12.1.OMIMS-NumOpt

December 12, 2019

1 Numerical Optimization in Python

The **scipy.optimize** package provides several commonly used optimization algorithms. This module contains the following aspects

- Unconstrained and constrained minimization of multivariate scalar functions (minimize()) using a variety of algorithms (e.g. BFGS, Nelder-Mead simplex, Newton Conjugate Gradient, COBYLA or SLSQP)
- Global (brute-force) optimization routines (e.g., anneal(), basinhopping())
- Least-squares minimization (leastsq()) and curve fitting (curve_fit()) algorithms
- Scalar univariate functions minimizers (minimize_scalar()) and root finders (newton())
- Multivariate equation system solvers (root()) using a variety of algorithms (e.g. hybrid Powell, Levenberg-Marquardt or large-scale methods such as Newton-Krylov)

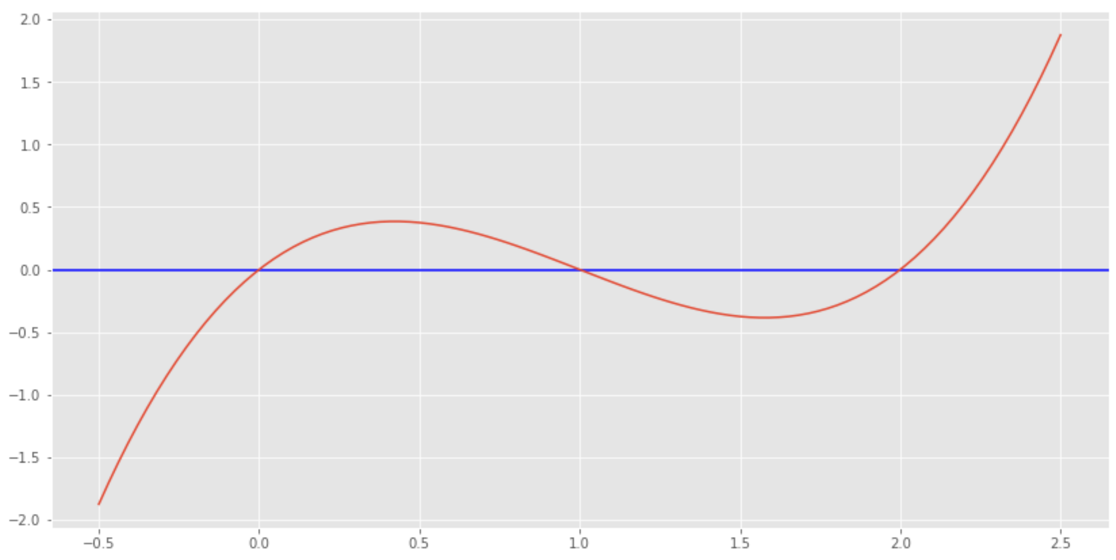
```
In [1]: import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        %matplotlib inline
        %precision 4
        plt.style.use('ggplot')

        import scipy.linalg as la
```

1.1 Univariate roots

We would like to determine the zeroes of a function $f(x)$, i.e. the values x such that $f(x) = 0$.
In the example below, we consider the function given by :

$$f(x) = x * (x - 1) * (x - 2)$$

```
In [2]: def f(x):
            return x*(x-1)*(x-2)

        x = np.linspace(-0.5,2.5,100)
        plt.figure(figsize=(14, 7))
        plt.axhline(0,color ='b')
        plt.plot(x, f(x));
```



scipy.optimize.brentq

Uses the classic Brent (1973) method to find a zero of the function f on the sign changing interval [a , b] (concretely the sign of $f(a)$ must be different from $f(b)$). Generally considered the best of the rootfinding routines here. It is a safe version of the secant method that uses inverse quadratic extrapolation

```
In [3]: from scipy.optimize import brentq

        #first zero
        zero1 = brentq(f, -0.5, 0.5)

        #second zero
        zero2 = brentq(f, 0.5, 1.5)

        #third zero
        zero3 = brentq(f, 1.5,2.5)


        print('First zero : {:3.2f}' .format(zero1))
        print('Second zero : {:3.2f}' .format(zero2))
        print('Third zero : {:3.2f}' .format(zero3))
```

```
First zero : -0.00
Second zero : 1.00
Third zero : 2.00
```

scipy.optimize.newton

Find a zero of a real or complex function using the Newton-Raphson method :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

```
In [4]: from scipy.optimize import newton

        #first zero
        zero1 = newton(f, -0.5)

        #second zero
        zero2 = newton(f, 0.5)

        #third zero
        zero3 = newton(f,1.5)


        print('First zero : {:3.2f}' .format(zero1))
        print('Second zero : {:3.2f}' .format(zero2))
        print('Third zero : {:3.2f}' .format(zero3))
```

```
First zero : -0.00
Second zero : 2.00
Third zero : -0.00
```

Second attempt…

```
In [5]: #first zero
        zero1 = newton(f, -0.5)

        #second zero
        zero2 = newton(f, 0.8)

        #third zero
        zero3 = newton(f,2.5)


        print('First zero : {:3.2f}' .format(zero1))
        print('Second zero : {:3.2f}' .format(zero2))
        print('Third zero : {:3.2f}' .format(zero3))
```

```
First zero : -0.00
Second zero : 1.00
Third zero : 2.00
```

