# Stochastic steepest descent - Matteo Bunino

January 22, 2020

## 1 Code

[2]:
```
# Useful libraries
import numpy as np
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (13,5)
```

First of all I write the analytic function.
`f_noise` basically adds gaussian noise to $f(x, y)$ with mean 0 and standard deviation 15.

[97]:
```
def f(x,y):
    """Compute the analytic funciton"""
    return 18 + 11.4*x - 31*x**2 + 0.6*x**3 + x**4 + 50*y**2

def f_noise(x,y):
    """
    Add a random noise drawn from a normal distribution with mean 0 and std.␣
 ↪dev. 15
    x,y are scalar values. Return a sclar value.
    Is made to be avaluated in a single point.
    """
    mean = 0
    std = 15
    return f(x,y) + np.random.normal(mean, std)

def f_noise_vector(x,y):
    """
    Add a random noise drawn from a normal distribution with mean 0 and std.␣
 ↪dev. 15
    x,y are arrays. Return a sclar ND array.
    Useful to be plotted.
    """
    mean = 0
    std = 15
    return f(x,y) + np.random.normal(mean, std, size=f(x,y).shape)
```

The following function is used to locally estimate the stochastic function ($f(x, y)$ with noise added).

[103]:
```python
def estimate(f, x_0, y_0, N):
    """Estimates a stochastic valued function using a Monte Carlo simulation"""
    # N is the number of local estimations that will be averaged together
    est = 0
    for i in range(N):
        est += f(x_0,y_0) / N
    return est
```

`stochastic_sd` is the stochastic steepest descent algorithm with simultaneous perturbations. Note that in python functions is possible to specify default values for parameters. In this case I gave the default values to A end B as explained on the book.

[446]:
```python
def stochastic_sd(x_0,y_0,f_noise,mu_min,A=10,B=100,maxiter=500):
    """Stochastic steepest descent method"""
    # Variables of f(x1,x2)
    x1 = x_0
    x2 = y_0

    # Paramenters
    N = 10000 # Number of iterations in the estimation phase
    C = 0.1
    t = 1/6 #theo: 0.101
    l = 1 # theo: 0.602

    #Initial mu: I assume m=0
    mu = A/B**l

    for m in range(1,maxiter):

        # Perturbation coefficient
        c = C/m**t

        # Bernoulli RV: binomial(n_trials=1, probability=.5)
        H1 = 1 if np.random.binomial(1,0.5) else -1
        H2 = 1 if np.random.binomial(1,0.5) else -1

        h1 = c*H1
        h2 = c*H2

        # Function evaluation
        F_plus = estimate(f_noise, x1+h1, x2+h2, N)
        F_minus = estimate(f_noise, x1-h1, x2-h2, N)

        # Upgrade x,y
        x1 = x1 - mu*(F_plus-F_minus)/(2*h1)
        x2 = x2 - mu*(F_plus-F_minus)/(2*h2)
```

```
        # Update step size
        mu = A/(B+m)**l

        # Stopping criterion
        if mu < mu_min:
            break

    return x1,x2,m,mu
```

`stochastic_sd_fin_diff` is a function to compute the stochastic steepest descent using the finite differences approach, instead of the simultaneous perturbations.

