william's class lecture

Numerical Optimization

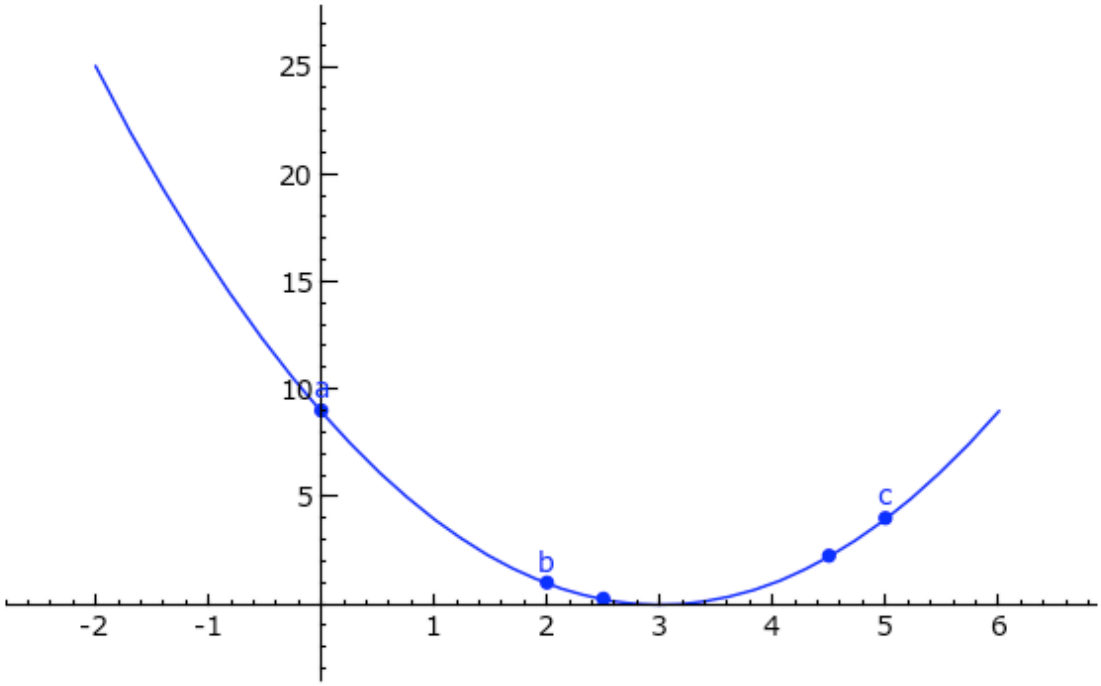
Related to root finding is the problem or minimizing (or equivalently maximizing) a function. They are related problems because a root of a function f minimizes $|f(x)|$, this fact is exploited by some multidimensioal root finding algorithms. I'm going to talk about optimization algorithms in general dimensions.

Note that optimizing in one dimension is similar to one dimensional root finding:

Quick aside on minimization in One dimenson

There is a simple modification of the one dimensional bracketing algorithm for root finding to finding minimums. It involves observing that bracketing a minimum requires 3 points $a < b < c$ with $f(b) < f(a)$, $f(b) < f(c)$. Starting with an initial a, b, c, choose d with $a < b < d < c$. If $f(b) < f(d)$ our triple becomes (a, b, d). If $f(b) > f(d)$ our triple becomes (b, d, c). Now iterate to refine the bracket.

```
%hide
f=(x-3)^2
plot(f,(-2,6))+point((0,f(0)),pointsize=30)+point((2,f(2)),pointsize=3
```



Rough Idea of Minimization Algorithms

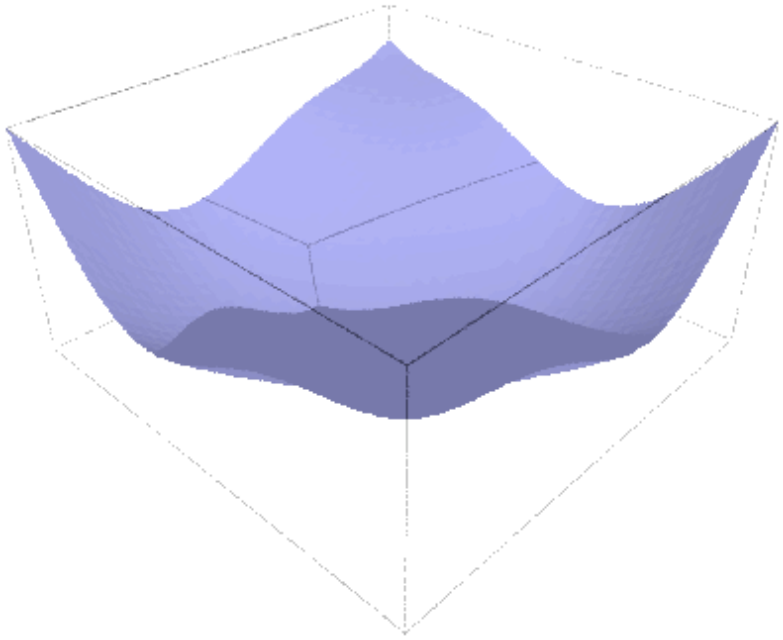
Suppose $f : \mathbf{R}^n \to \mathbf{R}$ and we want to find a minimum of f (hopefully f has at least a local minimum). Let us assume that f is smooth. Recall the gradient of a function

