Lec 12 - Numerical optimization

Statistical Computing and Computation

Sta 663 | Spring 2022

Dr. Colin Rundel

Numerical optimization - line search

Today we will be discussing one particular approach for numerical optimization - line search. This is a family of algorithmic approaches that attempt to find (global or local) minima via iteration on an initial guess. Generally they are an attempt to solve,

$$\min_{\alpha > 0} f(x_k + \alpha p_k)$$

where $f()$ is the function we are attempting to minimize, x_k is our current guess at iteration k and α is the step length and p_k is the direction of movement.

We will only be dipping our toes in the water of this area but the goal is to provide some context for some of the more common (and easier) use cases. With that in mind, we will be looking at methods for smooth functions (2nd derivative exists and is continuous).

Naive Gradient Descent

We will start with a naive approach to gradient descent where we choose a fixed step size and determine the direction based on the gradient of the function at each iteration.

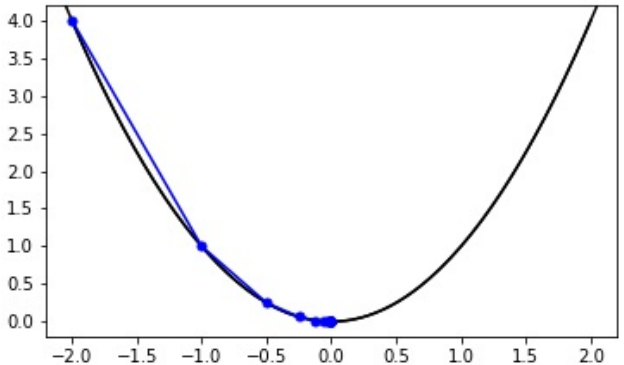
```
def grad_desc_1d(x0, f, grad, step, max_step=100, tol = 1e-6):
  all_x_i = [x0]
  all_f_i = [f(x0)]

  x_i = x0

  try:
    for i in range(max_step):
      dx_i = grad(x_i)
      x_i = x_i - dx_i * step
      f_x_i = f(x_i)

      all_x_i.append(x_i)
      all_f_i.append(f_x_i)

      if np.abs(dx_i) < tol:
        break

  except OverflowError as err:
    print(f"{type(err).__name__}: {err}")
```

A basic example

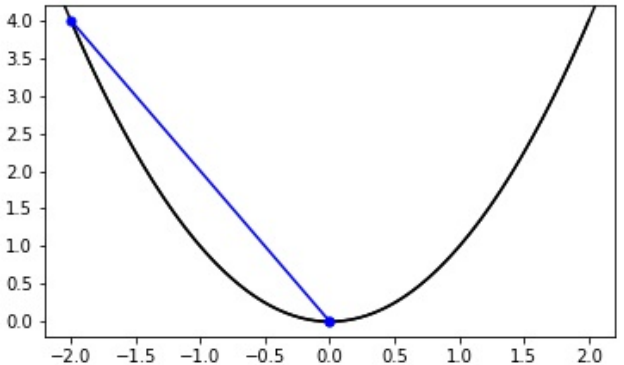
$$f(x) = x^2$$
$$\nabla f(x) = 2x$$

```
f = lambda x: x**2
grad = lambda x: 2*x
```

```
opt = grad_desc_1d(-2., f, grad, step=0.25)
plot_1d_traj( (-2, 2), f, opt )
```

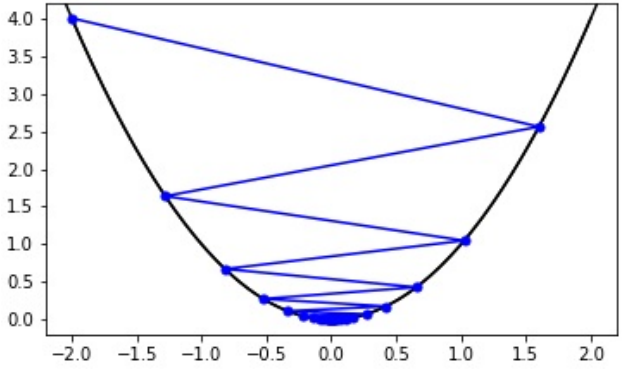



```
opt = grad_desc_1d(-2, f, grad, step=0.5)
plot_1d_traj( (-2, 2), f, opt )
```



Where can it go wrong?

