ADAPTATIVE INTERIOR-POINT METHODS

Michail Dadopoulos

LOG-BARRIER METHOD

Newton step for modified KKT equations

In the barrier method, the Newton step Δx_{nt}, and associated dual variable are given by the linear equations

$$\begin{bmatrix} t\nabla^2 f_0(x) + \nabla^2 \phi(x) & A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} \Delta x_{\mathrm{nt}} \\ \nu_{\mathrm{nt}} \end{bmatrix} = - \begin{bmatrix} t\nabla f_0(x) + \nabla \phi(x) \\ 0 \end{bmatrix}. \qquad (11.14)$$

In this section we show how these Newton steps for the centering problem can be interpreted as Newton steps for directly solving the modified KKT equations

$$\begin{array}{rcll} \nabla f_0(x) + \sum_{i=1}^m \lambda_i \nabla f_i(x) + A^T \nu & = & 0 & \\ -\lambda_i f_i(x) & = & 1/t, & i = 1, \ldots, m \\ Ax & = & b & \end{array} \qquad (11.15)$$

in a particular way.

DEFINING THE PROBLEM

```
c = np.array([2, 3])  # Coefficients in the objective function
A = np.array([[1, 1], [-1, 1], [-1, 0], [0, -1]])# Coefficients in the constraints
b = np.array([5, 3, 0, 0])# RHS values of the constraints
# Define the objective function, its gradient and hessian
def f(x):
    return c @ x
def grad_f(x):
    return c.T
def hess_f(X):
    return np.zeros((c.shape[0],c.shape[0]))

#-A@[t,t] or -A@[1,1]*t
#linear trajectory
def d(t):
    #direction = np.array([1, 1, -0.5, -0.5]) #to move all faces of polyhedron equally
    #return direction * t  # Linear trajectory
    return A@[t,t]

# The inequality constraints are defined as g(x) <= 0
def g(x,t):
    return (A @ x.T).T - (b+d(t))
# Define the log barrier function and its gradient and hessian
def phi(x,k,t):
    return k * f(x) - np.sum(np.log(-g(x,t)))

def grad_phi(x,k,t):
    return k * grad_f(x) + A.T @ (1. / (-g(x,t)))

#def hessian_phi(x,k,t):
#    return k * hess_f(x) - np.sum(A.T @ np.diag((1. / g(x,t)**2)) @ A.T, axis=0) #ME to T stin parenthesi xoris diafora toso
def hessian_phi(x,k,t):
    n = x.shape[0]
    hessian =np.zeros((n,n))
    for a, val in zip(A, g(x,t)):
        hessian += -(1.0 /val**2) *np.outer(a, a)
    return k * hess_f(x)+hessian
```

