Al trabajar con los números en el ordenador la precisión está limitada por el nº de bits en la mantisa de la representación. En precisión doble son 53 bits, lo que supone un máximo de 16 cifras significativas (2-53 ~ 10-16). En esta práctica empezaremos calculando la fracción del número e (0.71828...) en doble precisión viendo cómo la precisión real obtenida puede llegar a ser menor de 16 decimales dependiendo de cómo hagamos las operaciones. Luego exploraremos algunas estrategias para obtener algunos miles de decimales del número e.

1. Cálculo de los decimales de e

Para obtener los decimales de número e (e-2=0.71828...) usaremos la serie:

$$e=\sum_{k=0}^{\infty}\frac{1}{k!} \quad \rightarrow \quad e-2=\sum_{k=2}^{\infty}\frac{1}{k!}=\frac{1}{2\cdot 1}+\frac{1}{3\cdot 2\cdot 1}+\frac{1}{4\cdot 3\cdot 2\cdot 1}+\frac{1}{5\cdot 4\cdot 3\cdot 2\cdot 1}+\ldots$$

que converge bastante rápido debido al factorial en el denominador. Como solo queremos la fracción de e sumaremos la serie desde k=2 (eliminando así los dos primeros 1's de la suma).

Lo primero es determinar cuántos términos de esta serie tendría sentido sumar en doble precisión. Sabemos que al hacer una suma, si el término a sumar es menor que la mitad del intervalo entre números máquina el resultado ya no cambia. En este caso (intervalo entre ½ y 1) el salto entre números máquina es de 2^{-53}, por lo que la mínima contribución para provocar un cambio será de 2^{-54}. Evaluad 1/k! para diferentes valores de k y determinar hasta qué término k tendría sentido sumar de esta serie en doble precisión.

Vamos a comprobarlo sumando los N=20 primeros términos. Para ello inicializad S1=0, T=1 y haced un bucle desde k=2 a N. En cada paso se divide T por k y se suma el resultado a S1. De esta forma en T tenemos los sucesivos términos de la serie (dividiendo por k el término anterior) y en S1 su suma total. En cada paso del bucle volcad el valor de S1 obtenido usando 18 decimales (%.18f). Adjuntad vuestro volcado a partir de k=10. ¿A partir de qué término k la suma S1 ya no cambia?

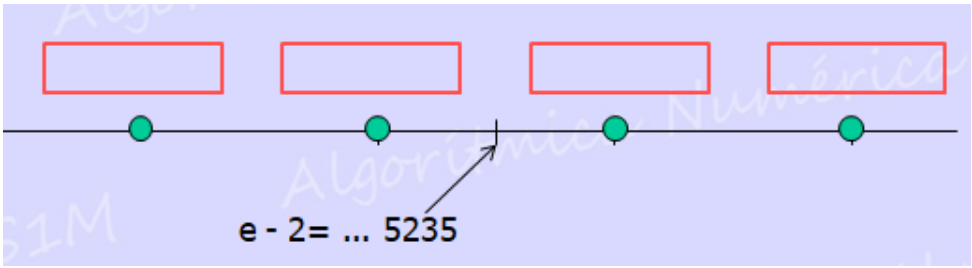
En un sitio como https://onlinetools.com/math/generate-e-digits podéis pedir los decimales que queráis (dentro de un orden) de e (que obviamente coincidirán con los de e-2). Si pedís 18 decimales obtendréis 718281828459045235.
¿Cuál es el error cometido en la suma S1? ¿Se alcanzan las 16 cifras correctas que se podían esperar de la doble precisión?

La serie anterior también puede escribirse en su forma anidada o de Horner de la siguiente forma:

$$e-2=\frac{1}{2}\left(1+\frac{1}{3}\left(1+\frac{1}{4}\left(1+\frac{1}{5}\left(1+\frac{1}{6}(1+\ldots)\right)\right)\right)\right)$$

Escrita de esta forma vemos otra posibilidad para evaluar la serie. Si os fijáis, en cada paso sumamos 1 a lo que ya tenemos y se divide el resultado por k (6, 5, 4,…,2). Si desprecio los últimos términos (...) bastaría inicializar S2=0 y en cada paso sumarle 1 y dividir el resultado por k (haciendo ahora el bucle al revés, desde k=20 hasta 2). Adjuntad el código usado y el nuevo resultado de S2 también con 18 decimales. ¿Alcanza S2 las 16 cifras correctas esperadas?