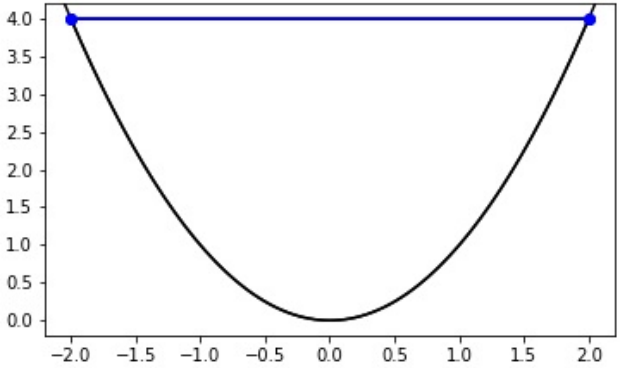
```
opt = grad_desc_1d(-2, f, grad, step=0.9)
plot_1d_traj( (-2,2), f, opt )
```




```
opt = grad_desc_1d(-2, f, grad, step=1)
```

```
## Warning - Failed to converge!
```

```
plot_1d_traj( (-2,2), f, opt )
```

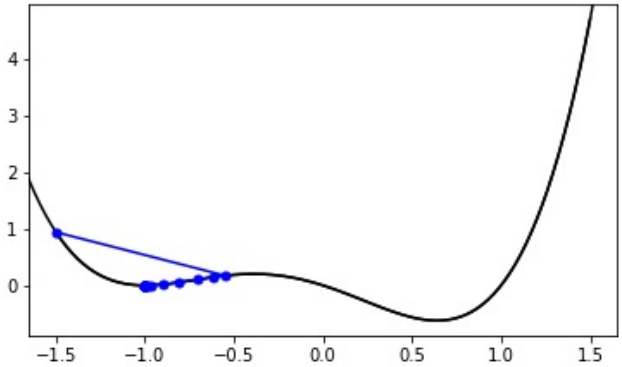


Local minima of a quartic

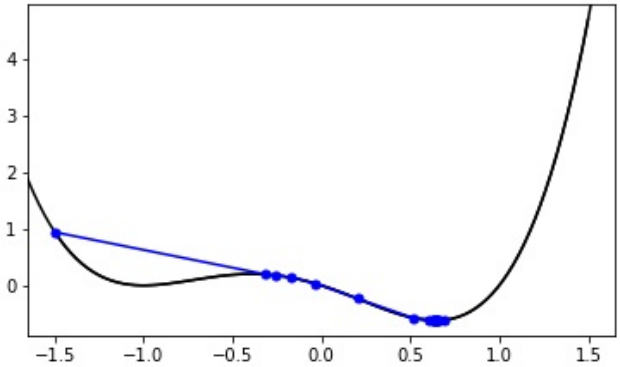
$$f(x) = x^4 + x^3 - x^2 - x$$
$$\nabla f(x) = 4x^3 + 3x^2 - 2x - 1$$

```
f = lambda x: x**4 + x**3 - x**2 - x
grad = lambda x: 4*x**3 + 3*x**2 - 2*x - 1
```

```
opt = grad_desc_1d(-1.5, f, grad, step=0.2)
plot_1d_traj( (-1.5, 1.5), f, opt )
```

