convex_optimization

October 22, 2018

1 Continuous Optimisation and Approximative Optimisation

```
In [31]: %matplotlib inline
         import matplotlib.pyplot as plt
         import seaborn as sns; sns.set()
         import numpy as np
         from main import *
```

1.1 1.1 Optimisation without constraints

1.1.1 Gradient methods Given the implementation of the gradient descent algorithm with a constant stepsize, we try to study its convergence with different step sizes

```
In [32]: rhos = [0.1, 0.05, 0.01, 0.001]
         for rho in rhos :
             results = gradient_rho_constant(f1,df1,x0,rho=rho,tol=1e-6,args=(B,S))
             print('for '+ str(rho) +', did the algorithm converge? ', results['converged'])
```

```
/home/otmane/3A/centrale/opt/tp/main.py:57: RuntimeWarning: invalid value encountered in subtrac
  xnp1=xn-rho*dfx # nouveau point courant (x_{n+1})


for 0.1, did the algorithm converge?  False
for 0.05, did the algorithm converge?  False
for 0.01, did the algorithm converge?  True
for 0.001, did the algorithm converge?  True
```

We can see that for high values of rho, the algorithm does not converge, it even results on invalid values for the update rule (xnp1 = inf). So we either reduce the stepsize parameter or define an adaptative algorithm that can handle this issue.

We define an adaptative version of the gradient descend algorithm

```
In [33]: def gradient_rho_adaptatif(fun, fun_der, U0, rho, tol,args):

             # Fonction permettant de minimiser la fonction f(U) par rapport au vecteur U
             # Méthode : gradient à pas fixe
             # INPUTS :
```

```
    # - han_f   : handle vers la fonction à minimiser
    # - han_df  : handle vers le gradient de la fonction à minimiser
    # - U0      : vecteur initial
    # - rho     : paramètre gérant l'amplitude des déplacement
    # - tol     : tolérance pour définir le critère d'arrêt
    # OUTPUT :
    # - GradResults : structure décrivant la solution


    itermax=10000  # nombre maximal d'itérations
    xn=U0
    f=fun(xn,*args) # point initial de l'algorithme
    it=0          # compteur pour les itérations
    converged = False;

    while (~converged & (it < itermax)):
        it=it+1
        dfx=fun_der(xn,*args)# valeur courante de la fonction à minimiser

        xnp1=xn-rho*dfx
        fnp1 = fun(xnp1,*args)

        if fnp1 < f :

            if abs(fnp1-f)<tol:
                converged = True

            xn, f = xnp1, fnp1
            rho *= 2
        else :
            rho /= 2

    GradResults = {
            'initial_x':U0,
            'minimum':xnp1,
            'f_minimum':fnp1,
            'iterations':it,
            'converged':converged
            }
    return GradResults
```

First, let's see if both versions give us the same minimum or not

```
In [34]: res_gd_const = gradient_rho_constant(f1,df1,x0,rho=0.01,tol=1e-6,args=(B,S))
         res_gd_adaptative = gradient_rho_adaptatif(f1,df1,x0,rho=0.01,tol=1e-6,args=(B,S))
         print('the minimum value we reach with a constant stepsize:', res_gd_const['f_minimum']
         print('the minimum value we reach with an adaptative stepsize:', res_gd_adaptative['f_m
```

```
the minimum value we reach with a constant stepsize: -1.8368912354010196
```

```
the minimum value we reach with an adaptative stepsize: -1.817778210582762
```

We can see that for the same stepsize, the simple version reaches a better minima.
Let's see if the adaptative version can deal with the convergence issue discussed above.

