Advanced hump analysis with `scipy()` Mathematical Programming with Python

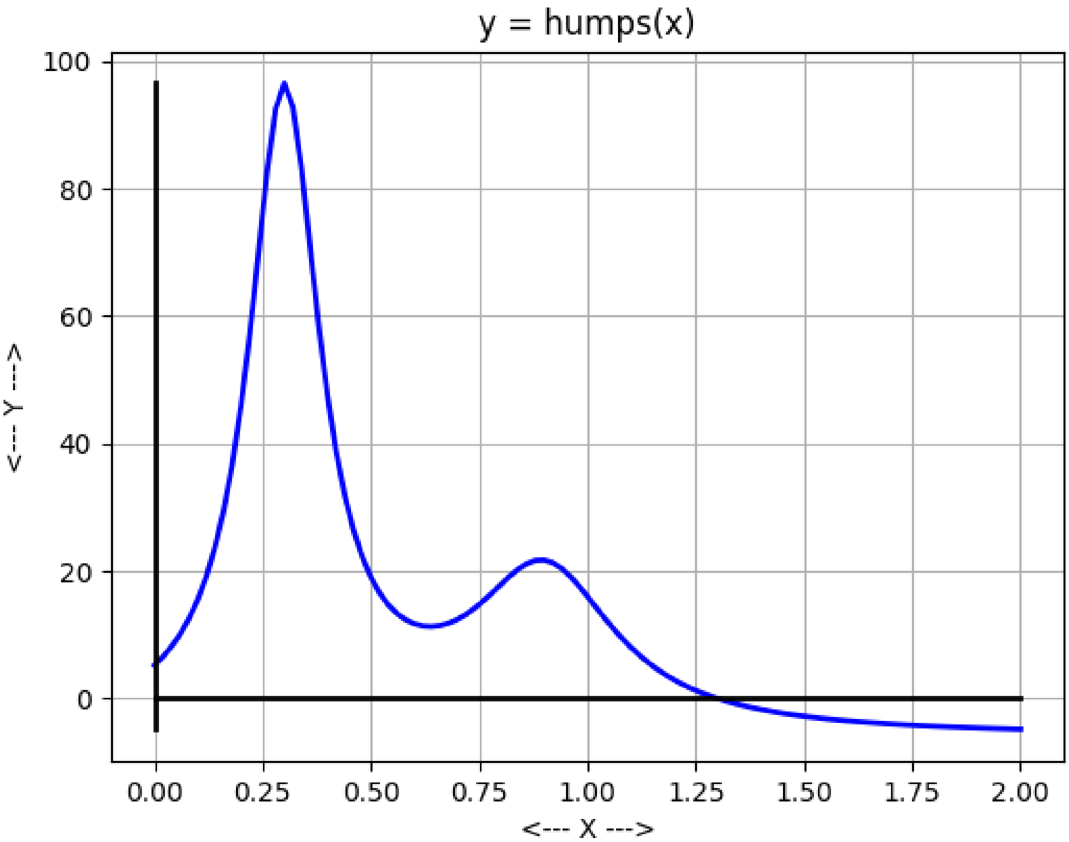
MATH 2604: Advanced Scientific Computing 4
Spring 2025
Monday/Wednesday/Friday, 1:00-1:50pm

https://people.sc.fsu.edu/~jburkardt/classes/python_2025/humps2/humps2.pdf

---

Today we will demonstrate some more advanced `scipy` functions by working on a test examples made famous by MATLAB, known as `humps(x)`. This function is defined mathematically as:

$$y(x) = \frac{1}{(x-0.3)^2+0.01} + \frac{1}{(x-0.9)^2+0.04} - 6$$



The humps(x) function for $0 \leq x \leq 2$.

We will generally focus on this function over the interval $0 \leq x \leq 2$.

We will try to use `scipy` to investigate properties of this function. From a plot, we can see that the function seems to have a zero near $x = 1.25$, a local minimum near $x = 0.6370$, and two local maximum values near $x = 0.3$ and $x = 0.9$. The function values at the endpoints are $y(0) = 5.1764...$ and $y(2) = -4.8551....$ The integral of the function over $[0, 2]$ is approximately 29.3262... We will now look at how, instead of guessing from a plot, we could determine this information by calling the appropriate `scipy` functions.

The file `humps.py` will contain functions we will find useful during this work:

- `humps_antideriv(x)`: antiderivative function;
- `humps_deriv(x)`: first derivative;
- `humps_deriv2(x)`: second derivative;
- `humps_fun(x)`: evaluates humps(x);
- `humps_ode(x,y)`: like humps_deriv(), but includes y as second argument.