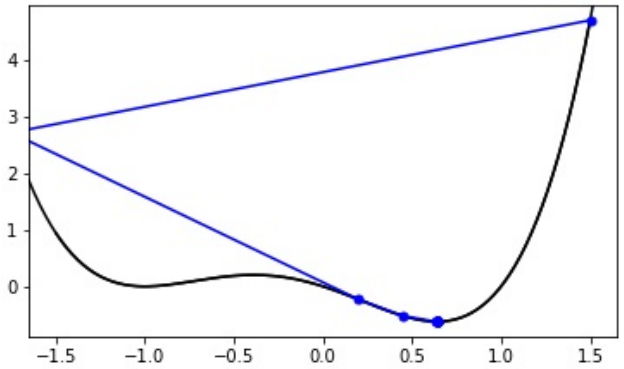



```
opt = grad_desc_1d(-1.5, f, grad, step=0.25)
plot_1d_traj( (-1.5, 1.5), f, opt)
```

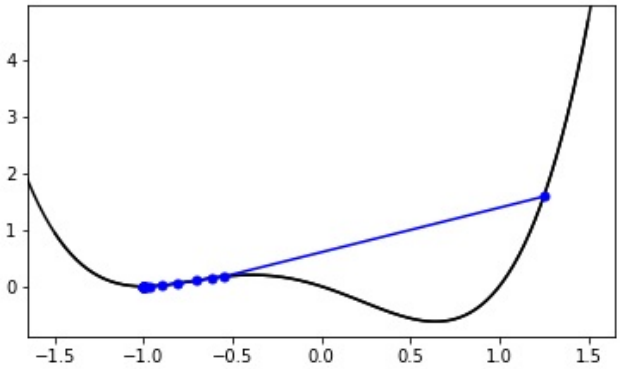


Alternative starting points

```
opt = grad_desc_1d(1.5, f, grad, step=0.2)
plot_1d_traj( (-1.5, 1.5), f, opt )
```




```
opt = grad_desc_1d(1.25, f, grad, step=0.2)
plot_1d_traj( (-1.5, 1.5), f, opt)
```



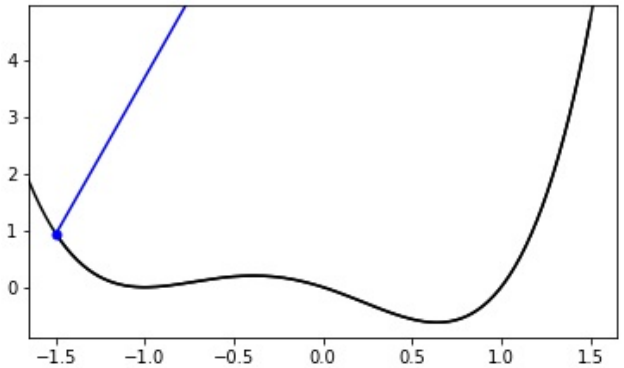
Problematic step sizes

If the step size is too large it is possible for the algorithm to

```
opt = grad_desc_1d(-1.5, f, grad, step=0.75)
```

```
## OverflowError: (34, 'Result too large')
```

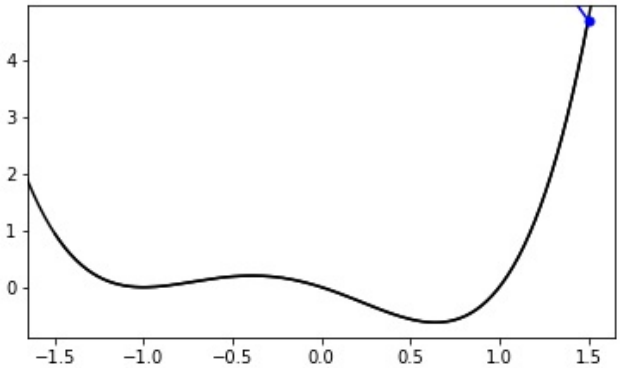
```
plot_1d_traj( (-1.5, 1.5), f, opt )
```




```
opt = grad_desc_1d(1.5, f, grad, step=0.25)
```

```
## OverflowError: (34, 'Result too large')
```

```
plot_1d_traj( (-1.5, 1.5), f, opt)
```



Gradient Descent w/ backtracking

As we have just seen having too large of a ste
be problematic, one solution is to allow the s
to adapt.