[482]:
```
def stochastic_sd_fin_diff(x_0,y_0,f_noise,mu_min,A=10, B=100,maxiter=500):
    # Tuning variables
    x1 = x_0
    x2 = y_0

    N = 10000 # Number of iterates in the estimation phase
    C = 0.1
    t = 0.01 #1/6 #theo: 0.101
    l = 1 # theo: 0.602

    #Initial mu: I assume m=0
    mu = A/B**l

    for m in range(1,maxiter):

        # Perturbation coefficient
        c = C/m**t

        # Bernoulli RV: binomial(n_trials=1, probability=.5)
        H1 = 1 if np.random.binomial(1,0.5) else -1
        H2 = 1 if np.random.binomial(1,0.5) else -1

        h1 = c*H1
        h2 = c*H2

        # Function evaluation
        F1_plus = estimate(f_noise, x1+h1, x2, N)
        F1_minus = estimate(f_noise, x1-h1, x2, N)
        gradf_x1 = (F1_plus-F1_minus)/(2*h1)

        F2_plus = estimate(f_noise, x1, x2+h2, N)
        F2_minus = estimate(f_noise, x1, x2-h2, N)
        gradf_x2 = (F2_plus-F2_minus)/(2*h2)

        # Upgrade x,y
```

```
        x1 = x1 - mu*gradf_x1
        x2 = x2 - mu*gradf_x2

        # Update step size
        mu = A/(B+m)**1

        # Stopping criterion
        if mu < mu_min:
            break

    return x1,x2,m,mu
```

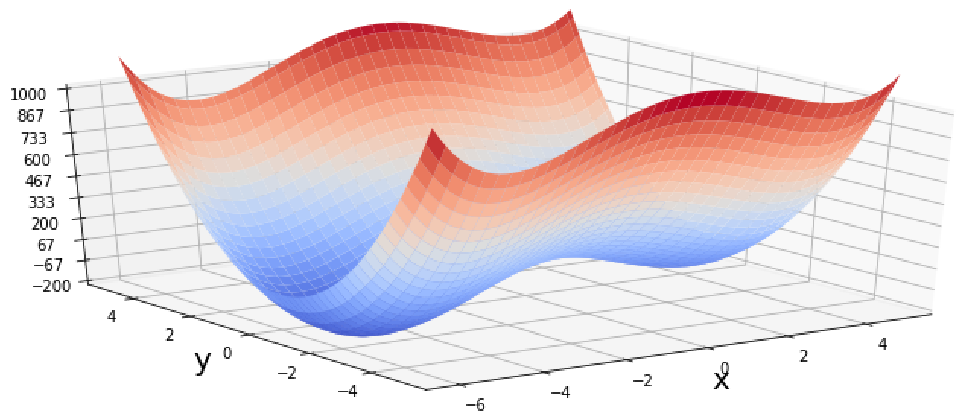
The following function is for 3D plotting $f(x, y)$.

[389]:
```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
from matplotlib import cm
import numpy as np

def plot_surface(f,min_x,max_x,min_y,max_y):
    """A function to 3d plot f(x,y)"""
    fig = plt.figure()
    ax = fig.gca(projection='3d')

    # Data
    X = np.arange(min_x, max_x, 0.1)
    Y = np.arange(min_y, max_y, 0.1)
    X, Y = np.meshgrid(X, Y)
    Z = f(X,Y)

    # Plot the surface
    surf = ax.plot_surface(X, Y, Z, cmap=cm.coolwarm,
                           linewidth=0, antialiased=True)
    ax.set_xlabel('x', fontsize=20)
    ax.set_ylabel('y', fontsize=20)

    # Rotation
    ax.view_init(azim=-125)

    # Customize the z axis
    ax.set_zlim(-200, 1000)
    ax.zaxis.set_major_locator(LinearLocator(10))

    plt.show()
```

## 2 Discussion

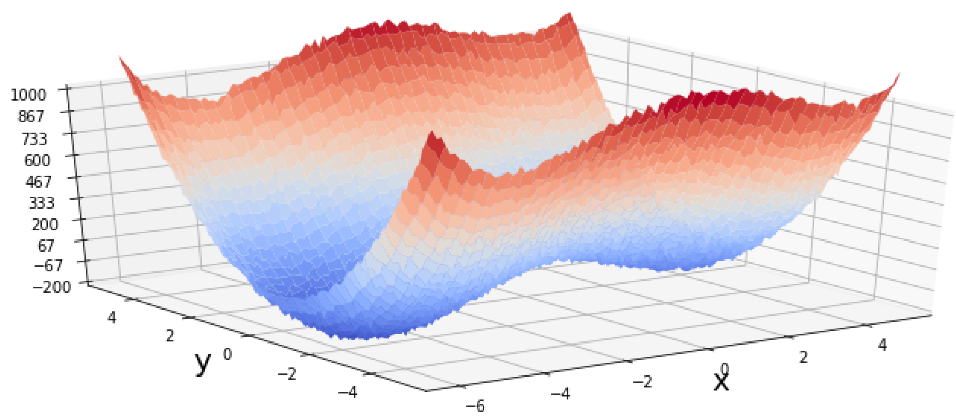
First of all, since in this case we know the closed analytical form of the function, it may be a good idea to visualize it, plotting it in a 3D surface.

