Optimization in multiple dimensions

Goal is to find a local (ideally global) minimizer of $f : \mathbb{R}^n \to \mathbb{R}$.

Objective: best roast turkey
D.o.f: bake time and temperature

Newton on gradient

If f is 3 times continuously differentiable we could note that a local minimizer of f is a zero of the function $\nabla f : \mathbb{R}^n \to \mathbb{R}^n \in C^2$, and we could use Newton's method on ∇f and expect quadratic convergence.

But note that zeros of ∇F can be minimizers, but also maximizers and even saddle points.

Example: Find local minimizers of

$$f(x, y) = x^3 + y^3 + 3x^2 - 3y^2 + 12.$$

Let's define $g(x) = \nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix}$. Then $g'(x) = \begin{bmatrix} \frac{\partial}{\partial x_1}\left(\frac{\partial f}{\partial x_1}\right), \frac{\partial}{\partial x_2}\left(\frac{\partial f}{\partial x_1}\right) \cdots \\ \frac{\partial}{\partial x_1}\left(\frac{\partial f}{\partial x_2}\right), \cdots \\ \vdots \end{bmatrix}$

or $g'(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_2 \partial x_1} & \cdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots \\ \vdots \end{bmatrix}$ known as he Hessian (matrix) of f.

In the example, $g(x) = \begin{bmatrix} 3x^2 + 6x \\ 3y^2 - 6y \end{bmatrix}$, $g'(x) = \begin{bmatrix} 6x+6 & 0 \\ 0 & 6y-6 \end{bmatrix}$

(It's not typical for the Hessian to be diagonal as it is here.)

To save the hassle of computing the gradient and the Hessian of f by hand, in case we want to try other examples, I made this sympy code to generate them for us:

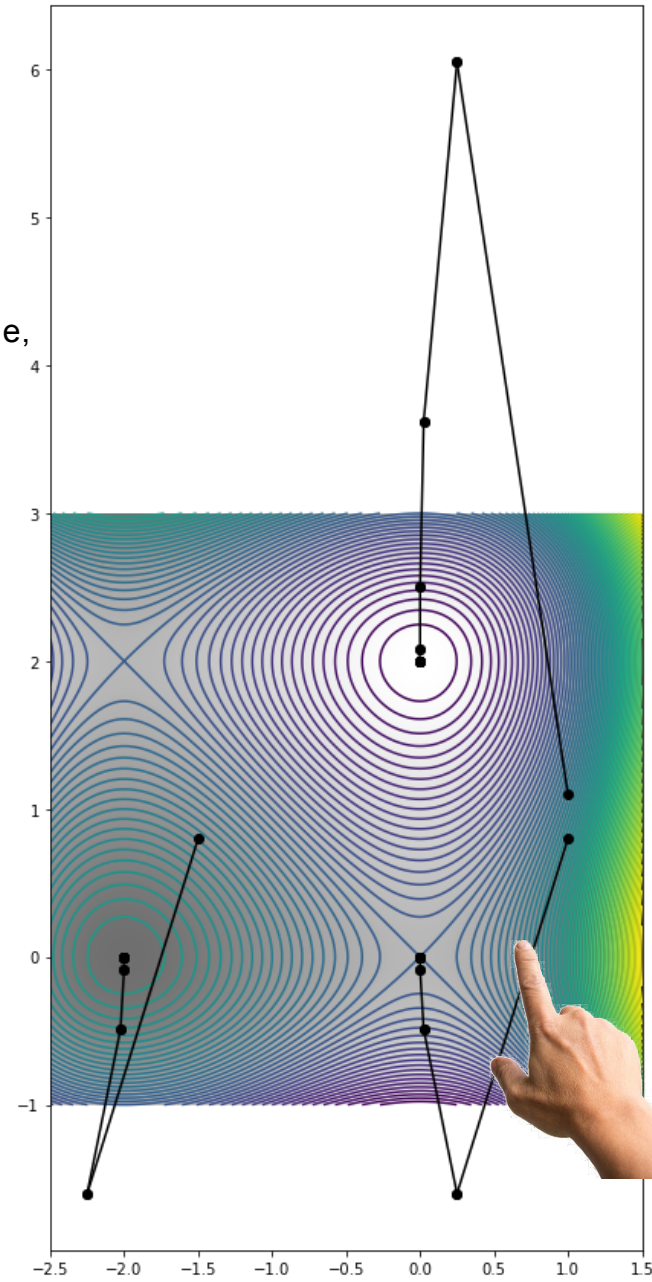
```
import numpy as np

def makeGDG(f):
    # Use sympy to create numpy-friendly functions for grad f and jacobian of grad f
    # Assume that f is a function of 2 variables
    # Still thinking how best to do this.
    from sympy.abc import x,y
    from sympy import diff,lambdify
    fsymbolic =  f((x,y))
    gradfsymbolic = ( diff(fsymbolic,x),diff(fsymbolic,y) )
    gradf = lambdify( (x,y), gradfsymbolic, 'numpy' )
    def g(X):
        x,y = X
        return np.array(gradf(x,y))
    jacgradfsymbolic = [[diff(fsymbolic,x,x),diff(fsymbolic,x,y)],\
                        [diff(fsymbolic,y,x),diff(fsymbolic,y,y)]]
    jacgradf = lambdify( (x,y), jacgradfsymbolic, 'numpy' )
    def Dg(X):
        x,y = X
        return np.array(jacgradf(x,y))
    return g,Dg
```

And a function to draw contours of f, to help us understand what's happening:

```
from pylab import plot,show,xlim,ylim,subplot,contour,imshow,figure,subplot
import matplotlib.cm as cm

def mycontourplot(f,xmin,xmax,ymin,ymax):
    #subplot(111,aspect='equal')
    x = np.linspace(xmin,xmax,300 )
    y = np.linspace(ymin,ymax,300 )
    X, Y = np.meshgrid(x, y)
    Z = f((X,Y))
    imshow( -Z, interpolation='bilinear', origin='lower',
                cmap=cm.gray,extent=(xmin,xmax,ymin,ymax))
    contour( X,Y,f((X,Y)),80 )
```

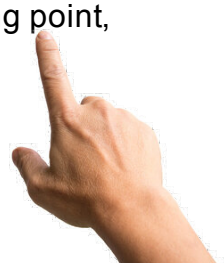
And finally let's apply Newton-on-gradient to our example, with 3 different starting points:

```
def f(X):
    x,y = X
    return x**3 + y**3 + 3*x**2 - 3*y**2 + 12.

G,DG = makeGDG(f)

figure(figsize=(10,15))
subplot(111,aspect=1)
mycontourplot(f,-2.5,1.5,-1.,3.)

tol = 1.e-7

startingpoints = [[1.,0.8],[1.,1.1],[-1.5,0.8]]
for x in startingpoints:
    while True:
        s = np.linalg.solve( DG(x), -G(x) )
        newx = x + s
        plot( [x[0],newx[0]], [x[1],newx[1]], 'ko-' )
        x = newx

        if np.linalg.norm(s) <= tol: break
```



Observe that depending on the starting point, Newton converges here to

- a minimizer,
- a maximizer, and
- a saddle-point.

