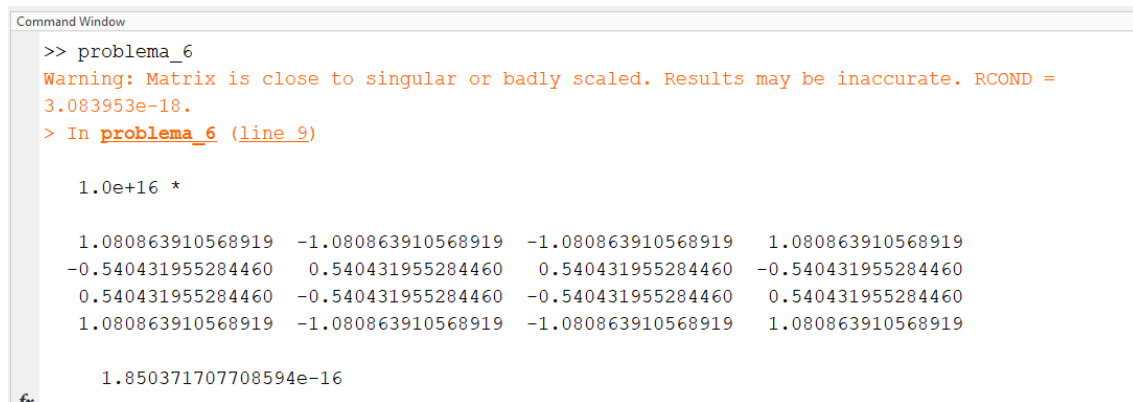


Verifica si los siguientes vectores en \mathbb{R}^4 son linealmente independientes $v_1 = [0 \ 1 \ 0 \ 1]$, $v_2 = [1 \ 2 \ 3 \ 4]$, $v_3 = [1 \ 0 \ 1 \ 0]$, $v_4 = [0 \ 0 \ 1 \ 1]$.

Una primera forma para determinar si estos vectores son L.I. es ver que la matriz formada por ellos como vectores columna es no singular, es decir que existe su inversa o bien, su determinante es distinto de cero. Utilizando las funciones `inv()` y `det()` se trató de comprobar este hecho.

```
%Declarar vectores columna v1,v2,v3 y v4
v1=[0,1,0,1]';
v2=[1,2,3,4]';
v3=[1,0,1,0]';
v4=[0,0,1,1]';
%La matriz A está formada por los vectores en columnas
A=[v1,v2,v3,v4];
%Imprimimos su inversa y su determinante
inversa=inv(A);
determinante=det(A);
disp(inversa);
disp(determinante);
```



```
Command Window
>> problema_6
Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND =
3.083953e-18.
> In problema_6 (line 9)

1.0e+16 *

    1.080863910568919   -1.080863910568919   -1.080863910568919    1.080863910568919
   -0.540431955284460    0.540431955284460    0.540431955284460   -0.540431955284460
    0.540431955284460   -0.540431955284460   -0.540431955284460    0.540431955284460
    1.080863910568919   -1.080863910568919   -1.080863910568919    1.080863910568919

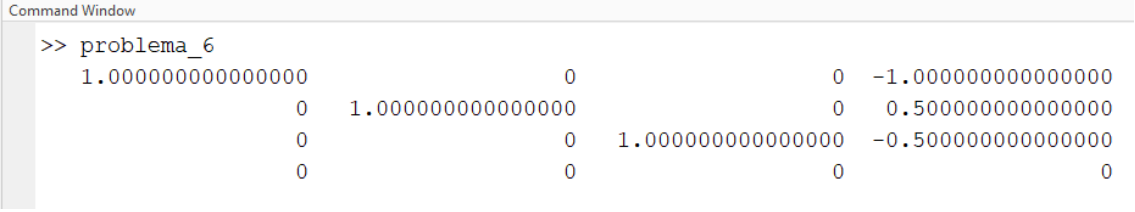
1.850371707708594e-16
```

Se obtuvo una advertencia de la misma función `inv()` que se puede consultar en la referencia. Si la matriz es mal condicionada (por esto aparece el parámetro `RCOND`) entonces el cálculo tiene imprecisión numérica, por esto la inversa obtenida tiene tales entradas, por lo tanto, no podemos concluir si son L.I. o L.D. Con el cálculo del determinante sucede lo mismo, debido al condicionamiento de la matriz obtenemos un determinante muy pequeño y con mucho error numérico, por lo que no podemos concluir si son L.I. o L.D.

