Optimización Estocástica

Universidad de los Andes

Septiembre 2022

Sergio Arango 201921814

1 Introducción al Problema

Carlos dirige dos sucursales de una empresa nacional de alquiler de carros. Cada d´ıa llegan clientes a cada sucursal con distribuciones f1 y f2, respectivamente, a alquilar carros. Si al llegar un cliente, Carlos tiene disponible un carro para alquilar ´el recibe r por parte de la empresa. Similarmente, cada d´ıa llegan clientes a entregar los carros alquilados en su sucursal o en otras de la misma empresa con distribuciones g1 y g2 respectivamente para cada sucursal. Carlos solo puede usar los carros devueltos hasta el d´ıa siguiente de la entrega pues debe hacerles limpieza. Adicionalmente, Carlos puede trasladar carros de una sus sucursales a otra a un costo c por carro. Carlos quiere maximizar su ganancia.

1.1 Marco Teórico

El problema se plantea como un Problema de Decisión de Markov de horizonte infinito descontado.

Se define el conjunto de estados, el conjunto de acciones, junto con sus restricciones por estado, y las funciones de recompensa y transición.

Se define:

$$S = \{\#carrosSucursal1\} x \{\#carrosSucursal2\}$$

$$A = \{\#transfer : Suc1 \rightarrow Suc2\}$$

$$p(s'|s,a) = p(s'_1, s'_2|s_1, s_2, a) =$$
$$\sum_{c_1 \in PosCli} f_1(c_1) g_1(s'_1 - (s_1 - t - min(s_1 - t, c_1)))$$
$$* \sum_{c_2 \in PosCli} f_2(c_2) g_2(s'_2 - (s_2 + t - min(s_2 + t, c_2)))$$

$$r(s,a) = r(s_1, s_2, a) = \left(\sum_{c_1 \in PosCli} f_1(c_1) min(s_1 - t, c_1) + \sum_{c_2 \in PosCli} f_2(c_2) min(s_2 + t, c_2) \right) * r - |t| * c$$

Las restricciones para $A(s)$ son las siguientes:

$$\forall a \in A(s) = A(s_1, s_2) : -s_2 \leq t \leq s_1$$

Los estados son $s = (s_1, s_2) \in S$ la cantidad de carros disponibles en cada sucursal. Se asume que Carlos no decide si alquilar algún carro o no ya que es obvio que para maximizar su ganancia siempre que pueda lo hace, por eso la única acción que se plantea para el PDM es la cantidad de carros netos que transfiere de una sucursal a otra y la dirección. De la forma en que se define la acción a representa a carros transferidos de la Sucursal 1 a la Sucursal 2, para representar la misma transferencia en dirección opuesta se tiene la acción $-a$. Note que solo se tiene en cuenta la transferencia neta ya que como todo se decide en el mismo instante no tiene sentido para Carlos transferir carros en ambas direcciones ya que esto solo incrementaría su costo y no cambiaría su recompensa por venta. Por último el límite de carros que Carlos puede transferir en cada dirección está dado por cuantos carros hay en cada Sucursal de salida. Nota: PosCli se refiere al conjunto de posibles valores que pueden tomar las variables aleatorias de cantidad de clientes en cada sucursal. (Se asume el mismo conjunto para ambas que a priori puede ser N) En tanto las recompensas como las probabilidades de transición se entiende la cantidad de carros alquilados en un día en una sucursal es el mínimo entre la cantidad de clientes en la sucursal y la cantidad de carros contando la transferencia del día.

2 Detalles de implementación

2.1 Value Iteration

Primero que todo se modifica el problema para realizar calculos computacionales. Se define una cota para la posible cantidad de clientes en ambas sucursales. Esto es para acotar el calculo de las sumatorias para distintos valores de clientes. Además se define una cota para la posible cantidad de carros en una sucursal. Esto es para acotar la cantidad de estados y así limitar el tamaño de los vectores de valor V. Se muestra el código en la Figura 1.

Luego se define el algoritmo de iteración por valor de la siguiente manera.

Se inicia el Vector inicial en cero y para cada componente(estado) se calcula el valor de la mejor acción descontada. Este proceso se realiza hasta estar que dos vectores consecutivos estén en una vecindad $\epsilon \frac{(1-\lambda)}{2\lambda}$.

