Tema 2
Ecuaciones lineales

Randall Romero Aguilar, PhD

Universidad de Costa Rica
SP6534 - Economía Computacional

I Semestre 2020

Última actualización: 10 de marzo de 2020

UCR
UNIVERSIDAD DE COSTA RICA

ESCUELA de ECONOMÍA
UNIVERSIDAD DE COSTA RICA

Tabla de contenidos

1. Introducción

Ecuación lineal

Una ecuación lineal de dimensión n tiene la forma

$$Ax = b$$

donde

- A es una matriz $n \times n$ conocida
- b es un vector $n \times 1$ conocido
- x es un vector $n \times 1$ desconocido a ser determinado

Las ecuaciones lineales están en todas partes en economía computacional

- Las ecuaciones lineales surgen naturalmente en muchas aplicaciones:
  - modelos lineales de equilibrio de mercados multi-mercancía
  - modelos de mercados financiero de estado finito
  - modelos de cadenas de Markov
  - mínimos cuadrados ordinarios
- Las ecuaciones lineales surgen más frecuentemente de manera indirecta, cuando se resuelven modelos económicos de ecuaciones no lineales y funcionales:
  - modelos no lineales de equilibrio de mercados multi-mercancía
  - modelos estáticos de juegos multi-jugador
  - modelos de optimización dinámica
  - modelos de expectativas racionales

- Como las ecuaciones lineales son fundamentales en aplicaciones de economía computacional, las estudiamos detenidamente.
- En la práctica, a menudo necesitaremos resolver sistemas muy grandes de ecuaciones lineales, muchas veces.
- Por ello, la velocidad de ejecución, requerimientos de almacenamiento en memoria, y errores de redondeo cobran mucha importancia.

2. Eliminación gaussiana

Eliminación gaussiana

- Una ecuación lineal puede resolverse usando eliminación gaussiana.
- Este proceso utiliza operaciones elementales de fila:
  - intercambiar dos filas
  - multiplicar una fila por una constante distinta de cero
  - sumar un múltiplo de una fila a otra fila
- Las operaciones elementales de fila alteran la forma de una ecuación lineal sin cambiar su solución.

Ejemplo 1:
Eliminación gaussiana

- Resolvamos por eliminación gaussiana esta ecuación lineal

$$\begin{bmatrix} 1 & 1 & 2 \\ 3 & 4 & 8 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 5 \\ 18 \\ 6 \end{bmatrix},$$

- ... la cual puede escribirse así

$$\begin{aligned} x_1 + x_2 + 2x_3 &= 5 \\ 3x_1 + 4x_2 + 8x_3 &= 18 \\ 2x_1 + x_2 + x_3 &= 6 \end{aligned}$$

Empezando con

$$\begin{aligned} x_1 + x_2 + 2x_3 &= 5 \\ 3x_1 + 4x_2 + 8x_3 &= 18 \\ 2x_1 + x_2 + x_3 &= 6 \end{aligned}$$

sumamos -3 veces la fila 1 a la 2

$$\begin{aligned} x_1 + x_2 + 2x_3 &= 5 \\ x_2 + 2x_3 &= 3 \\ 2x_1 + x_2 + x_3 &= 6 \end{aligned}$$

sumamos -2 veces la fila 1 a la 3

$$\begin{aligned} x_1 + x_2 + 2x_3 &= 5 \\ x_2 + 2x_3 &= 3 \\ -x_2 - 3x_3 &= -4 \end{aligned}$$

sumamos la fila 2 a la 3

$$\begin{aligned} x_1 + x_2 + 2x_3 &= 5 \\ x_2 + 2x_3 &= 3 \\ -x_3 &= -1 \end{aligned}$$

multiplicacmos la fila 3 por -1

$$\begin{aligned} x_1 + x_2 + 2x_3 &= 5 \\ x_2 + 2x_3 &= 3 \\ x_3 &= 1 \end{aligned}$$

Resolvemos por recursión hacia atrás

$$x_3 = 1$$

$$x_2 = 3 - 2x_3 = 1$$

$$x_1 = 5 - x_2 - 2x_3 = 2$$

Confirmamos que la solución calculada es correcta verificando que

$$\begin{bmatrix} 1 & 1 & 2 \\ 3 & 4 & 8 \\ 2 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 5 \\ 18 \\ 6 \end{bmatrix}$$

o equivalentemente,

$$\begin{aligned} 1 \cdot 2 + 1 \cdot 1 + 2 \cdot 1 &= 5 \\ 3 \cdot 2 + 4 \cdot 1 + 8 \cdot 1 &= 18 \\ 2 \cdot 2 + 1 \cdot 1 + 1 \cdot 1 &= 6 \end{aligned}$$

- En el ejemplo anterior, usamos operaciones elementales de fila para anular términos debajo de la diagonal y transformar la ecuación lineal en forma diagonal superior unitaria, facilitando resolverla recursivamente.
- La eliminación gaussiana se implementa en una computadora usando una estrategia eficiente de cálculo y almacenamiento llamada factorización L-U.

