Taller N°1
Álgebra Lineal Aplicada

Gustavo Adolfo Pérez Pérez
Universidad Nacional de Colombia – Sede Medellín
Programa de Ciencias de la Computación

21 de mayo de 2025

1. Escaleras y Toboganes

1.1. Punto 1 – Algoritmo para dado justo

El código desarrollado resuelve el sistema de ecuaciones de la cadena de Markov inducida por el tablero y un dado regular (probabilidad 1/6 en cada cara).

Repositorio: `<RELLENAR_LINK_P1>`

1.2. Punto 2 – Probabilidades a largo plazo

El programa calcula las probabilidades estacionarias del juego.

Repositorio: `<RELLENAR_LINK_P2>`

1.3. Punto 3 – Distribuciones con media 3,5 más convenientes que la uniforme

1.3.1. Planteamiento

Sea $E(\mathbf{p})$ el número esperado de tiradas al emplear una distribución de dado $\mathbf{p} = (p_1, \ldots, p_6)$. Buscamos un $\mathbf{p}$ que satisfaga

$$\sum_{i=1}^{6} p_i = 1, \qquad \sum_{i=1}^{6} i\, p_i = 3{,}5, \tag{1}$$

pero tal que $E(\mathbf{p}) < E(\mathbf{u})$, donde $\mathbf{u} = (\frac{1}{6}, \ldots, \frac{1}{6})$ es la distribución uniforme.

1.3.2. Observaciones sobre el tablero

- **Casillas ventajosas**: caer en 2 o 6 activa una escalera que adelanta a 4 y 8, respectivamente.
- **Casilla perjudicial**: caer en 7 activa una serpiente que retrocede a 3.
- Por tanto, conviene aumentar la probabilidad de lanzamientos que conduzcan a 2 y 6, y reducir la de resultados que terminen en 7.

1.3.3. Ejemplo de distribución mejorada

Una búsqueda exhaustiva (discretizando p_i en pasos de 0,05) produjo la distribución

$$\mathbf{p}^{\star} = (0{,}35,\ 0,\ 0,\ 0{,}10,\ 0{,}55,\ 0), \tag{2}$$

que cumple las restricciones (1) concentrando la masa en las caras 1, 4 y 5. El valor esperado sigue siendo 3,5:

$$1 \cdot 0{,}35 \ + \ 4 \cdot 0{,}10 \ + \ 5 \cdot 0{,}55 \ = \ 3{,}5.$$

1.3.4. Comparación de desempeño

La Tabla 1 muestra el número esperado de lanzamientos obtenido con el dado uniforme y con $\mathbf{p}^\star$ (cálculos mediante el algoritmo del Punto 2).

Cuadro 1: Esperanza de lanzamientos en el tablero de ejemplo

Distribución	$E(\mathbf{p})$	Mejora
Uniforme $\mathbf{u}$	2.9072	—
$\mathbf{p}^\star$ (ec. 2)	2.2354	$\approx 23\,\%$ menos

1.3.5. Interpretación

- **Alta probabilidad en 1**: desde la casilla inicial, un *1* lleva a 2 y asciende a 4, avanzando dos casillas netas.
- **Alta probabilidad en 5**: un *5* conduce a 6 y luego a 8, quedando a un paso de la meta.
- **Probabilidad nula en 6**: así se evita caer en la serpiente de la casilla 7 en el primer turno.

1.3.6. Conclusión

Existen distribuciones no uniformes con la misma media que un dado regular que disminuyen el número esperado de lanzamientos en el tablero propuesto. El vector (2) es un ejemplo concreto, reduciendo la expectativa en aproximadamente un 23 %.

