Optimization

Lecture 13

Dr. Colin Rundel

Optimization

Optimization problems underlie nearly everything we do in Machine Learning and Statistics. Most models can be formulated as

$$P \quad : \quad \underset{\boldsymbol{x} \in \boldsymbol{D}}{\arg\min} f(\boldsymbol{x})$$

- Formulating a problem P is not the same as being able to solve P in practice
- Many different algorithms exist for optimization but their performance varies widely depending the exact nature of the problem

Gradient Descent

Naive Gradient Descent

The basic idea behind this approach is that the gradient of a function tells us the direction of steepest ascent (or descent). Therefore, to find the minimum we should take our next step in the direction of the negative gradient to most quickly approach the nearest minima.

Given an n-dimensional function $f(x_1, \ldots, x_n)$, and an initial position $\boldsymbol{x}_k$ then our update rule becomes,

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - \alpha \nabla f(\boldsymbol{x}_k)$$

here α refers to the step length or the learning rate which determines how big a step we will take.

Implementation

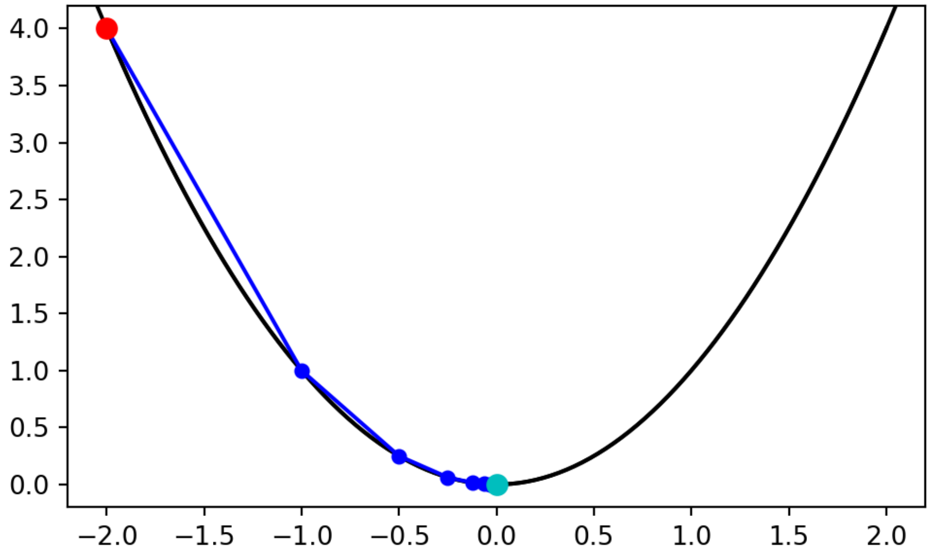
```
def grad_desc_1d(x, f, grad, step, max_step=100, tol = 1e-6):
  res = {"x": [x], "f": [f(x)]}

  try:
    for i in range(max_step):
      x = x - grad(x) * step
      if np.abs(x - res["x"][-1]) < tol:
        break

      res["f"].append( f(x) )
      res["x"].append( x )

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

  if i == max_step-1:
    warnings.warn("Failed to converge!", RuntimeWarning)

  return res
```

A basic example

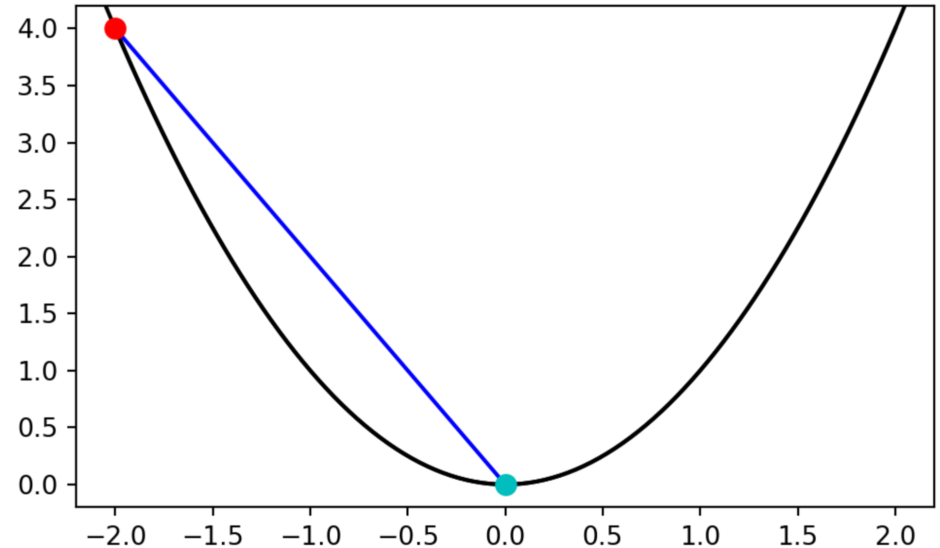
$$f(x) = x^2$$
$$\nabla f(x) = 2x$$

```
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 )
```

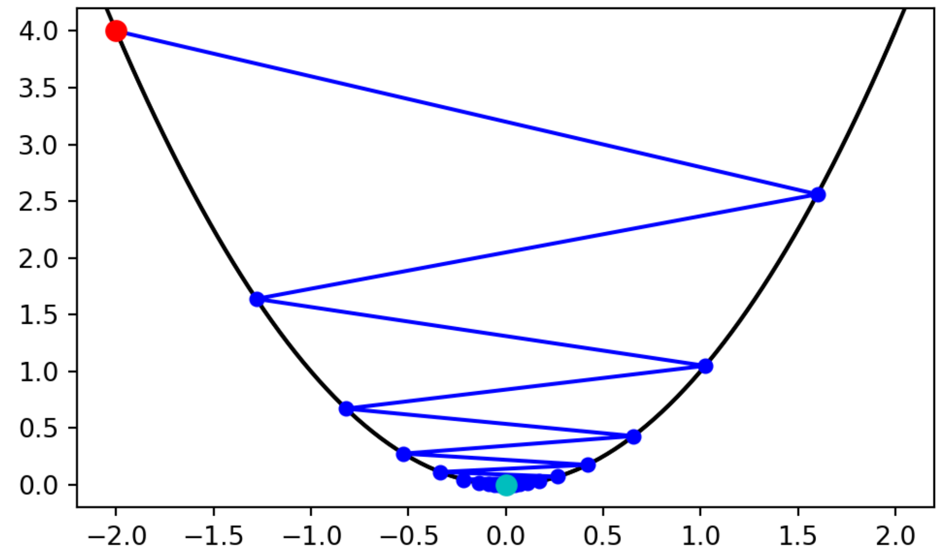


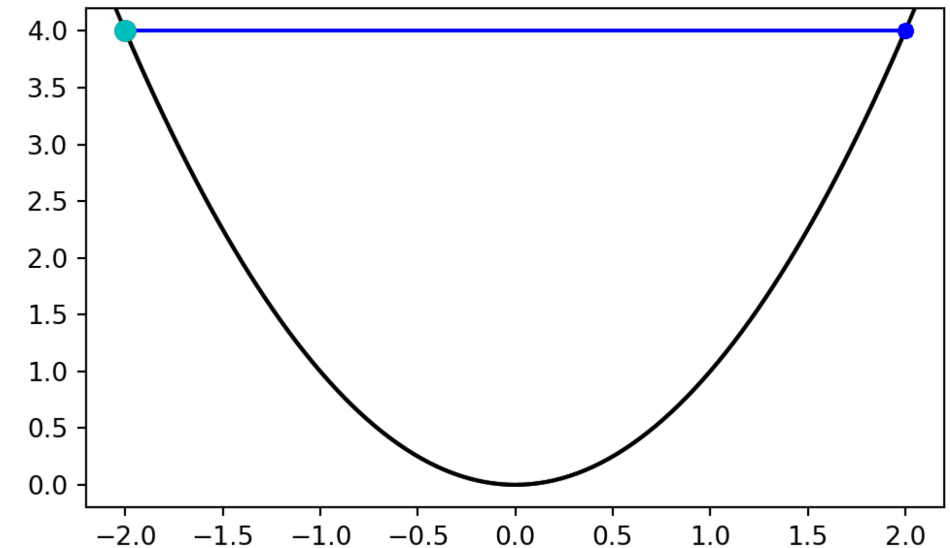
Where can it go wrong?

If you pick a bad step size …

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





Local minima

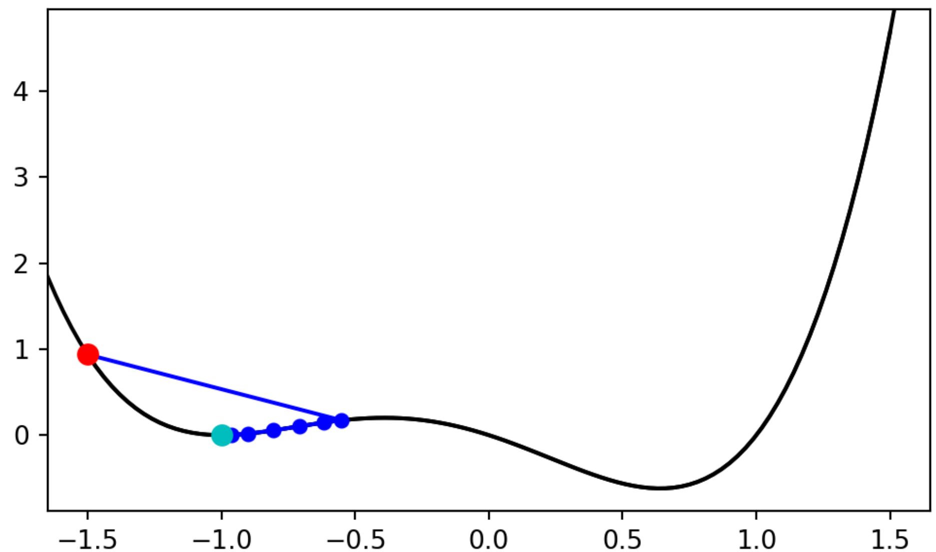
The function now has multiple minima, both starting point and step size affect the solution we obtain,

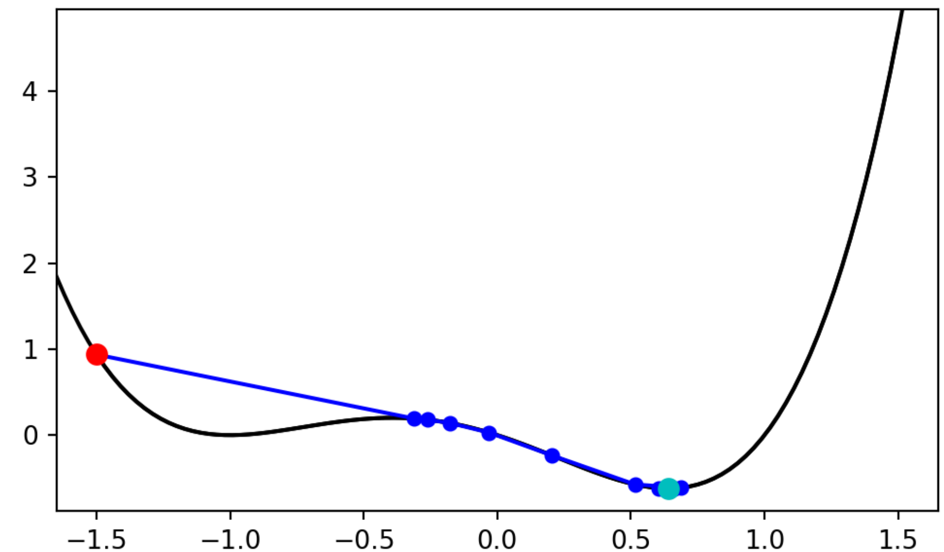
$$f(x) = x^4 + x^3 - x^2 - x$$
$$\nabla f(x) = 4x^3 + 3x^2 - 2x - 1$$

```
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)
```

