INSTITUTO TECNOLÓGICO AUTÓNOMO DE MÉXICO

Optimización Numérica | Proyecto 1 – Reporte

Juan Carlos Sigler   Alonso Martinez   Paulina Carretero

Índice general

1 Marco teórico

En el presente documento discutimos una implementación del algoritmo del conjunto activo para resolver problemas de programación cuadrática con restricciones tanto de igualdad como desigualdad, como en el problema (P) a continuación:

$$\begin{array}{ll} \min & \frac{1}{2}\vec{x}^\top G\vec{x} + \vec{c}^\top \vec{x} \\ \text{sujeto a} & Ax_i = b_i \quad i \in \mathcal{E} \\ & Ax_j \leq b_j \quad j \in \mathcal{I} \end{array} \tag{P}$$

Nos servimos de las librerías `JuMP` [DHL17], y `MAT` del ecosistema Julia para aplicar el método Simplex y para leer archivos en formato `.mat` respectivamente. Adicionalmente llevamos a cabo *benchmarks* con el paquete `BenchmarkTools` para evaluar el desempeño de nuestro algoritmo y pruebas unitarias para hacer el proceso de desarrollo más fácil.

1.1 Algunos comentarios y aclaraciones

Puesto que no suponemos familiaridad con el lenguaje Julia queremos dar comentarios para hacer más sencilla su interacción con él en caso de que quiera reproducir los resultados mediante los jupyter notebooks provistos.

- Para añadir el kernel de Julia a Jupyter necesita una instalación de Julia y el paquete `IJulia`. Para desarrollar el proyecto usamos la versión 1.6.3 de Julia.
- Otros paquetes que usamos son:
  - `JuMP`
  - `GLPK`
  - `MAT`
  - `BenchmarkTools` (no esencial para probar el algoritmo)

Para evitarle el problema de instalarlos, puede seguir estos pasos:

1. Abrir un Julia REPL escribiendo `julia` en bash.
2. Escribir `]` para que el prompt cambie de `julia>` a `pkg>`
3. Escribir `activate .` en el prompt `pkg>`

- Julia es compilado *just in time*, por lo que al iniciarse y la primera vez que se llama una función que no ha sido compilada hay un tiempo de espera considerable. Le aseguramos que el programa no falló y podrá ver que las llamadas siguientes son mucho más rápidas.
- Añadimos como apéndices a éste reporte la documentación de los métodos auxiliares usados en la implementación del algoritmo con la esperanza de hacerle amena la lectura del código fuente.
- Julia permite usar caracteres unicode como identificadores, que aprovechamos para hacer más legible nuestro código. Si tiene problemas con caracteres faltantes o *artifacts*, le pedimos paciencia y sugerimos usar un emulador de terminal o font diferente.

2 Problemas

Probamos nuestra implementación con tres problemas y presentamos los resultados de acuerdo a lo especificado en la asignación del proyecto.

2.1 Problema 1: Problema chico

$$\min q(x) = (x-1)^2 + (y-2.5)^2$$

Sujeto a

$$\begin{aligned} -x + 2y - 2 &\leq 0 \\ x + 2y - 6 &\leq 0 \\ x - 2y - 2 &\leq 0 \\ -x &\leq 0 \\ -y &\leq 0 \end{aligned}$$

Con $x_0 = (2,0)^\top$ & $W_0 = \{3\}$.

Para este problema, nuestro algoritmo imprime lo siguiente:

```
Rama 1. ||d_k|| = 1.8, q(x) = -4.05, α=1.667
Rama 2.

Rama 1. ||d_k|| = 1.6, q(x) = -6.45, α=0.5k = 1
Rama 2. j = 2, μ=0.0

Concluyó método del conjunto activo en 2 iteraciones
El punto de paro fue:
2-element Vector{Float64}:
 1.4
 1.7
```

Obtenemos que el punto Karush-Kuhn-Tucker (KKT) es:

$$(\vec{x}^\star, \vec{\mu}^\star) = \left(\begin{bmatrix} 1.4 \\ 1.7 \end{bmatrix}, \begin{bmatrix} 0.399 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \right),$$

y se alcanza en dos iteraciones. El valor de la función objetivo en el óptimo es -1.60.