1 Where does `humps(x)` have a root?

The `scipy.optimize` library includes several functions which seek a root of a function $f(x)$, that is, a value such that $f(x) = 0$. A cautious coder might want to include the derivative value $f'(x)$ for a Newton method, and the careless coder might not specify a change of sign interval, inside of which a root is guaranteed. We will take the semi-cautious approach. We know that $f(0) = 5.1764...$ and $f(2) = -4.8551...$ and so (assuming continuity!) there must be a value within this interval at which the value of zero is reached.

We can use the function `brentq()` to seek this value:

```
def humps_zero( ):
  from humps import humps_fun
  from scipy.optimize import brentq
  import numpy as np

  x = brentq ( humps_fun, 0.0, 2.0 )

  print ( '  Root found at x = ', x )
  print ( '  f(x) = ', humps_fun(x) )
  return
```

2 Estimate the integral $I = \int_0^2 \textbf{humps}(x)dx$

The `scipy.integrate` library can estimate the integral over some interval $[a, b]$ or rectangular domain, of a function $f(x)$ given as a formula. It can also estimate integrals when the function is only available as sample data values. Special cases can handle integration over rectangular regions in 2D and 3D.

The integrator `quad()` is recommended for the most common case, in which a function $f(x)$ is to be integrated over an interval. Here is a sample code:

```
def humps_quad ( ):
  from humps import humps_fun
  from scipy.integrate import quad
  result, err = quad ( humps_fun, 0.0, 2.0 )
  print ( '  Integral estimate is ', result )
  print ( '  Error estimate is    ', err )
  return
```

Notice that we are passing the absolute minimum information to `quad()`, and that we get an estimated error for the integral that is returned.

Other functions are available for which the user can request a specific error tolerance, require that a certain Gauss quadrature rule be used, or in other ways modify the procedure by which an integral is estimated.

3 Numerically estimate $\frac{d\,\textbf{humps}(x)}{dx}$

If we have a formula for a function $f(x)$, `scipy` used to offer a function `derivative()` to estimates the value of $f'(x)$ at one or more points x. This function is no longer available. However, we can easily make a simple version ourselves, which uses a central difference estimate, with a stepsize `dx`.

```
def derivative ( f, x, dx ):
  dfdx = ( f(x+dx) - f(x-dx) ) / 2.0 / dx
  return dfdx
```

For our experiment, we want the `humps(x)` function to be the value of $f'(x)$, so we need to start with $f(x)$ equal to the antiderivative of the humps function:

$$f(x) = 10\ \arctan(10\,(x - 0.3)) + 5\ \arctan(5\,(x - 0.9)) - 6\,x$$

Here is a function to do this:

```
def humps_derivative ( ):
  from humps import humps_fun, humps_antideriv
  import matplotlib.pyplot as plt
  import numpy as np

  x1 = np.linspace ( 0.0, 2.0, 11 )
  dx = 1.0E-01
  y1 = derivative ( humps_antideriv, x1, dx )
  x2 = np.linspace ( 0.0, 2.0, 51 )
  y2 = humps_fun ( x2 )

  plt.plot ( x1, y1, 'ro' )
  plt.plot ( x2, y2, 'b-' )
  plt.show ( )
  return
```

4 Solve an ODE whose solution is humps(x)

In an earlier class, we already encountered the function `solve_ivp()` for solving one or several ordinary differential equations. We also just saw a formula for the derivative of the `humps()` function. That means we can pretend we have to solve an ODE of the form:

$$\frac{dy}{dy} = \frac{d\,\text{humps(x)}}{dx}$$

with initial condition $y(0) = humps(0)$, to be solved over the interval $0 \leq x \leq 2$.

Here is how we might use `solve_ivp()` and plot the resulting solution:

```
def humps_ode ( ):
  from humps import humps_fun, humps_ode
  from scipy.integrate import solve_ivp
  import matplotlib.pyplot as plt
  import numpy as np

  xmin = 0.0
  xmax = 2.0
  y0 =  np.array ( [ humps_fun(xmin) ] )
  sol = solve_ivp ( humps_ode, [xmin,xmax], y0 )

  plt.plot ( sol.t, sol.y[0] )
  plt.show ( )
  return
```

The `solve_ivp()` function requires a derivative function which has two arguments. So we have to invoke `humps_ode()`, which has arguments x, y, rather than the simpler `humps_deriv()`. We can plot the individual ODE results versus a plot of the continuous formula. We will see a close match.