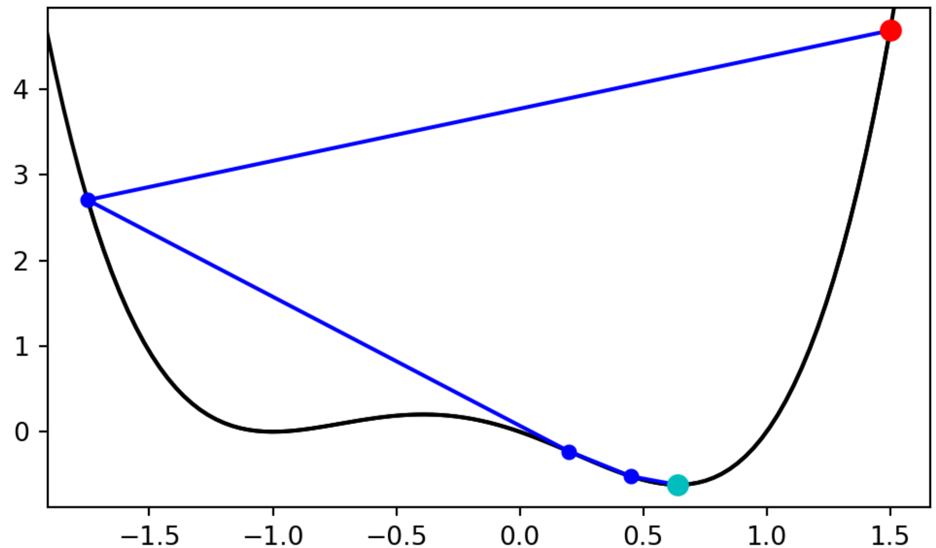


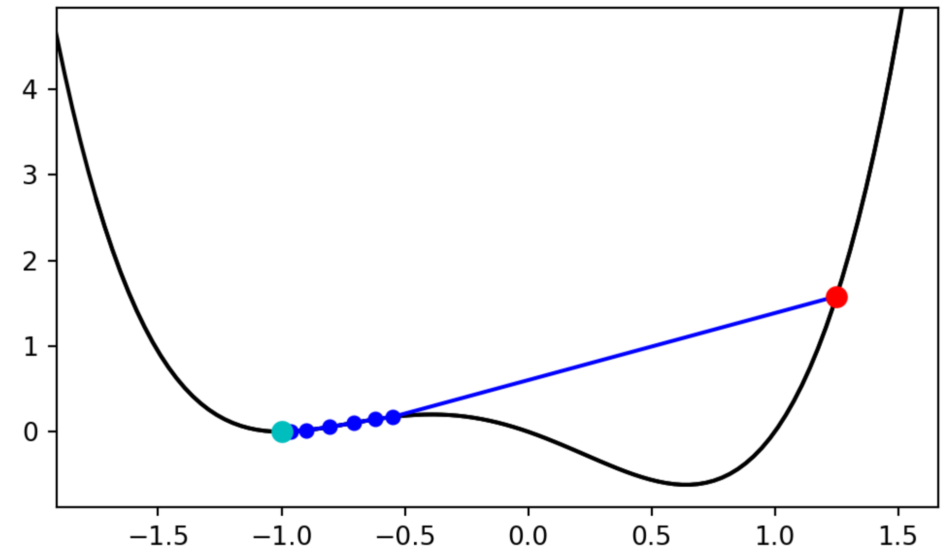


Alternative starting points

```python
opt = grad_desc_1d(1.5, f, grad, step=0.2)
plot_1d_traj( (-1.75, 1.5), f, opt )
```

```python
opt = grad_desc_1d(1.25, f, grad, step=0.2)
plot_1d_traj( (-1.75, 1.5), f, opt)
```





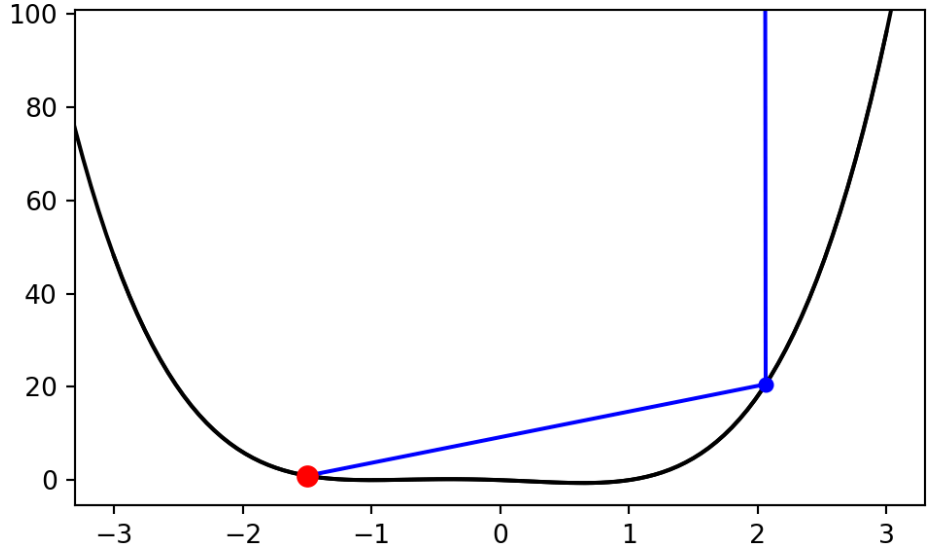
Problematic step sizes

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

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

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

```python
plot_1d_traj( (-3, 3), f, opt)
```




```python
opt['x']
```

```
[-1.5, 2.0625, -29.986083984375,
78789.99556875888, -1467366557235808.0,
9.478445237313853e+45]
```

```python
opt['f']
```

```
[0.9375, 20.552993774414062, 780666.4923959533,
3.853805712579921e+19, 4.636117851941789e+60,
8.071391646153008e+183]
```

Gradient Descent w/ backtracking

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

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

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

Pick larger α to start (but not so large so as to overflow) and then let the backtracking tune things.

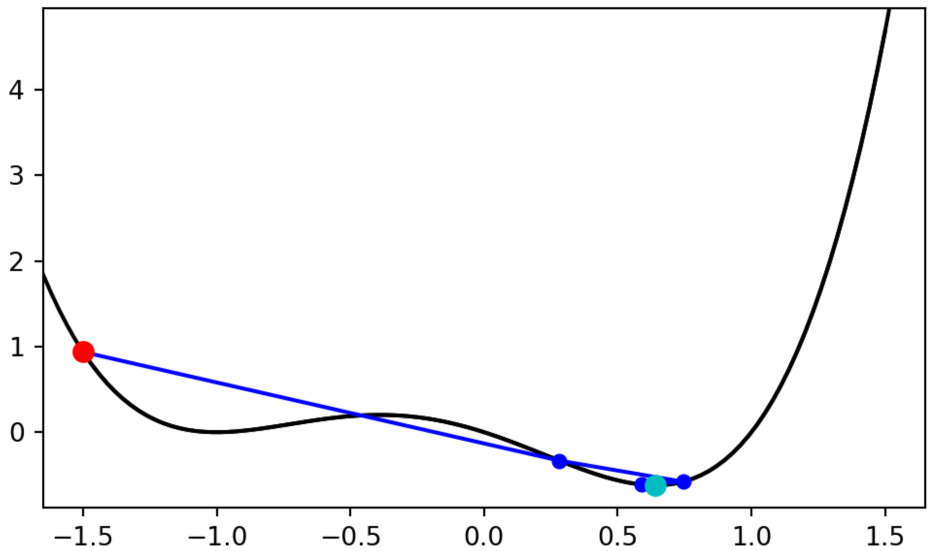
This is a simplified (hand wavy) version of the Armijo-Goldstein condition

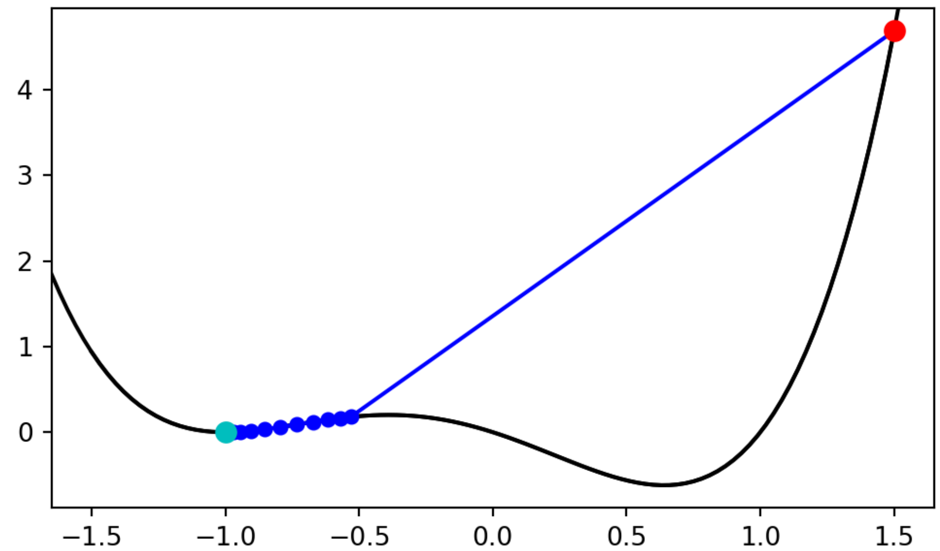
Implementation

```python
def grad_desc_1d_bt(x, f, grad, step, tau=0.5, max_step=100, max_back=10, tol = 1e-6):
  res = {"x": [x], "f": [f(x)]}

  for i in range(max_step):
    grad_f = grad(x)
    for j in range(max_back):
      x = res["x"][-1] - step * grad_f
      f_x = f(x)
      if (f_x < res["f"][-1]):
        break
      step = step * tau

    if np.abs(x - res["x"][-1]) < tol:
      break
    res["x"].append(x)
    res["f"].append(f_x)

  if i == max_step-1:
    warnings.warn("Failed to converge!", RuntimeWarning)

  return res
```

```python
opt = grad_desc_1d_bt(
  -1.5, f, grad, step=0.75, tau=0.5
)
plot_1d_traj( (-1.5, 1.5), f, opt )
```

```python
opt = grad_desc_1d_bt(
  1.5, f, grad, step=0.25, tau=0.5
)
plot_1d_traj( (-1.5, 1.5), f, opt)
```





A 2d cost function

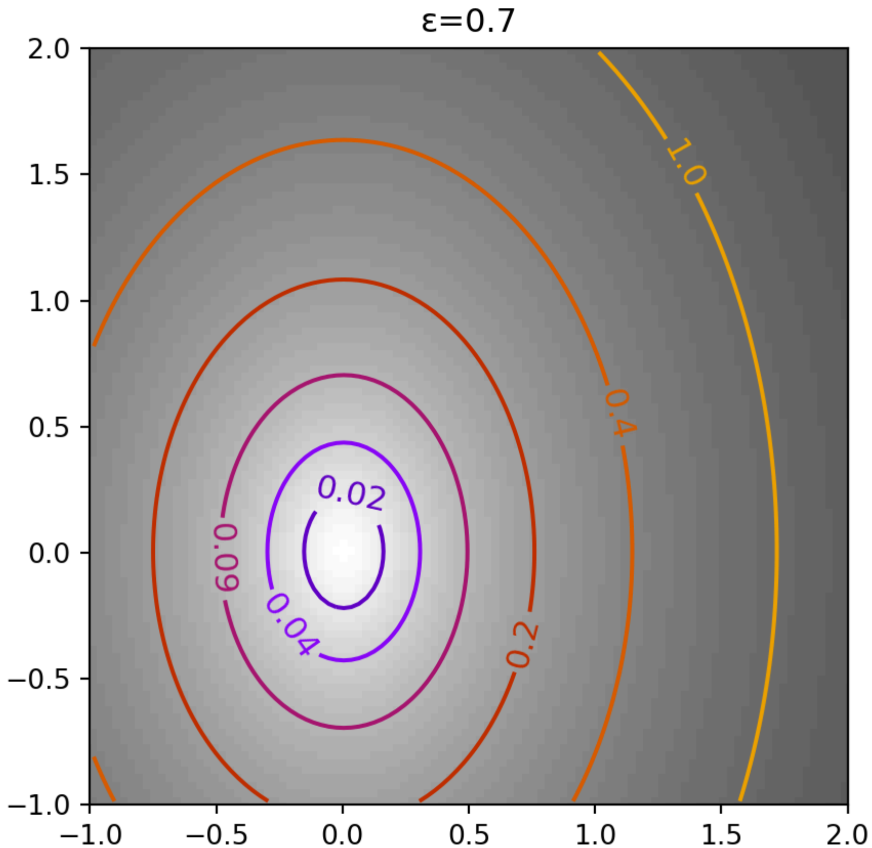
We will be using `mk_quad()` to create quadratic functions with varying conditioning (as specified by the `epsilon` parameter).

