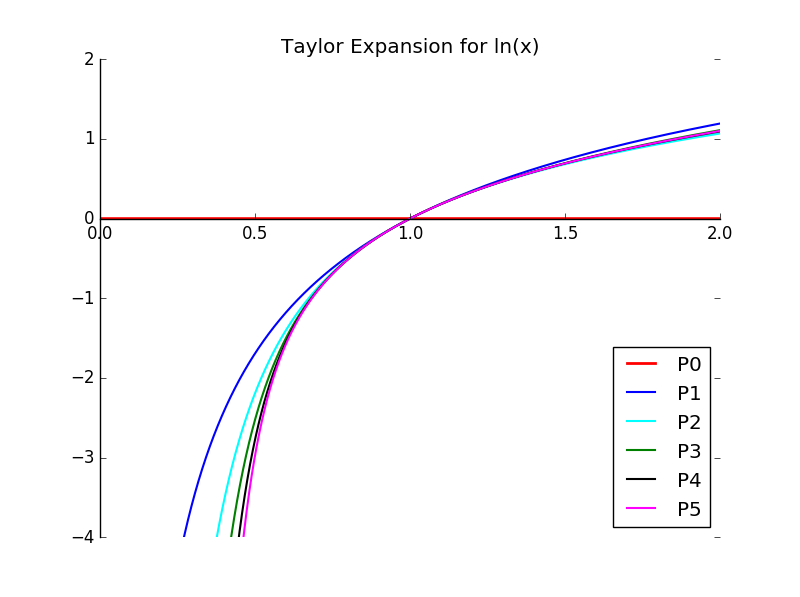
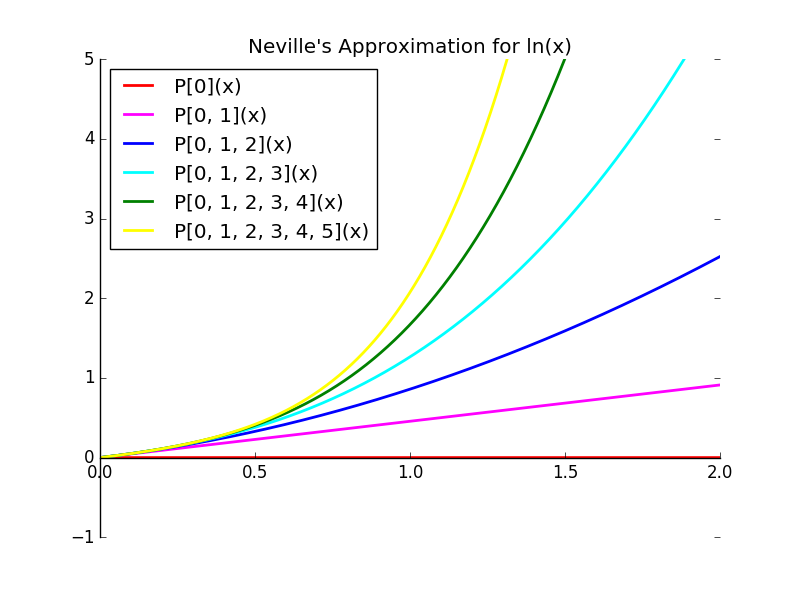
# 1

|  |
| --- |
| from \_\_future\_\_ import print\_function  from sympy import \*  from math import factorial  import numpy as np  import matplotlib.pyplot as plt  def maximum(funct, start, end):  functderivative = diff(funct, x)  zeros = solve(functderivative,x)  zeros = [zero for zero in zeros if start<zero<end]  tmpmax = funct.subs(x, start)  if funct.subs(x, end) > tmpmax:  tmpmax = funct.subs(x, end)  for num in zeros:  if funct.subs(x, num) > tmpmax:  tmpmax = funct.subs(x, num)  return tmpmax  x = symbols("x") # define x to be a symbol  #functions and derivatives  f = ln(x)  fprime1 = diff(ln(x), x)  fprime2 = diff(fprime1, x)  fprime3 = diff(fprime2, x)  fprime4 = diff(fprime3, x)  fprime5 = diff(fprime4, x)  fprime6 = diff(fprime5, x)  # Taylor Series approximations of ln(x)  P0 = f.subs(x, 1)  P1 = f + fprime1\*(x-1)  P2 = P1 + fprime2\* (x-1)\*\*2 /factorial(2)  P3 = P2 + fprime3\* (x-1)\*\*3 /factorial(3)  P4 = P3 + fprime4\* (x-1)\*\*4 /factorial(4)  P5 = P4 + fprime5\* (x-1)\*\*5 /factorial(5)  print("P0(x) = " + str(P0))  print("P1(x) = " + str(P1))  print("P2(x) = " + str(P2))  print("P3(x) = " + str(P3))  print("P4(x) = " + str(P4))  print("P5(x) = " + str(P5))  #print approximations and errors  print("P0(1.5) = " + str(P0.subs(x, 1.5)), end=" ")  print("with maximum error: " + str(abs(maximum(fprime1, 1, 1.5)\*(1.5-1)\*\*1/factorial(1))))  print("P1(1.5) = " + str(P1.subs(x, 1.5)), end=" ")  print("with maximum error: " + str(abs(maximum(fprime2, 1, 1.5)\*(1.5-1)\*\*2/factorial(2))))  print("P2(1.5) = " + str(P2.subs(x, 1.5)), end=" ")  print("with maximum error: " + str(abs(maximum(fprime3, 1, 1.5)\*(1.5-1)\*\*3/factorial(3))))  print("P3(1.5) = " + str(P3.subs(x, 1.5)), end=" ")  print("with maximum error: " + str(abs(maximum(fprime4, 1, 1.5)\*(1.5-1)\*\*4/factorial(4))))  print("P4(1.5) = " + str(P4.subs(x, 1.5)), end=" ")  print("with maximum error: " + str(abs(maximum(fprime5, 1, 1.5)\*(1.5-1)\*\*5/factorial(5))))  print("P5(1.5) = " + str(P5.subs(x, 