MCOC2021-P0

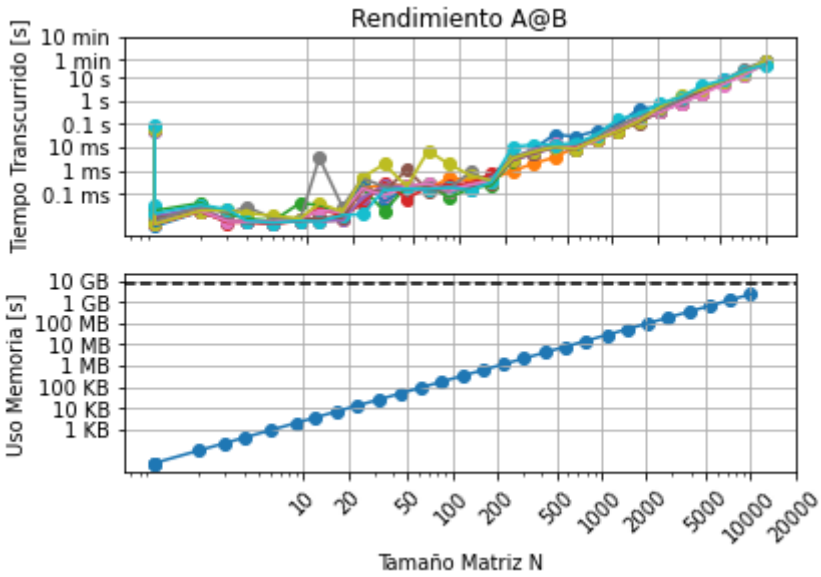
Mi computador principal

- Marca/modelo: Apple MacBook Air
- Tipo: Notebook
- Año adquisición: 2018
- Procesador:
    - Marca/Modelo: Intel Core i5 de Doble Núcleo
    - Velocidad Base: 1.80 GHz
    - Velocidad Máxima: 2.90 GHz
    - Numero de núcleos: 2
    - Humero de hilos: 1
    - Arquitectura: x86_64
    - Set de instrucciones: Intel® SSE, Intel® SSE2, Intel® SSE3 e Intel® SSE4
- Tamaño de las cachés del procesador
    - L1: 256KB
    - L2: 256KB
    - L3: 3MB
- Memoria
    - Total: 8 GB
    - Tipo memoria: DDR3
    - Velocidad 1600 MHz
    - Numero de (SO)DIMM: 2
- Tarjeta Gráfica
    - Marca / Modelo: Intel HD Graphics 6000
    - Memoria dedicada: 1536 MB
    - Resolución: 1440 x 900
- Disco 1:
    - Marca: APPLE SSD SM0128G
    - Tipo: SSD
    - Tamaño: 121 GB (128 GB)
    - Particiones: 2 (EFI + System)
    - Sistema de archivos: FAT32
- Dirección MAC de la tarjeta wifi: d0:81:7a:c9:a0:fe
- Dirección IP (Interna, del router): 192.168.100.1

- Dirección IP (Externa, del ISP): 181.43.154.109
- Proveedor internet: Entel Chile S.A.

P0E2: Desempeño MATMUL

A continuación, se presenta el rendimiento de mi PC al calcular multiplicaciones de matrices de diferentes tamaños, realizando diez corridas. Se grafica el tiempo transcurrido en segundos que demora en realizar cada operación versus el tamaño de la matriz en el primer subplot. En el segundo, se grafica el uso de memoria versus el tamaño de la matriz, además de la memoria RAM del PC.



1. ¿Cómo difiere del gráfico del profesor/ayudante?

Viendo el gráfico "tamaño de matriz vs tiempo transcurrido", se nota cómo mi computador se demoró más que el del profesor en realizar las corridas, especialmente los últimos puntos correspondientes a matrices cuadradas de dimensión 10000, donde se alcanzaron tiempos alrededores de 1 minuto, mientras que el profesor se demoró 30 segundos máximo aproximadamente. Por otro lado, el gráfico de uso de memoria es idéntico al del profesor, ya que, independientemente del computador que se tiene, las matrices que se arman ocupan el mismo espacio.

2. ¿A qué se pueden deber las diferencias en cada corrida?

Estas diferencias se deben a múltiples factores. Uno de estos es el hecho de que, en mi computador, se tienen se tienen diferentes procesos que se están realizando a la vez, los cuales influyen en el tiempo en que se ejecuta el código. Otro factor puede ser el hecho de que python, al usar la memoria, tiene el siguiente orden: caché, RAM, disco. Cada vez que necesita usar más memoria, tiene que ir de una fuente a otra, y estos cambios de memoria son variables, generando diferencias entre los tiempos de ejecución de cada corrida del código.

3. El gráfico de uso de memoria es lineal con el tamaño de matriz, pero el de tiempo transcurrido no lo es ¿porqué puede ser?