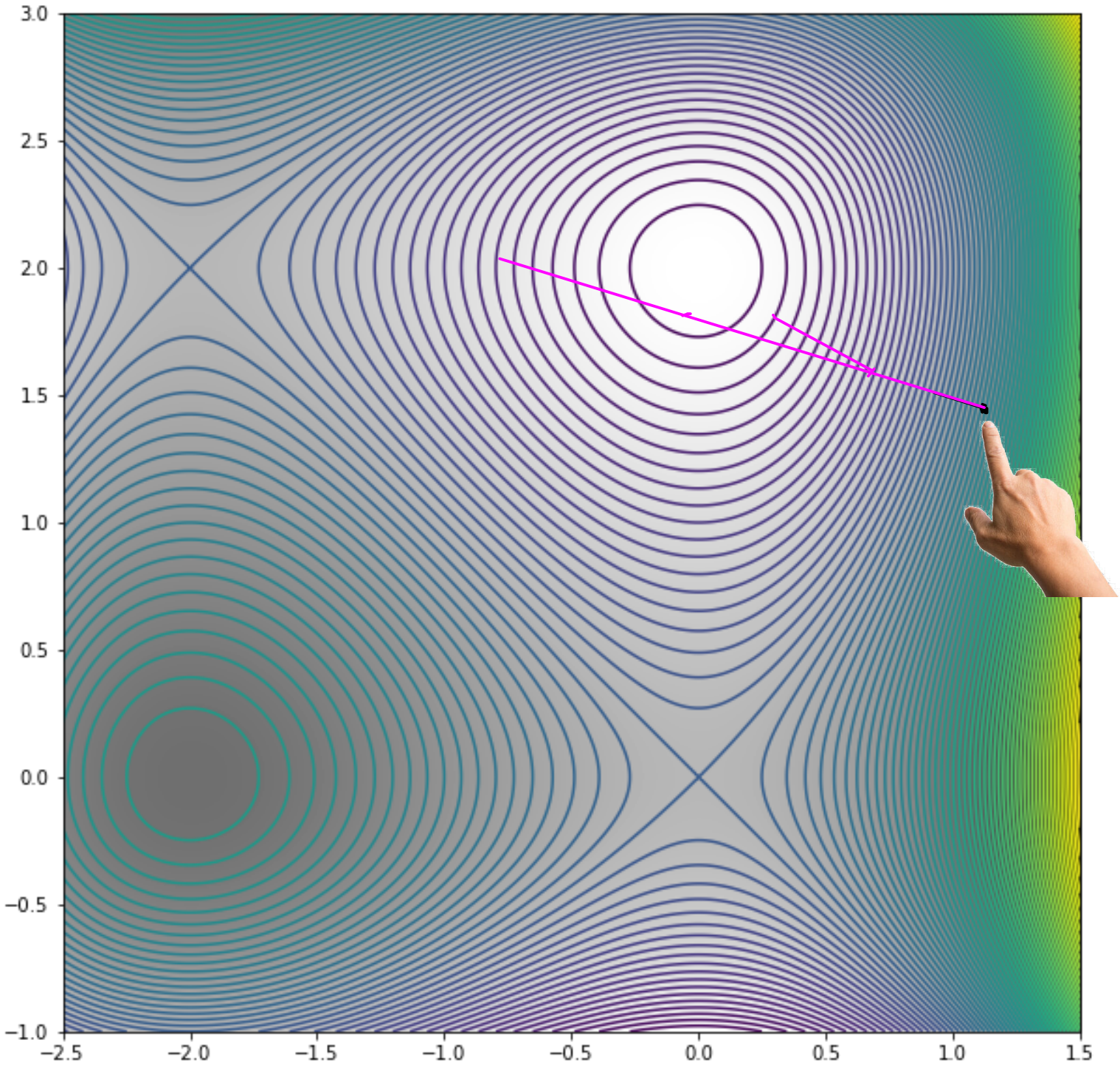


Another way to use the gradient is

Gradient descent

We adjust x in R^n in the direction along which f decreases most rapidly, that is, along the negative gradient.

```
¶
figure(figsize=(10,10))
subplot(111,aspect=1)
mycontourplot(f,-2.5,1.5,-1.,3.)
```



There are multiple possible ways of doing this, such as:

- Do a full 1D minimization along the line through the current point parallel to gradient.
- Take a step of a certain size along the negative gradient.
- Move with instantaneous velocity along the negative gradient, that is, solve the DE x' = -grad f(x).

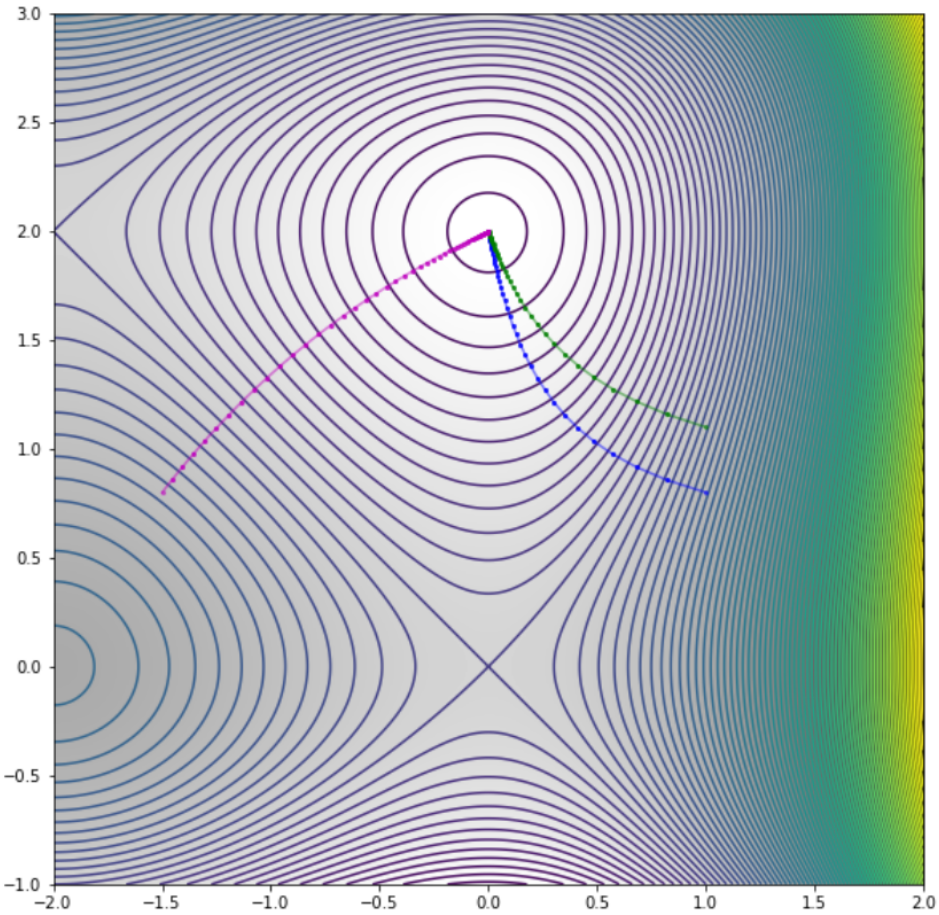
ODE approach

One option to "constantly" follow the local negative gradient - would be to use numerical ODE methods (MTH 538) to solve the system $x'(t) = -\nabla f(x(t))$.