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n}\right)$$

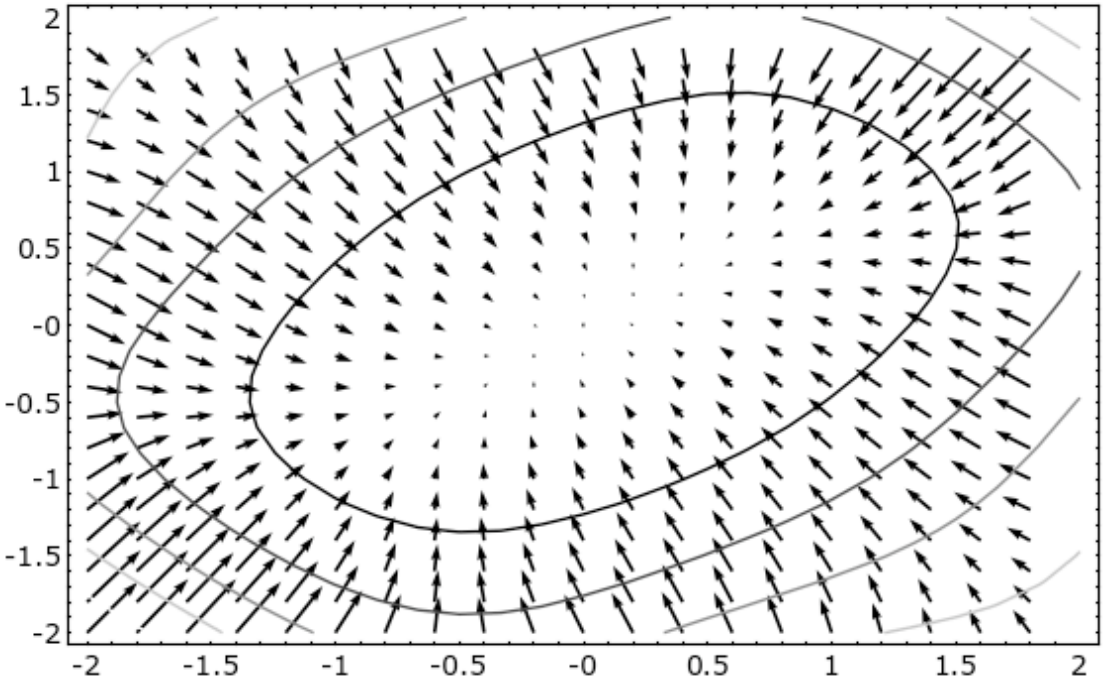
Note that

1. At a minimum $\nabla f = 0$
2. $-\nabla f$ points in the direction that f is decreasing most rapidly. It is orthogonal to the level sets of the functions.

```
var('x y')
f=x^2+y^2-sin(x*y)
plot3d(f,(x,-2,2),(y,-2,2),viewer='tachyon')
```



```
contour_plot(f,(-2,2),(-2,2),fill=false)+plot_vector_field((-f).gradie
```