1.5)), end=" ")  print("with maximum error: " + str(abs(maximum(fprime6, 1, 1.5)\*(1.5-1)\*\*6/factorial(6))))  # set axis labels  ax = plt.gca()  ax.spines['right'].set\_color('none')  ax.spines['top'].set\_color('none')  ax.xaxis.set\_ticks\_position('bottom')  ax.spines['bottom'].set\_position(('data',0))  ax.set\_ylim([-4, 2])  # add in first line  x\_1 = np.arange(0.001, 2, 0.00001);  f = lambdify( (x), P0 )  f = np.vectorize(f)  plt.plot(x\_1, f(x\_1), linewidth="2", color="red", label="P0")  f1 = lambdify( (x), P1 )  f1 = np.vectorize(f1)  plt.plot(x\_1, f1(x\_1), linewidth="1.5", color="blue", label="P1")  f2 = lambdify( (x), P2 )  f2 = np.vectorize(f2)  plt.plot(x\_1, f2(x\_1), linewidth="1.5", color="cyan", label="P2")  f3 = lambdify( (x), P3 )  f3 = np.vectorize(f3)  plt.plot(x\_1, f3(x\_1), linewidth="1.5", color="green", label="P3")  f4 = lambdify( (x), P4 )  f4 = np.vectorize(f4)  plt.plot(x\_1, f4(x\_1), linewidth="1.5", color="black", label="P4")  f5 = lambdify( (x), P5 )  f5 = np.vectorize(f5)  plt.plot(x\_1, f5(x\_1), linewidth="1.5", color="magenta", label="P5")  # add legend  plt.legend(loc='best')  plt.title('Taylor Expansion for ln(x)')  plt.show() |
| P0(x) = 0  P1(x) = log(x) + (x - 1)/x  P2(x) = log(x) + (x - 1)/x - (x - 1)\*\*2/(2\*x\*\*2)  P3(x) = log(x) + (x - 1)/x - (x - 1)\*\*2/(2\*x\*\*2) + (x - 1)\*\*3/(3\*x\*\*3)  P4(x) = log(x) + (x - 1)/x - (x - 1)\*\*2/(2\*x\*\*2) + (x - 1)\*\*3/(3\*x\*\*3) - (x - 1)\*\*4/(4\*x\*\*4)  P5(x) = log(x) + (x - 1)/x - (x - 1)\*\*2/(2\*x\*\*2) + (x - 1)\*\*3/(3\*x\*\*3) - (x - 1)\*\*4/(4\*x\*\*4) + (x - 1)\*\*5/(5\*x\*\*5)  P0(1.5) = 0 with maximum error: 0.500000000000000  P1(1.5) = 0.738798441441498 with maximum error: 0.0555555555555556  P2(1.5) = 0.683242885885942 with maximum error: 0.0416666666666667  P3(1.5) = 0.695588564898288 with maximum error: 0.00308641975308642  P4(1.5) = 0.692502145145201 with maximum error: 0.00625000000000000  P5(1.5) = 0.693325190412691 with maximum error: 0.000228623685413809 |



# 2

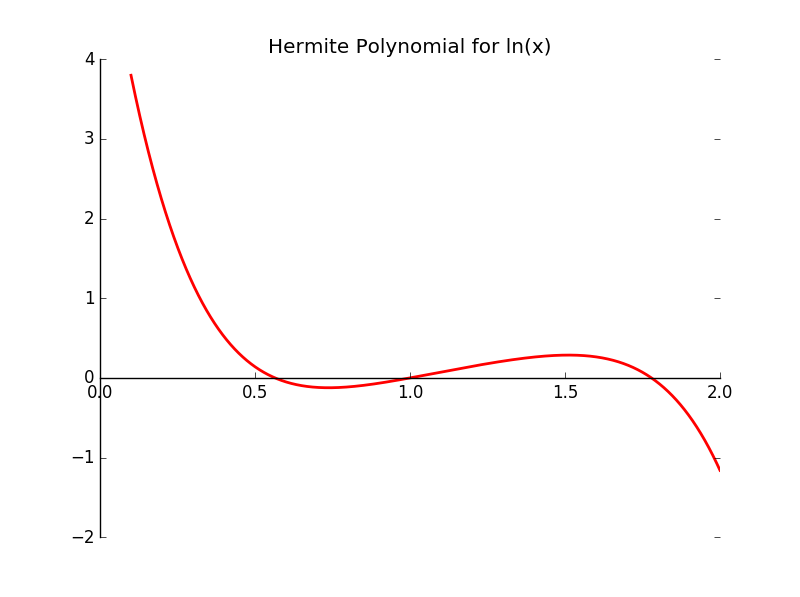
|  |
| --- |
