lecture_9_optimisation_i

April 19, 2021

1 Lecture 9: Optimisation I

1.1 Data Fundamentals (H)

1.2 ## Parameters, objective functions, classification of optimisation problems

DF(H) - University of Glasgow - John Williamson - 2020/2021

2 Summary

By the end of this unit you should know: * what an objective function, constraint function and a parameter vector is in the context of optimisation * how to play the piano * the difference between discrete and continuous optimisation * convex and nonconvex optimisation and how to recognise them * what constrained optimisation is and the difference between soft and hard constraints * key properties of objective functions, including convexity and continuity * basic uses of optimisation and how to come up with objective functions * what linear least squares is * how iterative optimisation works * the principles of heuristic optimisation * the properties of random search, with the metaheuristics: locality, memory, temperature and population

LaTeX commands

$$\vec{x}\mathbb{R}$$

```
[ ]: import IPython.display
     IPython.display.HTML("""
     <script>
       function code_toggle() {
         if (code_shown){
           $('div.input').hide('500');
           $('#toggleButton').val('Show Code')
         } else {
           $('div.input').show('500');
           $('#toggleButton').val('Hide Code')
         }
         code_shown = !code_shown
       }

       $( document ).ready(function(){
         code_shown=false;
```

```
      $('div.input').hide()
    });
</script>
<form action="javascript:code_toggle()"><input type="submit" id="toggleButton"␣
  →value="Show Code"></form>""")
```

```
[1]: from __future__ import print_function, division
     import numpy as np
     import matplotlib as mpl

     import scipy.optimize      # scipy's optimisation routines
     %matplotlib inline
     import matplotlib.pyplot as plt
     plt.rc('figure', figsize=(8.0, 4.0), dpi=140)
```

3 Example: a synthesizer

4 Introduction to optimisation

4.1 What is optimisation?

Optimisation is the process of adjusting things to make them better. In computer science, we want to do this *automatically* by a algorithm. An enormous number of problems can be framed as optimisation, and there are a plethora of algorithms which can then do the automatic adjustment *efficiently*, in that they find the best adjustments in few steps. In this sense, *optimisation is search*, and optimisation algorithms search efficiently using mathematical structure of the problem space.

Optimisation is at the heart of machine learning; it is a critical part of all kinds of manufacturing and industrial processes, from shipbuilding to circuit design; it can even be used to automatically make scatterplot graphs easier to read.

4.1.1 One algorithm to rule them all: no special cases

Optimisation algorithms allow us to apply *standard algorithms* to an enormous number of problems. We don't have special cases for every specific problem; instead we formulate the problems so that generic algorithms can solve them. As a consequence, to apply optimisation algorithms, the problems must be specified formally. There is a real art in specifying problems so that optimisation can tackle them.

> What is often called "artificial intelligence" often comes down to optimisation. With optimisation we can specify problems, instead of solutions.

4.2 Parameters and objective function

Image credit: derived from](https://flickr.com/photos/doctorow/14638932732 "Enticingly technical synthesizer 1, Control Voltage, Mississippi Street, Portland, Oregon, USA") by gruntzooki CC (BY-SA)

There are two parts to an optimisation problem:

- **parameters**: the things we can adjust, which might be a scalar or vector or other array of values, denoted θ. The parameters exist in a **parameter space** – the set of all possible configurations of parameters denoted Θ. This space is often a **vector space** like $\mathbb{R}^n$, but doesn't need to be. For example, the set of all knob/slider positions on the synthesizer panel above could be considered points in a subset of a vector space. If the parameters do lie in a vector space, we talk about the **parameter vector** θ.

- **the objective function**: a function that maps the parameters onto a *single numerical measure* of how good the configuration is. $L(\theta)$. The output of the objective function is a single scalar. The objective function is sometimes called the *loss function*, the *cost function*, *fitness function*, *utility function*, *energy surface*, all of which refer to (roughly) the same concept. It is a quantitative ("objective") measure of "goodness".

The desired output of the optimisation algorithm is the parameter configuration that minimises the objective function.

Writing this mathematically, this is the `argmin` (the argument that produces the minimum value) of the objective function:

$$\theta^* = \text{argmin}_{\theta \in \Theta} L(\theta)$$

- θ^* is the configuration that we want to find; the one for which the objective function is lowest.
- Θ is the set of all possible configurations that θ could take on, e.g. $\mathbb{R}^N$.

Most optimisation problems have one more component:

- **constraints**: the limitations on the parameters. This defines a region of the parameter space that is feasible, the **feasible set** or **feasible region**. For example, the synthesizer above has knobs with a fixed physical range, say 0-10; it isn't possible to turn them up to 11. Most optimisation problems have constraints of some kind;

  "design a plane *(adjust parameters)* that flies as fast as possible *(objective function)*, and costs less than $180M *(constraints)*.

We usually think of the objective function as a **cost** which is *minimised*. Any maximisation problem can be reframed as a minimisation problem by a simple switch of sign, so this does not lose generality. If if we wanted to optimise the knob settings on our synthesizer to make a really good piano sound ("maximise goodness"), we could instead frame this as a problem of minimising the difference between the sound produced and the sound of a piano. We would, of course, need to have a **precise** way of measuring this difference; one that results in a single real number measure of cost.

4.2.1 Minimising differences

As in this example, it is common to have express problems in a form where the objective function is a **distance between an output and a reference is measured**. Not every objective function has this form, but many do.

That is, we have some function $y' = f(\vec{x}; \theta)$ that produces an output from an input $\vec{x}$ governed by a set of parameters θ, and we measure the difference between the output and some reference y (e.g. using a vector norm):

$$L(\theta) = \|y' - y\| = \|f(\vec{x}; \theta) - y\|$$

This is very common in **approximation problems**, where we want to find a function that approximates a set of measured observations. This is the core problem of machine learning.

Note that the notation $f(\vec{x}; \theta)$ just means that the output of f depends both on some (vector) input $\vec{x}$ and on a parameter vector θ. Optimisation only ever adjusts θ, and the vector $\vec{x}$ is considered fixed during optimisation (it might, for example, represent a collection of real-world measurements). In the synthesizer example, $\vec{x}$ might represent the keys pressed, which affect the sound but we *do not* optimise; while $\vec{\theta}$ represents the knob settings which affect that sound and we do optimise.

4.2.2   Evaluating the objective function

It may be **expensive** to evaluate the objective function. For example: * the computation might take a long time (invert a 10000x10000 matrix); * or it might require a real-world experiment to be performed (do the users like the new app layout?); * or it might be dangerous (which wire on the bomb should I cut next?); * or it might require data that must be bought and paid for (literally expensive).

In all cases, it will take some computational power to evaluate the objective function, and therefore will have a time cost.

This means that a *good* optimisation algorithm will find the optimal configuration of parameters with few queries (evaluations of the objective function). To do this, there must be mathematical **structure** which can help guide the search. Without any structure at all, the best that could be done would be to randomly guess parameter configurations and choose the lowest cost configuration after some number of iterations. This isn't typically a feasible approach.

4.3   Discrete vs. continuous

If the parameters are in a continuous space (typically $\mathbb{R}$), the problem is one of **continuous optimization**; if the parameters are discrete, the problem is **discrete optimization**. Continuous optimisation is usually easier because we can exploit the concept of **smoothness** and **continuity**.

4.3.1   Properties of optimisation

Every optimisation problem has two parts: * **Parameters**, the things that can be adjusted. * **Objective function**, which measures how good a particular set of parameters are.

An optimisation problem usually also has: * **Constraints**, that define the feasible set of parameters.

The **objective function** is a function *of the parameters* which returns a *single scalar value*, representing how good that parameter set is.

4.4 Throwing a stone

For example, if I wanted to optimise how far I could throw a stone, I might be able to adjust the throwing angle. This is the *parameter* I could tweak (just one parameter $\theta = [\alpha]$, in this case).