Obviamente ninguno de los resultados obtenidos con el ordenador será correcto ya que (e-2) caerá entre dos números máquina. Completad el esquema adjunto etiquetando los 4 números máquina (dos a cada lado) que rodean a (e-2) usando sus últimas 4 cifras (decimales 15 a 18). Recordad que la separación entre números máquina en el intervalo [½ ,1) es de 2^{-53}. También podéis obtener este salto en Matlab como eps(0.5).



Sobre el esquema indicad cuál de los números máquina mostrados son S1 y S2.

Calculad el resultado que da Matlab (**exp(1)-2**) y volcadlo con 18 decimales. ¿Qué nº máquina del esquema es? ¿Es más o menos preciso que S2? ¿Por qué creéis que Matlab no alcanza los 16 decimales exactos esperados al calcular e-2?

2. Cálculos en multiprecisión

Hemos visto que la doble precisión solo puede darnos (en el mejor de los casos) 16 decimales. Si queremos reproducir los resultados de una de esas páginas y calcular miles de cifras del número e habrá que usar técnicas de multi-precisión, usando números con un tamaño muy superior a los de las representaciones "nativas" del ordenador (32 o 64 bits).

En estos casos es habitual adaptar nuestros números multi-precisión al problema en cuestión. En nuestro caso usaremos números de la forma:

$$d_1 + d_2 \cdot B^{-1} + d_3 \cdot B^{-2} + \ldots$$

usando una base B, con "dígitos" d_k entre 0 y B-1. Estos números se guardarán como arrays conteniendo sus dígitos x = [d_1 , d_2 , d_3, …]. Dependiendo del número de cifras decimales que deseemos podemos ajustar la longitud del array.

Por comodidad B será una potencia de 10 para no tener que hacer cambios de base al presentar los resultados. Nosotros usaremos $B=10^6=1000000$, de forma

que cada "dígito" de esta representación sea un número entre 000000 y 999999 que nos dará 6 cifras decimales. El no usar un B aún más grande es para evitar que al hacer operaciones (por ejemplo al multiplicar 2 "dígitos") el resultado sea mayor que el máximo entero que es posible manejar en precisión doble. De esta forma garantizamos que las operaciones realizadas sean exactas.

Una vez decidida la base, el tamaño L del array escogido para nuestros números (longitud de la mantisa) vendrá dado por las cifras decimales deseadas. Si por ejemplo queremos trabajar con 6000 cifras decimales usaríamos un array con un mínimo de 1+6000/6 =1001 casillas (la 1ª casilla corresponde a las unidades y las otras 1000 nos permiten guardar 1000·6=6000 decimales).

Para el cálculo usaremos la segunda forma de hacer la suma que vimos en el apartado anterior. Es más precisa y solo requiere mantener un número multi-precisión (S) e implementar una única operación. Esto es muy importante porque al usar nuestros propios "números" hay que implementar en software todas las operaciones que vayamos a necesitar. En este caso la única operación a implementar es (S/k), la división de un número multi-precisión S por k. El divisor k será un único "dígito" (k<B) si nos limitamos a sumar menos de un millón de términos de la serie. En el algoritmo también hay que sumar 1 a S (antes de dividir por k), pero ésta es una operación trivial: basta con sumar 1 a S(1), la casilla que corresponde a las unidades.

La división de un número "largo" por un "dígito" equivale a hacer una operación como 364785/7 y su implementación es totalmente similar al de la división que aprendisteis en el colegio. Escribiremos una función **`function x=dividir(x,d)`** que recibe un array de multi-precisión x y un dígito d<B y devuelve su división en el mismo array x, siguiendo este pseudocódigo:

- Inicializamos acarreo=0
- Hacer un bucle desde k=1 (dígito más significativo) hasta L (longitud de x) y en cada paso:
  - A = x(k)+acarreo*B
  - Calculad el cociente entero y resto de la división A/d. El cociente se guarda en x(k) y el resto es el acarreo para el siguiente paso.

