BlackBoxOptimization

February 21, 2015

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

```
In [2]: import scipy.linalg as la
```

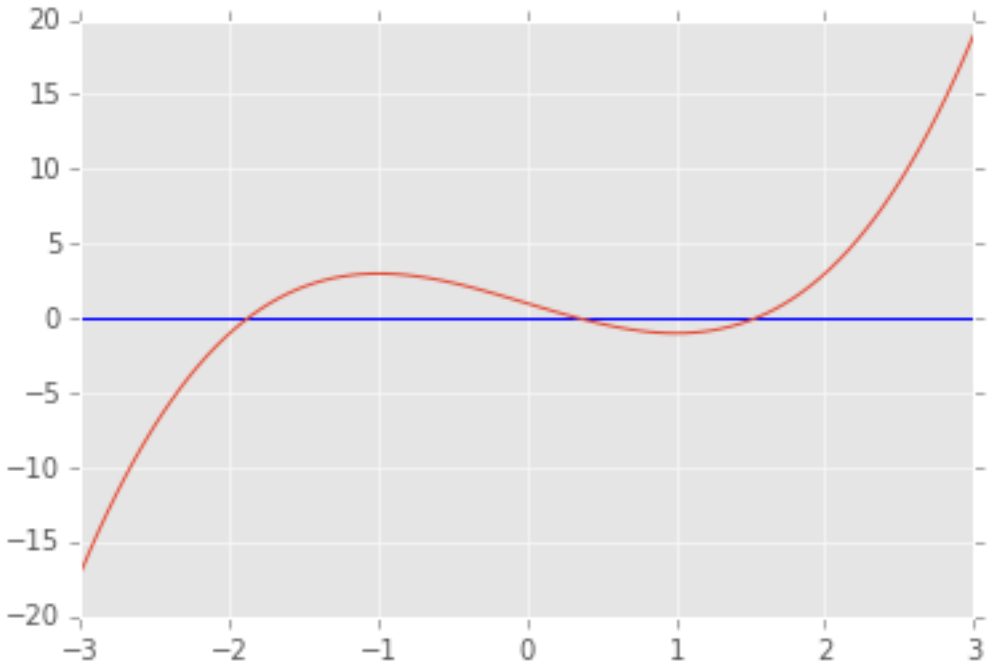
0.1 Finding roots

For root finding, we generally need to proivde a starting point in the vicinitiy of the root. For iD root finding, this is often provided as a bracket (a, b) where a and b have opposite signs.

0.1.1 Univariate roots and fixed points

```
In [3]: def f(x):
            return x**3-3*x+1
```

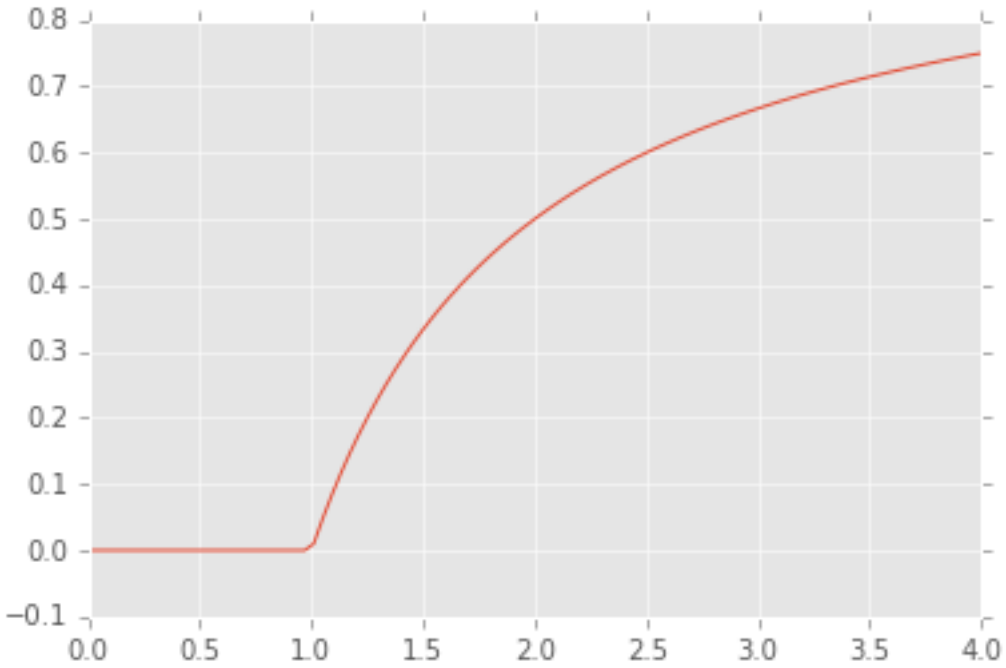
```
In [4]: x = np.linspace(-3,3,100)
        plt.axhline(0)
        plt.plot(x, f(x));
```




```
In [5]: from scipy.optimize import brentq, newton

In [6]: brentq(f, -3, 0), brentq(f, 0, 1), brentq(f, 1,3)

Out[6]: (-1.8794, 0.3473, 1.5321)

In [7]: newton(f, -3), newton(f, 0), newton(f, 3)

Out[7]: (-1.8794, 0.3473, 1.5321)
```

Finding fixed points Finding the fixed points of a function $g(x) = x$ is the same as finding the roots of $g(x) - x$. However, specialized algorihtms also exist - e.g. using `scipy.optimize.fixedpoint`.

```
In [8]: from scipy.optimize import fixed_point

In [9]: def f(x, r):
            """Discrete logistic equation."""
            return r*x*(1-x)

In [10]: n = 100
         fps = np.zeros(n)
         for i, r in enumerate(np.linspace(0, 4, n)):
             fps[i] = fixed_point(f, 0.5, args=(r, ))

In [11]: plt.plot(np.linspace(0, 4, n), fps);
```



0.1.2 Mutlivariate roots and fixed points

```
In [12]: from scipy.optimize import root, fsolve

In [13]: def f(x):
             return [x[1] - 3*x[0]*(x[0]+1)*(x[0]-1),
                     .25*x[0]**2 + x[1]**2 - 1]

In [14]: sol = root(f, (0.5, 0.5))
         sol

Out[14]:   status: 1
          success: True
              qtf: array([ -1.4947e-08,   1.2702e-08])
             nfev: 21
                r: array([ 8.2295, -0.8826, -1.7265])
              fun: array([ -1.6360e-12,   1.6187e-12])
                x: array([ 1.1169,  0.8295])
          message: 'The solution converged.'
             fjac: array([[-0.9978,  0.0659],
                [-0.0659, -0.9978]])

In [15]: f(sol.x)

Out[15]: [-0.0000, 0.0000]

In [16]: sol = root(f, (12,12))
         sol
```

```
Out[16]:   status: 1
          success: True
              qtf: array([ -1.5296e-08,   3.5475e-09])
             nfev: 33
                r: array([-10.9489,   6.1687,  -0.3835])
              fun: array([  4.7062e-13,   1.4342e-10])
                x: array([ 0.778 , -0.9212])
          message: 'The solution converged.'
             fjac: array([[ 0.2205, -0.9754],
                 [ 0.9754,  0.2205]])
```

```
In [17]: f(sol.x)
```

```
Out[17]: [0.0000, 0.0000]
```

0.2 Optimization Primer

We will assume that our optimization problem is to minimize some univariate or multivariate function $f(x)$. This is without loss of generality, since to find the maximum, we can simply minime $-f(x)$. We will also assume that we are dealing with multivariate or real-valued smooth functions - non-smooth, noisy or discrete functions are outside the scope of this course and less common in statistical applications.

To find the minimum of a function, we first need to be able to express the function as a mathemtical expresssion. For example, in lesst squares regression, the function that we are optimizing is of the form $y_i - f(x_i, \theta)$ for some parameter(s) θ. To choose an appropirate optimization algorihtm, we should at least answr these two questions if possible:

1. Is the function convex?
2. Are there any constraints that the solution must meet?

Finally, we need to realize that optimization mehthods are nearly always designed to find local optima. For convex problems, there is only one minimum and so this is not a problem. However, if there are multiple local minima, often heuristics such as multiple random starts must be adopted to find a "good" enouhg solution.

0.2.1 Is the function convex?

Convex functions are very nice becuase they have a single global minimum, and there are very efficient algorithms for solving large convex systems.