2.2 Problema 2: Klee-Minty

$$\begin{array}{ll} \min & \frac{1}{2}\vec{x}^\top G\vec{x} - \sum_{i=1}^{n} x_i \\ \text{sujeto a} & x_1 \leq 1 \\ & 2\sum_{j=1}^{i-1} x_j + x_i \leq 2^i - 1 \quad , i = 2, \dots, n \\ x_1, \dots, x_n \geq 0 & \end{array}$$

Sea $n = 15$. Aplica el método empezando con W_0 un subconjunto aleatorio de 5 entradas de los últimos 10 restricciones de positividad.

Para encontrar W_0 que cumple las restricciones impuestas usamos el siguiente comando:

```julia
W_0 = [falses(length(b)-10); rand((false, true), 10)]
# Ahora hay que asegurar que son exactamente 5
Δ = sum(W_0) - 5
if Δ < 0
    # Faltan restricciones
    candidates = findall(W_0 .== false)
    # Eligiendo solo los que están entre las 10 restricciones de positividad
    candidates = candidates[candidates .>= length(b) - 10]
    selected = rand(candidates, abs(Δ))

    W_0[selected] .= trues(abs(Δ))
elseif Δ > 0
    # Hay que quitar
    candidates = findall(W_0 .== true)
    selected = rand(candidates, abs(Δ))

    W_0[selected] .= falses(abs(Δ))
end
```

La selección es aleatoria, entonces la W_0 que presentamos está sujeta a cambios, pero la presentamos con fines ilustrativos de cualquier manera.

```
julia>
Bool[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1]
```

Para este problema, nuestro algoritmo imprime lo siguiente:

```
julia> or = activeSetMethod(G, c, A, b, n_eq, W_0)
α = 0.0, j = 16

Rama 1. ||d_k|| = 10000.0, q(x) = 0.0, α=0.0k = 0
α = 0.0, j = 17

Rama 1. ||d_k|| = 10000.0, q(x) = 0.0, α=0.0k = 1
α = 0.0, j = 18

Rama 1. ||d_k|| = 10000.0, q(x) = 0.0, α=0.0k = 2
α = 0.0, j = 19

Rama 1. ||d_k|| = 10000.0, q(x) = 0.0, α=0.0k = 3
α = 0.0, j = 20

Rama 1. ||d_k|| = 10000.0, q(x) = 0.0, α=0.0k = 4
α = 0.0, j = 22

Rama 1. ||d_k|| = 10000.0, q(x) = 0.0, α=0.0k = 5
α = 0.0, j = 24

Rama 1. ||d_k|| = 10000.0, q(x) = 0.0, α=0.0k = 6
α = 0.0, j = 26

Rama 1. ||d_k|| = 10000.0, q(x) = 0.0, α=0.0k = 7
α = 0.0, j = 27

Rama 1. ||d_k|| = 10000.0, q(x) = 0.0, α=0.0k = 8
α = 0.0, j = 30

Rama 1. ||d_k|| = 10000.0, q(x) = 0.0, α=0.0k = 9
Rama 2. j = 1, μ=0.0

Concluyó método del conjunto activo en 10 iteraciones
El punto de paro fue:
15-element Vector{Float64}:
 0.0
 0.0
 ⋮
 0.0
```

El óptimo se obtiene en 10 iteraciones y el punto KKT es:

$$
(\vec{x}^{\star}, \vec{\mu}^{\star}) = \left(\begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 1 \end{bmatrix} \right)
$$

.