El cálculo de la acción que maximiza el estado se hace con el vector anterior v_{n-1}(en el código $oldV$).

```python
def value_iteration(S, A, P, R):
    k = epsilon*(1-lam)/(2*lam)
    V = {s: 0 for s in S}
    optimal_policy = {s: 0 for s in S}
    numero_it = []
    VectoresValor = []
    n = 0
    while True:
        oldV = V.copy()
        numero_it.append(n)
        VectoresValor.append(oldV)
        print(n)
        n = n + 1
        for s in S:
            Q = {}
            for a in A:
                Q[a]= R(s,a) + lam*sum(P(s_next,s,a) * oldV[s_next] for s_next in S)

            #print(s)
            V[s] = max(Q.values())
            optimal_policy[s] = max(Q, key=Q.get)


        if all(oldV[s] <= V[s] + k and oldV[s] >= V[s] - k  for s in S):
            break

    return numero_it, VectoresValor, optimal_policy
```

Figure 1: Value Iteration Python

2.2 Jacobi

Se realiza el mismo proceso excepto que dentro del cálculo de valor para todos los estados, no se tiene en cuenta el término del estado sobre el cuál se está buscando la acción óptima. Además todo el valor de cada acción a va dividido por

$$1 - \lambda P(s|s, a)$$

donde s es el estado para el cuál se está calculando la acción óptima. La implementación se puede ver en la Figura 2.

2.3 Gauss-Seidel

Se realiza el mismo proceso excepto que a medida que se itera sobre los componentes del vector actual (los estados), se usa el valor del vector actual para cada uno de los estados que ya se calcularon para el vector actual para los restantes se utiliza el vector anterior. Se muestra el código en la Figura 3.

3 Resultados Numéricos

Se elige un $\epsilon = 0.01$.
Se elige 5 como la misma cota de carros para ambas sucursales y para la cantidad

```python
def jacobi(S, A, P, R):
    print("Jacobi:")
    k = epsilon*(1-lam)/(2*lam)
    V = {s: 0 for s in S}
    optimal_policy = {s: 0 for s in S}
    numero_it = []
    VectoresValor = []
    n = 0
    while True:

        oldV = V.copy()
        numero_it.append(n)
        VectoresValor.append(oldV)
        print(n)
        n = n + 1
        for j in range(len(S)):

            Q = {}
            for a in A:

                Q[a]= (R(S[j],a) + lam*(sum(P(s_next,S[j],a) * oldV[s_next] for s_next in S[:j])
                +sum(P(s_next,S[j],a) * oldV[s_next] for s_next in S[j+1:])))/(1-lam*P(S[j],S[j],a))

            #print(s)
            V[S[j]] = max(Q.values())
            optimal_policy[S[j]] = max(Q, key=Q.get)


        if all(oldV[s] <= V[s] + k and oldV[s] >= V[s] - k  for s in S):
            break

    return numero_it, VectoresValor, optimal_policy
```

Figure 2: Jacobi Python

de clientes que pueden llegar en un día.

La política óptima resulta estacionaria definida por una sola regla de decisión. La regla de decisión óptima obtenida se muestra en la Figura 4. El resultado es el mismo independiente del algoritmo utilizado.

La convergencia de V_n está dada por la Figura 5. Se muestra para todos los algoritmos implementados.

3.1 λ-Análisis

Finalmente, usando Gauss-Seidel, se obtiene V_λ para varios λ y se calcula $(1 - \lambda V_\lambda)$. Los valores de λ que se usan son 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 y 0.95. El vector final de valor para cada λ se muestra a continuación junto con la cantidad de iteraciones que costó encontrarlo usando Gauss-Seidel para $\epsilon = 0.01$. Claramente el numero de iteraciones necesario incrementa con λ. Además los vectores de valor esperado no cambian mucho a partir de cierto λ, por lo que al multiplicar por $1 - \lambda$ este domina y la tendencia será a cero cuando λ tienda a 1.

```python
def gauss_Seidel(S, A, P, R):
    k = epsilon*(1-lam)/(2*lam)
    V = {s: 0 for s in S}
    optimal_policy = {s: 0 for s in S}
    numero_it = []
    VectoresValor = []
    n = 0
    while True:

        oldV = V.copy()
        numero_it.append(n)
        VectoresValor.append(oldV)
        print(n)
        n = n + 1
        for j in range(len(S)):

            Q = {}
            for a in A:

                Q[a]= R(S[j],a) + lam*(sum(P(s_next,S[j],a) * V[s_next] for s_next in S[:j])
                +sum(P(s_next,S[j],a) * oldV[s_next] for s_next in S[j:]))

            #print(s)
            V[S[j]] = max(Q.values())
            optimal_policy[S[j]] = max(Q, key=Q.get)


        if all(oldV[s] <= V[s] + k and oldV[s] >= V[s] - k  for s in S):
            break

    return numero_it, VectoresValor, optimal_policy
```