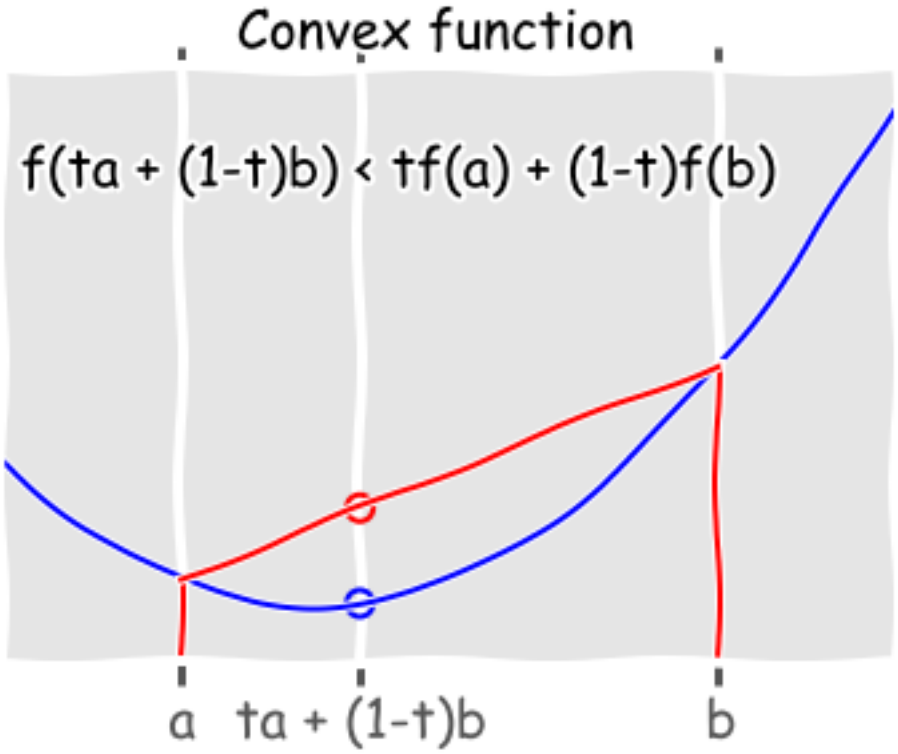
Intuitively, a function is convex if every chord joining two points on the function lies above the function. More formally, a function is convex if

$$f(ta + (1-t)b) < tf(a) + (1-t)f(b)$$

for some t between 0 and 1 - this is shown in the figure below.

```
In [18]: def f(x):
             return (x-4)**2 + x + 1

         with plt.xkcd():
             x = np.linspace(0, 10, 100)

             plt.plot(x, f(x))
             ymin, ymax = plt.ylim()
             plt.axvline(2, ymin, f(2)/ymax, c='red')
             plt.axvline(8, ymin, f(8)/ymax, c='red')
             plt.scatter([4, 4], [f(4), f(2) + ((4-2)/(8-2.))*(f(8)-f(2))],
```

```
                edgecolor=['blue', 'red'], facecolor='none', s=100, linewidth=2)
plt.plot([2,8], [f(2), f(8)])
plt.xticks([2,4,8], ('a', 'ta + (1-t)b', 'b'), fontsize=20)
plt.text(0.2, 40, 'f(ta + (1-t)b) < tf(a) + (1-t)f(b)', fontsize=20)
plt.xlim([0,10])
plt.yticks([])
plt.suptitle('Convex function', fontsize=20)
```



Checking if a function is convex using the Hessian The formal definition is only useful foc checking if a function is convex if you can find a counter-example. More practically, a twice differntiable function is convex if its Hessian is positive semi-definite, and strictly convex if the Hessian is positive definite.

For example, suppose we want to minimize the function

$$f(x_1, x_2, x_3) = x_1^2 + 2x_2^2 + 3x_3^2 + 2x_1x_2 + 2x_1x_3$$

Note: A univariate function is convex if its second derivative is positive everywhere.

```
In [19]: from sympy import symbols, hessian, Function, N
         x, y, z = symbols('x y z')
         f = symbols('f', cls=Function)

In [20]: f = x**2 + 2*y**2 + 3*z**2 + 2*x*y + 2*x*z

In [21]: H = np.array(hessian(f, (x, y, z)))
         H
```

```
Out[21]: array([[2, 2, 2],
                [2, 4, 0],
                [2, 0, 6]], dtype=object)
```

```
In [22]: e, v = la.eig(H)
         np.real_if_close(e)
```

```
Out[22]: array([ 0.2412,  7.0642,  4.6946])
```

Since all eigenvalues are positive, the Hessian is positive defintie and the function is convex.

Combining convex functions The following rules may be useful to determine if more complex functions are covex:

1. The intersection of convex functions is convex
2. If the functions f and g are convex and $a \geq 0$ and $b \geq 0$ then the function $af + bg$ is convex.
3. If the function U is convex and the function g is nondecreasing and convex then the function f defined by $f(x) = g(U(x))$ is convex.

Many more technical deetails about convexity and convex optimization can be found in this book.

0.2.2 Are there any constraints that the solution must meet?

In general, optimizaiton without constraints is easier to solve than optimization in the presence of constraints. In any case, the solutions may be very different in the prsence or absence of constraints, so it is important to know if there are any constraints.

We will see some examples of two general strategies - convert a problme with constraints into one without constraints, or use an algorithm that can optimize with constraints.

0.3 Using `scipy.optimize`

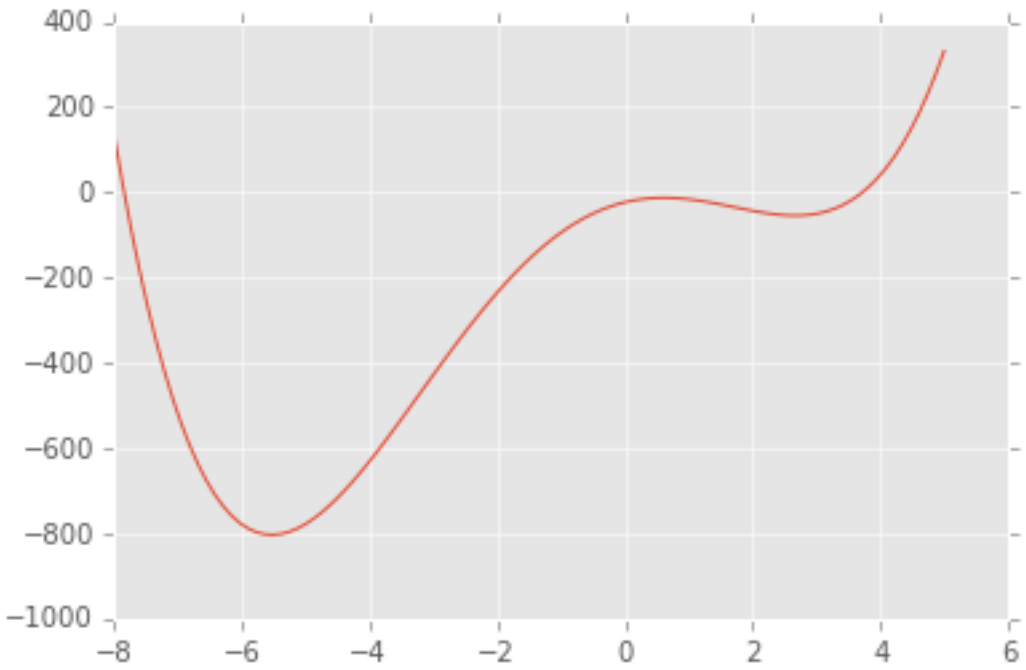
One of the most convenient libraries to use is `scipy.optimize`, since it is already part of the Anaconda interface and it has a fairly intuitive interface.

```
In [23]: from scipy import optimize as opt
```

Minimizing a univariate function $f : \mathbb{R} \to \mathbb{R}$

```
In [24]: def f(x):
             return x**4 + 3*(x-2)**3 - 15*(x)**2 + 1
```

```
In [25]: x = np.linspace(-8, 5, 100)
         plt.plot(x, f(x));
```



The `minimize_scalar` function will find the minimum, and can also be told to search within given bounds. By default, it uses the Brent algorithm, which combines a bracketing strategy with a parabolic approximation.

```
In [26]: opt.minimize_scalar(f, method='Brent')
```

```
Out[26]:   fun: -803.39553088258845
          nfev: 12
           nit: 11
             x: -5.5288011252196627
```