[70]:
```
plot_surface(f,-6,5,-5,5)
```




Here we can see how the shape of the function appears after having added gaussian noise with mean 0 and standard deviation 15.

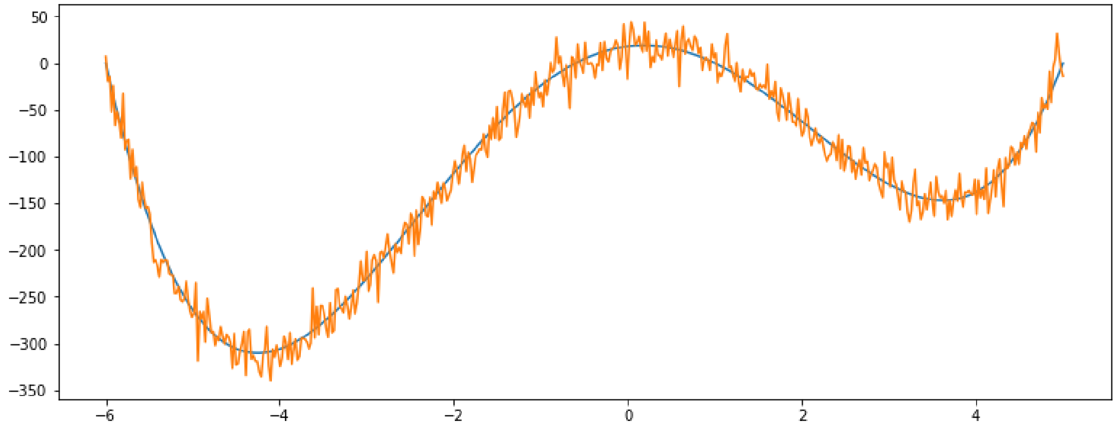
[91]:
```
plot_surface(f_noise_vector,-6,5,-5,5)
```




This is a 2D visualization of how the noise version of $f(x, y)$ may appear.

[122]:
```
x = np.linspace(-6,5,500)
y = np.linspace(-5,5,500)
plt.plot(x,f(x,0))
plt.plot(x,f_noise_vector(x,0))
```

```python
plt.show()
```




### 2.0.1 Algorithm description

The stochastic steepest descent is a simulation-based technique for solving stochastic problems of continous parametric optimization.
This method is in the model-free techniques family, namely the techniques that do not require structural properties (analytical form, density of the random variables…) of the objective function. These techniques are used assuming that it is possible to estimate the true value of the objective function ($f\_noise(x, y)$ in this case) at any given point by averaging the objective values, obtained from numerous simulation runs at the same point.

Simultaneous perturbations means that when computing the gradient for the steepest descent phase, we simultaneously perturbate all the $n$ variables of the function $f \in \Re^n$, resulting in only two function evaluation per step. Using finite differences approach instead, we need $2n$ function avaluations, which may be very computationally expensive when the objective function has to be estimated.

After havign computed the gradient, the method makes a step of length $\mu^m$ in the opposite direction. The steplength decreases at each iteration and its updating rule has to be tuned to guarantee convergence.
Since in this case the gradient may never vanish due to noise, the stopping criterion is set on $\mu$. When it becomes small enough, the method stops.