The objective function must be a function which depends on this parameter. I would have to *simulate* throwing the ball to work out how far it went and try and make it go further and further.

```python
[2]: import numpy as np

def L(theta):
    # L *must* depend on the parameters I can adjust
    # in this case, there is only one; the throw angle

    # pull out the angle, convert from degrees to radians
    angle = np.radians(theta[0])

    # initial throw position x=0m, y=1m
    pos = np.array([0.0, 1.0])

    # initial throw velocity, depends on angle
    vel = np.array([np.cos(angle), np.sin(angle)])

    posns = []
    # simulate throwing the ball, until it hits the ground
    while pos[1] >= 0:
        pos += vel
        vel[1] -= 0.005  # gravity
        vel -= 0.01 * vel  # air resistance
        posns.append(list(pos))
    posns = np.array(posns)

    plt.gca().plot(posns[:, 0], posns[:, 1], "k", lw=0.1)

    # return how far our throw went
    # remember that we want to minimise
    # our objective function, so this value
    # must get *lower* as our throw gets longer

    return -np.abs(pos[0])
```

```python
[3]: import scipy.optimize

fig, ax = plt.subplots()
ax.set_xlabel("Horizontal location (m)")
ax.set_ylabel("Vertical location (m)")
# use a built in optimiser to solve this
res = scipy.optimize.minimize(L, [65.0], method="nelder-mead")
```

```
print(res)
```

```
final_simplex: (array([[30.85437012],
       [30.8544693 ]]), array([-70.14102702, -70.14095448]))
          fun: -70.14102701610933
      message: 'Optimization terminated successfully.'
         nfev: 50
          nit: 24
       status: 0
      success: True
            x: array([30.85437012])
```
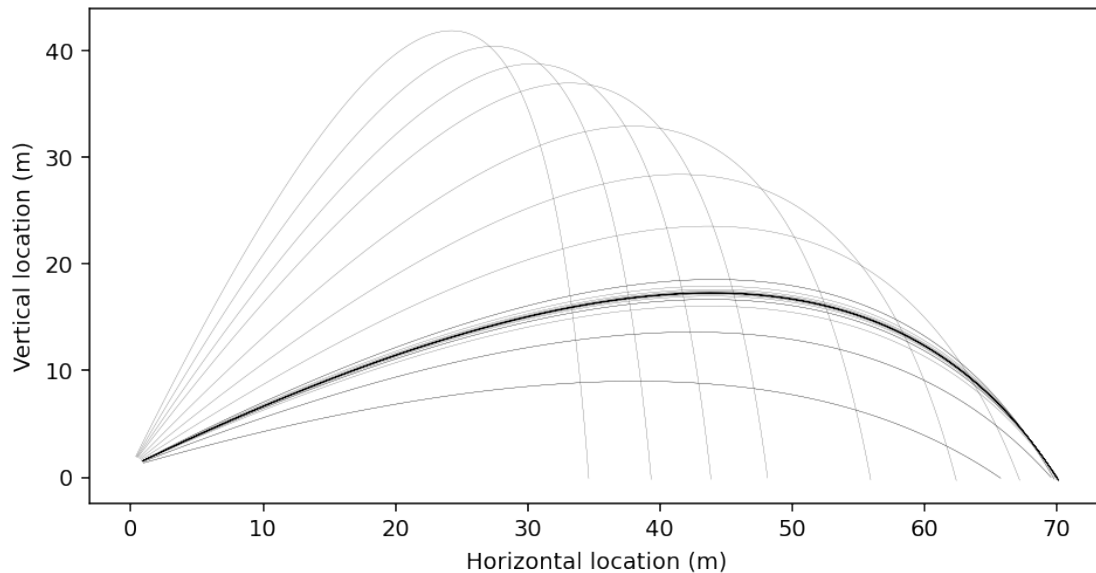

We will get a result that throwing the ball at around 30 degrees to the horizontal will give the longest throw. Note that the answer will only be 45 degrees if there is no air resistance; something that would be annoying to work out but is trivial to optimise for. If air resistance is higher, then the throw should be more horizontal.

4.4.1 Focus: continuous optimisation in real vector spaces

This course will focus on optimisation of continuous problems in $\mathbb{R}^n$. That is

$$\theta \in \mathbb{R}l^n = [\theta_1, \theta_2, \ldots, \theta_n],$$

and the optimisation problem is one of:

$$\theta^* = \text{argmin}_{\theta \in \mathbb{R}l^n} L(\theta), \text{subject to constraints}$$

This it the problem of searching a continuous vector space to find the point where $L(\theta)$ is smallest. We will typically encounter problems where the objective function is *smooth* and *continuous* in this

6

vector space; note that the parameters being elements of a continuous space does not necessarily imply that the objective function is continuous in that space.

Some optimisation algorithms are **iterative**, in that they generate successively better approximations to a solution. Other methods are **direct**, like linear least squares (which we'll briefly discuss), and involving finding a minimum exactly in one step. We will focus primarily on **iterative, approximate** optimisation in this course.

A function of space   The objective function maps points in space to values; i.e. it defines a curve/surface/density/etc. which varies across space. We want to find, as quickly as possible, a point in space where this is as small as possible, without going through any "walls" we have defined via constraints.

- **Problem** Find the median of a >1D dataset. The standard median is computed by sorting and then selecting the middle element (with various rules for even sized datasets). This doesn't work for higher dimensions, and there is no straightforward direct algorithm. But there is an easy definition of the median: it is the vector that minimises the sum of distances to all vectors in the dataset.

A very simple optimisation example is to find a point that minimises the distance to a collection of other points (with respect to some norm). We can define:

- **parameters** $\theta = [x, y \ldots]$, a position in 2D.
- **objective function** the sum of distances between a point and a collection of target points $\vec{x}_i$:

$$L(\theta) = \sum_i ||\theta - \vec{x}_i||_2$$

This will try and find a point in space (represented as θ) which minimises the distances to the target points. We can solve this, starting from some random initial condition (guess for θ):

```python
[4]:  # some random points
      targets = np.random.normal(0,1,(50,2)) + np.array([0.5, -1])

      # This is the objective function
      def loss(x):
          # how far are we away from being minimal distance to every point?
          return np.sum(np.linalg.norm(x[:,None] -
                                       targets.T, axis=1,
                                       ord=np.inf))
```

```python
[5]:
      # Initial state -- guess for theta
      theta = np.random.uniform(-2, 2, 2) # 64 random 2D points

      # Plot the original data
      ###############
      fig = plt.figure(figsize=(8,8))
      ax = fig.add_subplot(1,1,1)
      ax.scatter(targets[:,0], targets[:,1], c='C0', s=20, label="Target points",
        ↪alpha=1.0)
```

7

```python
ax.scatter(theta[0], theta[1], c='C2', s=30, label="Original $\\theta$", alpha=1.
 ↪0)
ax.set_xlabel("$\\theta_0$")
ax.set_ylabel("$\\theta_1$")
last_theta = theta

def plot_intermediate(x):
        ax.plot([last_theta[0], x[0]], [last_theta[1], x[1]], c='C1',alpha=0.5,␣
 ↪marker='o')
        last_theta[:] = x # hack to show a history
##################

# minimise the loss using a standard optimizer
result = scipy.optimize.minimize(loss, theta.ravel(), method='Nelder-Mead',␣
 ↪callback=plot_intermediate, )

###########
# plot the final positions
ax.scatter(result.x[0], result.x[1], c='C2', s=160, marker='*', label="Optimised␣
 ↪$\\theta^*$ (estimated median)", zorder=100)
ax.set_title("Optimising a point for equal distance")
ax.legend()
ax.set_aspect(1.0)
###########
```