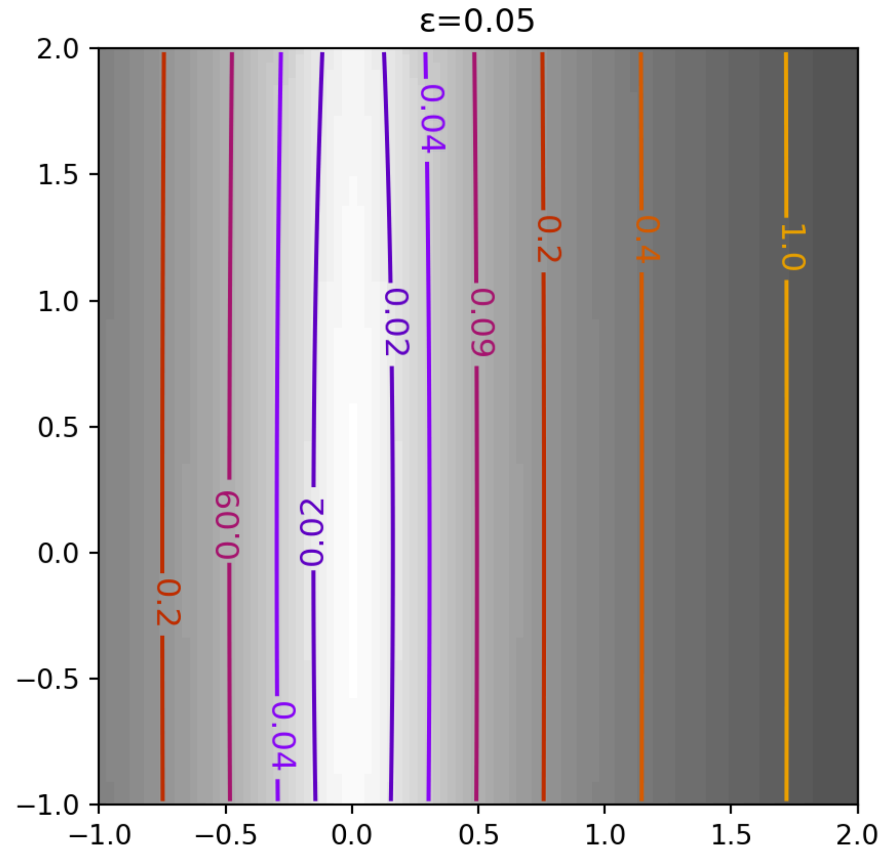
$$
\begin{align}
f(x,y) &= 0.33(x^2 + \epsilon^2 y^2) \\
\nabla f(x,y) &= \begin{bmatrix} 0.66 \, x \\ 0.66 \, \epsilon^2 \, y \end{bmatrix} \\
\nabla^2 f(x,y) &= \begin{bmatrix} 0.66 & 0 \\ 0 & 0.66 \, \epsilon^2 \end{bmatrix}
\end{align}
$$

Examples

```python
f, grad, hess = mk_quad(0.7)
plot_2d_traj(
  (-1,2), (-1,2), f, title="ε=0.7"
)
```

```python
f, grad, hess = mk_quad(0.05)
plot_2d_traj(
  (-1,2), (-1,2), f, title="ε=0.05"
)
```





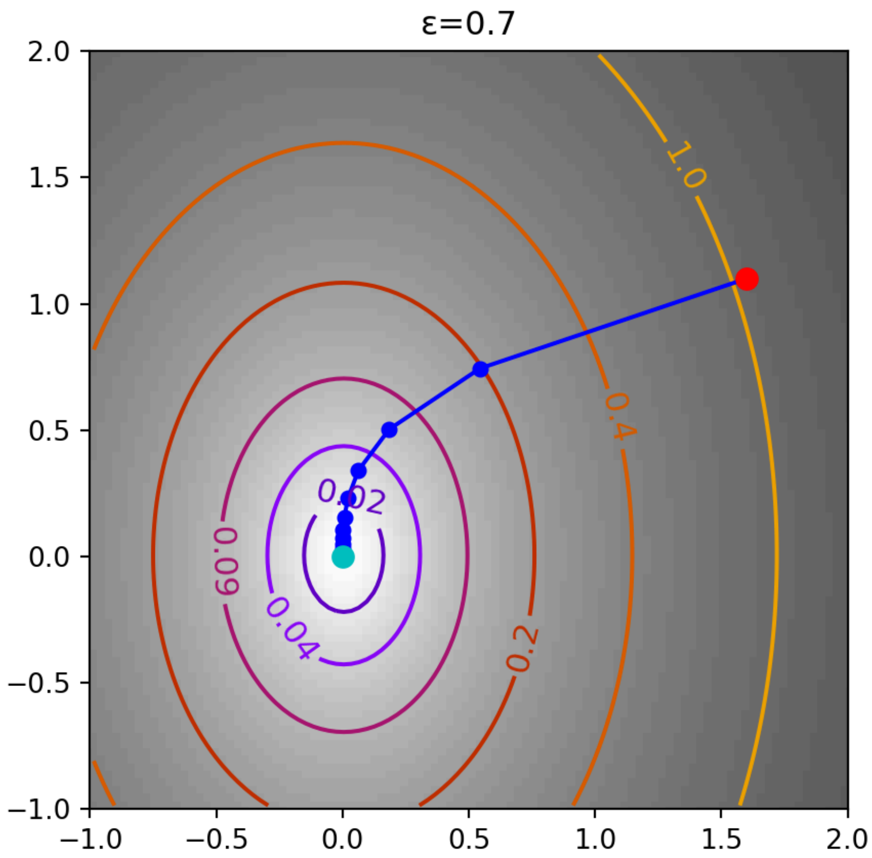
n-d gradient descent w/ backtracking

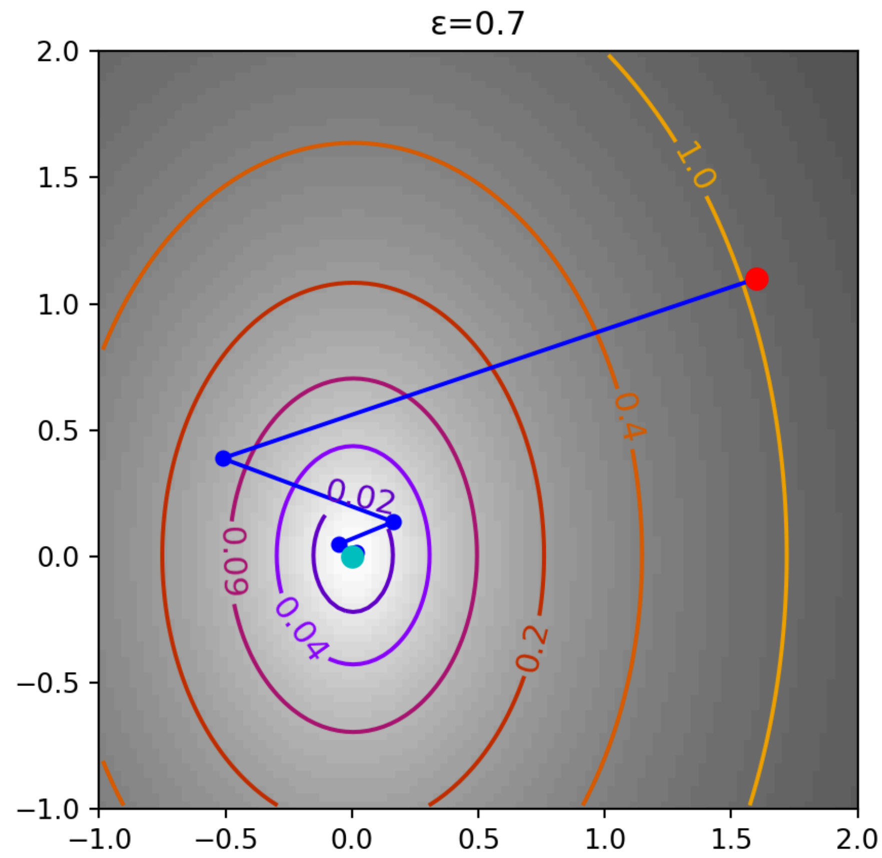
```
def grad_desc(x, f, grad, step, tau=0.5, max_step=100, max_back=10, tol = 1e-8):
  res = {"x": [x], "f": [f(x)]}

  for i in range(max_step):
    grad_f = grad(x)

    for j in range(max_back):
      x = res["x"][-1] - grad_f * step
      f_x = f(x)
      if (f_x < res["f"][-1]):
        break
      step = step * tau

    if np.sqrt(np.sum((x - res["x"][-1])**2)) < tol:
      break

    res["x"].append(x)
    res["f"].append(f_x)

  if i == max_step-1:
    warnings.warn("Failed to converge!", RuntimeWarning)

  return res
```

