L2 Computational Physics

Optimisation Techniques

Function minimisation

Optimisation Techniques

- Background
- Aside – visualising 2D data as images
- Brute force methods
- Deterministic Methods
  - Gradient Descent
  - Nelder-Mead Simplex
- Stochastic Methods
  - Genetic Algorithms

Background

Function minimisation

- Given f(x,y,....) find the coordinates and value at the minima of the function
- Analytical techniques – only useful for some functions
- Numerical techniques
  - This is what we will look at today

Applications of minimisation

- Science
  - Fitting a model function to experimental data
  - Orbital mechanics
  - Optical design
    - Maximise image quality
    - Minimise cost
  - Adaptive Optics
  - Protein folding
  - Optimising control system parameters
- An example in your pocket:
  - Auto-focus cameras (smart phone)

Aside

Visualising 2D functions

- It’s going to be really useful to be able to visualise these functions

- Let’s look at how we do that with Python, numpy and matplotlib

2D plotting

- We want to visualise f(x, y)
- Evaluate f(x, y) over a range of evenly spaced x and y values
- Store the results in a 2D numpy array
- Display this with matplotlib

2D plotting

```
from __future__ import division
import numpy
import matplotlib.pyplot as pyplot
import matplotlib.colors as colors
import matplotlib.cm

def f(x, y):
    a=numpy.cos(0.2*x**2-0.3*y**2+3)
    b=numpy.sin(2*y-1+numpy.e**x)
    return a*b
```

2D plotting

```
# Define bounds
x0, x1 = -2.5, 2
y0, y1 =  -2, 2

#explore 1000 points in x and y
N_POINTS=1000
dx=(x1-x0)/N_POINTS
dy=(y1-y0)/N_POINTS

#generate x and y values
xs=numpy.arange(x0,x1,dx)
ys=numpy.arange(y0,y1,dy)

#array to hold function values
dat=numpy.zeros((len(xs), len(ys)))
```

2D plotting

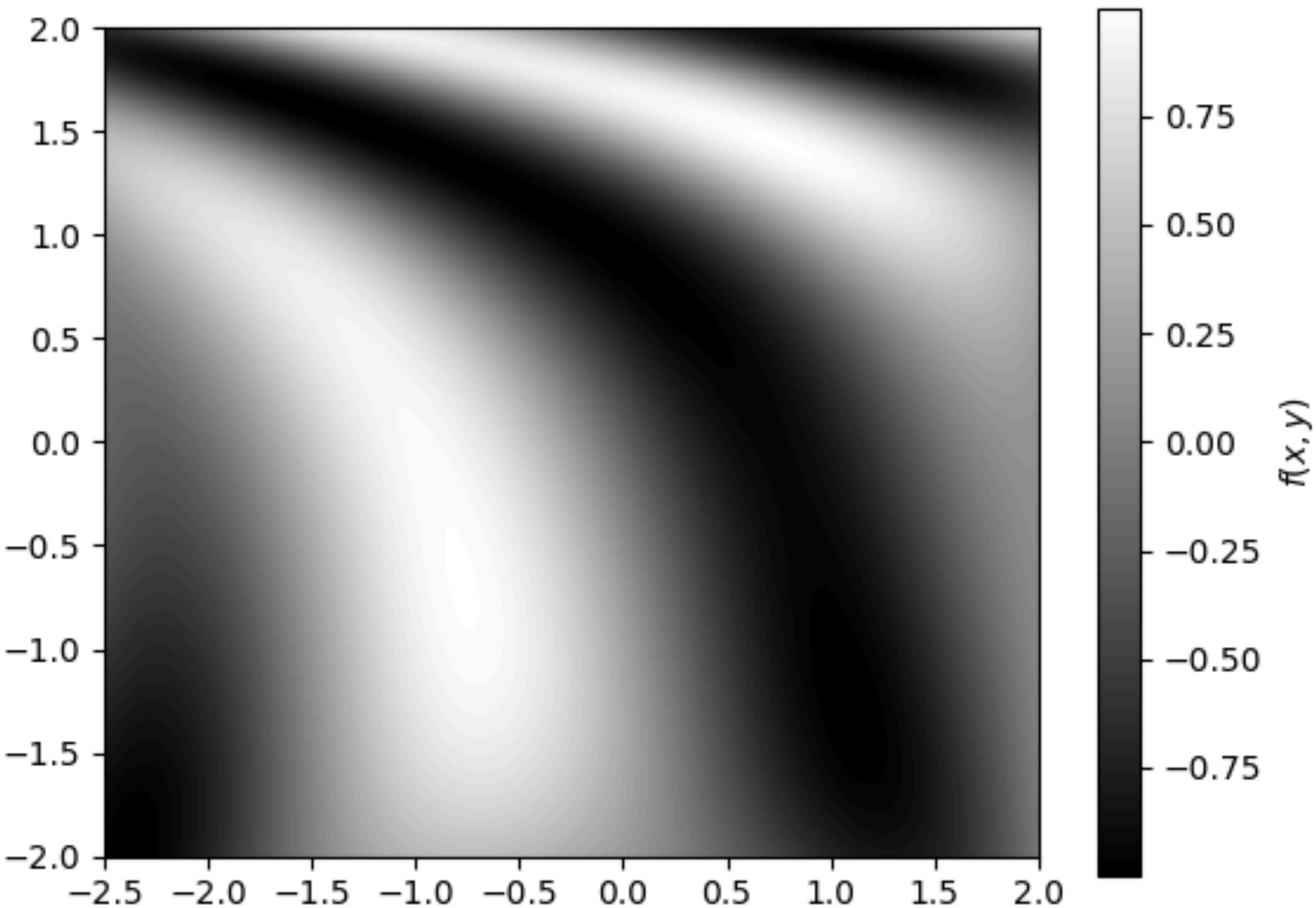
```
for ix, x in enumerate(xs):
    for iy, y in enumerate(ys):
        dat[ix,iy]=f(x,y)

pyplot.figure()

# Show a greyscale colourmap of the data
im = pyplot.imshow(dat,
        extent=(x0, x1,y0, y1),
        origin='lower',
        cmap=matplotlib.cm.gray)
pyplot.xlabel('x')
pyplot.ylabel('y')

pyplot.colorbar(im, orientation='vertical',
label='$f(x,y)$')

pyplot.show()
```


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
f(x, y)

Methods

- Brute Force and Ignorance
  - **Exhaustive Search**
- Deterministic searches
  - **Nelder-Mead Simplex**
  - **Gradient Descent**
  - Hill Climbing
- Stochastic Searches
  - **Genetic Algorithm**
  - Stochastic Gradient Descent
  - Simulated Annealing

Brute Force and Ignorance

Fast to code
Slow to run

Exhaustive Search

- For every x
  - For every y
    - Is this the smallest f(x, y)?

- Benefits
  - Trivial to code

- Drawbacks
  - Slow
  - Not very accurate – it must operate on some finite, quantised grid

Deterministic Methods

Deterministic Methods

- These methods all start from some initial position
- If you run the same method from the same position multiple times, you get the same result

Gradient Descent

Gradient Descent: Algorithm

- **Walk downhill**

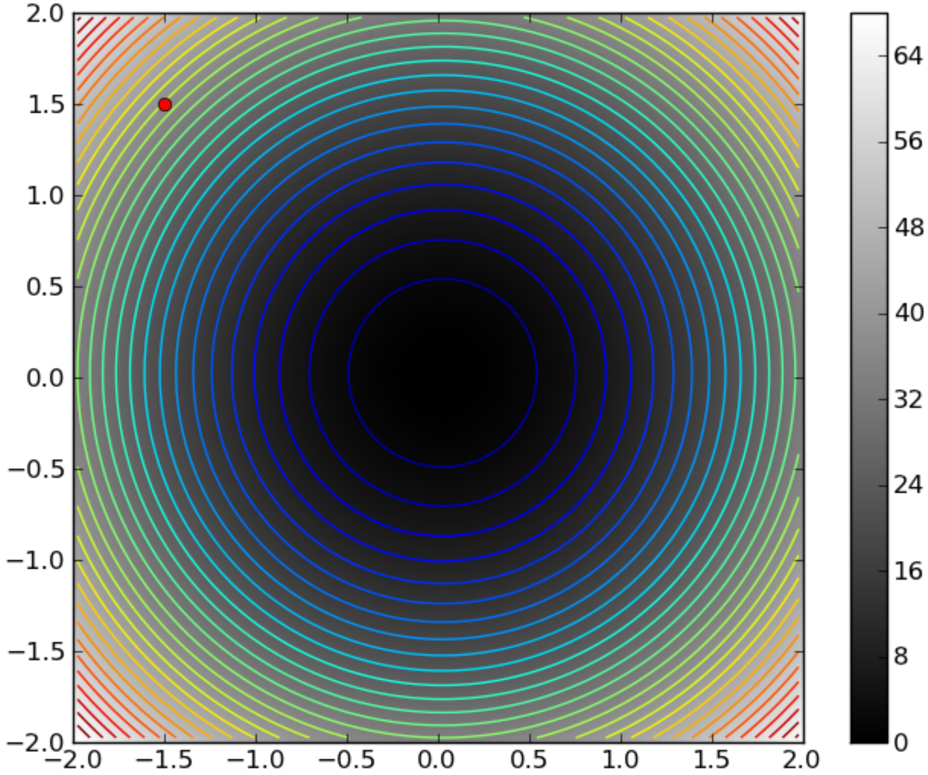
Gradient Descent: Algorithm

Maths notation	Quantity	Python
$\vec{r}$	Position vector	`r = numpy.array(x, y)`
$f(\vec{r})$	Function at $\vec{r}$	`def f((x, y)):` `    return ???`

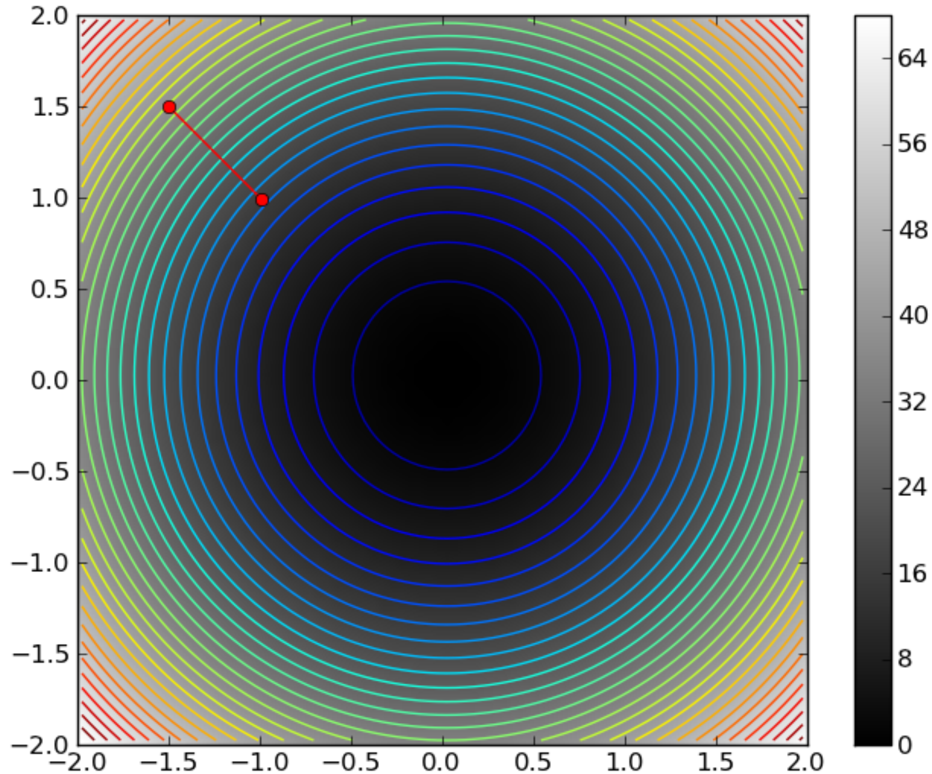
$\nabla f(\vec{r})$	Vector differential of $f(\vec{r})$	`def f((x, y)):`
		`    df_dx = ???`
		`    df_dy = ???`
		`    grad = numpy.array((df_dx, dy_dy)`
		`    return grad`

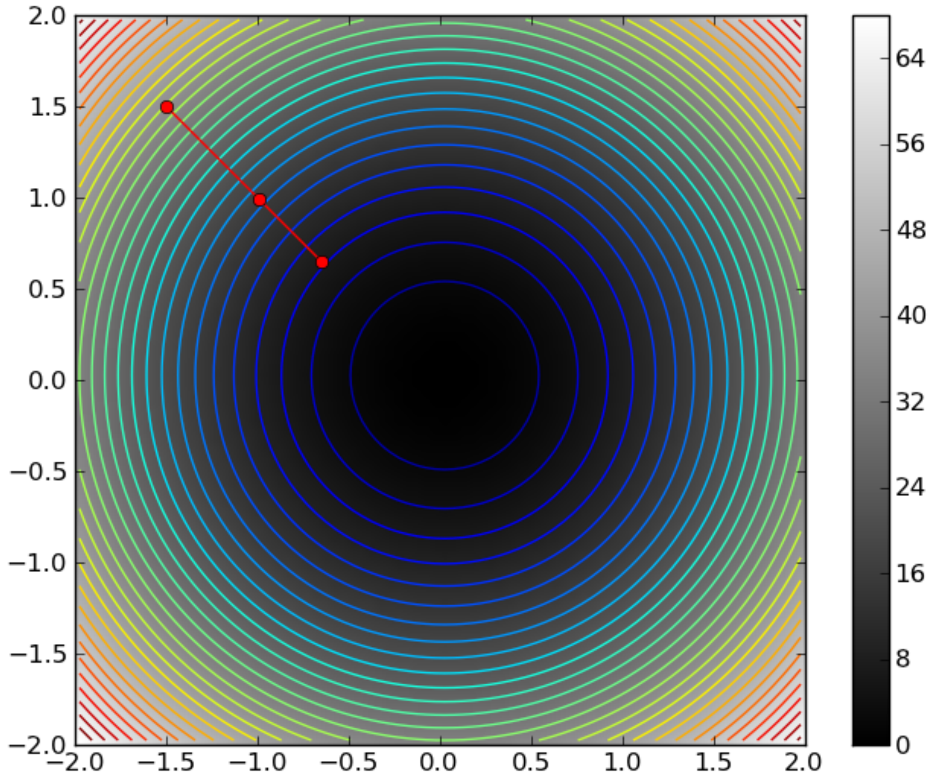
$\vec{r}_0$	Initial position	`r0 = numpy.array((x0, y0))`
γ	Step size	`gamma = 0.something`

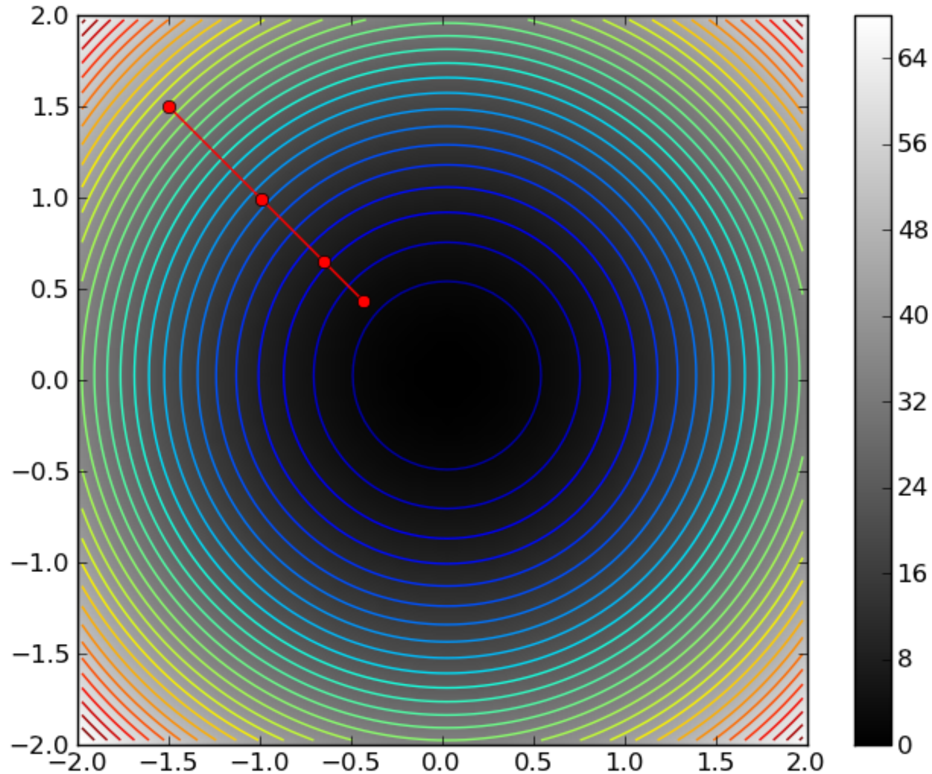
$$\vec{r}_{n+1} = \vec{r}_n - \gamma \nabla f(\vec{r}_n)$$