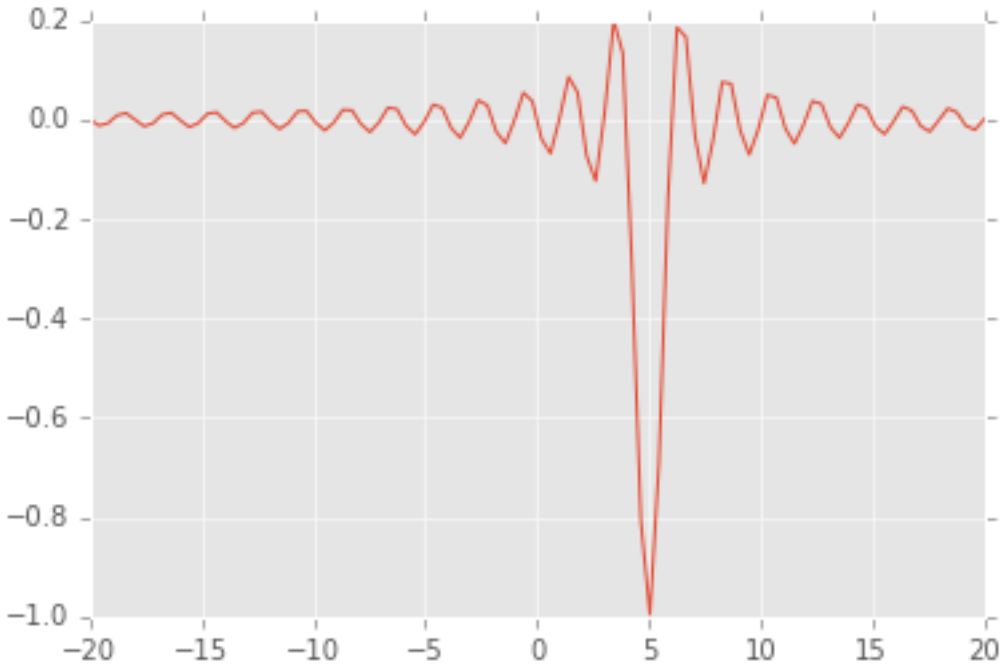
```
In [27]: opt.minimize_scalar(f, method='bounded', bounds=[0, 6])
```

```
Out[27]:   status: 0
             nfev: 12
          success: True
              fun: -54.210039377127622
                x: 2.6688651040396532
          message: 'Solution found.'
```

0.3.1 Local and global minima

```
In [28]: def f(x, offset):
             return -np.sinc(x-offset)
```

```
In [29]: x = np.linspace(-20, 20, 100)
         plt.plot(x, f(x, 5));
```

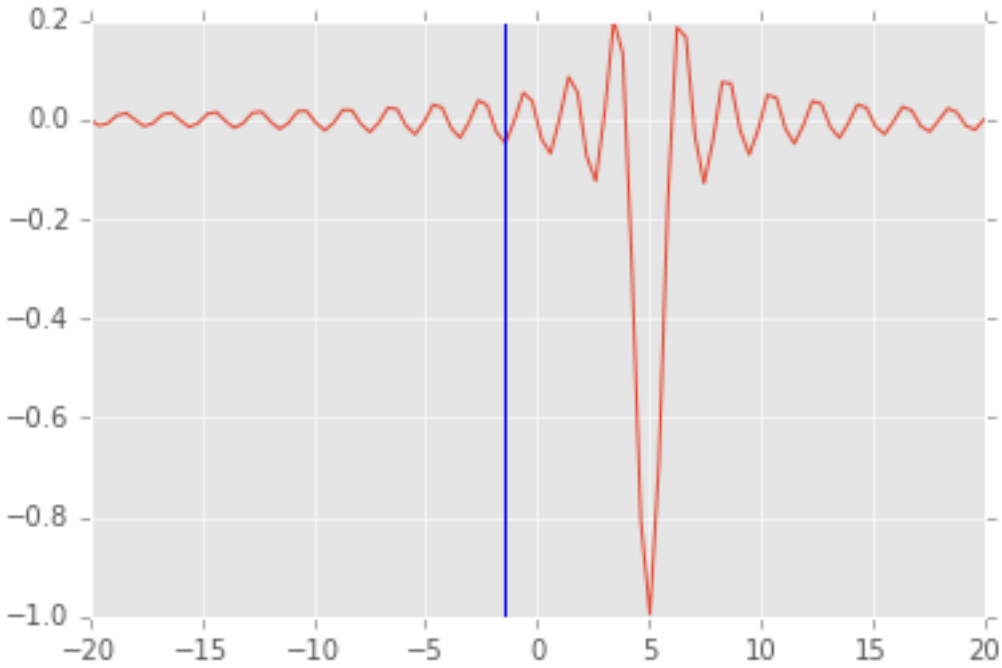


In [30]:
```
# note how additional function arguments are passed in
sol = opt.minimize_scalar(f, args=(5,))
sol
```

Out[30]:
```
  fun: -0.049029624014074166
 nfev: 11
  nit: 10
    x: -1.4843871263953001
```

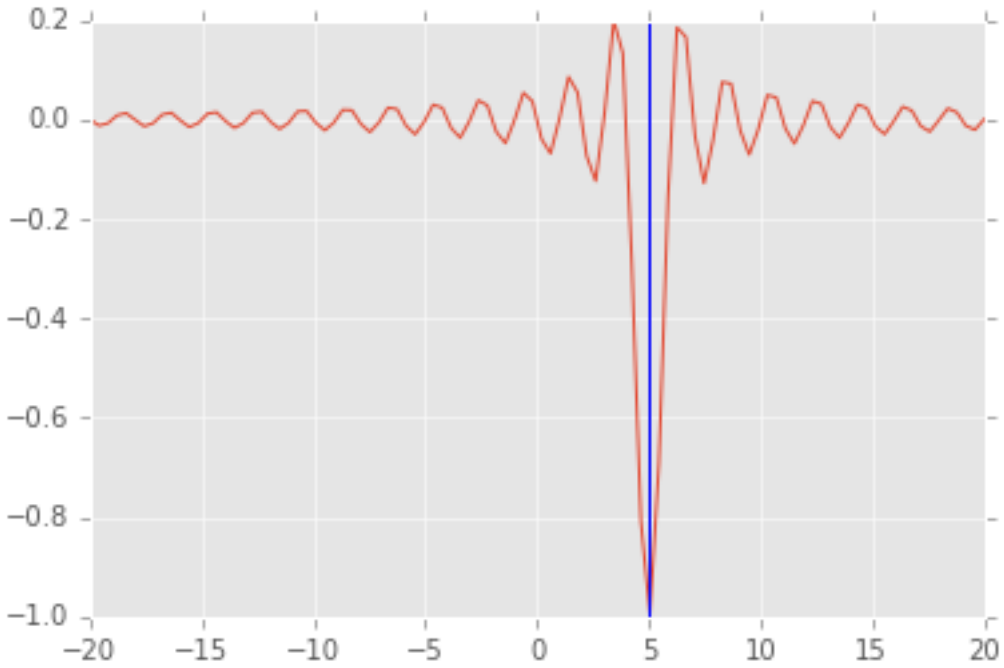
In [31]:
```
plt.plot(x, f(x, 5))
plt.axvline(sol.x)
```

Out[31]:
```
<matplotlib.lines.Line2D at 0x115211c90>
```



We can try multiple ranodm starts to find the global minimum

```
In [32]: lower = np.random.uniform(-20, 20, 100)
         upper = lower + 1
         sols = [opt.minimize_scalar(f, args=(5,), bracket=(l, u)) for (l, u) in zip(lower, upper)]

In [33]: idx = np.argmin([sol.fun for sol in sols])
         sol = sols[idx]

In [34]: plt.plot(x, f(x, 5))
         plt.axvline(sol.x);
```

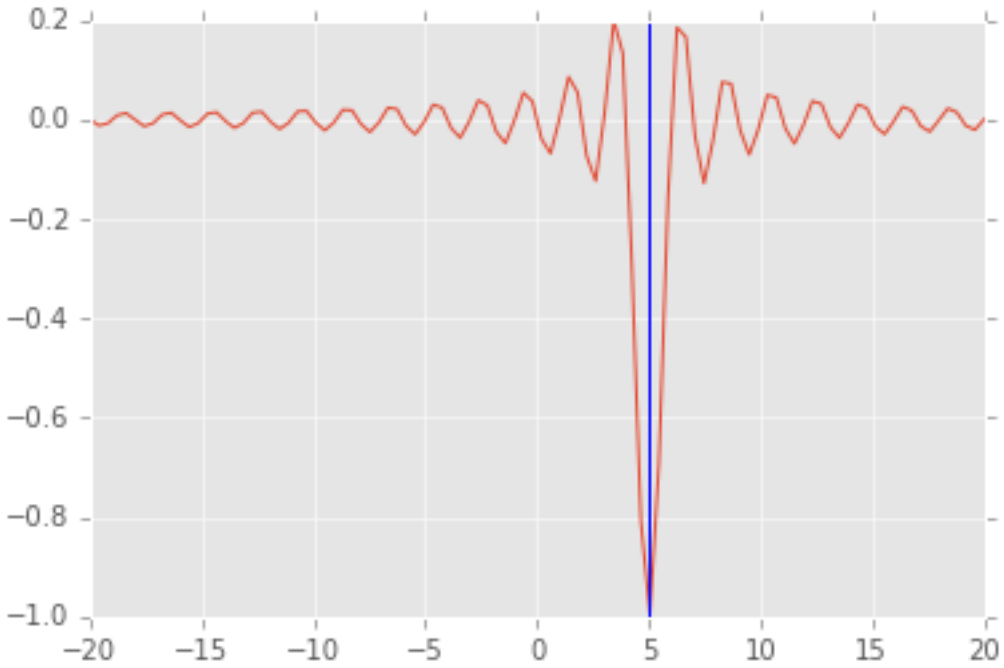


Using a stochastic algorithm See doucmentation for the `basinhopping` algorithm, which also works with multivariate scalar optimization.

```
In [35]: from scipy.optimize import basinhopping

         x0 = 0
         sol = basinhopping(f, x0, stepsize=1, minimizer_kwargs={'args': (5,)})
         sol
```

```
Out[35]:                    nfev: 2017
          minimization_failures: 0
                            fun: -1.0
                              x: array([ 5.])
                        message: ['requested number of basinhopping iterations completed successfully']
                           njev: 671
                            nit: 100
```

```
In [36]: plt.plot(x, f(x, 5))
         plt.axvline(sol.x);
```



Minimizing a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$ We will next move on to optimization of multivariate scalar functions, where the scalar may (say) be the norm of a vector. Minimizing a multivariable set of equations $f : \mathbb{R}^n \to \mathbb{R}^n$ is not well-defined, but we will later see how to solve the closely related problme of finding roots or fixed points of such a set of equations.