| '''  Neville's Method  '''  from \_\_future\_\_ import print\_function  from math import log  import numpy as np  import matplotlib.pyplot as plt  from sympy import \*  def funct(x):  return log(x)  x = [1, 1.2, 1.4, 1.6, 1.8, 2.0]  y = [funct(val) for val in x]  value = 1.5  P = y  Q = np.zeros((len(x), len(x)))  Q[:, 0] = x  Q[:, 0] = y  for i in range(len(x)):  for j in range(1,i+1):  Q[i,j] = ((value-x[i-j])\*Q[i,j-1] - (value-x[i])\*Q[i-1,j-1])/(x[i]-x[i-j])  print("Q: \n" + str(Q), end="\n\n")  # set up graph  # set axis labels  ax = plt.gca()  ax.spines['right'].set\_color('none')  ax.spines['top'].set\_color('none')  ax.xaxis.set\_ticks\_position('bottom')  ax.spines['bottom'].set\_position(('data',0))  colors = ['red', 'magenta', 'blue', 'cyan', 'green', 'yellow', 'black']  x\_range = np.arange(0.001, 2, 0.000001);  x\_sym = symbols("x")  ax.set\_ylim([-1, 5])  #output polynomials, values, and errors  for j in range(len(x)):  P = 0  print("P" + str(range(j+1)) + "(x): ", end=" ")  num = 0  for i in range(j+1):  if i!= j:  print(str(Q[i,i]) + "(x-1)\*\*" + str(i), end=" + ")  else:  print(str(Q[i,i]) + "(x-1)\*\*" + str(i))  P += Q[i,i] \* (x\_sym)\*\*i  num += Q[i,i]\*(1.5-1)\*\*i  print("P" + str(range(j+1)) + "(1.5): " + str(num), end=" ")  print("with error: " +str(abs(num-log(1.5))))  f = lambdify( (x\_sym), P )  f = np.vectorize(f)  plt.plot(x\_range, f(x\_range), linewidth="2", color=colors[i], label="P"+str(range(j+1))+"(x)")  plt.legend(loc='best')  plt.title("Neville's Approximation for ln(x)")  plt.show() |
| Q:  [[ 0. 0. 0. 0. 0. 0. ]  [ 0.18232156 0.45580389 0. 0. 0. 0. ]  [ 0.33647224 0.41354758 0.4029835 0. 0. 0. ]  [ 0.47000363 0.40323793 0.40581534 0.40534337 0. 0. ]  [ 0.58778666 0.41111211 0.40520648 0.40551091 0.40544808 0. ]  [ 0.69314718 0.42974589 0.40645367 0.40541434 0.4054747 0.40546139]]  P[0](x): 0.0(x-1)\*\*0  P[0](1.5): 0.0 with error: 0.405465108108164  P[0, 1](x): 0.0(x-1)\*\*0 + 0.455803891985(x-1)\*\*1  P[0, 1](1.5): 0.227901945992 with error: 0.177563162115721  P[0, 1, 2](x): 0.0(x-1)\*\*0 + 0.455803891985(x-1)\*\*1 + 0.402983497672(x-1)\*\*2  P[0, 1, 2](1.5): 0.328647820411 with error: 0.0768172876976384  P[0, 1, 2, 3](x): 0.0(x-1)\*\*0 + 0.455803891985(x-1)\*\*1 + 0.402983497672(x-1)\*\*2 + 0.405343369474(x-1)\*\*3  P[0, 1, 2, 3](1.5): 0.379315741595 with error: 0.0261493665134424  P[0, 1, 2, 3, 4](x): 0.0(x-1)\*\*0 + 0.455803891985(x-1)\*\*1 + 0.402983497672(x-1)\*\*2 + 0.405343369474(x-1)\*\*3 + 0.405448082736(x-1)\*\*4  P[0, 1, 2, 3, 4](1.5): 0.404656246766 with error: 0.000808861342415201  P[0, 1, 2, 3, 4, 5](x): 0.0(x-1)\*\*0 + 0.455803891985(x-1)\*\*1 + 0.402983497672(x-1)\*\*2 + 0.405343369474(x-1)\*\*3 + 0.405448082736(x-1)\*\*4 + 0.405461390154(x-1)\*\*5  P[0, 1, 2, 3, 4, 5](1.5): 0.417326915208 with error: 0.0118618070998847 |



# 3

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| x | f(x) | f'(x) = 1/x |  |  |  |  |
| 1 | 0 | 0.714285714 | 0.317237193 | -1.586185966 | 2.14247946 | -2.748202 |
| 1 | 0 | 0.841180592 | -0.317237193 | 0.127797605 | -0.0560819 |  |
| 1.4 | 0.3364722 | 0.714285714 | -0.214999109 | 0.082932053 |  |  |
| 1.4 | 0.3364722 | 0.628286071 | -0.181826288 |  |  |  |
| 1.8 | 0.5877867 | 0.555555556 |  |  |  |  |
| 1.8 | 0.5877867 |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
| P(x) | 0 + 0.714285714285714\*(x-1) + 0.317237193168295\*(x-1)\*\*2 + -1.58618596584148\*(x-1)\*\*3 + 2.14247946387946\*(x-1)\*\*4 + -2.74820175568131\*(x-1)\*\*5 | | | | | |
| P(1.5) | 0.2862026 |  |  |  |  |  |
| 1.5 | 1 |  |  |  |  |  |
| f(1.5) | 0.4054651 |  |  |  |  |  |
| Absolute Error | 0.1192625 |  |  |  |  |  |

|  |
| --- |
| #plotting  import numpy as np  import matplotlib.pyplot as plt  from math import log  def fun(x):  return 0 + 0.714285714285714\*(x-1) + 0.317237193168295\*(x-1)\*\*2 + -1.58618596584148\*(x-1)\*\*3 + 2.14247946387946\*(x-1)\*\*4 + -2.74820175568131\*(x-1)\*\*5  print(fun(1.5))  x = np.arange(0.1, 2, 0.000001);  vfun = np.vectorize(fun)  y = vfun(x)  # set axis labels  ax = plt.gca()  ax.spines['right'].set\_color('none')  ax.spines['top'].set\_color('none')  ax.xaxis.set\_ticks\_position('bottom')  ax.spines['bottom'].set\_position(('data',0))  plt.title('Hermite Polynomial for ln(x)')  plt.plot(x, map(fun,x), linewidth="2", color="red")  plt.show() |