Backtracking involves checking if the propos
advantageous (i.e. $f(x_k + \alpha p_k) < f(x_k)$),

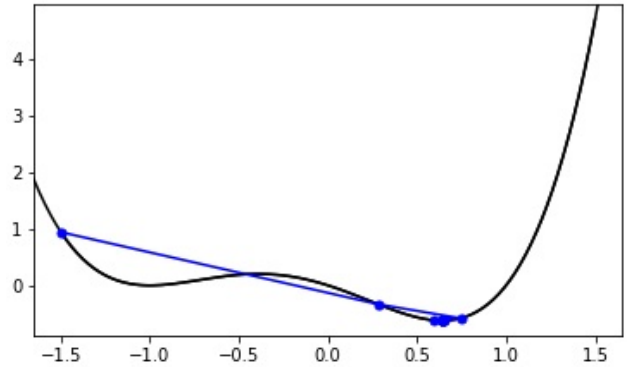
- If it is advantageous then accept $x_{k+1} = x_k + \alpha p_k$.
- If not, shrink α by a factor τ (e.g. 0.5) and check again.

Pick larger α to start as this will not fix inefficiency of small step size.

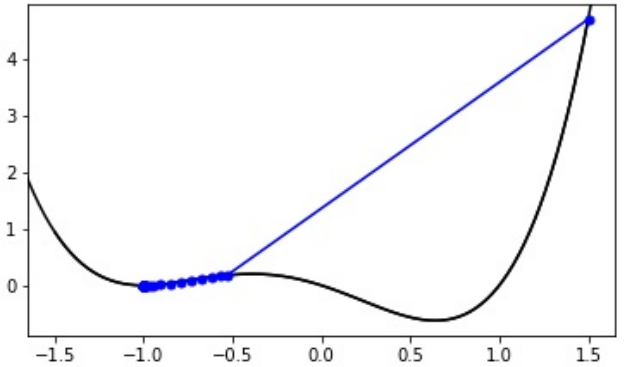
This is a hand-wavy version of the Armijo-Goldstein condition.
Check $f(x_k - \alpha \nabla f(x_k)) \leq f(x_k) - c\alpha(\nabla f(x_k))^2$.

```
def grad_desc_1d_bt(x, f, grad, step, tau=0.5, max_step=100, ma
  all_x_i = [x]
  all_f_i = [f(x)]

  try:
    for i in range(max_step):
      dx = grad(x)

      for j in range(max_back):
        new_x = x + step * (-dx)
        new_f_x = f(new_x)

        if (new_f_x < all_f_i[-1]):
          break

        step = step * tau

      x = new_x
      f_x = new_f_x

      all_x_i.append(x)
      all_f_i.append(f_x)

      if np.abs(dx) < tol:
        break

  except OverflowError as err:
    print(f"{type(err).__name__}: {err}")

  if len(all_x_i) == max_step+1:
```

```
opt = grad_desc_1d_bt(-1.5, f, grad, step=0.75, t
plot_1d_traj( (-1.5, 1.5), f, opt )
```




```
opt = grad_desc_1d_bt(1.5, f, grad, step=0.25, ta
plot_1d_traj( (-1.5, 1.5), f, opt)
```



A 2d cost function

We will be using `mk_quad()` to create quadratic functions with varying conditioning (as specified by the `epsilon` parameter).

$$f(x, y) = 0.33(x^2 + \epsilon^2 y^2)$$

$$\nabla f(x, y) = \begin{bmatrix} 0.66\, x \\ 0.66\, \epsilon^2\, y \end{bmatrix}$$

$$\nabla^2 f(x, y) = \begin{bmatrix} 0.66 & 0 \\ 0 & 0.66\, \epsilon^2 \end{bmatrix}$$