Una segunda forma de atacar este problema es por medio de un teorema del álgebra lineal sobre esta misma matriz A . La matriz A es singular (lo que implicaría que los vectores son L.D.) si su rango es menor que su dimensión, es decir $\text{rank}(A) < 4$.

Para calcular $\text{rank}(A)$ llevamos la matriz A a su forma escalonada reducida por filas mediante eliminación de Gauss-Jordan usando la función `rref()`.

```
%Declarar vectores columna v1,v2,v3 y v4
v1=[0,1,0,1]';
v2=[1,2,3,4]';
v3=[1,0,1,0]';
v4=[0,0,1,1]';
%La matriz A está formada por los vectores en columnas
A=[v1,v2,v3,v4];
%Imprimimos su forma escalonada reducida por filas
escalonada_reducida=rref(A);
disp(escalonada_reducida);
```



Command Window

```
>> problema_6
    1.0000000000000000      0      0 -1.0000000000000000
           0  1.0000000000000000      0  0.5000000000000000
           0      0  1.0000000000000000 -0.5000000000000000
           0      0      0      0      0
```

Se obtuvo la matriz escalonada reducida por filas. Ya que la última fila tiene solo ceros (es L.D. con respecto a las demás) tenemos que $\text{rank}(A) = 3$ y por lo tanto la matriz A es singular, luego los vectores v_1, v_2, v_3, v_4 son linealmente dependientes (hecho que se puede demostrar, ya que $v_2 = v_3 + 2v_4 + 2v_1$).

Para todo vector v de dimensión n , usando el comando `c=poly(v)` uno puede construir el polinomio $p(x) = \sum_{k=1}^{n+1} c(k) x^{n+1-k}$ el cual es igual a $\prod_{k=1}^n (x - v(k))$. En aritmética exacta, uno puede encontrar que $v=\text{roots}(\text{poly}(v))$. Sin embargo, esto no ocurre debido a errores de redondeo, como puede checarse usando el comando `roots(poly([1:n]))`, donde n va desde 2 hasta 25.

En este ejercicio basta con operar la instrucción requerida y ver los errores por redondeo. Es necesario mostrar todos los decimales con `format long` para evitar ver el redondeo.

```
%Ingresamos el n que queremos
n=input('n: ');
%Calculamos las raices del polinomio generado
%estas raices deben ser 1,2,...,n
raices=roots(poly([1:n]));
%Mostramos todos los decimales con "format long" e imprimimos
format long;
disp(raices);
```

```
Command Window

>> problema_8
n: 2
    2
    1

>> problema_8
n: 3
 3.0000000000000002
 1.9999999999999998
 1.0000000000000000

>> problema_8
n: 4
 3.9999999999999982
 3.0000000000000039
 1.9999999999999980
 1.0000000000000002
```

```
Command Window

>> problema_8
n: 20
19.999874055724192
19.001295393676987
17.993671562737585
17.018541647321989
15.959717574548915
15.059326234074415
13.930186454760916
13.062663652011070
11.958873995343460
11.022464271003383
 9.991190949230132
 9.002712743189727
 7.999394310958664
 7.000096952230211
 5.999989523351082
 5.000000705531480
 3.999999973862455
 3.000000000444877
 1.999999999998383
 0.999999999999949
```

Es interesante ver que efectivamente no se cumple que $[1:20]=\text{roots}(\text{poly}[1:20])$. Este comportamiento se atribuye, además del error por redondeo, a la función `roots()` ya que incluso se menciona en la referencia que las raíces encontradas para el polinomio no son totalmente exactas pero están dentro del error por redondeo (que crece proporcionalmente a n).

Considera el siguiente algoritmo para calcular π . Genera n parejas $\{(x_k, y_k)\}$ de números aleatorios en el intervalo $[0,1]$, luego computa el número m de estos que caen dentro del cuarto de círculo unitario. Obviamente, π se obtiene del límite de la secuencia $\pi_n = \frac{4m}{n}$. Escribe un programa de MATLAB que compute esta secuencia y cheque el error para valores incrementales de n .

De acuerdo con lo requerido, leeremos el número n correspondiente y utilizando un ciclo `for` generaremos para cada iteración de 1 a n un par de números aleatorios con la función `rand`, que devuelve justamente un número aleatorio uniformemente distribuido en el intervalo $[0,1]$.

Luego calcularemos su distancia con el origen (evitaremos la raíz cuadrada para minimizar el error cometido por su cálculo) y si resulta ser menor o igual que 1.0, entonces está dentro del círculo unitario, en este caso sumaremos uno a una variable auxiliar que cuenta estos puntos.