1.2 Mutlivariate roots

We would to determine the zeroes (roots) of a multivatiate function $f : \mathbb{R}^2 \to \mathbb{R}^2$ given by :

$$f(x, y) = ((y - 3x(x+1)(x-1), 0.25x^2 + y^2 - 1)$$

```
In [6]: from scipy.optimize import root

        def f(x):
            return [x[1] - 3*x[0]*(x[0]+1)*(x[0]-1),
                .25*x[0]**2 + x[1]**2 - 1]


        # initial guesss = (0.5, 0.5)
        sol = root(f, (0.5, 0.5))
        sol

Out[6]:     fjac: array([[-0.9978,  0.0659],
               [-0.0659, -0.9978]])
             fun: array([ -1.6360e-12,   1.6187e-12])
         message: 'The solution converged.'
            nfev: 21
             qtf: array([ -1.4947e-08,   1.2702e-08])
               r: array([ 8.2295, -0.8826, -1.7265])
          status: 1
         success: True
               x: array([ 1.1169,  0.8295])
```

The solution represented as a OptimizeResult object. Important attributes are: x the solution array, success a Boolean flag indicating if the algorithm exited successfully and message which describes the cause of the termination

```
In [7]: print(sol.success)

        print(sol.message)

        print(sol.x)

        print(f(sol.x))

True
The solution converged.
[ 1.1169  0.8295]
[-1.6360246490876307e-12, 1.6187051699034782e-12]
```

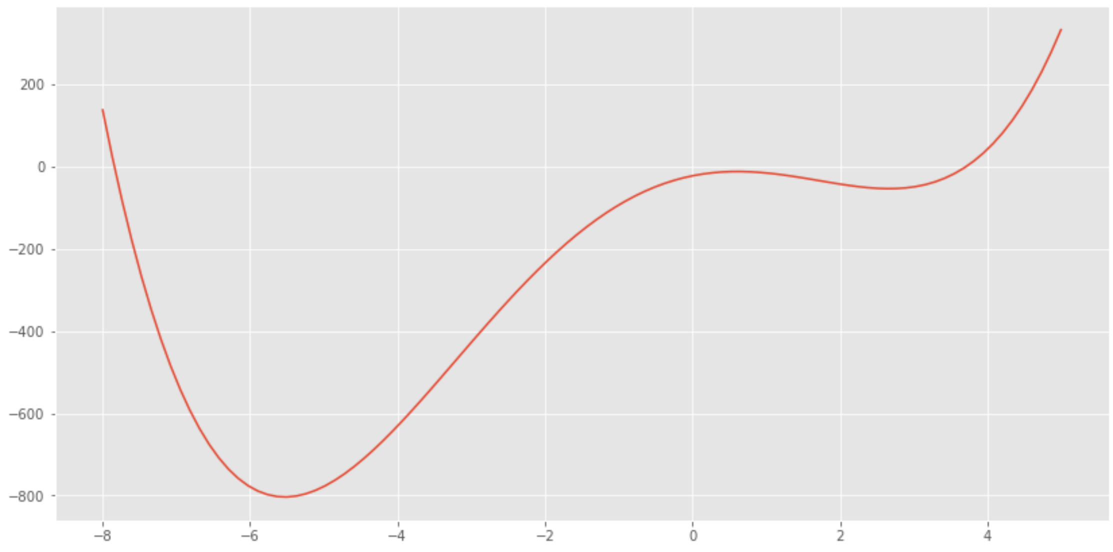
1.3 Optimization with scipy.optimize: univariate case

We would like to minimize the following function :

$$f(x) = x^4 + 3(x-2)^3 - 15x^2 + 1$$

```
In [8]: from scipy import optimize as opt

        def f(x):
```

```
        return x**4 + 3*(x-2)**3 - 15*(x)**2 + 1

    x = np.linspace(-8, 5, 100)
    plt.figure(figsize=(14, 7))
    plt.plot(x, f(x));
```