Well conditioned cost function

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

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

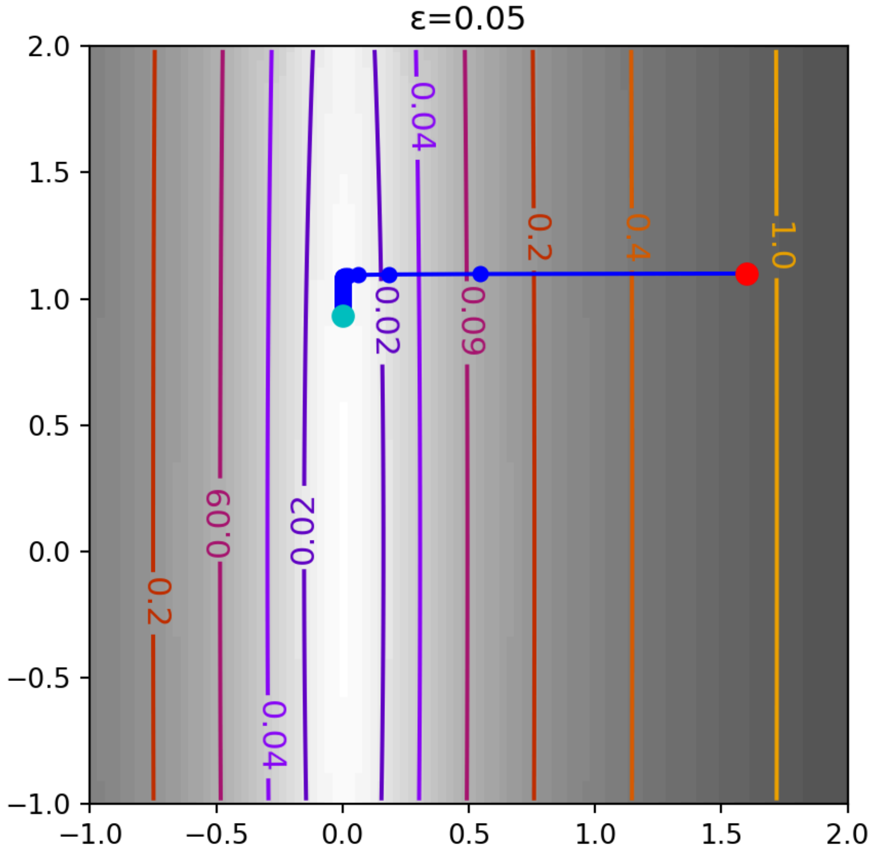


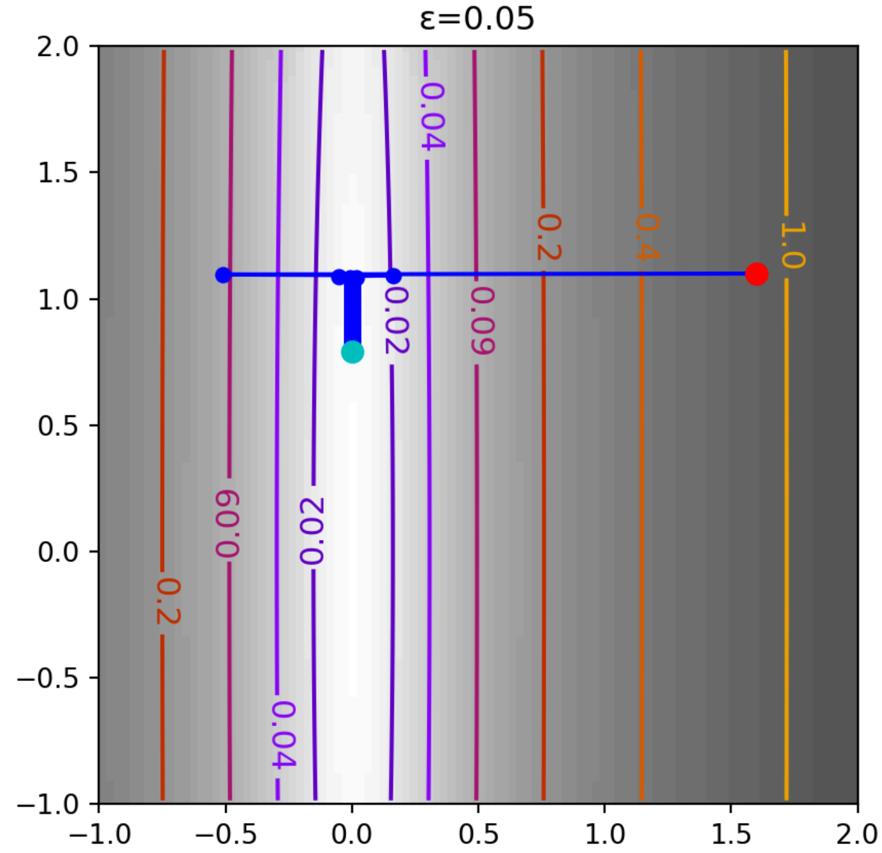


Ill-conditioned cost function

```
f, grad, hess = mk_quad(0.05)
opt = grad_desc((1.6, 1.1), f, grad, step=1
plot_2d_traj(
  (-1,2), (-1,2), f, title="ε=0.05", traj=op
)
```

```
f, grad, hess = mk_quad(0.05)
opt = grad_desc((1.6, 1.1), f, grad, step=2)
plot_2d_traj(
  (-1,2), (-1,2), f, title="ε=0.05", traj=op
)
```

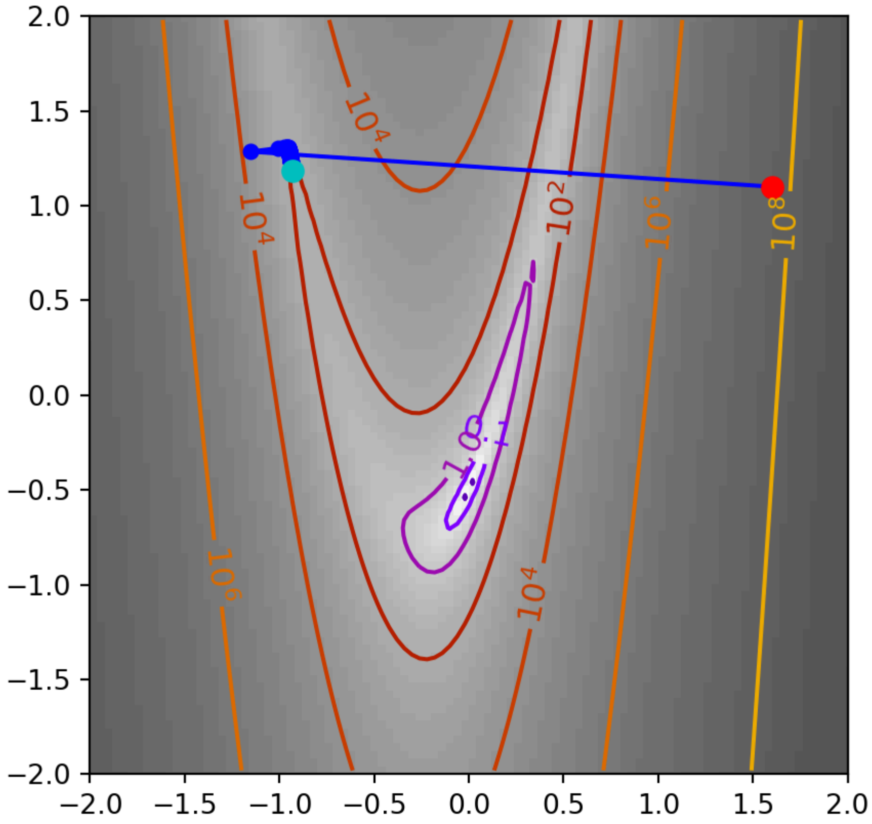


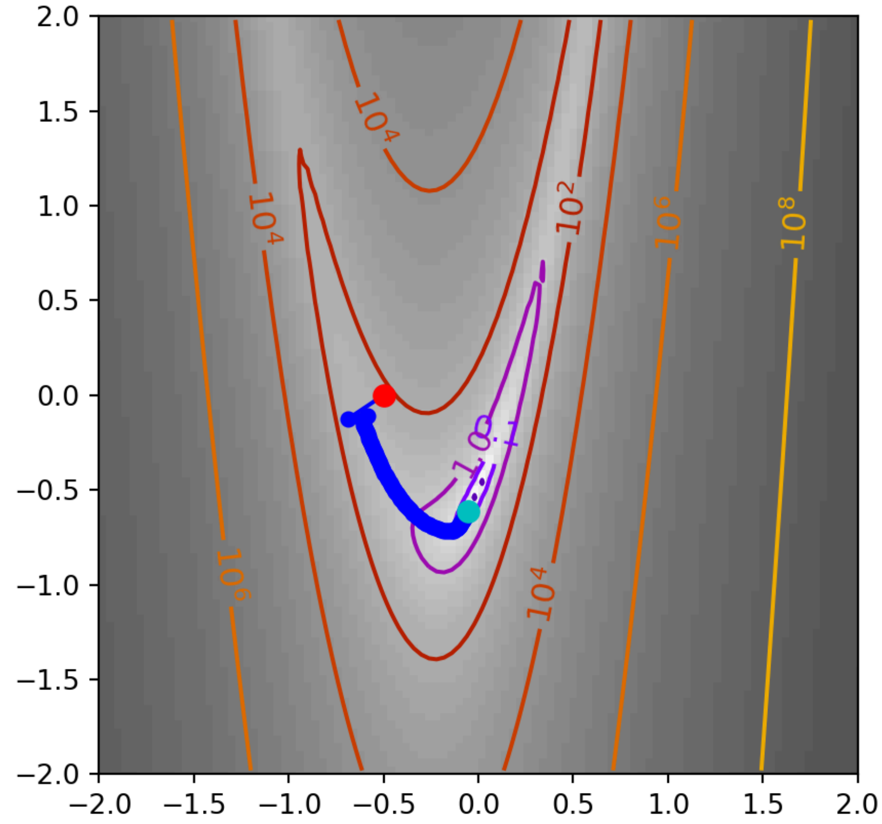


Rosenbrock function (very ill conditioned)

```
f, grad, hess = mk_rosenbrock()
opt = grad_desc((1.6, 1.1), f, grad, step=0
plot_2d_traj((-2,2), (-2,2), f, traj=opt)
```

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





Some regression examples

```python
from sklearn.datasets import make_regression
X, y, coef = make_regression(
  n_samples=200, n_features=20, n_informative=4,
  bias=3, noise=1, random_state=1234, coef=True
)
```

```python
y
```

```
array([
-36.2252,
9.6357,
66.4583,
48.9574,
24.1885,
-13.2444,

18.1455,
-135.047 ,
116.5772,
60.2524,
30.9319,
107.148 ,

21.6209,
66.2401,
-132.8878,
58.636 ,
```

```python
X
```

```
array([[-0.6465,  2.0803,  0.1412, -0.8419, -0.1595,  1.3321, -0.4262,
        -0.0351, -0.1938, -0.6093, -0.3433,  0.6126,  0.3777, -1.2062,
        -0.2277, -0.8896, -0.4674, -1.3566,  1.4989, -0.7468],
       [-0.3834, -0.3631, -1.2196,  0.6   ,  0.3315,  1.1056,  0.2662,
        -0.7239,  0.0259, -0.2172, -0.6841,  0.0991,  0.2794, -1.208 ,
        -0.7818, -1.7348, -1.3397, -0.5723, -0.5882,  0.2717],
       [-0.1637, -0.8118,  0.9551,  0.5711,  0.8719, -0.9619,  1.9846,
        -1.1806, -1.1261,  0.297 ,  1.2499,  0.7109, -0.1183,  0.6708,
         0.6895,  1.4705,  0.0634, -0.3079, -2.2512, -0.0216],
       [-0.9292, -0.4897, -2.1196, -1.142 ,  1.266 , -0.2988,  1.0016,
        -2.1969, -1.0739, -0.1149,  0.5122,  0.302 , -0.0974,  1.3461,
         0.1909,  1.1223,  0.6268,  2.2035, -0.5135,  2.0118],
       [ 0.1645, -0.5847,  0.2708, -3.5635,  0.1526,  0.5283,  0.7674,
         1.392 , -0.0819,  1.3211,  0.4644, -1.0279,  0.9849, -1.069 ,
        -0.4301,  0.0798, -0.5119, -0.3448,  0.8166, -0.4   ],
       [ 0.4134,  1.9511, -0.5013, -1.4894,  0.4191, -1.4104,  0.2617,
        -0.6981,  0.0368, -1.151 ,  2.0752,  0.5001, -0.2428,  0.45  ,
         0.7176,  1.3846,  0.5155,  0.4459, -0.2784, -0.2864],
       [-0.0628, -1.424 , -1.1023,  0.1445, -0.4836,  1.4795, -0.5921,
```

```python
coef
```

```
array([ 0.
,  0.     ,  0.
,  9.6106,
43.4239,  0.
,  0.     ,
        0.
,  0.     ,
34.453 ,
9.2929,  0.
,  0.     ,  0.
,
        0.
,  0.     ,  0.
,  0.     ,  0.
,  0.     ])
```