A crude version of this would be to use Euler's method with a fixed small step-size h:

$$x^{(k+1)} = x^{(k)} - h\nabla f(x^{(k)})$$

```
def f(X):
    x,y = X
    return x**3 + y**3 + 3*x**2 - 3*y**2 + 12.

g,Dg = makeGDG(f) # (only g actually needed here)

figure(figsize=(10,10))
mycontourplot(f,-2,2.,-1,3)

startingpoints = ([1.,0.8],[1.,1.1],[-1.5,.5])
startingpoints = [[1.,0.8],[1.,1.1],[-1.5,0.8]]

def linef(t, f,X,V):
    return f( X + t*V )

tol = 1.e-7
h = .02
for x,color in zip(startingpoints,'bgm'):
    while True:
        v = -g(x)
        newx = x + h*v
        plot( [x[0],newx[0]], [x[1],newx[1]], 'o-',color=color,markersize=2,alpha=0.5 )
        x = newx
        if np.linalg.norm(h*v) < tol: break  # stopping criterion needs work
```



Numerical solutions of the DE

x' = - grad f(x)

for 3 different initial conditions.

Successive line minimizations

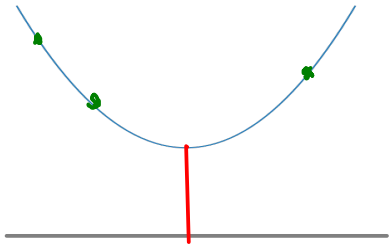
Another is to do successive 1D minimizations along the negative gradient direction:

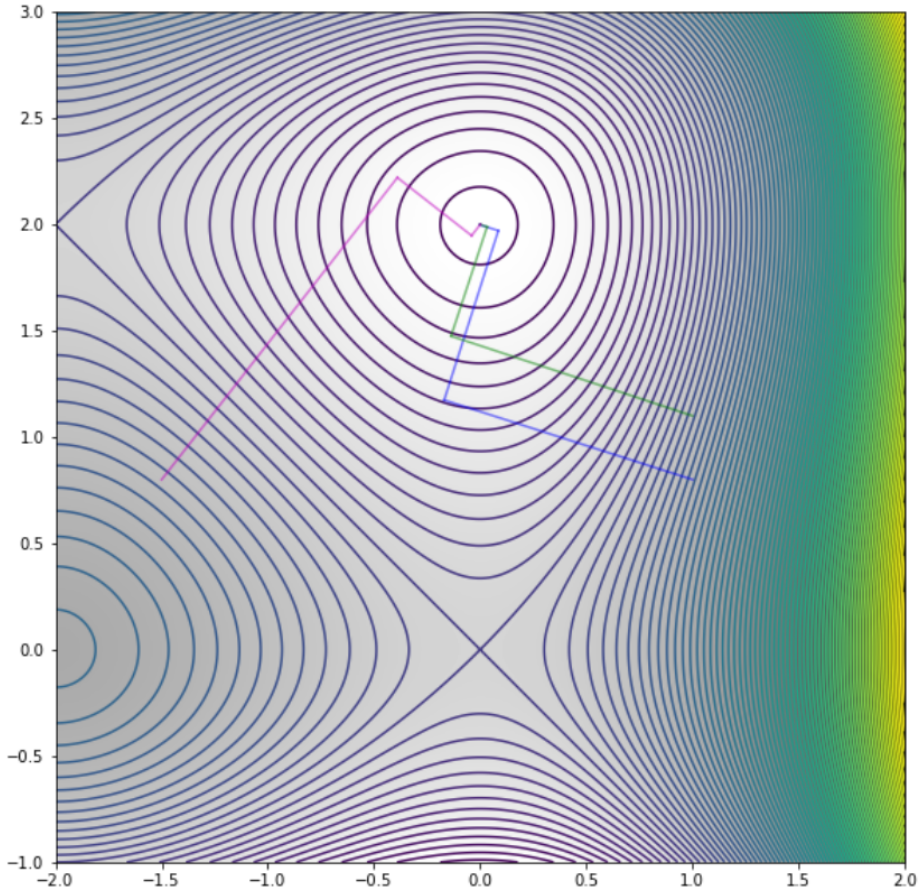
$$x^{(k+1)} = x^{(k)} - t^* \nabla f(x^{(k)})$$

where t^* is a value of t that minimizes $f\left(x^{(k)} - t \nabla f(x^{(k)})\right)$.

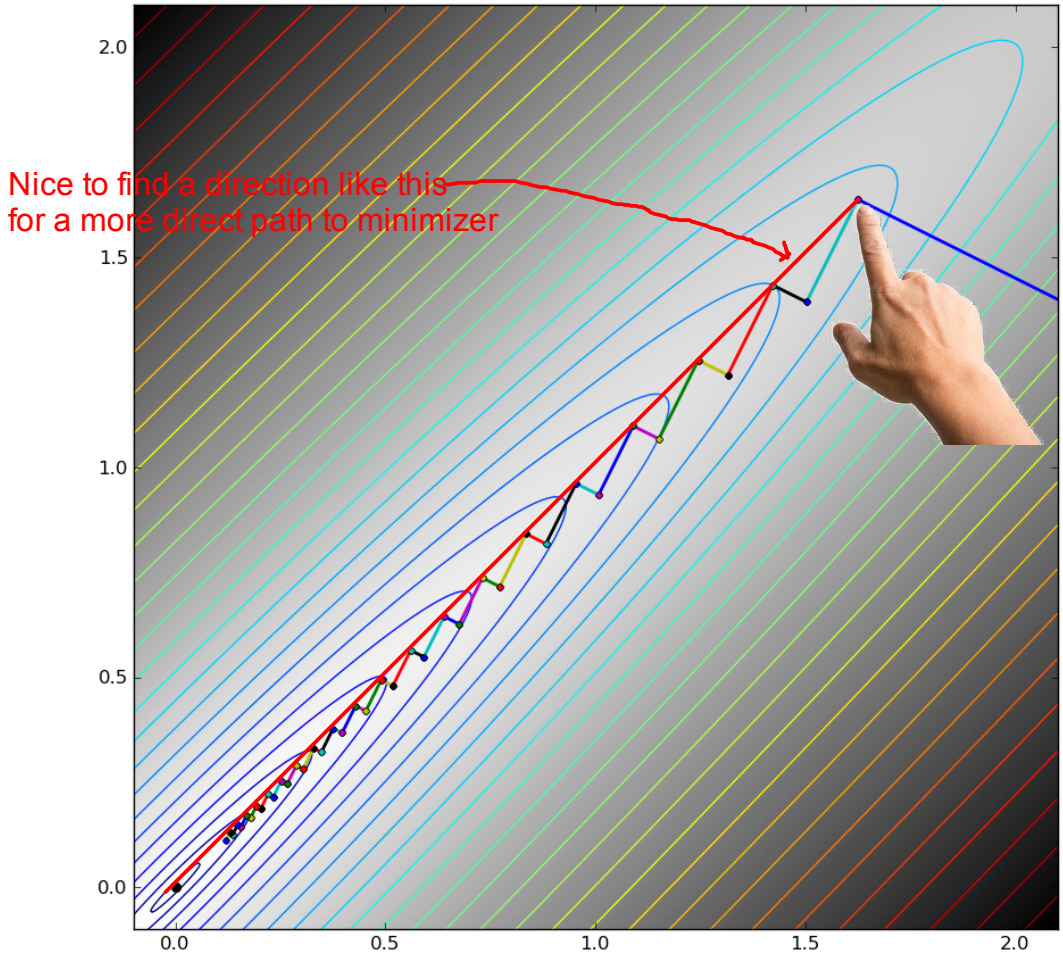
```
def spi(f,r,s,t,tol,*any_parameters_needed_by_f):
    fr = f(r,*any_parameters_needed_by_f)
    fs = f(s,*any_parameters_needed_by_f)
    ft = f(t,*any_parameters_needed_by_f)
    while np.abs(r-t) > tol: # stopping criterion needs work to be robust
        tstar = (r+s)/2 - (fs-fr)*(t-r)*(t-s)/2/( (s-r)*(ft-fs) - (fs-fr)*(t-s) ) # minimizer of interpolating quadratic
        t,s,r = s,r,tstar
        ft,fs,fr = fs,fr,f(tstar,*any_parameters_needed_by_f)
    return tstar
```

```
def myf(X):
    x,y = X
    return x**3 + y**3 + 3*x**2 - 3*y**2 + 12.

g,Dg = makeGDG(myf) # (only g actually needed here)

figure(figsize=(10,10))
mycontourplot(myf,-2,2.,-1,3)

startingpoints = ([1.,0.8],[1.,1.1],[-1.5,.5])
startingpoints = [[1.,0.8],[1.,1.1],[-1.5,0.8]]

def linef(t, f,X,V):
    return myf( X + t*V )

itol = 1.e-7
otol = 1.e-6
for x,color in zip(startingpoints,'bgm'):
    while True:
        v = -g(x)
        v /= np.linalg.norm(v) # scale gradient vector to length 1
        tstar = spi(linef,0,2,1,itol, myf,x,v)  # initial values of line parameter naively 0,1,2
        newx = x + tstar*v
        plot( [x[0],newx[0]], [x[1],newx[1]], color=color,markersize=2,alpha=0.5)
        x = newx
        if np.linalg.norm(tstar*v) < otol: break
```