Optimising a point for equal distance
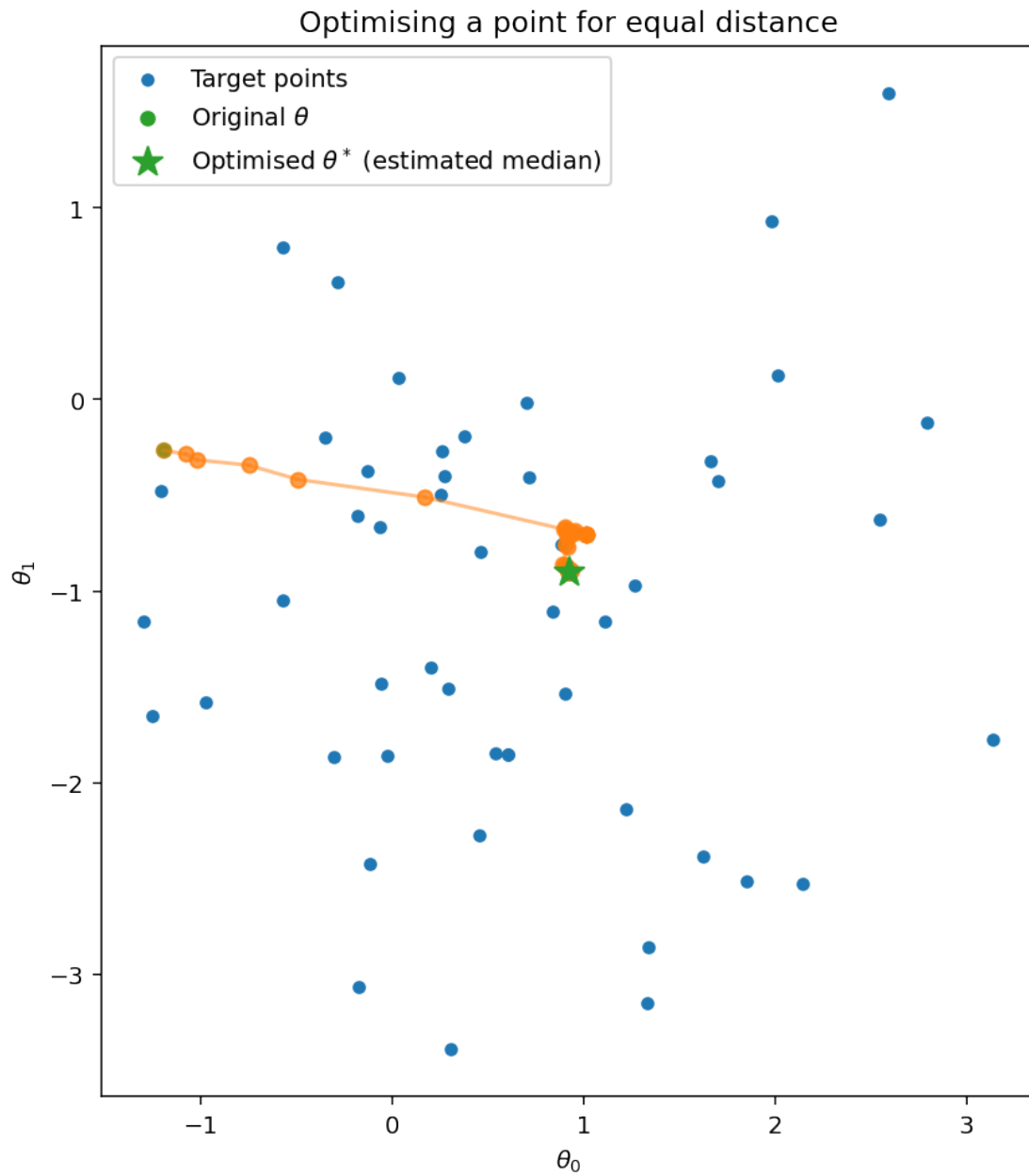
```
[6]:  ## Show the objective function as a function of parameters
      space = np.linspace(-4,4,50)
      xm, ym = np.meshgrid(space, space)
      pts = np.stack([xm.ravel(), ym.ravel()]).T
      zm = []

      # compute the objective function at a grid
      # of different parameter values
```

```
for pt in pts:
    zm.append(loss(pt))
%matplotlib inline

## plot in 3D
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure(figsize=(12,8))
ax = fig.add_subplot(111, projection='3d')
zm = np.array(zm).reshape(xm.shape)
ax.plot_surface(xm,ym,zm,cmap='viridis')
ax.set_zlabel("$L(\\theta)$")
ax.set_xlabel("$\\theta_0$")
ax.set_ylabel("$\\theta_1$")
ax.set_title("Objective function as a function of the parameters")
```

[6]: Text(0.5, 0.92, 'Objective function as a function of the parameters')

Objective function as a function of the parameters
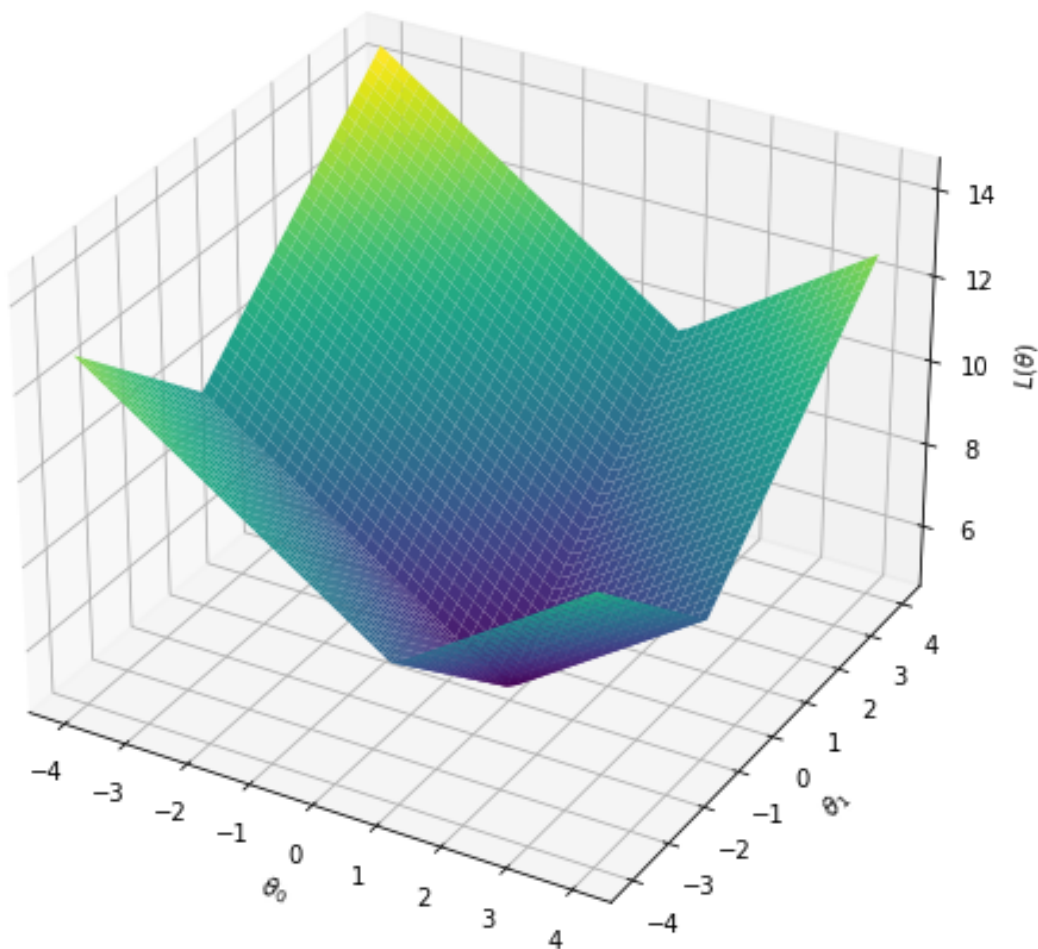
4.4.2   An example of optimisation in $\mathbb{R}^N$

We can work in higher dimensions just as easily. A slightly different problem is to try and find a layout of points in such that the points are **evenly spaced** (with respect to some norm). In this case we have to optimise a whole collection of points, which we can do by rolling them all up into a single parameter vector.

We can define:

- **parameters** $\theta = [x_1, y_1, x_2, y_2, \dots]$, an array of positions of points in 2D. Note: we have "unpacked" a sequence of 2D points into a higher dimensional vector, so that a *whole config-uration* of points is a single point in a vector space.
- **loss function** the sum of squares of differences between the Euclidean pairwise distances between points and some target distance:

$$\sum_i \sum_j (\alpha - ||x_i - x_j||_2)^2$$

This will try and find a configuration of points that are all α units apart.