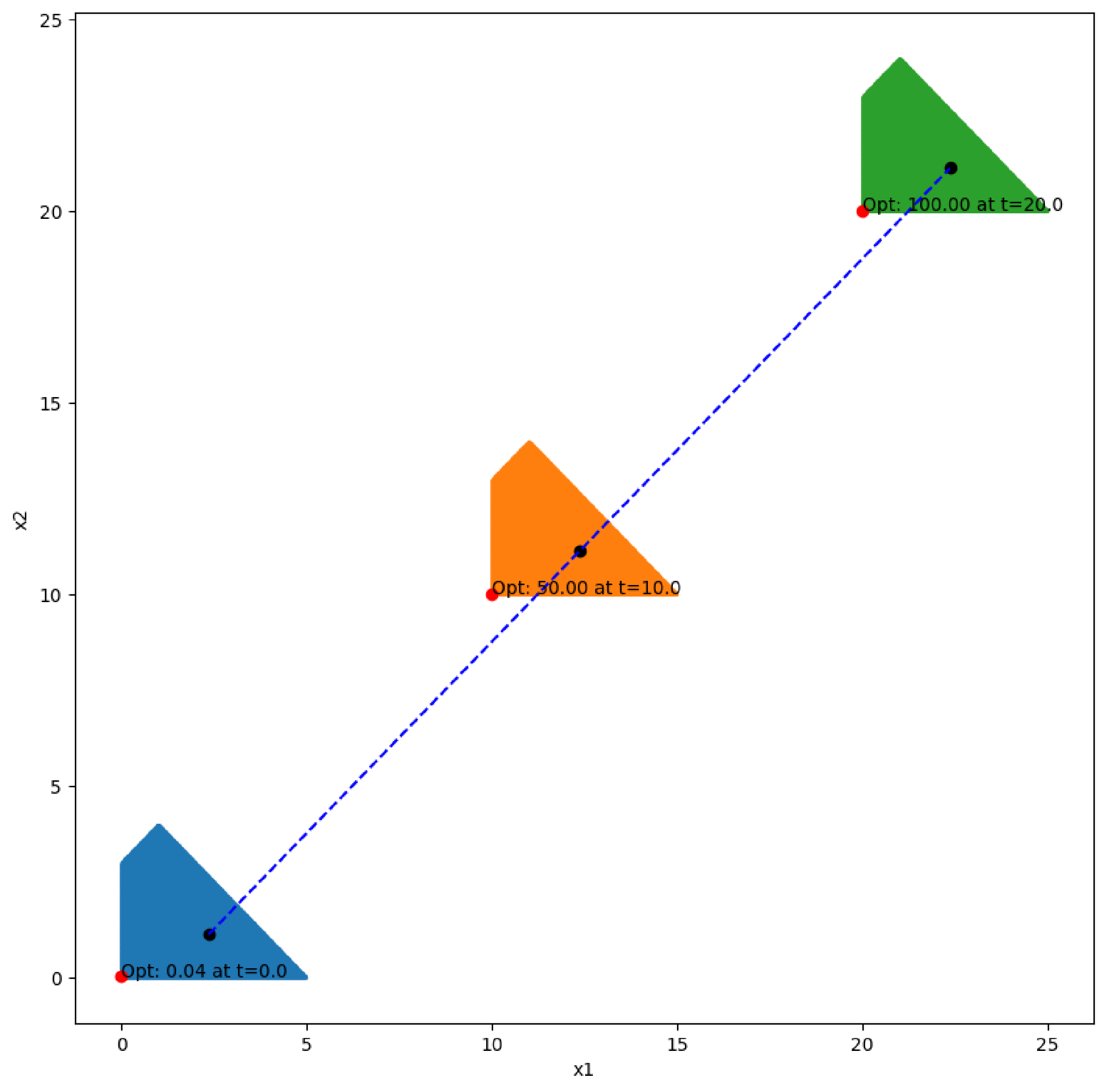
```python
def newton_method(x0, t, tol=1e-5, max_iter=100):
    x = x0
    k=0.2
    for _ in range(max_iter):
        grad = grad_phi(x,k,t)
        hessian = hessian_phi(x,k,t)
        #dx = -np.linalg.solve(hessian, grad) ,problem if not inverse
        dx = -np.linalg.lstsq(hessian, grad, rcond=None)[0]
        #dx = backtracking(x, dx, t, k)
        if np.linalg.norm(dx) < tol or np.linalg.norm(hessian)<tol or np.linal
            break
        x += dx
        #t = update_barrier_parameter(x, t)
    return x
```

Run the adaptive algorithm and visualize results

```python
T=20
x0=np.array([0.5,0.5])
x=np.copy(x0)
history = []
#x0 = newton_method(x0, 0.1)
# Run the adaptive interior-point method and visualize the results
for t in np.arange(0, T+0.1 ,0.1):
    # Run the Newton method and update the history of optimal solutions
    x = newton_method(np.copy(x), t)
    history.append(x)
history = np.array(history)
# Widgets to interact with the plot
interact(plot_trajectories,
         t1=FloatSlider(min=0, max=T, step=0.1, value=0),
         t2=FloatSlider(min=0, max=T, step=0.1, value=T/2),
         t3=FloatSlider(min=0, max=T, step=0.1, value=T));

# Plot the solution over time
history = np.array(history)
plt.figure()
plt.plot(history[:, 0], history[:, 1])
plt.title('Solution over time')
plt.xlabel('Variable 1 (x1)')
plt.ylabel('Variable 2 (x2)')
plt.show()
```

RESULTS



Opt: 100.00 at t=20.0
Opt: 50.00 at t=10.0
Opt: 0.04 at t=0.0
x1
x2
0
5
10
15
20
25

PRIMAL DUAL METHOD

$$y = (x, \lambda, \nu), \qquad \Delta y = (\Delta x, \Delta\lambda, \Delta\nu),$$

respectively. The Newton step is characterized by the linear equations

$$r_t(y + \Delta y) \approx r_t(y) + Dr_t(y)\Delta y = 0,$$

i.e., $\Delta y = -Dr_t(y)^{-1} r_t(y)$. In terms of x, λ, and ν, we have

$$\begin{bmatrix} \nabla^2 f_0(x) + \sum_{i=1}^m \lambda_i \nabla^2 f_i(x) & Df(x)^T & A^T \\ -\,\mathbf{diag}(\lambda) Df(x) & -\,\mathbf{diag}(f(x)) & 0 \\ A & 0 & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta\lambda \\ \Delta\nu \end{bmatrix} = - \begin{bmatrix} r_{\text{dual}} \\ r_{\text{cent}} \\ r_{\text{pri}} \end{bmatrix}. \tag{11.54}$$

The *primal-dual search direction* $\Delta y_{\text{pd}} = (\Delta x_{\text{pd}}, \Delta\lambda_{\text{pd}}, \Delta\nu_{\text{pd}})$ is defined as the solution of (11.54).

```
def primal_dual(x0, λ0, v0, t,μ=10,alpha=0.01,beta=0.5, tol=1e-8, max_iter=10):
    x, v, λ = x0, v0, λ0
    m=A.shape[0] #number of inequality constraints
    n=D.shape[0]#number of equality constraints
    N=A.shape[1]#number of variables
    for _ in range(max_iter):

        #surrogate duality gap
        h=-g(x,t)*λ
        k=μ*m/h

        # Compute the residuals
        rdual = grad_f(x) + grad_g(x).T @ λ + D.T@v
        rcent = -np.diag(λ) @g(x,t).T -(1/k)
        rprimal=D @ x -E
        residuals=np.concatenate([rdual,rcent,rprimal])

        # Formulate the Newton system
        M1 = np.block([[hess_f(x), grad_g(x).T, D.T], [-np.diag(λ)@grad_g(x), -np.diag(g(x,t)), np.zeros((m, n))], [D, np.zeros(
        M2 = residuals
        sol = np.linalg.lstsq(M1, M2, rcond=None)[0]

        # Extract dx, dy, dλ from the solution
        dx = sol[:N]
        dλ = sol[N:N+m]
        dv = sol[N+m:]
        # Perform line search(backtracking) and update x, λ, v
        #smax = np.minimum(np.amin([-λ[i]/dλ[i] for i in range(m) if dλ[i] < 0]), 1)
        #smax=np.amin(1, np.amin(-λ[dλ < 0] / dλ[dλ < 0]))
        smax=1
        for i in range(m):
            if dλ[i]<0:
                if smax>-λ[i]/dλ[i]:
                    smax=-λ[i]/dλ[i]
        s = 0.99 * smax
        while True:
            x_new = x + s * dx
            λ_new = λ + s * dλ
            v_new = v + s * dv
            rdual = grad_f(x_new) + grad_g(x_new).T @ λ_new + D.T@v_new
            rcent = -np.diag(λ_new) @g(x_new,t).T -(1/k)
            rprimal=D @ x_new -E
            new_residuals=np.concatenate([rdual,rcent,rprimal])
            if np.linalg.norm(new_residuals)<=(1-alpha*s)*np.linalg.norm(residuals):#or np.min(x_new) > 0 and np.min(s_new) > 0 
                break
            s *=beta
        # Update primal and dual variables
        x, λ, v = x_new, λ_new, v_new

        # Check the convergence
        if np.linalg.norm(rdual)<tol or np.linalg.norm(rprimal)<tol or np.linalg.norm(h)<tol or np.linalg.norm(dx) < tol or np.l
            break
```