Note that the right-angle turns in successive line minimization method mean it is not optimal. The inefficiency is clear in the following example: each line minimization "spoils" the progress in the previous one.



The ineffiency of the right-angle turns in successive line minimizations is avoided by the

"conjugate gradient" method

Instead of finding the best local direction for descent, let's find the direction that is best without spoiling the reductions that we've achieved on previous descents.

In particular, let's devise a sequence of n directions for line minimizations that will bring us exactly to the minimizer of a quadratic function in n variables with a minimum, which we can write in this form:

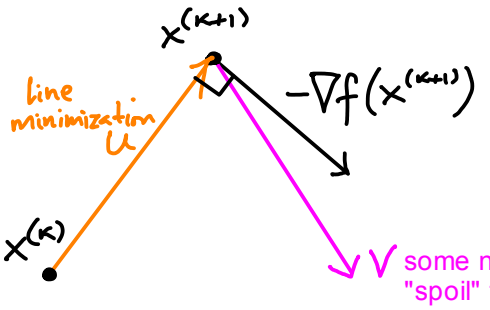
$$f(x) = c - b^T x + \frac{1}{2} x^T A x$$ where w.o.l.o.g. A is symmetric

Example:

$$f(x) = 2 + 4x + 7y + 5x^2 + 13y^2 + 6xy$$

$$= \underbrace{2}_{c} - \underbrace{\begin{bmatrix} -4, -7 \end{bmatrix}}_{b^T} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} x, y \end{bmatrix} \underbrace{\begin{bmatrix} 5 & 3 \\ 3 & 13 \end{bmatrix}}_{\frac{1}{2}A} \begin{bmatrix} x \\ y \end{bmatrix}$$

Suppose we have just done a line minimization from $x^{(k)}$ to $x^{(k+1)}$ along a direction u.



At $x^{(k+1)}$, the directional derivative of f along u is zero:

$$D_u f(x^{(k+1)}) = 0$$

What we'd like is to choose a new descent direction v such that

$$D_v(D_u f) = 0 \quad \text{at } x^{(k+1)}.$$

How does this constrain v explicitly?

Well $D_u f = \nabla f \cdot u$, and for our quadratic f,

$$\nabla f = -b + Ax,$$

so $D_u f(x) = -b^T u + x^T A u \overset{\text{call it}}{=} g(x)$.

We want $D_v g = 0$ at $x^{(k+1)}$.

Now $D_v g(x) \equiv \lim_{h \to 0} \frac{g(x + h\frac{v}{\|v\|}) - g(x)}{h}$ with $g(x) = -b^T u + x^T A u$

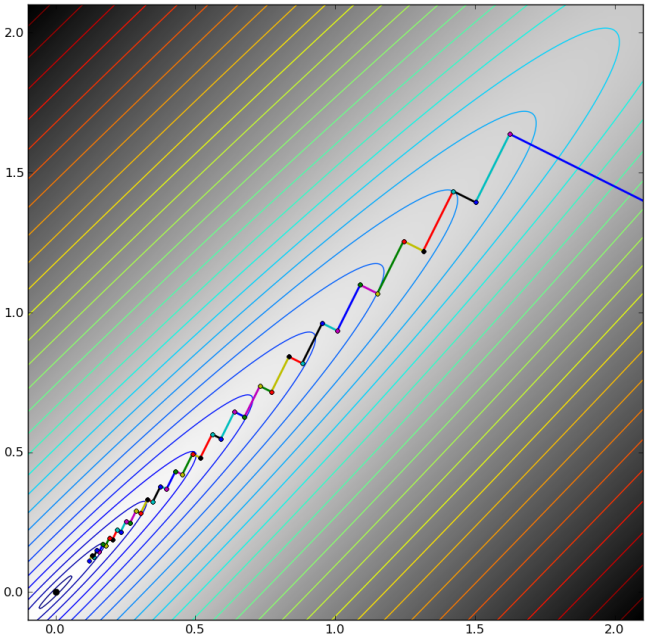
So $D_v(D_u f) = D_v g = \lim_{h \to 0} \left(\frac{-b^T u + (x + h\frac{v}{\|v\|})^T A u - (-b^T u + x^T A u)}{h} \right)$

$$= \lim_{h \to 0} \frac{h v^T A u / \|v\|}{h} = \frac{v^T A u}{\|v\|} \overset{\text{want}}{=} 0.$$

That is, we want $\boxed{v^T A u = 0}$. We say such a v is "conjugate" to u (w.r.t. A).