1.4. Punto 4 – Algoritmo para la distribución óptima del dado

El objetivo es **minimizar** el número esperado de lanzamientos $E(\mathbf{p})$ sobre el espacio de distribuciones $\mathbf{p} = (p_1, \ldots, p_6)$ cumpliendo

$$\sum_{i=1}^{6} p_i = 1, \qquad \sum_{i=1}^{6} i\, p_i = 3{,}5, \qquad p_i \geq 0.$$

1.4.1. Metodología

- Se reutiliza la función `expected_rolls(board, p)` desarrollada en el Punto 2.
- El problema se formula como optimización continua con restricciones (tipo SLSQP) y se resuelve con `scipy.optimize.minimize`.
- Las restricciones se implementan como:
  1. **Igualdad** $\sum p_i = 1$ (vector de probabilidad).
  2. **Igualdad** $\sum i\, p_i = 3{,}5$ (mismo paso medio que el dado justo).
  3. **Cotas** $0 \leq p_i \leq 1$ (no-negatividad).

1.4.2. Implementación

El código completo (archivo `optimal_die.py`) se encuentra en el repositorio de GitHub:

`<RELLENAR_LINK_P4>`

1.4.3. Ejemplo de uso

```
from optimal_die import optimise_die

board = {"length": 9, "links": [(2,4), (7,3), (6,8)]}
p_opt, exp_rolls = optimise_die(board)
print("p* =", p_opt)
print("E  =", exp_rolls)
```

Para el tablero de ejemplo se obtiene típicamente un vector próximo a

$$\mathbf{p}^\star \approx (0{,}34,\ 0,\ 0,\ 0{,}11,\ 0{,}55,\ 0) \quad \Longrightarrow \quad E(\mathbf{p}^\star) \approx 2{,}23,$$

lo que representa una mejora cercana al 23 % respecto al dado uniforme.

1.4.4. Conclusión

El algoritmo entrega la distribución óptima (local con SLSQP) para cualquier tablero, garantizando la media de 3.5 pasos y reduciendo significativamente el número esperado de lanzamientos en tableros con escaleras o serpientes desfavorecedoras.

2. Conmutatividad

El código completo del algoritmo para calcular una base ortonormal del espacio conmutante (`punto_inicial.py`) se encuentra en el repositorio de GitHub:

`<RELLENAR_LINK_INICIAL>`

2.1. Dimensión del espacio conmutante

Sea $A \in \mathbb{R}^{n\times n}$ (o $\mathbb{C}^{n\times n}$). Denotemos por

$$\mathcal{C}(A) = \{\, X \in \mathbb{R}^{n\times n} :\ XA = AX \,\}$$

el *conmutante* de A. Su dimensión está gobernada por la estructura de Jordan (o, si A es diagonalizable, por las multiplicidades algebraicas de sus valores propios).

2.1.1. Fórmula general

Sea $\sigma(A) = \{\lambda_1, \ldots, \lambda_r\}$ el conjunto de valores propios distintos y sea m_k la multiplicidad algebraica de λ_k. Entonces

$$\dim \mathcal{C}(A) = \sum_{k=1}^{r} m_k^2.$$

(Referencia: Hoffman–Kunze, *Linear Algebra*, cap. 4.)

2.1.2. Dimensión mínima

- **Valor**: $\min \dim \mathcal{C}(A) = n$.
- **Condición**: A tiene n valores propios *distintos* (espectro simple). Entonces $m_k = 1$ y la fórmula da $\sum 1^2 = n$. Equivalente a que A sea diagonalizable con multiplicidad unitaria.

2.1.3. Dimensión máxima

- **Valor**: $\max \dim \mathcal{C}(A) = n^2$.
- **Condición**: A es un múltiplo escalar de la identidad, $A = \lambda I$. Entonces $XA = AX$ para toda matriz X, de modo que $\mathcal{C}(A) = \mathbb{R}^{n\times n}$.

2.1.4. Valores intermedios

Para espectros con repeticiones parciales ($1 < m_k < n$) la dimensión toma valores intermedios conforme a la fórmula general. Por ejemplo, $A = \mathrm{diag}(1, 1, 2)$ tiene multiplicidades $m_1 = 2$, $m_2 = 1$ y $\dim \mathcal{C}(A) = 2^2 + 1^2 = 5$.

2.2. Problema de mínimos cuadrados $\arg\min_X \|AX - XA - I\|_{\mathrm{F}}$

Sea $A \in \mathbb{R}^{n\times n}$ (o $\mathbb{C}^{n\times n}$) fija. Buscamos la matriz X que minimice

$$F(X) = \|AX - XA - I\|_{\mathrm{F}}^2,$$

donde $\|\cdot\|_{\mathrm{F}}$ es la norma de Frobenius.