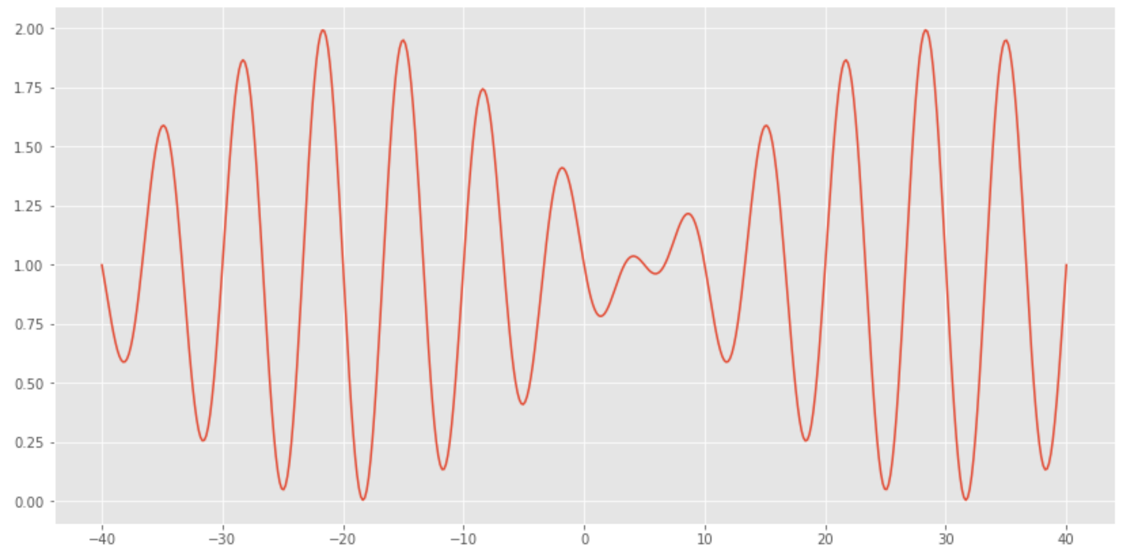


minimize_scalar : minimization of scalar function of one variable

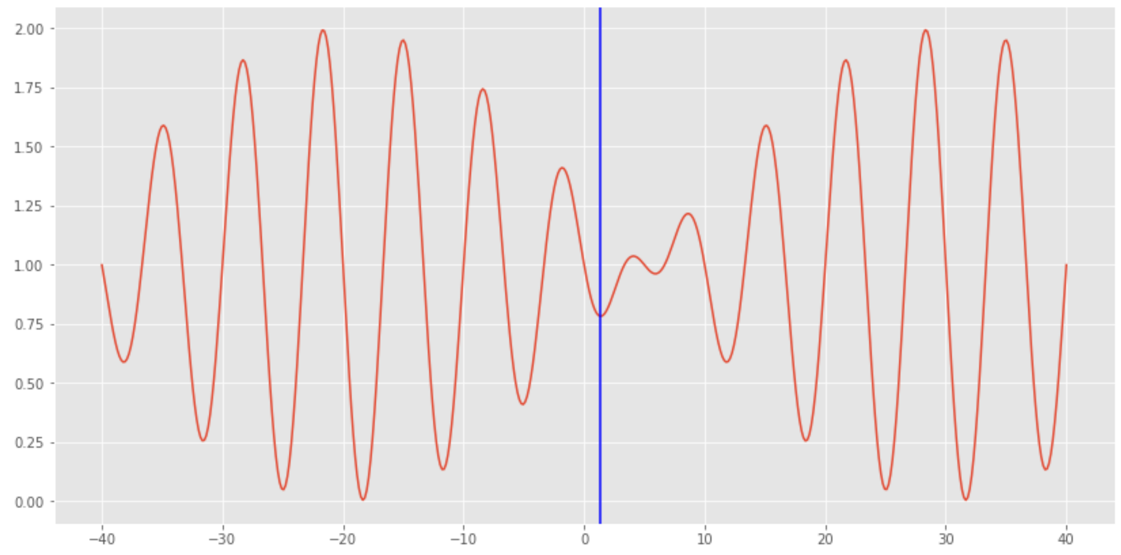
```
In [9]: opt.minimize_scalar(f, method='Brent')

Out[9]:      fun: -803.39553088258845
            nfev: 17
             nit: 11
         success: True
               x: -5.5288011252196627
```

Let's work now on another example with many local optima.

```
In [10]: def f(x):
             return (1-np.cos(2*np.pi*0.01*(x-30))*np.sin(2*np.pi*0.15*(x-30)))


         x = np.linspace(-40,40, 500)
         plt.figure(figsize=(14, 7))
         plt.plot(x, f(x));
```



```
In [11]: # note how additional function arguments are passed in
         sol = opt.minimize_scalar(f)
         sol

Out[11]:      fun: 0.7826734796684246
            nfev: 16
             nit: 12
         success: True
               x: 1.369921727991408

In [12]: plt.figure(figsize=(14, 7))
         plt.plot(x, f(x))
         plt.axvline(sol.x, color = 'b')

Out[12]: <matplotlib.lines.Line2D at 0x24118717e10>
```

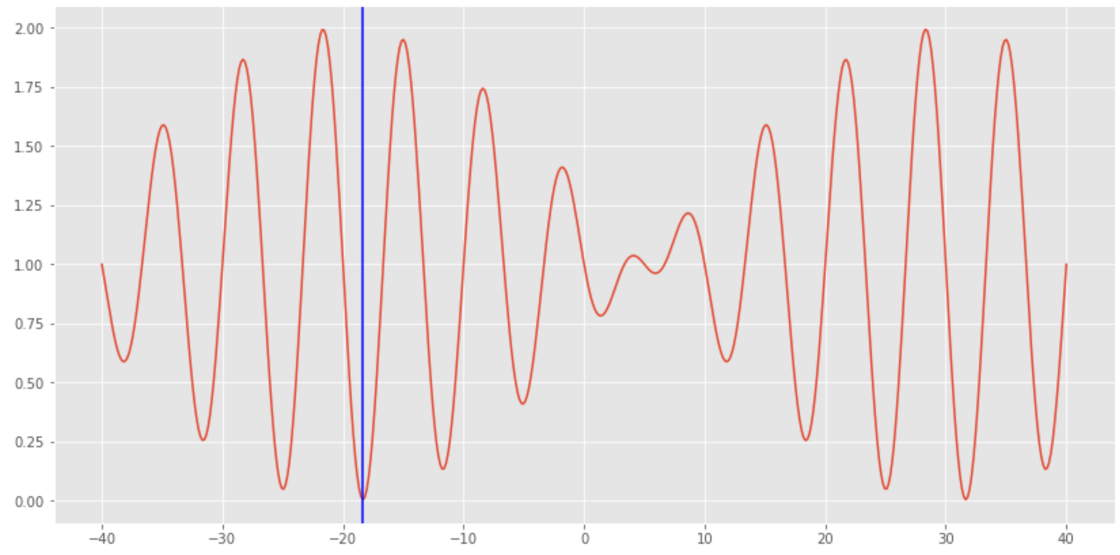


The local minimum given *optimize_scalar* is not a good one ! Let's try to determine the global minimum.

Bracketing methods determine successively smaller intervals (brackets) that contain a root. They generally use the intermediate value theorem, which asserts that if a continuous function has values of opposite signs at the end points of an interval, then the function has at least one root in the interval.