Figure 3: Gauss-Seidel Python

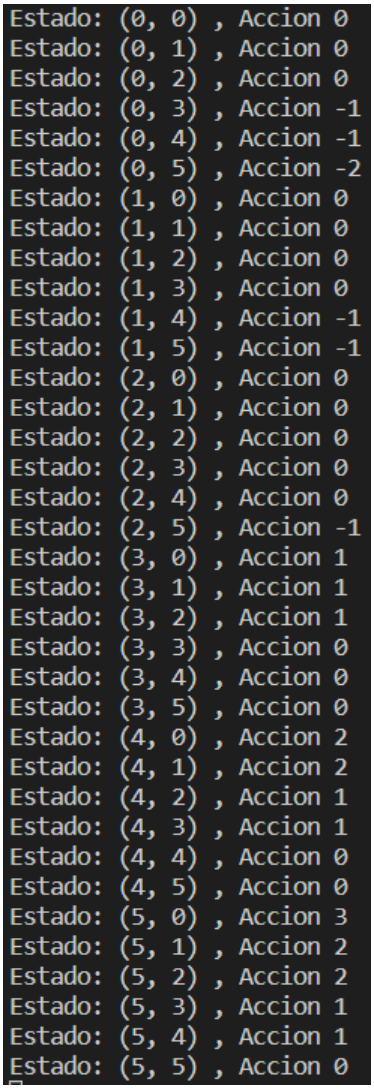

Figure 4: Optimal Policy

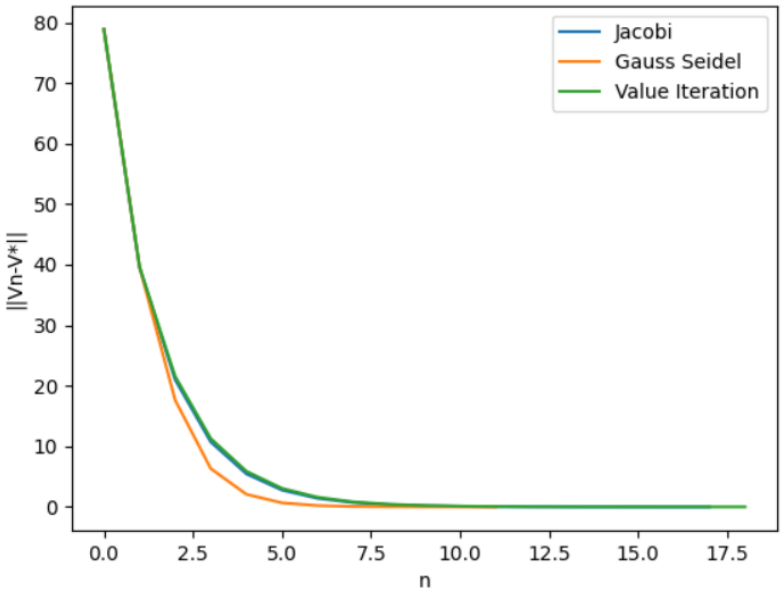

Figure 5: Convergencia con $\epsilon = 0.01$ de Gauss-Seidel vs Jacobi vs Value Iteration