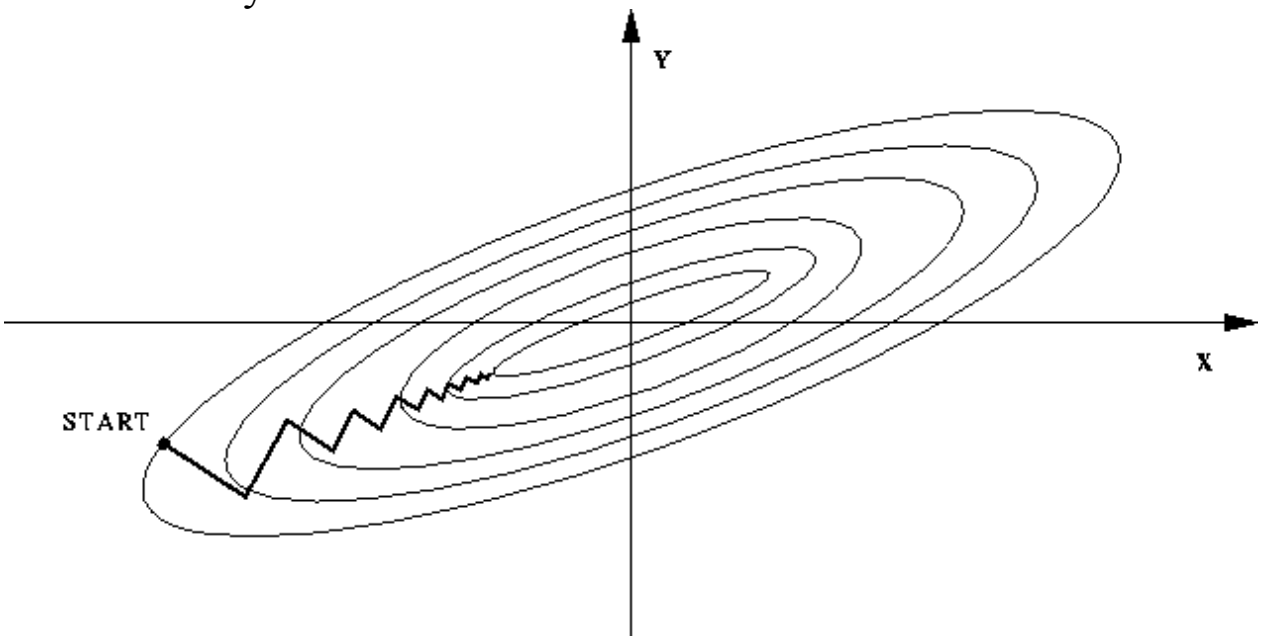
Idea

To find a minimum, pick a point, compute the gradient, take a step in the direction of the gradient. Compute the gradient again, take another step. Keep going till the gradient vanishes, hopefully you have found the minimum. Many algorithms work more or less this way. The details being in how one decides how large a step to take, and in sometimes finding better directions. Often one will repeatedly do line minimization in different directions. Some algorithms incorporate second derivative information (hessian), others approximate the derivatives implicitly.

Potential ways minimization can Fail

The most common example of problematic functions are long,

narrow valleys



Application to Statistics

Suppose you have a list of measurements $[x_1, x_2, \ldots, x_m]$ and you have a function $p(x, a_1, a_2, ..., a_n)$. Suppose you believe there is a set of model parameters $(a_1, \ldots, a_n)$, such that $p(x, a_1, \ldots, a_n)$ is probablity of measuring the value x. You want to determine an estimate of the parameters $(a_1, \ldots a_n)$ from your data.

Recall that for two independent events the probabilty of both happening is the product of their individual probabilties. So the probability of measuring x_1 and x_2 is

$$p(x_1, a_1, \ldots, a_n)p(x_2, a_1, \ldots, a_n).$$

So if our measurements are independent the probability of obtaining $[x_1, \ldots x_n]$ is

$$\Pi_{i=1}^{m} p(x_i, a_1, \ldots, a_n).$$

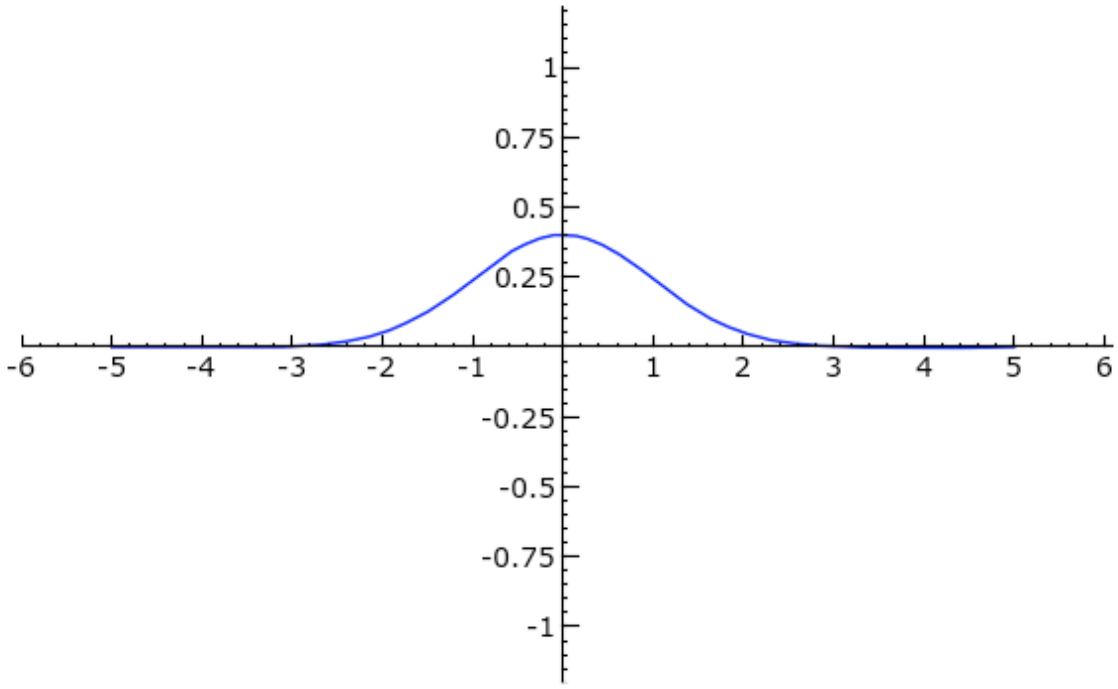
Intuitively the correct parameters should give a high probability of producing the measured values so should maximimize this product. To make things easier we will actually minimize the function