Finalmente, calcularemos el valor de la sucesión para este n dado a través del número de puntos dentro, obteniendo nuestra aproximación de π . Para compararla con el π almacenado calcularemos el error absoluto y relativo, que miden la distancia entre nuestra aproximación y el π almacenado y que tan grande es esta diferencia con respecto al valor almacenado respectivamente.

```
n=input('n: ');
%La variable "dentro" es "m" en la descripción
dentro=0;
for i=1:n
    %Generamos el par (x_i,y_i)
    x=rand;
    y=rand;
    %Calculamos su distancia cuadrada con el origen
    dist=x^2+y^2;
    %Si es menor o igual a 1, (x_i,y_i) está en el círculo
    if (dist<=1.0)
        dentro=dentro+1;
    end
end
%Calculamos el valor de la secuencia para este n particular
pi_n=4*dentro/n;
%Calculamos los errores absoluto y relativo
error_abs=abs(pi_n-pi);
error_rel=error_abs/pi;
%Mostramos los resultados
format long;
disp(pi_n);
disp(error_abs);
disp(error_rel);
```

```

Command Window
>> problema_11
n: 100
3.4400000000000000

0.298407346410207

0.094986008472240

>> problema_11
n: 10000
3.1852000000000000

0.043607346410207

0.013880649472610

>> problema_11
n: 1e6
3.1422280000000000

6.353464102066830e-04

2.022370435201693e-04

Command Window
>> problema_11
n: 1e8
3.1413218800000000

2.707735897931052e-04

8.618991054861974e-05

>> problema_11
n: 1e9
3.1416494600000000

5.680641020688881e-05

1.808204196746450e-05

>> problema_11
n: 5e8
3.1416547360000000

6.208241020688732e-05

1.976144492696971e-05

```

Para $n=100$ obtenemos $\pi_{100} = 3.44$, que es un valor muy alejado con respecto al almacenado (lo podemos ver con el error absoluto de aproximadamente 0.3). Para este caso será más fácil analizar el error relativo, en este caso tenemos 0.094, esto implica una diferencia de 9.4% con el valor almacenado.

Para $n=10000$ obtenemos un valor más cercano con $\pi_{10000} = 3.1852$. Notemos que ha disminuido el error absoluto y el relativo, por tanto, nos acercamos cada vez más al valor almacenado.

Para $n=1000000$ hemos obtenido dos decimales de precisión y errores aun más bajos.

Desde este punto se nota que la sucesión converge muy lentamente al valor almacenado.

Para valores de n como 1×10^8 , 5×10^8 y 1×10^9 hemos obtenido valores cada vez más próximos al valor de π entre más grande sea n , minimizando los errores obtenidos.

Sin embargo, a partir de este punto se puede intuir que para tener una aproximación tan buena como la almacenada requeriríamos de un tiempo muy grande (hasta este entonces hemos tardado unos cuantos segundos), por lo que se concluye que este no es un método muy bueno para calcular π .

Dado que π es la suma de la serie

$$\pi = \sum_{n=0}^{\infty} 16^{-n} \left(\frac{4}{8n+1} - \frac{2}{8n+4} - \frac{1}{8n+5} - \frac{1}{8n+6} \right)$$

podemos computar una aproximación de π sumando hasta el n -ésimo término, para un n suficientemente grande. Escriba una función de MATLAB que compute las sumas finitas de la serie mostrada. ¿Qué tan grande debe ser n para obtener una aproximación de π al menos tan precisa como la almacenada en la variable π ?

En este problema se implementa una función que calcule, dado un n_{max} , la suma parcial de esta serie hasta este término y que además calcule el error absoluto con respecto a la variable π almacenada para verificar que tanto se aproxima en función del n_{max} .