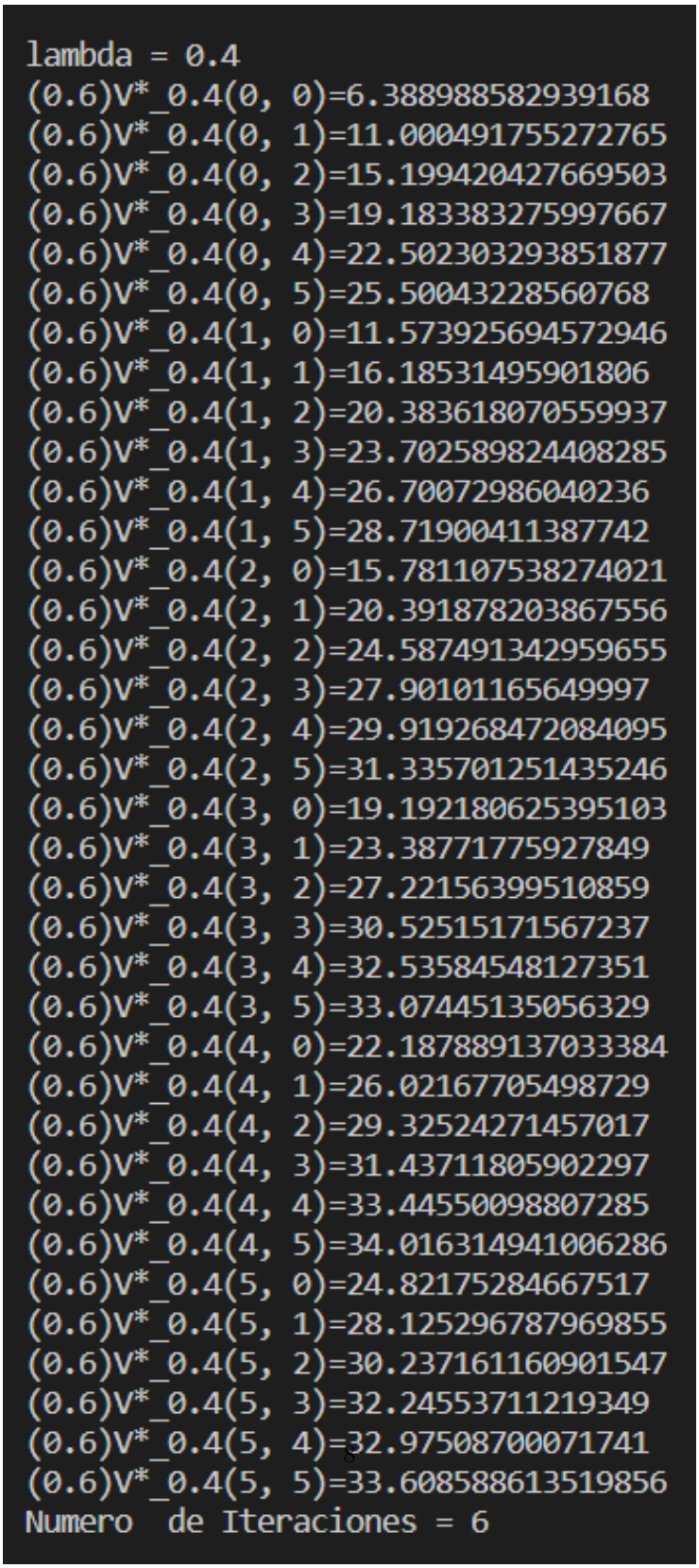

Figure 6: $(1 - \lambda V_\lambda)$ para $\lambda = 0.4$

```
lambda = 0.5
(0.5)V*_0.5(0, 0)=7.236211849983997
(0.5)V*_0.5(0, 1)=11.08048544238582
(0.5)V*_0.5(0, 2)=14.58247905888733
(0.5)V*_0.5(0, 3)=17.89528468323487
(0.5)V*_0.5(0, 4)=20.65314946360812
(0.5)V*_0.5(0, 5)=23.111282849797153
(0.5)V*_0.5(1, 0)=11.54994806352857
(0.5)V*_0.5(1, 1)=15.394076949090094
(0.5)V*_0.5(1, 2)=18.895380175490757
(0.5)V*_0.5(1, 3)=21.653266295810052
(0.5)V*_0.5(1, 4)=24.111404474680473
(0.5)V*_0.5(1, 5)=25.73958052654206
(0.5)V*_0.5(2, 0)=15.017029038656798
(0.5)V*_0.5(2, 1)=18.8605203275889
(0.5)V*_0.5(2, 2)=22.359073868469842
(0.5)V*_0.5(2, 3)=25.111523036245508
(0.5)V*_0.5(2, 4)=26.739692782845417
(0.5)V*_0.5(2, 5)=27.806527410338607
(0.5)V*_0.5(3, 0)=17.860649420370734
(0.5)V*_0.5(3, 1)=21.359172107031615
(0.5)V*_0.5(3, 2)=24.44140852458887
(0.5)V*_0.5(3, 3)=27.184263800407628
(0.5)V*_0.5(3, 4)=28.806593796490546
(0.5)V*_0.5(3, 5)=29.114413708581935
(0.5)V*_0.5(4, 0)=20.359251837996617
(0.5)V*_0.5(4, 1)=23.4414624004618
(0.5)V*_0.5(4, 2)=26.184308404365602
(0.5)V*_0.5(4, 3)=27.806632260676693
(0.5)V*_0.5(4, 4)=29.346797029156164
(0.5)V*_0.5(4, 5)=29.695912159950062
(0.5)V*_0.5(5, 0)=22.441501982977833
(0.5)V*_0.5(5, 1)=25.18433742295101
(0.5)V*_0.5(5, 2)=26.806654470628175
(0.5)V*_0.5(5, 3)=28.346817409952465
(0.5)V*_0.5(5, 4)=28.695927607302337
(0.5)V*_0.5(5, 5)=29.062329856545063
Numero  de Iteraciones = 7
```