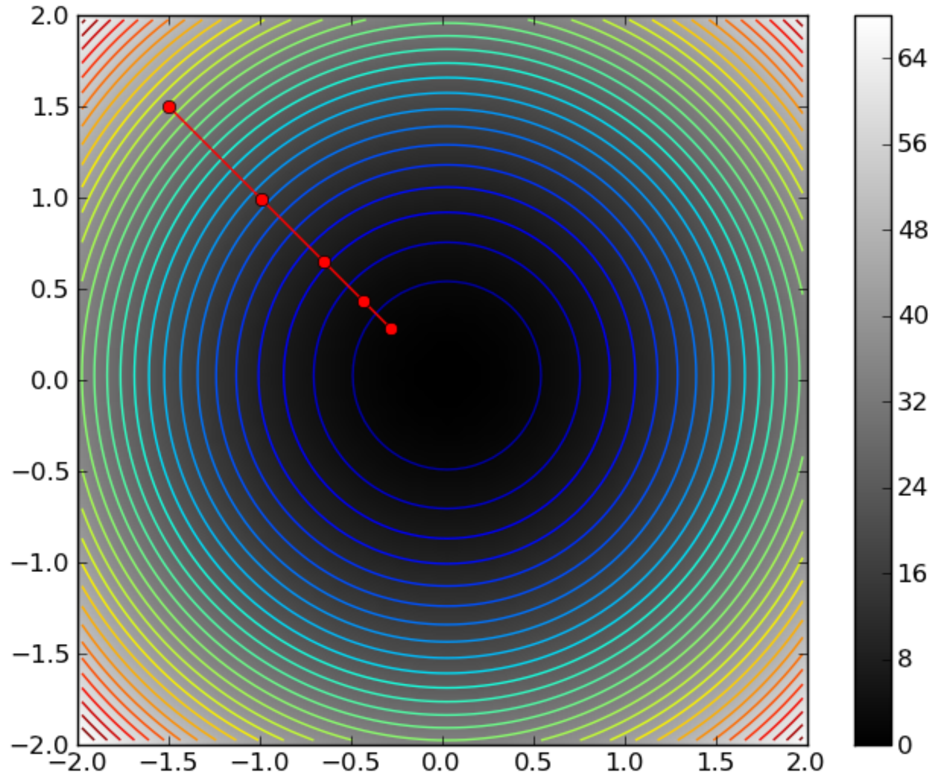
Coloured lines are contours - see http://matplotlib.org/examples/pylab_examples/contour_demo.html


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0
64
56
48
40
32
24
16
8
0

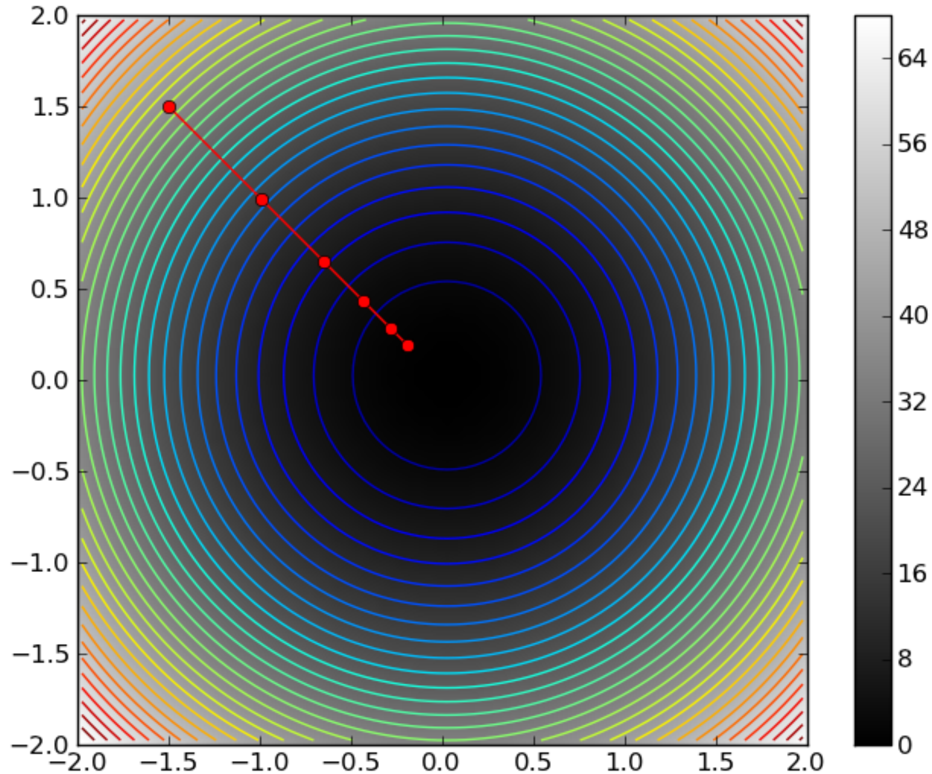

2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0
64
56
48
40
32
24
16
8
0



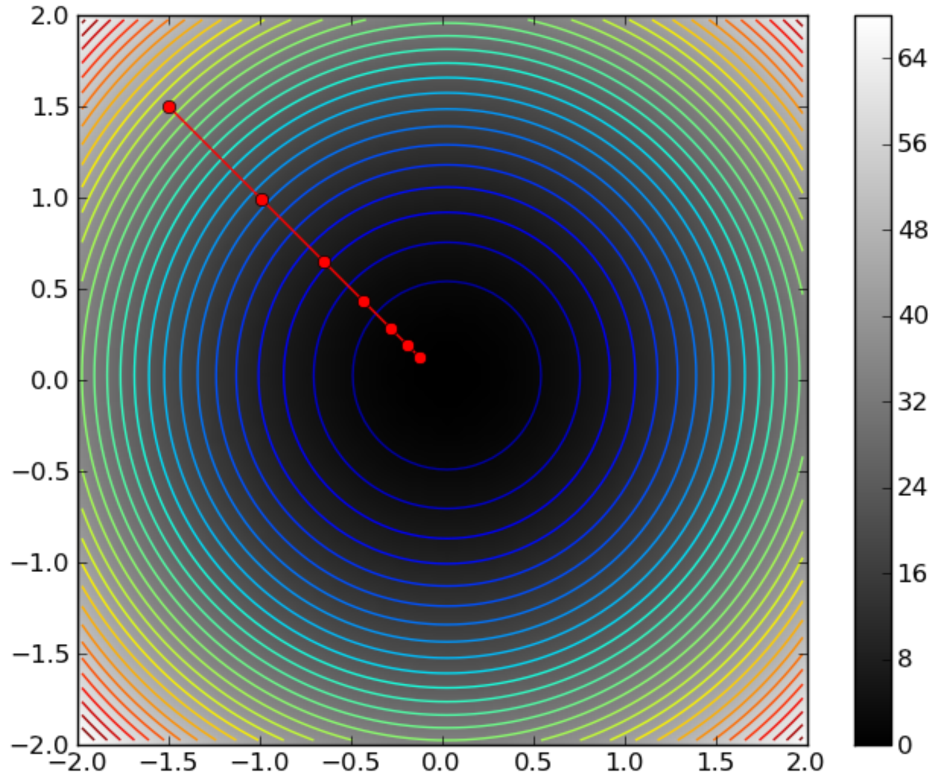
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
64
56
48
40
32
24
16
8
0



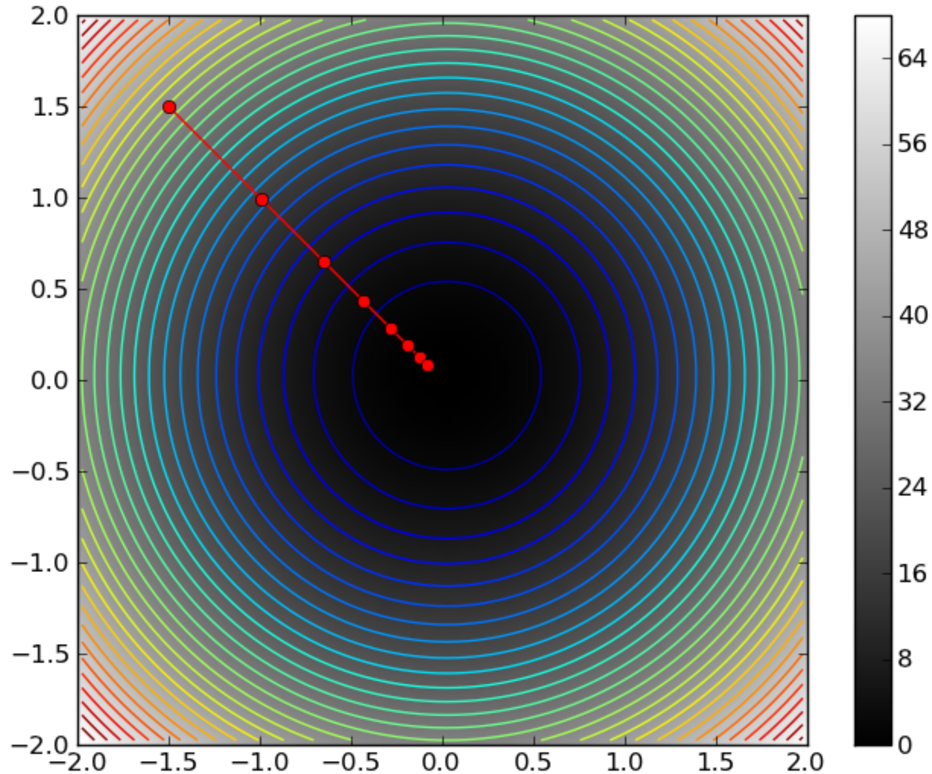
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0
64
56
48
40
32
24
16
8
0



2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
64
56
48
40
32
24
16
8
0



2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0
64
56
48
40
32
24
16
8
0


2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0 −1.5 −1.0 −0.5 0.0 0.5 1.0 1.5 2.0
64
56
48
40
32
24
16
8
0

Step Size

- Gradient Descent is highly sensitive to the step size, gamma
- Too small a step and convergence is very slow
- Too large a step and it may overshoot and the method becomes unstable

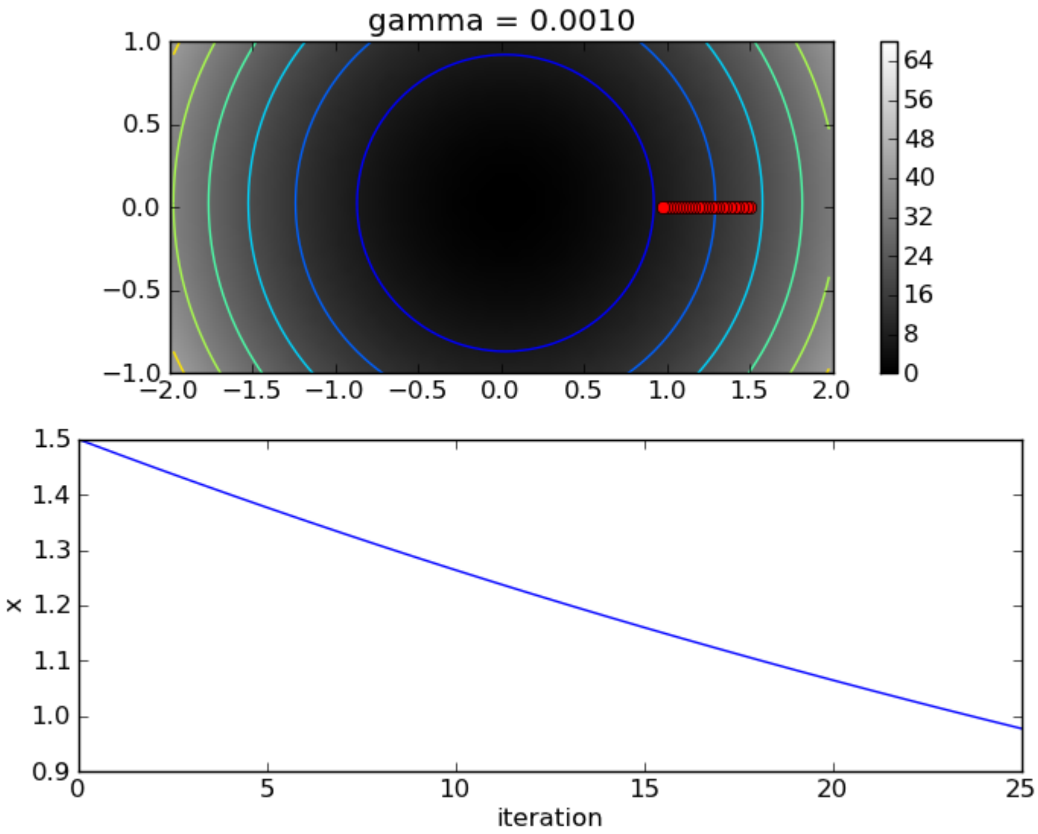
- Audience Question: What causes this to happen?

Step Size

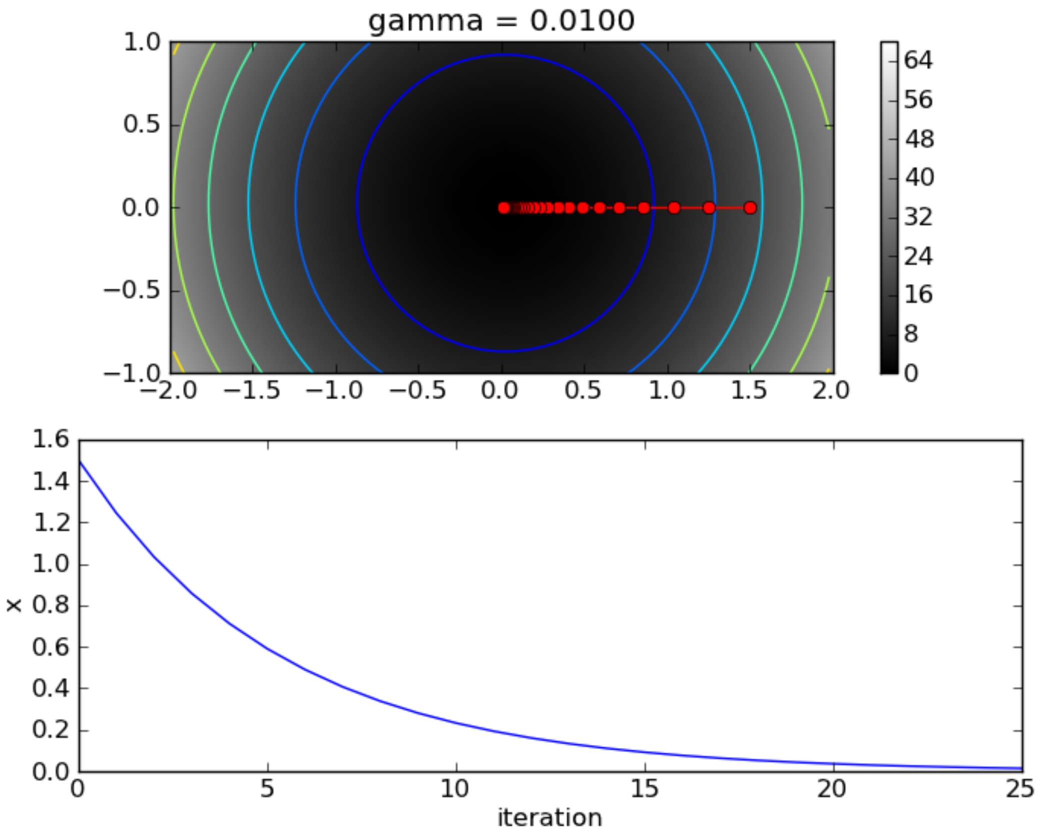
- Gradient Descent is highly sensitive to the step size, gamma
- Too small a step and convergence is very slow
- Too large a step and it may overshoot and the method becomes unstable

- Curvature and higher order terms mean the gradient is only locally constant – *adaptive step size* can choose a gamma based on curvature measurements etc.