After having implemented the algoritm, I test it generating some starting points uniformely distributed in the intervals $x \in [-6, 5]$, $y \in [-5, 5]$.

The default values for the parameters A and B are 10 and 100 respectively, as suggested on the book

[438]:
```python
import pandas as pd

# Data Frame: data structure used to collect results
```

```
results = pd.DataFrame()

n_points = 10

# I randomly generate, with an uniform distribution
# a number of initial x and y for the method,
# in the intervals said above.
x = np.random.uniform(-6,5,n_points)
y = np.random.uniform(-5,5,n_points)

# min mu: the smallest step before stopping (stopping criterion)
mu_min = 0.005

for i in range(n_points):
    # First I compute the minimum, starting by (x[i], y[i])
    x_min, y_min, n_iter, last_mu =␣
 ↪stochastic_sd(x[i],y[i],f_noise,mu_min,maxiter=10)

    # Then I save the results in the dataframe
    results = results.append([[x[i], y[i], x_min, y_min, n_iter, last_mu]])
```

[439]:
```
# I give names to columns to make it more understandable
results.columns = ['x_start','y_start','x_min','y_min','n_iterations','last_mu']
results
```

[439]:

| | x_start | y_start | x_min | y_min | n_iterations | last_mu |
|---|---|---|---|---|---|---|
| 0 | -4.219449 | -4.555746 | -9.836961e+38 | 9.836961e+38 | 9 | 0.091743 |
| 0 | 4.498385 | -4.673252 | -5.817831e+36 | 5.817831e+36 | 9 | 0.091743 |
| 0 | -5.106813 | 2.872490 | 8.280147e+27 | -8.280147e+27 | 9 | 0.091743 |
| 0 | 4.983088 | -2.725367 | 7.544380e+38 | 7.544380e+38 | 9 | 0.091743 |
| 0 | 3.397470 | -0.935545 | -3.235899e+26 | -3.235899e+26 | 9 | 0.091743 |
| 0 | -2.013085 | 0.426922 | 1.587006e+17 | 1.587006e+17 | 9 | 0.091743 |
| 0 | 2.274317 | -3.096531 | 4.416369e+30 | 4.416369e+30 | 9 | 0.091743 |
| 0 | 1.658723 | 0.241786 | -1.767675e+23 | -1.767675e+23 | 9 | 0.091743 |
| 0 | 2.972071 | -3.763985 | -3.381964e+38 | -3.381964e+38 | 9 | 0.091743 |
| 0 | -2.724931 | 3.437916 | -5.035863e+29 | -5.035863e+29 | 9 | 0.091743 |

As we can see form the results, even if it's making a very small number of iterations, the computed minima are quite far from the starting point. In other words this model, in this configuration, isn't able to converge.

The motivation of this behaviour can be found in the steplength that is done at each iteration. Unlike in the steepest descent or conjugate gradient methods, in this case we cannot compute a suitable steplength using a line search since we don't a closed analytical form for the function we're minimizing. In fact, in this method the steplength is decreasing at each iteration, but its magnitude has to be tuned.
With the default A and B the starting mu, according to the book definition is:

$$\mu = \frac{A}{(B+m)^l} = \frac{A}{B} \text{ if } m = 0 \text{ and } l = 1 \tag{1}$$

If A = 10 and B = 100, $\mu_0 = 0.1$

Where the funciton is steep, the magnitude of gradient may be pretty large, pushing us far form the solution, unless we take a suitable steplength. A reduction of 0.1 times may not be enough. On the other hand if $\mu_0$ is too small, this method will take a too large amount of iterations to reach the solution.