A jax implementation of GD

```python
def grad_desc_jax(x, f, step, tau=0.5, max_step=100, max_back=10, tol = 1e-8):
  grad_f = jax.grad(f)
  f_x = f(x)
  converged = False

  for i in range(max_step):
    grad_f_x = grad_f(x)

    for j in range(max_back):
      new_x = x - grad_f_x * step
      new_f_x = f(new_x)
      if (new_f_x < f_x):
        break
      step *= tau

    cur_tol = jnp.sqrt(jnp.sum((x - new_x)**2))

    x = new_x
    f_x = new_f_x

    if cur_tol < tol:
      converged = True
      break

```

Linear regression

```python
lm = LinearRegression().fit(X,y)
np.r_[lm.intercept_, lm.coef_]
```

```
array([ 3.0616, -0.0121, -0.0096,  0.096 ,  9.6955, 43.406 ,  0.0253,
        0.0284,  0.0962,  0.1069, 34.4884,  9.3445, -0.0165, -0.0147,
       -0.0396,  0.0969, -0.1057, -0.0943,  0.11  , -0.0096, -0.0875])
```

```python
def jax_linear_regression(X, y, beta):
  Xb = jnp.c_[jnp.ones(X.shape[0]), X]
  return jnp.sum((y - Xb @ beta)**2)

grad_desc_jax(
  np.zeros(X.shape[1]+1),
  lambda beta: jax_linear_regression(X,y,beta),
  step = 1, tau = 0.5
)
```

```
{'x': Array([ 3.0617, -0.0121, -0.0097,  0.0961,  9.6958, 43.4061,  0.0255,
        0.0285,  0.0963,  0.1066, 34.488 ,  9.3445, -0.0166, -0.0146,
       -0.0396,  0.0966, -0.1056, -0.0941,  0.1101, -0.0099, -0.0879],      dtype=float32),
'n_iter': 18, 'converged': True, 'final_tol': 1e-08, 'final_step': 7.450580596923828e-09}
```

Ridge regression

```
r = GridSearchCV(Ridge(), param_grid = {"alpha": np.logspace(-3,0)}).fit(X,y)
r.best_estimator_
```

```
Ridge(alpha=np.float64(0.05963623316594643))
```

```
np.r_[r.best_estimator_.intercept_, r.best_estimator_.coef_]
```

```
array([ 3.0631, -0.0122, -0.0102,  0.0957,  9.6928, 43.3936,  0.0265,
        0.0282,  0.0959,  0.1074, 34.478 ,  9.3399, -0.0179, -0.0134,
       -0.0403,  0.0956, -0.106 , -0.0933,  0.1101, -0.0108, -0.0877])
```

```
def jax_ridge(X, y, beta, alpha):
  Xb = jnp.c_[jnp.ones(X.shape[0]), X]
  ls_loss = jnp.sum((y - Xb @ beta)**2)
  coef_loss = alpha * jnp.sum(beta[1:]**2)
  return ls_loss + coef_loss

grad_desc_jax(
  np.zeros(X.shape[1]+1),
  lambda beta: jax_ridge(X, y, beta, r.best_estimator_.alpha),
  step = 1, tau = 0.5, tol=1e-8
)
```

```
{'x': Array([ 3.0633, -0.0122, -0.0104,  0.0957,  9.6931, 43.3938,  0.0267,
        0.0282,  0.0961,  0.1071, 34.4775,  9.3398, -0.0179, -0.0134,
       -0.0404,  0.0952, -0.1059, -0.093 ,  0.1102, -0.011 , -0.0882],      dtype=float32),
'n_iter': 18, 'converged': True, 'final_tol': 1e-08, 'final_step': 1.862645149230957e-09}
```

Lasso

```
ls = GridSearchCV(Lasso(), param_grid = {"alpha": np.logspace(-3,0)}).fit(X,y)
ls.best_estimator_
```

```
Lasso(alpha=np.float64(0.10481131341546852))
```

```
np.r_[ls.best_estimator_.intercept_, ls.best_estimator_.coef_]
```

```
array([ 3.0555, -0.    , -0.    ,  0.    ,  9.5849, 43.312 ,  0.    ,
        0.    ,  0.    ,  0.    , 34.3712,  9.2233, -0.    ,  0.    ,
       -0.    ,  0.    , -0.    , -0.    ,  0.    , -0.    , -0.    ])
```

```
def jax_lasso(X, y, beta, alpha):
  n = X.shape[0]
  Xb = jnp.c_[jnp.ones(n), X]
  ls_loss = (1/(2*n))*jnp.sum((y - Xb @ beta)**2)
  coef_loss = alpha * jnp.sum(jnp.abs(beta[1:]))
  return ls_loss + coef_loss

grad_desc_jax(
  np.zeros(X.shape[1]+1),
  lambda beta: jax_lasso(X, y, beta, ls.best_estimator_.alpha),
  step = 1, tau = 0.5, tol=1e-10
)
```

```
{'x': Array([ 3.0623,  0.    , -0.    ,  0.    ,  9.5842, 43.303 ,  0.    ,
       -0.    ,  0.0031,  0.0098, 34.3752,  9.2158,  0.    ,  0.    ,
       -0.    ,  0.    , -0.0144, -0.0124,  0.0191, -0.    , -0.    ],      dtype=float32),
'n_iter': 44, 'converged': True, 'final_tol': 1e-10, 'final_step': 1.4551915228366852e-11}
```

Limitation of gradient descent

- GD is deterministic
- GD finds *local* minima
- GD is sensitive to starting location
- GD can be computationally expensive because Gradients are often computationally expensive to calculate
- GD is sensitive to choices of learning rates
- GD treats all directions in parameter space uniformly

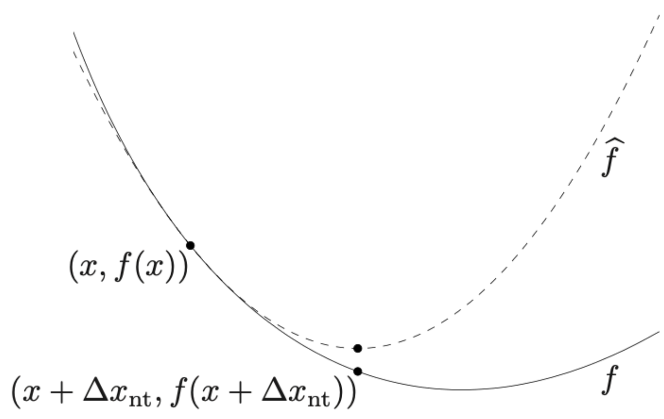
Newton's Method

Newton's Method in 1d

Lets assume we have a 1d function $f(x)$ we are trying to optimize, our current guess is x and we want to know how to generate our next step Δx.

We start by constructing the 2nd order Taylor approximation of our function at $x + \Delta x$,

$$f(x + \Delta x) \approx \widehat{f}(x + \Delta x) = f(x) + \Delta x f'(x) + \frac{1}{2}(\Delta x)^2 f''(x)$$



Finding the Newton step

Our optimal step then becomes the value of Δx that minimizes the quadratic given by our Taylor approximation.

$$
\begin{gathered}
\frac{\partial}{\partial \Delta x} \widehat{f}(x + \Delta x) = 0 \\
\frac{\partial}{\partial \Delta x} \left(f(x) + \Delta x f'(x) + \frac{1}{2} (\Delta x)^2 f''(x) \right) = 0 \\
f'(x) + \Delta x f''(x) = 0 \\
\Delta x = -\frac{f'(x)}{f''(x)}
\end{gathered}
$$

which suggests an iterative update rule of

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

Generalizing to nd

Based on the same argument we can see the follow result for a function in $\mathbb{R}^n$,

$$f(\boldsymbol{x} + \boldsymbol{\Delta x}) \approx \widehat{f}(\boldsymbol{x}) = f(\boldsymbol{x}) + \boldsymbol{\Delta x}^T \nabla f(\boldsymbol{x}) + \frac{1}{2} \boldsymbol{\Delta x}^T \, \nabla^2 f(\boldsymbol{x}) \, \boldsymbol{\Delta x}$$

then

$$\frac{\partial}{\partial \boldsymbol{\Delta x}} \widehat{f}(\boldsymbol{x}) = 0$$

$$\nabla f(\boldsymbol{x}) + \nabla^2 f(\boldsymbol{x}) \, \boldsymbol{\Delta x} = 0$$

$$\boldsymbol{\Delta x} = -\left(\nabla^2 f(\boldsymbol{x})\right)^{-1} \nabla f(\boldsymbol{x})$$

where

- $\nabla f(\boldsymbol{x})$ is the $n \times 1$ gradient vector
- and $\nabla^2 f(\boldsymbol{x})$ is the $n \times n$ Hessian matrix.

based on these results our nd update rule is

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k - (\nabla^2 f(\boldsymbol{x}_k))^{-1} \, \nabla f(\boldsymbol{x}_k)$$

Implementation

```
def newtons_method(x, f, grad, hess, max_iter=100, tol=1e-8):
    x = np.array(x)
    s = x.shape
    res = {"x": [x], "f": [f(x)]}

    for i in range(max_iter):
      x = x - np.linalg.solve(hess(x), grad(x))
      x = x.reshape(s)

      if np.sqrt(np.sum((x - res["x"][-1])**2)) < tol:
        break

      res["x"].append(x)
      res["f"].append(f(x))

    return res
```