If you use *zip()* with n arguments, then the function will return an iterator that generates tuples of length n. An iterator in PYthon is an object that contains a countable number of values. It can be iterated upon, meaning that you can traverse through all the values.

```
In [13]: #Example : use of the zip function

         numbers = [1, 2, 3]
         letters = ['a', 'b', 'c']
         zipped = zip(numbers, letters)


         list(zipped)

Out[13]: [(1, 'a'), (2, 'b'), (3, 'c')]

In [14]: # we genrate the brackets

         lower = np.random.uniform(-40, 40, 100)
         upper = lower + 2

         # call to minimze_scalar on the intervals provided by the zip function
         sols = [opt.minimize_scalar(f, bracket=(l, u)) for (l, u) in zip(lower, up

         # we determine the best solution given by the algorithm
         # note that sol is a sols object and sol.fun just return the value of the
         idx = np.argmin([sol.fun for sol in sols])

         sol = sols[idx]
         # plot of the optimal solution
         plt.figure(figsize=(14, 7))
         plt.plot(x, f(x))
         plt.axvline(sol.x, color = 'b');
```



1.4 Minimzing a Multivariate Function

Rosenbrock function :

$$f(x,y) = 100(y-x^2)^2 + (1-x)^2$$

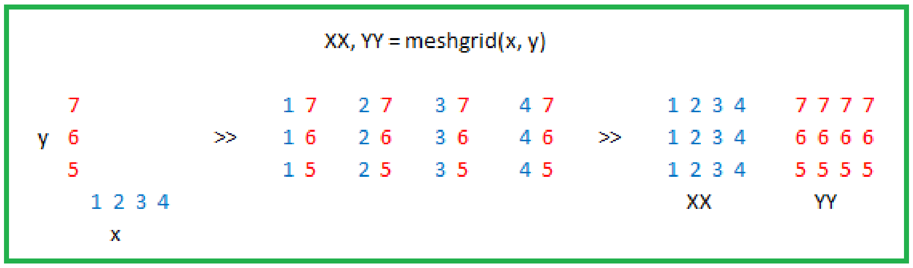
The function has a global minimum at (1,1).

```
In [15]: def rosen(x):
             """Generalized n-dimensional version of the Rosenbrock function"""
             return sum(100*(x[1:]-x[:-1]**2.0)**2.0 +(1-x[:-1])**2.0)
```

1.4.1 Conditionning of optimization problem

The problem is ill-conditioned. One important concept in optimization is the condition number of the curvature (Hessian) of the objective function. When the condition number is high, the gradient may not point in the direction of the minimum, and simple gradient descent methods may be inefficient since they may be forced to take many sharp turns.

SymPy is a Python library for symbolic mathematics. It can be used to compute, for example, the hessian of a function.



A test image

```
In [16]: from sympy import symbols, hessian, Function, N


         # define the variables for sympy
         x, y = symbols('x y')

         f = symbols('f', cls=Function)
         f = 100*(y - x**2)**2 + (1 - x)**2

         # hessian evaluated at (1,1)
         H = hessian(f, [x, y]).subs([(x,1), (y,1)])
         print(np.array(H), "\n")


         # N for Numerical Evaluation
         print(N(H.condition_number()))
```

```
[[802 -400]
 [-400 200]]

2508.00960127744
```

The **meshgrid** function is very usefull to generate a grid at which an objective function will be evaluated. The illustration below shows how it works :

A small example to illustrate how to generate the points of a mesh grid.

```
In [17]: # np.arrange(1,5,1) = [1,2,3,4]
         # np.arange(5, 8, 1) = [5,6,7]

         xx_, yy_ = np.meshgrid(np.arange(1, 5, 1),
          np.arange(5, 8, 1))

         print("xx_ :\n", xx_, "\n")

         print("yy_ :\n", yy_, "\n")

         print("xx_.ravel() :", xx_.ravel(), "\n")
```

```
        print("yy_.ravel() :", yy_.ravel(), "\n")

        print("np.c_ :\n", np.c_[xx_.ravel(), yy_.ravel()], "\n")

        print("np.vstack :\n", np.vstack([xx_.ravel(), yy_.ravel()]), "\n")
```

```
xx_ :
 [[1 2 3 4]
 [1 2 3 4]
 [1 2 3 4]]

yy_ :
 [[5 5 5 5]
 [6 6 6 6]
 [7 7 7 7]]

xx_.ravel() : [1 2 3 4 1 2 3 4 1 2 3 4]

yy_.ravel() : [5 5 5 5 6 6 6 6 7 7 7 7]

np.c_ :
 [[1 5]
 [2 5]
 [3 5]
 [4 5]
 [1 6]
 [2 6]
 [3 6]
 [4 6]
 [1 7]
 [2 7]
 [3 7]
 [4 7]]

np.vstack :
 [[1 2 3 4 1 2 3 4 1 2 3 4]
 [5 5 5 5 6 6 6 6 7 7 7 7]]
```