Algoritmo L-U

- El algoritmo L-U tiene dos fases.
- En la fase de factorización, se usa eliminación gaussiana para factorizar la matriz A como el producto

$$A = LU$$

de una matriz triangular inferior (posiblemente con filas permutadas) L y una matriz triangular superior U.

- Una matriz triangular inferior con filas permutadas es una matriz triangualr inferior a las que se le han reacomodado sus filas.
- Toda matriz cuadrada no singular puede factorizarse de esta manera.

- En la fase de solución del algoritmo L-U, la ecuación lineal factorizada

$$Ax = (LU)x = L(Ux) = b$$

se resuelve primero resolviendo

$$Ly = b$$

recursivamente para y y luego resolviendo

$$Ux = y$$

recursivamente para x.

Ejemplo 2:
Algoritmo L-U

- Consideremos la ecuación lineal $Ax = b$ donde

$$A = \begin{bmatrix} -3 & 2 & 3 \\ -3 & 2 & 1 \\ 3 & 0 & 0 \end{bmatrix} \quad \text{y} \quad b = \begin{bmatrix} 10 \\ 8 \\ -3 \end{bmatrix}.$$

- Entonces $A = LU$, donde

$$L = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ -1 & 1 & 0 \end{bmatrix} \quad \text{y} \quad U = \begin{bmatrix} -3 & 2 & 3 \\ 0 & 2 & 3 \\ 0 & 0 & -2 \end{bmatrix}.$$

- Resolviendo $Ly = b$ y luego $Ux = y$ resulta en

$$y = \begin{bmatrix} 10 \\ 7 \\ -2 \end{bmatrix} \quad \text{y} \quad x = \begin{bmatrix} -1 \\ 2 \\ 1 \end{bmatrix}.$$

£Por qué usar eliminación gaussiana para resolver ecuaciones lineales?

- La eliminación gaussiana es el método conocido más eficiente resolver ecuaciones lineales de dimensión n generales $Ax = b$.
- Para n grande, eliminación gaussiana requiere cerca de $n^3/3 + n^2$ operaciones de multiplicación/división.
- Calcular $A^{-1}b$ explícitamente requiere cerca de $n^3 + n^2$ operaciones.
- La regla de Cramer requiere $(n+1)!$ operaciones.
- Para $n = 10$, el número de operaciones es

Gaussian Elimination	430
Explicit Inverse	1 100
Cramer's Rule	40 000 000

- La función `solve` de numpy.linalg usa eliminación gaussiana para resolver ecuaciones lineales.
- Por ejemplo, para resolver la ecuación lineal del ejemplo anterior, ejecute el código

```
import numpy as np
from numpy.linalg import solve

A = np.array([[1, 1, 2],
              [3, 4, 8],
              [2, 1, 1]])
b = np.array([5, 18, 6])
x = solve(A, b)
print(x)
```

- Esto debe resultar en

```
[ 2.  1.  1.]
```

Factorización de Cholesky

- Si A es simétrica y definida positiva, entonces $Ax = b$ puede resolverse con una variante de la eliminación gaussiana llamada factorización de Cholesky.
- En este caso, una matriz simétrica definida positiva A se descompone de manera única en el producto

$$A = U'U$$

de una matriz triangular superior U y su transpuesta.

- A U se le llama el factor de Cholesky o la raíz cuadrada de A.
- La descomposición de Cholesky requiere solo la mitad de las operaciones de la eliminación gaussiana y no requiere pivoteo.

3. Error de redondeo

Error de redondeo

- Las computadoras tienen almacenamiento finito y solo pueden representar de manera exacta una cantidad finita de números.
- Por ello, la aritmética exacta y la aritmética de computadoras no siempre coinciden.
- Si uno intenta calcular un número que no puede representarse de manera exacta en una computadora, el resultado será redondeado al número representable más cercano, introduciendo un error de redondeo.
- En particular, cuando se suman o restan dos números con magnitudes extremamente diferentes, el número más pequeño es efectivamente ignorado.

