Computación

Subrutinas 2

Repaso de la clase anterior

- Las subrutinas no son funciones, su uso es como un “Sub Programa”
- Se llaman con la orden CALL NOMBRE()
- Dentro de los () van los argumentos (variables que intercambian) sin distinción de entrada ni de salida
- La subrutina se construye fuera del programa que la llama. Y una subrutina puede llamar a otra.

- Los tipos de variables en los argumentos deben coincidir en tipo y dimensión.
- Hay libros y páginas web con subrutinas ya hechas y probadas.
- Veamos un ejemplo para conseguir números al azar.

Aplicaciones actuales:

Simulaciones con números al azar

$$X_{i+1} = \frac{(AX_i + C)mod(M)}{M}$$

```
Subroutine azar(X)
real*8 X
integer A,C
A = 24298
C = 99991
M = 199017
X = MOD(X * A + C,M) /M
return
end
```

Al X_0 se lo llama la “semilla”,
Ya que inicia la secuencia
Con el X_i se calcula el X_{i+1}

Puedo cambiar la escala de los X_i

Simulaciones

- Se usan para testar situaciones de todo tipo.
- Suelen consumir grandes cantidades de números a la azar, como valores a simular y sus errores
- Veamos como calcular el número π

Frecuencia

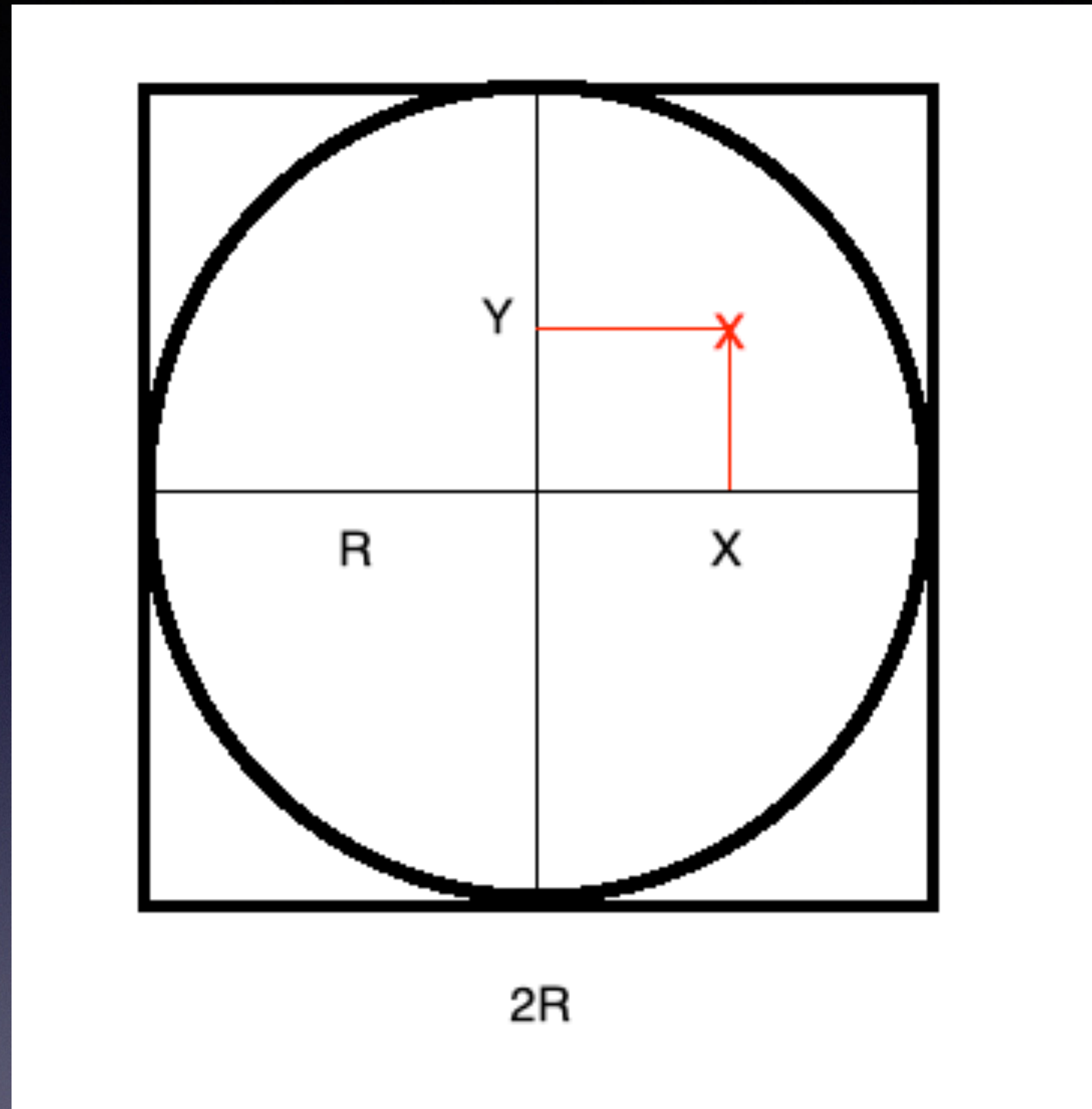
- La frecuencia la mido, por ejemplo si tiro una moneda: cuanto cuantas caras y secas por separados y divido estos números por a cantidad de veces que tiré la moneda:

- Por ejemplo: 10 tiradas las frecuencias podrían ser 0.7 caras y 0.3 secas

Probabilidad

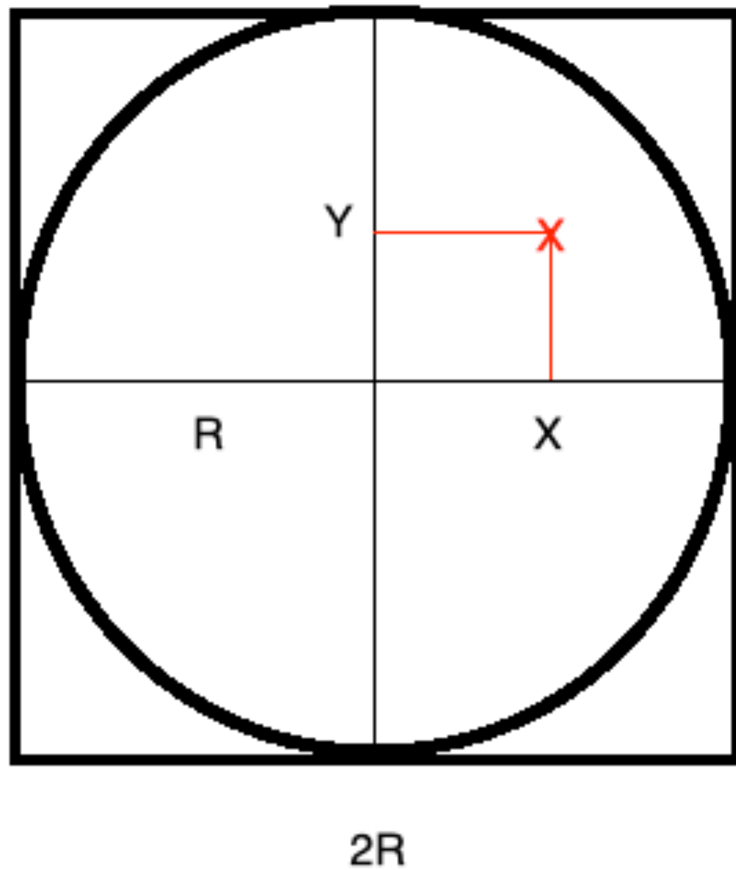
- La Probabilidad es la frecuencia pero si tiro la moneda infinitas veces. Es un límite al infinito de mi experimento.
- La probabilidad se calcula a partir de
- $Prob = caso\ de\ interes/casos\ totales$
- Pero estos casos NO son los medidos, para una moneda serían:
- Prob = 1/2 por ejemplo si me interesa la prob d que salga cara, es cara/(cara+ seca)

Calcularemos el número π a partir de números al azar



Frecuencia = Piedras en el círculo/Total de Piedras
En el infinito la frecuencia se vuelve la probabilidad

Calcularemos el número PI a partir de números al azar



Prob = Casos favorables/Total de casos

Prob = Área de círculo/Área del cuadrado

$$Prob = \frac{\pi R^2}{(2R)^2} = \frac{\pi}{4}$$

$$\pi \sim 4\text{frec}$$

Prob ~ frec
(en el infinito)

C. Feinstein 2021

Programas

$$Recordar\ que\ \pi = 3.141592653$$

- PiA -> primer programa, sólo calcula el número pi.
- PiB -> Cálculo y escribe el resultado parcial cada millón de puntos.
- PiC -> Escribe cada millón de puntos y genera una nueva semilla en el generador de números al azar.
- PiD -> Ahora tiene un estimador del error
- PiE -> Probamos un generador de números al azar más complejo, pero que consume más recursos.