Examples

```
f, grad, hess = mk_quad(0.7)
plot_2d_traj((-1,2), (-1,2), f, title="$\\epsilon
```

```
f, grad, hess = mk_quad(0.02)
plot_2d_traj((-1,2), (-1,2), f, title="$\\epsilon
```

2d gradient descent w/ backtracking

```
def grad_desc_2d(x0, f, grad, step, tau=0.5, max_step=100, max_back=10, tol = 1e-6):
  x_i = x0
  all_x_i = [x_i[0]]
  all_y_i = [x_i[1]]
  all_f_i = [f(x_i)]

  for i in range(max_step):
    dx_i = grad(x_i)

    for j in range(max_back):
      new_x_i = x_i - dx_i * step
      new_f_i = f(new_x_i)

      if (new_f_i < all_f_i[-1]):
        break

      step = step * tau

    x_i, f_i = new_x_i, new_f_i

    all_x_i.append(x_i[0])
    all_y_i.append(x_i[1])
    all_f_i.append(f_i)

    if np.sqrt(np.sum(dx_i**2)) < tol:
```

Well conditioned cost function

```
f, grad, hess = mk_quad(0.7)
opt = grad_desc_2d((1.6, 1.1), f, grad, step=1)
plot_2d_traj((-1,2), (-1,2), f, title="$\\epsilon
```

```
f, grad, hess = mk_quad(0.7)
opt = grad_desc_2d((1.6, 1.1), f, grad, step=2)
plot_2d_traj((-1,2), (-1,2), f, title="$\\epsilon
```

Ill-conditioned cost function

```
f, grad, hess = mk_quad(0.02)
opt = grad_desc_2d((1.6, 1.1), f, grad, step=1)
plot_2d_traj((-1,2), (-1,2), f, title="$\\epsilon
```

```
f, grad, hess = mk_quad(0.02)
opt = grad_desc_2d((1.6, 1.1), f, grad, step=2)
plot_2d_traj((-1,2), (-1,2), f, title="$\\epsilon
```

Rosenbrock function (very ill conditioned)

```
f, grad, hess = mk_rosenbrock()
opt = grad_desc_2d((1.6, 1.1), f, grad, step=0.25
plot_2d_traj((-2,2), (-2,2), f, traj=opt)
```

```
f, grad, hess = mk_rosenbrock()
opt = grad_desc_2d((-0.5, 0), f, grad, step=0.25)
plot_2d_traj((-2,2), (-2,2), f, traj=opt)
```

Taylor Expansion

For any arbitary smooth function, we can construct a 2nd order taylor approximation as follows,

$$
\begin{aligned}
f(x_k + \alpha p_k) &= f(x_k) + \alpha p_k^T \nabla f(x_k + \alpha p_k) \\
&= f(x_k) + \alpha p_k^T \nabla f(x_k) + \frac{1}{2} \alpha^2 p_k^T \nabla^2 f(x_k + \alpha p_k)\, p_k \\
&\approx f(x_k) + \alpha p_k^T \nabla f(x_k) + \frac{1}{2} \alpha^2 p_k^T \nabla^2 f(x_k)\, p_k
\end{aligned}
$$