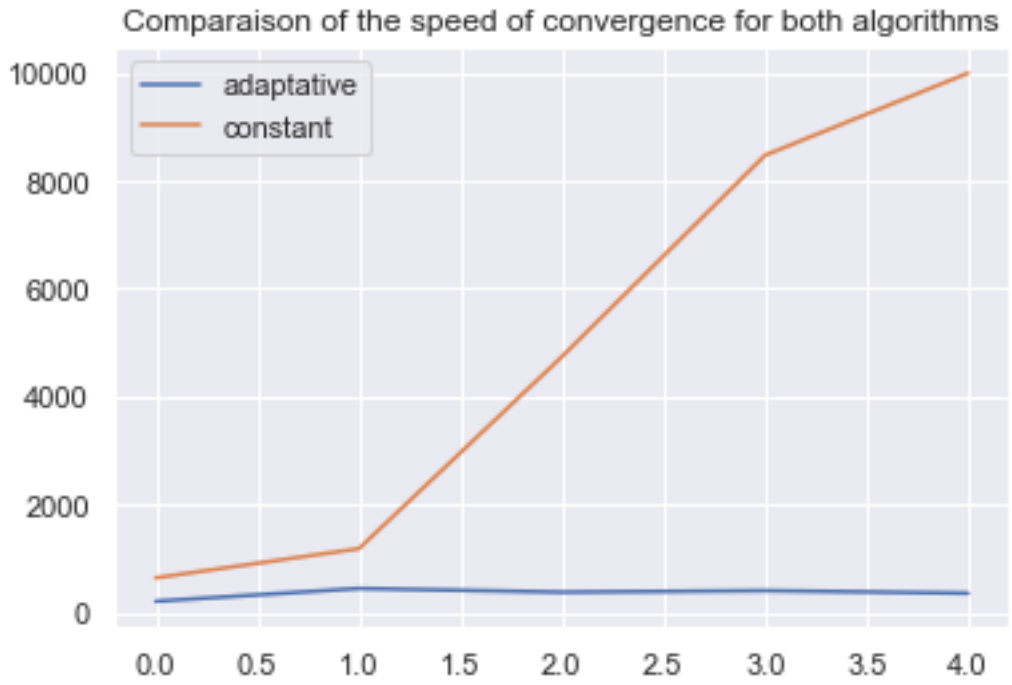
```
In [35]: rhos = [1, 0.1, 0.05, 0.01, 0.001]
         for rho in rhos :
             results = gradient_rho_adaptatif(f1,df1,x0,rho=rho,tol=1e-6,args=(B,S))
             print('for '+ str(rho) +', did the algorithme converge? ', results['converged'])
```

```
for 1, did the algorithme converge?  True
for 0.1, did the algorithme converge?  True
for 0.05, did the algorithme converge?  True
for 0.01, did the algorithme converge?  True
for 0.001, did the algorithme converge?  True
```

Indeed, it can converge for any stepsize we choose.

Another great benefit of using the adaptative version is the **speed of convergence**. We can further compare both algorithms by ploting the number of iterations required for the convergence of both versions depending of the stepsize we pick. For instance, we take values for which both algorithms can converge.

```
In [36]: rhos = [0.01, 0.005, 0.001, 0.0005, 0.0001]
         adaptative = [gradient_rho_adaptatif(f1,df1,x0,rho=rho,tol=1e-6,args=(B,S))['iterations
         constant = [gradient_rho_constant(f1,df1,x0,rho=rho,tol=1e-6,args=(B,S))['iterations']

         plt.plot(adaptative, label= 'adaptative')
         plt.plot(constant, label = 'constant')
         plt.title('Comparaison of the speed of convergence for both algorithms')
         plt.legend()
         plt.show()
```



The two algorithms make one call of the objective function in an iteration, and perform basic arithmetic operations inside the iterations loop. The complexity of such algorithms is **O(n)** with n the number of iterations. All the comparaisons then boil down to one criterion which is the convergence speed.

As we know, the speed of convergence of the simple gradient descent is highly affected by the value of the stepsize we choose. The adaptative version however have a constant convergence speed and does not depend on the value of the stepsize.

So if we want to conclude, we can say that in terms of convergence and speed, the adaptative version outperforms the constant one, but the simple gradient descent can give better minimas if we use a suitable stepsize.