La representación completa del vector $\vec{\mu}$ es

```
julia> print(or.μ)
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.
    0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

.

El valor óptimo de la función objetivo en $\vec{x}_{\star}$ es 0.0.

Para este problema la rutina `quadprog` de MATLAB imprime:

```
>> quadprog(G,c,A,b)

Minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in
feasible directions, to within the value of the optimality tolerance,
and constraints are satisfied to within the value of the constraint tolerance.



ans =

   1.0e-10 *

    0.2220
    0.1175
    0.0685
    0.0744
    0.1014
```

Nuestra solución coincide con la de MATLAB salvo error de redondeo.

2.2.1 Comentarios

A primera vista parece que nuestra implementación se estanca, ya que en el *output* presentado se puede notar que $\|d_k\|$ siempre es relativamente grande (alrededor de mil) y que $\alpha = 0$ en todas las iteraciones. Lo cual indica que no se encuentra una dirección factible. Haciendo una inspección más cercana notamos que el conjunto W_k para cada iteración si estaba cambiando, y en cada iteración el algoritmo añadía un índice de manera secuencial. Por lo tanto, concluimos que no es un error de implementación, ya que el algoritmo si está explorando posibles direcciones factibles y añadiendo índices de restricciones a W_k como marca el procedimiento de Conjunto Activo, pero no logra encontrar una dirección de descenso. Lo cual tiene sentido porque notamos que el punto inicial x_0 coincide con el mínimo del problema. El punto inicial lo obtenemos de aplicar la rutina de optimización de problemas lineales de la librería GLPK

2.3 Problema 3: Problema Afiro

$$\begin{array}{ll} \text{minimizar} & \frac{1}{2}\vec{x}^\top G\vec{x} + \vec{c}^\top \vec{x} \\ \text{sujeto a} & A\vec{x} = \vec{b} \\ & x_i \geq \ell_i \quad \text{si } \ell_i \text{ es finito,} \\ & x_i \leq u_i \quad \text{si } u_i \text{ es finito.} \end{array}$$

- Definimos $J \subset I$ como sigue:
  - Para $x_j \geq \ell_j$ definimos $\left|g_j(x_0)\right| \leq 8\varepsilon_m \max\left\{\left|\ell_j\right|, 1\right\} \Longrightarrow j \in J$
  - Para $x_j \leq u_j$ definimos $\left|g_j(x_0)\right| \leq 8\varepsilon_m \max\left\{\left|u_j\right|, 1\right\} \Longrightarrow j \in J$

Donde ε_m es el épsilon de la máquina `eps(Float64)`.

A continuación presentamos el código que se utilizó para construir el problema y encontrar las restricciones que pertenecen a J. El código se presenta recortado (excluimos el código para obtener los datos del archivo .mat y comentarios aclaratorios) en interés de la brevedad, pero se puede encontrar código completo en el script incluido `script3.ipynb`.

```
n_eq = length(b)
A_eq = problem["A"]
G = I(length(c))

```

```
A = [A_eq; -I(length(l))]
b = [b; -l]

mask = isfinite.(u)
A = [A; I(length(u))[mask, :]]
# Igual cambiando b para que dimensiones coincidan
b = [b; u[mask]]
b = b[:]

x_0 = linprog(A, b, n_eq)εₘ

  = eps(Float64)

J_l = abs.(l - x_0) .<= 8 * εₘ *  max.(abs.(l), ones(length(l)))

J_u = abs.(x_0 - u) .<= 8 * εₘ *  max.(abs.(u), ones(length(u)))
J_u = J_u[isfinite.(u)]

J = [J_l; J_u]

# Concatenando con un índice aleatorio
R_index = rand(findall(J), 1)
notJ = falses(size(A, 1) - n_eq)
notJ[R_index] .= true
W_0 = [trues(n_eq); notJ]
```

De el cómputo anterior obtenemos W_0 que documentamos a continuación a manera de ejemplo puesto que dada la naturaleza aleatoria del índice, puede cambiar.