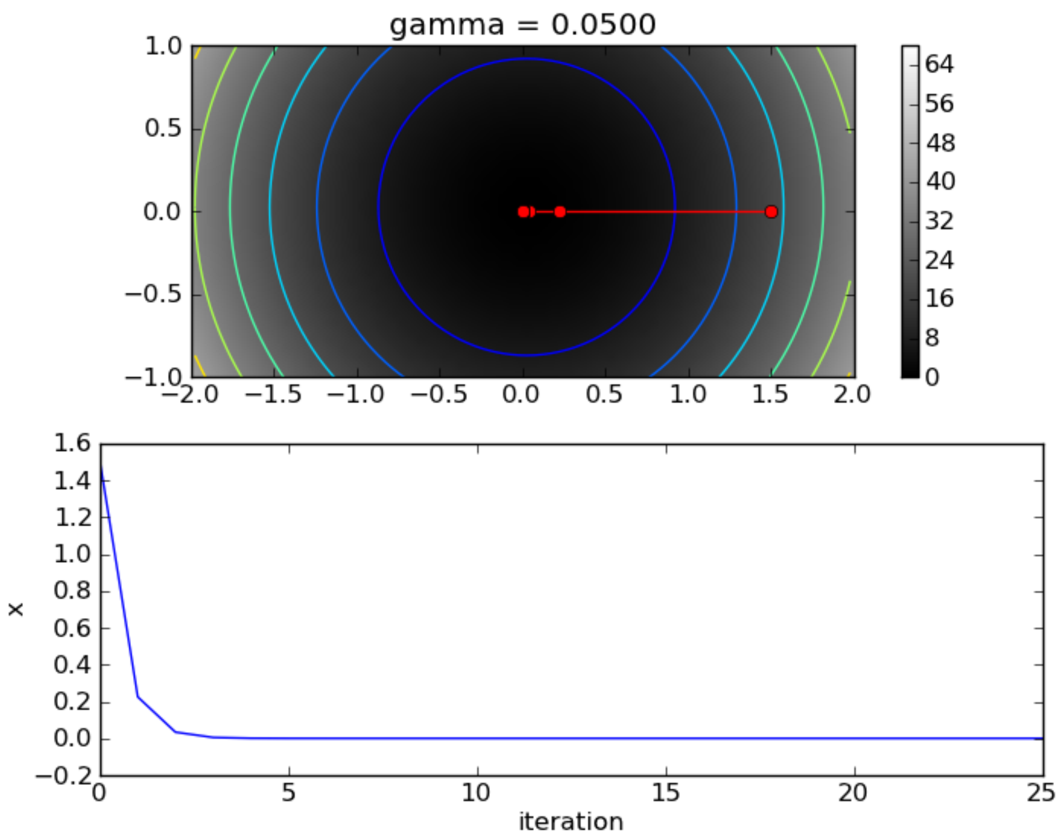
γ to small – slow convergence



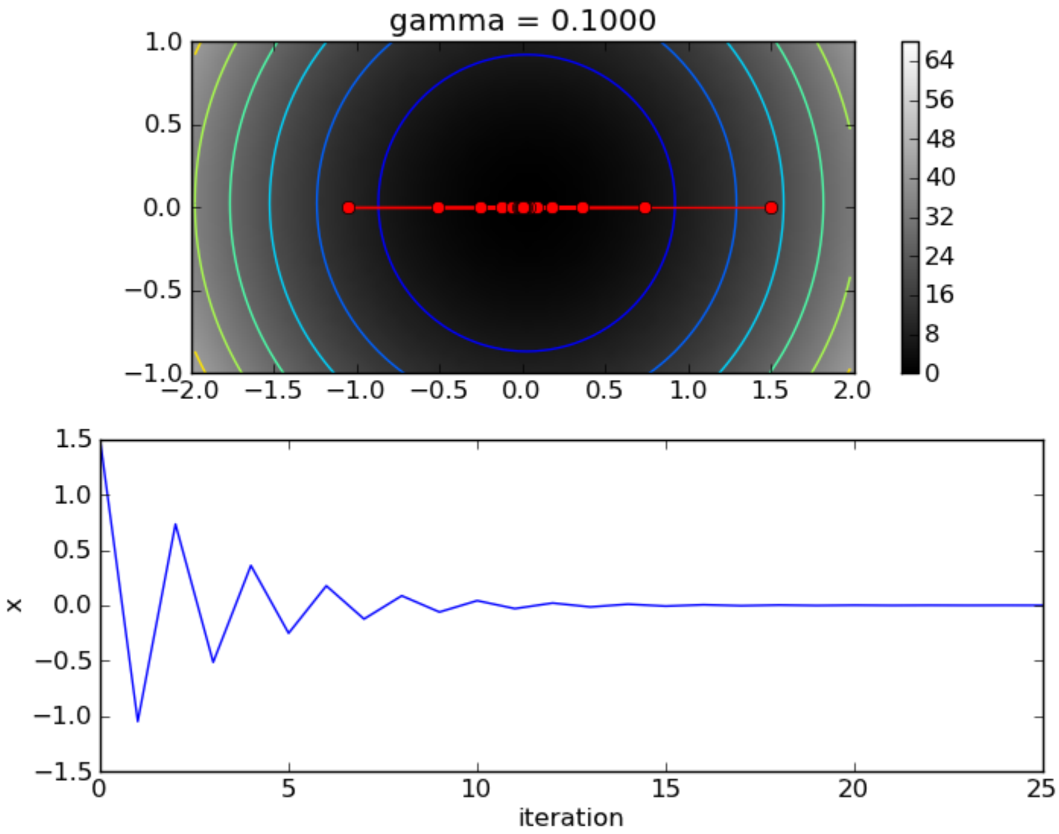
γ larger – faster convergence



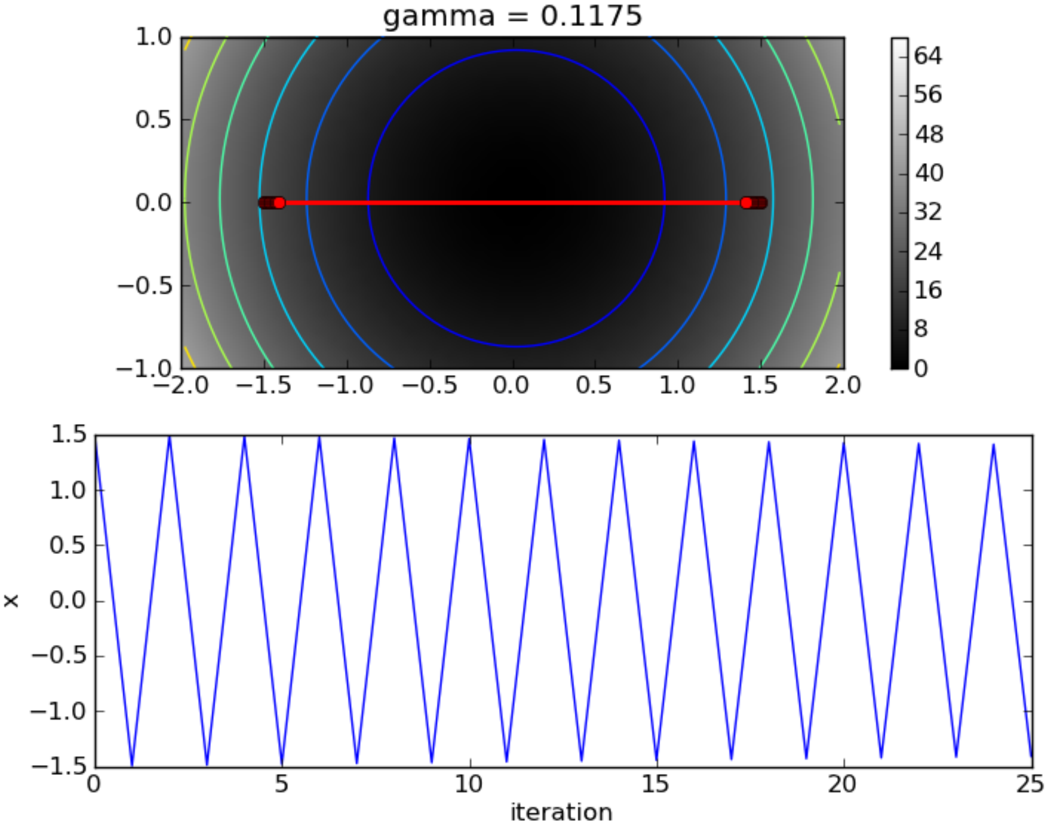
γ about right



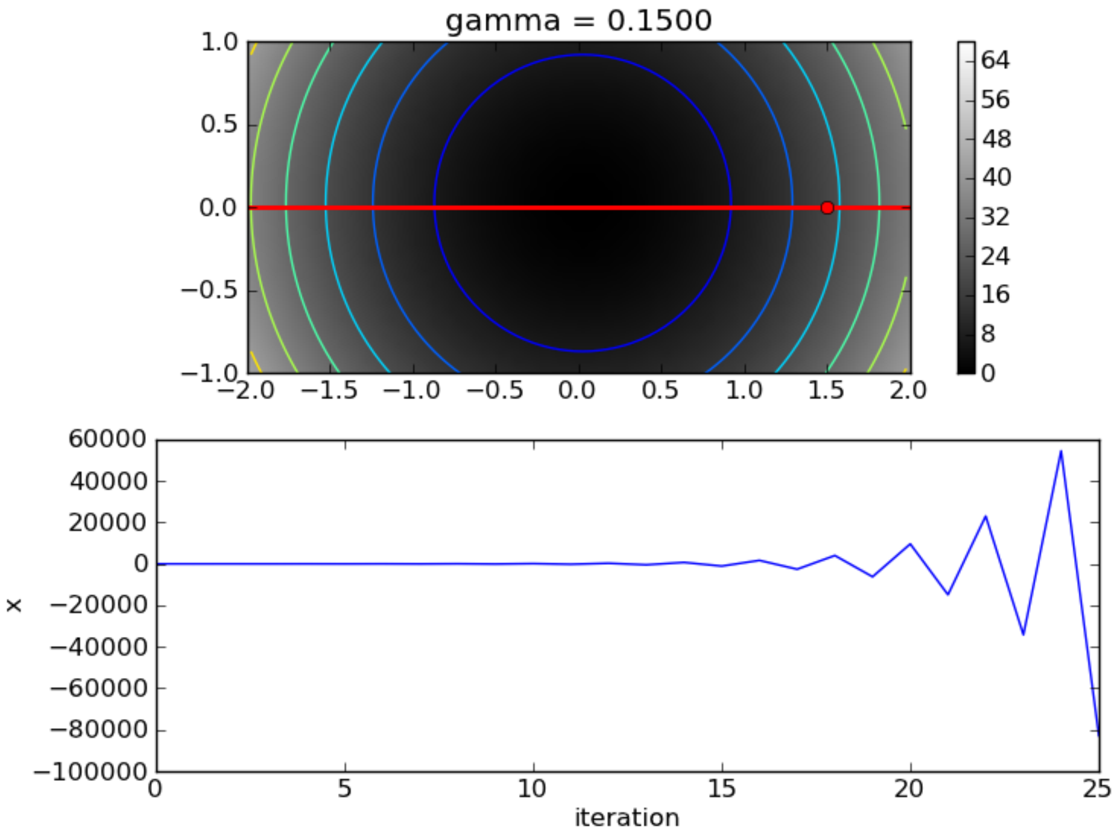
γ to big – oscillatory convergence



γ – perfectly wrong



γ far to big - divergence



GD Example 2

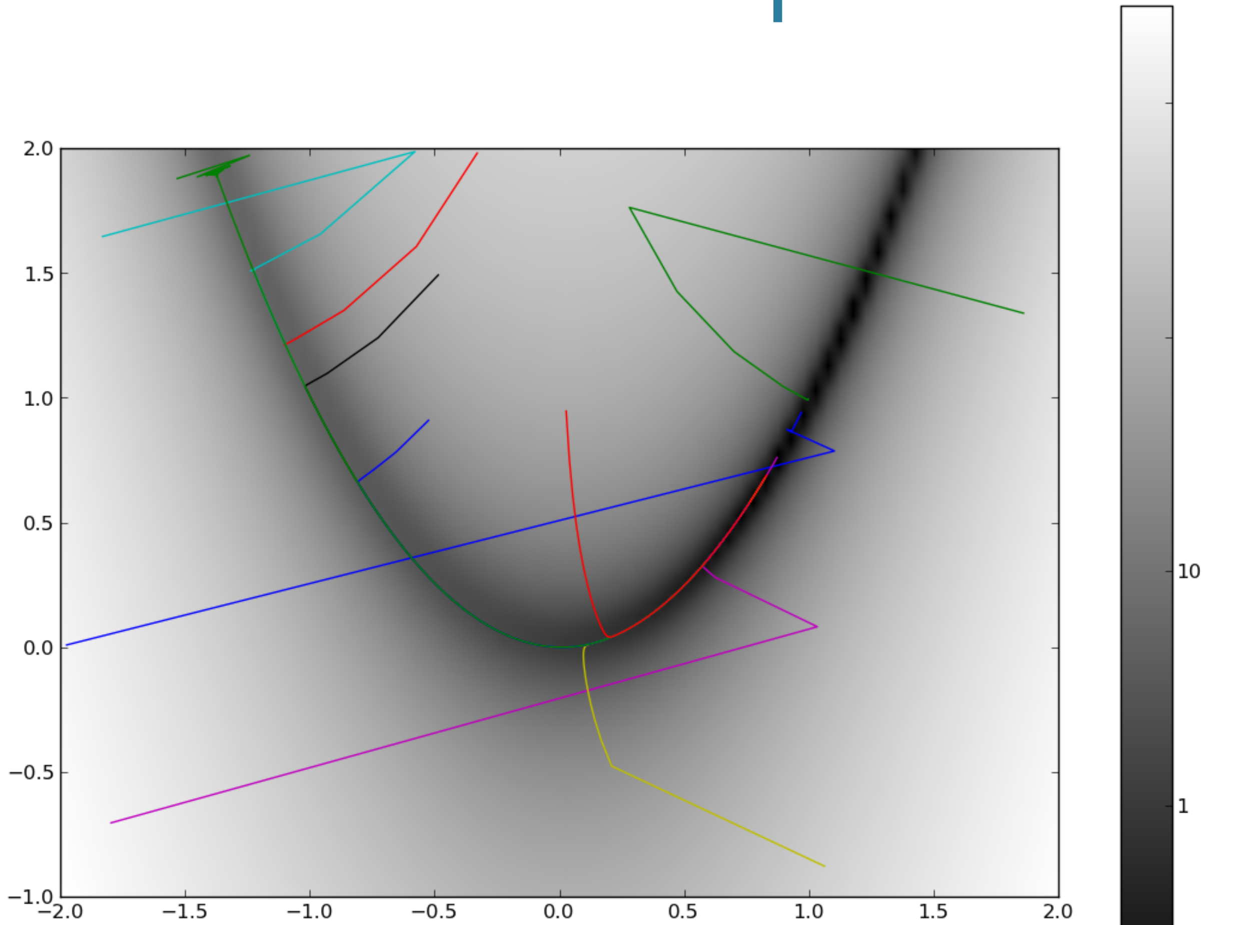
- Rosenbrock's Banana Function

$$f(x,y) = (1-x)^2 + 100(y-x^2)^2$$

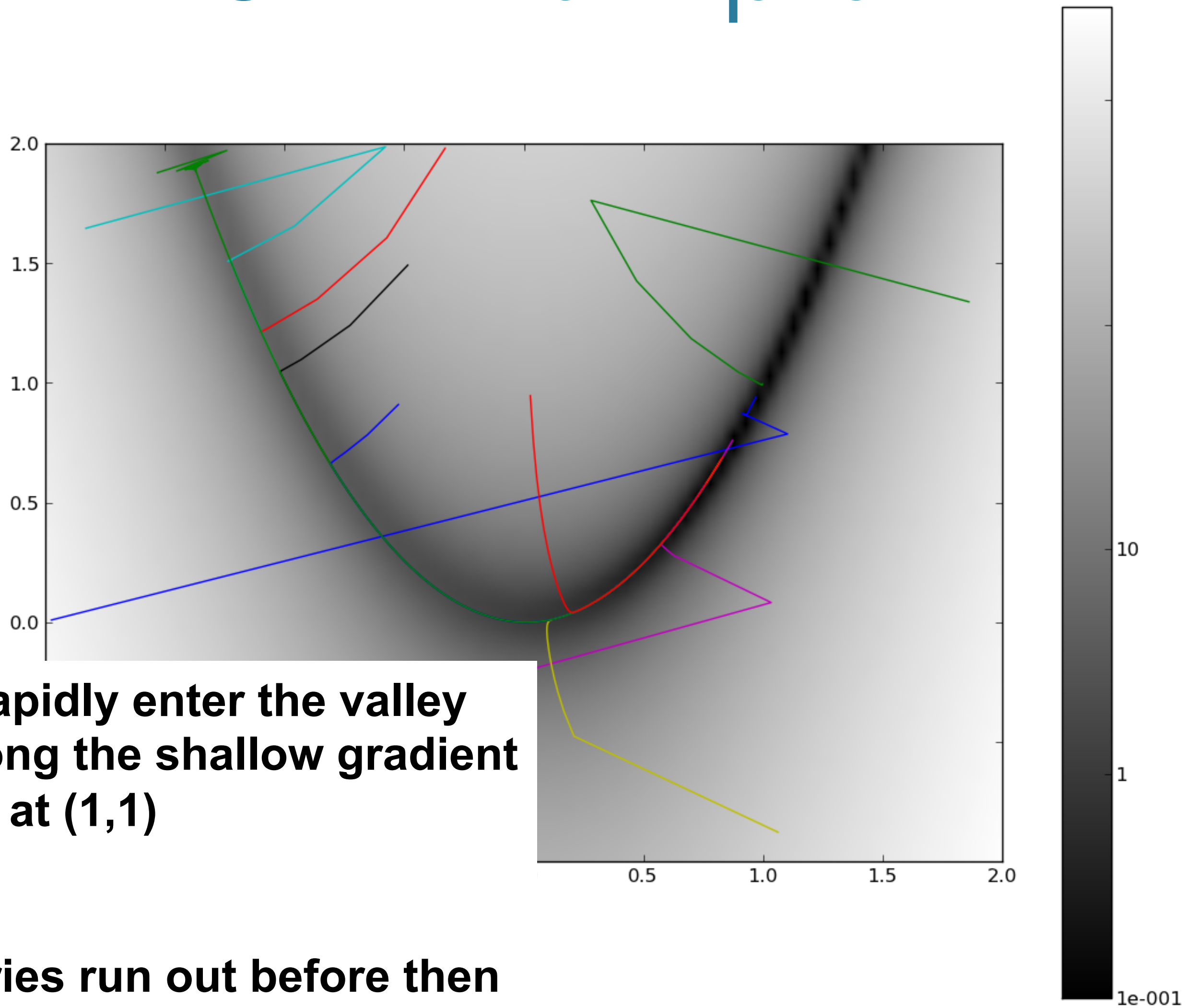
- A tough test case for minima finding

- Steep cliffs
- Very shallow valley

GD Example 2

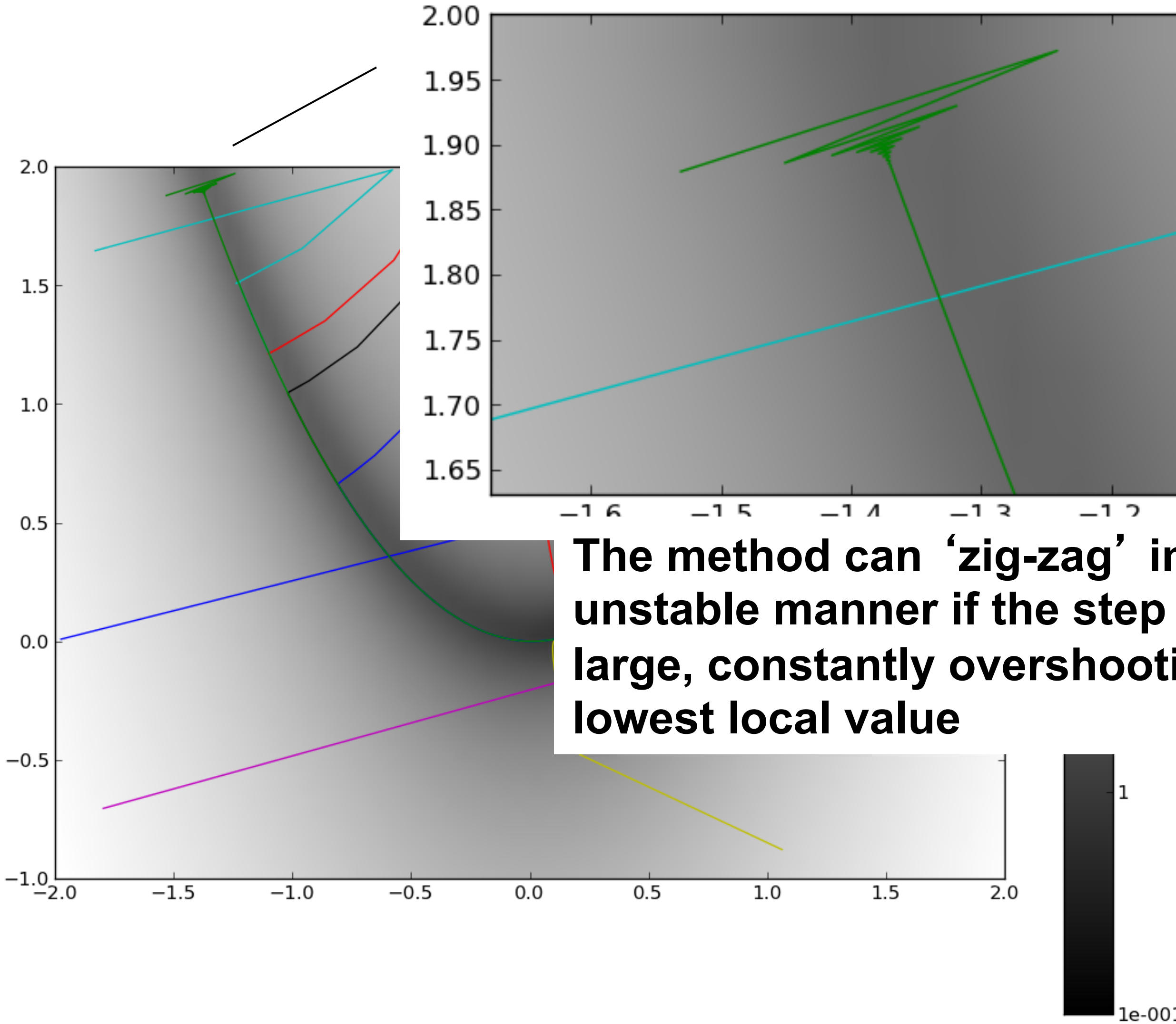


GD Example 2



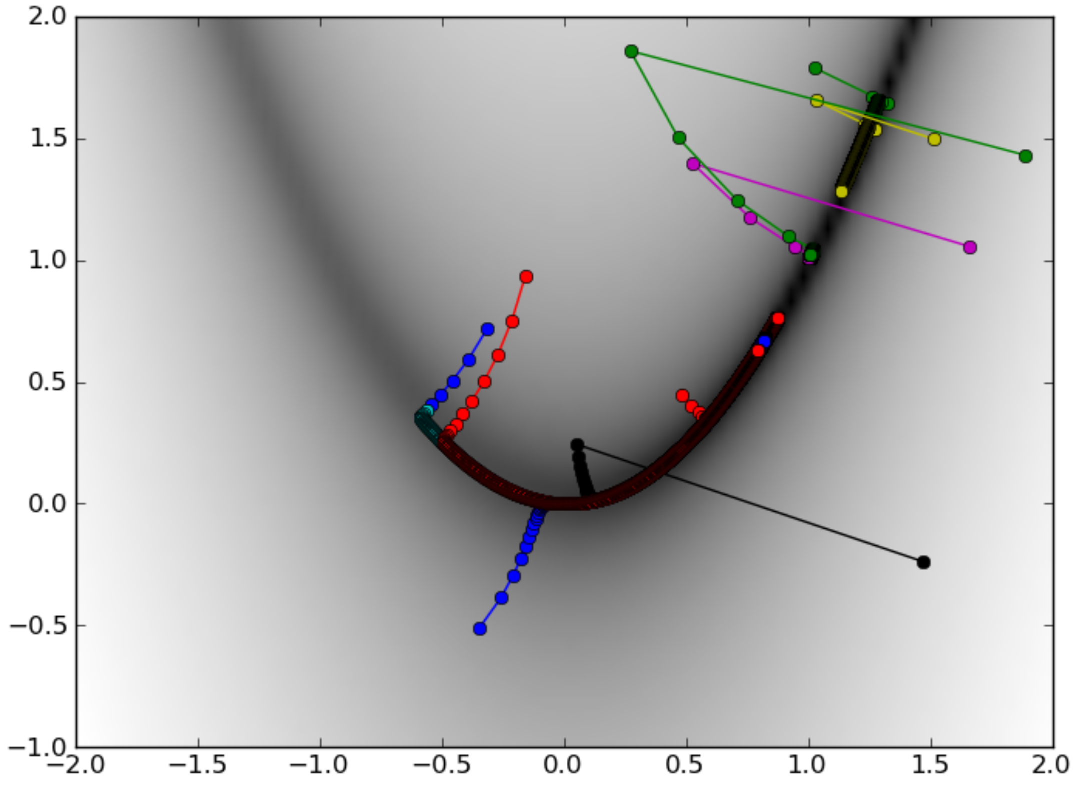
Trajectories rapidly enter the valley then crawl along the shallow gradient to the minima at (1,1)

Most trajectories run out before then as we have not run for enough itterations

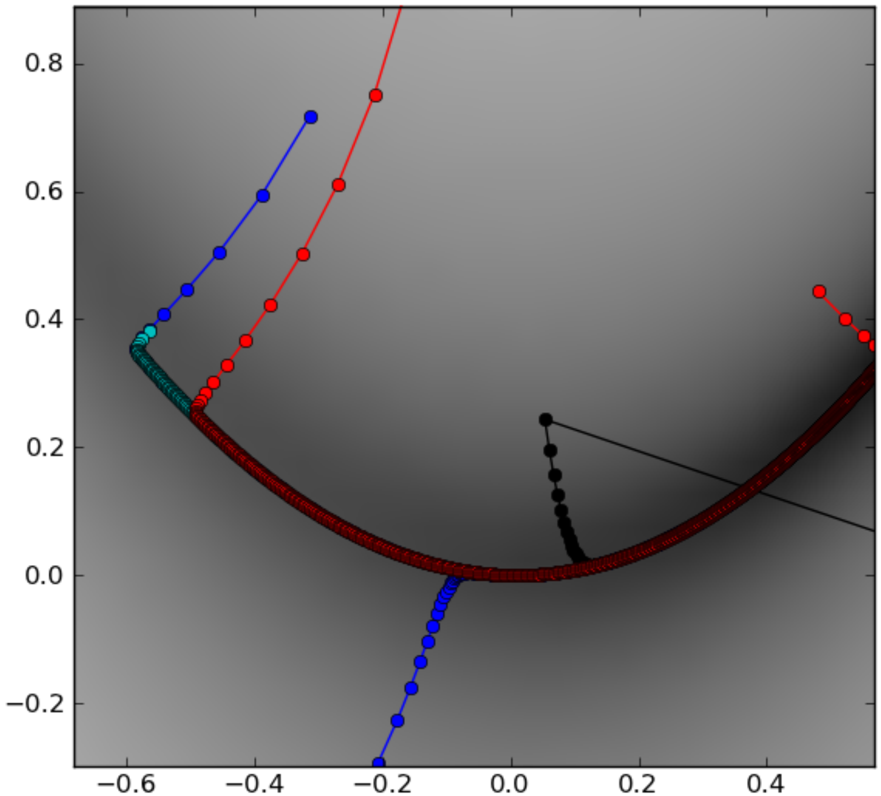


The method can 'zig-zag' in an unstable manner if the step size is to large, constantly overshooting the lowest local value