$$f(a_1, \ldots a_n) = -log(\Pi_{i=1}^{m} p(x_i, a_1, \ldots, a_n)) = -\sum_{i}^{m} log(p(x_i, a_1, \ldots, a_n))$$

Note that if we have a lot of data this will be pretty expensive. To maybe illustrate this, let is give an example showing that the usual mean and standard deviation arise from estimating parameters in a model where the function p is gaussian of the form.

$$p(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{(2\pi)}} e^{-(x-\mu)^2/(2\sigma^2)}.$$

```
var('u s x')
p=1/(s*sqrt(2*pi))*e^(-(x-u)^2/(2*s^2))
plot(p.substitute(u=0,s=1),(-5,5))
```



To emphasize: the interpretation of this, is that the probability of measuring a given x value is the height of the function at the point x

.

```
l=range(100)
p_gaussian_log=-log(1/(s*sqrt(2*pi))*e^(-(x-u)^2/(2*s^2)))
p_log=sum([ p_gaussian_log.substitute(x=v) for v in l])
```

```
minimize?
```

```
minimize(p_log,(50,50))
```

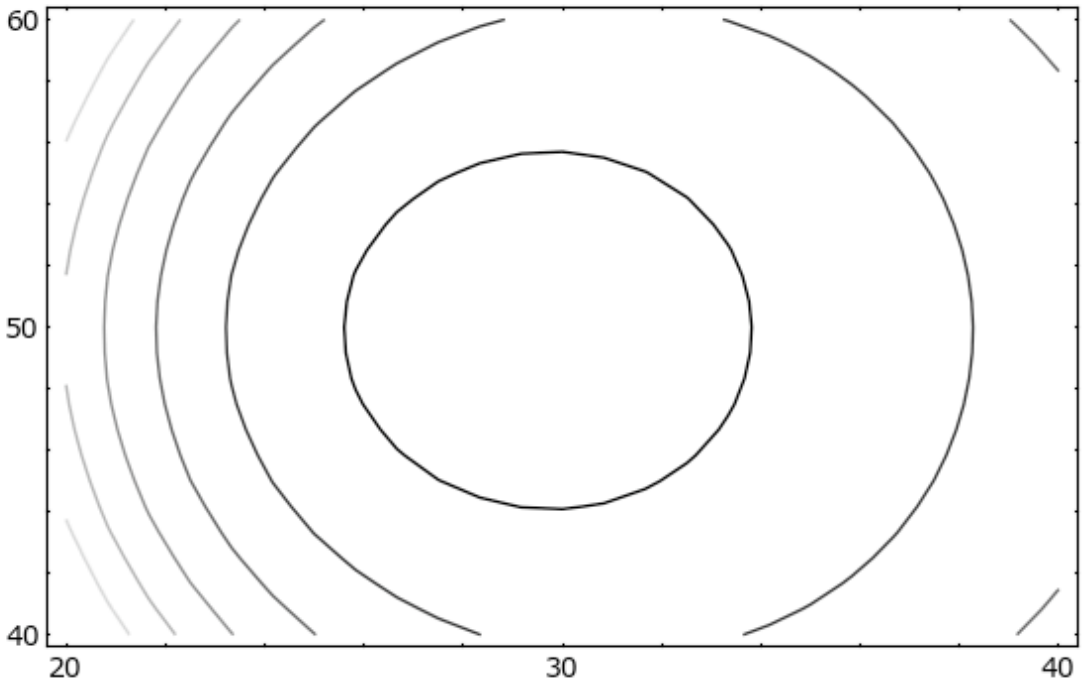
```
    Optimization terminated successfully.
             Current function value: 478.160539
             Iterations: 11
             Function evaluations: 16
             Gradient evaluations: 16
    (28.8660707322, 49.4999890179)
```

```
from scipy import stats
[stats.std(l),stats.mean(l)]
```

```
    [29.011491975882016, 49.5]
```