C. Feinstein 2021

Resultados

# de piedras	Valor de π obtenido	Error estadístico
10^6	3.1424319999999999	$1.00000005\ 10^{-3}$
10^7	3.1407848888888887	$3.33333330\ 10^{-4}$
10^8	3.1416244799999999	$9.99999975\ 10^{-5}$
10^9	3.1415663320000000	$3.16227779\ 10^{-5}$
10^{10}	3.1416044472000002	$9.99999975\ 10^{-6}$
10^{11}	3.1415875904799999	$3.16227761\ 10^{-6}$
10^{12}	3.1415921563320000	$9.99999997\ 10^{-7}$
10^{13}	3.1415922104799999	$3.16227761\ 10^{-7}$

$$\pi = 3.14159265358979323844$$

C. Feinstein 2021

Numerical Recipes
(Recetas numéricas)

Ejemplo de un libro donde hay subrutinas ya escritas

Se puede encontrar acá:

https://websites.pmc.ucsc.edu/~fnimmo/eart290c_17/NumericalRecipesinF77.pdf

- Sistemas de ecuaciones -> Página 22
- Integración de funciones -> Página 123
- Números al azar -> Página 266

C. Feinstein 2021

2.3 LU Decomposition and Its Applications

Suppose we are able to write the matrix **A** as a product of two matrices,

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A} \tag{2.3.1}$$

where **L** is *lower triangular* (has elements only on the diagonal and below) and **U** is *upper triangular* (has elements only on the diagonal and above). For the case of

a 4×4 matrix **A**, for example, equation (2.3.1) would look like this:

$$\begin{bmatrix} \alpha_{11} & 0 & 0 & 0 \\ \alpha_{21} & \alpha_{22} & 0 & 0 \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & 0 \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \alpha_{44} \end{bmatrix} \cdot \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ 0 & \beta_{22} & \beta_{23} & \beta_{24} \\ 0 & 0 & \beta_{33} & \beta_{34} \\ 0 & 0 & 0 & \beta_{44} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \tag{2.3.2}$$

We can use a decomposition such as (2.3.1) to solve the linear set

$$\mathbf{A} \cdot \mathbf{x} = (\mathbf{L} \cdot \mathbf{U}) \cdot \mathbf{x} = \mathbf{L} \cdot (\mathbf{U} \cdot \mathbf{x}) = \mathbf{b} \tag{2.3.3}$$

by first solving for the vector **y** such that

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b} \tag{2.3.4}$$

and then solving

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y} \tag{2.3.5}$$

What is the advantage of breaking up one linear set into two successive ones? The advantage is that the solution of a triangular set of equations is quite trivial, as we have already seen in §2.2 (equation 2.2.4). Thus, equation (2.3.4) can be solved by *forward substitution* as follows,

$$\begin{aligned} y_1 &= \frac{b_1}{\alpha_{11}} \\ y_i &= \frac{1}{\alpha_{ii}} \left[b_i - \sum_{j=1}^{i-1} \alpha_{ij} y_j \right] \qquad i = 2, 3, \ldots, N \end{aligned} \tag{2.3.6}$$

while (2.3.5) can then be solved by *backsubstitution* exactly as in equations (2.2.2)–(2.2.4),

$$\begin{aligned} x_N &= \frac{y_N}{\beta_{NN}} \\ x_i &= \frac{1}{\beta_{ii}} \left[y_i - \sum_{j=i+1}^{N} \beta_{ij} x_j \right] \qquad i = N-1, N-2, \ldots, 1 \end{aligned} \tag{2.3.7}$$

To summarize, this is the preferred way to solve the linear set of equations $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$:

```
call ludcmp(a,n,np,indx,d)
call lubksb(a,n,np,indx,b)
```

The answer $\mathbf{x}$ will be returned in `b`. Your original matrix $\mathbf{A}$ will have been destroyed.

If you subsequently want to solve a set of equations with the same $\mathbf{A}$ but a different right-hand side $\mathbf{b}$, you repeat *only*

```
call lubksb(a,n,np,indx,b)
```

not, of course, with the original matrix $\mathbf{A}$, but with `a` and `indx` as were already returned from `ludcmp`.

```
SUBROUTINE ludcmp(a,n,np,indx,d)
INTEGER n,np,indx(n),NMAX
REAL d,a(np,np),TINY
PARAMETER (NMAX=500,TINY=1.0e-20)   Largest expected n, and a small number.
```

Given a matrix `a(1:n,1:n)`, with physical dimension `np` by `np`, this routine replaces it by the LU decomposition of a rowwise permutation of itself. `a` and `n` are input. `a` is output, arranged as in equation (2.3.14) above; `indx(1:n)` is an output vector that records the row permutation effected by the partial pivoting; `d` is output as ± 1 depending on whether the number of row interchanges was even or odd, respectively. This routine is used in combination with `lubksb` to solve linear equations or invert a matrix.

Sentencia Common - Include

Permite enviar argumento que no necesariamente esté en el nombre de la subrutina. Sirve cuando se trabaja con muchas variables

common /nombre1/ Lista de variables 1
common /nombre2/ Lista de variables 2

Ejemplo:

```
COMMON /listado1/ A,B,C,IK,X(1000)
COMMON /listado2/ B1,B2,B3,B4
```

Y en la subroutine pondría

```
SUBROUTINE SUB1()
COMMON /listado1/ x,y,j,es(1000)
…
…
RETURN
END
```