```
julia> println(W_0)
Bool[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
     0, 0, 1]
```

Corriendo el algoritmo con las especificaciones dadas obtenemos el output que se imprime a continuación:

```
julia> or = activeSetMethod(G, c, A, b, n_eq, copy(W_0))
α = -0.0, j = 29

Rama 1. ||d_k|| = 265.3, q(x) = 323100.0, α=-0.0k = 0
α = -0.0, j = 35

Rama 1. ||d_k|| = 265.4, q(x) = 323100.0, α=-0.0k = 1
α = -0.0, j = 41

Rama 1. ||d_k|| = 265.2, q(x) = 323100.0, α=-0.0k = 2
α = -0.0, j = 44

Rama 1. ||d_k|| = 265.2, q(x) = 323100.0, α=-0.0k = 3
α = -0.0, j = 68

Rama 1. ||d_k|| = 258.3, q(x) = 323100.0, α=-0.0k = 4
α = -0.0, j = 69

Rama 1. ||d_k|| = 256.1, q(x) = 323100.0, α=-0.0k = 5
α = -0.0, j = 70

Rama 1. ||d_k|| = 251.6, q(x) = 323100.0, α=-0.0k = 6
α = -0.0, j = 54

Rama 1. ||d_k|| = 240.2, q(x) = 323100.0, α=-0.0k = 7
α = -0.0, j = 71
```

```

Rama 1. ||d_k|| = 240.2, q(x) = 323100.0, α=-0.0k = 8
α = -0.0, j = 31

Rama 1. ||d_k|| = 235.5, q(x) = 323100.0, α=-0.0k = 9
α = -0.0, j = 55

Rama 1. ||d_k|| = 235.5, q(x) = 323100.0, α=-0.0k = 10
α = -0.0, j = 53

Rama 1. ||d_k|| = 235.8, q(x) = 323100.0, α=-0.0k = 11
α = 1.245, j = 28

Rama 1. ||d_k|| = 235.8, q(x) = 200800.0, α=1.245
Rama 2.
α = 24.36, j = 28

Rama 1. ||d_k|| = 1.928, q(x) = 200800.0, α=24.36
Rama 2. j = 1, μ=0.0

Concluyó método del conjunto activo en 14 iteraciones
El punto de paro fue:
51-element Vector{Float64}:
  15.09
   0.0
  33.95
  -7.994e-15
   ⋮
 113.5
  15.94
  60.88
  -1.137e-13
```

El punto óptimo completo es:

```
julia> print(round.(or.x_star; sigdigits=4))
[15.09, 0.0, 33.95, -7.994e-15, 2.336, 2.357, 264.5, -7.105e-14, 358.4, 2.153, 2.182, 2.211, 1.94, 1.928, 10.62, 13.14, 0.0, 139.9, 183.3, 64.91, 37.31, 27.6, 68.8, 46.05, 6.65, -7.172e-14, 7.816e-14, 1.066e-14, 6.65, 2.336, 2.357, 26.65, 26.05, 55.87, 235.5, 158.9, 32.68, 43.88, 101.3, 141.6, -5.684e-14, 0.0, -2.842e-14, -5.684e-14, 2.153, 2.182, 2.211, 113.5, 15.94, 60.88, -1.137e-13]
```

con un valor óptimo de 401820.555, que se obtuvo en 14 iteraciones.

3 Conclusiones

Gracias a que verificamos los resultados utilizando la rutina `quadprog` de Matlab y la librería JuMP, además de seguir un conjunto de pruebas unitarias, confiamos en que los resultados de nuestra implementación del algoritmo del conjunto activo es correcta y robusta. Adicionalmente, notamos que nuestra implementación es sumamente rápida.

Para investigar sobre el desempeño cuantitativamente utilizamos el paquete `BenchmarkTools`, y a continuación presentamos los resultados de un *benchmark*. Un *benchmark* consiste en un número definido de *samples*, de los cuales se mide el tiempo de ejecución excluyendo el tiempo que toma preparar los datos. Abajo presentamos los resultados de un *benchmark* aplicado a el problema 3, que es el el más grande del cual disponemos. Para el problema 3 la matriz A es una matriz sparse de 78×51 con 153 entradas distintas de cero.