Actually it gives a different estimate of standard deviation, this is related to that $1/(n-1)$ in the formula for standard deviation, this gives the same estimate but with $1/n$.

```
contour_plot(p_log,(20,40),(40,60),fill=false)
```

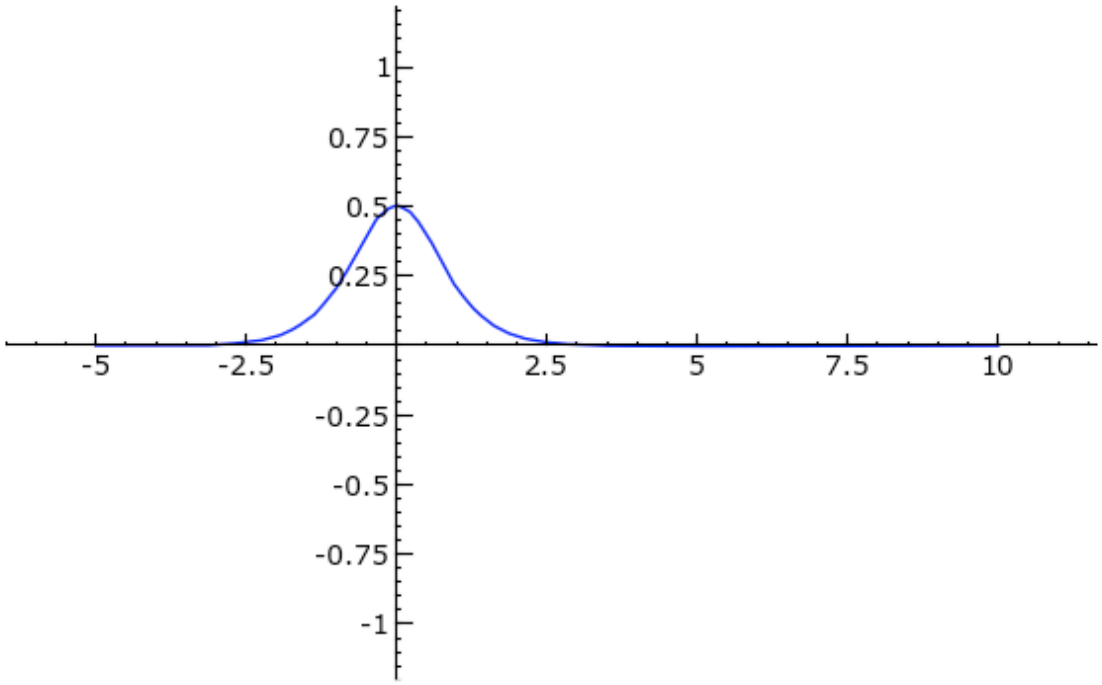


We will now do a more complicated example where assume that our

p is of the form

$$p(x,a,b) = \frac{ae^{-ax-b}}{(1+e^{-ax-b})^2}.$$

```
var('a b x')
p=a*e^(-a*x-b)/(1+e^(-a*x-b))^2
plot(p.substitute(a=2,b=0),(-5,10))
```



This looks similar to the standard normal distribution but (though its hard to see visually) it allows extreme events to have higher probability. I have a data file of data consistent with this model, lets try to estimate the parameters. Lets actually write a simple gradient descent algorithm and see if that works for us.

```
import numpy
import scipy
from scipy import optimize
```

```
A=numpy.loadtxt('%s/hw1-logistic-data.dat'%DATA)
```

We symbolically compute the functions and their derivatives. The

minimize function does this for you.

```
p=-log(a*e^(-a*x-b)/(1+e^(-a*x-b))^2)
p_a=p.derivative(a)
p_b=p.derivative(b)


p_fast=p._fast_float_('x','a','b')
p_a_fast=p_a._fast_float_('x','a','b')
p_b_fast=p_b._fast_float_('x','a','b')


def p_func(param):
  l=[p_fast(x,param[0],param[1]) for x in A]
  return sum(l)

def p_gradient(param):
  l_a=[p_a_fast(x,param[0],param[1]) for x in A]
  l_b=[p_b_fast(x,param[0],param[1]) for x in A]
  return scipy.array([sum(l_a),scipy.sum(l_b)])
```

```
from numpy import linalg
def steepest_descent(p,n,f,gradient,alpha):
  point_array=[point(p),text(str(0),p)]
  tol=float(1.0/10^3)
  for i in range(n):
     grad=gradient(p)
     g=lambda alpha: f(p+alpha*grad)
     p=grad*optimize.bracket(g,-alpha,alpha)[1]+p
     if linalg.norm(grad)<tol:
        print "final point: %s"%str(p)+"\n"
        print "iteration: %d"%i
        break
     point_array.append(point(p))
     point_array.append(text(str(i+1),p+numpy.array([.1,.1])))

  return p,point_array
```