Figure 7: $(1 - \lambda V_\lambda)$ para $\lambda = 0.5$

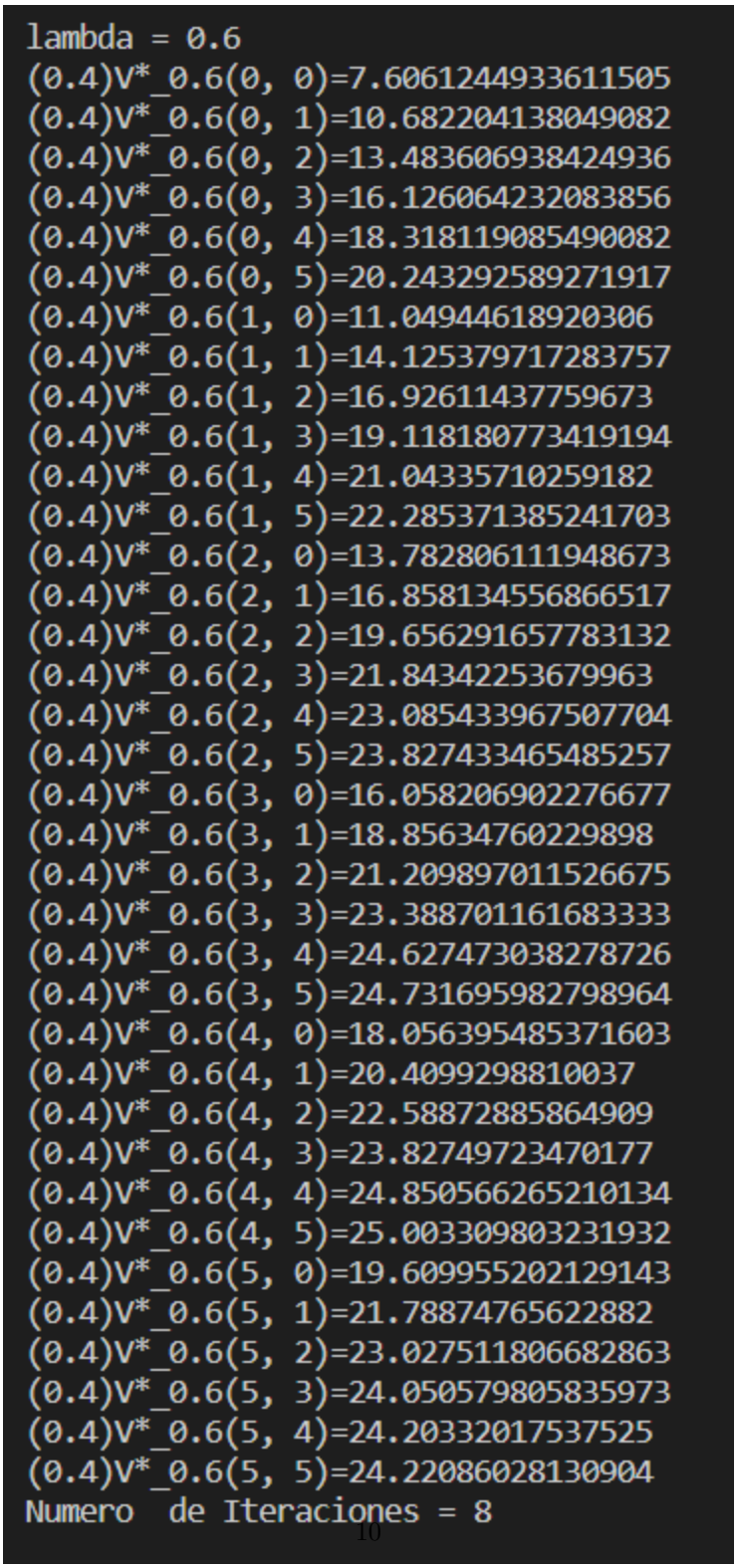

Figure 8: $(1 - \lambda V_\lambda)$ para $\lambda = 0.6$

```
lambda = 0.7
(0.30000000000000004)V*_0.7(0, 0)=7.3438879546754
(0.30000000000000004)V*_0.7(0, 1)=9.65099731768477
(0.30000000000000004)V*_0.7(0, 2)=11.749330820810219
(0.30000000000000004)V*_0.7(0, 3)=13.723264412187028
(0.30000000000000004)V*_0.7(0, 4)=15.34865268892651
(0.30000000000000004)V*_0.7(0, 5)=16.752486579017862
(0.30000000000000004)V*_0.7(1, 0)=9.918564682443087
(0.30000000000000004)V*_0.7(1, 1)=12.225544264772013
(0.30000000000000004)V*_0.7(1, 2)=14.323297263439255
(0.30000000000000004)V*_0.7(1, 3)=15.948693336126063
(0.30000000000000004)V*_0.7(1, 4)=17.352529289589086
(0.30000000000000004)V*_0.7(1, 5)=18.221284235120088
(0.30000000000000004)V*_0.7(2, 0)=11.929169150508171
(0.30000000000000004)V*_0.7(2, 1)=14.235626472752886
(0.30000000000000004)V*_0.7(2, 2)=16.33119100178581
(0.30000000000000004)V*_0.7(2, 3)=17.95257427670614
(0.30000000000000004)V*_0.7(2, 4)=18.821327652355503