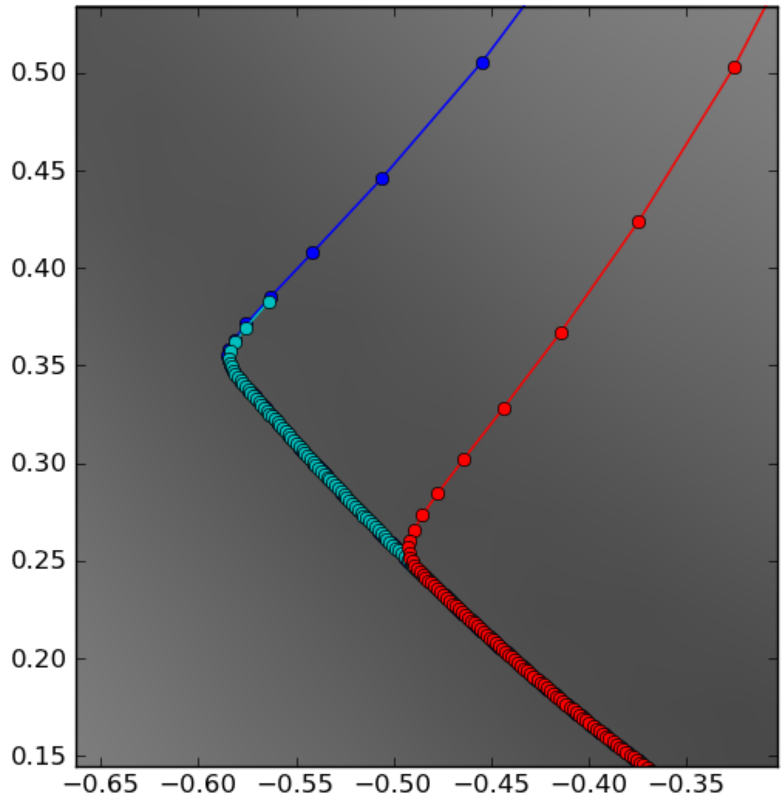
GD in the valley - slow



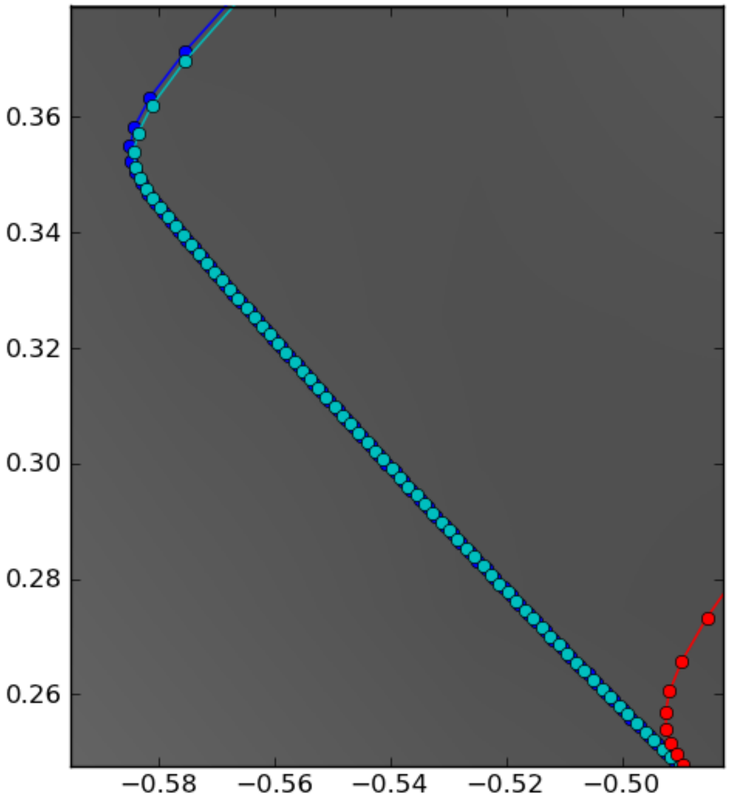
GD in the valley - slow



GD in the valley - slow



GD in the valley - slow



GD – when to stop?

- It's common to have a maximum number of iterations
- Another common pattern is to terminate early upon reaching some *convergence criteria*

cp6.concvergence.py – /Users/cds/cp6.concvergence.py

```
r = initital_position
for i in range(max_iter)
    fLast = f(r)
    r = r + ... # gradient descent step
    fNew =  f(r)
    if abs(fNew - fLast) < CONVERGENCE_CRITERA:
        break # we are done, exit for loop early
print 'Found minimum in %i iterations ' % i
```

Ln: 9 Col: 0

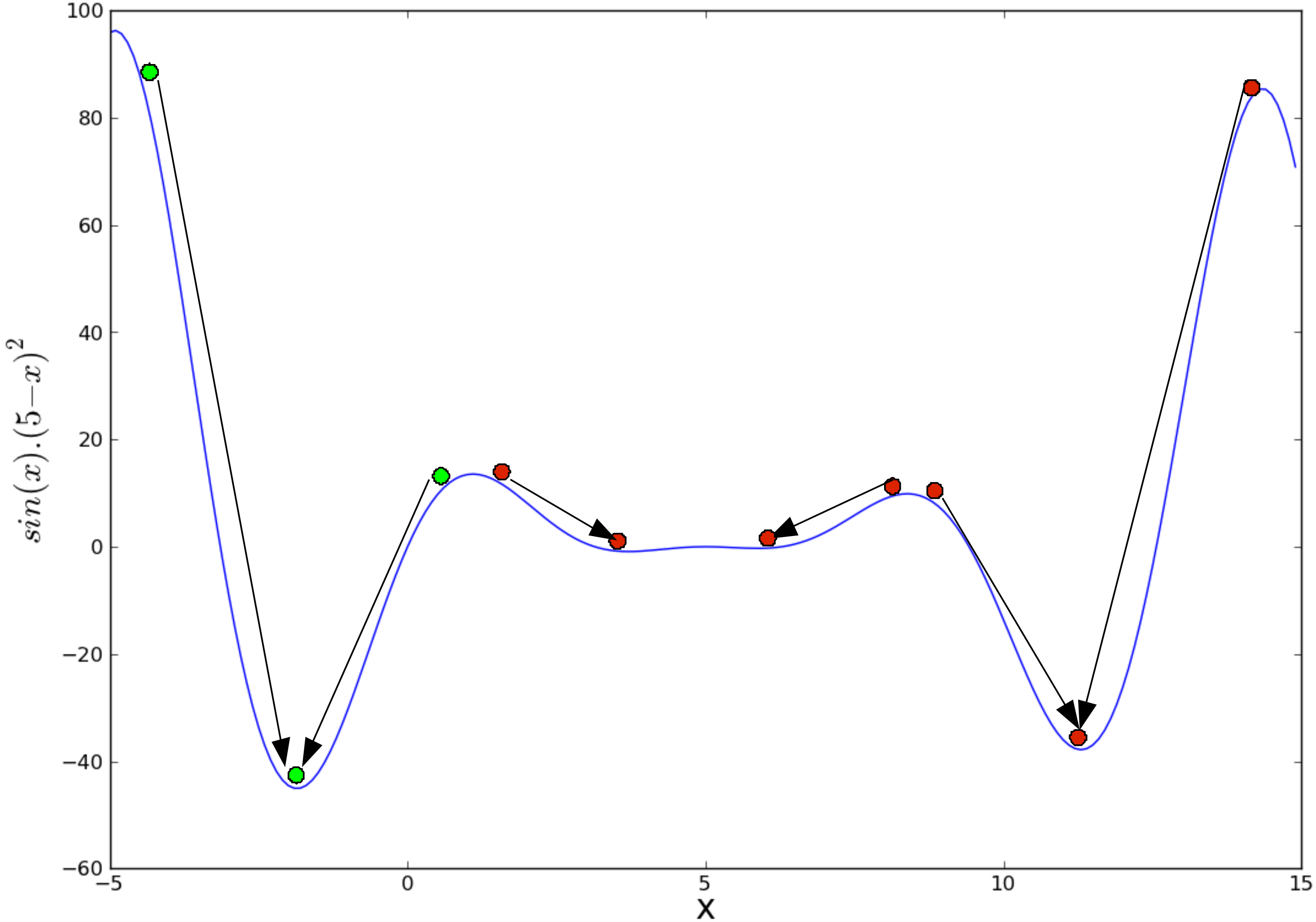
Hill Climbing

Hill Climbing

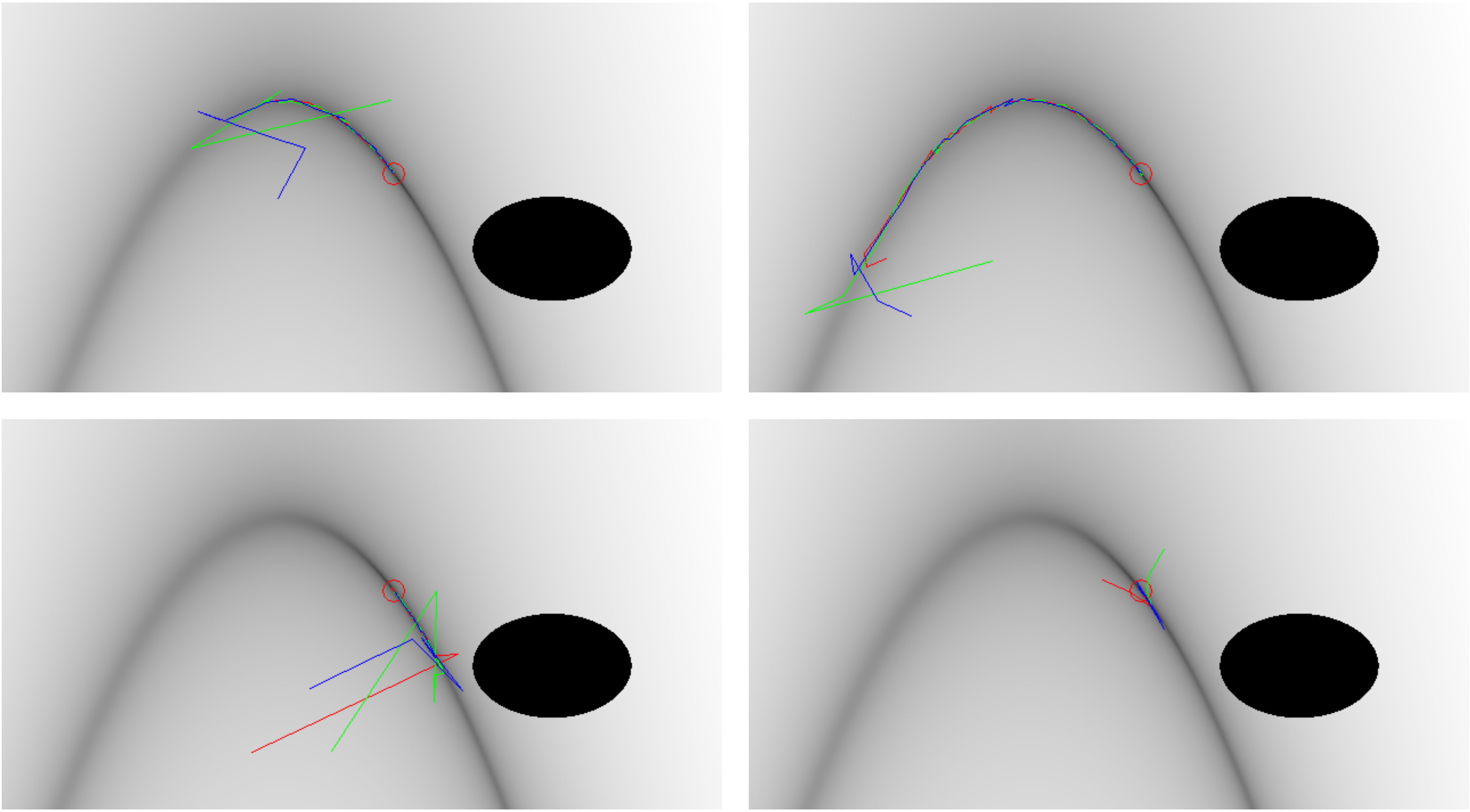
- Hill climbing is similar to gradient descent but simpler
- Hill climbing tries moving a small distance in one dimension only at a time
- When this no longer works, another dimension is explored
- Very simple to code