A medida que va aumentando el tamaño de la matriz, también sube el uso de memoria. Este incremento es lineal debido a que cada dimensión extra que se le agrega a la matriz ocupa el mismo espacio. Por otro lado, a diferencia del uso de memoria, el tiempo transcurrido no tiene un comportamiento lineal, debido a que, al aumentar el tamaño de la matriz, también se incrementa la cantidad de operaciones a realizar pero de manera exponencial, lo cual genera que el PC necesite un tiempo exponencialmente mayor para efectuar las operaciones. Un ejemplo que explica este fenómeno es el siguiente: se quiere multiplicar dos matrices de 2x2 y otras dos de 4x4. Las matrices 2x2 ocupan un espacio equivalente a un cuarto (aproximadamente) al que ocupan las matrices 4x4, y esto es solo debido a su tamaño, es decir, el uso de memoria es lineal con el tamaño de la matriz. Sin embargo, la cantidad de operaciones a realizar al multiplicar las matrices 2x2 es 12, mientras que las de 4x4 es 112. Este incremento es exponencial lo cual provoca que el tiempo necesario para realizar dichas operaciones también aumente exponencialmente.

4. ¿Qué versión de python está usando?

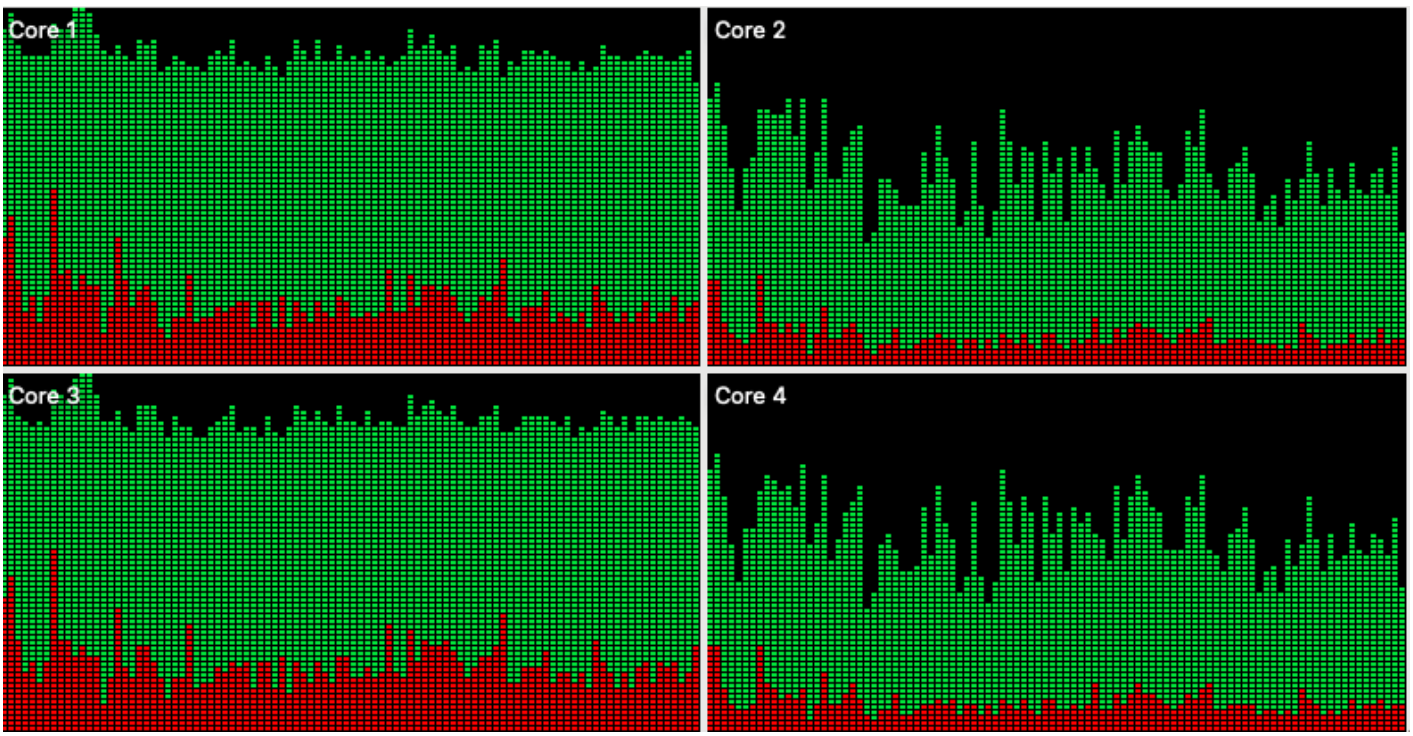
Estoy usando Python 3.8.5

5. ¿Qué versión de numpy está usando?

Estoy usando Numpy 1.19.2

6. Durante la ejecución de su código ¿se utiliza más de un procesador? Muestre una imagen (screenshot) de su uso de procesador durante alguna corrida para confirmar.