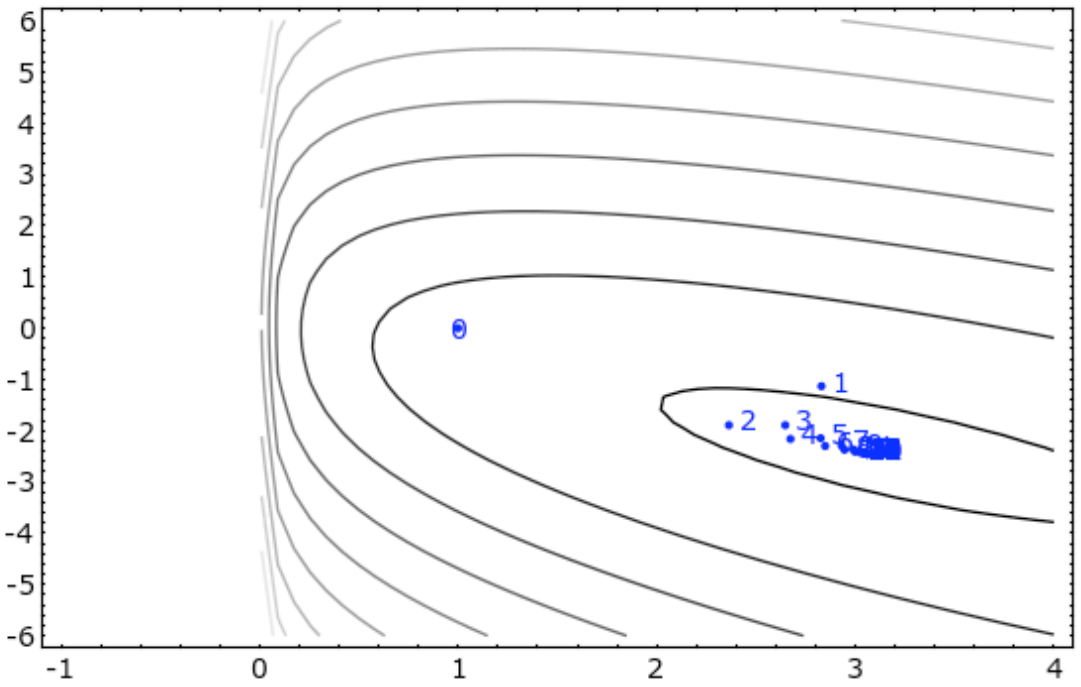
```
RealNumber=float
Integer=int
```

```
output_1_steepest=steepest_descent([1.0,0.0],1000,p_func,p_gradient,.(
```

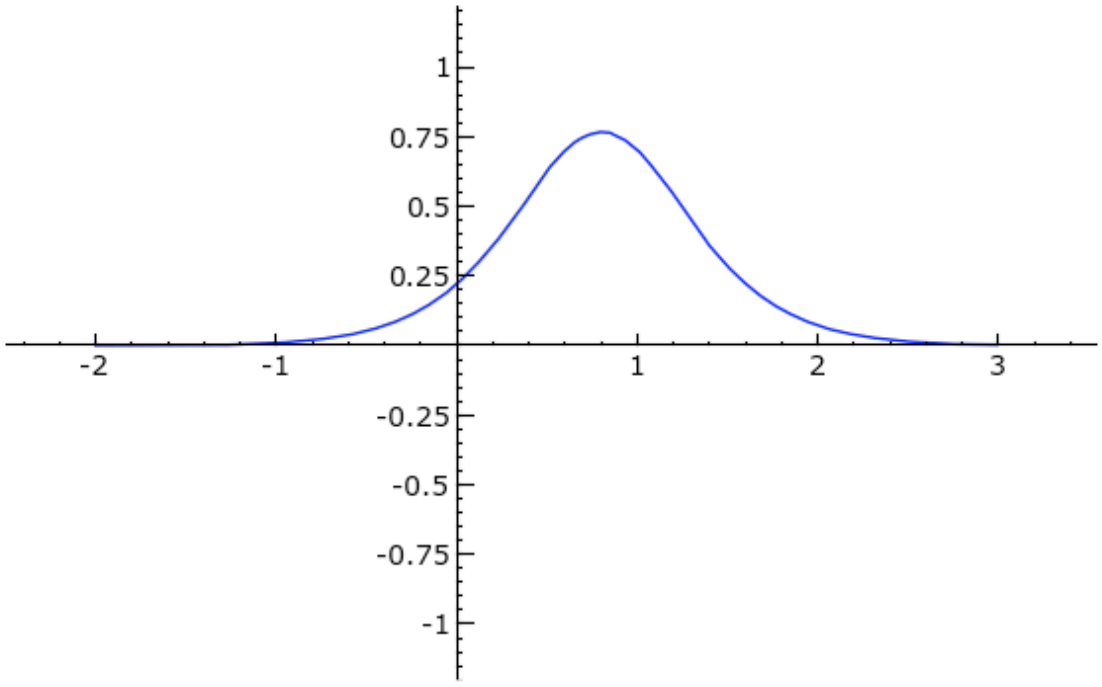
```
final point: [ 3.06027597 -2.45228936]
```

```
iteration: 41
```