We again start from some random initial condition, with 64 2D points; a 128 dimensional θ.

```
[7]:  # This is the objective function
      # How far are we away from optimality?
      def loss(x):
          # reshape back to the original shape
          x = x.reshape(theta.shape)
          distances = np.linalg.norm(x[:,:,None] - x[:,:,None].T, axis=1, ord=2)
          # compute difference between pairwise distance and 0.8
          return np.sum((0.8-distances)**2)

      # Initial state -- guess for theta
      theta = np.random.uniform(0, 1, (64,2)) # 64 random 2D points

      # Plot the original data
      fig = plt.figure(figsize=(8,8))
      ax = fig.add_subplot(1,1,1)
      ax.scatter(theta[:,0], theta[:,1], c='C0', s=30, label="Original $\\theta$",
        →alpha=0.5)

      def plot_intermediate(x):
          theta_star = x.reshape(theta.shape)
          ax.scatter(theta_star[:,0], theta_star[:,1], c='C1', s=20, alpha=0.1)

      # minimise the loss (we have to convert to a 1D array for minimize() to work)
      result = scipy.optimize.minimize(loss, theta.ravel(), method='BFGS',
        →callback=plot_intermediate)
```
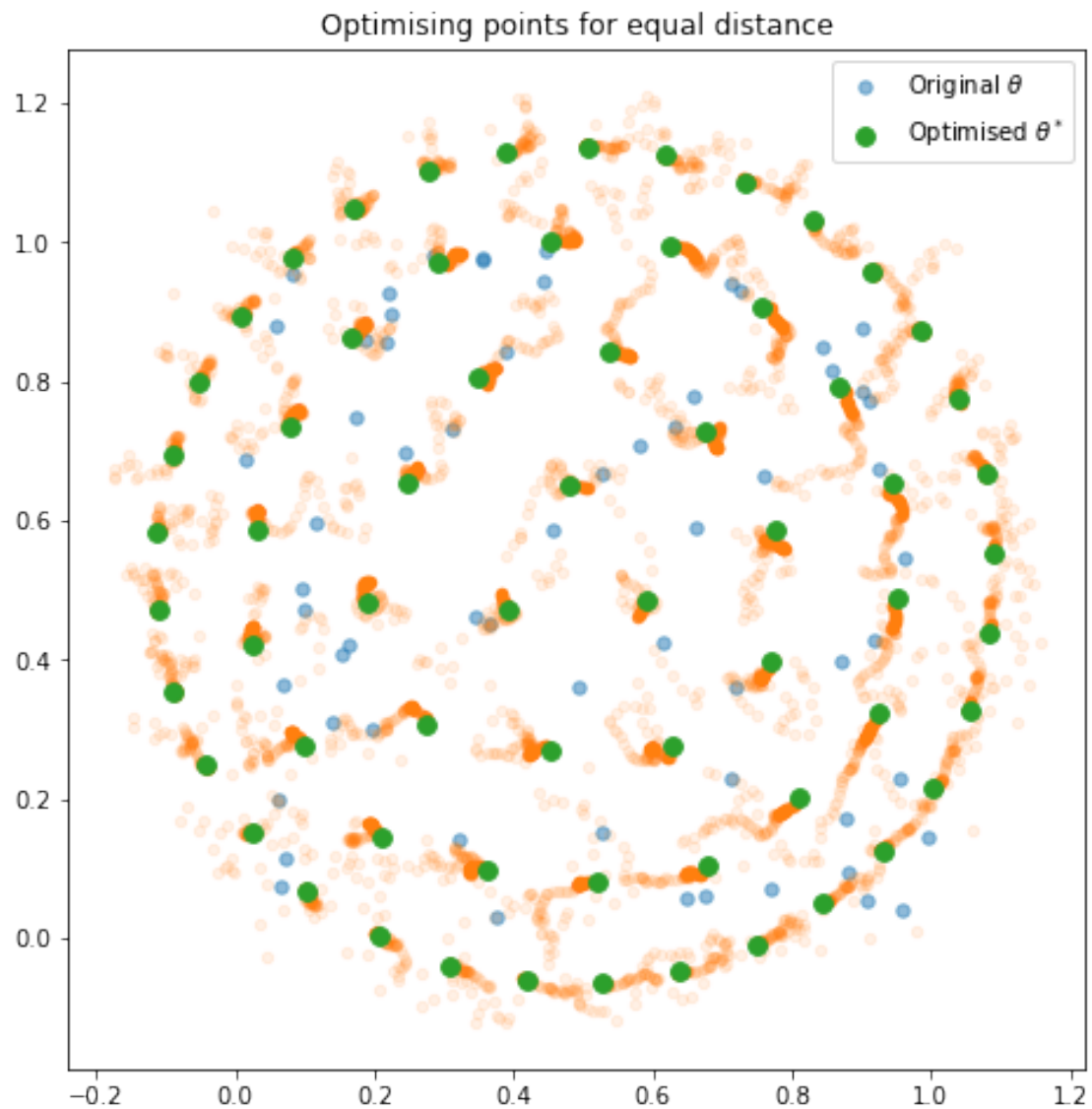
11

```python
# reshape the result back to an array
theta_star = result.x.reshape(theta.shape)

# plot the final positions
ax.scatter(theta_star[:,0], theta_star[:,1], c='C2', s=60, label="Optimised␣
 ↪$\\theta^*$")
ax.set_title("Optimising points for equal distance")
ax.legend()
ax.set_aspect(1.0)
```


Optimising points for equal distance

5 Constrained optimisation

If a problem has constraints on the parameters beyond purely minimising the objective function then the problem is **constrained optimisation**. For example, in the synthesizer above, it's not much use if the optimal solution requires one the knobs to be turned to an impossible value; the parameter space is limited in extent on every dimension. This limits the **feasible set** of the parameters.

A constrained optimisation might be written in terms of an equality constraint:

$$\theta^* = \texttt{argmin}_{\theta \in \Theta} L(\theta) \text{ subject to } c(\theta) = 0,$$

or an inequality:

$$\theta^* = \texttt{argmin}_{\theta \in \Theta} L(\theta) \text{ subject to } c(\theta) \leq 0,$$

where $c(\theta)$ is a function that represents the constraints.

- An **equality** constraint can be thought of as constraining the parameters to a *surface*, to represent a tradeoff. For example, $c(\theta) = \|\theta\|_2 - 1$ forces the parameters to lie on the surface of a unit sphere. An equality constraint might be used when trading off items where the total value must remain unchanged (e.g. the payload weight in a satellite might be fixed in advance).

- An **inequality** constraint can be thought of as constraining the parameters to a *volume*, to represent bounds on the values. For example, $c(\theta) = \|\theta\|_\infty - 10$ forces the parameters to lie within a box extending (-10, 10) around the origin – perhaps the range of the knobs on the synthesizer.

Common constraint types A **box constraint** is a simple kind of constraint, and is just a requirement that θ lie within a box inside R^n; for example, that every element $0 < \theta_i < 1$ (all parameters in the positive unit cube) or $\theta_i > 0$ (all parameters in the positive **orthant**). This is an inequality constraint with a simple form of $c(\vec{\theta})$. Many optimisation algorithms support box constraints.

A **convex constraint** is another simple kind of constraint, where the constraint is a collection of inequalities on a convex sum of the parameters θ. Box constraints are a specific subclass of convex constraints. This is equivalent to the feasible set being limited by the intersection of many of **planes/hyperplanes** (possibly an infinite number in the case of curved convex constraints).

Unconstrained optimization does not apply any constraints to the parameters, and any parameter configuration in the search space is possible. In many problems, pure unconstrained optimisation will lead to unhelpful results ("the airplane will get the best lift if the wing is two hundred miles long" – which might be true but impossible to construct).

5.1 Constraints and penalties

Unconstrained optimisation rarely gives useful answers on its own. Consider the example of the airfoil. Increasing lift might be achieved by making the airfoil length longer and longer. At some point, this might become physically impossible to build.

Although we often represent θ as being in $\mathbb{R}^N$, the feasible set is typically not the entire vector space. There are two approaches to deal with this:

Constrained optimisation

- Use an optimisation algorithm that supports hard constraints inherently. This is straightforward for certain kinds of optimisation, but trickier for general optimisation.Typically constraints will be specified as either a **convex region** or a simple (hyper)rectangular region of the space (a **box constraint**).
- **Pros**:
  - Guarantees that solution will satisfy constraints.
  - May be able to use constraints to speed up optimisation.
- **Cons**:
  - may be less efficient than unconstrained optimization.
  - Fewer algorithms available for optimisation.
  - may be hard to specify feasible region with the parameters available in the optimiser.

Soft constraints

- Apply penalties to the objective function to "discourage" solutions that violate the constraints. This is particularly appropriate if the constraints really are soft (it doesn't perhaps matter if the maximum airfoil length is 1.5m or 1.6m, but it can't be 10m). In this case, the penalties are just terms added to the objective function. The optimiser stays the same, but the objective function is modified.

$$L(\theta') = L(\theta) + \lambda(\theta),$$

where $\lambda(\theta)$ is a **penalty function** with an increasing value as the constraints are more egregiously violated.

- **Pros**
  - any optimiser can be used
  - can deal with *soft* constraints sensibly
- **Cons:**
  - may not respect important constraints, particularly if they are very sharp
  - can be hard to formulate constraints as penalties
  - cannot take advantage of efficient search in constrained regions of space

5.2 Relaxation of objective functions

It can be much harder to solve discrete optimization and constrained optimization problems efficiently; some algorithms try and find similar continuous or unconstrained optimization problems to solve instead. This is called **relaxation**; a **relaxed** version of the problem is solved instead of the original hard optimization problem. For example, sometimes the constraints in a problem can be absorbed into the objective function, to convert a constrained problem to an unconstrained problem.

5.2.1 Penalisation

Penalisation refers to terms which augment an objective function to minimise some other property of the solution, typically to approximate constrained optimisation. This is widely used in approximation problems to find solutions that **generalise well**; that is which are tuned to approximate some data, but not *too* closely.

This is a relaxation of a problem with hard constraints (which needs specialised algorithms) to a problem with a simple objective function which works with any objective function. If you have encountered **Lagrange multipliers** before, these are an example of a relaxation of hard constraints to penalty terms.

5.2.2   Penalty functions

A **penalty function** is just a term added to an objective function which will disfavour "bad solutions".

We can return to the stone throwing example, and extend our model. Say I can control the angle of a stone throw; perhaps I can also control how hard I throw it. But there is a maximum limit to my strength. This is a constraint (an inequality constraint, which limits the maximum value of the strength parameter).

- Objective function: how far away does the stone land? $L(\theta) = \text{throw\_distance}(\theta)$
- Parameters: angle of the throw α and strength of the throw v (exit velocity), $\theta = [\alpha, v]$
- Constraint: strength of throw $0 \le v \le v_k$, more than zero and less than some maximum strength.