1.1.2 Quasi-Newton methods Let's compare now the performance of the BFGS method with the algorithms we implemented above.

```
In [37]: from scipy.optimize import minimize

In [38]: bfgs_results = minimize(fun = f1, x0 = x0, args=(B,S), method='BFGS', tol=1e-6)
         bfgs_results

Out[38]:       fun: -1.836962311965238
          hess_inv: array([[ 0.5004328 , -0.21223272,  0.05691395, -0.28056012,  0.50919827],
                 [-0.21223272,  0.22383377,  0.0370393 ,  0.11542843, -0.34043246],
                 [ 0.05691395,  0.0370393 ,  0.1247386 , -0.04905887, -0.09108766],
                 [-0.28056012,  0.11542843, -0.04905887,  0.20504998, -0.28961765],
                 [ 0.50919827, -0.34043246, -0.09108766, -0.28961765,  0.77691404]])
               jac: array([-2.98023224e-08, -8.94069672e-08,  1.49011612e-08, -1.49011612e-08,
```

```
      1.49011612e-08])
 message: 'Optimization terminated successfully.'
    nfev: 91
     nit: 10
    njev: 13
  status: 0
 success: True
       x: array([-0.69603139,  0.15793134, -0.61407083,  0.49414155, -0.05345803])
```

The speed of the BFGS method is phenomenal, it did converge after **10** iterations and a gave us a slighly better minima than our gradient descent implementations.

Closed form of the minima The objective function we have is a simple quadratic function, so an analytical solution can be easily found. It's Hessian matrix is **S** which is positive definite.

```
In [39]: np.linalg.eigvals(S)
```

```
Out[39]: array([45.48200751, 14.49627115,  0.34399006,  1.89569112,  7.21954015])
```

The eigenvalues of S are strictly positive, so our function is convex and an analytical solution of the minimum can be found by settinh the first derivative to zero.

```
In [40]: S_1 = np.linalg.inv(S)
         minimum = 0.5*S_1*B
         f_minimum = f1(minimum, B, S)
         print('the analytical minimum is :', f_minimum)
```

```
the analytical minimum is : -1.8369623119652505
```

To confirm the performances of the quasi-newton methods, we can check the relative absolute difference between the analytical minimum and the result of the BFGS method.

```
In [41]: np.abs((f_minimum - bfgs_results['fun'])/f_minimum)
```

```
Out[41]: 6.7690544301362755e-15
```

The solution of the BFGS method is hugely precise.

1.2 1.2 Optimisation under constraints

1.2.1 Scipy method usage For the following results, we're gonna try to find the minimum of our functions in $ U = [0, 1]^5$

we define the f2 function :

```
In [42]: def f2(U,S):
             n=U.shape[0]
             U=np.matrix(U)
             U.shape=(n,1)
             fU = np.transpose(U) * S * U + np.transpose(U) * np.exp(U);
             return float(fU)
```

We use the **Sequential Quadratic Programming** method to minimise both functions f_1 and f_2.

```
In [43]: sqp_f1 = minimize(fun = f1, x0 = x0, args=(B,S), method='SLSQP', bounds=[(0,1)]*5, tol=
         sqp_f2 = minimize(fun = f2, x0 = x0, args=(S), method='SLSQP', bounds=[(0,1)]*5, tol=1e

In [44]: print('the argmins and minimums for both function are: \n')
         print('f1: argmin =', np.round(sqp_f1['x'], 5), 'f_min =', np.round(sqp_f1['fun'], 5),
         print('f2: argmin =', np.round(sqp_f2['x'], 5), 'f_min =', np.round(sqp_f2['fun'], 5),
```

```
the argmins and minimums for both function are:

f1: argmin = [0.      0.12698 0.      0.01945 0.     ] f_min = -0.13853

f2: argmin = [0. 0. 0. 0. 0.] f_min = 0.0
```

Both solutions respect the boundaries (constraints).

1.2.2 Optimisation under constraints and penalisation We define a simple and classical penalisation function that is C^∞

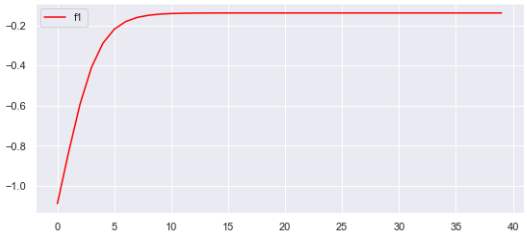
```
In [45]: Beta = lambda u : np.sum(np.maximum(u-1, 0)**2 + (np.maximum(-u,0))**2)
         print('Penalisation for x0:', Beta(x0))
         print('Penalisation for x0 + 2:', Beta(x0 + 1))
         print('Penalisation for x0 - 1:', Beta(x0 - 2))
```