After some tuning I set the parameters A and B to 1 and 100 respectively. This gives a initial $\mu = 0.01$.
The results on 10 random starting points are presented below:

[472]:
```python
results_tuned = pd.DataFrame()

n_points = 10

x = np.random.uniform(-6,5,n_points)
y = np.random.uniform(-5,5,n_points)

# min mu: the smallest step before stopping
mu_min = 0.005

for i in range(n_points):
    x_min, y_min, n_iter, last_mu = stochastic_sd(x[i],y[i],f_noise,mu_min,A=1,B=100,maxiter=200)
    results_tuned = results_tuned.append([[x[i], y[i], x_min, y_min, n_iter, last_mu]])

results_tuned.columns = ['x_start','y_start','x_min','y_min','n_iterations','last_mu']
results_tuned
```

[472]:

| | x_start | y_start | x_min | y_min | n_iterations | last_mu |
|---|---|---|---|---|---|---|
| 0 | 4.088829 | 2.468821 | 3.612665e+00 | 6.794508e-04 | 101 | 0.004975 |
| 0 | 2.018045 | -4.277550 | -2.451272e+33 | 2.451272e+33 | 101 | 0.004975 |
| 0 | 2.044320 | 0.655128 | 3.638497e+00 | -3.549272e-03 | 101 | 0.004975 |
| 0 | -1.170946 | 0.378306 | 1.534917e+31 | 1.534917e+31 | 101 | 0.004975 |
| 0 | -5.548760 | -2.096768 | 3.603503e+00 | 3.958749e-03 | 101 | 0.004975 |
| 0 | -1.603375 | -1.530543 | -4.264412e+00 | -1.875660e-02 | 101 | 0.004975 |
| 0 | -2.144947 | -2.402796 | -4.238084e+00 | 2.130416e-02 | 101 | 0.004975 |
| 0 | -0.101619 | -2.008859 | 3.613258e+00 | 2.475120e-02 | 101 | 0.004975 |
| 0 | -4.542058 | -4.109976 | 1.862334e+15 | 1.862334e+15 | 101 | 0.004975 |
| 0 | -5.108166 | -2.653799 | 1.009473e+20 | 1.009473e+20 | 101 | 0.004975 |

As we can see, with this new configuration only in four cases the algorithm didn' converge.

With this new knowledge it's possible to try to further reduce the initial step (which is indeed the biggest one), setting B = 150, that gives a initial $\mu \simeq 0.007$.

[473]:
```
results_tuned = pd.DataFrame()

n_points = 10

x = np.random.uniform(-6,5,n_points)
y = np.random.uniform(-5,5,n_points)

# min mu: the smallest step before stopping
mu_min = 0.005

for i in range(n_points):
    x_min, y_min, n_iter, last_mu =␣
 ↪stochastic_sd(x[i],y[i],f_noise,mu_min,A=1,B=150,maxiter=200)
    results_tuned = results_tuned.append([[x[i], y[i], x_min, y_min, n_iter,␣
 ↪last_mu]])

results_tuned.columns =␣
 ↪['x_start','y_start','x_min','y_min','n_iterations','last_mu']
results_tuned
```

[473]:

| | x_start | y_start | x_min | y_min | n_iterations | last_mu |
|---|---|---|---|---|---|---|
| 0 | 3.185360 | -4.062680 | -4.259711 | -0.005402 | 51 | 0.004975 |
| 0 | -1.957413 | 1.254194 | -4.269899 | 0.016928 | 51 | 0.004975 |
| 0 | -3.092821 | 2.889610 | -4.253163 | 0.016713 | 51 | 0.004975 |
| 0 | 1.816423 | -0.936746 | 3.612582 | -0.019575 | 51 | 0.004975 |
| 0 | 3.856183 | 0.745777 | 3.602398 | 0.010796 | 51 | 0.004975 |
| 0 | 1.825838 | -3.328405 | -4.254548 | -0.016623 | 51 | 0.004975 |
| 0 | 2.011935 | -0.935928 | 3.618551 | 0.009862 | 51 | 0.004975 |
| 0 | -0.263404 | -3.273082 | -4.253926 | 0.012769 | 51 | 0.004975 |
| 0 | -2.700119 | -1.195369 | -4.245650 | 0.007997 | 51 | 0.004975 |
| 0 | -3.825884 | 2.904681 | -4.263372 | 0.000363 | 51 | 0.004975 |

The situation has furtherly improved: none of the starting points made the method diverge. This confirms the previous assumption that tuning the steplength can be crucial to the convergence of this method.