There are two options: * Use a constrained optimisation algorithm, which will not even search solutions which exceed the maximum strength. * Change the objective function to make overstrenuous throwing unacceptable.

```
[8]:  def throw(theta):

          angle = np.radians(theta[0])
          strength = theta[1]   # now we have a strength
          # initial throw position x=0m, y=1m
          pos = np.array([0.0, 1.0])

          # initial throw velocity, depends on angle and strength
          vel = np.array([np.cos(angle), np.sin(angle)]) * strength
          posns = []
          # simulate throwing the ball, until it hits the ground
          while pos[1] >= 0:
              pos += vel
              vel[1] -= 0.005   # gravity
              vel -= 0.005 * vel   # air resistance
              posns.append(np.array(pos))

          # plot trajectory
          posns = np.array(posns)
          ax.plot(posns[:, 0], posns[:, 1], "k", lw=0.1)

          return -np.abs(pos[0])
```

Option 1: constrained optimisation   Use a (pre-existing) algorithm which already supports constraints directly. Guarantees solutions will lie inside bounds.

```
[9]: max_strength = 1.0

     import scipy.optimize

     fig = plt.figure()
     ax = fig.add_subplot(1, 1, 1)
     ax.set_xlabel("Horizontal location (m)")
     ax.set_ylabel("Vertical location (m)")

     ## optimise using an optimiser that has bounds
     ## will never generate solutions outside of these
     res = scipy.optimize.minimize(
         throw, [65.0, 0.1], method="tnc",
         bounds=[[0, 90],
                                              [0, max_strength]]
     )

     print(
         "Angle {angle:.1f} degrees, distance={m:.1f}m".format(
             angle=res.x[0], m=np.abs(res.fun)
         )
     )
```
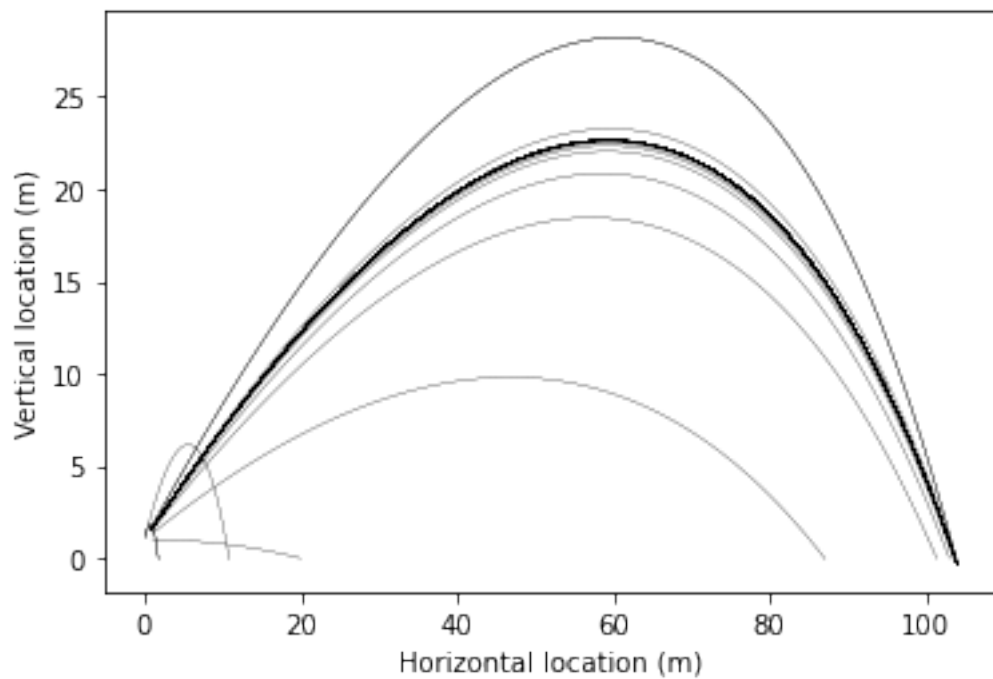
Angle 32.4 degrees, distance=104.0m

Option 2: add a penalty term

$$L'(\theta) = L(\theta) + \lambda(theta)$$

```
[10]: def penalty(strength):
          # start applying constraint just before we hit
          # the hard limit
          cutoff = max_strength * 0.90
          # increase loss rapidly as we excced max strength
          return np.where(strength > cutoff,
                          (cutoff - strength) ** 2 * 500, 0)
```

```
[11]: fig = plt.figure()
      ax = fig.add_subplot(1, 1, 1)
      xs = np.linspace(0, 2.0, 100)
      ax.plot(xs, penalty(xs))
      ax.set_title("Penalty function")
      ax.axvline(1.0, c="k", alpha=0.1)

      ax.set_xlabel("Strength")
      ax.set_ylabel("Penalty")
```

```
[11]: Text(0, 0.5, 'Penalty')
```
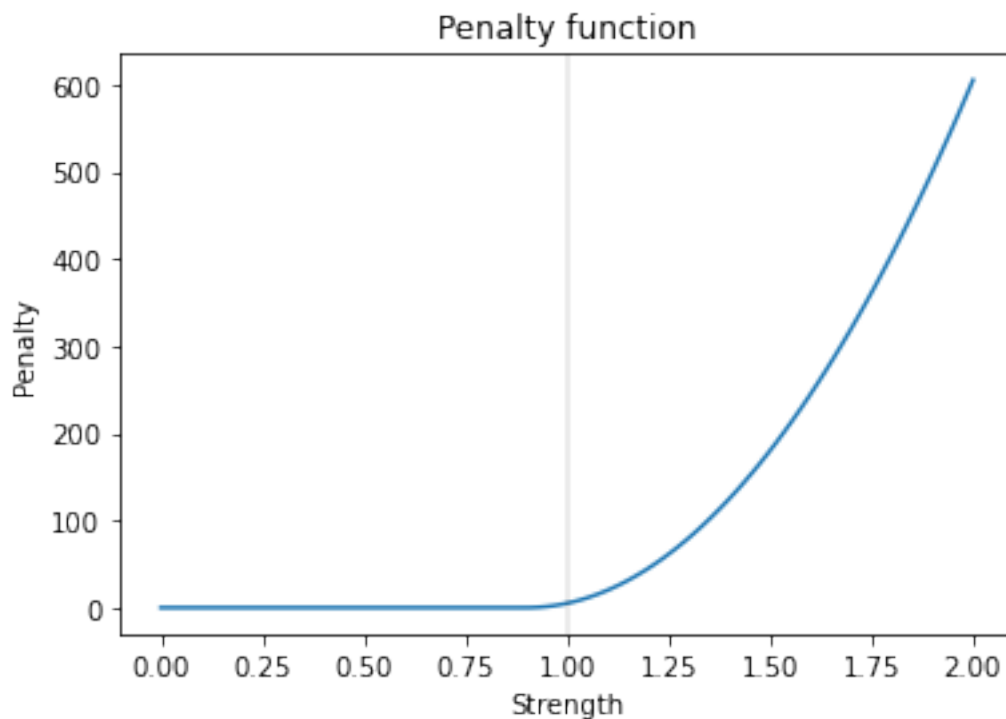
```
[12]: # super simple
      # add the penalty to the objective function
      def throw_penalised(theta):
          loss = throw(theta) + penalty(theta[1])
          return loss
```

```
[13]: fig = plt.figure()
      ax = fig.add_subplot(1, 1, 1)
      ax.set_xlabel("Horizontal location (m)")
      ax.set_ylabel("Vertical location (m)")

      # no constraints this time
      res = scipy.optimize.minimize(throw_penalised,
                                    [65.0, 0.1], method="nelder-mead")



      print(
          "Angle {angle:.1f} degrees, distance={m:.1f}m".format(
              angle=res.x[0], m=np.abs(res.fun)
          )
      )
```