(0.30000000000000004)V*_0.7(2, 5)=19.27388720291911
(0.30000000000000004)V*_0.7(3, 0)=13.635676894121847
(0.30000000000000004)V*_0.7(3, 1)=15.731230536962318
(0.30000000000000004)V*_0.7(3, 2)=17.390416056706556
(0.30000000000000004)V*_0.7(3, 3)=19.005465225393536
(0.30000000000000004)V*_0.7(3, 4)=19.873916241970445
(0.30000000000000004)V*_0.7(3, 5)=19.81714039026784
(0.30000000000000004)V*_0.7(4, 0)=15.131265893134568
(0.30000000000000004)V*_0.7(4, 1)=16.79044064058224
(0.30000000000000004)V*_0.7(4, 2)=18.405486245016384
(0.30000000000000004)V*_0.7(4, 3)=19.273934808313946
(0.30000000000000004)V*_0.7(4, 4)=19.85052902965922
(0.30000000000000004)V*_0.7(4, 5)=19.845613674943287
(0.30000000000000004)V*_0.7(5, 0)=16.19046035338978
(0.30000000000000004)V*_0.7(5, 1)=17.805501040160994
(0.30000000000000004)V*_0.7(5, 2)=18.673946408047843
(0.30000000000000004)V*_0.7(5, 3)=19.250539930656235
(0.30000000000000004)V*_0.7(5, 4)=19.245622106769638
(0.30000000000000004)V*_0.7(5, 5)=19.01309414950007
Numero  de Iteraciones = 9
```