---

Let's make one last try, furtherly reducng the initial $\mu$ and the $\mu_{min}$ to get a more accurate solution, setting A=1, B=200.

[476]:
```
results_tuned = pd.DataFrame()

n_points = 10

x = np.random.uniform(-6,5,n_points)
```

```
y = np.random.uniform(-5,5,n_points)

# min mu: the smallest step before stopping
mu_min = 0.002

for i in range(n_points):
    x_min, y_min, n_iter, last_mu =␣
 ↪stochastic_sd(x[i],y[i],f_noise,mu_min,A=1,B=200,maxiter=200)
    results_tuned = results_tuned.append([[x[i], y[i], x_min, y_min, n_iter,␣
 ↪last_mu]])

results_tuned.columns =␣
 ↪['x_start','y_start','x_min','y_min','n_iterations','last_mu']
results_tuned
```

[476]:

| | x_start | y_start | x_min | y_min | n_iterations | last_mu |
|---|---|---|---|---|---|---|
| 0 | -3.528896 | 3.725619 | -4.249007 | -0.011023 | 199 | 0.002506 |
| 0 | -3.949378 | -0.089264 | -4.246553 | -0.000857 | 199 | 0.002506 |
| 0 | -3.912869 | 3.204159 | -4.261565 | -0.001884 | 199 | 0.002506 |
| 0 | 3.435550 | -1.851255 | 3.616793 | 0.011378 | 199 | 0.002506 |
| 0 | 4.462190 | -2.186947 | 3.606486 | 0.004014 | 199 | 0.002506 |
| 0 | 0.439962 | -4.627775 | -4.259147 | -0.009933 | 199 | 0.002506 |
| 0 | -3.337469 | -3.057935 | -4.248163 | -0.001001 | 199 | 0.002506 |
| 0 | -2.938167 | 4.816117 | -4.265235 | -0.006779 | 199 | 0.002506 |
| 0 | 2.611318 | 0.886971 | 3.610473 | 0.012998 | 199 | 0.002506 |
| 0 | -3.533064 | -2.947078 | -4.265567 | 0.006355 | 199 | 0.002506 |

Now the steplength has become too small, in fact it takes a larger number of iterations to converge, than before.
This is a confirm that A=1, B=150 should be a good configuration in this case.

---

As we can see from the above results, the algorithm is only capable to find local minima. In this case there are basically two local minima in the function: $(-4.25, 0)$ and $(3.6, 0)$.
This behaviour is specific of any steepest descent algorithm, which implements a greedy stratety of finding the best local solution, without guarantee a global optimal solution.
For this reason, this algorithm should be runned few times using different starting points, keeping the best result obtained.
To check which is the global minimum we have to estimate `f_noise` through simulation one last time in the two local minima:

[487]:
```
# (-4.25, 0)
estimate(f_noise, -4.25, 0, 10000)
```

[487]: -310.2588083548404

[488]:
```
# (3.6, 0)
estimate(f_noise, 3.6, 0, 10000)
```

[488]: -146.85704411854837

Hence, the global minimum is at $(-4.25, 0)$, as expected knowing the shape of the surface previously plotted.

---

On Gosavi textbook is also presented the classic approach of computing the gradient using finite differences. As said before, this should be avoided in large scale problems due to the large number of costly function evaluations. I decided to compare the previous results obtained with the simultaneous perturbation with the ones obtained with finite differences.