Angle 35.2 degrees, distance=101.1m
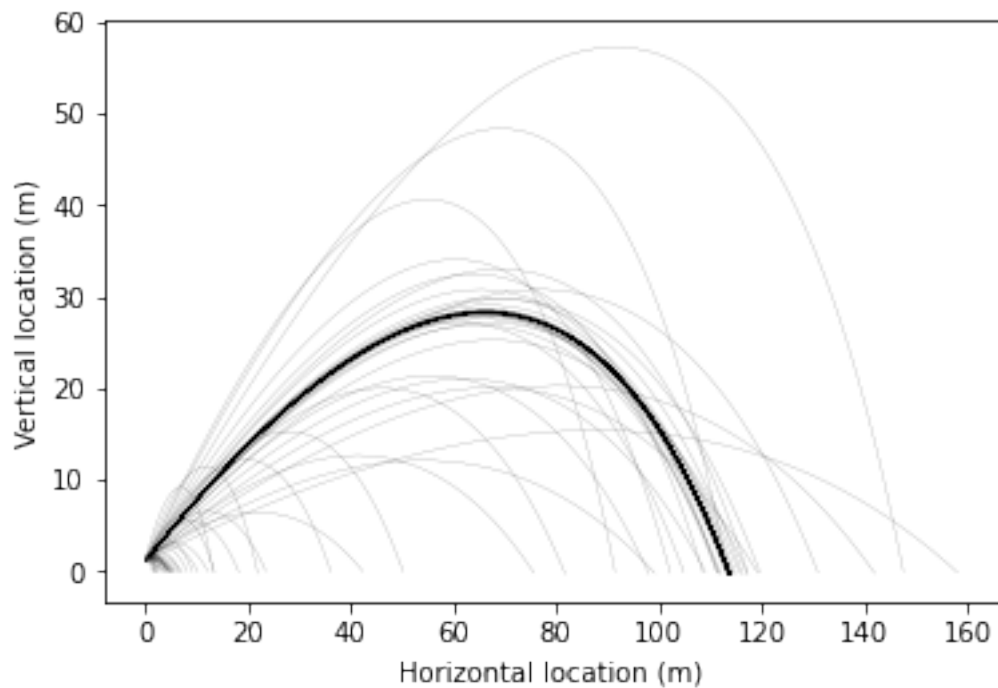
6 Properties of the objective function

6.1 Convexity, global and local minima

An objective function may have **local minima**. A **local minimum** is any point where the objective functions increases in every direction around that point (that parameter setting). Any change in the parameters at that point increases the objective function.

An objective function is **convex** if it has a *single, global minimum*. For example, every quadratic function is a parabola (in any number of dimensions), and thus has exactly one minimum. Other functions might have regions that have local minimums but which **aren't** the smallest possible value the function could take on.

Convexity implies that finding any minimum is equivalent to finding the global minimum – the guaranteed best possible solution. This minimum is the global minimum. In a convex problem, if we find a minimum, we can stop searching. If we can show there *is no minimum*, we can also stop searching.

Images: examples of convex and non-convex objective functions

6.1.1 Convex optimisation

If the objective function is **convex** *and* any constraints form convex portions of the search space, then the problem is **convex optimisation**. There are very efficient methods for solving convex optimisation problems, even with tens of thousands of variables. These include:

- the constraints and objective function are linear (**linear programming**)
- quadratic objective function and linear constraints (**quadratic programming**)
- or a some specialised cases (**semi-quadratic programming**, **quadratically constrained quadratic program**).

These are incredible powerful algorithms for solving these specific classes of optimisation problems.

Nonconvex problems require the use of **iterative** methods (although ways of *approximating* nonconvex problems with convex problems do exist).

6.1.2 Continuity

An objective function is **continuous** if for some very small adjustment to θ there is an *arbitrarily* small change in $L(\theta)$. This means that there will never be "nasty surprises" if we move slowly enough through the space of θ; no sudden jumps in value.

If a function is discontinuous, local search methods are not guaranteed to converge to a solution. Optimisation for discontinuous objective functions is typically much harder than for continuous functions. This is because there could be arbitrary changes in the objective function for any adjustment to the parameters.

Continuity is what can make continuous optimisation easier than discrete optimisation. As we will see next week, being continuous and **differentiable** makes continuous optimisation even more powerful.

7 Algorithms

7.1 Direct convex optimisation: least squares

Sometimes we have an optimisation problem which we can specify such that the solution can be computed in one step. An example is **linear least squares**, which solves objective functions of the form:

$$\texttt{argmin}_x L(\vec{x}) = \|A\vec{x} - \vec{y}\|_2^2,$$

that is, it finds $\vec{x}$ that is closest to the solution $A\vec{x} = \vec{y}$ in the sense of minimising the squared L_2 norm. The squaring of the norm just makes the algebra easier to derive.

This equation is **convex** – it is a quadratic function and even in multiple dimensions it must have a single, global minimum, which can be found directly. The reason we **know** it is convex is that it has no terms with powers greater than 2 (no x^3 etc.) and so is quadratic. Quadratic functions only ever have zero or one minimum.

7.1.1 Line fitting

We will examine this process for the simplest possible **linear regression** example: finding gradient m and offset c for the line equation

$$y = mx + c$$

such that the squared distance to a set of observed (x, y) data points is minimised. This is a search over the $\theta = [m, c]$ space; these are the parameters. The objective function is $L(\theta) = \sum_i (y - mx_i + c)^2$, for some known data points $[x_0, y_0], [x_1, y_1]$, etc.

We can solve this directly using the **pseudo-inverse** via the SVD. This is a problem that can be solved directly in one step.

For demonstration, we will use a line with the equation

$$y = 2x + 1, m = 2, c = 1$$

where we have a collection of noisy observations from this function.

```
[14]:  ## generate some random data, with a linear trend
       # the linear regression problem; find m and c in the equation y=mx+c
       line_x = np.sort(np.random.normal(0, 1, (20,)))
       gradient, offset = 2, 1   # these are the "true" parameters that generated this
        ↪data

       # 2x + 1 and a bit of noise
       line_y = (
           gradient * line_x + offset + np.random.normal(0, 0.5 * np.abs(line_x),
        ↪line_x.shape)
       )

       # loss function
       def loss(theta):
```

```
        # sum of squares of differences with estimated parameters
        e = np.sum(((line_x * theta[0] + theta[1]) - line_y) ** 2)
        return e
```

[15]:
```
## Solution via SVD
# Using the "Vandermonde matrix" approach
## NOTE: You don't have to understand how this code works

A = np.stack([line_x ** 0, line_x ** 1]).T
A_plus = np.linalg.pinv(A)
regression = A_plus @ line_y
```

[16]:
```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.scatter(line_x, line_y, label="Data")
xt = np.linspace(-3, 3, 100)
ax.plot(xt, xt * regression[1] + regression[0], c="C1", label="Regression")
for i in range(len(line_x)):
    ax.plot(
        [line_x[i], line_x[i]],
        [line_x[i] * regression[1] + regression[0], line_y[i]],
        "C0",
    )

ax.legend()
ax.set_title("Some data points and a least-squares linear regression")
ax.text(-2, 5, "Estimate y=%.2fx+%.2f" % (regression[1], regression[0]))
```
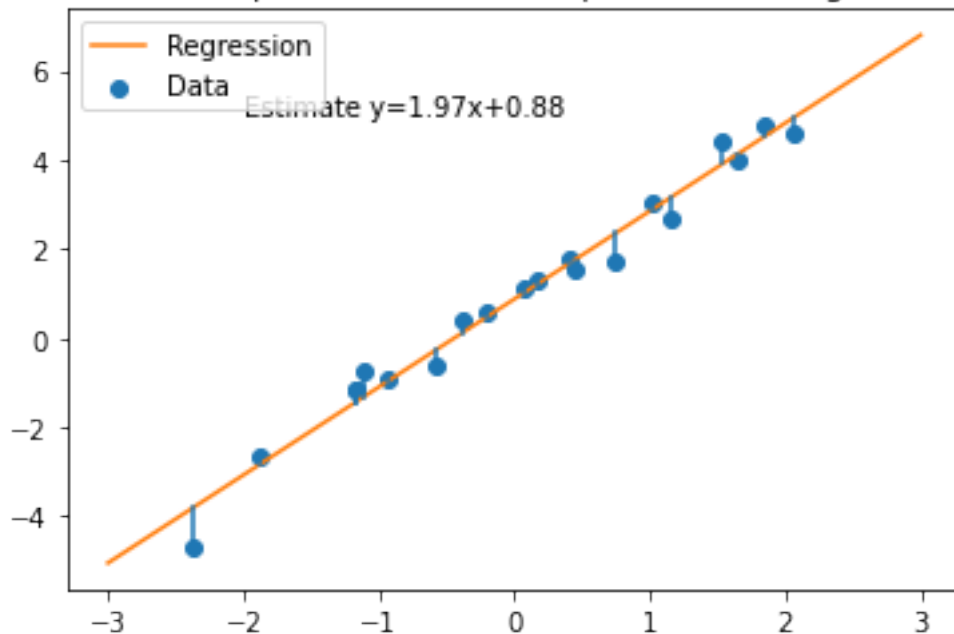
[16]: Text(-2, 5, 'Estimate y=1.97x+0.88')