Then *np.vstack* can be used as an input for the rosenbrock function

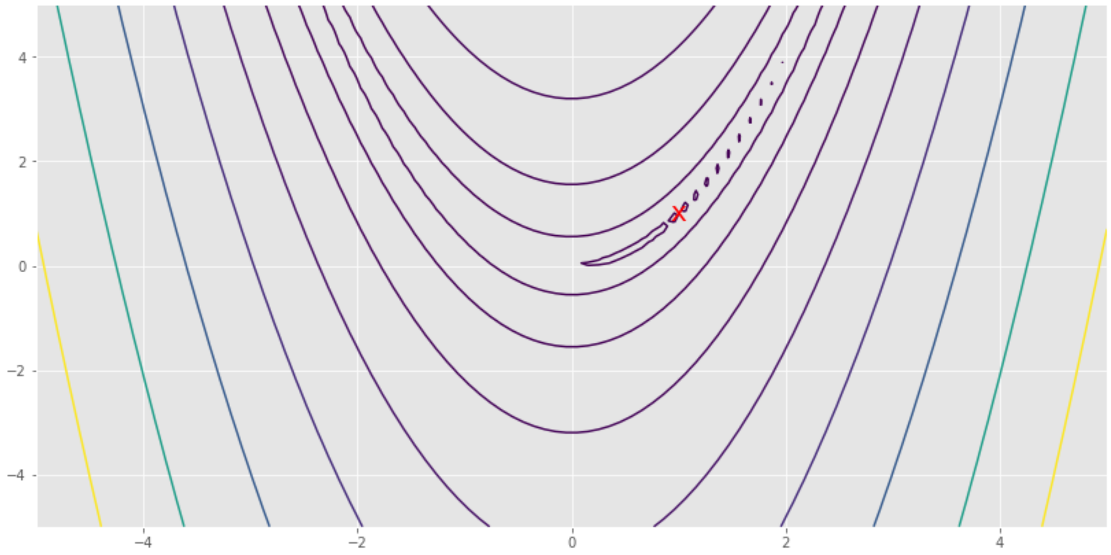
```
In [18]: x = np.linspace(-5, 5, 100)
         y = np.linspace(-5, 5, 100)
         X, Y = np.meshgrid(x, y)
         np.vstack([X.ravel(), Y.ravel()])
         Z = rosen(np.vstack([X.ravel(), Y.ravel()]))
```

```
        Z = Z.reshape((100,100))
```

matplotlib.pyplot.contour
contour([X, Y,] Z, [levels])

```
In [19]: levels = np.arange(10)**5
```

```
In [20]: # Note: the global minimum is at (1,1) in a tiny contour island
         plt.figure(figsize=(14, 7))
         plt.contour(X, Y, Z, levels)
         plt.text(1, 1, 'x', va='center', ha='center', color='red', fontsize=20);
```



1.5 Gradient Descent

```
In [21]: #Indexing

         x = np.arange(5)

         print("x:\n", x, "\n")
         print("x starting from the 2nd elt and without the last elt:\n", x[1:-1],
         print("x without the last two elts:\n", x[:-2], "\n")
         print("x starting from the 3rd:\n", x[2:], "\n")
```

```
x:
 [0 1 2 3 4]

x starting from the 2nd elt and without the last elt:
 [1 2 3]
```

```
x without the last two elts:
 [0 1 2]

x starting from the 3rd:
 [2 3 4]
```

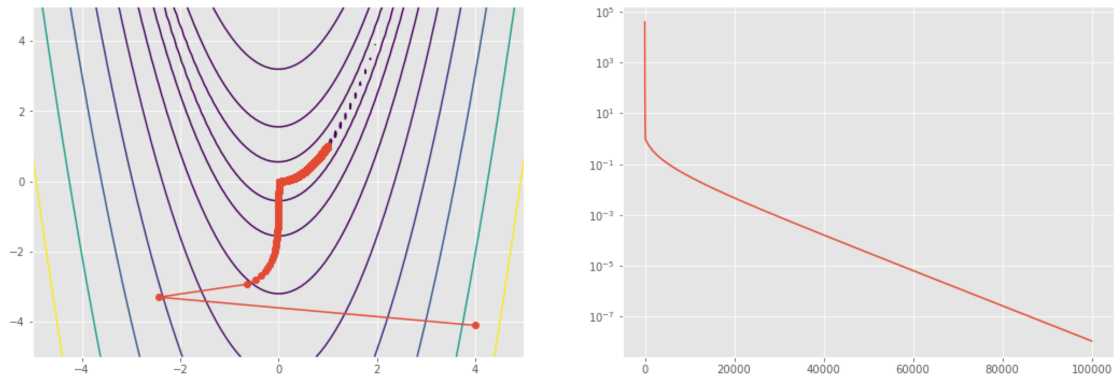
```
In [22]: def rosen_der(x):
             """Derivative of generalized Rosen function."""
             xm = x[1:-1]
             xm_m1 = x[:-2]
             xm_p1 = x[2:]
             # numpy.zeros_like(a, dtype=None, order='K', subok=True, shape=None)[s
             # Return an array of zeros with the same shape and type as a given arr
             der = np.zeros_like(x)
             der[1:-1] = 200*(xm-xm_m1**2) - 400*(xm_p1 - xm**2)*xm - 2*(1-xm)
             der[0] = -400*x[0]*(x[1]-x[0]**2) - 2*(1-x[0])
             der[-1] = 200*(x[-1]-x[-2]**2)
             return der
```

```
In [23]: def custmin(fun, x0, args=(), maxfev=None, alpha=0.0002,
             maxiter=100000, tol=1e-10, callback=None, **options):
             """Implements simple gradient descent for the Rosen function."""
             bestx = x0
             besty = fun(x0)
             funcalls = 1
             niter = 0
             improved = True
             stop = False

             while improved and not stop and niter < maxiter:
                 niter += 1
                 # the next 2 lines are gradient descent
                 step = alpha * rosen_der(bestx)
                 bestx = bestx - step
                 besty = fun(bestx)
                 funcalls += 1

                 if la.norm(step) < tol:
                     improved = False
                 if callback is not None:
                     callback(bestx)
                 if maxfev is not None and funcalls >= maxfev:
                     stop = True
                     break

             return opt.OptimizeResult(fun=besty, x=bestx, nit=niter, nfev=funcalls
In [24]: def reporter(p):
             """Reporter function to capture intermediate states of optimization.""
             global ps
             ps.append(p)
```

```
In [25]: # Initial starting position
         x0 = np.array([4,-4.1])

         ps = [x0]
         opt.minimize(rosen, x0, method=custmin, callback=reporter)
```

```
Out[25]:      fun: 1.0604663473448339e-08
             nfev: 100001
              nit: 100000
          success: True
                x: array([ 0.9999,  0.9998])
```

```
In [26]: ps = np.array(ps)
         plt.figure(figsize=(18,6))
         plt.subplot(121)
         plt.contour(X, Y, Z, np.arange(10)**5)
         plt.plot(ps[:, 0], ps[:, 1], '-o')
         plt.subplot(122)
         plt.semilogy(range(len(ps)), rosen(ps.T));
```