Para la división entera usad fix() que devuelve la parte entera de un número. Para obtener el resto, restad a A el producto de d por el cociente entero obtenido

Probad vuestra división creando un array de tamaño L=5, con un 1 en la 1ª casilla y 0 en el resto (equivalente a 1.000…). Divididlo por 81 y volcad el array resultante. Comparadlo la fracción **`1/81 = 0.012345679012345679012345679...`** Adjuntad el código de vuestra función de división.

Recordad que cada casilla de S aporta 6 cifras: si en dos casillas consecutivas obtenemos 123 y 45678 debemos interpretarlas como 000123 y 045678. Por esta razón, para volcar los decimales obtenidos tras calcular S puede ser más conveniente agrupar los dígitos (solo los decimales, ignorando el contenido de S(1)) en una única cadena de texto haciendo **`txt=sprintf('%06d',S(2:end));`**

La función **sprintf()** es muy similar a **fprintf()** pero su salida se vuelca a una cadena de texto en vez de sacarla por pantalla. El descriptor **%06d** hace que cada casilla se vuelque con 6 dígitos (si es necesario precedidos de 0's). Una vez que tenemos los decimales en una cadena de texto podéis volcarlos usando (%s) usando por ejemplo **fprintf('%s\n',txt);** o **fprintf('%s\n',txt(1:100));** para volcar sólo los 100 primeros. Calculad con vuestra función el resultado de dividir 12.3456789 por 77 usando un array de tamaño 5 y volcad los primeros 24 decimales del resultado, que deben empezar por 0.160… Adjuntad código usado.

Con esta función tenemos lo necesario para calcular un montón de los decimales de e. Al igual que cuando trabajamos en doble precisión, antes de ponernos a operar determinaremos cuántos términos hay que sumar de la serie para obtener Ndec decimales correctos en el resultado final. Para asegurar Ndec decimales correctos podemos parar cuando el último término 1/N! sea del orden de 10^{-Ndec} :

$$\frac{1}{N!} \approx 10^{-Ndec} \rightarrow N! \approx 10^{Ndec} \rightarrow Ndec \approx \log_{10}(N!)$$

El siguiente término 1/(N+1)! sería N+1 veces más pequeño que 10^{-Ndec} y como N será grande (del orden de cientos o miles) ya no cambiaría el decimal Ndec.

El problema para estimar Ndec cuando sumamos un número N alto de términos es que al calcular N! es fácil que nos salgamos del rango de números que maneja Matlab. Afortunadamente como lo que aparece en la fórmula es el logaritmo de N! podemos aplicar el hecho de que:

$$\log_{10}(N!) = \log_{10}(1 \cdot 2 \cdot 3 \cdot \ldots \cdot N) = \log_{10}(1) + \log_{10}(2) + \ldots + \log_{10}(N) = \sum_{k=1}^{N} \log_{10}(k)$$

Esta expresión nos da el número de decimales correctos (Ndec) que esperamos obtener al sumar N términos de la serie. Tras determinar Ndec, el tamaño L del número multi-precisión S a usar será de 1+Ndec/6 elementos (cada dígito en base B aporta 6 cifras decimales y se necesita una casilla adicional para manejar las unidades). Redondear hacia arriba (ceil) el tamaño L encontrado por si Ndec no fuese múltiplo exacto de 6.

En nuestro caso sumaremos N=10000 términos de la serie, ¿cuántos decimales correctos podemos esperar? ¿De qué tamaño L debe ser el array multi-precisión usado para poder trabajar con ese número de decimales?

Tras determinar L, inicializad un array S de ese tamaño relleno con ceros (lo que corresponde a un valor de 0.000000 ...) y haced el bucle desde k=N hasta 2 aplicando las operaciones que hicimos antes en cada paso (sumar 1 a S y dividir el resultado por k). Al final del bucle volcad los primeros 4 elementos de S para comprobar que se ha calculado correctamente el resultado, que debería ser:

S = 0, 718281, 828459, 045235, …

Adjuntad código del bucle. Si la comprobación inicial es correcta, pasaremos a verificar el número exacto de decimales correctas obtenidas. Para ello, volcad los decimales obtenidos en una cadena de texto como se ha indicado antes. Si hacéis **`>>load digits_e`** veréis una variable llamada e que es una cadena de texto con el primer millón de decimales de e. Comparar vuestros resultado con estos dígitos hasta ver cuando empiezan a diferir. ¿Número de decimales correctos en el resultado? ¿Coincide con los que habíais previsto (Ndec)?