Some data points and a least-squares linear regression

Estimate y=1.97x+0.88

```
[17]:  # now manually compute the least squares fit loss function across a reasonable␣
       ↪area

       # generate a grid
       div = np.linspace(-5, 5, 50)
       mx, my = np.meshgrid(div, div)

       # we need to compute distance for *each* data point (for 20 of them)

       mmx = np.tile(mx[:, :, None], (1, len(line_x)))
       mmy = np.tile(my[:, :, None], (1, len(line_x)))

       # cost function: |y - f'(x)|^2
       cost_fn = np.sum(((line_x * mmy + mmx) - line_y) ** 2, axis=2)

       # show the contours
       fig = plt.figure()
       ax = fig.add_subplot(1, 1, 1)

       res = ax.contourf(my, mx, np.log(cost_fn), 20)

       ax.contour(my, mx, np.log(cost_fn), 19, colors="k", linewidths=0.5)
       fig.colorbar(res)
       ax.set_title("(log) Objective function for regression $y = mx + c$")
```
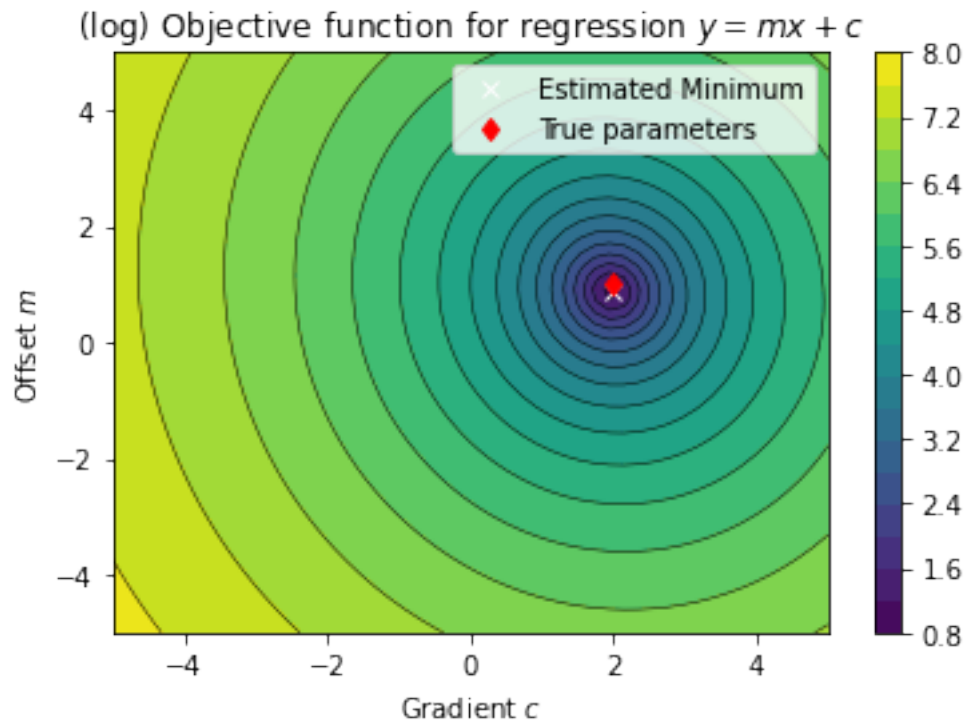
22

```
ax.set_xlabel("Gradient $c$")
ax.set_ylabel("Offset $m$")

ax.plot(regression[1], regression[0], "wx", label="Estimated Minimum")
ax.plot(gradient, offset, "rd", label="True parameters")
plt.legend()
```

[17]: <matplotlib.legend.Legend at 0x1cab4574a30>

7.2 Iterative optimisation

Iterative optimisation involves making a series of steps in parameter space. There is a **current parameter vector** (or collection of them) which is adjusted at each iteration, hopefully decreasing the objective function, until optimisation terminates after some **termination criteria** have been met.

Iterative optimisation algorithm:

1. choose a starting point x_0
2. while objective function changing
     1. adjust parameters
     2. evaluate objective function
     3. if better solution found than any so far, record it
3. return best parameter set found

7.3 Regular search: grid search

Grid search, is a straightforward but inefficient optimisation algorithm for multidimensional problems. The parameter space is simply sampled by equally dividing the feasible set in each dimension, usually with a fixed number of divisions per dimension.

The objective function is evaluated at each θ on this grid, and the lowest loss θ found so far is tracked. This is simple, and can work for 1D optimisation problems. It is sometimes used to optimise *hyperparameters* of machine learning problems where the objective function may be complex but finding the absolute minimum isn't essential.

```python
[18]: import itertools
      # history lets us track the best solutions as we find them
      # import utils.history; import imp; imp.reload(utils.history)
      from utils.history import History, linear_regression_plot

      def grid_search(L, ranges, divs):
          """L: loss function
          ranges: Parameter ranges for each dimension (e.g. [[0,1], [-1,1], [0,2]])
          divs: division per range
          """
          o = History()
          divisions = [np.linspace(r[0], r[1], divs) for r in ranges]
          for theta in itertools.product(*divisions):
              o.track(theta, L(theta))
          return o.finalise()
```

```python
[22]: grid_results = grid_search(loss, [[-5,5], [-5,5]], 15)
```

```python
[23]: linear_regression_plot(grid_results, gradient, offset, line_x, line_y,␣
      ↪opt_title="Grid search")
```

24

Grid search over possible line configurations

Parameters

Predictions

Objective function

25
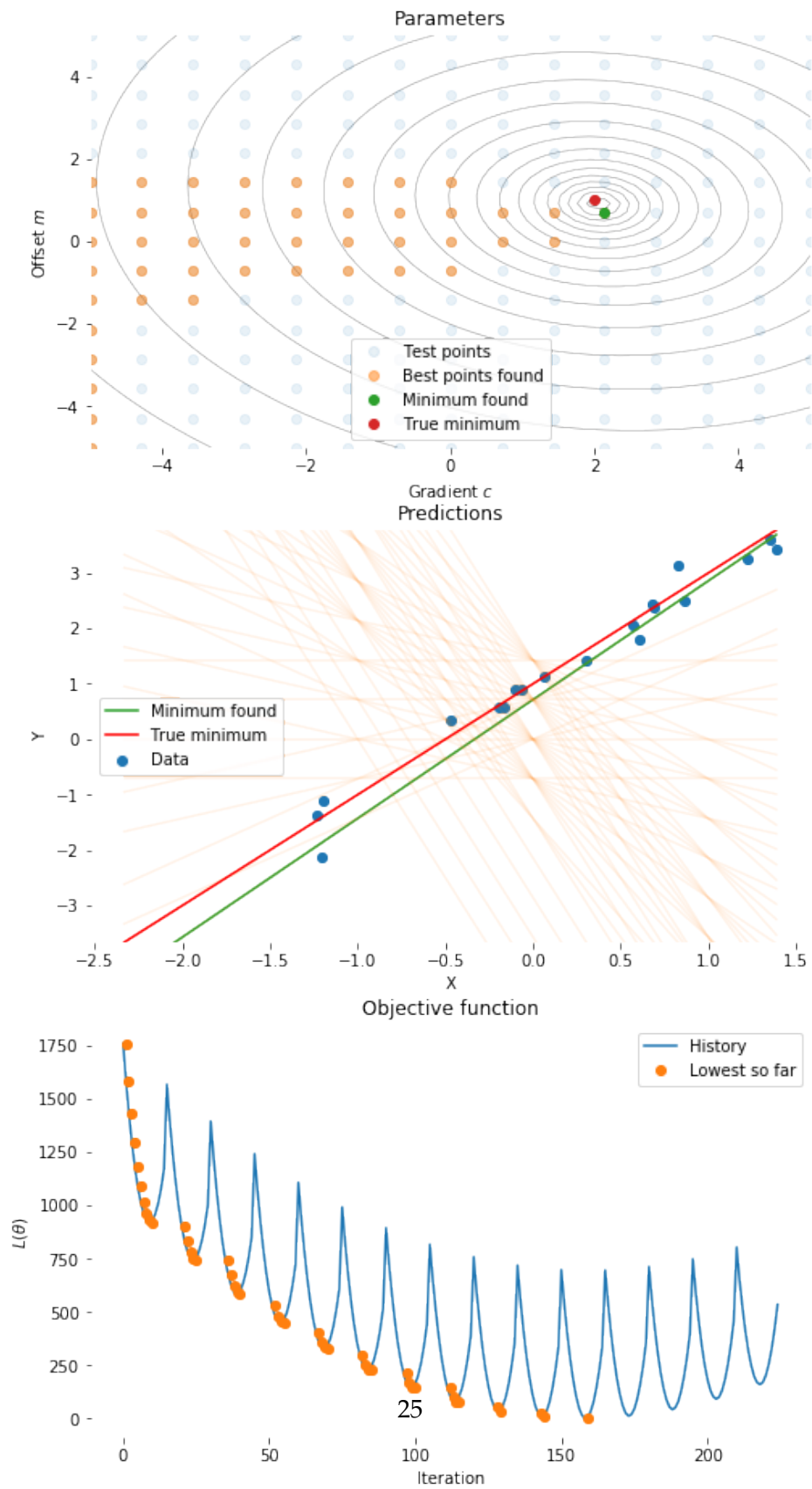
7.3.1 Revenge of the curse of dimensionality

Why bother optimising? Why not just search every possible parameter configuration?