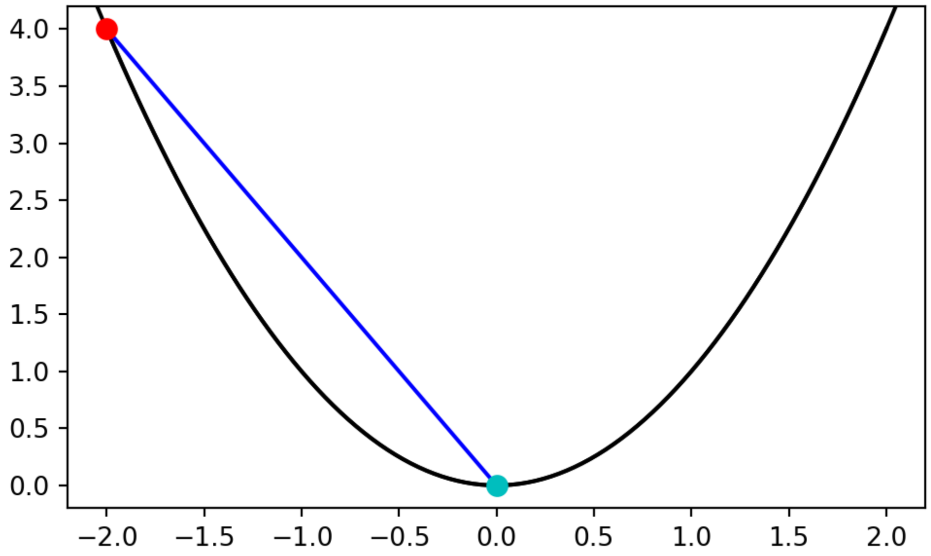
A basic example

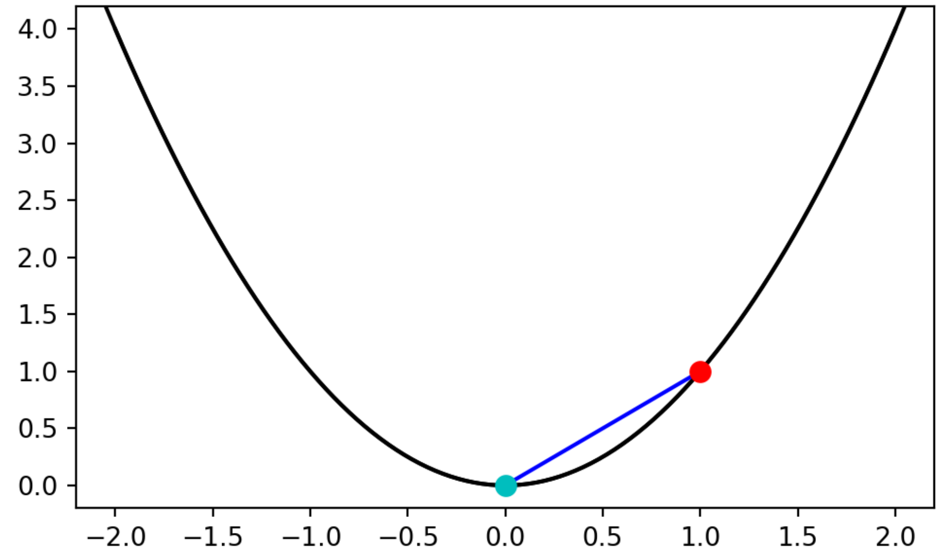
$$f(x) = x^2$$
$$\nabla f(x) = 2x$$
$$\nabla^2 f(x) = 2$$

```
f = lambda x: np.array(x**2)
grad = lambda x: np.array([2*x])
hess = lambda x: np.array([[2]])
```

```
opt = newtons_method(-2., f, grad, hess)
plot_1d_traj( (-2, 2), f, opt )
```

```
opt = newtons_method(1., f, grad, hess)
plot_1d_traj( (-2, 2), f, opt )
```



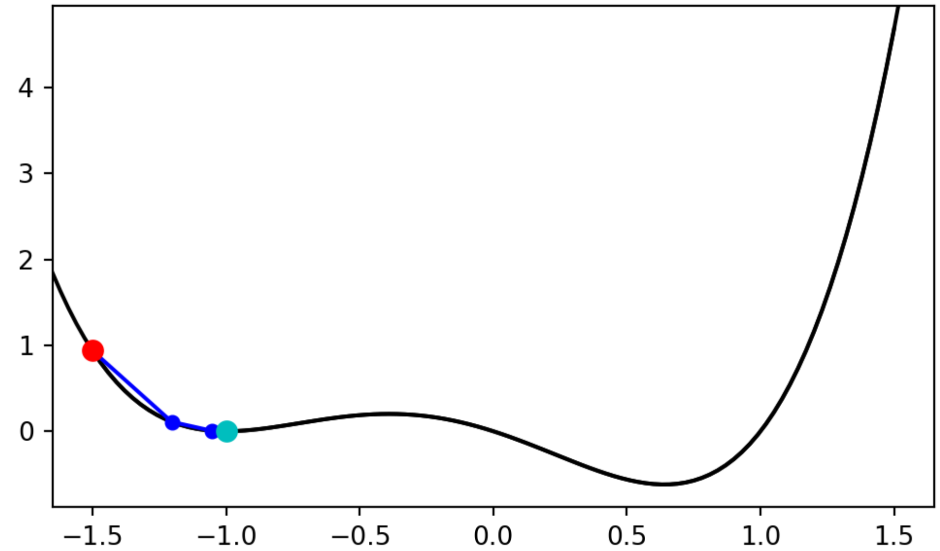


1d Cubic

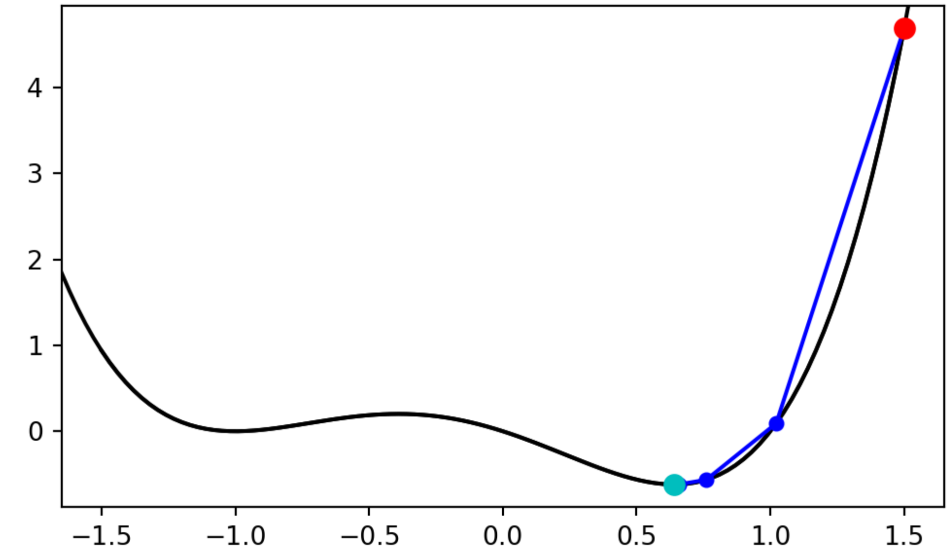
$$f(x) = x^4 + x^3 - x^2 - x$$
$$\nabla f(x) = 4x^3 + 3x^2 - 2x - 1$$
$$\nabla^2 f(x) = 12x^2 + 6x - 2$$

```
f = lambda x: x**4 + x**3 - x**2 - x
grad = lambda x: np.array([4*x**3 + 3*x**2 -
hess = lambda x: np.array([[12*x**2 + 6*x -
```

```
opt = newtons_method(-1.5, f, grad, hess)
plot_1d_traj( (-1.5, 1.5), f, opt )
```



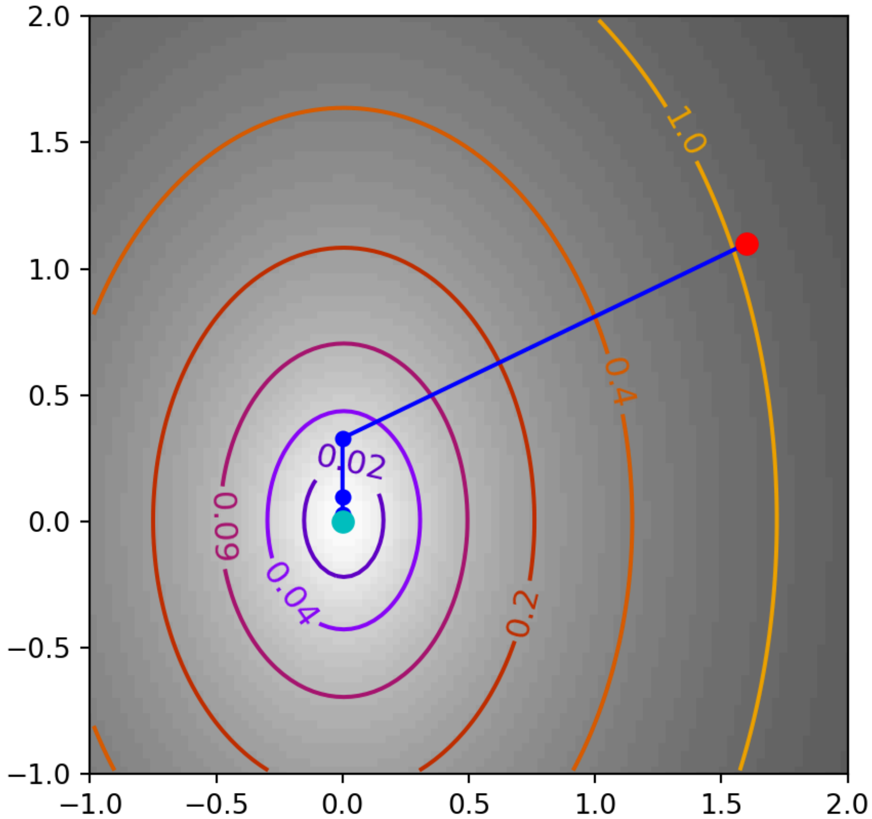

```
opt = newtons_method(1.5, f, grad, hess)
plot_1d_traj( (-1.5, 1.5), f, opt)
```

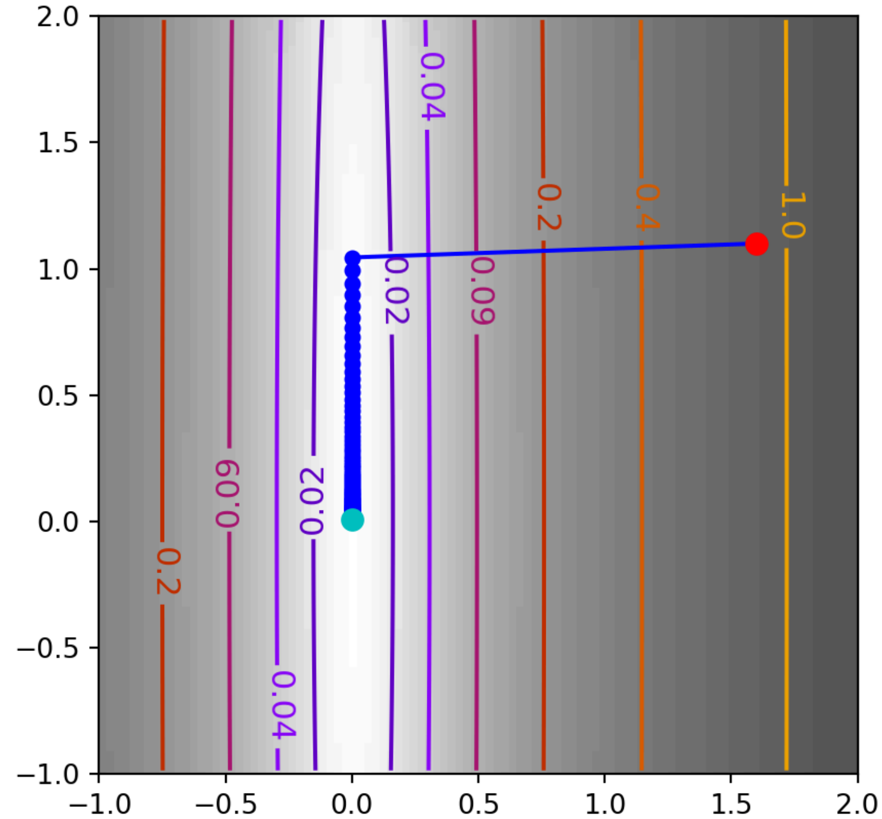


2d quadratic cost function

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

```
f, grad, hess = mk_quad(0.05)
opt = newtons_method((1.6, 1.1), f, grad, he
plot_2d_traj((-1,2), (-1,2), f, traj=opt)
```