[485]:
```
results_findiff = pd.DataFrame()

n_points = 10

x = np.random.uniform(-6,5,n_points)
y = np.random.uniform(-5,5,n_points)

# min mu: the smallest step before stopping
mu_min = 0.005

for i in range(n_points):
    # The structure is the same, I call another function:
    x_min, y_min, n_iter, last_mu =␣
 ↪stochastic_sd_fin_diff(x[i],y[i],f_noise,mu_min,maxiter=10)

    results_findiff = results_findiff.append([[x[i], y[i], x_min, y_min,␣
 ↪n_iter, last_mu]])

results_findiff.columns =␣
 ↪['x_start','y_start','x_min','y_min','n_iterations','last_mu']
results_findiff
```

[485]:

| | x_start | y_start | x_min | y_min | n_iterations | last_mu |
|---|---|---|---|---|---|---|
| 0 | -1.489483 | -4.684198 | 9.305586e+21 | 3297.188102 | 9 | 0.091743 |
| 0 | 4.594153 | 0.734503 | 3.421229e+21 | -532.793201 | 9 | 0.091743 |
| 0 | -5.859058 | -2.866105 | -6.127733e+34 | 1561.605372 | 9 | 0.091743 |
| 0 | -2.145978 | 2.668139 | 4.610800e+24 | -1886.257837 | 9 | 0.091743 |
| 0 | -2.270238 | -4.382721 | 9.429405e+24 | 3087.661911 | 9 | 0.091743 |
| 0 | -0.724759 | -1.202363 | -1.039072e+39 | 95.683827 | 9 | 0.091743 |
| 0 | -4.251343 | -3.315163 | 2.469478e+14 | 12671.717106 | 9 | 0.091743 |
| 0 | -0.198065 | -4.471411 | 9.889791e+24 | -27416.755577 | 9 | 0.091743 |
| 0 | -5.714717 | -0.073507 | -1.259296e+33 | 4.817422 | 9 | 0.091743 |
| 0 | -4.670182 | 4.660743 | -1.243833e+20 | 28568.324085 | 9 | 0.091743 |

[486]:

```
results_findiff = pd.DataFrame()

n_points = 10

x = np.random.uniform(-6,5,n_points)
y = np.random.uniform(-5,5,n_points)

# min mu: the smallest step before stopping
mu_min = 0.005

for i in range(n_points):
    x_min, y_min, n_iter, last_mu = stochastic_sd_fin_diff(x[i],y[i],f_noise,mu_min,A=1,B=150,maxiter=200)
    results_findiff = results_findiff.append([[x[i], y[i], x_min, y_min, n_iter, last_mu]])

results_findiff.columns = ['x_start','y_start','x_min','y_min','n_iterations','last_mu']
results_findiff
```

[486]:

| | x_start | y_start | x_min | y_min | n_iterations | last_mu |
|---|---|---|---|---|---|---|
| 0 | -4.775026 | 3.934956 | -4.252289 | 0.000075 | 51 | 0.004975 |
| 0 | 2.870311 | -4.659549 | 3.609608 | -0.005511 | 51 | 0.004975 |
| 0 | 2.910304 | -0.778146 | 3.615538 | 0.004520 | 51 | 0.004975 |
| 0 | 0.319405 | 2.512547 | 3.623564 | -0.009114 | 51 | 0.004975 |
| 0 | -2.742841 | -3.170928 | -4.252260 | -0.002781 | 51 | 0.004975 |
| 0 | -1.476911 | -4.614053 | -4.257686 | -0.005159 | 51 | 0.004975 |
| 0 | 3.388527 | -0.869814 | 3.620165 | -0.005673 | 51 | 0.004975 |
| 0 | -1.196740 | 2.442601 | -4.261121 | 0.007228 | 51 | 0.004975 |
| 0 | 2.140026 | -4.820674 | 3.614850 | -0.001173 | 51 | 0.004975 |
| 0 | -0.743595 | -1.997896 | -4.248420 | 0.009108 | 51 | 0.004975 |

The outcomes of the finite differences approach is the same as simultaneous perturbations, which is a confirm of the correctness of the methods.

## 2.1 Further comments

This script is written using jupyter notebook, based on python 3.
The source code can be visualized and get also here: https://github.com/matbun/Num-opt/tree/master/p2