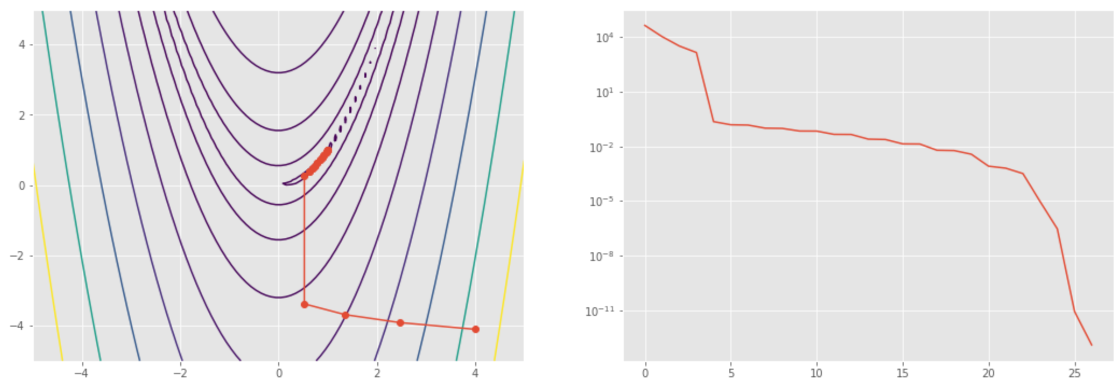
1.6 Newton - Conjuguate Gradient Algorithm

Newton-Conjugate Gradient algorithm is a modified Newton's method and uses a conjugate gradient algorithm to (approximately) invert the local Hessian. Newton's method is based on fitting the function locally to a quadratic form where the matrix is provided is the hessian of the objective function. If the Hessian is positive definite then the local minimum of this function can be found by setting the gradient of the quadratic form to zero. Then we use the CG method to determine the solution of this approximation.

```
In [27]: from scipy.optimize import rosen, rosen_der, rosen_hess

         ps = [x0]
         opt.minimize(rosen, x0, method='Newton-CG', jac=rosen_der, hess=rosen_hess
```

```
Out[27]:      fun: 1.3642782750354208e-13
              jac: array([  1.2120e-04,  -6.0850e-05])
          message: 'Optimization terminated successfully.'
             nfev: 38
             nhev: 26
              nit: 26
             njev: 63
           status: 0
          success: True
                x: array([ 1.,  1.])
```

```
In [28]: ps = np.array(ps)
         plt.figure(figsize=(18,6))
         plt.subplot(121)
         plt.contour(X, Y, Z, np.arange(10)**5)
         plt.plot(ps[:, 0], ps[:, 1], '-o')
         plt.subplot(122)
         plt.semilogy(range(len(ps)), rosen(ps.T));
```



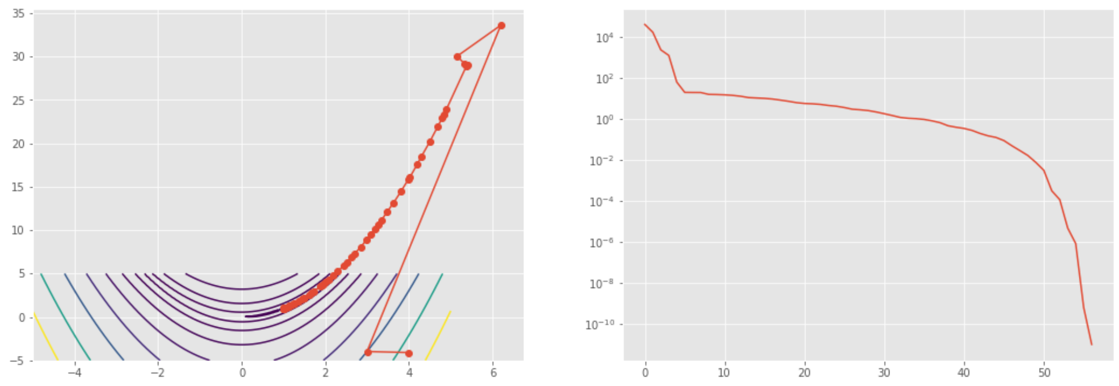
1.7 BFGS

```
In [29]: ps = [x0]
         opt.minimize(rosen, x0, method='BFGS', callback=reporter)
```

```
Out[29]:        fun: 9.48988612333806e-12
          hess_inv: array([[ 0.5    ,  0.9999],
                 [ 0.9999,  2.0047]])
               jac: array([  4.3925e-05,  -2.0365e-05])
           message: 'Desired error not necessarily achieved due to precision loss.
```

```
       nfev: 540
        nit: 56
       njev: 132
     status: 2
    success: False
          x: array([ 1.,  1.])
```

```
In [30]: ps = np.array(ps)
         plt.figure(figsize=(18,6))
         plt.subplot(121)
         plt.contour(X, Y, Z, np.arange(10)**5)
         plt.plot(ps[:, 0], ps[:, 1], '-o')
         plt.subplot(122)
         plt.semilogy(range(len(ps)), rosen(ps.T));
```