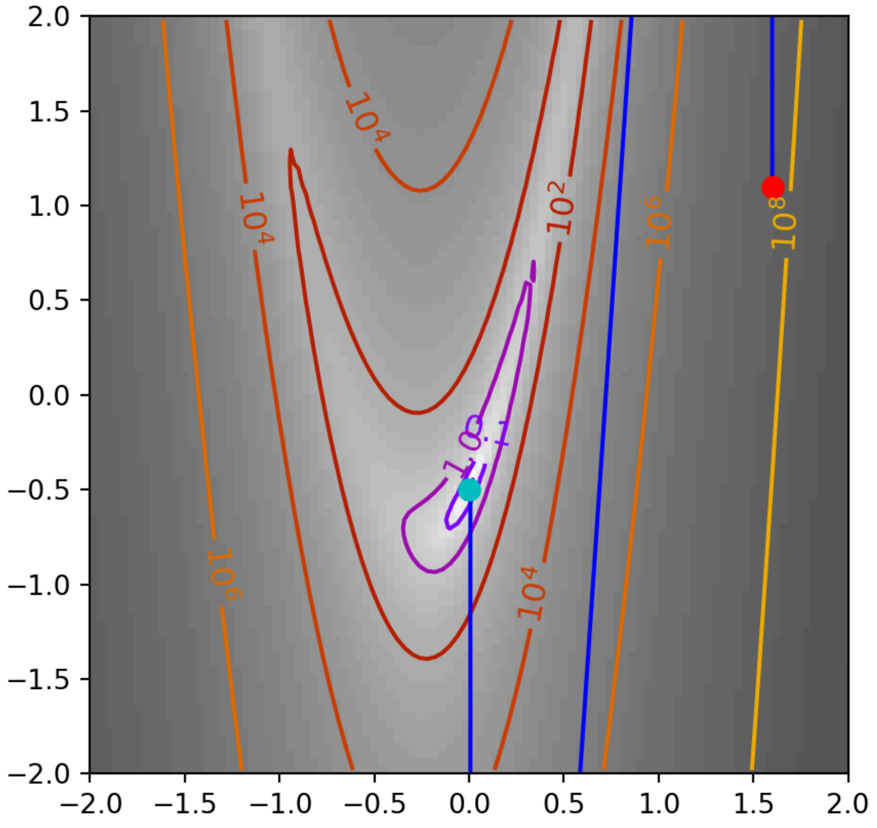


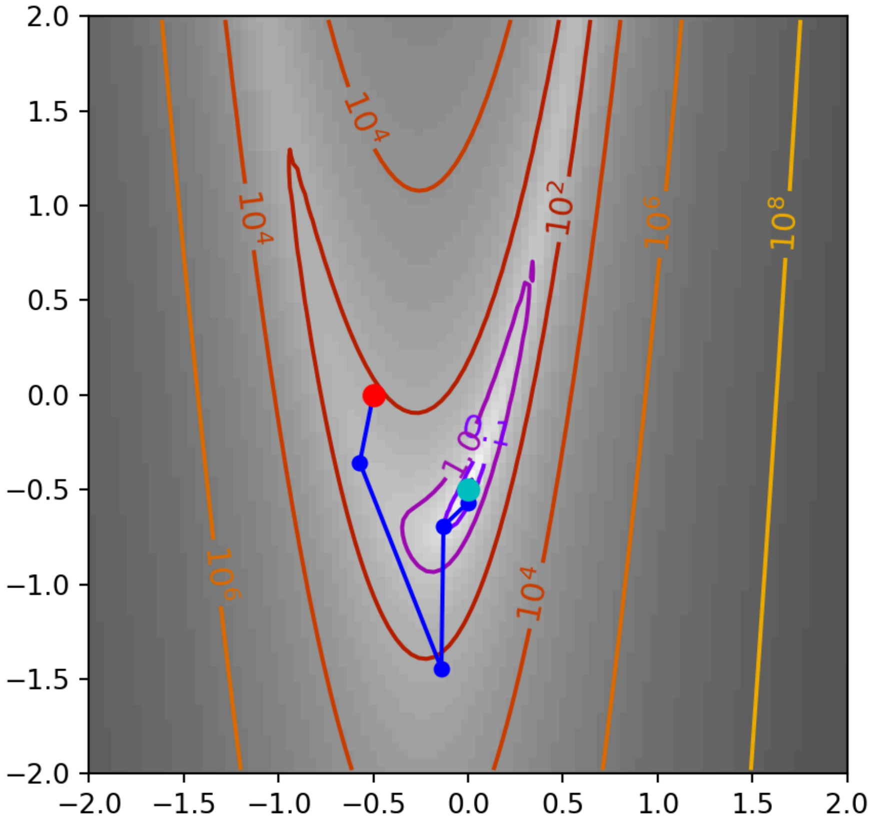


Rosenbrock function

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

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





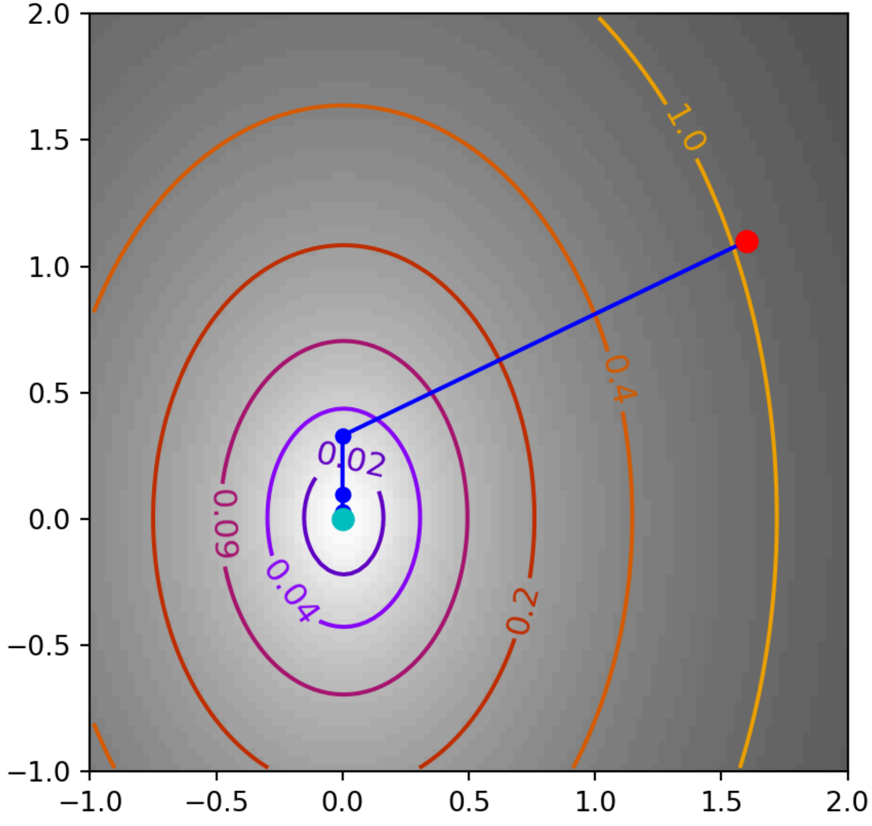
Damped / backtracking implementation

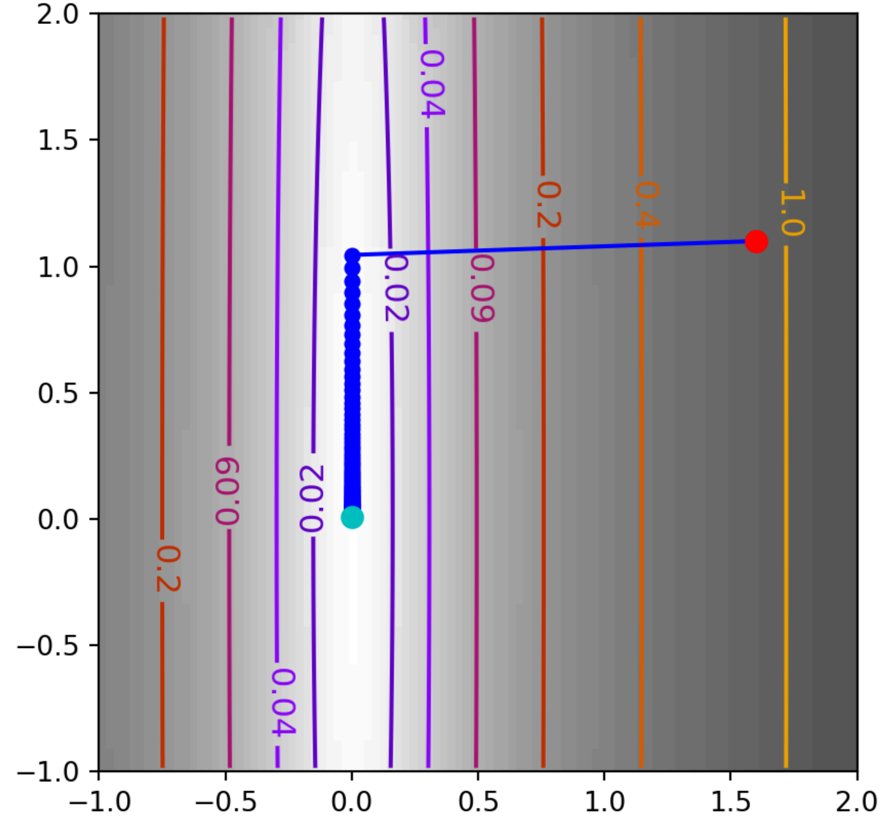
```
def newtons_method_damped(
  x, f, grad, hess, max_iter=100, max_back=10, tol=1e-8,
  alpha=0.5, beta=0.75
):
    res = {"x": [x], "f": [f(x)]}

    for i in range(max_iter):
      grad_f = grad(x)
      step = - np.linalg.solve(hess(x), grad_f)
      t = 1
      for j in range(max_back):
        if f(x+t*step) < f(x) + alpha * t * grad_f @ step:
          break
        t = t * beta

      x = x + t * step

      if np.sqrt(np.sum((x - res["x"][-1])**2)) < tol:
        break

      res["x"].append(x)
      res["f"].append(f(x))

    return res
```

2d quadratic cost function

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

```
f, grad, hess = mk_quad(0.05)
opt = newtons_method_damped((1.6, 1.1), f, g
plot_2d_traj((-1,2), (-1,2), f, traj=opt)
```