Even in relatively small parameter spaces, and where the objective function is known to be smooth this doesn't scale well. Simply divide up each dimension into a number of points (maybe 8), and then try every combination on the grid of points that this forms, choosing the smallest result.

[Image: grid search breaks down in high dimensions]

While this is fine in 1D (just check 8 points) and 2D (just check 64 points), it breaks down completely if you have a 100 dimensional parameter space. This would need

```
[24]: 8**100
```

```
[24]: 2037035976334486086268445688409378161051468393665936250636140449354381299763336706183397376
```

evaluations of the objective function! The synthesizer above has around 100 dimensions, as an example.

Even just 3 points in each dimension is totally unreasonable:

```
[25]: 3**100
```

```
[25]: 515377520732011331036461129765621272702107522001
```

7.3.2 Density of grid search

If the objective function is not very smooth, then a much denser grid would be required to catch any minima.

Real optimisation problems might have hundreds, thousands or even billions of parameters (in big machine learning problems). Grid search and similar schemes are *exponential* in the number of dimensions of the parameter space.

7.3.3 Pros

- Works for any continuous parameter space.
- Requires no knowledge of the objective function.
- Trivial to implement.

7.3.4 Cons

- **Incredibly** inefficient
- Must specify search space bounds in advance.
- Highly biased to finding things near the "early corners" of the space.
- Depends heavily on number of divisions chosen.
- Hard to tune so that minima are not missed entirely.

7.4 Hyperparameters

Grid seach depends on the **range** searched and the spacing of the **divisions** of the grid. Most optimisation algorithms have similar properties that can be tweaked.

These properties, which affect the way in which the optimiser finds a solution, are called **hyperparameters**. They are not parameters of the objective function, but they do affect the results obtained.

A perfect optimiser would have no hyperparameters – a solution should not depend on how it was found. But in practice, all useful optimisers have some number of hyperparameters which will affect their performance. Fewer hyperparameters is usually better, as it is less cumbersome to tune the optimiser to work.

7.5 Simple stochastic: random search

The simplest such algorithm, which makes *no* assumptions other than we can draw random samples from the parameter space, is **random search**.

The process is simple: * Guess a random parameter θ * Check the objective function $L(\theta)$ * If $L(\theta) < L(\theta^*)$ (the previous best parameter θ^*), set $\theta^* = \theta$

There are many possibilities for a termination condition, such as stopping after a certain number of iterations after the last change in the best loss. The simple code below uses a simple fixed iteration count and therefore makes no guarantee that it finds a good solution at all.

7.5.1 Pros

- Random search cannot get trapped in local minima, because it uses no local structure to guide the search.
- Requires no knowledge of the structure of the objective function - not even a topology.
- Very simple to implement.
- Better than grid search, almost always.

7.5.2 Cons

- *Extremely inefficient* and is usually only appropriate if there is no other mathematical structure to exploit.
- Must be possible to randomly sample from the parameter space (usually not a problem, though).
- Results do not necessarily get better over time. Best result might be found in the first step or a million steps later. There is no way to predict how the optimisation will proceed.

```
[26]: def random_search(L, sample_fn, iters):
          """L: loss function
          sample_fn: calling this should draw one random sample from the parameter␣
      ↪space
          iters: number of iterations to run the optimisation for
          """
          o = History()
          for i in range(iters):
```

```
        theta = sample_fn()
        o.track(theta, L(theta))
    return o.finalise()
```

```
[27]: def sample():
          return np.random.normal(0,5,size=(2,))

      random_results = random_search(loss, sample, 100)
      linear_regression_plot(random_results, gradient, offset, line_x, line_y, "Random␣
       ↪search")
```

Random search over possible line configurations
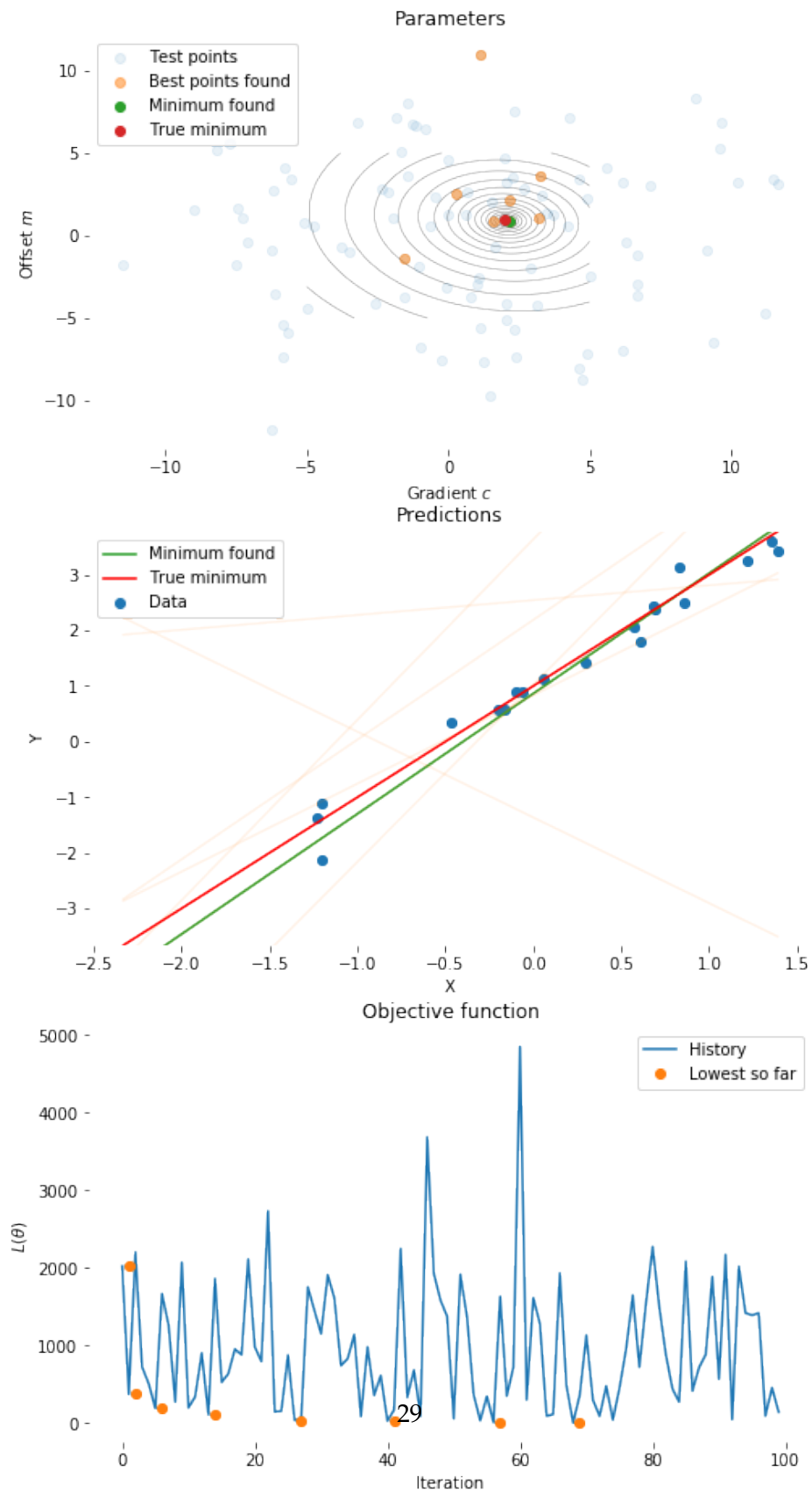
7.5.3 bogosort

The (joke) sorting algorithm **bogosort** uses random search to sort sequences. The algorithm is simple:

- randomise the order of the sequence
- check if it is sorted
  - if it is, stop; otherwise, repeat

This is amazingly inefficient, taking $O(n!)$ time to find a solution, which is even worse than exponential time. In this application, the parameter space (all possible orderings) is so huge that random search is truly hopeless. It is particularly poor because of the binary nature of the loss function – either it is perfect, or it is disregarded, so we will never even get approximately correct results.

However, it is a correct implementation of a sorting algorithm.

```
[28]: def bogosort(x):
          # check if we have any decreasing elements left
          while np.any(np.diff(x)<0):
              np.random.shuffle(x)
          return x
```

```
[29]: bogosort([5,6,1,3,2,9])
```

```
[29]: [1, 2, 3, 5, 6, 9]
```

```
[30]: %%timeit
      # bogosort will sort a 5 element array in ~500 microseconds
      bogosort([3,2,1,4,2])

      # it would take 3 years to sort a 13 element array
      # and longer than the age of the universe to sort a 21 element array
```

```
637 µs ± 53.9 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

8 Resources

8.0.1 Beyond this course

- **When least is best: How Mathematicians Discovered Many Clever Ways to Make Things as Small (or as Large) as Possible** *by Paul J. Nahin* An interesting and mathematically thorough description of the history of optimisation from a mathematical standpoint.