Dad los tiempos que tarda programa en sumar N=10000, 50000 y 100000 términos de la serie, indicando en cada caso los decimales correctos obtenidos.

	N=10000	N=50000	N=100000
Tiempo			
N dec correctos			

¿Cómo crece el tiempo (N, N^2, N^3, ...) al aumentar el nº de términos N sumados?

En 2004 Google levanto unas vallas publicitarias con objeto de atraer a posibles programadores a una página web (7427466391.com) donde podían iniciar un proceso de selección. 7427466391 es el primer primo de 10 cifras que se encuentra en los decimales de e, empezando en la posición 99. Se trata de que encontréis los 2 primeros primos de 15 cifras que aparecen en los decimales de e.

Adjuntad código usado, los primos encontrados y decimal en el que empiezan.

Os resultará útil la función isprime(p) que os dice si el número p es o no primo y la función str2double() que convierte una cadena de caracteres en un número.

Uso de una nueva serie para e

La siguiente serie (ligeramente distinta) nos da otra alternativa para el cálculo de los decimales del número e:

$$e-1=\sum_{k=1}^{\infty}\frac{2k+1}{(2k)!}=\frac{3}{2\cdot 1}+\frac{5}{4\cdot 3\cdot 2\cdot 1}+\frac{7}{6\cdot 5\cdot 4\cdot 3\cdot 2\cdot 1}+\ldots$$

Esta serie converge incluso más rápido que la anterior debido al (2n)! en el denominador.

Observad que esta serie calcula e-1 y no e-2, pero trabajando con nuestros números en multi-precisión la única diferencia es que la casilla S(1) (unidades) terminará con un 1 en vez de un 0, obteniéndose igualmente los decimales de e en el resto de las casillas de S.

Se pide:

a) Reescribir esta serie en su forma anidada, para sumarla como se ha hecho antes con un bucle inverso, empezando en k=N y bajando. Adjuntad captura de la expresión de la serie anidada.

b) Comprobad que vuestro algoritmo es correcto implementando esta suma para N=10 en doble precisión. El resultado debería ser 1. 718281828459045313. Adjuntad código usado para el bucle.

c) Para esta serie, deducir la relación entre N (términos sumados) y Ndec (nº decimales correctos obtenidos). Adjuntad captura de vuestra deducción.

d) Implementad la suma (usando 10000 términos) de la serie en multiprecisión (no se necesitan funciones adicionales a las que ya tenemos). Adjuntad código del bucle y comparar los decimales exactos conseguidos con la predicción.

3. Algoritmos "spigot"

Los algoritmos implementados antes tienen el problema de que hay que esperar a que termine el bucle para obtener el resultado final. Si nos aburrimos y lo paramos antes de completarse el bucle no habremos terminado de sumar todos los términos y S no contendrá el resultado correcto.

En este apartado presentamos unos algoritmos denominados "spigot" (grifo o espita). Estos algoritmos van sacando ("goteando") los sucesivos dígitos (en este caso de e) según avanzan en sus cálculos. De esta forma si nos cansamos de esperar tendremos al menos las cifras obtenidas hasta entonces. Además, como nos vamos desprendiendo de los dígitos que van saliendo, no hace falta trabajar con números multiprecisión como antes. Lo único que se requiere es un array de enteros de tamaño N (siendo N el número de términos a sumar en la serie).

Estos algoritmos (refs) se basan en los métodos de cambio de base. Recordad el algoritmo que vimos en clase para convertir un número decimal a base 2. El número se multiplicaba repetidamente por 2 y el algoritmo iba "escupiendo" los sucesivos bits de la versión en base 2 del número inicial. Este algoritmo puede generalizarse para cualquier otro cambio de base. Considerad el número en base 7 dado por $(0.3561)_7$ que en decimal correspondería a:

$$(0.3561)_7 = \frac{3}{7} + \frac{5}{7^2} + \frac{6}{7^3} + \frac{1}{7^4} = 0.5485...$$

El algoritmo para obtener los decimales (5, 4, 8, 5,...) en base 10 a partir de los "septimales" (3, 5, 6, 1) en base 7 consiste en multiplicar repetidamente por 10 para ir sacando los dígitos en base 10. Se parte de un array A = [3 5 6 1] con los dígitos iniciales (base 7) y a partir de ahí:

- Se multiplican todos los números de A por 10 (la base de destino).