We will use the Rosenbrock "banana" function to illustrate unconstrained multivariate optimization. In 2D, this is

$$f(x, y) = b(y - x^2)^2 + (a - x)^2$$

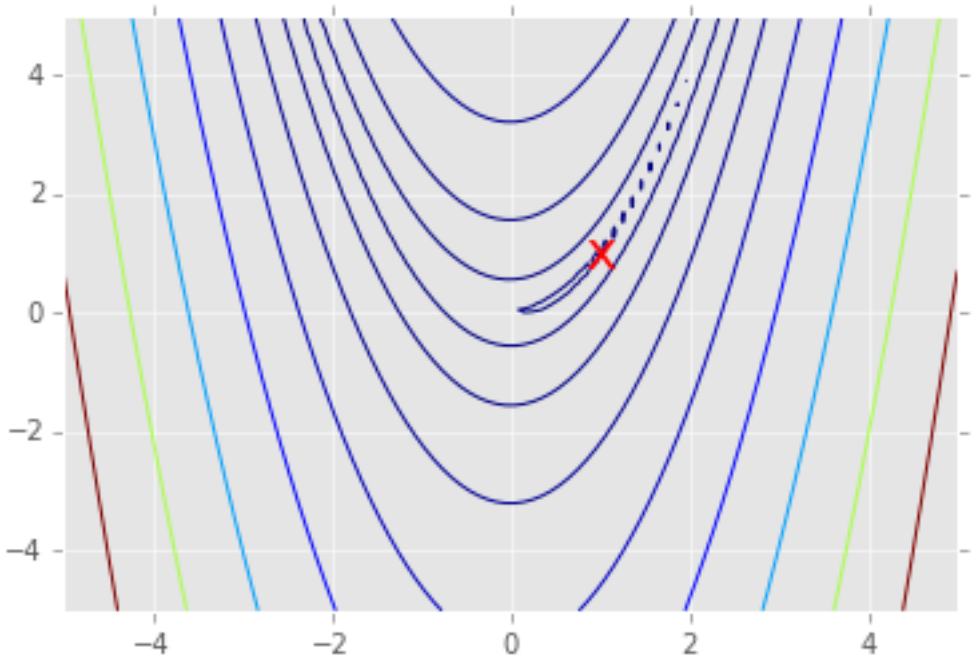
The function has a global minimum at (1,1) and the standard expression takes $a = 1$ and $b = 100$.

Conditinoning of otpimization problem With these values fr a and b, the problem is ill-conditioned. As we shall see, one of the factors affecting the ease of optimization is the condition number of the curvature (Hessian). When the codition 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.

```
In [37]: from sympy import symbols, hessian, Function, N

         x, y = symbols('x y')
         f = symbols('f', cls=Function)

         f = 100*(y - x**2)**2 + (1 - x)**2

         H = hessian(f, [x, y]).subs([(x,1), (y,1)])
         print np.array(H)
         print N(H.condition_number())
```

```
[[802 -400]
 [-400 200]]
2508.00960127744
```

```
In [38]: 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)

In [39]: x = np.linspace(-5, 5, 100)
         y = np.linspace(-5, 5, 100)
         X, Y = np.meshgrid(x, y)
         Z = rosen(np.vstack([X.ravel(), Y.ravel()])).reshape((100,100))

In [40]: # Note: the global minimum is at (1,1) in a tiny contour island
         plt.contour(X, Y, Z, np.arange(10)**5)
         plt.text(1, 1, 'x', va='center', ha='center', color='red', fontsize=20);
```



0.4 Gradient deescent

The gradient (or Jacobian) at a point indicates the direction of steepest ascent. Since we are looking for a minimum, one obvious possibility is to take a step in the opposite direction to the graident. We weight the size of the step by a factor α known in the machine learning literature as the learning rate. If α is small, the algorithm will eventually converge towards a local minimum, but it may take long time. If α is large, the algorithm may converge faster, but it may also overshoot and never find the minimum. Gradient descent is also known as a first order method because it requires calculation of the first derivative at each iteration.

Some algorithms also determine the appropriate value of α at each stage by using a line search, i.e.,

$$\alpha^* = \arg\min_{\alpha} f(x_k - \alpha \nabla f(x_k))$$

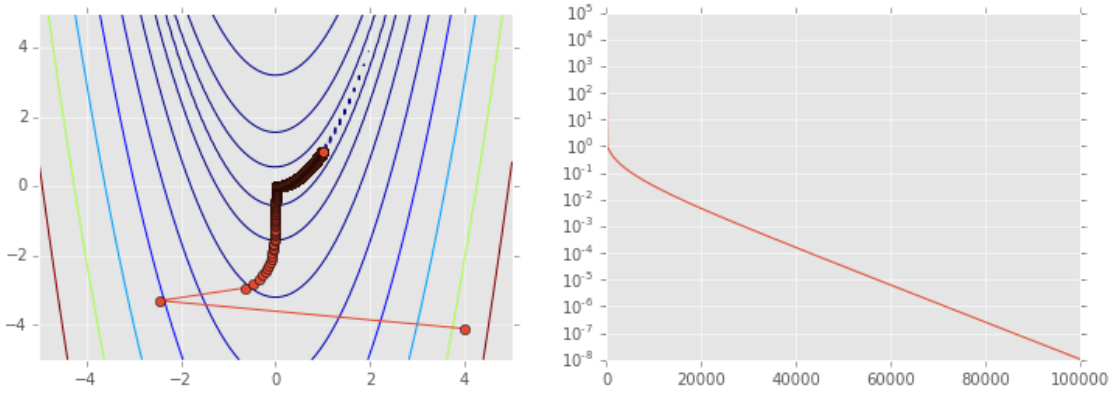
which is a 1D optimization problem.

As suggested above, the problem is that the gradient may not point towards the global minimum especially when the condition number is large, and we are forced to use a small α for convergence. Becasue gradient descent is unreliable in practice, it is not part of the scipy optimize suite of functions, but we will write a custom function below to ilustrate how it works.

```
In [41]: def rosen_der(x):
             """Derivative of generalized Rosen function."""
             xm = x[1:-1]
             xm_m1 = x[:-2]
             xm_p1 = x[2:]
             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 [42]: 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, success=(niter > 1))

In [43]: def reporter(p):
             """Reporter function to capture intermediate states of optimization."""
             global ps
             ps.append(p)

In [44]: # Initial starting position
         x0 = np.array([4,-4.1])

In [45]: ps = [x0]
         opt.minimize(rosen, x0, method=custmin, callback=reporter)

Out[45]:      fun: 1.0604663473471188e-08
             nfev: 100001
          success: True
```

```
            nit: 100000
              x: array([ 0.9999,  0.9998])
```

In [46]:
```
ps = np.array(ps)
plt.figure(figsize=(12,4))
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));
```