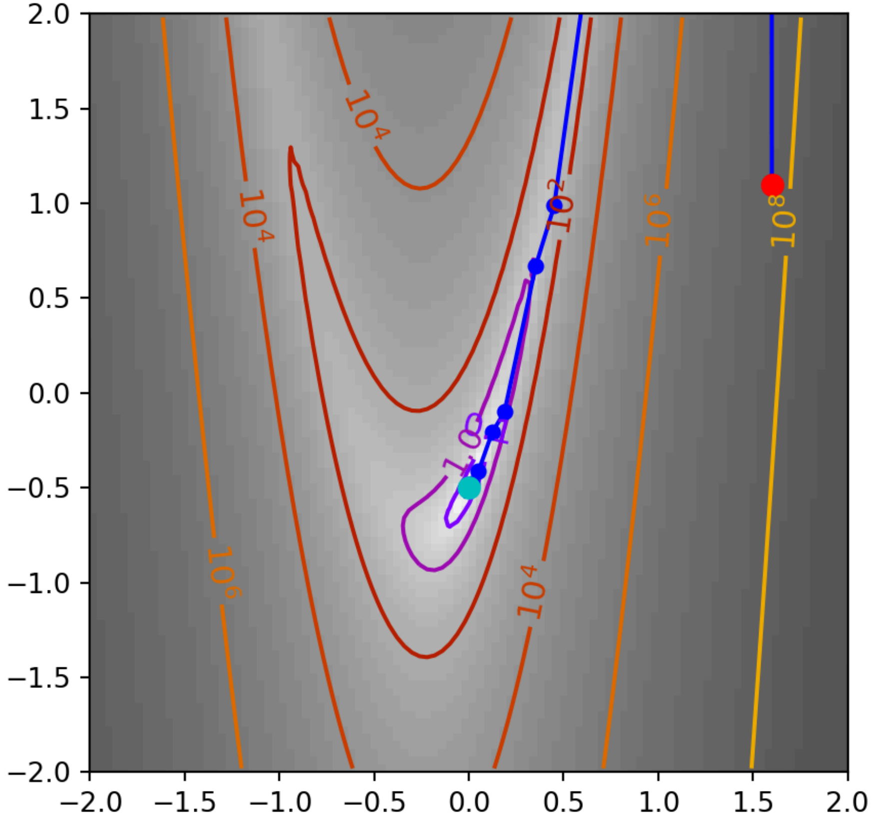


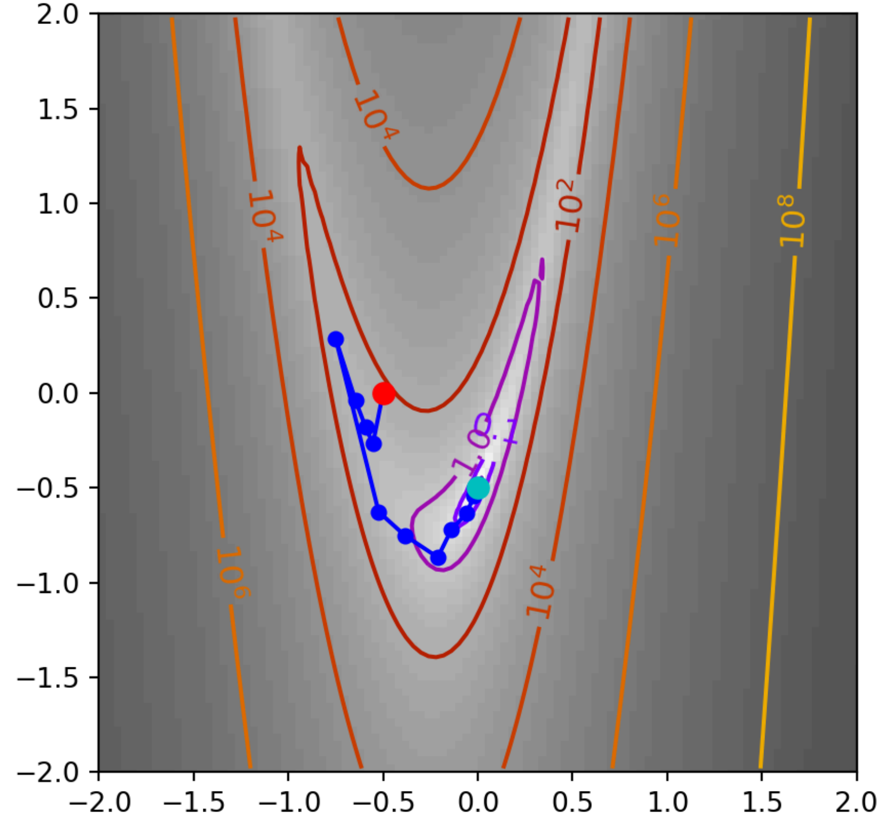


Rosenbrock function

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

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





Conjugate gradients

Conjugate gradients

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

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

$$\underset{x}{\arg\min} \; f(x) = \underset{x}{\arg\min} \; \frac{1}{2}\boldsymbol{x}^T \, \boldsymbol{A} \, \boldsymbol{x} - \boldsymbol{x}^T \boldsymbol{b}$$

since the solution is given by

$$\nabla f(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x} - \boldsymbol{b} = 0$$

A? Conjugate?

Taking things one step further we can also see that the matrix $\boldsymbol{A}$ is given by the hessian of $f(x)$

$$\nabla^2 f(x) = \boldsymbol{A}$$

Additionally recall, two non-zero vectors $\boldsymbol{u}$ and $\boldsymbol{v}$ are conjugate with respect to $\boldsymbol{A}$ if

$$\boldsymbol{u}^T \boldsymbol{A} \boldsymbol{v} = 0$$

Our goal then is to find n conjugate vectors $P = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_n\}$ to use as "optimal" step directions to traverse our objective function.

As line search

Restated, our goal is to find n conjugate vectors

$$\boldsymbol{p}_i^T \, \boldsymbol{A} \, \boldsymbol{p}_j = 0 \qquad \text{for all } i \neq j$$

and their coefficients (step sizes) such that our minimization solution ($\boldsymbol{x}^*$) is given by

$$\boldsymbol{x}^* = \boldsymbol{x}_0 + \sum_{i=1}^{n} \alpha_i \, \boldsymbol{p}_i$$

where the α_i's are our step sizes.

Algorithm Sketch

For the kth step:

- Define the residual $\boldsymbol{r}_k = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_k$
- Define conjugate vector $\boldsymbol{p}_k$ using current residual and all previous search directions

$$\boldsymbol{p}_k = \boldsymbol{r}_k - \sum_{i<k} \frac{\boldsymbol{r}_k^T \boldsymbol{A}\boldsymbol{p}_i}{\boldsymbol{p}_i^T \boldsymbol{A}\boldsymbol{p}_i} \boldsymbol{p}_i$$

- Define step size α_k using

$$\alpha_k = \frac{\boldsymbol{p}_k^T \boldsymbol{r}_k}{\boldsymbol{p}_k^T \boldsymbol{A}\boldsymbol{p}_k}$$

- Update $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{p}_k$

Algorithm in practice

Given x_0 we set the following initial values,

$$
\begin{aligned}
r_0 &= \nabla f(x_0) \\
p_0 &= -r_0 \\
k &= 0
\end{aligned}
$$

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

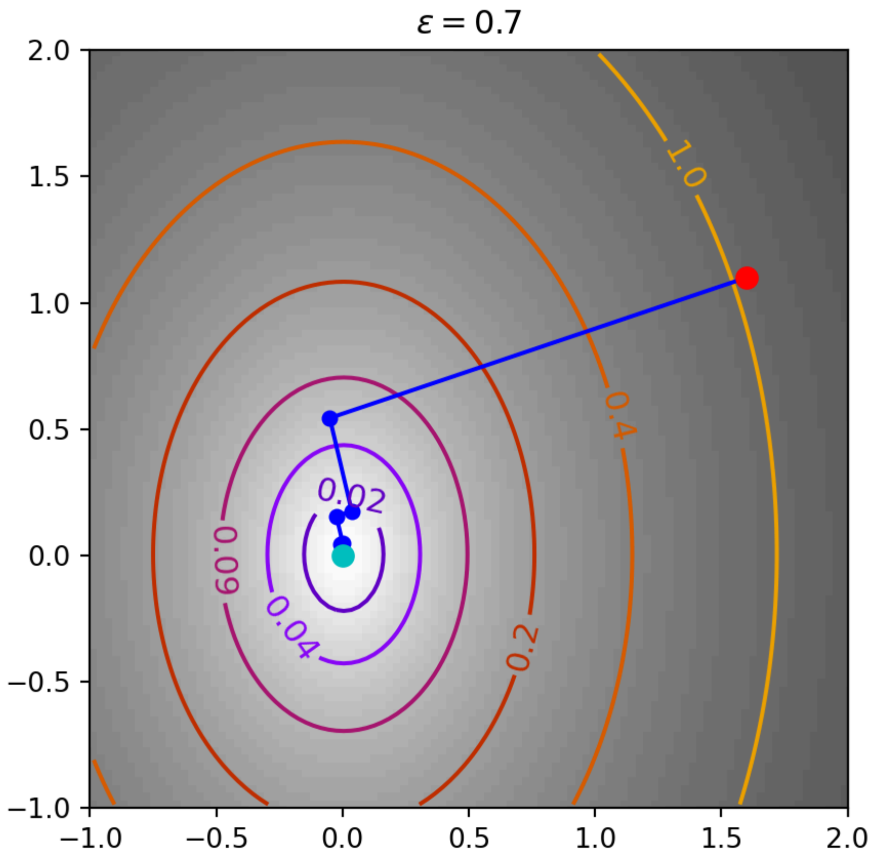
$$
\begin{aligned}
\alpha_k &= \frac{r_k^T \, p_k}{p_k^T \, \nabla^2 f(x_k) \, p_k} \\
x_{k+1} &= x_k + \alpha_k \, p_k \\
r_{k+1} &= \nabla f(x_{k+1}) \\
\beta_{k+1} &= \frac{r_{k+1}^T \, \nabla^2 f(x_k) \, p_k}{p_k^T \, \nabla^2 f(x_k) \, p_k} \\
p_{k+1} &= -r_{k+1} + \beta_k \, p_k \\
k &= k + 1
\end{aligned}
$$

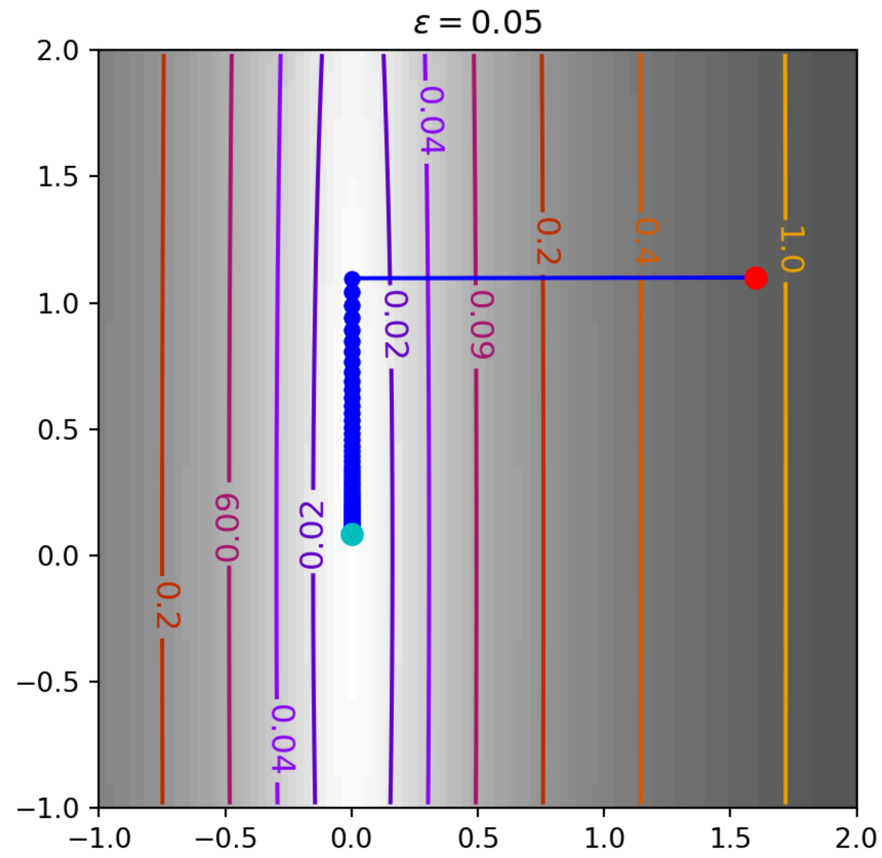
```
def conjugate_gradient(x, f, grad, hess, ma
    res = {"x": [x], "f": [f]}
    r = grad(x)
    p = -r

    for i in range(max_iter):
      H = hess(x)
      a = - r.T @ p / (p.T @ H @ p)
      x = x + a * p
      r = grad(x)
      b = (r.T @ H @ p) / (p.T @ H @ p)
      p = -r + b * p

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

      res["x"].append(x)
      res["f"].append(f(x))

    return res
```

Trajectory

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

```python
f, grad, hess = mk_quad(0.05)
opt = conjugate_gradient((1.6, 1.1), f, grac
plot_2d_traj((-1,2), (-1,2), f, title="$\\ep
```





Rosenbrock's function

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

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