y value near 0 and number of itterations required
```

[ ]:
```
(2.0800838470458984, 3.1144795542559223e-07, 18)
```

[ ]:
```
def dfunc(x):
    return 3*x*x
nm.newton_raphson(func, dfunc, 3, 1.e-6, 1.e-6)
#Takes extra argumets of derivative and guess value of 3
```

[ ]:
```
(2.0800838230519054, 1.7763568394002505e-14, 5)
```

[ ]:
```
nm.secant(func, 4, 3, 1.e-6, 1.e-6)
#Takes first guess as 4 and 2nd guess as 3
```

[ ]:
```
(2.0800838230519845, 1.042721464727947e-12, 7)
```

[ ]:
```
import numpy as np
def func(x):
    return x*np.exp(-x)
```

```python
nm.trapezoidal(func, 2, 0.25, 2.5)
# Trapezoidal rule of integration using 3 intervals from 0.25 to 2.5
```

[ ]: 0.6160621391947412

[ ]:
```python
nm.simpsons1_3(func, 4, 0.25, 2.5)
#Simpsons 1/3 rule using 4 segments (minimum 2 segments reuired)
```

[ ]: 0.6851050681107044

[ ]:
```python
nm.simpsons3_8(func, 3, 0.25, 2.5)
#Simpson's 3/8 rule using 3 segments (Minimum 3 seegments required)
```

[ ]: 0.6794618115815894