Local Minima

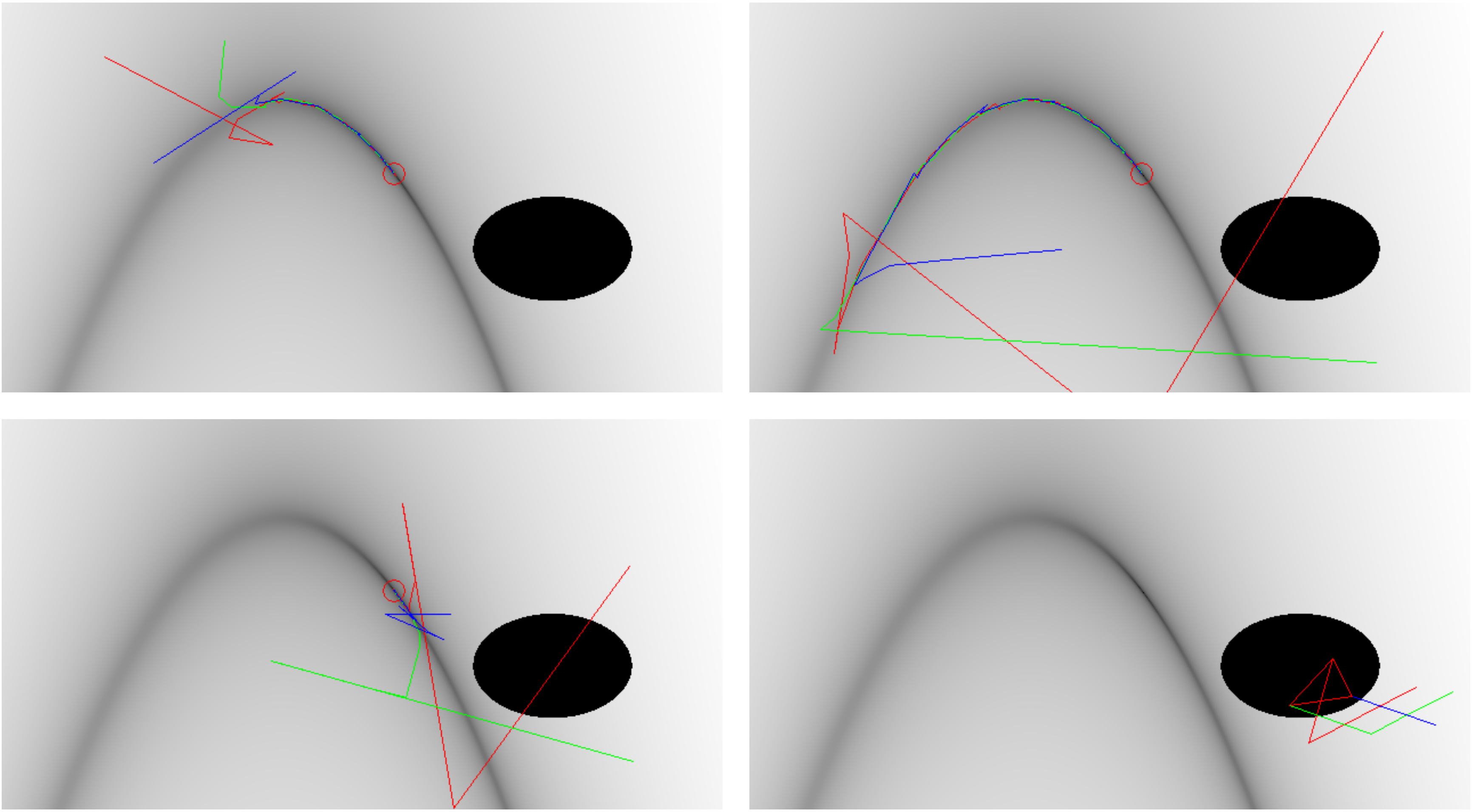
GD & Local Minima



Pathological example



No gradient into the minima



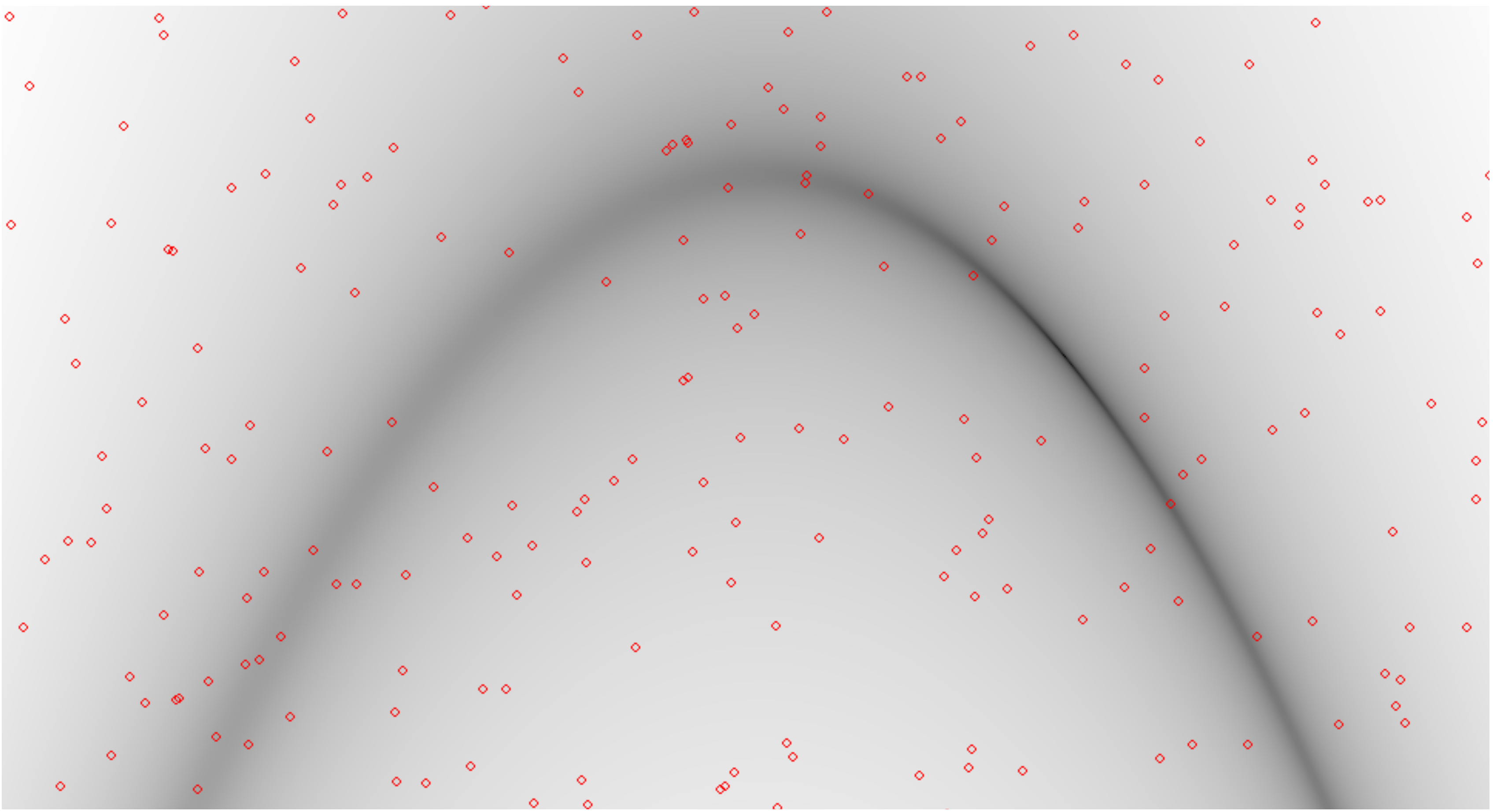
Stochastic Methods

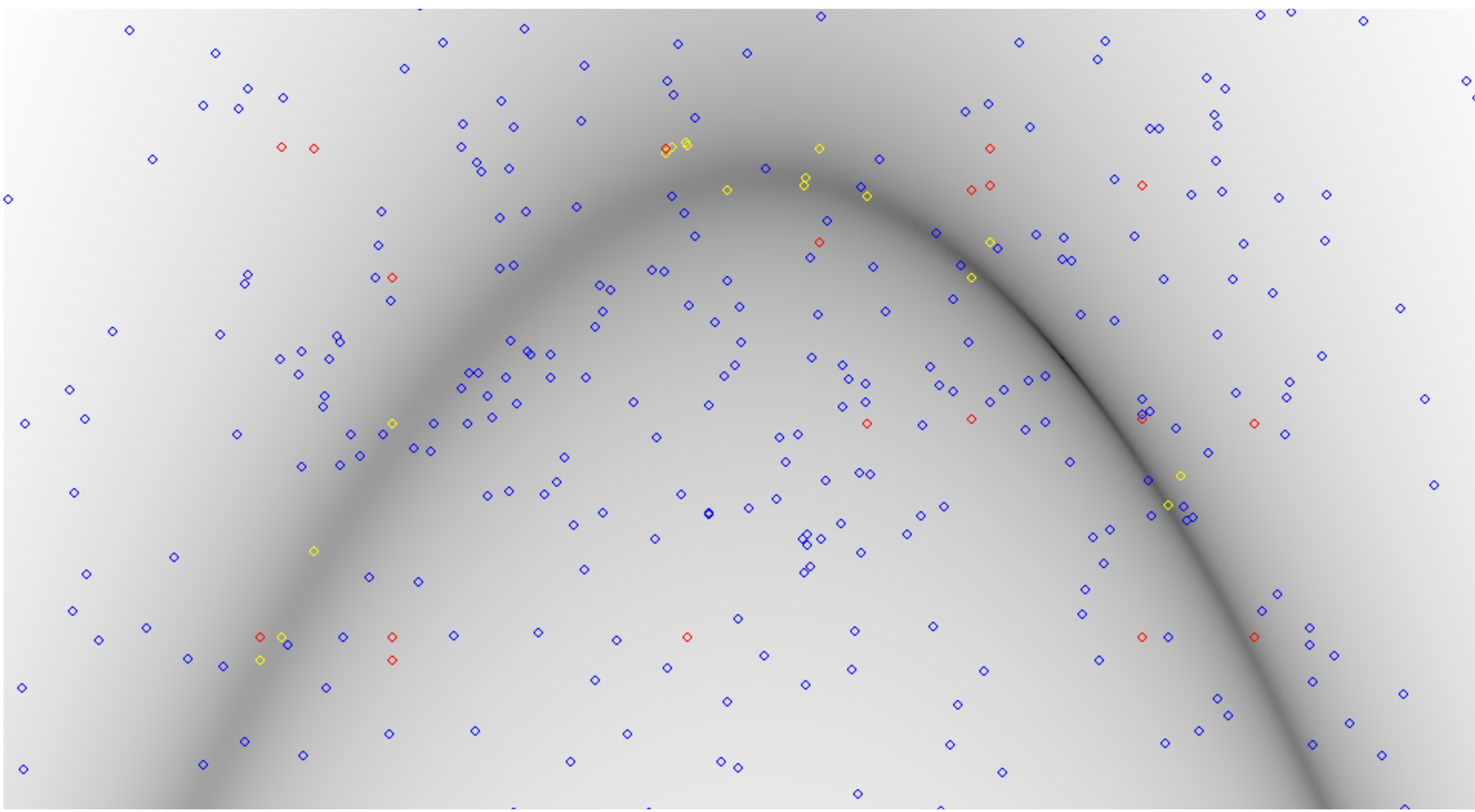
Often, a little bit of randomness goes a long way