Para calcular los términos se emplea un ciclo `for` iterando desde 0 hasta n_{max} , calculando la expresión por cada iteración y sumándolo. Finalmente se retorna la suma obtenida y el error absoluto.

```
%Función que calcula Pi por una serie
function [pi_aprox,err_abs]=problema_12(n_max)
    %Recibimos el termino limite por parametro en "n_max"
    %Inicializamos la suma finita en cero
    pi_aprox=0;
    %Iteramos desde 0 hasta n_max para sumar los terminos
    for n=[0:n_max]
        %Usamos la función power() para la potencia
        termino_1=power(16,n);
        %Calculamos la parte restante del termino
        termino_2=(4/(8*n+1)-2/(8*n+4)-1/(8*n+5)-1/(8*n+6));
        %Obtenemos el termino completo
        termino=termino_2/termino_1;
        %Sumamos la contribución a la suma parcial
        pi_aprox=pi_aprox+termino;
    end
    %Por ultimo calculamos el error relativo con el almacenado
    err_abs=abs(pi_aprox-pi);
    return;
end
```

```
Command Window
>> [pi_aprox,error]=problema_12(1)

pi_aprox =

    3.141422466422466

error =

    1.701871673267519e-04
```

```
Command Window
>> [pi_aprox,error]=problema_12(3)

pi_aprox =

    3.141592457567436

error =

    1.960223574570819e-07
```

```
Command Window
>> [pi_aprox,error]=problema_12(6)

pi_aprox =

    3.141592653572881

error =

    1.691224937871993e-11
```

```
Command Window
>> [pi_aprox,error]=problema_12(7)

pi_aprox =

    3.141592653588973

error =

    8.202327705930657e-13
```

```
Command Window
>> [pi_aprox,error]=problema_12(9)

pi_aprox =

    3.141592653589791

error =

    1.776356839400250e-15
```

```
Command Window
>> [pi_aprox,error]=problema_12(10)

pi_aprox =

    3.141592653589793

error =

    0
```

Se evaluó la función para distintos valores de n_{\max} , es decir, para distintos límites de las sumas parciales. Desde $n_{\max}=1$ se obtuvo una aproximación de π sorprendentemente cercana al valor almacenado (aunque el error es relativamente grande, apenas dos dígitos de precisión), mientras que para $n_{\max}=3, 6, 7$ y 9 se obtuvieron resultados cada vez más cercanos, lo cual es prueba de que el error numérico cometido es bajo y se aproxima muy rápidamente al valor almacenado de π .

Finalmente, con $n_{\max}=10$ se obtiene una aproximación con un error absoluto de 0 con respecto al π almacenado.

Escriba un programa para el computo del coeficiente binomial $\binom{n}{k} = n!/(k!(n-k)!)$, donde n y k son dos números naturales con $k \leq n$.

Una primera solución es escribir el cómputo directo con la expresión del coeficiente utilizando la función `factorial()`. Con ésta se obtienen resultados consistentes con algunos coeficientes binomiales bien conocidos.

```
%Introducimos el n y k para calcular
n=input('n: ');
k=input('k: ');
%Implementacion directa del calculo del coeficiente
respuesta=factorial(n)/(factorial(k)*(factorial(n-k)));
disp(respuesta);
```

```
Command Window
>> problema_13
n: 7
k: 3
    35

>> problema_13
n: 20
k: 5
   15504
```

```
Command Window
>> problema_13
n: 0
k: 0
    1

>> problema_13
n: 4
k: 2
    6
```

```
Command Window
>> problema_13
n: 2
k: 1
    2

>> problema_13
n: 25
k: 4
   12650
```

Una segunda solución para este problema es utilizar la definición recursiva del coeficiente binomial general dada por

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}, n, k \in \mathbb{N}$$

Con los casos especiales de $\binom{n}{0} = 1, \forall n \geq 0$ y $\binom{0}{k} = 0, \forall k > 0$. La ventaja de esta solución es que el cálculo de cada coeficiente no depende de la función factorial sino de la suma recursiva de los dos factoriales que la componen. La desventaja es precisamente que para cada coeficiente se necesita de otro par de llamadas recursivas, generando una cantidad considerable de tiempo de cómputo. Una forma de solucionar este último problema es implementando una matriz de memorización de resultados para evitar recalcular coeficientes en la recursión, sin embargo, no se implementa en el código descrito. Finalmente se comenta que al igual que la primera solución, se obtienen los mismos valores para los coeficientes requeridos.


```
%Introducción del n y k
n=input('n: ');
k=input('k: ');
%Llamado de la función recursiva
respuesta=binomial(n,k);
disp(respuesta);

%Función que calcula el coeficiente binomial por recursión
function [res]=binomial(ene,ka)
    %Caso base k=0, n>=0
    if(ka==0 && ene>=0)
        res=1;
        return;
    end
    %Caso base n=0, k>0
    if(ene==0 && ka>0)
        res=0;
        return;
    end
    %Caso distinto se define por recursión
    res=0;
    res=res+binomial(ene-1,ka-1);
    res=res+binomial(ene-1,ka);
    return;
end
```