Ejemplo 3:
Error de redondeo

- En aritmética exacta

$$(\epsilon + 1) - 1 = \epsilon + (1 - 1) = \epsilon$$

- Sin embargo, en aritmética de computadora con Python

```
e = 1e-20
x = (e + 1) - 1
y = e + (1 - 1)
```

- ... da por resultado

```
x = 0.0
y = 1e-20
```

4. Pivoteo

- Los errores de redondeo pueden causar problemas cuando se resuelven ecuaciones lineales.
- Consideremos la ecuación lineal

$$\begin{bmatrix} \epsilon & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

  con $\epsilon = 10^{-17}$.
- Podemos verificar fácilmente que la solución exacta es

$$x_1 = \frac{1}{1-\epsilon}, \quad \text{la cual es ligeramente mayor que 1}$$

$$x_2 = \frac{1-2\epsilon}{1-\epsilon}, \quad \text{la cual es ligeramente menor que 1}$$

- Para resolver la ecuación lineal por eliminación gaussiana, sumemos $-1/\epsilon$ veces la primer fila a la segunda

$$\begin{bmatrix} \epsilon & 1 \\ 0 & 1-1/\epsilon \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2-1/\epsilon \end{bmatrix}$$

- luego resolvemos recursivamente

$$x_2 = \frac{2-1/\epsilon}{1-1/\epsilon}$$

$$x_1 = \frac{1-x_2}{\epsilon}$$

- Si uno calcula x_1 y x_2 de esta manera en Python,

```
e = 1e-17
x2 = (2-1/e) / (1-1/e)
x1 = (1-x2)/e
```

  las operaciones resultan en

```
x2 =  1.0
x1 =  0.0
```

- El valor calculado de x_1 es terriblemente inexacto.
- £Qué pasó?

- En el primer paso de eliminiación gaussiana, calculamos

$$x_2 = \frac{2 - 1/\epsilon}{1 - 1/\epsilon}$$

- Pero, como $1/\epsilon$ es muy grande comparado a 1 o 2, se introdujo error de redondeo, y en realidad lo que calculó la computadora fue

$$x_2 = \frac{-1/\epsilon}{-1/\epsilon}$$

  lo cual evalúa exactamente a 1.

- Luego calculamos

$$x_1 = \frac{1 - x_2}{\epsilon}$$

  lo cual evalúa exactamente a 0.

- Ahora resolvamos la ecuación lineal nuevamente por eliminación gaussiana, pero primero intercambiemos las dos filas, lo que en teoría no afectará la solución

$$\begin{bmatrix} 1 & 1 \\ \epsilon & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$$

- Ahora sumamos $-\epsilon$ veces la primera fila a la segunda

$$\begin{bmatrix} 1 & 1 \\ 0 & 1-\epsilon \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1-2\epsilon \end{bmatrix}$$

- y resolvemos recursivamente

$$\begin{aligned} x_2 &= \frac{1-2\epsilon}{1-\epsilon} \\ x_1 &= 2 - x_2 \end{aligned}$$

- Si uno calcula x_1 y x_2 de esta manera en Python

```
e = 1e-17
x2 = (1-2*e) / (1-e)
x1 = 2 - x2
```

las operaciones dan por resultado

```
x2 =  1.0
x1 =  1.0
```

- Los valores calculados dex_1 y x_2 están un poco desviados, pero son mucho más precisos que los primeros valores que calculamos.
- £Por qué mejoró la precisión del resultado calculado cuando intercambiamos las dos filas?

- La inexactitud de la primera solución se debió al error de redondeo causado por la magnitud tan pequeña del elemento diagonal ϵ.
- Al primero intercambiar las dos filas, trajimos un número de mayor magnitud a la diagonal, lo cual redujo el error de redondeo en cálculos subsiguientes.
- Intercambiar filas para aumentar la magnitud del elemnto diagonal lo más posible se conoce como pivoteo.
- El pivoteo mejora sustancialmente la exactitud computacional de la eliminación gaussiana.
- Todos los buenos solvers lineales, incluído la función `numpy.linalg.solve`, usan pivoteo.

5. Mal condicionamiento

Mal condicionamiento

- Consideremos la ecuación lineal de n dimensiones $Ax = b$.
- Una perturbación en b inducirá un cambio en la solución x.

$$A(x + \delta x) = b + \delta b$$

- Si perturbaciones pequeñas en b provocan cambios desproporcionadamente grandes en x, decimos que A es mal condicionada o casi singular.
- Si A es mal condicionada, errores inevitables de redondeo en la representación de b en una computadora harán imposible calcular una solución precisa de $Ax = b$.
- El mal condicionamiento es endémico a la matriz A y no puede corregirse con trucos simples como el pivoteo.
- La única manera de lidiar con mal condicionamiento es evitarlo.