- Tras esta multiplicación cada casilla puede contener dígitos muy grandes, que hay que "reducir" para que vuelvan a ser compatibles con la base 7 (menores que 7). Para ello se recorre el array de derecha a izquierda y en cada casilla k, calculamos el cociente entero q y resto r de A(k)/7. El resto r (<7) se deja en A(k) y el cociente q se suma a la casilla A(k-1).

- El cociente final obtenido en la casilla A(1) (que ya no puede pasarse a una casilla anterior) será el primer dígito (en base 10) de la conversión.

En la tabla siguiente se muestra el proceso detallado para el array [3 5 6 1], que tras multiplicarse por 10 queda como [30 50 60 10]. A continuación se muestran los resultados de calcular cociente y resto (mod 7) para cada casilla empezando por la última:

	7	**7**	**7**	**7**	**q,r**
A	30	50	60	10	10/7 --> q=1, r=3
A	30	50	61	3	61/7 --> q=8, r=5
A	30	58	5	3	58/7 --> q=8, r=2
A	38	2	5	3	38/7 --> q=5, r=3
A	3	2	5	3	Dígito out = 5

El vector A queda como [3 2 5 3] y el cociente q=5 obtenido en la última casilla es el 1er digito (0.**5**4...) de la fracción (base 10). Si se desean dígitos adicionales basta repetir este proceso (multiplicación de A por 10 + reducción). Tras una 2ª pasada los contenidos de A serían [5 6 5 2], siendo el último cociente que sale fuera q=4 (0.5**4**85...).

Para ver la relación de estos algoritmos de cambio de base con el problema de extraer decimales de e, escribiremos la expresión anterior del número $(0.3561)_7$ en base 7 de una forma un poco diferente:

$$(0.3561...)_7 = \frac{3}{7}+\frac{5}{7^2}+\frac{6}{7^3}+\frac{1}{7^4}+...=\frac{1}{7}\left(3+\frac{1}{7}\left(5+\frac{1}{7}\left(6+\frac{1}{7}(1+...)\right)\right)\right)$$

Mientras que la expresión de la serie anidada para (e-2) era:

$$e-2=\frac{1}{1\cdot 2}+\frac{1}{1\cdot 2\cdot 3}+\frac{1}{1\cdot 2\cdot 3\cdot 4}+...=\frac{1}{2}\left(1+\frac{1}{3}\left(1+\frac{1}{4}\left(1+\frac{1}{5}(1+...)\right)\right)\right)$$

Vemos que la serie de (e-2) tiene el mismo aspecto que la expresión de arriba. La diferencia es que ahora no se usa siempre la misma base 7 en el desarrollo, sino que cada término tiene un denominador distinto (2, 3, 4, 5,...). A esto se le llama una base "mixta", donde en vez de limitarnos a potencias de un único número ($1/7, 1/7^2$, ...), podemos mezclar números distintos. Notad que en base

10 la fracción de (e-2) es **0.71828182**..., con infinitos decimales que a todos los efectos parecen aleatorios. Sin embargo en esta nueva base mixta su expresión es trivial, **(e-2) = 0.11111111**..., siendo 1 todos sus dígitos.

El mismo algoritmo explicado para pasar de base 7 a base 10 nos vale ahora con esta base mixta con un mínimo cambio. El array A se inicializa ahora con 1's (los "dígitos" de e-2 en esta base mixta) y como antes, en cada paso se multiplica A por 10 y se reducen los valores obtenidos a la base original. La única diferencia es que al regularizar A(k) hallando su cociente q y resto r, en vez de dividir siempre por 7, usamos un número distinto 2, 3, 4, … (k+1) en cada columna. Por ejemplo si partimos de A=[1 1 1 1] (usando 4 dígitos en la nueva base), el proceso de reducción (tras multiplicar por 10) sería el siguiente

Base	**2**	**3**	**4**	**5**	**q,r**
A	10	10	10	10	10/5 --> q=2, r=0
A	10	10	12	0	12/4 --> q=3, r=0
A	10	13	0	0	13/3 --> q=4, r=1
A	14	1	0	0	14/2 --> q=7, r=0
A	0	1	0	0	Dígito out =7

El último cociente (7) es el 1er decimal de la fracción buscada (0.**7**182...). Si multiplicáis por 10 los valores finales de A y los volvéis a reducir, obtendríais A=[1 1 0 0], siendo el último cociente obtenido q=1 (0.7**1**82...).

Implementad este algoritmo arrancando con **A=ones(1,6)**. Al final de cada paso volcad los valores finales de A y el dígito extraído (el cociente q de A(1)). Algo así como:

```
Paso  1: A=[  0  1  0  1  5  3 ] -> digito out = 7
...
```

Adjuntad código usado y volcado obtenido haciendo 10 pasadas. ¿Son correctos los 10 dígitos obtenidos? ¿Cuál es el problema?

Para obtener más decimales correctos hay que partir de un array A inicial de 1's más grande, ya que el tamaño del array N corresponde al número de términos sumados en la serie. Partir de un array inicial de tamaño N=10000 lleno de 1's e ir generando dígitos. ¿A partir de qué dígito el algoritmo empieza a dar decimales incorrectos? ¿Obtenemos el mismo número de decimales que al sumar N=10000 términos de la serie usando la multi-precisión del apartado anterior?

Una modificación sencilla para obtener el mismo número de dígitos con menos pasadas es multiplicar en cada paso el array A por una base más grande como p.e. B=10000. Ahora cada "dígito" extraído sería un número entre 0000 y 9999, obteniendo así 4 dígitos decimales en cada paso. Si partimos de un array con 100000 1's, ¿cuántos decimales correctos podríamos esperar?
Usando $B=10^{10}$, ¿cuántas pasadas tendríamos que dar para estar seguros de haber extraído todos los decimales correctos posibles? Justificad vuestras respuestas. *No hace falta implementarlo, basta responder a las preguntas.*

4. Aproximación a la función √x

Hemos visto en clase que hay situaciones donde buscamos justo lo opuesto de estos métodos multi-precisión. Queremos un cálculo rápido, aunque tengamos una menor precisión de la que nos podría dar la doble precisión del ordenador.

Explicamos en clase un algoritmo para obtener una aproximación a $y=1/\sqrt{x}$ usando la interpretación como un entero Ix de un número en precisión simple x:

$$\mathbf{Iy \sim (3/2)\cdot 127\cdot 2^{23} - Ix/2 = 0x5F400000 - Ix/2}$$

Interpretando el entero Iy como un número real, se obtenía una aproximación y0 a $y=1/\sqrt{x}$. Esta aproximación podía mejorarse aplicando una o dos iteraciones adicionales usando el método de Newton.

Siguiendo el mismo razonamiento, deducir la expresión que usarías para obtener una primera aproximación y0 a la raíz cuadrada √x usando esa técnica. Adjuntad captura de vuestra deducción (la relación entre el entero Iy correspondiente a $y=\sqrt{x}$ y el entero Ix asociado al número inicial x).

En el código original aparecía un número "mágico" (**0x5F3759DF = 1597463007**) obtenido modificando el valor original **0x5F400000** tratando de minimizar el error cometido en la aproximación y0. Se trata de modificad vuestro propio número "mágico" para obtener el menor error posible en vuestra aproximación inicial a √x. Para ello haced la gráfica del error relativo cometido en el intervalo [1,2] y modificad la constante que aparece en la relación encontrada para obtener el menor error posible. Adjuntad vuestra elección, gráfica del error relativo y error máximo que tenéis.

Una vez que dispongáis de una primera aproximación y0 la podemos refinar usando el método de Newton. Preguntar a vuestra IA favorita como aproximar iterativamente la raíz cuadrada de un número. Aplicad un par de iteraciones de la iteración sugerida a la aproximación inicial obtenida. Adjuntad código de la iteración usada.

Adjuntad una gráfica superponiendo el error relativo de y0 (aproximación inicial), y1 (1ª iteración a partir de y0) e y2 (2ª iteración a partir de y1) para un rango de valores de x entre $0.001=10^{-3}$ y $1000=10^{3}$. Usar una escala logarítmica en ambos ejes para mostrar estos 3 errores. En las transparencias podéis encontrar código para crear estas gráficas.

Adjuntad todo el código usado en este apartado