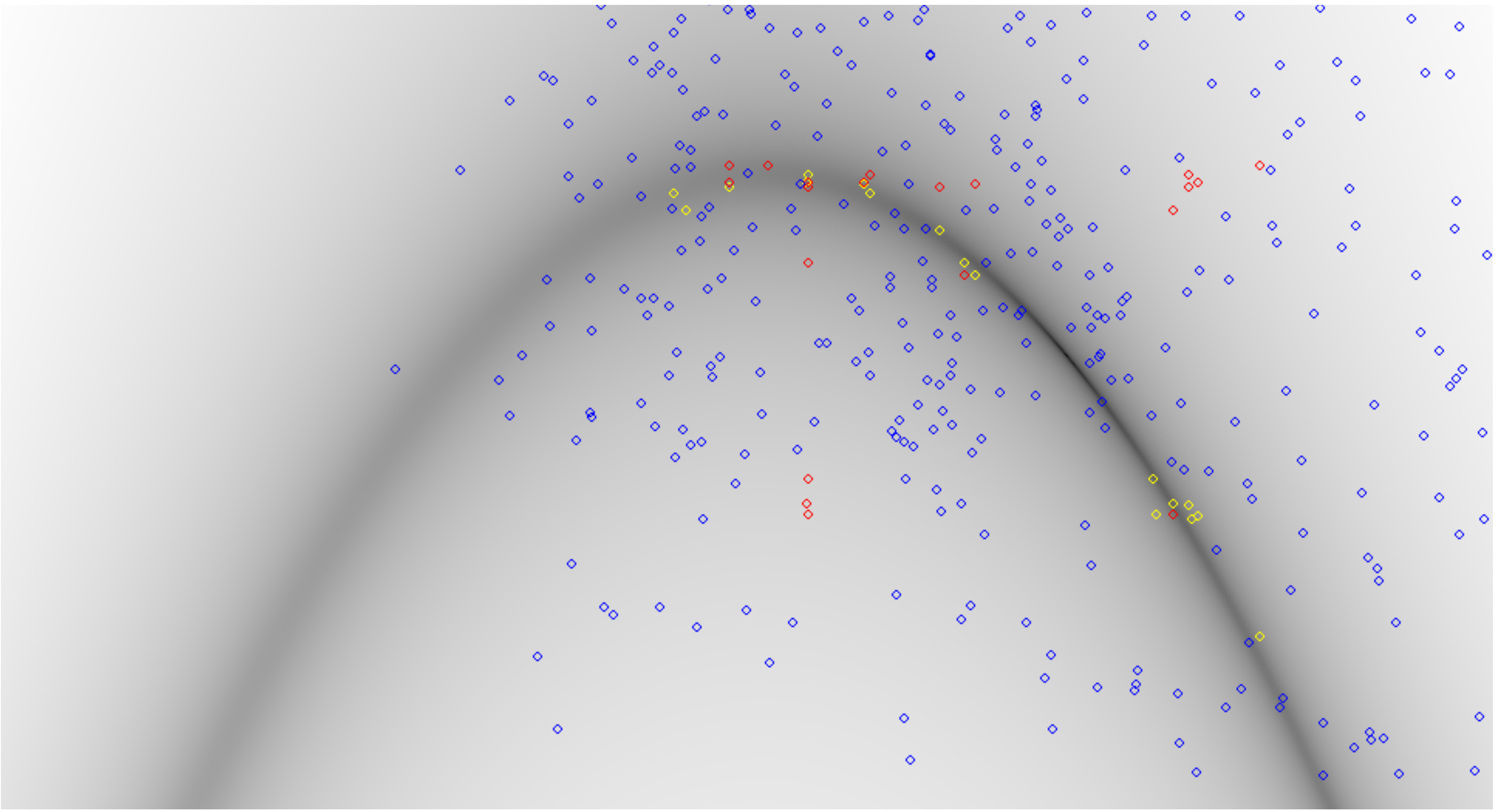
Genetic Algorithms

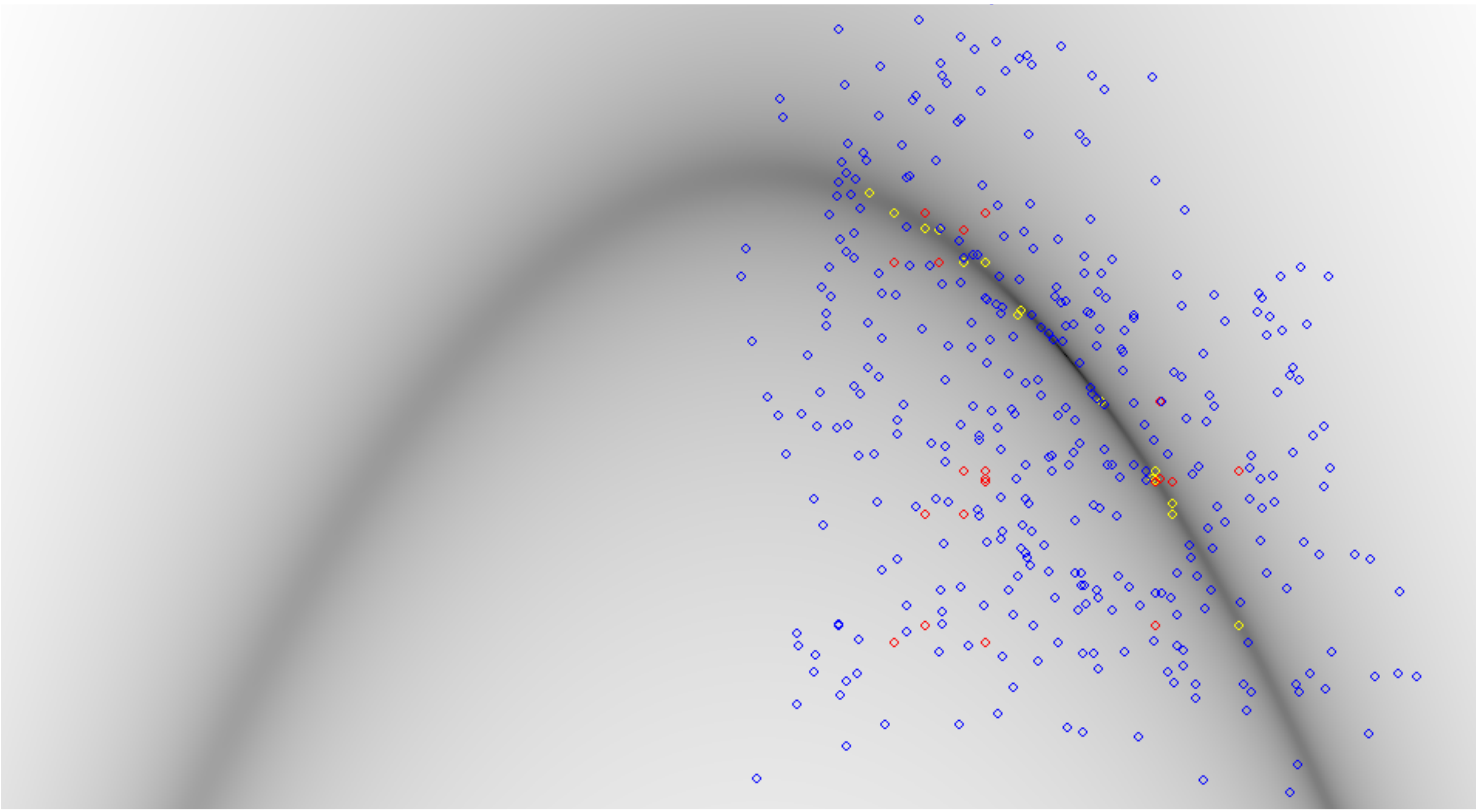
Genetic Algorithms

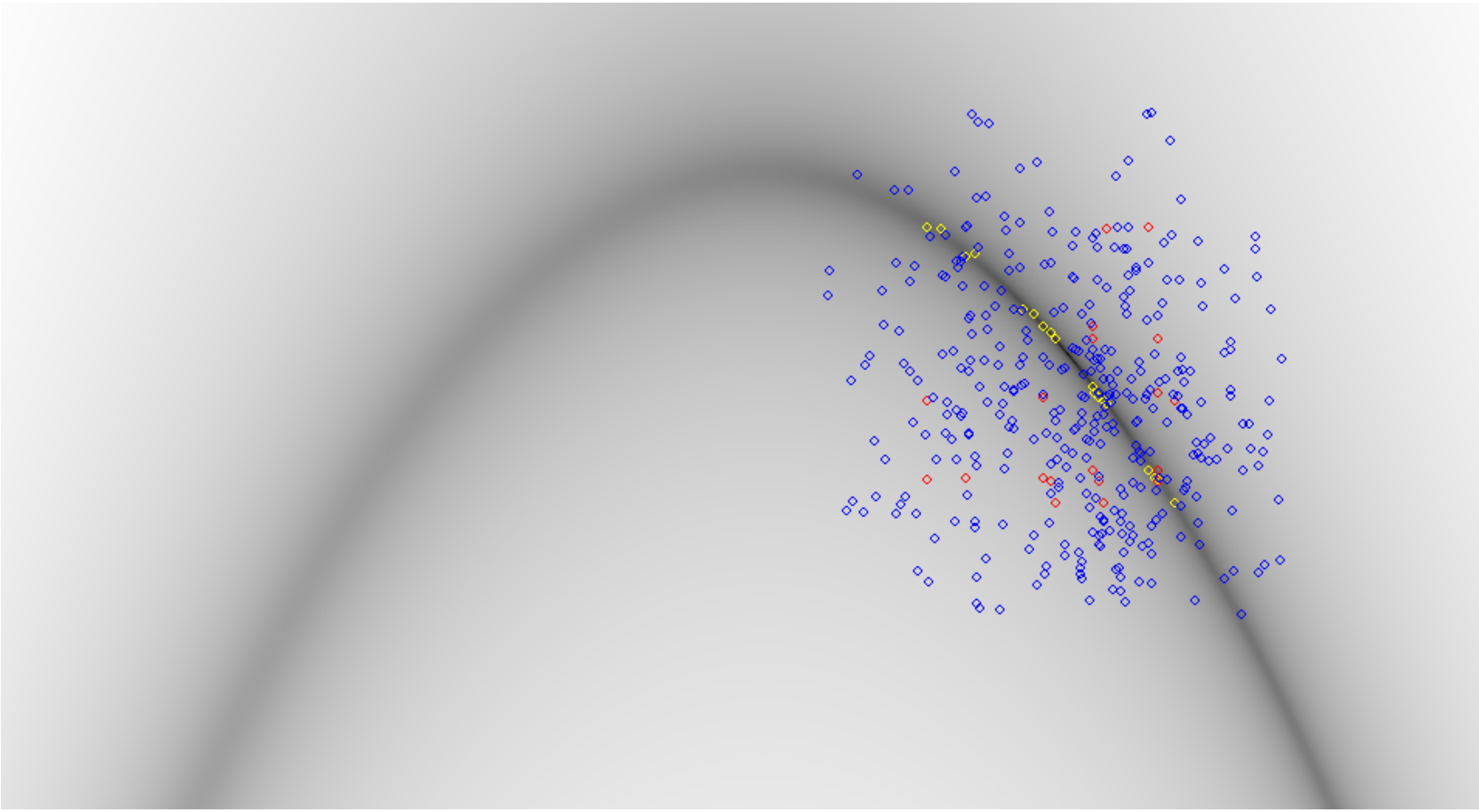
- Random solutions
- Survival of the fittest
- Sexual Reproduction
- Mutation