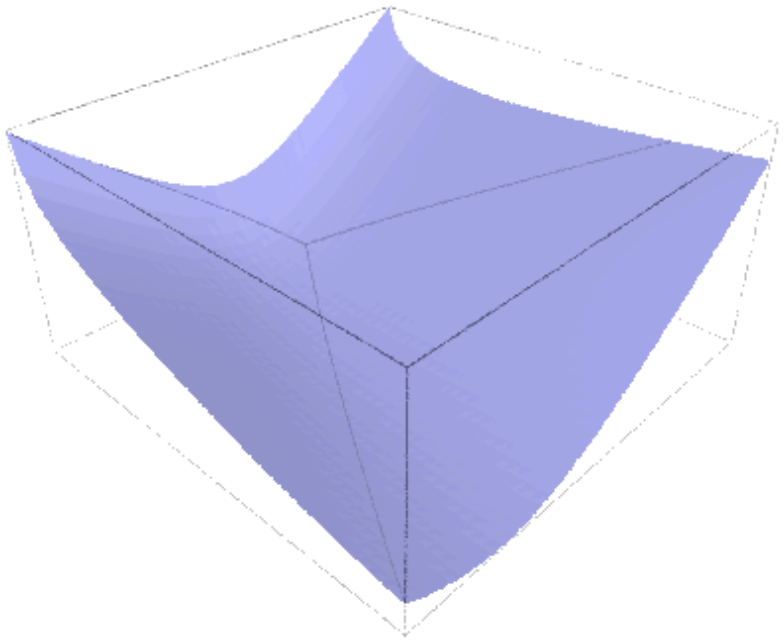
```
contour_plot(lambda
x,y:p_func([x,y]),(.01,4),(-6,6),fill=false,plot_points=50,contours=1(
```




```
p=a*e^(-a*x-b)/(1+e^(-a*x-b))^2
plot(p.substitute(a=3.06027786,b=-2.45229071),-2,3)
```




```
plot3d(lambda x,y: p_func([x,y]),[.05,4],[-6,6],viewer='tachyon')
```



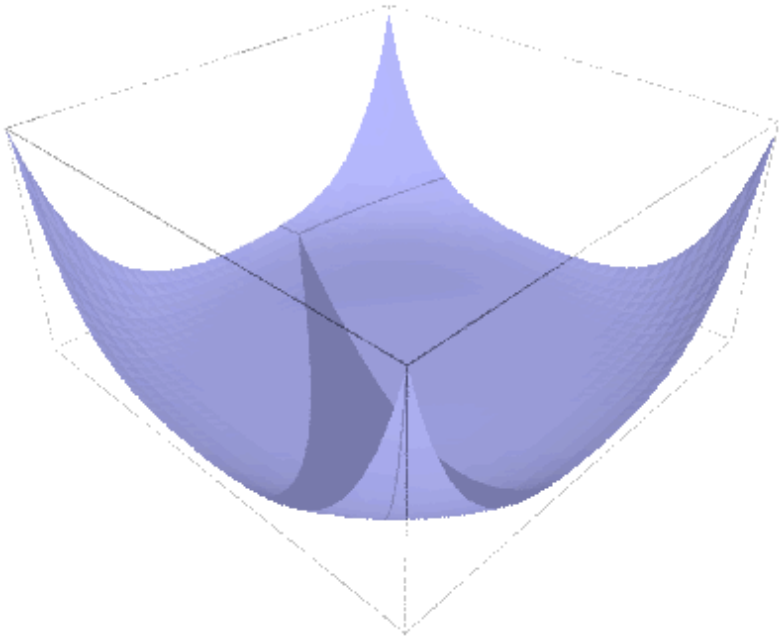
```
minimize(p_func,[1.0,0],p_gradient,algorithm='bfgs')
```

```
    Optimization terminated successfully.
             Current function value: 1767.095292
             Iterations: 7
             Function evaluations: 10
             Gradient evaluations: 10
    (3.06027835164, -2.45229123483)
```