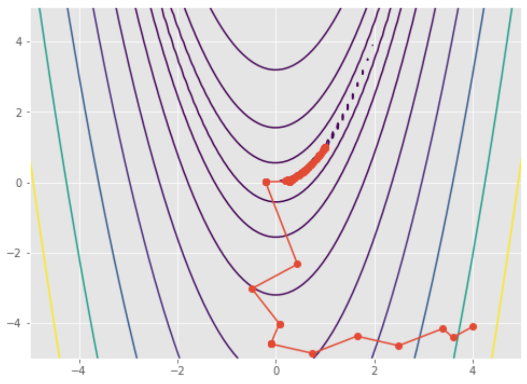


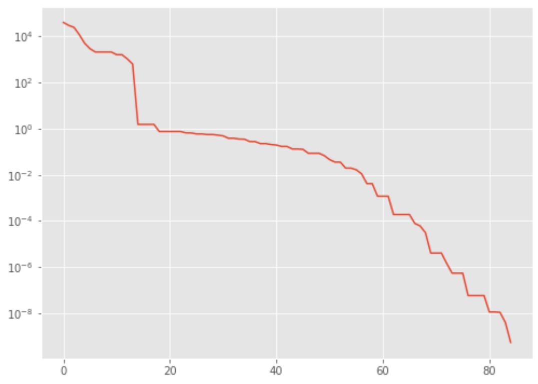
1.8 Nelder-Mead

```
In [31]: ps = [x0]
         opt.minimize(rosen, x0, method='nelder-mead', callback=reporter)

Out[31]:  final_simplex: (array([[ 1.    ,  1.    ],
                [ 0.9999,  0.9999],
                [ 1.    ,  1.    ]]), array([  5.2628e-10,   3.8753e-09,   1.0609e-
                    fun: 5.262756878429089e-10
                message: 'Optimization terminated successfully.'
                   nfev: 162
                    nit: 85
                 status: 0
                success: True
                      x: array([ 1.,  1.])
```

```
In [32]: ps = np.array(ps)
         plt.figure(figsize=(18,6))
         plt.subplot(121)
```

```
        plt.contour(X, Y, Z, np.arange(10)**5)
        plt.plot(ps[:, 0], ps[:, 1], '-o')
        plt.subplot(122)
        plt.semilogy(range(len(ps)), rosen(ps.T));
```





1.9 Linear Programming

Linear programming solver is based on the following function in Scipy :

scipy.optimize.linprog(c, A_ub=None, b_ub=None, A_eq=None, b_eq=None, bounds=None, method='simplex', callback=None, options=None)

Example :

$ min   f(x_0, x_1) = -x_0 + 4x_1$

s.t. : -$3x_0 + x_1 \leq 6$

$x_0 + 2x_1 \leq 4$

$x_1 \geq$ -3

$-\infty \leq x_0 \leq \infty$

```
In [33]: c = [-1, 4]
         A = [[-3, 1], [1, 2]]
         b = [6, 4]
         x0_bounds = (None, None)
         x1_bounds = (-3, None)
         from scipy.optimize import linprog
         res = linprog(c, A_ub=A, b_ub=b, bounds=(x0_bounds, x1_bounds), options={'
         print(res.keys(),'\n')
         print('x :', res.x, '\n')
         print('value :', res.fun, '\n')
```

```
Optimization terminated successfully.
         Current function value: -22.000000
         Iterations: 1
dict_keys(['x', 'fun', 'nit', 'status', 'slack', 'message', 'success'])

x : [ 10.  -3.]
```

```
value : -22.0
```