# 4

## --Extracredit--

|  |
| --- |
| '''Free Cubic Spline'''  from \_\_future\_\_ import print\_function  import math  def f(x):  return math.log(x)  def fprime(x):  return 1/x  x = [1.0, (4.0/3), (5.0/3), 2.0]  y = [f(num) for num in x]  n = len(x) - 1  n= len(x)  b = [0]\*n  c = [0]\*n  d = [0]\*n  h = [1]\*n  for i in range(n-1):  h[i] = x[i+1] - x[i]  alpha = [0]\*n  for i in range(1, n-1):  alpha[i] = (3/h[i]) \* (y[i+1] - y[i]) - (3/h[i-1]) \* (y[i] - y[i-1])  # need to tri-diagonalize  l = [1]\*n  u = [0]\*n  z = [0]\*n  c = [0]\*n  for i in range(1, n-1):  l[i] = 2\*(x[i+1] - x[i-1]) - h[i-1]\*u[i-1]  u[i] = h[i]/l[i]  z[i] = (alpha[i] - h[i-1]\*z[i-1])/l[i]  l[n-1] = 1  z[n-1] = 0  c[n-1] = 0  #calculate c, b, and d  for j in range(n-2, -1, -1):  c[j] = z[j] - u[j]\*c[j+1]  b[j] = (y[j+1] - y[j])/h[j] - h[j]\*(c[j+1]+2\*c[j])/3  d[j] = (c[j+1] - c[j])/(3\*h[j])  #output  string= ""; value = 1.5  for i in range(len(x)-1):  print(str(y[i])+ " + " + str(b[i]) + " (x-" + str(x[i]) + ") + " +  str(c[i])+ " (x-" + str(x[i]) + ")\*\*2 + " + str(d[i])+ " (x-" + str(x[i]) + ")\*\*3" , end=",\t")  print("for " + str(x[i]) + " < x < " + str(x[i+1]))  if x[i] < value and value<= x[i+1]:  string = "P("+str(value)+") = " + str(y[i]+ b[i] \*(value -x[i]) + c[i]\*(value- x[i])\*\*2 + d[i]\*(value-x[i])\*\*3)  print(string) |
| 0.0 + 0.906512635361 (x-1.0) + 0.0 (x-1.0)\*\*2 + -0.391197762054 (x-1.0)\*\*3, for 1.0 < x < 1.33333333333  0.287682072452 + 0.776113381343 (x-1.33333333333) + -0.391197762054 (x-1.33333333333)\*\*2 + 0.213448739556 (x-1.33333333333)\*\*3, for 1.33333333333 < x < 1.66666666667  0.510825623766 + 0.586464453159 (x-1.66666666667) + -0.177749022498 (x-1.66666666667)\*\*2 + 0.177749022498 (x-1.66666666667)\*\*3, for 1.66666666667 < x < 2.0  P(1.5) = 0.407155886783  Absolute error: 0.0016907786748356357 |