5 Solve a BVP whose solution is humps(x)

A boundary value problem (BVP) asks for the computation of a function $y(x)$ over an interval $[a, b]$ for which the values $ya = y(a)$ and $yb = y(b)$ are known, as well as a formula for the second derivative $y”(x)$.

One way to solve this is to define a linear system $Ay = rhs$ which is

```
y[0] = ya
( y[0] - 2 y[1] + y[2] ) / dx^2 = y"(x[1])
```

```
( y[1] - 2 y[2] + y[3] ) / dx^2 = y"(x[2])
...
( y[n-3] - 2 y[n-2] + y[n-1] ) / dx^2 = y"(x[n-2])
y[n-1] = yb
```

and then solve the resulting linear system.

Here is a code to do this:

```
def humps_bvp ( ):
  from humps import humps_fun, humps_deriv2
  import matplotlib.pyplot as plt
  import numpy as np
  xa = 0.0
  xb = 2.0
  n = 41
  x = np.linspace ( xa, xb, n )
  rhs = humps_deriv2 ( x )
  rhs[0] = humps_fun ( xa )
  rhs[n-1] = humps_fun ( xb )

  dx = ( xb - xa ) / ( n - 1 )
  A = dif2_matrix ( n )  # tridiagonal  -1,+2,-1
  A = - A / dx**2        # tridiagonal (+1,-2,+1)/dx^2
  A[0,0] = 1.0
  A[0,1] = 0.0
  A[n-1,n-2] = 0.0
  A[n-1,n-1] = 1.0

  y = np.linalg.solve ( A, rhs )

  plt.plot ( x, y, 'ro' )
  x2 = np.linspace ( xa, xb, 101 )
  y2 = humps_fun ( x2 )
  plt.plot ( x,  y, 'ro-', linewidth = 3, label = 'Computed' )
  plt.show ( )
```

6 Use `scipy` to solve a `humps(x)` BVP

The `scipy` library also offers a function for solving boundary value problems, called `solve_bvp()`. To use it, however, we have to reformulate our problem, and supply some extra code.

First we have to rewrite the second order equation as a pair of first order equations, where y_0 represents the unknown, and y_1 the derivative:

$$y_0' = y_1$$
$$y_1' = \text{humps\_deriv2}(x, y)$$

Second, we have to create a function which returns the new form of the right hand side:

```
def humps_bvp_rhs ( x, y ):

  import numpy as np

  u1 = - 2.0 * ( x - 0.3 )
  v1 = ( ( x - 0.3 )**2 + 0.01 )**2
  u2 = - 2.0 * ( x - 0.9 )
  v2 = ( ( x - 0.9 )**2 + 0.04 )**2

  u1p = - 2.0
```

```
  v1p = 2.0 * ( ( x - 0.3 )**2 + 0.01 ) * 2.0 * ( x - 0.3 )
  u2p = - 2.0
  v2p = 2.0 * ( ( x - 0.9 )**2 + 0.04 ) * 2.0 * ( x - 0.9 )

  n = len ( x )

  rhs = np.zeros ( [ 2, n ] )
  rhs[0] = y[1]
  rhs[1] = ( u1p * v1 - u1 * v1p ) / v1**2 \
         + ( u2p * v2 - u2 * v2p ) / v2**2

  return rhs
```

Third, we have to create a function that evaluates the boundary conditions, that is, it simply reports the error in the requirements on (the first component of) y at the left and right endpoints.

```
def humps_bc ( ya, yb ):
  from humps import humps_fun
  import numpy as np
  xa = 0.0
  xb = 2.0
  resid = np.array ( [ ya[0] - humps_fun(xa), yb[0] - humps_fun(xb) ] )
  return resid
```

Here is how the main program looks now:

```
def humps_solve_bvp ( ):

  from humps import humps_fun
  from scipy.integrate import solve_bvp
  import matplotlib.pyplot as plt
  import numpy as np

  xa = 0.0
  xb = 2.0
  n = 21
  x = np.linspace ( xa, xb, n )  # initial choice for mesh nodes
  m = 2                          # two components, y0 and y1
  y = np.zeros ( [ m, n ] )      # initial guess for y values

  sol = solve_bvp ( humps_bvp_rhs, humps_bc, x, y )

  x = sol.x
  y = sol.y[0]

  plt.plot ( x, y )
  plt.show ( )
  return
```