0.4.1 Newton's method and variants

Recall Newton's method for finding roots of a univariate function

$$x_{K+1} = x_k - \frac{f(x_k}{f'(x_k)}$$

When we are looking for a minimum, we are looking for the roots of the *derivative*, so

$$x_{K+1} = x_k - \frac{f'(x_k}{f''(x_k)}$$

Newotn's method can also be seen as a Taylor series approximation

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x)$$

At the function minimum, the derivtive is 0, so

$$\frac{f(x+h) - f(x)}{h} = f'(x) + \frac{h}{2}f''(x) \tag{1}$$

$$0 = f'(x) + \frac{h}{2}f''(x) \tag{2}$$

and letting $\Delta x = \frac{h}{2}$, we get that the Newton stpe is

$$\Delta x = -\frac{f'(x)}{f''(x)}$$

The multivariate analog replaces f' with the Jacobian and f'' with the Hessian, so the Newton step is

$$\Delta x = -H^{-1}(x)\nabla f(x)$$

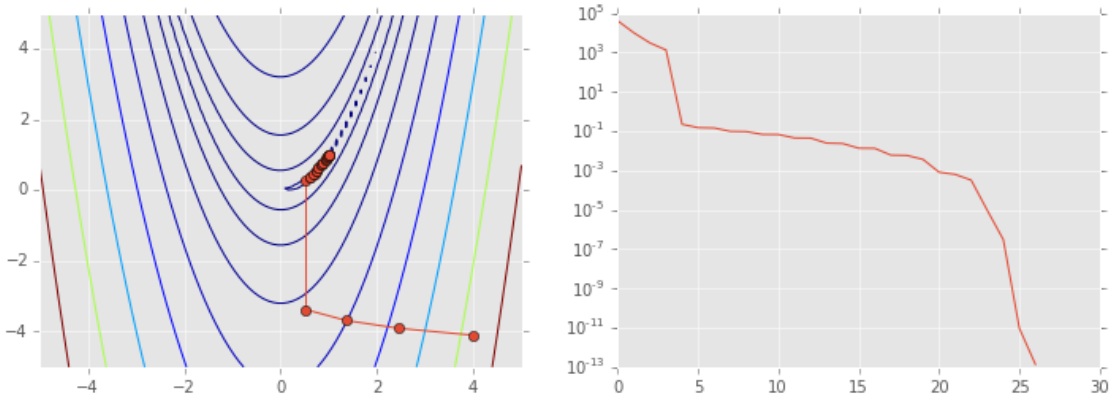
Second order methods Second order methods solve for H^{-1} and so require calculation of the Hessian (either provided or approximated uwing finite differences). For efficiency reasons, the Hessian is not directly inverted, but solved for using a variety of methods such as conjugate gradient. An example of a seocnd order method in the `optimize` package is `Newton-GC`.

```
In [47]: from scipy.optimize import rosen, rosen_der, rosen_hess
```

```
In [48]: ps = [x0]
         opt.minimize(rosen, x0, method='Newton-CG', jac=rosen_der, hess=rosen_hess, callback=reporter)
```

```
Out[48]:   status: 0
          success: True
             njev: 63
             nfev: 38
              fun: 1.3642782750354208e-13
                x: array([ 1.,  1.])
          message: 'Optimization terminated successfully.'
             nhev: 26
              jac: array([  1.2120e-04,  -6.0850e-05])
```

```
In [49]: ps = np.array(ps)
         plt.figure(figsize=(12,4))
         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));
```

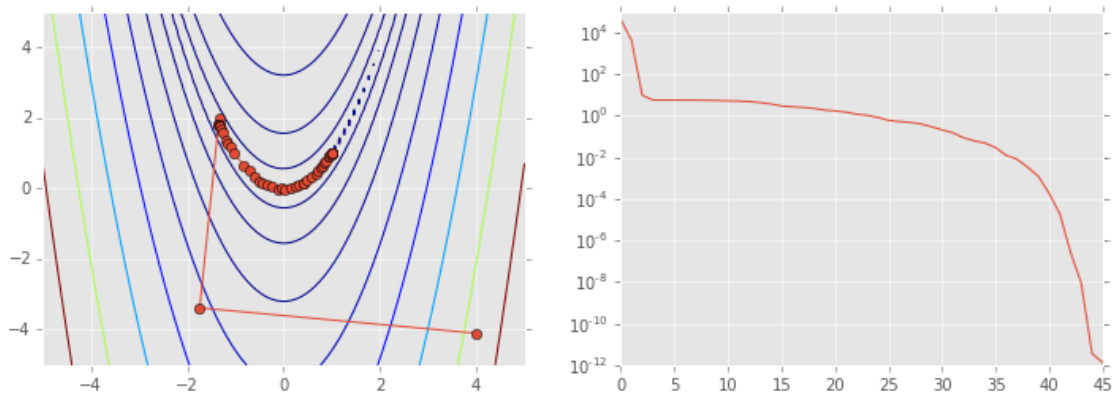


Frist order methods As calculating the Hessian is computationally expensive, first order methods only use the first derivatives. Quasi-Newton methods use functions of the first derivatives to approximate the inverse Hessian. A well know example of the Quasi-Newoton class of algorithjms is BFGS, named after the initials of the creators. As usual, the first derivatives can either be provided via the `jac=` argument or approximated by finite difference methods.

```
In [50]: ps = [x0]
         opt.minimize(rosen, x0, method='BFGS', callback=reporter)
```

```
Out[50]:    status: 2
           success: False
              njev: 92
              nfev: 379
          hess_inv: array([[ 0.5004,  1.0009],
                [ 1.0009,  2.0072]])
               fun: 1.2922663663359423e-12
                 x: array([ 1.,  1.])
           message: 'Desired error not necessarily achieved due to precision loss.'
               jac: array([  5.1319e-05,  -2.1227e-05])
```

```
In [51]: ps = np.array(ps)
         plt.figure(figsize=(12,4))
         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));
```



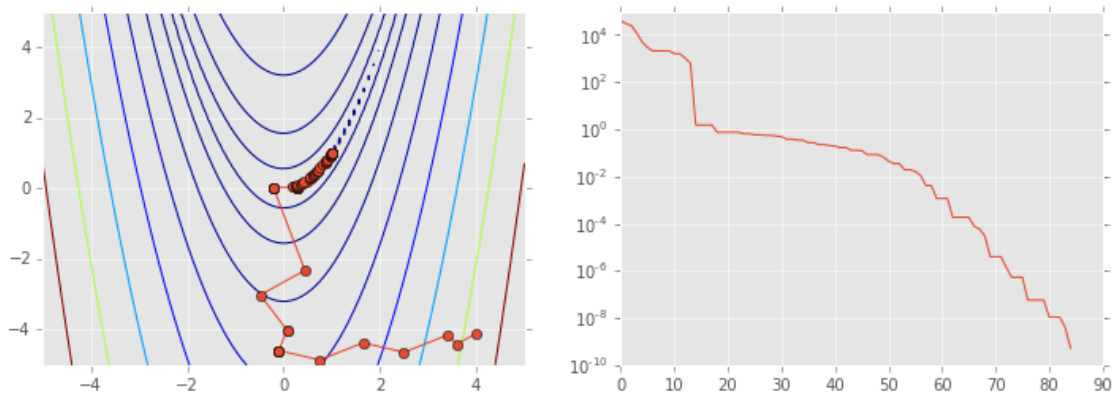
Zeroth order methods Finally, there are some optimization algorithms not based on the Newton method, but on other heuristic search strategies that do not require any derivatives, only function evaluations. One well-known example is the Nelder-Mead simplex algorithm.

```
In [52]: ps = [x0]
         opt.minimize(rosen, x0, method='nelder-mead', callback=reporter)
```

```
Out[52]:   status: 0
              nfev: 162
           success: True
               fun: 5.262756878429089e-10
                 x: array([ 1.,  1.])
           message: 'Optimization terminated successfully.'
              nit: 85
```

```
In [53]: ps = np.array(ps)
         plt.figure(figsize=(12,4))
         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));
```



0.4.2 Constrained optimization

Many real-world optimization problems have constraints - for example, a set of parameters may have to sum to 1.0 (eqquality constraint), or some parameters may have to be non-negative (inequality constraint). Sometimes, the constraints can be incorporated into the function to be minimized, for example, the non-negativity constraint $p > 0$ can be removed by substituting $p = e^q$ and optimizing for q. Using such workarounds, it may be possible to convert a constrained optimization problem into an unconstrained one, and use the methods discussed above to sovle the problem.