or v is A-conjugate to u

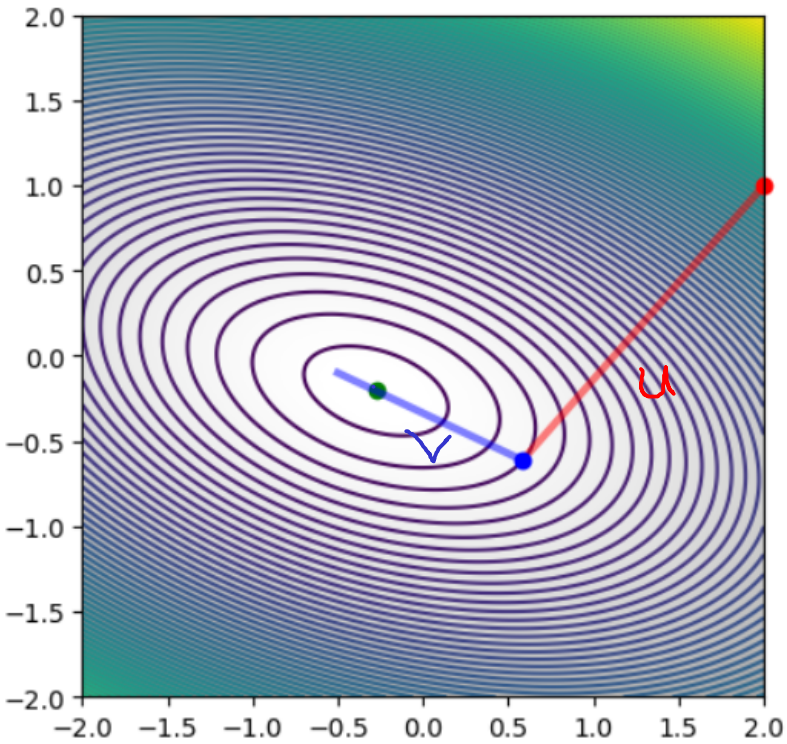
For a quadratic function of n=2 variables, this choice will take us directly to the minimum on our second line minimization.



Actual demo:

```
import numpy as np
import pylab as plt
import matplotlib.cm as cm
def mycontourplot(f,xmin,xmax,ymin,ymax):
    x = np.linspace(xmin,xmax,300 )
    y = np.linspace(ymin,ymax,300 )
    X, Y = np.meshgrid(x, y)
    Z = f((X,Y))
    plt.imshow( -Z, interpolation='bilinear', origin='lower',
                cmap=cm.gray,extent=(xmin,xmax,ymin,ymax))
    plt.contour( X,Y,f((X,Y)),80 )

c = 2
b = np.array([-4,-7])
A = 2*np.array([[5,3],[3,13]])
print('A:',A)
def f1(X):  # scalar x,y version
    x,y = X
    return 2 + 4*x+ 7*y + 5*x**2 + 13*y**2 + 6*x*y
def f(X):
    return c - np.dot(b,X) + np.dot(X, np.dot(A/2,X) )
def gradf(X):
    x,y = X
    return np.array( [4+2*5*x+5*y, 7+2*13*y])
    return -b + np.dot(A,X)

X0 = 2,1
u = gradf(X0)

mycontourplot(f1,-2,2,-2,2)
x0,y0 = X0
plt.plot(x0,y0,'ro',clip_on=False)
print('X0:',X0)
# find the line minimizer along the negative gradient direction from
x,y,t = sp.symbols('x,y,t')
fsym = sp.expand(f1((x0+u[0]*t,y0+u[1]*t)))
#display(fsym)
s = sp.solve( sp.diff(fsym,t), t)[0]
# also symbolically find minimizer of f
fsym = sp.expand(f1((x,y)))
r = sp.solve( (sp.diff(fsym,x),sp.diff(fsym,y)),(x,y))
xr,yr = (r[x],r[y])
plt.plot(xr,yr,'go')

x1 = x0+s*u[0]
y1 = y0+s*u[1]
plt.plot([x0,x1],[y0,y1],'r',lw=3,alpha=.5)
plt.plot(x1,y1,'bo')
print('X1:',x1,y1,'(minimizer on red line)')
Au = np.dot(A,u)
v = np.array([-Au[1],Au[0]])  # perp to Au
print('v:',v,'A-conjugate to u')
v = v/np.linalg.norm(v) # unitize
t = 1.2  # distance to move along v
x2 = x1 + t*v[0]
y2 = y1 + t*v[1]
plt.plot([x1,x2],[y1,y2],'b',lw=3,alpha=0.5);
```

```
X0: (2, 1)
X1: 28121/48208 -29507/48208 (minimizer on red line)
v: [-1032   488] A-conjugate to u
```



For n>2, we choose each "v" to be A-conjugate to all previous search directions,
and we'll reach the minimum after n successive conjugate line minimizations.

Now there are two problems with this:

(i) f is not exactly quadratic

(ii) even if it were, we don't know A.

(i) Can be addressed by iterating the above process, getting much closer on each cycle.

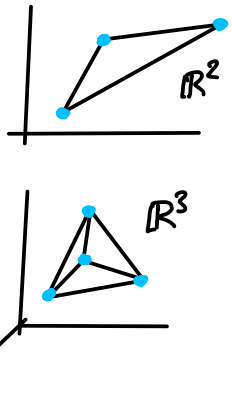
(ii) Amazingly, it turns out we can obtain conjugate directions
just from the f and grad f values we pick up along the way.

If f does not have the smoothness needed for the above methods,
there are "direct search" methods. Think of a muddy lake and a plumb line.
A well-known, though flawed, example is the

Nelder-Mead downhill simplex method

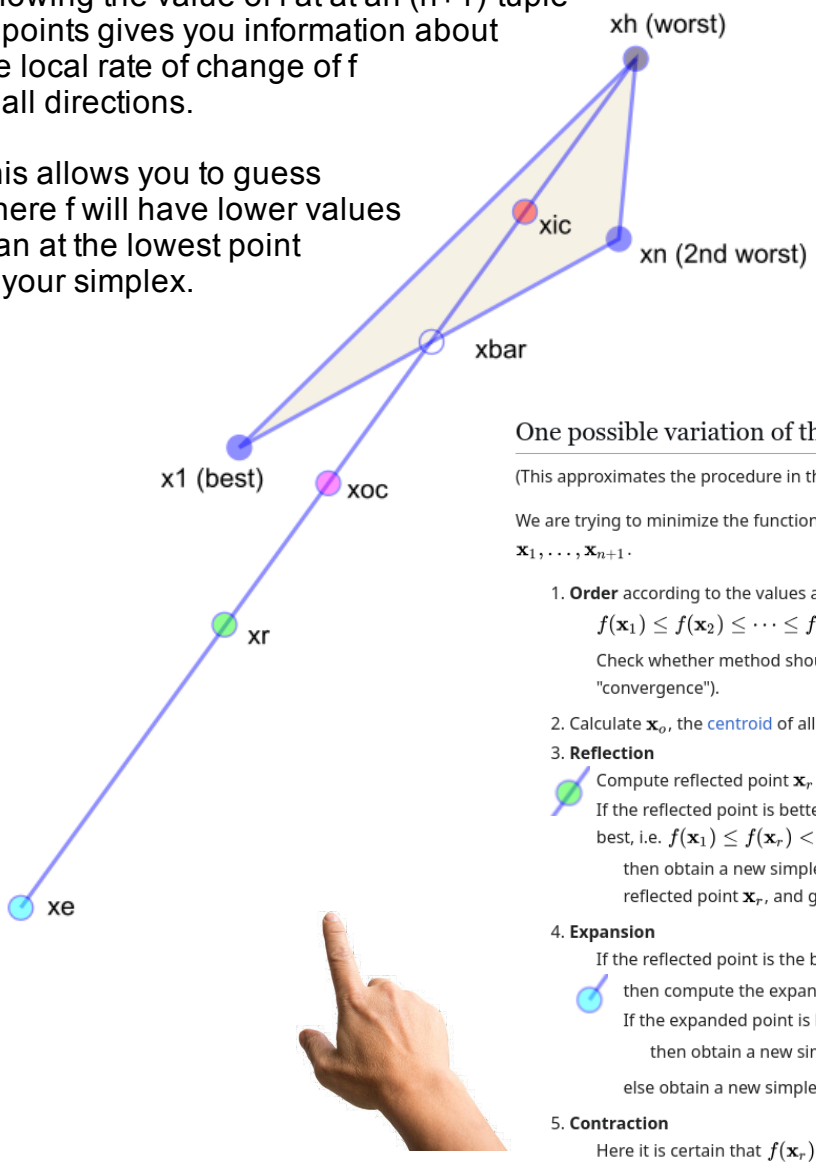
A simplex in R^n is an (n+1)-tuple of points whose convex hull has positive volume.