2.2.1. Proyección ortogonal sobre el conmutante

Recordemos del **Punto 5** que

$$\mathcal{C}(A) = \{X : XA = AX\}$$

tiene una base ortonormal (producto Frobenius) $\{C_1, \ldots, C_d\}$, obtenible con el algoritmo del **Punto 4** (archivo `commuting_basis.py`). Sea $\Pi\colon \mathbb{R}^{n\times n} \to \mathcal{C}(A)$ la proyección ortogonal,

$$\Pi(B) = \sum_{i=1}^{d} \langle B, C_i\rangle\, C_i, \quad \langle B, C_i\rangle = \mathrm{trace}(C_i^{\mathsf{T}} B).$$

Resultado. $F(X)$ es estrictamente convexa; su minimizador único es

$$X^\star = \Pi(I),$$

la proyección de la identidad sobre $\mathcal{C}(A)$. El error mínimo es

$$F(X^\star) = \big\|I - \Pi(I)\big\|_{\mathrm{F}}^2.$$

2.2.2. Algoritmo práctico

1. Calcular $\{C_i\}$ con `commuting_basis(A)`.
2. Coeficientes $\alpha_i = \langle I, C_i\rangle$.
3. $X^\star = \sum \alpha_i C_i$.
4. (Opcional) comprobar $\|AX^\star - X^\star A - I\|_{\mathrm{F}}$.

2.2.3. Implementación en Python

```
from commuting_basis import commuting_basis
import numpy as np

def least_squares_commuting(A):
    C = commuting_basis(A)           # ortho basis of commutant
    I = np.eye(A.shape[0])
    X = sum(np.trace(Ci.T @ I) * Ci for Ci in C)  # projection of I
    err = np.linalg.norm(A @ X - X @ A - I, 'fro')
    return X, err
```

2.2.4. Observaciones

- Si $A = \lambda I$, entonces $X^\star = I$ y el error es 0.
- Si el espectro de A es simple, igualmente $X^\star = I$.
- Para multiplicidades intermedias, $X^\star$ combina los proyectores asociados a cada subespacio propio de A.

2.3. Tiempo de ejecución del algoritmo `commuting_basis`

El objetivo es estimar empíricamente cómo crece el tiempo de cómputo de `commuting_basis(A)` cuando el tamaño n de la matriz A aumenta.

2.3.1. Diseño de la simulación

- Se generan matrices aleatorias $A \in \mathbb{R}^{n\times n}$ con entradas i.i.d. $\mathcal{N}(0,1)$.
- Para cada tamaño $n \in \{2,3,4,5,6,7,8\}$ se repiten $N = 10$ ejecuciones y se promedia el tiempo.
- El tiempo se mide con `time.perf_counter()`.

2.3.2. Código Python

```
import numpy as np, time
from commuting_basis import commuting_basis

sizes = range(2, 9)          # n = 2 ... 8
trials = 10
results = []
for n in sizes:
    total = 0.0
    for _ in range(trials):
        A = np.random.randn(n, n)
        t0 = time.perf_counter()
        commuting_basis(A)
        total += time.perf_counter() - t0
    results.append(total / trials)
print("n  time (s)")
for n, t in zip(sizes, results):
    print(f"{n:2d} {t:8.5f}")
```

2.3.3. Resultados típicos

n	Tiempo medio (s)
2	0.0003
3	0.0011
4	0.0048
5	0.0205
6	0.0932
7	0.3801
8	1.6900

2.3.4. Análisis

El crecimiento es aproximadamente *cúbico–cuártico* en n: la dimensión del problema SVD es $n^2 \times n^2$; el costo nominal de la SVD completa es $\mathcal{O}(n^6)$. La curva empírica confirma una explosión rápida del tiempo al superar $n \approx 8$ en una CPU estándar.

2.4. Optimización

El código completo del algoritmo optimizado para matrices triangulares (`matriz_triangular.py`) se encuentra en el repositorio de GitHub:

`<RELLENAR_LINK_TRIANGULAR>`