Newton's Method in 1d

Lets simplify things for now and consider just the 1d case and write αp_k as Δ,

$$f(x_k + \Delta) \approx f(x_k) + \Delta f'(x_k) + \frac{1}{2}\Delta^2 f''(x_k)$$

to find the Δ that minimizes this function we can take a derivative with regard to Δ and set the equation equal to zero which gives,

$$0 = f'(x_k) + \Delta f''(x_k) \quad \Rightarrow \quad \Delta = -\frac{f'(x_k)}{f''(x_k)}$$

which then suggests an iterative update rule of

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

Generalizing to *n*d

Based on the same argument we can see the follow result for a function in R^n,

$$f(x_k + \Delta) \approx f(x_k) + \Delta^T \nabla f(x_k) + \frac{1}{2}\Delta^T \nabla^2 f(x_k)\,\Delta$$

$$0 = \nabla f(x_k) + \nabla^2 f(x_k)\,\Delta \quad \Rightarrow \quad \Delta = -\left(\nabla^2 f(x_k)\right)^{-1} \nabla f(x_k) f(x_k)$$

which then suggests an iterative update rule of

$$x_{k+1} = x_k - (\nabla^2 f(x_k))^{-1} \nabla f(x_k)$$

```
def newtons_method(x0, f, grad, hess, max_iter=100, max_back=10, tol=1e-8):
    all_x_i = [x0[0]]
    all_y_i = [x0[1]]
    all_f_i = [f(x0)]

    x_i = x0

    for i in range(max_iter):
      g_i = grad(x_i)
      step = - np.linalg.solve(hess(x_i), g_i)

      for j in range(max_back):
        new_x_i = x_i + step
        new_f_i = f(new_x_i)

        if (new_f_i < all_f_i[-1]):
          break

        step /= 2

      x_i, f_i = new_x_i, new_f_i

      all_x_i.append(x_i[0])
      all_y_i.append(x_i[1])
      all_f_i.append(f_i)

      if np.sqrt(np.sum(g_i**2)) < tol:
        break
```

Well conditioned quadratic cost function

```
f, grad, hess = mk_quad(0.7)
opt = newtons_method((1.6, 1.1), f, grad, hess)
plot_2d_traj((-1,2), (-1,2), f, traj=opt)
```

```
f, grad, hess = mk_quad(0.02)
opt = newtons_method((1.6, 1.1), f, grad, hess)
plot_2d_traj((-1,2), (-1,2), f, traj=opt)
```

Rosenbrock function (very ill conditioned)

```
f, grad, hess = mk_rosenbrock()
opt = newtons_method((1.6, 1.1), f, grad, hess)
plot_2d_traj((-2,2), (-2,2), f, traj=opt)
```

```
f, grad, hess = mk_rosenbrock()
opt = newtons_method((-0.5, 0), f, grad, hess)
plot_2d_traj((-2,2), (-2,2), f, traj=opt)
```

Conjugate gradients

This is a general approach for solving a system of linear equations with the form $Ax = b$ where A is an $n \times n$ symmetric positive definite matrix and b is $n \times 1$ with x unknown.

This type of problem can also be expressed as a quadratic minimization problems of the form,

$$\min_x f(x) = \frac{1}{2} x^T A x - b^T x + c$$

The goal is then to find n conjugate vectors ($p_i^T A p_j = 0$ for all $i \neq j$) and their coefficients such that

$$x_* = \sum_{i=1}^{n} \alpha_i p_i$$

Conjugate gradient algorithm

Given x_0 we set the following initial values,

$$r_0 = \nabla f(x_0)$$
$$p_0 = -r_0$$
$$k = 0$$

while $\|r_k\|_2 > \text{tol}$,

From Chapter 5.1 of Numerical Optimization 2006

```
def conjugate_gradient(x0, f, grad, hess, max_ite
    all_x_i = [x0[0]]
    all_y_i = [x0[1]]
    all_f_i = [f(x0)]

    x_i = x0
    r_i = grad(x0)
    p_i = -r_i

    for i in range(max_iter):
      a_i = - r_i.T @ p_i / (p_i.T @ hess(x_i) @
      x_i_new = x_i + a_i * p_i
      r_i_new = grad(x_i_new)
      b_i = (r_i_new.T @ hess(x_i) @ p_i) / (p_i.
      p_i_new = -r_i_new + b_i * p_i

      x_i, r_i, p_i = x_i_new, r_i_new, p_i_new

      all_x_i.append(x_i[0])
      all_y_i.append(x_i[1])
      all_f_i.append(f(x_i))

      if np.sqrt(np.sum(r_i_new**2)) < tol:
        break
```