```
BenchmarkTools.Trial: 630 samples with 1 evaluation.
 Range (min … max):  5.764 ms … 29.671 ms  | GC (min … max): 0.00% … 52.42%
 Time  (median):     7.294 ms              | GC (median):    0.00%
 Time  (mean ± σ):   7.912 ms ±  3.213 ms  | GC (mean ± σ):  5.72% ± 10.52%
```

```
Memory estimate: 2.34 MiB, allocs estimate: 8800.
```

Como podemos ver, en 630 *samples* nuestra implementación del algoritmo toma en promedio 7 ms, lo cual es más o menos de esperarse dado que resolver el problema toma menos de 20 iteraciones. Sin embargo, cuando probamos con problemas que tomaban alrededor de 77 (woodinfe) el promedio era alrededor de 50 ms, lo que calificamos de relativamente rápido. Además, asigna alrededor de 2.3 MiB de memoria en total, lo cual es sumamente razonable tomando en cuenta la dimensión del problema. Todo esto gracias a el uso de matrices *sparse*. En la figura 1 se puede ver un histograma de la frecuencia de tiempos de ejecución.

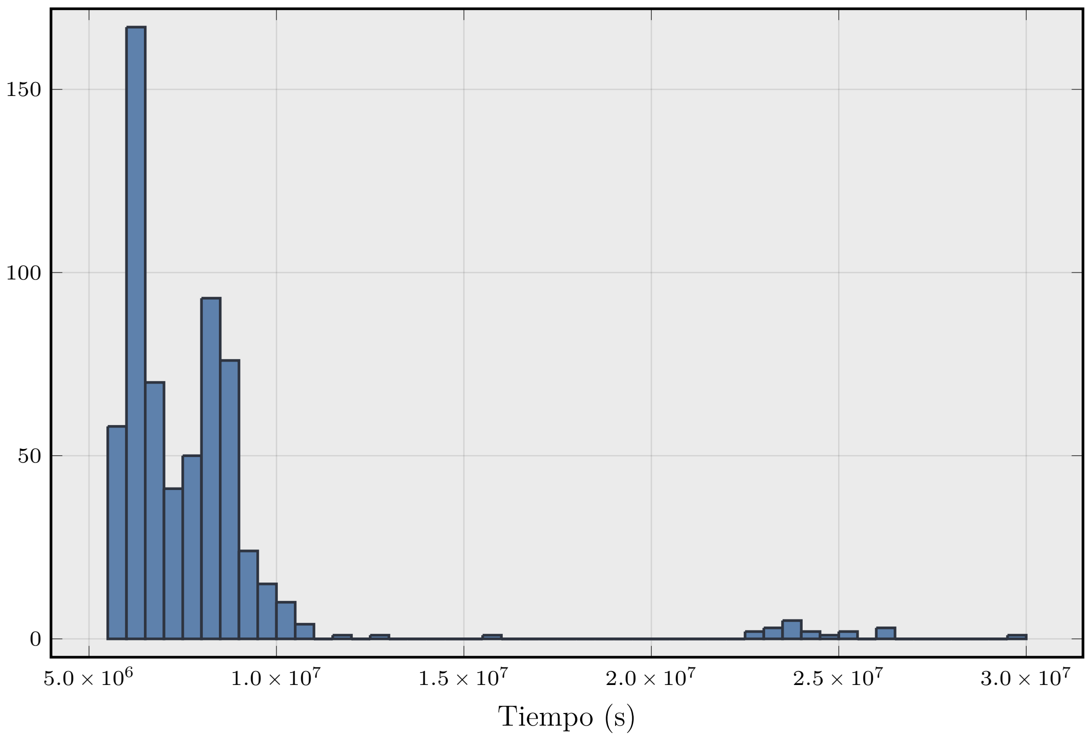

Figura 1: Histograma de frecuencia por tiempo

Referencias

[DHL17] Iain Dunning, Joey Huchette y Miles Lubin. "JuMP: A Modeling Language for Mathematical Optimization". En: *SIAM Review* 59.2 (2017), págs. 295-320. DOI: 10.1137/15M1020575.