| '''Clamped Cubic Spline'''  from \_\_future\_\_ import print\_function  import math  def f(x):  return math.log(x)  def fprime(x):  return 1/x  x = [1.0, (4.0/3), (5.0/3), 2.0]  n = len(x)-1  y = [f(num) for num in x]  FPO = fprime(x[0]) # fprime(x\_0)  FPN = fprime(x[n]) # fprime(x\_n)  n = len(x)  b = [0]\*n; c = [0]\*n; d = [0]\*n; h = [1]\*n  for i in range(n-1):  h[i] = x[i+1] - x[i]  alpha = [0]\*n  alpha[0] = 3\*(y[1]-y[0])/h[0] - 3\* FPO  alpha[n-1] = 3\*FPN - 3\*(y[n-1] - y[n-2])/h[n-1]  for i in range(1, n-1):  alpha[i] = (3/h[i]) \* (y[i+1] - y[i]) - (3/h[i-1]) \* (y[i] - y[i-1])  # need to tri-diagonalize  l = [1]\*n  u = [0]\*n  z = [0]\*n  c = [0]\*n  l[0] = 2 \* h[0]  u[0] = .5  z[0] = alpha[0]/l[0]  for i in range(1, n-1):  l[i] = 2\*(x[i+1] - x[i-1]) -h[i-1]\*u[i-1]  u[i] = h[i] / l[i]  z[i] = (alpha[i] - (h[i]/l[i]))/l[i]  for i in range(1, n-1):  l[i] = 2\*(x[i+1] - x[i-1]) - h[i-1]\*u[i-1]  u[i] = h[i]/l[i]  z[i] = (alpha[i] - h[i-1]\*z[i-1])/l[i]    l[n-1] = 1  z[n-1] = 0  c[n-1] = 0  #calculate c, b, and d  for j in range(n-2, -1, -1):  c[j] = z[j] - u[j]\*c[j+1]  b[j] = (y[j+1] - y[j])/h[j] - h[j]\*(c[j+1]+2\*c[j])/3  d[j] = (c[j+1] - c[j])/(3\*h[j])  #output  string= ""; value = 1.5  for i in range(len(x)-1):  print(str(y[i])+ " + " + str(b[i]) + " (x-" + str(x[i]) + ") + " +  str(c[i])+ " (x-" + str(x[i]) + ")\*\*2 + " + str(d[i])+ " (x-" + str(x[i]) + ")\*\*3" , end=",\t")  print("for " + str(x[i]) + " < x < " + str(x[i+1]))  if x[i] < value and value<= x[i+1]:  string = "P("+str(value)+") = " + str(y[i]+ b[i] \*(value -x[i]) + c[i]\*(value- x[i])\*\*2 + d[i]\*(value-x[i])\*\*3)  print(string) |
| 0.0 + 1.0 (x-1.0) + -0.485415162547 (x-1.0)\*\*2 + 0.223661443839 (x-1.0)\*\*3, for 1.0 < x < 1.33333333333  0.287682072452 + 0.750943706248 (x-1.33333333333) + -0.261753718708 (x-1.33333333333)\*\*2 + 0.0516436853736 (x-1.33333333333)\*\*3, for 1.33333333333 < x < 1.66666666667  0.510825623766 + 0.593655788901 (x-1.66666666667) + -0.210110033335 (x-1.66666666667)\*\*2 + 0.210110033335 (x-1.66666666667)\*\*3, for 1.66666666667 < x < 2.0  P(1.5) = 0.405807511332 |

Absolute error: 0.00034240322383560784