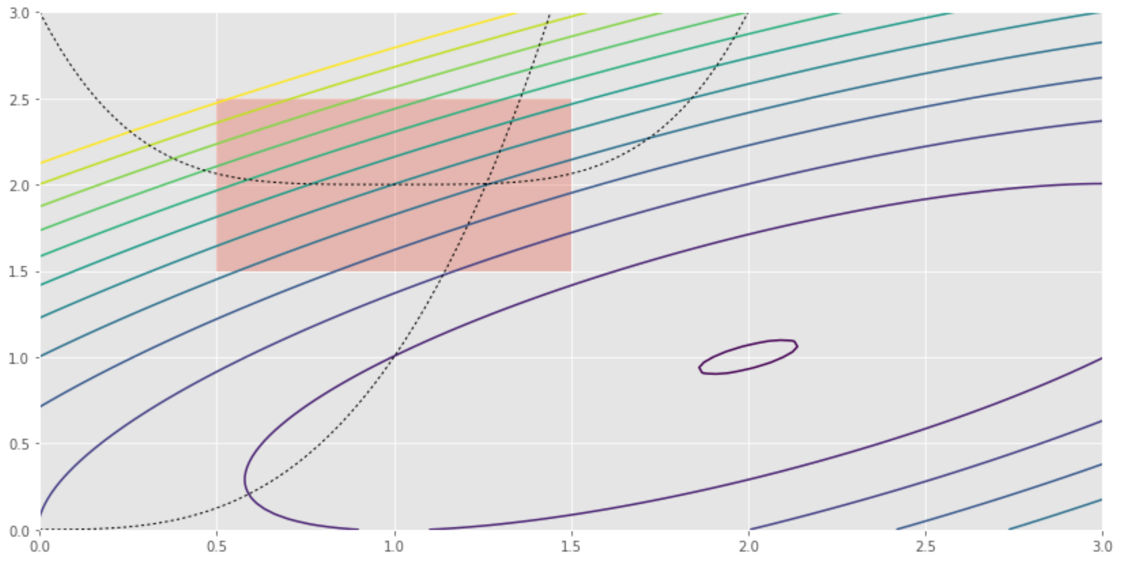
1.10 Nonlinear Constrained Optimization

$$\begin{aligned} &min\ \ f(x,y) = \text{-}(2xy + 2x - x^2 - 2y^2) \\ &\quad s.t.: \quad x^3 - y = 0 \\ &\qquad\quad y - (x-1)^4 - 2 \geq 0 \\ &\qquad\quad 0.5 \leq x \leq 1.5 \\ &\qquad\quad 1.5 \leq y \leq 2.5 \end{aligned}$$

```
In [34]: # objective function

         def f(x):
             return -(2*x[0]*x[1] + 2*x[0] - x[0]**2 - 2*x[1]**2)
```

```
In [35]: # contour lines and constraints

         x = np.linspace(0, 3, 100)
         y = np.linspace(0, 3, 100)
         X, Y = np.meshgrid(x, y)
         Z = f(np.vstack([X.ravel(), Y.ravel()])).reshape((100,100))
         plt.figure(figsize=(14, 7))
         plt.contour(X, Y, Z, np.arange(-1.99,10, 1));
         plt.plot(x, x**3, 'k:', linewidth=1)
         plt.plot(x, (x-1)**4+2, 'k:', linewidth=1)
         plt.fill([0.5,0.5,1.5,1.5], [2.5,1.5,1.5,2.5], alpha=0.3)
         plt.axis([0,3,0,3])
```

```
Out[35]: [0, 3, 0, 3]
```



Based on the analysis provided above, we conclude that the optimal solution is given by the intersection of the curve $y = x^3$ and $y = (x - 1)^4 + 2$. Let's check if the solver will be successful.

scipy.optimize.minimize(fun, x0, args=(), method=None, jac=None, hess=None, hessp=None, bounds=None, constraints=(), tol=None, callback=None, options=None)

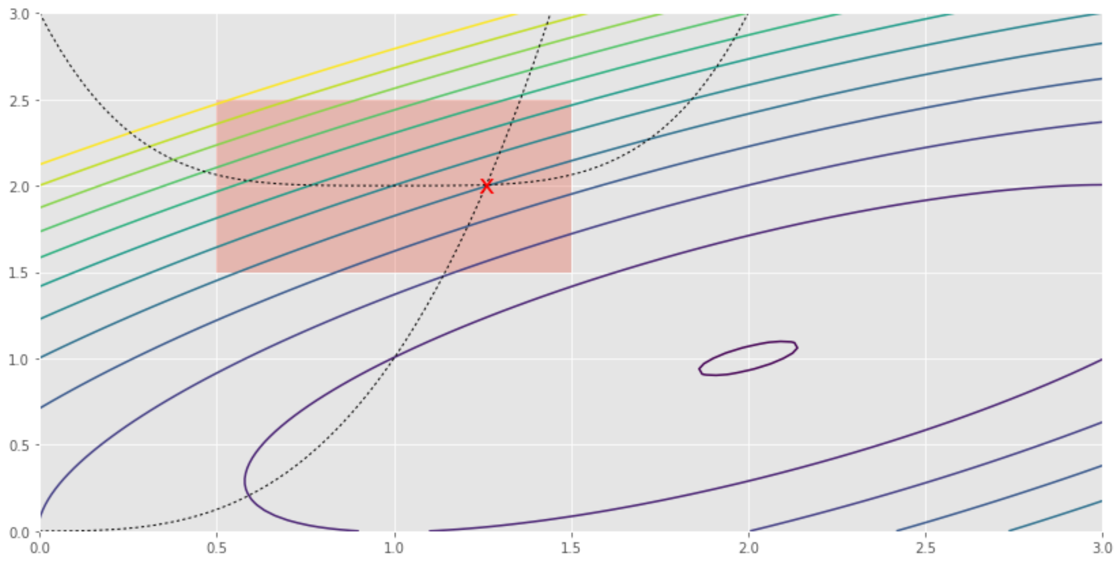
To set constraints, we pass in a dictionary with keys type, fun and jac. The jacobian van be provided for equality type constraints.

```
In [36]: cons = ({'type': 'eq',
         'fun' : lambda x: np.array([x[0]**3 - x[1]]),
         'jac' : lambda x: np.array([3.0*(x[0]**2.0), -1.0])},
         {'type': 'ineq',
         'fun' : lambda x: np.array([x[1] - (x[0]-1)**4 - 2])})
         bnds = ((0.5, 1.5), (1.5, 2.5))
```

Then we call the *minimze* function with an initial guess.

```
In [37]: x0 = [0, 2.5]

         cx = opt.minimize(f, x0, bounds=bnds, constraints=cons)
         cx

Out[37]:      fun: 2.049915472024102
              jac: array([-3.4875,  5.4967])
          message: 'Optimization terminated successfully.'
             nfev: 25
              nit: 6
             njev: 6
           status: 0
          success: True
                x: array([ 1.2609,  2.0046])
```

```
In [38]: x = np.linspace(0, 3, 100)
         y = np.linspace(0, 3, 100)
         X, Y = np.meshgrid(x, y)
         Z = f(np.vstack([X.ravel(), Y.ravel()])).reshape((100,100))
         plt.figure(figsize=(14, 7))
         plt.contour(X, Y, Z, np.arange(-1.99,10, 1));
         plt.plot(x, x**3, 'k:', linewidth=1)
         plt.plot(x, (x-1)**4+2, 'k:', linewidth=1)
         plt.text(cx['x'][0], cx['x'][1], 'x', va='center', ha='center', size=20, c
         plt.fill([0.5,0.5,1.5,1.5], [2.5,1.5,1.5,2.5], alpha=0.3)
         plt.axis([0,3,0,3]);
```



```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```