Alternatively, we can use optimization methods that allow the speicification of constraints directly in the problem statement as shown in this section. Internally, constraint violation penalties, barriers and Lagrange multpiliers are some of the methods used used to handle these constraints. We use the example provided in the Scipy tutorial to illustrate how to set constraints.

$$f(x) = -(2xy + 2x - x^2 - 2y^2)$$

subject to the constraint

$$x^3 - y = 0y - (x - 1)^4 - 2 \geq 0$$

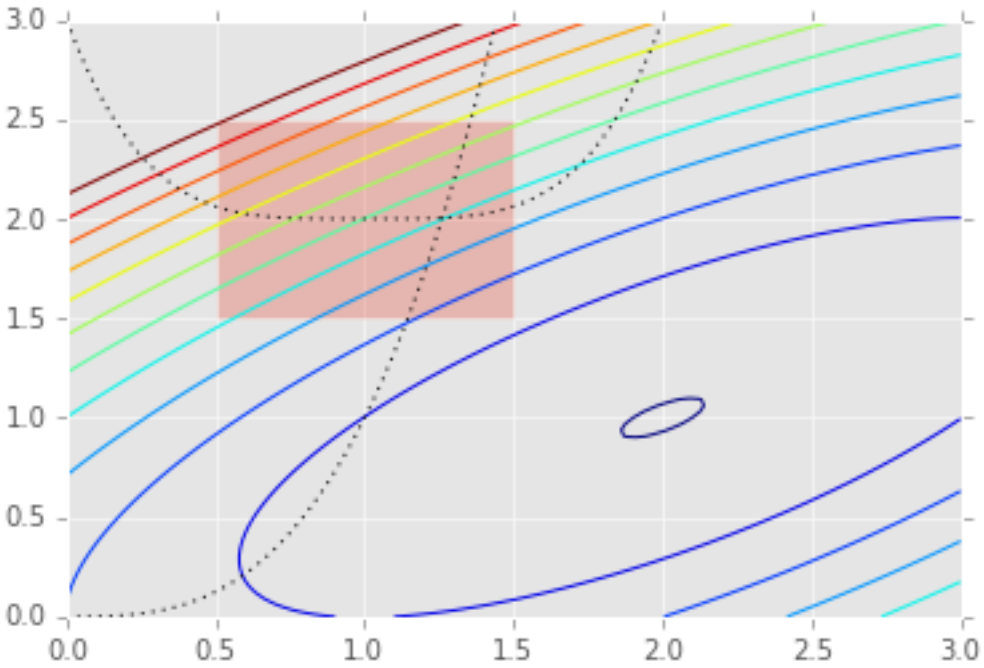
and the bounds

$$0.5 \leq x \leq 1.51.5 \leq y \leq 2.5$$

```
In [54]: def f(x):
             return -(2*x[0]*x[1] + 2*x[0] - x[0]**2 - 2*x[1]**2)
```

```
In [55]: 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.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[55]: [0, 3, 0, 3]
```



To set consttarints, we pass in a dictionary with keys `ty;pe`, `fun` and `jac`. Note that the inequlaity cosntraint assumes a $C_j x \geq 0$ form. As usual, the `jac` is optional and will be numerically estimted if not provided.

```
In [56]: 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))

In [57]: x0 = [0, 2.5]
```

Unconstrained optimization

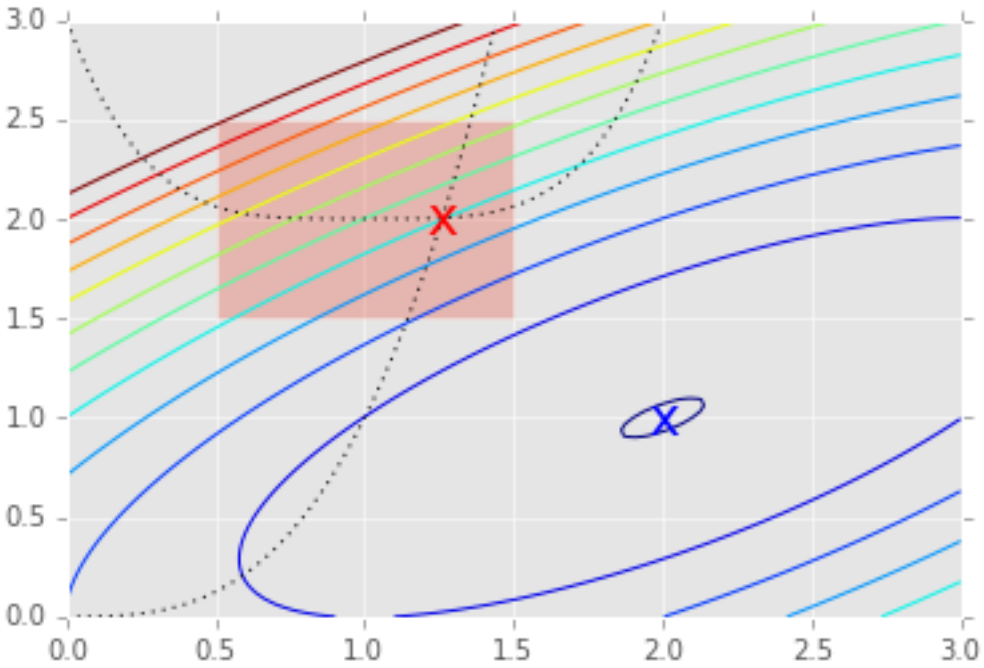
```
In [58]: ux = opt.minimize(f, x0, constraints=None)
         ux

Out[58]:    status: 0
          success: True
             njev: 5
             nfev: 20
         hess_inv: array([[ 1. ,  0.5],
                [ 0.5,  0.5]])
              fun: -1.9999999999999987
                x: array([ 2.,  1.])
          message: 'Optimization terminated successfully.'
              jac: array([ 0.,  0.])
```

Constrained optimization

```
In [59]: cx = opt.minimize(f, x0, bounds=bnds, constraints=cons)
         cx
```

```
Out[59]:   status: 0
          success: True
             njev: 5
             nfev: 21
              fun: 2.0499154720925521
                x: array([ 1.2609,  2.0046])
          message: 'Optimization terminated successfully.'
              jac: array([-3.4875,  5.4967,  0.    ])
              nit: 5
```

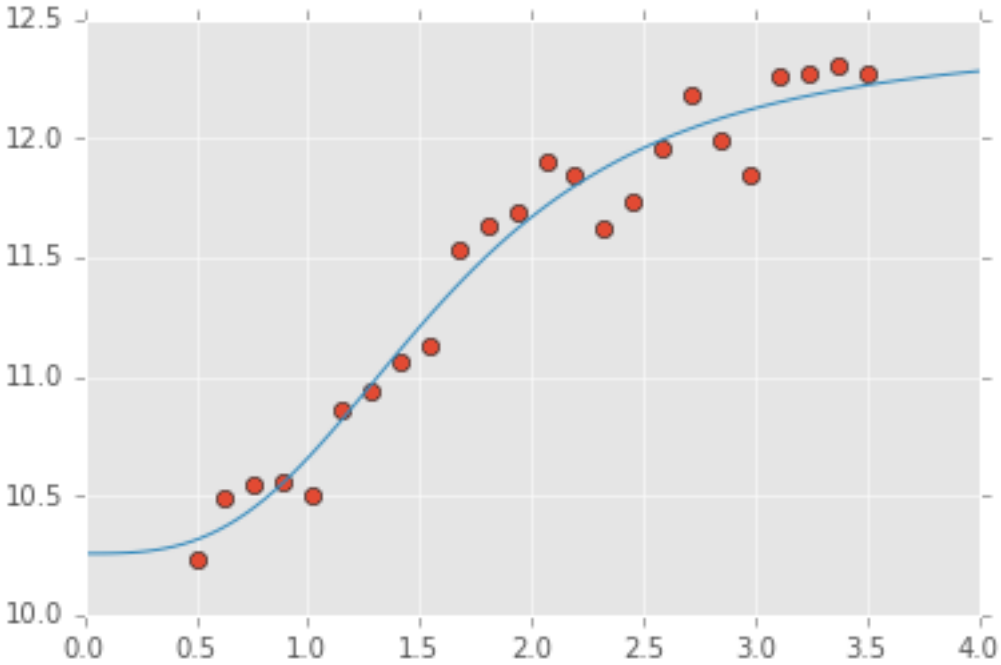
```
In [60]: x = np.linspace(0, 3, 100)
         y = np.linspace(0, 3, 100)
         X, Y = np.meshgrid(x, y)
         Z = f(np.vstack([X.ravel(), Y.ravel()])).reshape((100,100))
         plt.contour(X, Y, Z, np.arange(-1.99,10, 1));
         plt.plot(x, x**3, 'k:', linewidth=1)
         plt.plot(x, (x-1)**4+2, 'k:', linewidth=1)
         plt.text(ux['x'][0], ux['x'][1], 'x', va='center', ha='center', size=20, color='blue')
         plt.text(cx['x'][0], cx['x'][1], 'x', va='center', ha='center', size=20, color='red')
         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]);
```



0.4.3 Some applications of optimization

Curve fitting Sometimes, we simply want to use non-linear least squares to fit a function to data, perhaps to estimate paramters for a mechanistic or phenomenological model. The `curve_fit` function uses the quasi-Newton Levenberg-Marquadt aloorithm to perform such fits. Behind the scnees, `curve_fit` is just a wrapper around the `leastsq` function that we have already seen in a more conveneint format.

```
In [61]: from scipy.optimize import curve_fit

In [62]: def logistic4(x, a, b, c, d):
             """The four paramter logistic function is often used to fit dose-response relationships."""
             return ((a-d)/(1.0+((x/c)**b))) + d

In [63]: nobs = 24
         xdata = np.linspace(0.5, 3.5, nobs)
         ptrue = [10, 3, 1.5, 12]
         ydata = logistic4(xdata, *ptrue) + 0.5*np.random.random(nobs)

In [64]: popt, pcov = curve_fit(logistic4, xdata, ydata)

In [65]: perr = yerr=np.sqrt(np.diag(pcov))
         print 'Param\tTrue\tEstim (+/- 1 SD)'
         for p, pt, po, pe  in zip('abcd', ptrue, popt, perr):
             print '%s\t%5.2f\t%5.2f (+/-%5.2f)' % (p, pt, po, pe)

Param        True        Estim (+/- 1 SD)
a           10.00        10.26 (+/- 0.15)
b            3.00         3.06 (+/- 0.76)
c            1.50         1.62 (+/- 0.11)
d           12.00        12.41 (+/- 0.20)

In [66]: x = np.linspace(0, 4, 100)
         y = logistic4(x, *popt)
         plt.plot(xdata, ydata, 'o')
         plt.plot(x, y);
```



0.4.4 Finding paraemeters for ODE models

This is a specialized application of `curve_fit`, in which the curve to be fitted is defined implcitly by an ordinary differentail equation

$$\frac{dx}{dt} = -kx$$

and we want to use observed data to estiamte the parameters k and the initial value x_0. Of course this can be explicitly solved but the same approach can be used to find multiple paraemters for n-dimensional systems of ODEs.