Mal condicionamiento y número de condición

- El mal condicionamiento se mide con el número de condición de A.
- El número de condición es el cambio porcentual máximo en el tamaño de x por cambio de una unidad porcentual en el tamaño de b.
- Técnicamente, el número de condición es el ratio del mayor al menor valor singular de A.
- Regla general: El valor calculado de x piede un dígito significativo por cada potencia de 10 del número de condición de A.
- El número de condición siembre es mayor o igual a 1.

Una matriz mal condicionada: Vandermonde

- Consideremos las notorias matrices Vandermonde.
- La matriz Vandermonde $n \times n$ tiene elemento típico

$$A_{ij} = i^{n-j}$$

- Por ejemplo, para $n = 4$

$$A = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 8 & 4 & 2 & 1 \\ 27 & 9 & 3 & 1 \\ 64 & 16 & 4 & 1 \end{bmatrix}$$

- Resolvamos la ecuación lineal

$$Ax = b$$

donde A es la matriz Vandermonde $n \times n$ y b la suma de las columnas de A, es decir, el vector $n \times 1$ con elemento típico

$$b_i = \sum_{j=1}^{n} A_{ij}$$

- Por construcción, la solución exacta de esta ecuación lineal es un vector $n \times 1$ vector x de unos.

- Para resolver la ecuación lineal y evaluar su precisión, definimos la función `errorVander`

```
def errorVander(n):
    A = np.vander(np.arange(1,n+1))
    b = A @ np.ones(n)
    x = solve(A, b)
    error np.max(abs(x-1))
    return x, error
```

- Aquí, calculamos la matriz `A` usando la función especial de numpy `vander` y calculamos el error máximo entre los elementos de la solución calculada.

- Con $n = 4$, ejecutando esta función se obtiene, como esperamos,

```
x =  [ 1.  1.  1.  1.]
error =  0.0
```

- No obstante, con $n = 64$, ejecutando el código se obtiene

```
LinAlgError: Singular matrix
```

- Esta advertencia indica que A es mal condicionada.