Example: a triangle in R^2, a tetrahedron in R^3, etc.



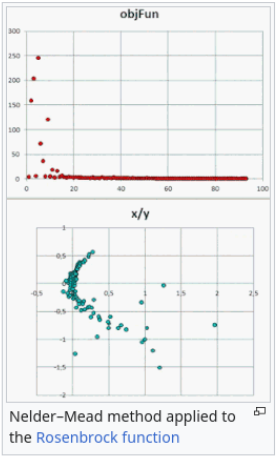
Knowing the value of f at at an (n+1)-tuple of points gives you information about the local rate of change of f in all directions.

This allows you to guess where f will have lower values than at the lowest point of your simplex.



One possible variation of the NM algorithm [edit]

(This approximates the procedure in the original Nelder–Mead article.)



Nelder–Mead method applied to the Rosenbrock function

We are trying to minimize the function $f(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^n$. Our current test points are $\mathbf{x}_1, \ldots, \mathbf{x}_{n+1}$.

1. **Order** according to the values at the vertices:
   $f(\mathbf{x}_1) \leq f(\mathbf{x}_2) \leq \cdots \leq f(\mathbf{x}_{n+1})$.
   Check whether method should stop. See **Termination** (sometimes called "convergence").
2. Calculate $\mathbf{x}_o$, the centroid of all points except $\mathbf{x}_{n+1}$.
3. **Reflection**
   Compute reflected point $\mathbf{x}_r = \mathbf{x}_o + \alpha(\mathbf{x}_o - \mathbf{x}_{n+1})$ with $\alpha > 0$.
   If the reflected point is better than the second worst, but not better than the best, i.e. $f(\mathbf{x}_1) \leq f(\mathbf{x}_r) < f(\mathbf{x}_n)$,
   then obtain a new simplex by replacing the worst point $\mathbf{x}_{n+1}$ with the reflected point $\mathbf{x}_r$, and go to step 1.
4. **Expansion**
   If the reflected point is the best point so far, $f(\mathbf{x}_r) < f(\mathbf{x}_1)$,
   then compute the expanded point $\mathbf{x}_e = \mathbf{x}_o + \gamma(\mathbf{x}_r - \mathbf{x}_o)$ with $\gamma > 1$.
   If the expanded point is better than the reflected point, $f(\mathbf{x}_e) < f(\mathbf{x}_r)$,
   then obtain a new simplex by replacing the worst point $\mathbf{x}_{n+1}$ with the expanded point $\mathbf{x}_e$ and go to step 1;
   else obtain a new simplex by replacing the worst point $\mathbf{x}_{n+1}$ with the reflected point $\mathbf{x}_r$ and go to step 1.
5. **Contraction**
   Here it is certain that $f(\mathbf{x}_r) \geq f(\mathbf{x}_n)$. (Note that $\mathbf{x}_n$ is second or "next" to the worst point.)
   If $f(\mathbf{x}_r) < f(\mathbf{x}_{n+1})$,
   then compute the contracted point on the outside $\mathbf{x}_c = \mathbf{x}_o + \rho(\mathbf{x}_r - \mathbf{x}_o)$ with $0 < \rho \leq 0.5$.
   If the contracted point is better than the reflected point, i.e. $f(\mathbf{x}_c) < f(\mathbf{x}_r)$,
   then obtain a new simplex by replacing the worst point $\mathbf{x}_{n+1}$ with the contracted point $\mathbf{x}_c$ and go to step 1;
   Else go to step 6;
   If $f(\mathbf{x}_r) \geq f(\mathbf{x}_{n+1})$,
   then compute the contracted point on the inside $\mathbf{x}_c = \mathbf{x}_o + \rho(\mathbf{x}_{n+1} - \mathbf{x}_o)$ with $0 < \rho \leq 0.5$.
   If the contracted point is better than the worst point, i.e. $f(\mathbf{x}_c) < f(\mathbf{x}_{n+1})$,
   then obtain a new simplex by replacing the worst point $\mathbf{x}_{n+1}$ with the contracted point $\mathbf{x}_c$ and go to step 1;
   Else go to step 6;
6. **Shrink**
   Replace all points except the best ($\mathbf{x}_1$) with
   $\mathbf{x}_i = \mathbf{x}_1 + \sigma(\mathbf{x}_i - \mathbf{x}_1)$ and go to step 1. i.e. shrink towards the best point

Note: α, γ, ρ and σ are respectively the reflection, expansion, contraction and shrink coefficients. Standard values are $\alpha = 1$, $\gamma = 2$, $\rho = 1/2$ and $\sigma = 1/2$.