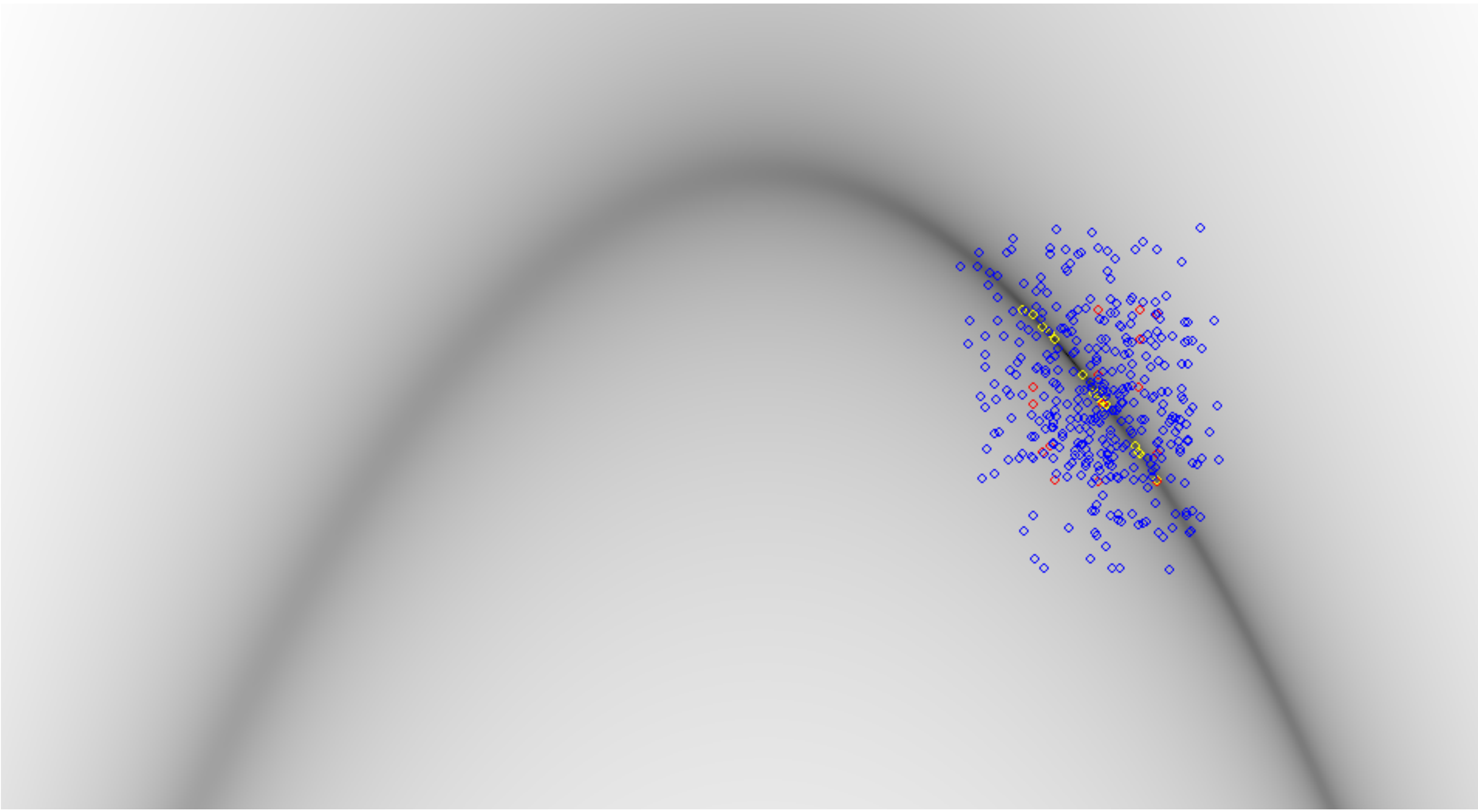


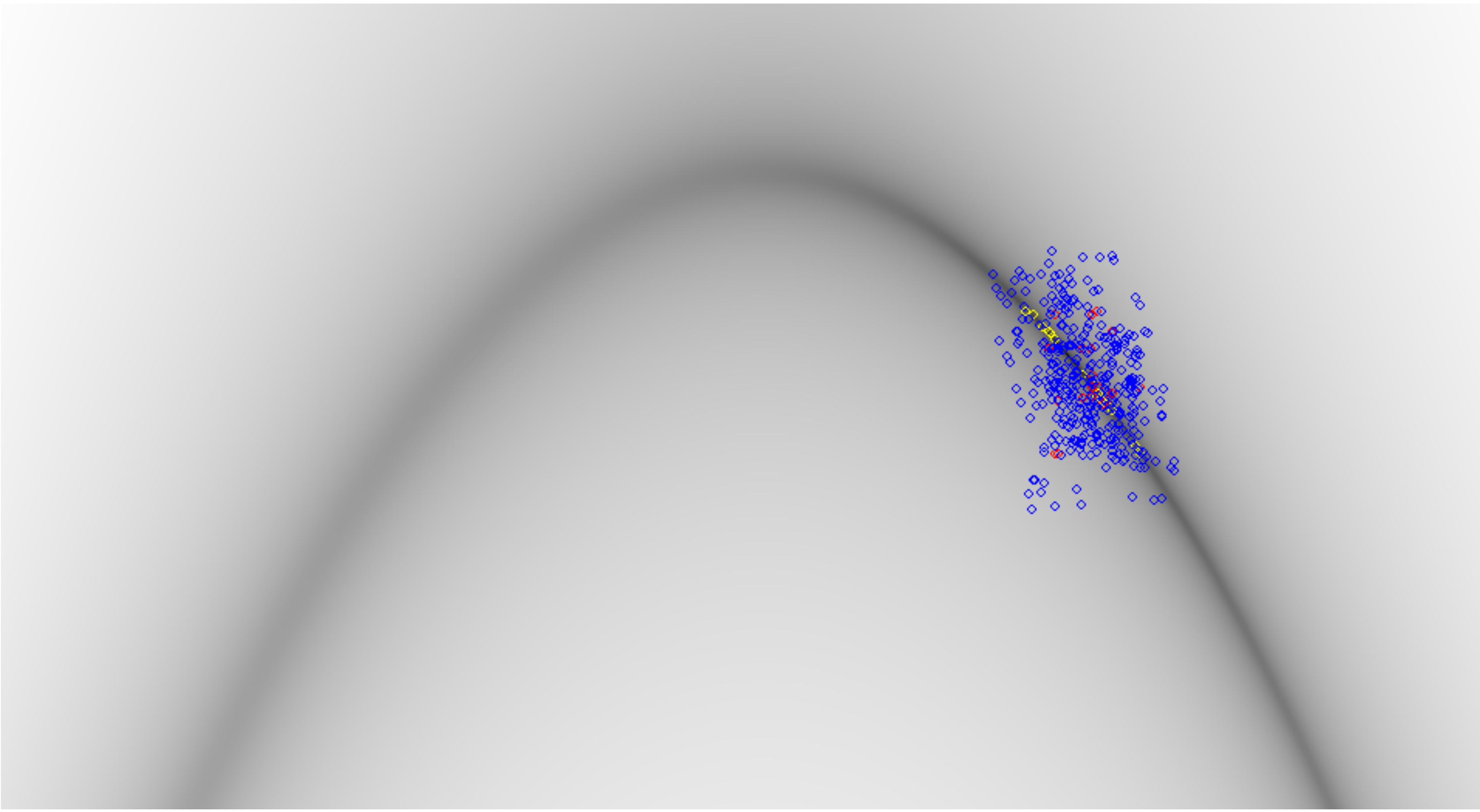


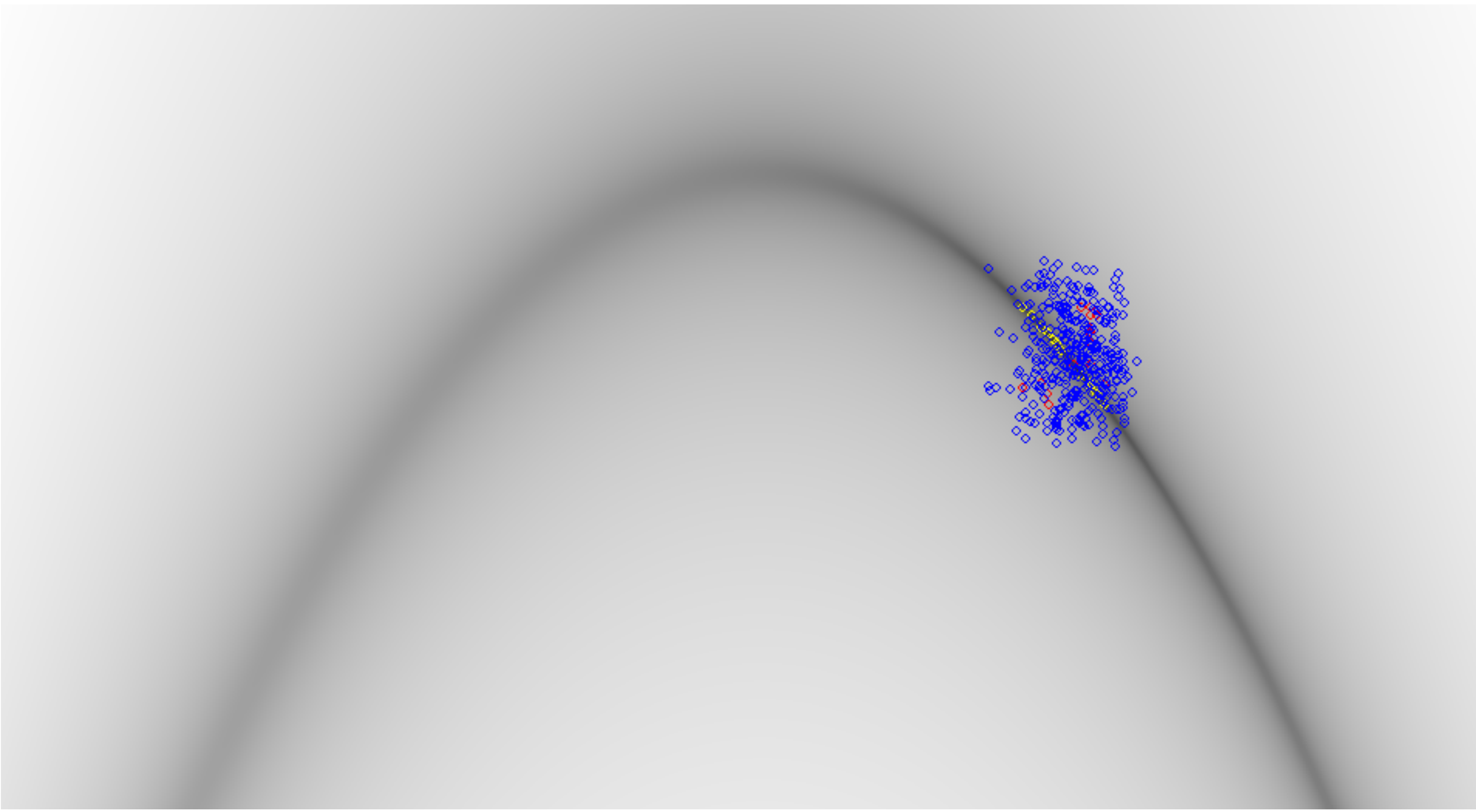


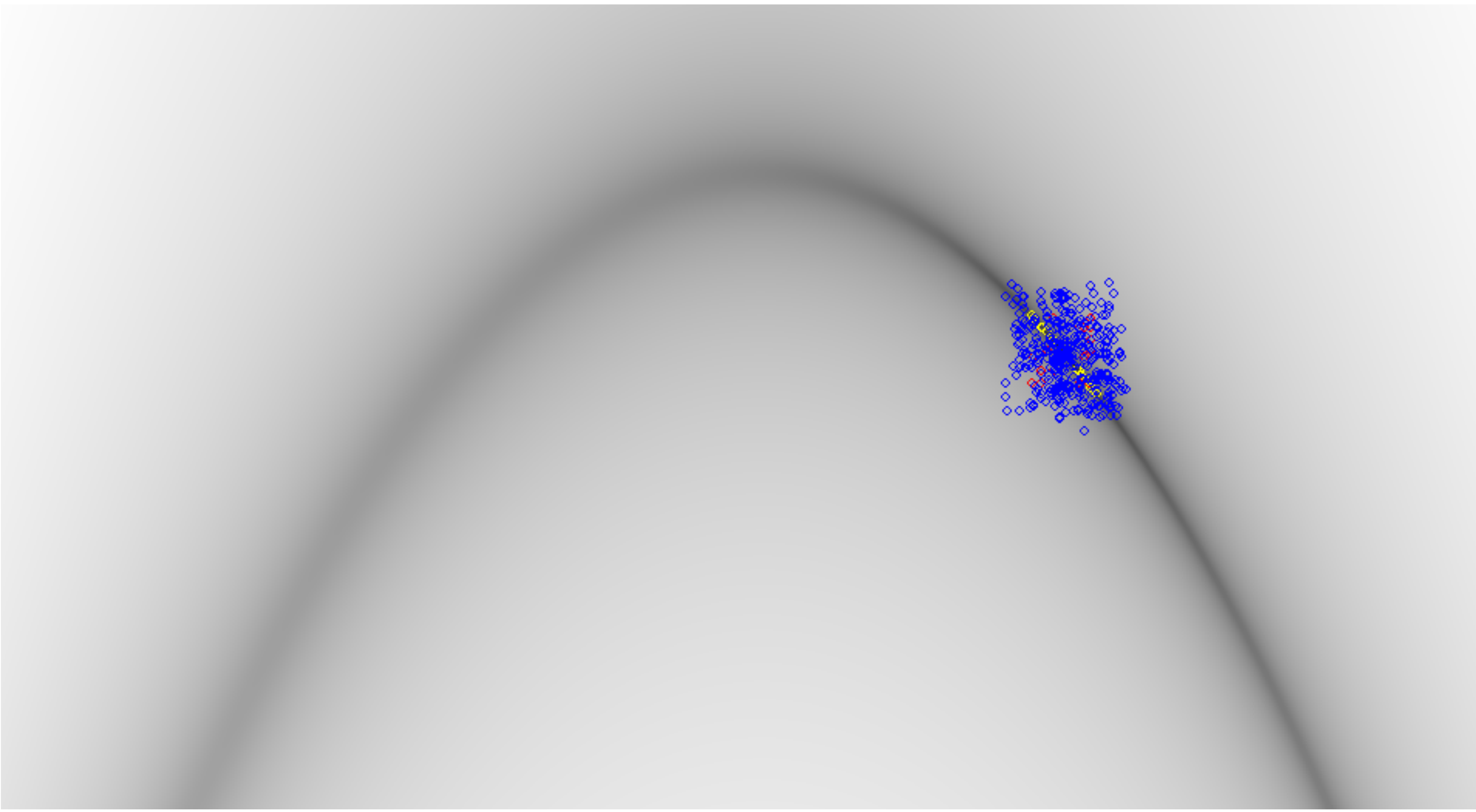


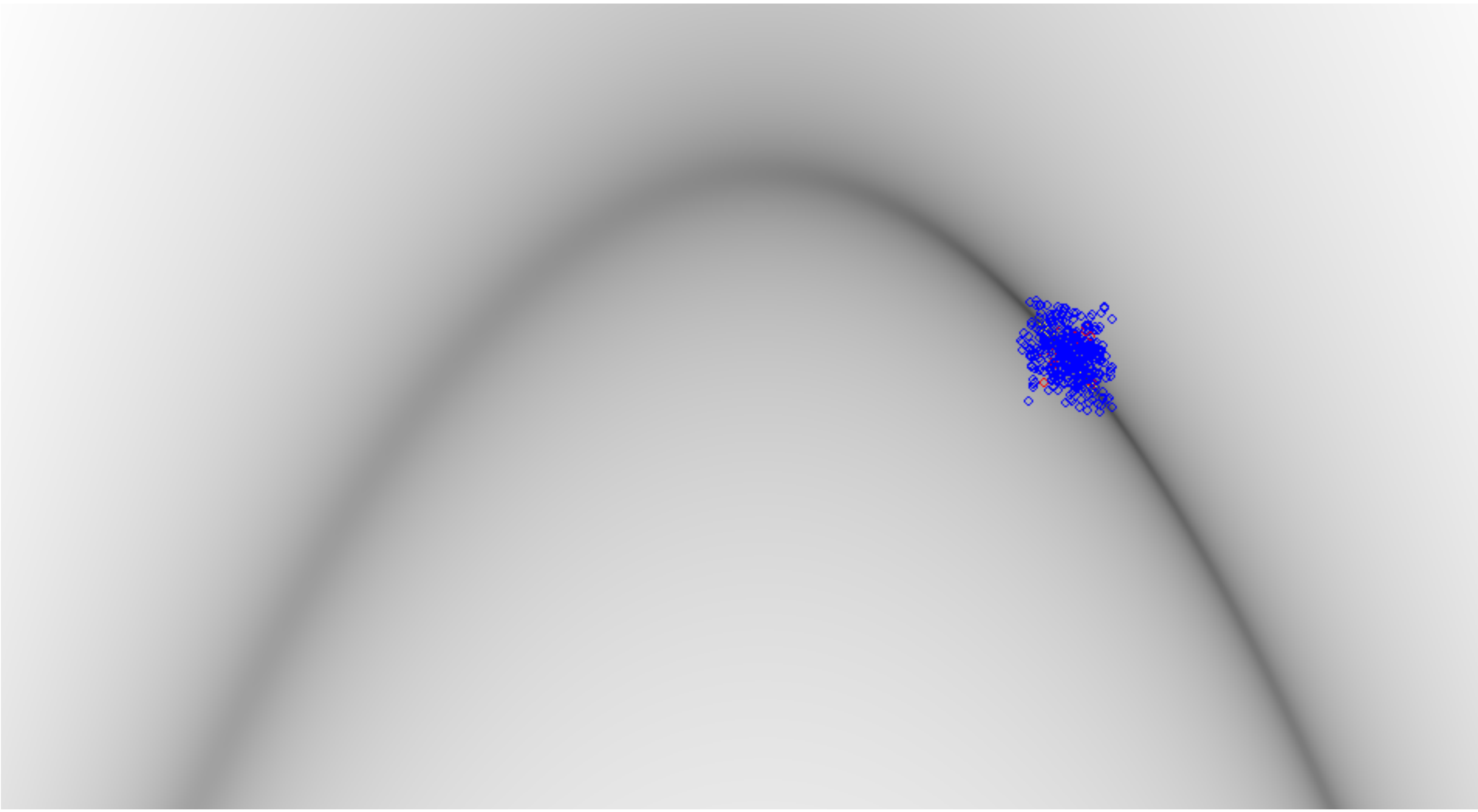


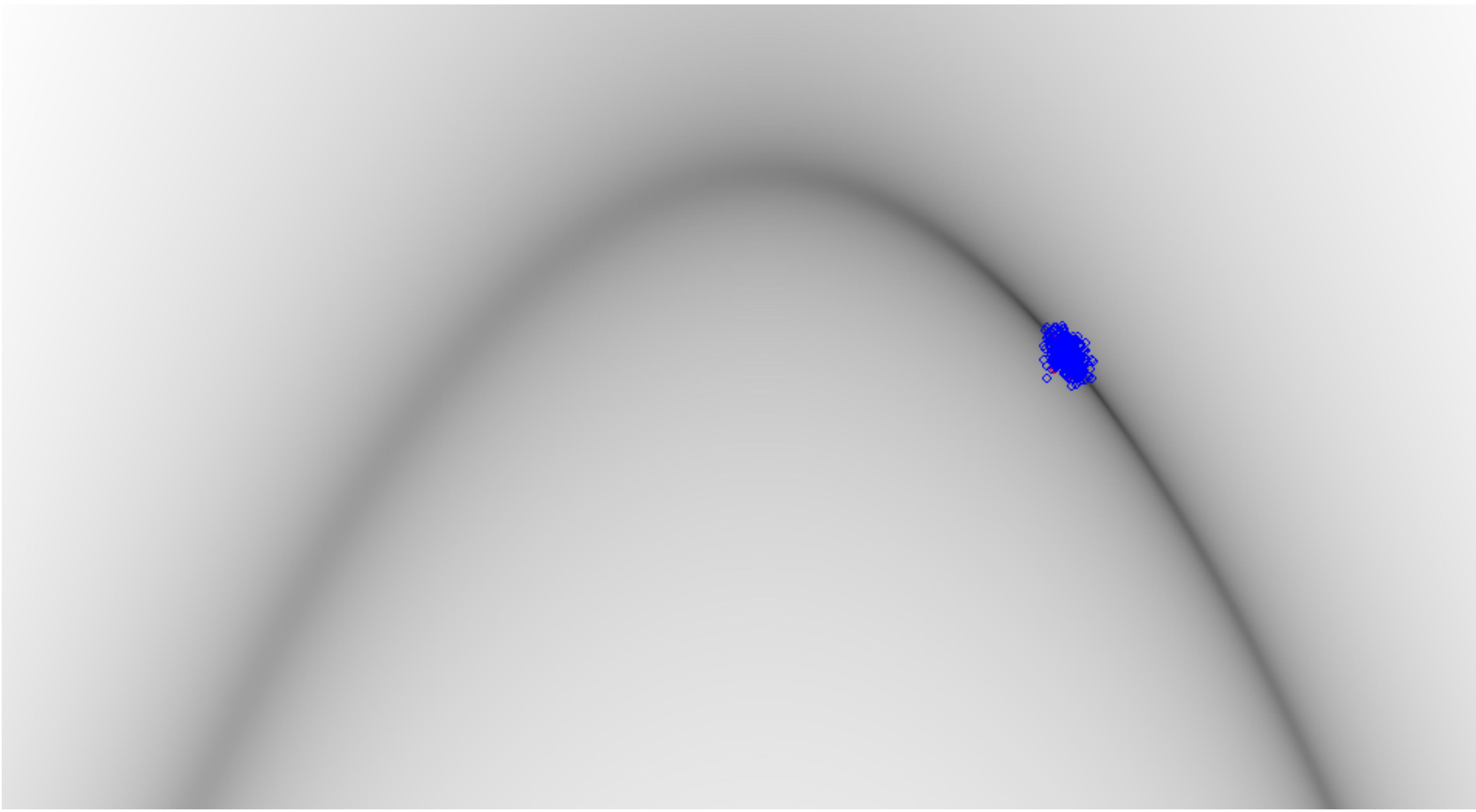


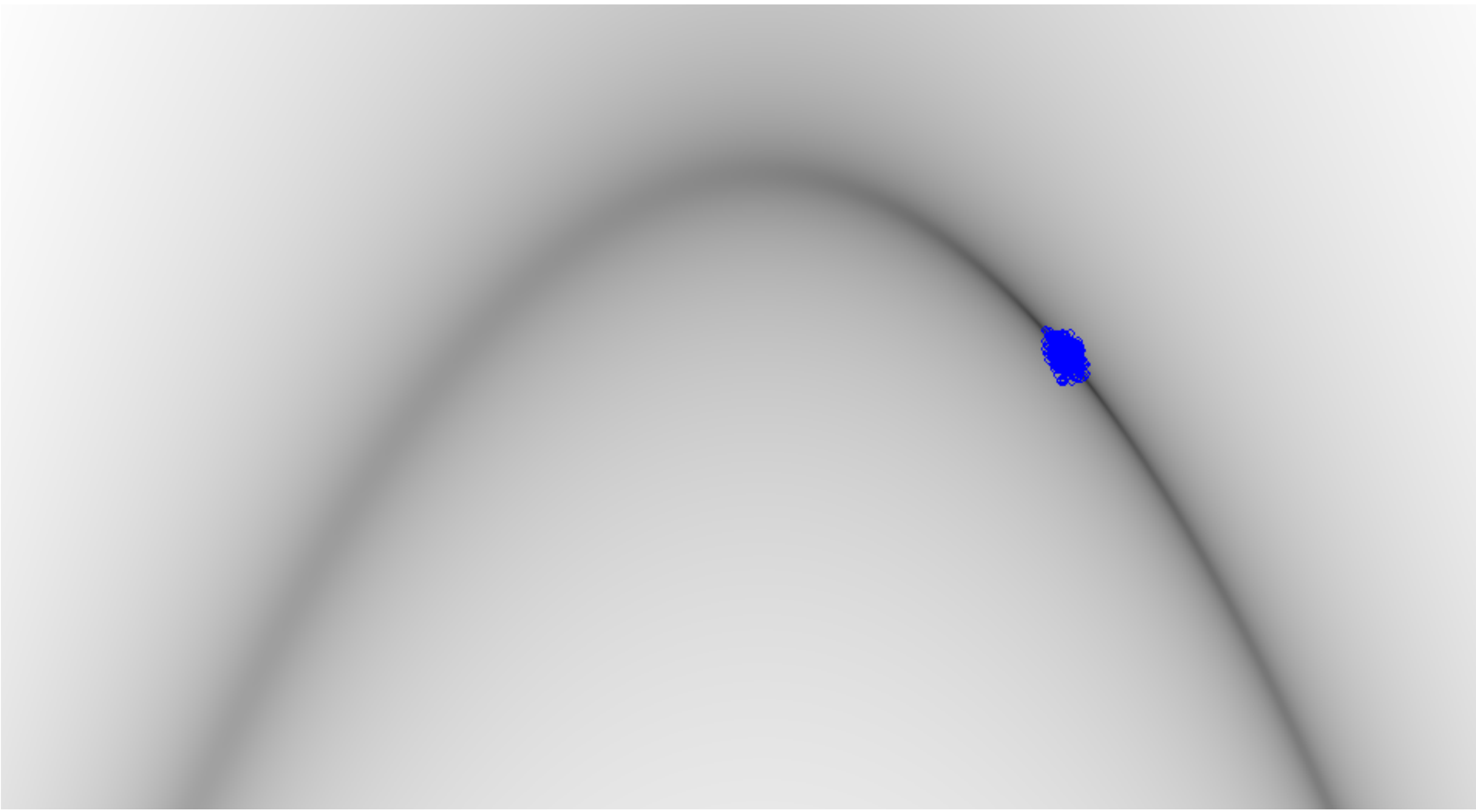


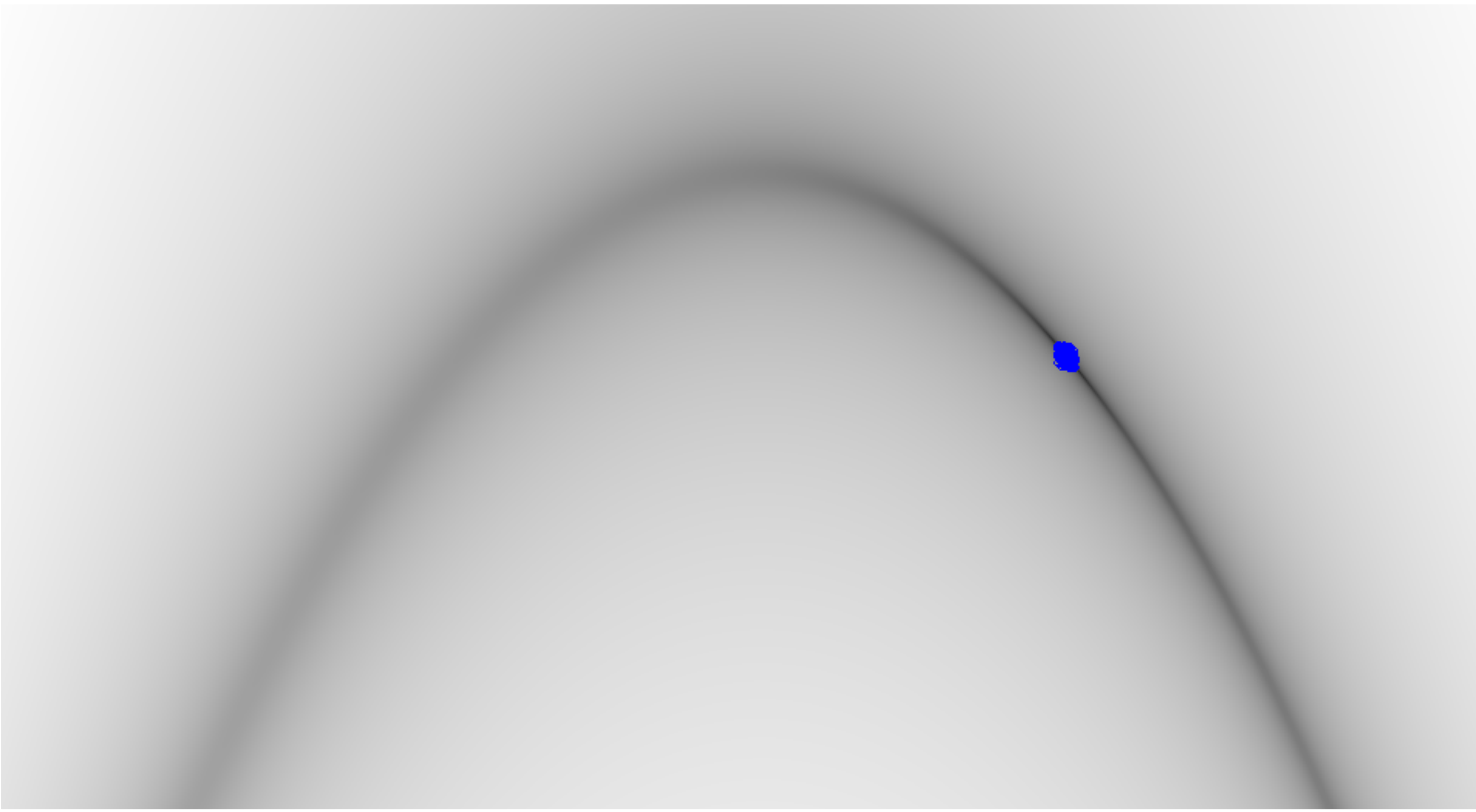


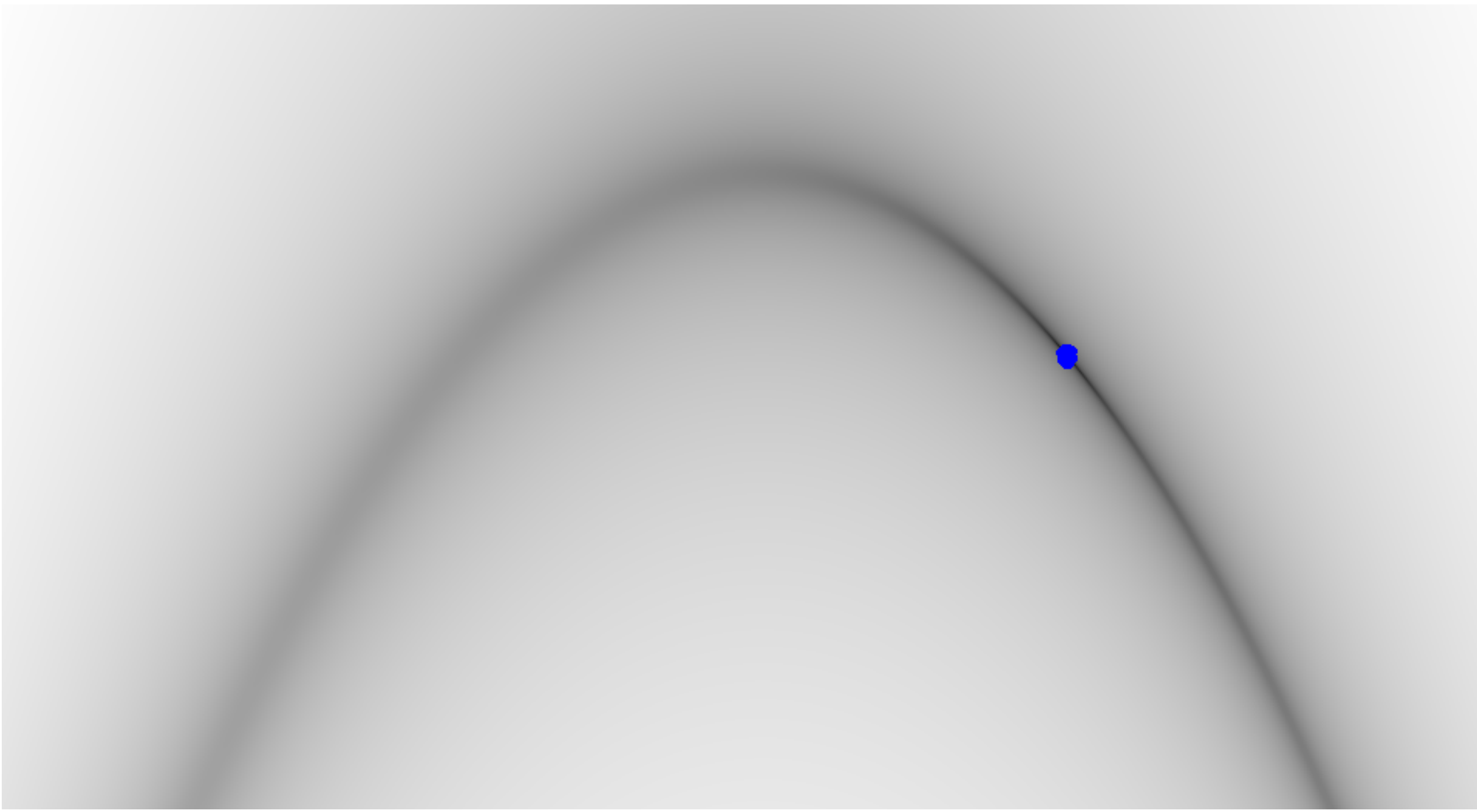










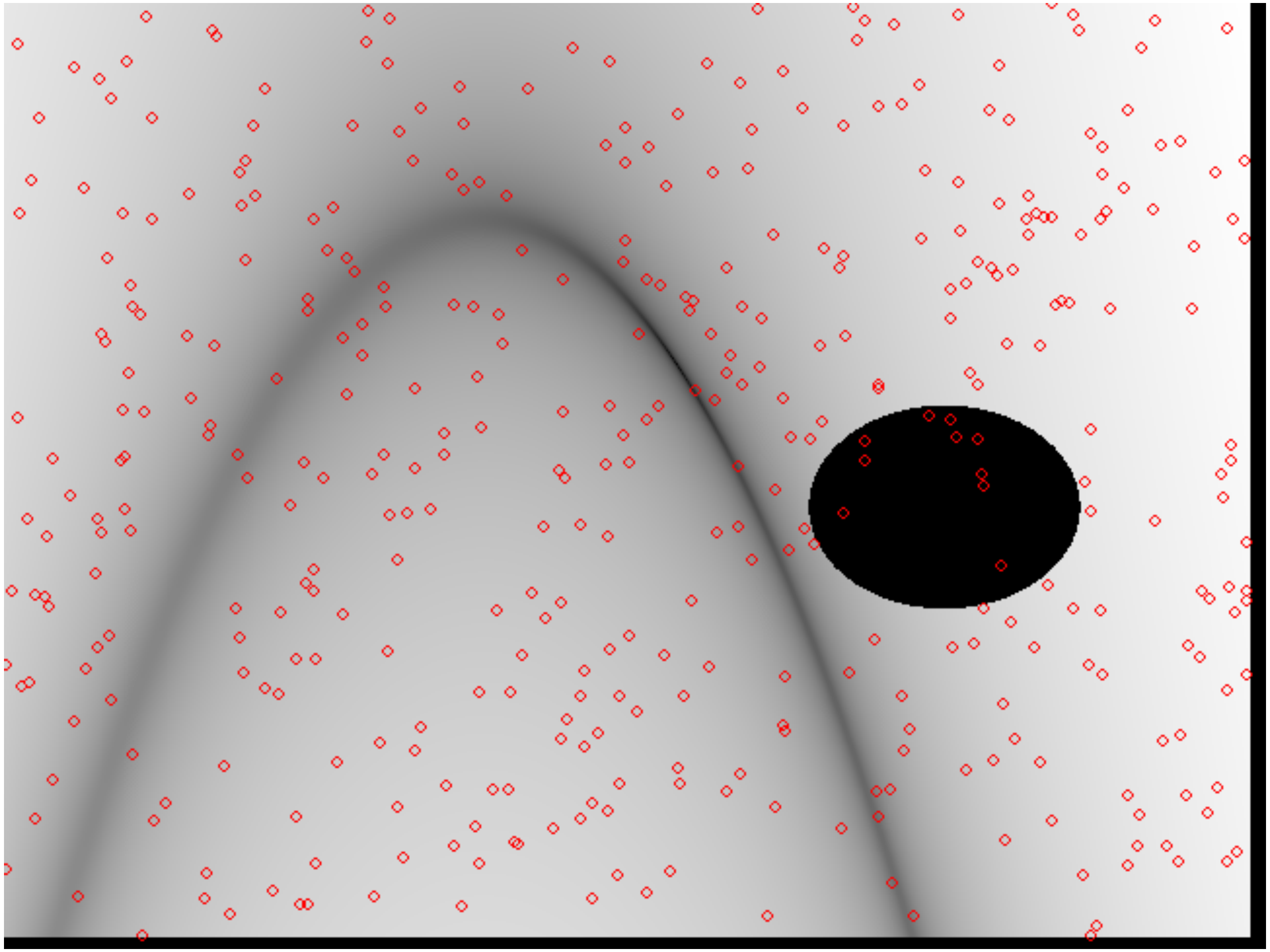


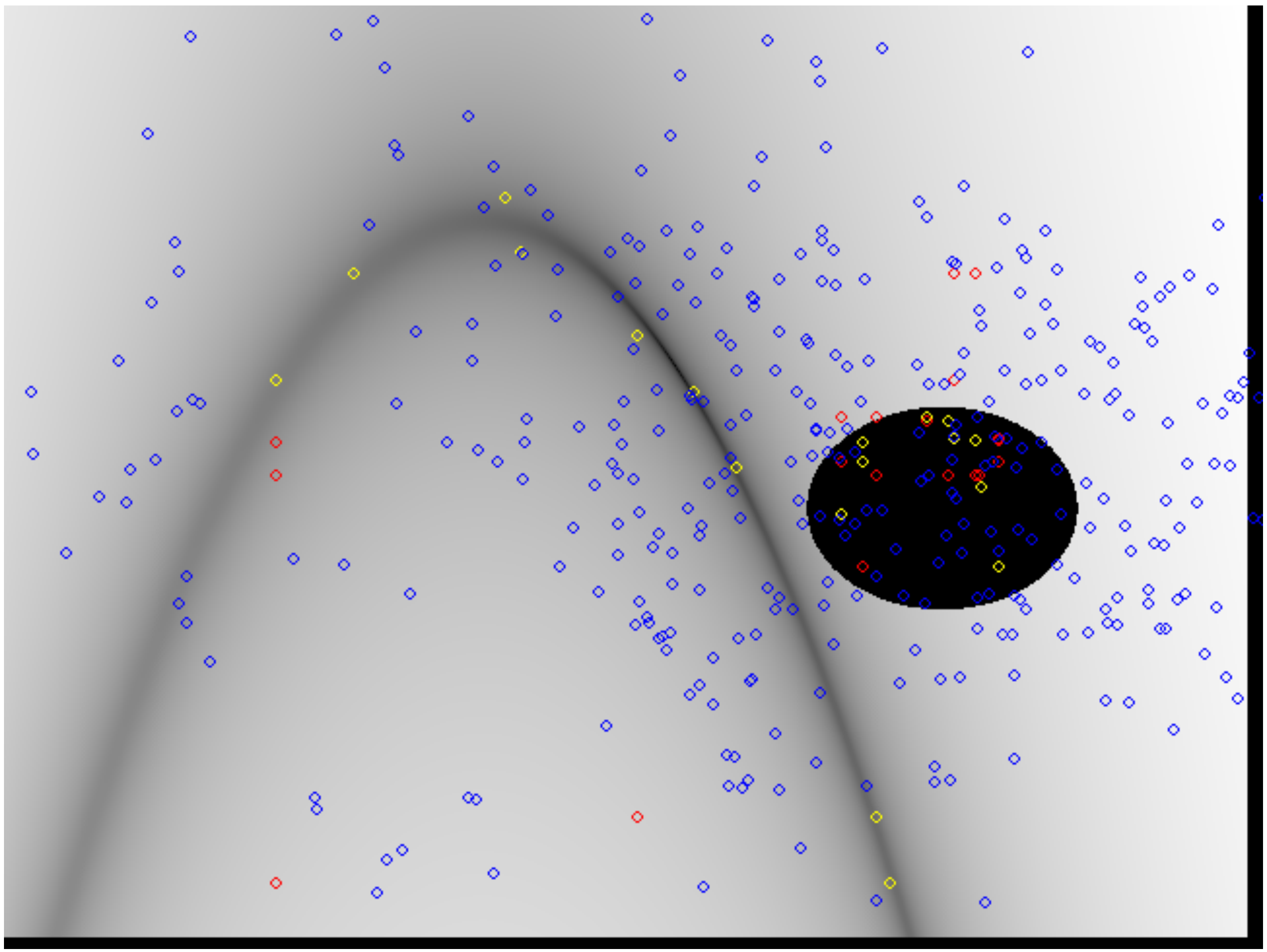
GA & Local Minima

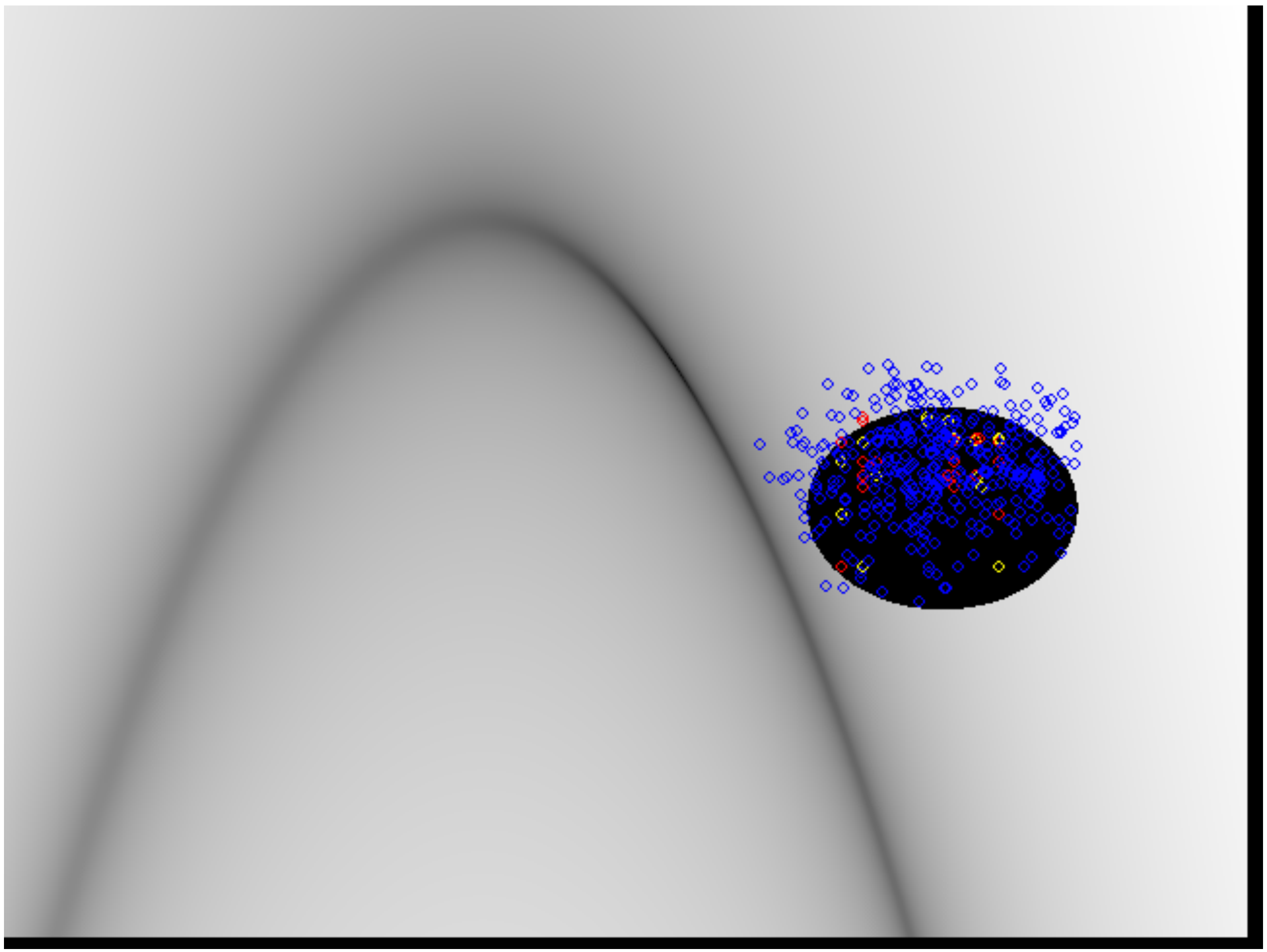
- The wide sampling of the initial 'scattergun' approach of the GA means that some points should fall near the global maxima

- Due to 'survival of the fittest' these rapidly form the basis of the population

GA & Local Minima







Weekly Assessment

- You are going to implement a 2D Gradient Descent and plot the results
- A gradient descent solver starts from a vector point
  - Just like your Euler solver for the spring
- The solver makes a series of steps that update the vector position with some function of the vector
    - Just like your Euler solver for the spring