Trajectory

```
f, grad, hess = mk_quad(0.7)
opt = conjugate_gradient((1.6, 1.1), f, grad, hes
plot_2d_traj((-1,2), (-1,2), f, title="$\\epsilon
```

```
f, grad, hess = mk_quad(0.02)
opt = conjugate_gradient((1.6, 1.1), f, grad, hes
plot_2d_traj((-1,2), (-1,2), f, title="$\\epsilon
```

Rosenbrock's function

```
f, grad, hess = mk_rosenbrock()
opt = conjugate_gradient((1.6, 1.1), f, grad, hes
plot_2d_traj((-2,2), (-2,2), f, traj=opt)
```

```
f, grad, hess = mk_rosenbrock()
opt = conjugate_gradient((-0.5, 0), f, grad, hess
plot_2d_traj((-2,2), (-2,2), f, traj=opt)
```

CG in scipy

Scipy's optimize module implements the conjugate gradient algorithm by Polak and Ribiere, a variant that does not require the hessian,

Differences:

- α_k is calculated via a line search along th
- β_{k+1} is replaced with

$$\beta_{k+1}^{PR} = \frac{\nabla f(x_{k+1})\left(\nabla f(x_{k+1}) - \nabla f(x_k)\right)}{\nabla f(x_k)^T \nabla f(x_k)}$$

```
def conjugate_gradient_scipy(x0, f, grad, tol=1e-
    all_x_i = [x0[0]]
    all_y_i = [x0[1]]
    all_f_i = [f(x0)]

    def store(X):
        x, y = X
        all_x_i.append(x)
        all_y_i.append(y)
        all_f_i.append(f(X))

    optimize.minimize(
      f, x0, jac=grad, method="CG",
      callback=store, tol=tol
    )

    return all_x_i, all_y_i, all_f_i
```

Trajectory

```
f, grad, hess = mk_quad(0.7)
opt = conjugate_gradient_scipy((1.6, 1.1), f, gra
plot_2d_traj((-1,2), (-1,2), f, title="$\\epsilon
```

```
f, grad, hess = mk_quad(0.02)
opt = conjugate_gradient_scipy((1.6, 1.1), f, gra
plot_2d_traj((-1,2), (-1,2), f, title="$\\epsilon
```

Rosenbrock's function

```
f, grad, hess = mk_rosenbrock()
opt = conjugate_gradient_scipy((1.6, 1.1), f, gra
plot_2d_traj((-2,2), (-2,2), f, traj=opt)
```

```
f, grad, hess = mk_rosenbrock()
opt = conjugate_gradient_scipy((-0.5, 0), f, grad
plot_2d_traj((-2,2), (-2,2), f, traj=opt)
```

Method: Newton-CG

Is a variant of Newtons method but does not require inverting the hessian, or even a hessian function - in which case it can be estimated by finite differencing of the gradient.

```
def newton_cg(x0, f, grad, hess=None, tol=1e-8):
    all_x_i = [x0[0]]
    all_y_i = [x0[1]]
    all_f_i = [f(x0)]

    def store(X):
        x, y = X
        all_x_i.append(x)
        all_y_i.append(y)
        all_f_i.append(f(X))

    optimize.minimize(
      f, x0, jac=grad, hess=hess, tol=tol,
      method="Newton-CG", callback=store
    )

    return all_x_i, all_y_i, all_f_i
```

Trajectory - well conditioned

```
f, grad, hess = mk_quad(0.7)
opt = newton_cg((1.6, 1.1), f, grad)
plot_2d_traj((-1,2), (-1,2), f, traj=opt, title="
```

```
f, grad, hess = mk_quad(0.7)
opt = newton_cg((1.6, 1.1), f, grad, hess)
plot_2d_traj((-1,2), (-1,2), f, traj=opt, title="
```