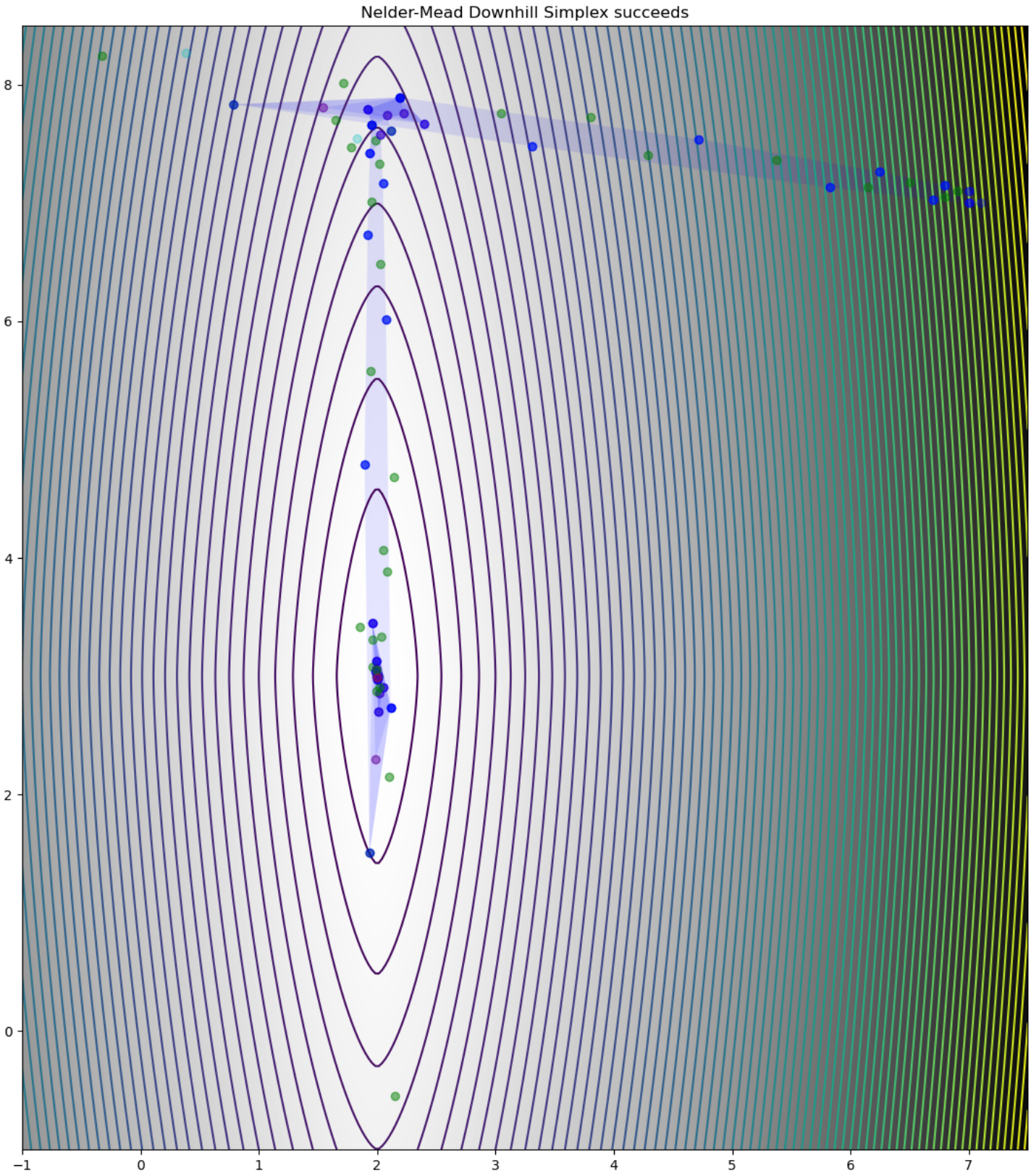
An implementation (translated from Sauer)

```
# Python translation of Sauer 2e neldermead.m by JR

# Program 13.3 Nelder-Mead Search
# Input: inline function f, best guess xbar (column vector),
#    initial search radius rad and number of steps k
# Output: matrix x whose columns are vertices of simplex,
#    function values y of those vertices
# Translation to Python of neldermead.m
import numpy as np
from time import sleep
import matplotlib.cm as cm
import pylab as plt
from os.path import exists
from os import makedirs
import sys

def mycontourplot(f,xmin,xmax,ymin,ymax):
        #subplot(111,aspect='equal')
        x = np.linspace(xmin,xmax,300 )
        y = np.linspace(ymin,ymax,300 )
        X, Y = np.meshgrid(x, y)
        #Z = sqrt( f((X,Y)) )
        Z = f((X,Y))
        plt.imshow( -Z, interpolation='bilinear', origin='lower',
                    cmap=cm.gray,extent=(xmin,xmax,ymin,ymax))
        #contour( X,Y,f((X,Y))**0.33,30 )
        plt.contour( X,Y,f((X,Y)),80 )

def neldermead1(f,x0,rad,k):
        print(x0)
        n = len(x0[0])-1
        x = np.empty((n,n+1),dtype=float)  # initial simplex provided as x0
        x[:,:] = x0
        y = np.empty(n+1)
        for j in range(n+1):
                y[j] = f( x[:,j] )   # evaluate obj function f at each vertex
        oy = np.argsort(y)                      # sort the function values in ascending order
        y = y[oy]
        x = x[:,oy]                                     # and rank the vertices the same way
        #print x
        #print x[0,:]+[x[0,0]],x[1,:]+[x[1,0]]
        #quit()
        for i in range(k):
                if do_dots:
                        plt.plot(x[0,:],x[1,:],'bo',alpha=0.3)
                        plt.fill(x[0,:],x[1,:],'b',alpha=0.1)
                xbar = np.mean( x[:,0:n], axis=1)       # xbar is the centroid of the face
                xh = x[:,n].copy()                                      # omitting the worst vertex xh
                xr = 2*xbar - xh; yr = f(xr)
                if do_dots: plt.plot(xr[0],xr[1],'co',alpha=0.5)
                if yr < y[n-1]:
                        if yr < y[0]:           # try expansion xe
                                xe = 3*xbar - 2*xh; ye = f(xe)
                                if do_dots: plt.plot(xe[0],xe[1],'go',alpha=0.3)
                                if ye < yr:             # accept expansion
                                        x[:,n] = xe; y[n] = f(xe)
                                else:                                   # accept reflection
                                        x[:,n] = xr; y[n] = f(xr)
                        else:                                           # xr is middle of pack, accept reflection
                                x[:,n] = xr; y[n] = f(xr)
                else:                                                   # xr is still the worst vertex, contract
                        if yr < y[n]:           # try outside contraction xoc
                                xoc = 1.5*xbar - 0.5*xh; yoc = f(xoc)
                                if do_dots: plt.plot(xoc[0],xoc[1],'mo',alpha=0.3)
                                if yoc < yr:    # accept outside contraction
                                        x[:,n] = xoc; y[n] = f(xoc)
                                else:                                   # shrink simplex toward best point
                                        for j in range(1,n+1):
                                                x[:,j] = 0.5*x[:,0] + 0.5*x[:,j]; y[j] = f(x[:,j])
                        else:                                           # xr is even worse than the previous worst
                                xic = 0.5*xbar + 0.5*xh; yic = f(xic)
                                if do_dots: plt.plot(xic[0],xic[1],'ro',alpha=0.3)
                                if yic < y[n]:  # accept inside contraction
                                        x[:,n] = xic; y[n] = yic
                                else:                                   # shrink simplex toward best point
                                        for j in range(1,n+1):
                                                x[:,j] = 0.5*x[:,0]+0.5*x[:,j]; y[j] = f(x[:,j])
                oy = np.argsort(y)              # re-sort the function values in ascending order
                y = y[oy]
                x = x[:,oy]                                     # and rank the vertices the same way
                if do_triangles: plt.plot(hstack((x[0,:],[x[0,0]])), hstack((x[1,:],[x[1,0]])),'b')
                p.set_xdata( np.hstack((x[0,:],[x[0,0]])) )
                p.set_ydata( np.hstack((x[1,:],[x[1,0]])) )
                plt.draw()
                plt.savefig(foldername +'/neldermead_anim'+str(i+1).zfill(3)+'.png')
                plt.pause(.0001)
                sleep(.5)
                if do_wait: input()

        return x
```