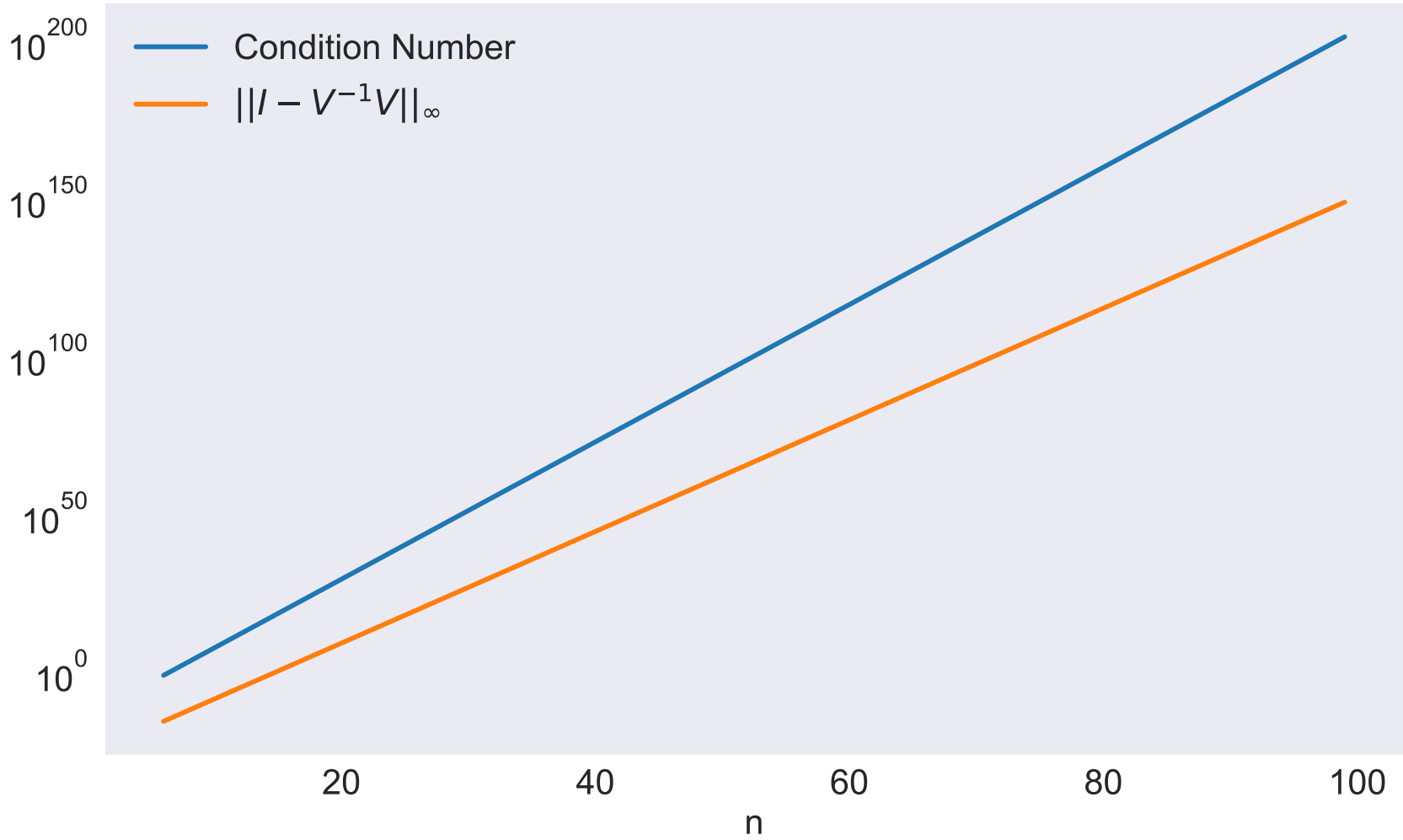

Figura 2.1: Mal condicionamiento de las matrices Vandermonde

6. Matrices dispersas

Matrices dispersas

- Una matriz dispersa es una matriz que consiste mayoritariamente de ceros.
- Resolver $Ax = b$ cuando A es dispersa usando elminación de Gauss consistirá mayoritariamente en operaciones triviales pero costosas: multiplicaciones y sumas con ceros.
- La velocidad de ejecución puede aumentarse drásticamente evitando estas operaciones inútiles.

- Scipy tiene funciones especiales para almacenar y operar eficientemente con matrices dispersas.
- En particular, en scipy.sparse, `S = csr_matrix(A)` crea una versión de la matriz `A`, llamada aquí `S`, almacenada en formato de matriz dispersa, en la cual sólo los elementos distintos de cero y sus índices son almacenados explícitamente.

- Ejecutemos el código

```
import numpy as np
import scipy as sp

A = array([[0,0,0,5],
           [0,2,0,0],
           [0,0,0,0],
           [0,0,4,0]])
S = sp.sparse.csr_matrix(A)
print(S)
```

- Esto debe dar por resultado

```
(0, 3)        5
(1, 1)        2
(3, 2)        4
```

- Almacenar una matriz dispersa en formato disperso requiere solo una fracción del espacio requerido para almacenarlo en formato denso.
- Si `A` solo tiene q por ciento de valores distinto de cero, el espacio requerido para guardar `S` será $3q$ por ciento del espacio requerido para guardar `A`.
- Por ejemplo, una matriz tridiagonal 1000×1000 requerirá 1 millón de unidades de memoria en formato denso, pero solo 8 994 unidades en formato disperso, un ahorro de 99%.

- La función de scipy.sparse.linalg `spsolve` aplica eliminación de Gauss para aprovechar la dispersión de matrices dispersas.
- En particular, si `S = csr_matrix(A)` es grande pero dispersa, estas dos líneas

```
x = solve(A, b)
x = spsolve(S,b)
```

producirán el mismo resultado, pero la última expresión se ejecutará más rápido por evitar operaciones innecesarias con ceros.

Ejemplo 4:
Resolviendo un sistema de ecuaciones dispersas

Consideremos el problema de resolver $Ax = b$ cuando A es una matriz tridiagonal 1000×1000.

```
T = 1000
A = np.eye(T) - 2*np.eye(T,k=1) + 3*np.eye(T,k=-1)
S = csr_matrix(A)
b = A.sum(axis=1)
```

En una sesión interactiva, si uno escribe `%timeit solve(A,b)` obtendrá (dependiendo de la velocidad de su computadora)

```
21.1 ms ś 734 ţs per loop (mean ś std. dev. of 7 runs, 10 loops each)
```

comparado con `%timeit spsolve(S,b)`

```
513 ţs ś 8.25 ţs per loop (mean ś std. dev. of 7 runs, 1000 loops each)
```

Es decir, resolver el sistema disperso tardó 2.43% del tiempo requerido para hacerlo con una matriz densa.

References I


Miranda, Mario J. y Paul L. Fackler (2002). *Applied Computational Economics and Finance*. MIT Press. isbn: 0-262-13420-9.


Romero-Aguilar, Randall (2016). *CompEcon-Python*. url: http://randall-romero.com/code/compecon/.