Trajectory - ill-conditioned

```
f, grad, hess = mk_quad(0.02)
opt = newton_cg((1.6, 1.1), f, grad)
plot_2d_traj((-1,2), (-1,2), f, traj=opt, title="
```

```
f, grad, hess = mk_quad(0.02)
opt = newton_cg((1.6, 1.1), f, grad, hess)
plot_2d_traj((-1,2), (-1,2), f, traj=opt, title="
```

Rosenbrock's function

```
f, grad, hess = mk_rosenbrock()
opt = newton_cg((1.6, 1.1), f, grad)
plot_2d_traj((-2,2), (-2,2), f, traj=opt, title="
```

```
f, grad, hess = mk_rosenbrock()
opt = newton_cg((1.6, 1.1), f, grad, hess)
plot_2d_traj((-2,2), (-2,2), f, traj=opt, title="
```

Method: BFGS

The Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm is a quasi-newton which iterative improves its approximation of the hessian,

```
def bfgs(x0, f, grad, hess=None, tol=1e-8):
    all_x_i = [x0[0]]
    all_y_i = [x0[1]]
    all_f_i = [f(x0)]

    def store(X):
        x, y = X
        all_x_i.append(x)
        all_y_i.append(y)
        all_f_i.append(f(X))

    optimize.minimize(
      f, x0, jac=grad, tol=tol,
      method="BFGS", callback=store
    )

    return all_x_i, all_y_i, all_f_i
```

Trajectory

```
f, grad, hess = mk_quad(0.7)
opt = bfgs((1.6, 1.1), f, grad)
plot_2d_traj((-1,2), (-1,2), f, traj=opt)
```

```
f, grad, hess = mk_quad(0.02)
opt = bfgs((1.6, 1.1), f, grad)
plot_2d_traj((-1,2), (-1,2), f, traj=opt)
```

Rosenbrock's function

```
f, grad, hess = mk_rosenbrock()
opt = bfgs((1.6, 1.1), f, grad)
plot_2d_traj((-2,2), (-2,2), f, traj=opt)
```

```
f, grad, hess = mk_rosenbrock()
opt = bfgs((-0.5, 0), f, grad)
plot_2d_traj((-2,2), (-2,2), f, traj=opt)
```

Method: Nelder-Mead

This is a gradient free method that uses a series of simplexes which are used to iteratively bracket the minimum.

```
def nelder_mead(x0, f, grad, hess=None, tol=1e-8)
    all_x_i = [x0[0]]
    all_y_i = [x0[1]]
    all_f_i = [f(x0)]

    def store(X):
        x, y = X
        all_x_i.append(x)
        all_y_i.append(y)
        all_f_i.append(f(X))

    optimize.minimize(
      f, x0, tol=tol,
      method="Nelder-Mead", callback=store
    )

    return all_x_i, all_y_i, all_f_i
```

Nelder-Mead

From https://github.com/greg-rychlewski/nelder-mead

Trajectory

```
f, grad, hess = mk_quad(0.7)
opt = nelder_mead((1.6, 1.1), f, grad)
plot_2d_traj((-1,2), (-1,2), f, traj=opt)
```

```
f, grad, hess = mk_quad(0.02)
opt = nelder_mead((1.6, 1.1), f, grad)
plot_2d_traj((-1,2), (-1,2), f, traj=opt)
```

Rosenbrock's function

```
f, grad, hess = mk_rosenbrock()
opt = nelder_mead((1.6, 1.1), f, grad)
plot_2d_traj((-2,2), (-2,2), f, traj=opt)
```

```
f, grad, hess = mk_rosenbrock()
opt = nelder_mead((-0.5, 0), f, grad)
plot_2d_traj((-2,2), (-2,2), f, traj=opt)
```