Application to the examples I'll show you below

```
McKinnon = False

if not McKinnon:
        def f(x):
                xx,yy = x
                p = 1.5
                return np.abs(xx-2.)**p + .1*abs(yy-3.)**p
                x0 = np.array([[0,0],[1,1],[l1,l2]]).T  # initial simplex
        x0 = np.array([[7,7],[7.1,7],[7,7.1]]).T  # initial simplex
        nsteps = 36
        xlo,xhi,ylo,yhi = -1,7.5,-1,8.5
        foldername = 'narrow_lake'
        title = 'Nelder-Mead Downhill Simplex succeeds'

else:
        def f(X):  # See McKinnon 1998 for functions problematic for NM:  p152
                tau,theta,phi = 2,6,60
                #assert 0
                x,y = X
                try:
                        fval =  theta*phi*np.abs(x)**tau + y + y**2
                        xpos = x>0
                        fval[xpos] = theta*x[xpos]**tau + y[xpos] + y[xpos]**2
                except:  # argument must be scalar
                        fval =  theta*phi*np.abs(x)**tau + y + y**2
                        if x>0:
                                fval = theta*x**tau + y + y**2
                return fval
        l1 = (1+np.sqrt(33))/8
        l2 = (1-np.sqrt(33))/8
        x0 = np.array([[0,0],[1,1],[l1,l2]]).T  # initial simplex
        nsteps = 15
        xlo,xhi,ylo,yhi = -0.5,2,-1.7,1.4
        foldername = 'mckinnon_1'
        title = 'Nelder-Mead Downhill Simplex fails on McKinnon example'

#foldername = sys.argv[1]
if not exists(foldername): makedirs(foldername)
plt.figure(figsize=(15,15))
plt.subplot(111,aspect=1)
do_dots      = True
do_triangles = False
do_wait      = False #True
mycontourplot(f,xlo,xhi,ylo,yhi)
plt.title(title)
plt.ion()
#x = array([7,7],dtype=float)
#print(x0)
print( x0[0], x0[1] )
plt.fill( x0[0], x0[1],'y', alpha=.2)
p, = plt.plot( x0[0], x0[1],'r')
plt.savefig(foldername +'/neldermead_anim'+str(0).zfill(3)+'.png')

neldermead1(f,x0,0.1,nsteps)
plt.ioff()
plt.show()
```

Example where Nelder-Mead is successful



Individual frames available at: https://https://blue.math.buffalo.edu/537_f25/narrow_valley/

An example where Nelder-Mead fails ...

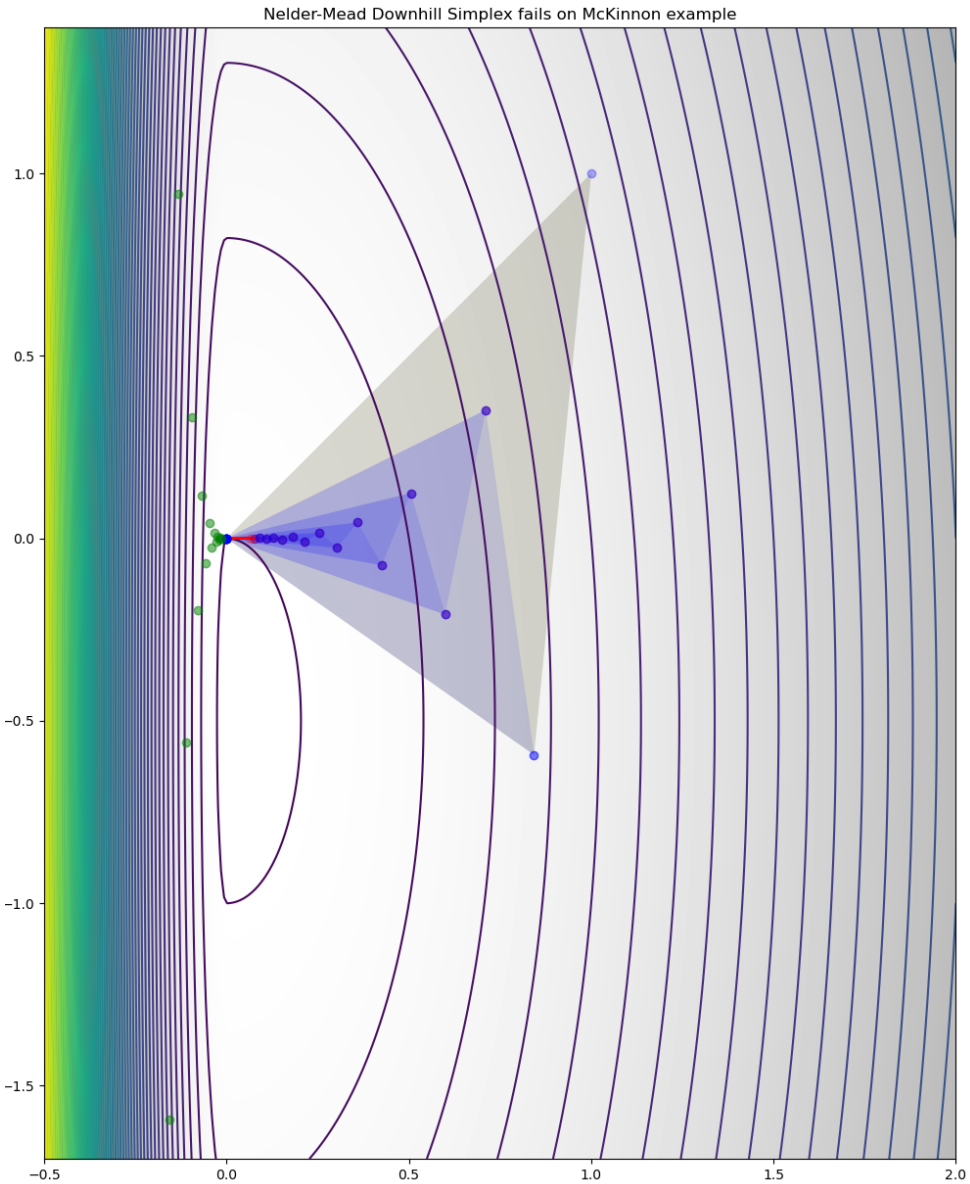
Simplex is asymptotic to a point where gradient is non-zero. BAD!

SIAM J. OPTIM.
Vol. 9, No. 1, pp. 148–158

CONVERGENCE OF THE NELDER–MEAD SIMPLEX METHOD TO A NONSTATIONARY POINT*

K. I. M. MCKINNON†

Abstract. This paper analyzes the behavior of the Nelder–Mead simplex method for a family of examples which cause the method to converge to a nonstationary point. All the examples use continuous functions of two variables. The family of functions contains strictly convex functions with up to three continuous derivatives. In all the examples the method repeatedly applies the inside contraction step with the best vertex remaining fixed. The simplices tend to a straight line which is orthogonal to the steepest descent direction. It is shown that this behavior cannot occur for functions with more than three continuous derivatives. The stability of the examples is analyzed.



Individual frames available at: https://https://blue.math.buffalo.edu/537_f25/mckinnon_1/

See "Optimization by Direct Search: New Perspectives on Some Classical and Modern Methods" by Kolda et al., SIAM Review Vol. 45, No. 3, pp. 385-482, 2003.