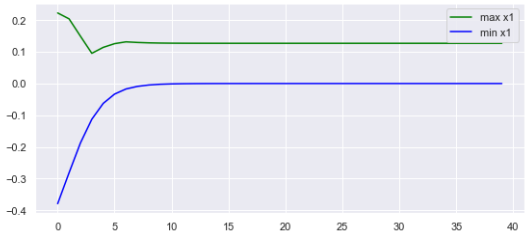
```
Penalisation for x0: 0.0
Penalisation for x0 + 2: 5.0
Penalisation for x0 - 1: 5.0
```

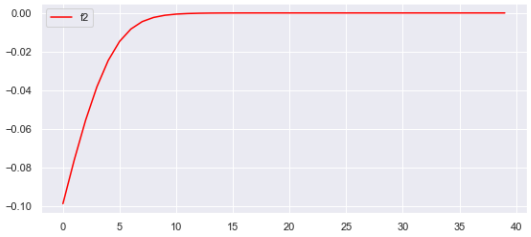
We define the penalisation method as follows :

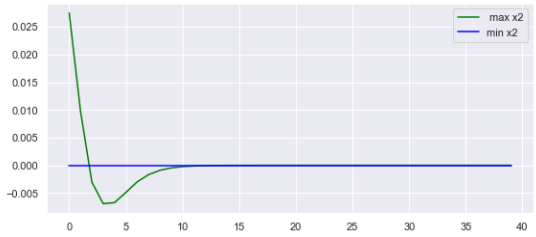
```
In [46]: epsilon = 1/2#The start values for epsilon
         decay = 2 #The decay parameter for epsilon in each iteration
         num_iter = 40 #number of iterations
         x_1, x_2 = x0, x0 # the starting point for both functions

         f1_penal_values = np.zeros(num_iter)
         f2_penal_values = np.zeros(num_iter)
         x_1_values =np.zeros(num_iter)
         x_2_values = np.zeros(num_iter)
         x_1_values_min = np.zeros(num_iter)
         x_2_values_min = np.zeros(num_iter)


         for k in range(num_iter):
             f1_penal, f2_penal = lambda U : f1(U,B,S) + (1/epsilon)*Beta(U), lambda U : f2(U,S)
```

```
        minimum_1 = minimize(fun = f1_penal, x0 = x_1, method='BFGS', tol=1e-6)
        minimum_2 = minimize(fun = f2_penal, x0 = x_2, method='BFGS', tol=1e-6)
        x_1 = minimum_1['x']
        x_2 = minimum_2['x']
        f1_penal_values[k] = minimum_1['fun']
        f2_penal_values[k] = minimum_2['fun']
        x_1_values[k] = max(minimum_1['x'])
        x_1_values_min[k] = min(minimum_1['x'])
        x_2_values[k] = max(minimum_2['x'])
        x_2_values[k] = max(minimum_2['x'])
        epsilon /= decay

plt.figure(figsize=(20,4))
plt.subplot(1,2,1)
plt.plot(np.arange(num_iter), f1_penal_values,c='red',label='f1')
plt.legend()
plt.subplot(1,2,2)
plt.plot(np.arange(num_iter), x_1_values,c='green',label='max x1')
plt.plot(np.arange(num_iter), x_1_values_min,c='blue',label='min x1')
plt.legend()
plt.show()

plt.figure(figsize=(20,4))
plt.subplot(1,2,1)
plt.plot(np.arange(num_iter), f2_penal_values,c='red',label='f2')
plt.legend()
plt.subplot(1,2,2)
plt.plot(np.arange(num_iter), x_2_values,c='green',label=' max x2')
plt.plot(np.arange(num_iter), x_2_values_min,c='blue',label='min x2')
plt.legend()
plt.show()
```









We can see that our function is getting bigger while we are trying to minimize it , and that's because we're trying to get our x back to the right boundaries so f has to follow that change.

```
In [47]: print('the argmins and minimums for both function are: \n')
         print('f1: argmin =', np.round(x_1, 5), 'f_min =', np.round(f1(x_1,B,S), 5), '\n')
         print('f2: argmin =', np.round(x_2, 5), 'f_min =', np.round(f2(x_2,S), 5), '\n')
```

```
the argmins and minimums for both function are:

f1: argmin = [-0.       0.12689 -0.       0.01941  0.     ] f_min = -0.13853

f2: argmin = [-0. -0. -0.  0. -0.] f_min = 0.0
```

We can see that both methods give close results.