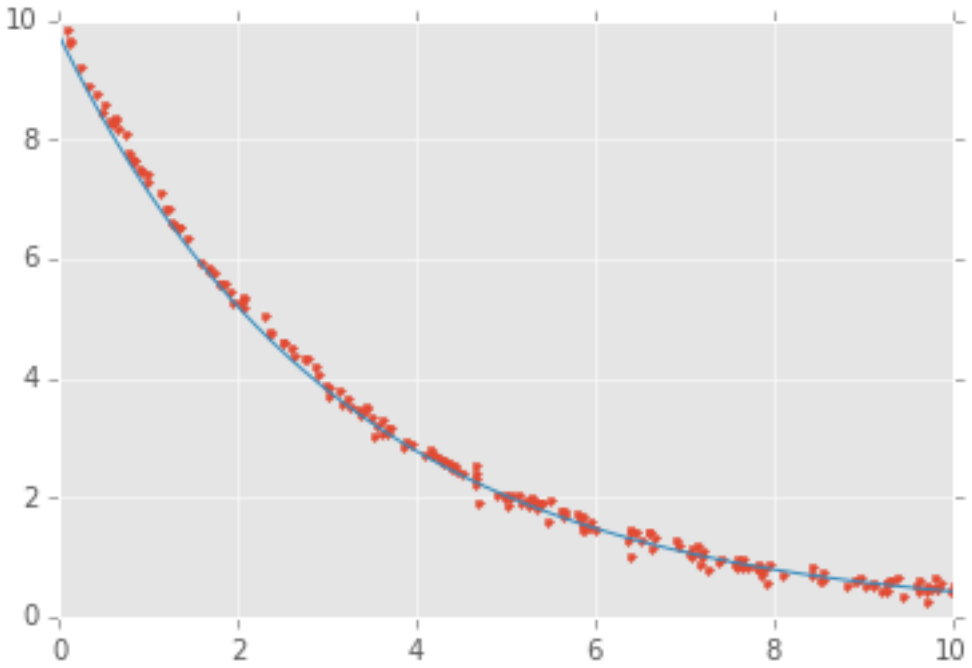
A more elaborate example for fitting a system of ODEs to model the zombie apocalypse

In [67]:
```
from scipy.integrate import odeint

def f(x, t, k):
    """Simple exponential decay."""
    return -k*x

def x(t, k, x0):
    """
    Solution to the ODE x'(t) = f(t,x,k) with initial condition x(0) = x0
    """
    x = odeint(f, x0, t, args=(k,))
    return x.ravel()
```

In [68]:
```
# True parameter values
x0_ = 10
k_ = 0.1*np.pi

# Some random data genererated from closed form soltuion plus Gaussian noise
ts = np.sort(np.random.uniform(0, 10, 200))
xs = x0_*np.exp(-k_*ts) + np.random.normal(0,0.1,200)

popt, cov = curve_fit(x, ts, xs)
k_opt, x0_opt = popt

print("k = %g" % k_opt)
print("x0 = %g" % x0_opt)
```

```
k = 0.314062
x0 = 9.754
```

In [69]:
```
import matplotlib.pyplot as plt
t = np.linspace(0, 10, 100)
plt.plot(ts, xs, '.', t, x(t, k_opt, x0_opt), '-');
```



Optimization of graph node placement To show the many different applications of optimization, here is an exmaple using optimization to change the layout of nodes of a graph. We use a physcial analogy - nodes are connected by springs, and the springs resist deformation from their natural length l_{ij}. Some nodes are pinned to their initial locations while others are free to move. Because the initial confiugraiton of nodes does not have springs at their natural lenght, there is tension resulting in a high potential energy U, given by the physics formula shown below. Optimization finds the configuraiton of lowest potential energy given that some nodes are fixed (set up as boundary constraints on the positions of the nodes).

$$U = \frac{1}{2}\sum_{i,j=1}^{n} ka_{ij}\left(||p_i - p_j|| - l_{ij}\right)^2$$

Note that the ordination algorithm Multi-Dimenisonal Scaling (MDS) works on a very similar idea - take a high dimensional data set in $\mathbb{R}^n$, and project down to a lower dimension ($\mathbb{R}^k$) such that the sum of distances $d_n(x_i, x_j) - d_k(x_i, x_j)$, where d_n adn d_k are some measure of distacce between two points x_i and x_j in n and d dimesniosn respectively, is minimized. MDS is often used in exploratory analysis of high-dimensional data to get some intuitive understanding of its "structure".

```
In [70]: from scipy.spatial.distance import pdist, squareform
```

- P0 is the initial location of nodes
- P is the minimal energy location of nodes given constraints
- A is a connectivity matrix - there is a spring between i and j if $A_{ij} = 1$
- L_{ij} is the resting length of the spring connecting i and j
- In addition, there are a number of `fixed` nodes whose positions are pinned.

```
In [71]: n = 20
         k = 1 # spring stiffness
         P0 = np.random.uniform(0, 5, (n,2))
         A = np.ones((n, n))
```

```
        A[np.tril_indices_from(A)] = 0
        L = A.copy()
```

In [72]:
```
def energy(P):
    P = P.reshape((-1, 2))
    D = squareform(pdist(P))
    return 0.5*(k * A * (D - L)**2).sum()
```

In [73]:
```
energy(P0.ravel())
```

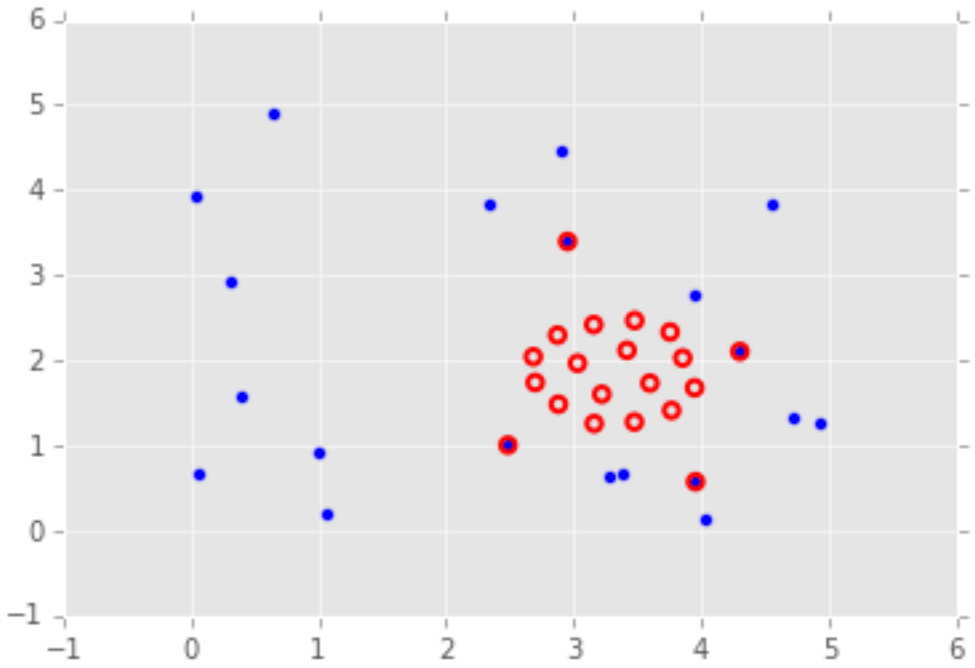
Out[73]: 542.8714

In [74]:
```
# fix the position of the first few nodes just to show constraints
fixed = 4
bounds = (np.repeat(P0[:fixed,:].ravel(), 2).reshape((-1,2)).tolist() +
          [[None, None]] * (2*(n-fixed)))
bounds[:fixed*2+4]
```

Out[74]:
```
[[4.3040, 4.3040],
 [2.1045, 2.1045],
 [2.4856, 2.4856],
 [1.0051, 1.0051],
 [2.9531, 2.9531],
 [3.3977, 3.3977],
 [3.9562, 3.9562],
 [0.5742, 0.5742],
 [None, None],
 [None, None],
 [None, None],
 [None, None]]
```

In [75]:
```
sol = opt.minimize(energy, P0.ravel(), bounds=bounds)
```

In [76]:
```
plt.scatter(P0[:, 0], P0[:, 1], s=25)
P = sol.x.reshape((-1,2))
plt.scatter(P[:, 0], P[:, 1], edgecolors='red', facecolors='none', s=30, linewidth=2);
```



0.5 Optimization of standard statistical models

When we solve standard statistical problems, an optimization procedure similar to the ones discussed here is performed. For example, consider multivariate logistic regression - typically, a Newton-like alogirhtm known as iteratively reweighted least squares (IRLS) is used to find the maximum likelihood estimate for the generalized linear model family. However, using one of the multivariate scalar minimization methods shown above will also work, for example, the BFGS minimization algorithm.