Figure 9: $(1 - \lambda V_\lambda)$ para $\lambda = 0.7$

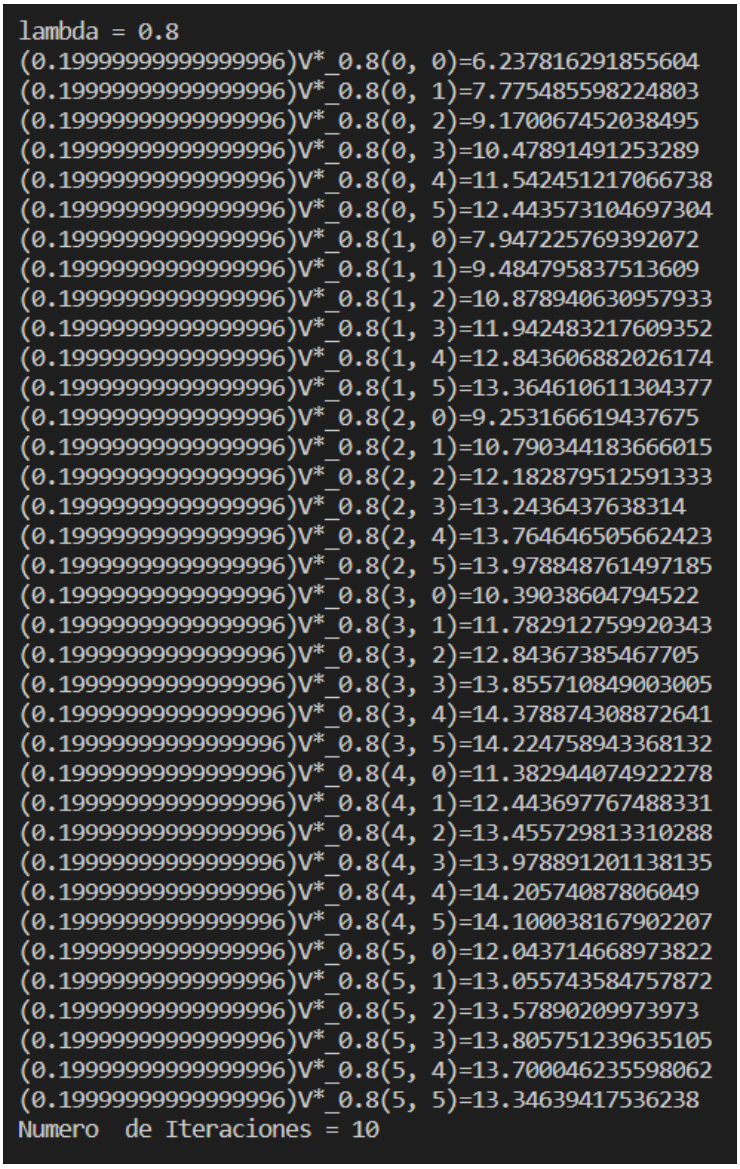

Figure 10: $(1 - \lambda V_\lambda)$ para $\lambda = 0.8$

```
lambda = 0.9
(0.09999999999999998)V*_0.9(0, 0)=3.9599486761631137
(0.09999999999999998)V*_0.9(0, 1)=4.7282728644505525
(0.09999999999999998)V*_0.9(0, 2)=5.421660934423098
(0.09999999999999998)V*_0.9(0, 3)=6.071257449141717
(0.09999999999999998)V*_0.9(0, 4)=6.587690906600535
(0.09999999999999998)V*_0.9(0, 5)=7.015301583179366
(0.09999999999999998)V*_0.9(1, 0)=4.809854205567437
(0.09999999999999998)V*_0.9(1, 1)=5.578120615269753
(0.09999999999999998)V*_0.9(1, 2)=6.271263376536915
(0.09999999999999998)V*_0.9(1, 3)=6.787698317507041
(0.09999999999999998)V*_0.9(1, 4)=7.215309436344114
(0.09999999999999998)V*_0.9(1, 5)=7.435429828129103
(0.09999999999999998)V*_0.9(2, 0)=5.439922010848341
(0.09999999999999998)V*_0.9(2, 1)=6.207970893202992
(0.09999999999999998)V*_0.9(2, 2)=6.900248360456765
(0.09999999999999998)V*_0.9(2, 3)=7.415318275915014
(0.09999999999999998)V*_0.9(2, 4)=7.635438493830175
(0.09999999999999998)V*_0.9(2, 5)=7.686286192829999
(0.09999999999999998)V*_0.9(3, 0)=6.007981037469543
(0.09999999999999998)V*_0.9(3, 1)=6.700256504705168
(0.09999999999999998)V*_0.9(3, 2)=7.215325736997907
(0.09999999999999998)V*_0.9(3, 3)=7.663247985168648
(0.09999999999999998)V*_0.9(3, 4)=7.886292644692149
(0.09999999999999998)V*_0.9(3, 5)=7.733019338500075
(0.09999999999999998)V*_0.9(4, 0)=6.500264453395852
(0.09999999999999998)V*_0.9(4, 1)=7.015331869234331
(0.09999999999999998)V*_0.9(4, 2)=7.463252909854451
(0.09999999999999998)V*_0.9(4, 3)=7.686297069311624
(0.09999999999999998)V*_0.9(4, 4)=7.7001443353816725
(0.09999999999999998)V*_0.9(4, 5)=7.580435858633693
(0.09999999999999998)V*_0.9(5, 0)=6.815336357472743
(0.09999999999999998)V*_0.9(5, 1)=7.263256589852387
(0.09999999999999998)V*_0.9(5, 2)=7.486300008523562
(0.09999999999999998)V*_0.9(5, 3)=7.500147155618639
(0.09999999999999998)V*_0.9(5, 4)=7.380438072282864
(0.09999999999999998)V*_0.9(5, 5)=7.080557420967389
Numero  de Iteraciones = 12
```