1.2.3 Dual methods for optimisation under constraints For the Uzawa algorithm, we need to define the Lagragian function and it's derivative. We know that our constraints are affine functions so the dual gap is equal to zero. We can surely obtain the true minimum by applying this method.

```
In [48]: # Lagrangian implementation

         def Lagragian1(U,p):
             low_bound, up_bound = -U, U-1
             return f1(U,B,S) + np.sum(p*np.concatenate([low_bound, up_bound], axis=0))


         def dLagrangian1(U,p):
             low_bound, up_bound = -U, U-1
             return np.concatenate([low_bound, up_bound], axis=0)
```

The Uzawa algorithm

```
In [49]: def uzawa(Lagrangian, dL, p0, x0, rho, iterations):
             p_max, x_min = p0, x0
             for i in range(iterations) :
                 x_min = minimize(fun = Lagrangian, x0 = x_min, args= (p_max), method='BFGS', to
                 p_max = np.maximum(0 , p_max + (rho)*dL(x_min,p_max))
             return {'x_min': x_min, 'p_max': p_max, 'l_min': Lagragian1(x_min, p_max)}
```

```
In [50]: p0 = np.zeros(10)
         results = uzawa(Lagrangian=Lagragian1, dL=dLagrangian1 , p0=p0 , x0=x0, rho=1, iteratio
         x_min, l_min = results['x_min'], results['l_min']

In [51]: print('Lagrangian: argmaxmin =', np.round(x_min, 5), 'l_min =', np.round(l_min, 5), '\n

Lagrangian: argmaxmin = [-0.       0.12689 -0.       0.01941 -0.     ] l_min = -0.13853
```

We can see that our claims are confirmed and there is no duality gap. the solution for the dual problem matches the first solutions we got

1.3 1.3 Non Convex Optimisation : Simulated Annealing

In this section, we're gonna tackle a non convex problem, and try the simulated annealing algorithm to deal with it.

We define f_3 wich is clearly a non convex function.

```
In [52]: def f3(U) :
             return f1(U,B,S) + 10*np.sin(2*f1(U,B,S))
```

Let's try a convex optimisation algorithm on this function with different starting points.

```
In [54]: rng = np.random.RandomState(1)
         for i in range(10) :
             x_init = rng.uniform(-1,1,5)
             print("for try {}/10 we get f_min: {}".format(i+1,
                         np.round(minimize(fun = f3, x0 = x_init, method='BFGS', tol=1e-6)['fun'

for try 1/10 we get f_min: -10.7979
for try 2/10 we get f_min: -1.37312
for try 3/10 we get f_min: -1.37312
for try 4/10 we get f_min: 14.33484
for try 5/10 we get f_min: 4.91006
for try 6/10 we get f_min: 26.90121
for try 7/10 we get f_min: 55.17554
for try 8/10 we get f_min: -1.37312
for try 9/10 we get f_min: 36.32599
for try 10/10 we get f_min: 11.19325
```

Indeed, the algorithm converges to different local minimas but we can't confirm that the minimum of them all isn't a global minima. This can say that in the ***general case***, the algorithm converges to a local minima.

Let's try the simulated annealing which is more suitable to non convex problems.