A pathological Example

```
var('x y')
f_2= y^2-1.98*y*(x^2+y^2)^2 + (x^2+y^2)^4
```

```
plot3d(f_2._fast_float_('x','y'),(-3,3),(-3,3),viewer='tachyon')
```



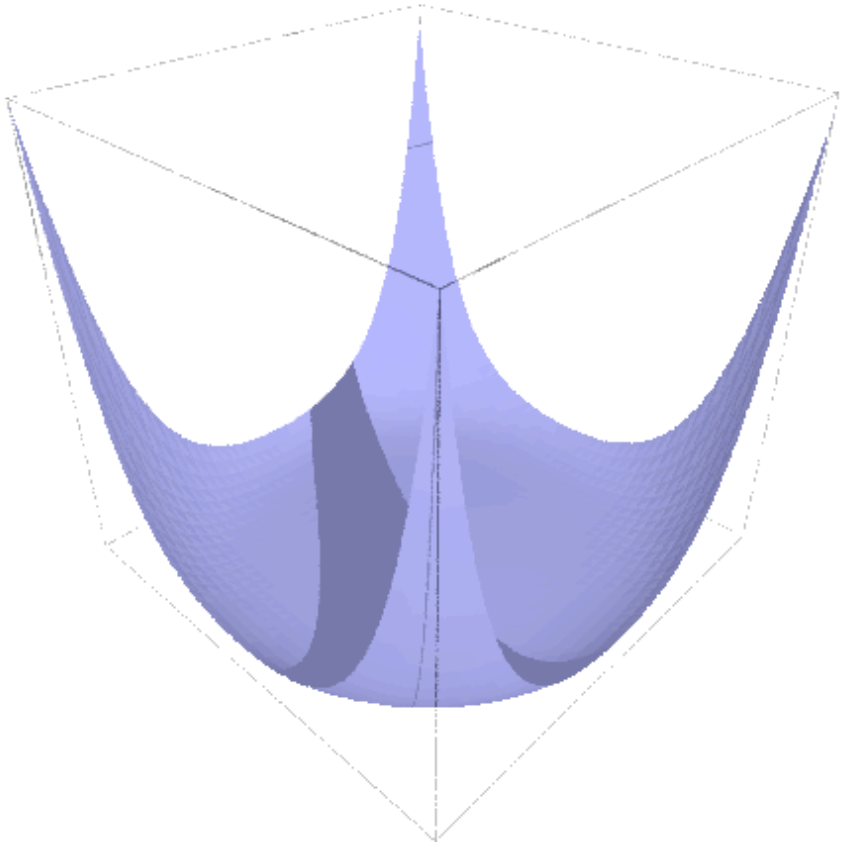
```
minimum_point=minimize(f_2,[.5,.5])
```

```
    Optimization terminated successfully.
             Current function value: 0.000000
             Iterations: 46
             Function evaluations: 53
             Gradient evaluations: 53
```

```
minimum_point
```

```
    (0.110515880355, 0.000147971811696)
```

```
plot3d(f_2._fast_float_('x','y'),(-3,3),(-3,3),viewer='tachyon')+point
```



Implementation Of Minimize/Directly Using Scipy

```
minimize??
```

```
scipy.optimize.fmin_bfgs
```

```
    <function fmin_bfgs at 0xe967cb0>
```

```
def rosen(x): # The Rosenbrock function
   return sum(100.0r*(x[1r:]-x[:-1r]**2.0r)**2.0r +
(1r-x[:-1r])**2.0r)
import numpy
from numpy import zeros
```

```
def rosen_der(x):
   xm = x[1r:-1r]
   xm_m1 = x[:-2r]
   xm_p1 = x[2r:]
   der = zeros(x.shape,dtype=float)
   der[1r:-1r] = 200r*(xm-xm_m1**2r) - 400r*(xm_p1 - xm**2r)*xm -
2r*(1r-xm)
   der[0] = -400r*x[0r]*(x[1r]-x[0r]**2r) - 2r*(1r-x[0])
   der[-1] = 200r*(x[-1r]-x[-2r]**2r)
   return der
minimize(rosen,[.1,.3,.4],gradient=rosen_der,algorithm="bfgs",disp=0)
```

```
(1.00000000342, 1.00000000613, 1.00000001208)
```

```
optimize.fmin_powell(rosen,[.1,.3,.4])
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 19
         Function evaluations: 762
array([0.999999999999956, 0.999999999999917, 0.999999999999843],
dtype=object)
```

```
optimize.fmin_bfgs?
```

```
optimize.fmin_bfgs(rosen,[.1r,.2r,.3r],fprime=rosen_der)
```

```
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 25
         Function evaluations: 33
         Gradient evaluations: 33
array([ 1.        ,  0.99999999,  0.99999999])
```