The take home message is that there is nothing magic going on when Python or R fits a statistical model using a formula - all that is happening is that the objective function is set to be the negative of the log likelihood, and the minimum found using some first or second order optimzation algorithm.

```
In [77]: import statsmodels.api as sm
```

0.5.1 Logistic regression as optimization

Suppose we have a binary outcome measure $Y \in 0, 1$ that is conditinal on some input variable (vector) $x \in (-\infty, +\infty)$. Let the conditioanl probability be $p(x) = P(Y = y|X = x)$. Given some data, one simple probability model is $p(x) = \beta_0 + x \cdot \beta$ - i.e. linear regression. This doesn't really work for the obvious reason that $p(x)$ must be between 0 and 1 as x ranges across the real line. One simple way to fix this is to use the transformation $g(x) = \frac{p(x)}{1-p(x)} = \beta_0 + x.\beta$. Solving for p, we get

$$p(x) = \frac{1}{1 + e^{-(\beta_0 + x \cdot \beta)}}$$

As you all know very well, this is logistic regression.

Suppose we have n data points (x_i, y_i) where x_i is a vector of features and y_i is an observed class (0 or 1). For each event, we either have "success" $(y = 1)$ or "failure" $(Y = 0)$, so the likelihood looks like the product of Bernoulli random variables. According to the logistic model, the probability of success is $p(x_i)$ if $y_i = 1$ and $1 - p(x_i)$ if $y_i = 0$. So the likelihood is

$$L(\beta_0, \beta) = \prod_{i=1}^{n} p(x_i)^y (1 - p(x_i))^{1-y}$$

and the log-likelihood is

$$l(\beta_0, \beta) = \sum_{i=1}^{n} y_i \log p(x_i) + (1 - y_i) \log 1 - p(x_i) \tag{3}$$

$$= \sum_{i=1}^{n} \log 1 - p(x_i) + \sum_{i=1}^{n} y_i \log \frac{p(x_i)}{1 - p(x_i)} \tag{4}$$

$$= \sum_{i=1}^{n} - \log 1 + e^{\beta_0 + x_i \cdot \beta} + \sum_{i=1}^{n} y_i(\beta_0 + x_i \cdot \beta) \tag{5}$$

Using the standard 'trick', if we augment the matrix X with a column of 1s, we can write $\beta_0 + x_i \cdot \beta$ as just $X\beta$.

```
In [78]: df_ = pd.read_csv("http://www.ats.ucla.edu/stat/data/binary.csv")
         df_.head()
```

```
Out[78]:    admit  gre   gpa  rank
         0      0  380  3.61     3
```

```
1      1  660  3.67     3
2      1  800  4.00     1
3      1  640  3.19     4
4      0  520  2.93     4
```

```
In [79]: # We will ignore the rank categorical value

         cols_to_keep = ['admit', 'gre', 'gpa']
         df = df_[cols_to_keep]
         df.insert(1, 'dummy', 1)
         df.head()
```

```
Out[79]:    admit  dummy  gre   gpa
         0      0      1  380  3.61
         1      1      1  660  3.67
         2      1      1  800  4.00
         3      1      1  640  3.19
         4      0      1  520  2.93
```

0.5.2 Solving as a GLM with IRLS

This is very similar to what you would do in R, only using Python's `statsmodels` package. The GLM solver uses a special variant of Newton's method known as iteratively reweighted least squares (IRLS), which will be further desribed in the lecture on multivarite and constrained optimizaiton.

```
In [80]: model = sm.GLM.from_formula('admit ~ gre + gpa',
                                      data=df, family=sm.families.Binomial())
         fit = model.fit()
         fit.summary()
```

```
Out[80]: <class 'statsmodels.iolib.summary.Summary'>
         """
                          Generalized Linear Model Regression Results
         ==============================================================================
         Dep. Variable:                   admit   No. Observations:                  400
         Model:                             GLM   Df Residuals:                      397
         Model Family:                Binomial   Df Model:                            2
         Link Function:                   logit   Scale:                           1.0
         Method:                           IRLS   Log-Likelihood:                -240.17
         Date:                Wed, 11 Feb 2015   Deviance:                       480.34
         Time:                        17:29:26   Pearson chi2:                     398.
         No. Iterations:                     5
         ==============================================================================
                          coef    std err          t      P>|t|      [95.0% Conf. Int.]
         ------------------------------------------------------------------------------
         Intercept     -4.9494      1.075     -4.604      0.000        -7.057    -2.842
         gre            0.0027      0.001      2.544      0.011         0.001     0.005
         gpa            0.7547      0.320      2.361      0.018         0.128     1.381
         ==============================================================================
         """
```

0.5.3 Solving as logistic model with bfgs

Note that you can choose any of the scipy.optimize algotihms to fit the maximum likelihood model. This knows about higher order derivatives, so will be more accurate than homebrew version.

```
In [81]: model2 = sm.Logit.from_formula('admit ~ %s' % '+'.join(df.columns[2:]), data=df)
         fit2 = model2.fit(method='bfgs', maxiter=100)
         fit2.summary()

Optimization terminated successfully.
         Current function value: 0.600430
         Iterations: 23
         Function evaluations: 65
         Gradient evaluations: 54

Out[81]: <class 'statsmodels.iolib.summary.Summary'>
         """
                                  Logit Regression Results
         ==============================================================================
         Dep. Variable:                  admit   No. Observations:                  400
         Model:                          Logit   Df Residuals:                      397
         Method:                           MLE   Df Model:                            2
         Date:                Wed, 11 Feb 2015   Pseudo R-squ.:                 0.03927
         Time:                        17:31:19   Log-Likelihood:                -240.17
         converged:                       True   LL-Null:                       -249.99
                                                 LLR p-value:                 5.456e-05
         ==============================================================================
                         coef    std err          z      P>|z|      [95.0% Conf. Int.]
         ------------------------------------------------------------------------------
         Intercept     -4.9494      1.075     -4.604      0.000        -7.057    -2.842
         gre            0.0027      0.001      2.544      0.011         0.001     0.005
         gpa            0.7547      0.320      2.361      0.018         0.128     1.381
         ==============================================================================
         """
```

0.5.4 Home-brew logistic regression using a generic minimization function

This is to show that there is no magic going on - you can write the function to minimize directly from the log-likelihood eqaution and run a minimizer. It will be more accurate if you also provide the derivative (+/- the Hessian for seocnd order methods), but using just the function and numerical approximations to the derivative will also work. As usual, this is for illustration so you understand what is going on - when there is a library function available, youu should probably use that instead.

```
In [82]: def f(beta, y, x):
             """Minus log likelihood function for logistic regression."""
             return -((-np.log(1 + np.exp(np.dot(x, beta)))).sum() + (y*(np.dot(x, beta))).sum())

In [83]: beta0 = np.zeros(3)
         opt.minimize(f, beta0, args=(df['admit'], df.ix[:, 'dummy':]), method='BFGS', options={'gtol':

Out[83]:    status: 0
          success: True
             njev: 16
             nfev: 80
         hess_inv: array([[  1.1525e+00,  -2.7800e-04,  -2.8160e-01],
                [ -2.7800e-04,   1.1663e-06,  -1.2190e-04],
                [ -2.8160e-01,  -1.2190e-04,   1.0259e-01]])
              fun: 240.1719908951104
                x: array([ -4.9493e+00,   2.6903e-03,   7.5473e-01])
          message: 'Optimization terminated successfully.'
              jac: array([  9.1553e-05,  -3.2158e-03,   4.5776e-04])
```

0.5.5 Resources

- Scipy Optimize refernce
- Scipy Optimize tutorial
- LMFit - a modeling interface for nonlinear least squares problems
- CVXpy- a modeling interface for convex optimization problems
- Quasi-Newton methods
- Convex optimization book by Boyd & Vandenberghe

In []: