Optimización

Primer Cuatrimestre 2025

Práctica de Laboratorio N° 2

Ejercicio 1 Implemente una función `optim`, que tome como entradas la función f, un punto inicial x^0 y el tamaño de paso (fijo) α, para encontrar el mínimo de la función usando descenso por el gradiente. La función debe retornar el valor encontrado, la cantidad de iteraciones y un gráfico donde se muestren conjuntamente los puntos que recorre el algoritmo junto con un gráfico de f. Para esto último puede utilizar la función `contourf`. Testear con la función $f(x_1, x_2) = \sin(x_1 + x_2) + \cos(x_1)^2$, tomando $x^0 = (0, -1)$ y pasos $\alpha = 0.1, 0.01, 1$. ¿Qué observa en cada caso?

Ejercicio 2 Implementar un algoritmo que reciba como entrada una función ϕ (que se asume unimodal), un intervalo $[a, b]$, y una tolerancia δ y calcule el mínimo de ϕ en $[a, b]$ con error menor o igual que δ, mediante el algortimo de búsqueda por la razón dorada.

Ejercicio 3 Dada $\phi : \mathbb{R} \to \mathbb{R}$, implementar funciones que realicen una búsqueda inexacta del valor de t que minimiza $\phi(t)$ para $t > 0$, según:

(a) La regla de Armijo.

(b) La regla de Goldstein.

(c) El criterio de Wolfe.

Los parámetros deben tener un valor predefinido, pero el usuario debe tener la opción de elegirlos. En el caso del criterio Wolfe, implementar dos métodos para la función: uno en el que el usuario sólo provee ϕ y la derivada se estima por diferencias finitas, y uno en el que usuario ingresa también ϕ'.

Ejercicio 4 Modificar la función implementada en el ejercicio 3, donde el el paso ahora es determinado por una busqueda lineal en cada iteración. El usuario debe tener la posibilidad de ingresar a mano el gradiente y que tipo de regla usar para definir el paso. En caso de que esto no ocurra, el programa debe calcular el gradiente usando el paquete `ForwardDiff.jl` y un paso α fijo.

Testear el algoritmo con las funciones:

(a) ***Rosenbrock:*** $f(x, y) = 100(y - x^2)^2 + (x - 1)^2$, cuyo mínimo es $(x, y) = (1, 1)$.

(b) ***Wood:*** $f(\mathbf{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2 + 90(x_4 - x_3^2)^2 + (1 - x_3)^2 + 10(x_2 + x_4 - 2)^2 + 0.1(x_2 - x_4)^2$, cuyo mínimo se encuentra en $x = (1, 1, 1, 1)$.

(c) ***Freudenstein y Roth:*** $f(\mathbf{x}) = (-13 + x_1 + ((5 - x_2)x_2 - 2)x_2)^2 + (-29 + x_1 + ((x_2 + 1)x_2 - 14)x_2)^2$, cuyo mínimo está en $(5, 4)$, pero que también tiene un mínimo local en $(11.41\ldots, -0.89\ldots)$.

Ejercicio 5 Implementar el algoritmo de Newton puro. El usuario debe tener la posibilidad de ingresar a mano el gradiente y la matriz hessiana. Testear el algoritmo con las funciones del Ejercicio 4.

Ejercicio 6 Un defecto del método del gradiente es que su convergencia puede resultar muy lenta, incluso para funciones cuadráticas. Esto se debe a que el gradiente se ve afectado por problemas de *escala*, que podría resolverse mediante un re-escale de las variables. Una heurística para lograr esto es tomar como dirección de descenso $-\mathbf{D}_k\mathbf{g}_k$, donde $\mathbf{D}_k$ es una matriz diagonal que aproxima la diagonal de $\mathbf{H}_k^{-1}$. Implementar este algoritmo.

Para las funciones del Ejercicio 4, comparar el número de iteraciones y el tiempo de ejecución (mediante el macro `@time`), de los métodos del gradiente, de Newton y el de gradiente re-escalado.

Ejercicio 7 El método de Levenberg-Marquardt consiste en tomar como dirección de descenso $-(\mathbf{H}_k + \mu_k\mathbf{I})^{-1}\mathbf{g}_k$, donde μ_k se elige en cada paso de manera tal que si $\mathbf{H}_k$ es definida positiva, $\mu_k = 0$, y en caso contrario, μ_k es tal que $\mathbf{H}_k + \mu_k\mathbf{I}$ resulte definida positiva. En la práctica, esto equivale a un método intermedio entre Newton ($\mu_k = 0$) y el gradiente (μ_k grande). La elección de μ_k puede resultar compleja, dado que uno querría que $\mu_k \sim -\lambda$, donde λ es el autovalor negativo de $\mathbf{H}_k$ de máximo módulo, y el cálculo de autovalores es un problema complicado.

Sin embargo, en este hilo se discute la razonabilidad de ese enfoque y se argumenta en favor de un método que consiste esencialmente en lo siguiente. Si $\mathbf{H}_k = \mathbf{U}^t\mathbf{D}\mathbf{U}$, con $\mathbf{D}$ diagonal, se calcula $\mathbf{H}_k^\ddagger = \mathbf{U}^t\mathbf{D}^\ddagger\mathbf{U}$, de modo que $\mathbf{H}^\ddagger$ opere como una versión aceptable de $\mathbf{H}_k^{-1}$. Para ello, se toma $\mathbf{D}^\ddagger$ diagonal tal que:

- Si d_{ii} es grande (por ejemplo, $|d_ii| > \delta \max|\mathbf{D}|$ para algún δ), entonces $d_{ii}^\ddagger = |d_ii|^{-1}$.
- Si d_{ii} es pequeño (no cumple la condición anterior), entonces $d_{ii}^\ddagger = \delta \max|\mathbf{D}|$.

Discutir las ideas detrás de esta propuesta. Suponiendo que la matriz $\mathbf{H}_k$ tiene autovalores negativos de módulo grande, ¿Qué ocurre en cada método con ellos? ¿Qué ocurre con los autovalores de módulo chico? ¿Cómo afecta esto la dirección de búsqueda?

El paquete `PositiveFactorizations` implementa estas ideas.

Por ejemplo: el comando `cholesky(Positive,H)` recibe una matriz simétrica $\mathbf{H}$ (no necesariamente definida positiva) y construye una descomposición de Cholesky de $\mathbf{H}^\ddagger$ (en el proceso de realizar la factorización, se determina cómo deben modificarse la matriz).

Implementar el algoritmo de Levenberg-Marquardt tomando μ_k como el mínimo autovalor de $\mathbf{H}_k$ + un valor ε_0.

Implementar también un algoritmo tipo Newton pero que en lugar de invertir $\mathbf{H}_k$, calcule $\mathbf{H}^\ddagger$ usando `PositiveFactorizations.cholesky`.

Aplicar ambos métodos a la función de Beale:

$$f(x,y) = (1.5 - x + xy)^2 + (2.25 - x + xy^2)^2 + (2.625 - x + xy^3)^2,$$

cuyo mínimo se encuentra en $(3, 0.5)$ y a la función de Ackley:

$$f(x,y) = -20e^{-0.2\sqrt{0.5(x^2+y^2)}} - e^{0.5(\cos 2\pi x + \cos 2\pi y)} + e + 20,$$

cuyo mínimo absoluto está en el origen. En cada caso, graficar la sucesión generada por cada método.

Ejercicio 8 Ajuste algebraico de círculos. Dados n puntos en el plano $(x_1, y_1), \ldots, (x_n, y_n)$ provenientes de mediciones que se sabe deberían corresponder a un círculo, se desea hallar el círculo:

$$C : x^2 + y^2 - 2\alpha x - 2\beta y - \gamma = 0,$$

que mejor ajusta los datos. El enfoque algebraico propone realizar el ajuste sobre la *ecuación* del círculo. Es decir, plantea buscar los valores de los parámetros α, β, γ de manera que se minimice el funcional:

$$F(\alpha, \beta, \gamma) = \sum_{i=1}^{n} \left(x_i^2 + y_i^2 - 2\alpha x_i - 2\beta y_i - \gamma \right)^2 = \sum_{i=1}^{n} \left(2\alpha x_i + 2\beta y_i + \gamma - r_i^2 \right)^2,$$

donde en la última expresión tomamos $r_i^2 = x_i^2 + y_i^2$.

(a) Mostrar que la expresión para C (elegida por conveniencia) es general y puede describir cualquier círculo.

(b) Dar las ecuaciones del centro y del radio de C en función α, β y γ.

(c) Observar que el ajuste resulta lineal en los parámetros y por lo tanto se trata de un problema de cuadrados mínimos estándar.

Ejercicio 9 Ajuste geométrico de círculos. En el mismo contexto del ejercicio anterior, se quiere realizar un ajuste *geométrico*, de modo que el círculo minimice la suma de las distancias a los puntos (x_i, y_i). Para esto conviene expresar el círculo como:

$$C : (x - a)^2 + (y - b)^2 = r^2.$$

(a) Dar una expresión para la distancia del punto (x_i, y_i) al círculo C.

(b) Escribir el funcional a minimizar, dado por la suma de los cuadrados de las distancias a C. Observar que la minimización de este funcional constituye un problema de cuadrados mínimos no lineal.

(c) Implemente un programa que genere datos aleatorios sobre un círculo, perturbados con un error alteatorio. Implemente el método de ajuste lineal del ejercicio anterior, y compare los resultados con una estimación no-lineal obtenida mediante la librería JuMP.jl.

Ejercicio 10 (Compresión de imagenes usando SVD) El problema de comprimir una imagen lo podemos pensar como un problema de cuadrados mínimos: dado un conjunto de datos (los valores de los píxeles de una imagen), buscamos una manera de aproximar los datos lo mejor posible, en algún sentido, almacenando una cantidad dada de información. Dada una matriz $A \in \mathbb{R}^{m \times n}$ y su descomposición en valores singulares con vectores singulares $\{u_j\}$ y $\{v_j\}$ a izquierda y derecha respectivamente y valores singulares $\{\sigma_j\}$, la matriz:

$$B_r = \sum_{j=1}^{r} \sigma_j u_j v_j^t,$$

es la que mejor aproxima a A en norma 2, entre todas las matrices de rango r. Observar que para almacenar la matriz B_r no hace falta guardar $m \times n$ casilleros, sino sólo $r \times m$ para los vectores u, más $r \times n$ para los vectores v, más r para los valores singulares.

Tomemos como ejemplo la siguiente imagen. Las siguientes operacions serán útiles:

Código en Julia

```
using LinearAlgebra,Plots, Images #importamos los paquetes necesarios

img = load("vermeer.jpg") #para guardar la imagen en una variable img
im_gris = Gray.(img) #para transformar la imagen a escala de grises
```

(a) Trabajemos con la imagen en escala de grises. Realizar la descomposición en valores singulares de la imagen y graficar los valores de σ. Utilizar los comandos `svd` y `scatter`.

(b) A partir del gráfico pensar un criterio razonable para la elección de un número r de valores singulares a conservar en la imagen comprimida. Recomponer la imagen usando sólo esos valores singulares. Tener en cuenta que la matriz recompuesta es una matriz de números que es necesario convertir a formato `Gray`.

(c) Como nos interesa generar una imagen comprimida, creamos un nuevo tipo de variable, al que llamaremos ImageSVD:

Código en Julia

```
struct ImageSVD
    u::Matrix{Float64}
    v::Matrix{Float64}
    s::Vector{Float64}
    m::Int64
    n::Int64
end
```

La idea de esta estructura es que: $u \in \mathbb{R}^{m\times r}$ y $v \in \mathbb{R}^{n\times r}$ contienen los primeros r vectores singulares de cada lado y $\sigma \in \mathbb{R}^r$ tiene los valores singulares almacenados. $m \times n$ es el tamaño de la imagen original. ImageSVD no es realmente una imagen, sino que se corresponde a la imagen comprimida. Dado un objeto de tipo ImageSVD, la imagen puede reconstruirse completando las matrices u y v y el vector σ para que tengan el mismo tamaño que los originales (usando n y m) y realizando las multiplicaciones para reconstruir la matriz.

Si ya tenemos disponibles los elementos que van a formar parte de la imagen comprimida (u,v,σ, n y m), podemos crear una variable de tipo ImageSVD, del siguiente modo:

```
im_comp = ImageSVD(u,v,s,m,n)
```

(d) Implementar una función 'comprimir' que reciba una imagen en escala de grises y un número r y genere un objeto de tipo ImageSVD con la información correspondiente a la imagen comprimida. Probar la función. Para acceder a las distintas componentes de la imagen comprimida se utiliza el nombre de la variable, punto, el nombre del campo. Por ejemplo:

```
U = im_comp.u
```

(e) Implementar una función descomprimir que tome como input una variable de tipo ImageSVD, reconstruya la matriz y devuelva una imagen en escala de grises. Testear la función.

Ejercicio 11 Implementar los ejercicios que tienen un **(*)** de la Práctica 1.