Figure 11: $(1 - \lambda V_\lambda)$ para $\lambda = 0.9$

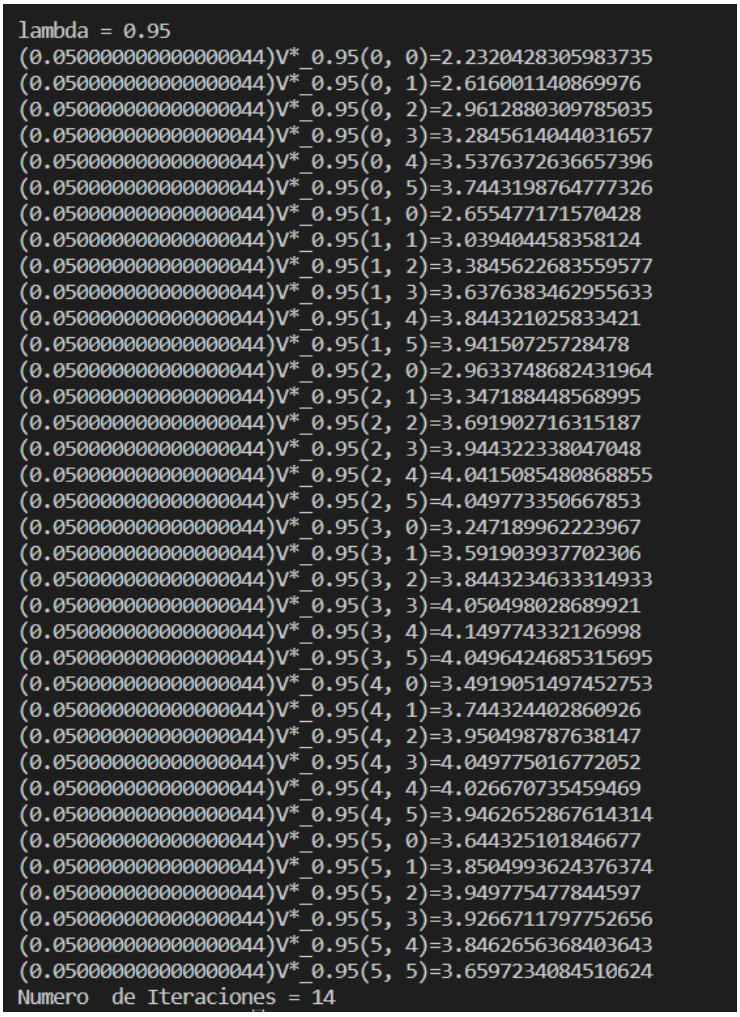

Figure 12: $(1 - \lambda V_\lambda)$ para $\lambda = 0.95$

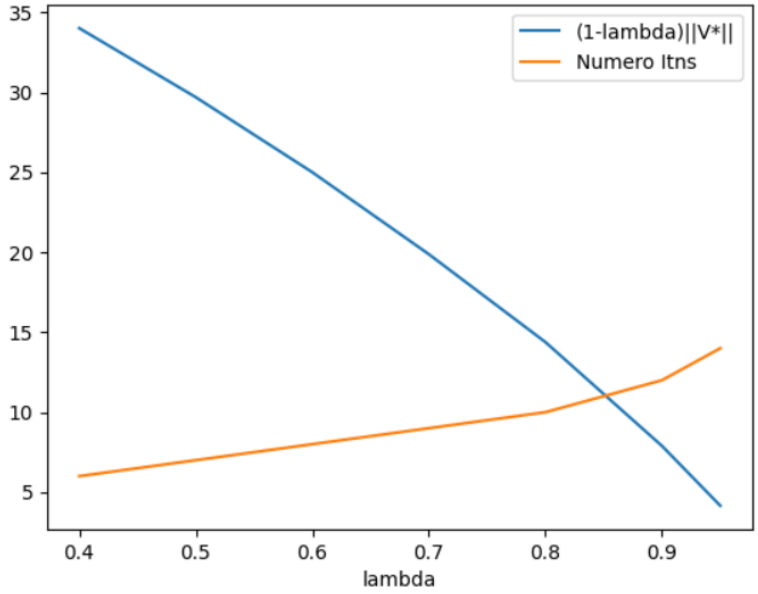

Figure 13: Analisis de convergencia a partir de λ