In scipy, the simulated annealing function is deprecated and they propose a better algorithm under the name of basinhopping

```
In [55]: from scipy.optimize import basinhopping
```

```
In [56]: rng = np.random.RandomState(1)
         for i in range(5) :
             x_init = rng.uniform(-1,1,5)
             results = basinhopping(f3, x0 )
             print("for try {}/5 we get f_min : {}".format(i+1,np.round(results['fun'],5)),
                   'argmin :', np.round(results['x'],5))
```

```
for try 1/5 we get f_min : -10.7979 argmin : [-0.98526 -0.07006 -0.55429  0.5312  -0.28515]
for try 2/5 we get f_min : -10.7979 argmin : [-0.71549 -0.24957 -0.84531  0.45352  0.23547]
for try 3/5 we get f_min : -10.7979 argmin : [-1.41242  0.09979 -0.83104  0.90962 -0.44485]
for try 4/5 we get f_min : -10.7979 argmin : [ 0.05673 -0.18656 -0.29297  0.13251  0.44697]
for try 5/5 we get f_min : -10.7979 argmin : [-0.07317 -0.04652 -0.34827  0.32006  0.24914]
```

We can see that we always converge to the same minima, which is the ***best minima*** we got so far, but not the the same argmin due to the fact that our function is undulating. that means that the algorithm explores well the space of interest and ends up giving us the best minima we got (probably **the global minima**).

1.4 1.4 Application to signal processing

We define H_0 the ideal frequency response to be :

$$H_0(\nu) = 1 \quad for \quad \nu \in [0, 0.1] \quad , \quad 0 \qquad for \quad \nu \in [0.15, 0.5]$$

and try to approximate it with an even function H defined as follow:

$$H(\nu) = \sum_{i=0}^{n} h[i] \cos(2\pi\nu i)$$

To do so, we try a discretisation of both frequencies intervals $\{\nu_j\}_{1\leq j\leq p}$ and define a criterion to minimize over all $h[i]$:

$$J(h) = \max_j \left| H_0(\nu_j) - H(\nu_j) \right|$$

```
In [57]: H0 = np.vectorize(lambda v : 1 if (v >= 0 and v<= 0.1) else 0 if (v >= 0.15 and v<= 0.5

         #h must be a (30,) dimension array
         H = np.vectorize(lambda h,v : np.sum(h*np.cos(2*np.pi*v*np.arange(30))))
         H.excluded.add(0)
```

The intervals have a length ratio of **3:8**, so for the sake of simplicity, we're gonna take 65 points for both intervals (15 for the first and 40 for the second).

```
In [58]: v = np.concatenate((np.linspace(0, 0.1, 15),np.linspace(0.15, 0.5, 40)))
```

```
In [59]: #the criterion to minimize
         J = lambda h : np.max(np.abs(H0(v) - H(h,v)))
```

```
In [60]: rng = np.random.RandomState(1)
         for i in range(3) :
             x0 = rng.normal(size = 30)
             results = minimize(J, x0, method='BFGS', tol=1e-6)
             print("for try {}/10 we get f_min: {}".format(i+1,results['fun']))

for try 1/10 we get f_min: 5.404121228018497
for try 2/10 we get f_min: 4.265804970594914
for try 3/10 we get f_min: 1.1470591461797353
```

```
In [61]: rng = np.random.RandomState(1)
         for i in range(3) :
             x0 = rng.normal(size = 30)
             results = minimize(J, x0, method='Nelder-Mead', tol=1e-6)
             print("for try {}/3 we get f_min: {}".format(i+1,results['fun']))

for try 1/3 we get f_min: 5.501589932460395
for try 2/3 we get f_min: 4.8510053676540785
for try 3/3 we get f_min: 3.6155587046744317
```

We can see that the algorithm can't find a global minima, it gets stuck in local minimas. it takes more time then on the other functions as this one isn't regular enough.

Reformulation

We can rewrite the problem as a linear one with constraints.

$$\min_{h} \quad \max_{j} \left| H_0(\nu_j) - H(\nu_j) \right|$$

which then becomes :

$$\min_{(h,z)} z$$

Such that

$$\left| H_0(\nu_j) - H(\nu_j) \right| \leq z$$

which can be rewritten as a linear constraint :

$$H_0(\nu_j) - H(\nu_j) \leq z$$

$$H(\nu_j) - H_0(\nu_j) \leq z$$

that gives us the classical linear constraints formulation :

$$\min_{(h,z)} f(h,z)$$

$$A[h,z] \leq 0$$

with f and A linear functions