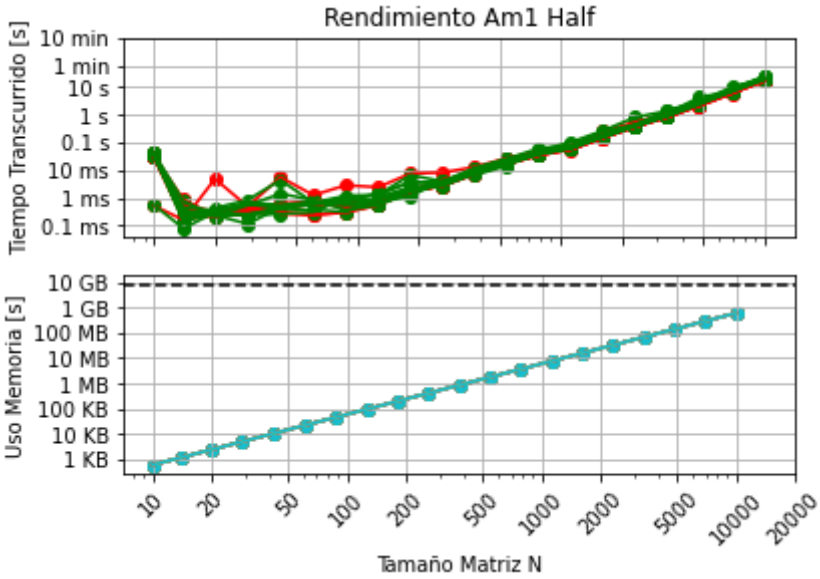
A continuación, se presenta una imagen que muestra el uso del procesador, donde se ven los 4 núcleos y su actividad.

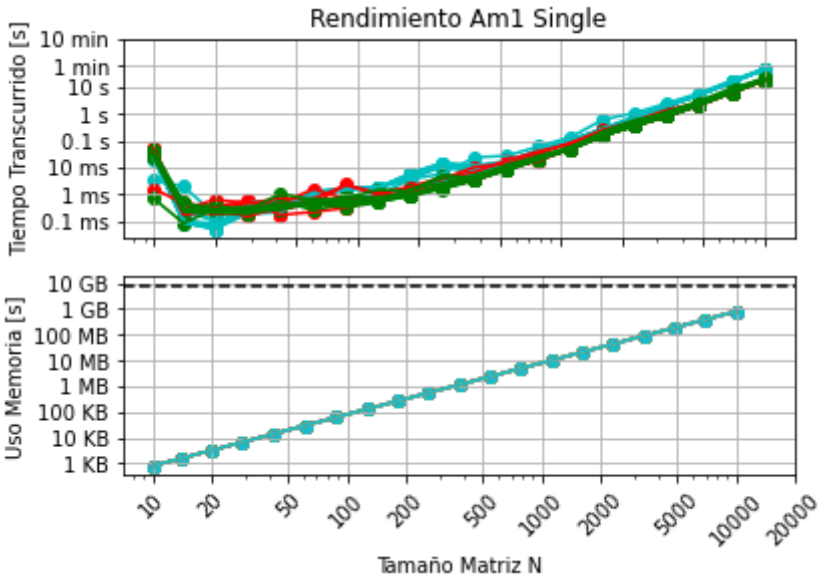


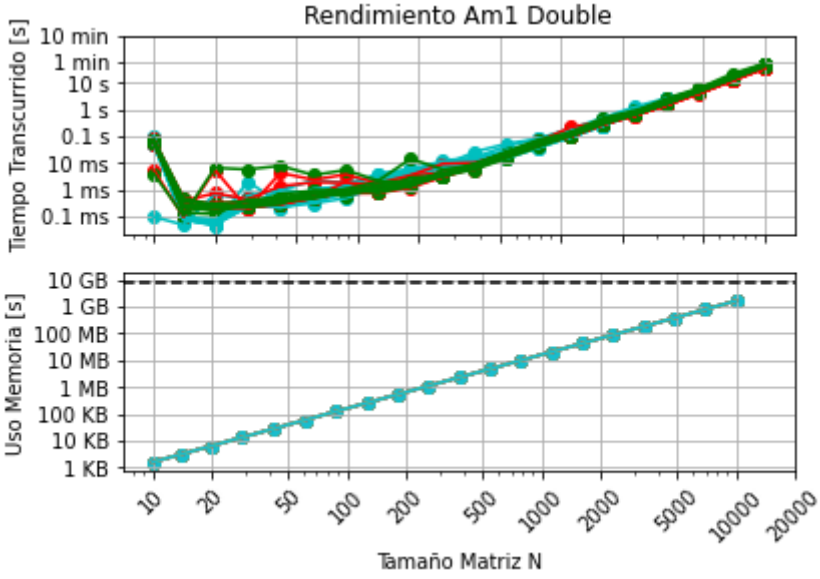
P0E3: Desempeño INV

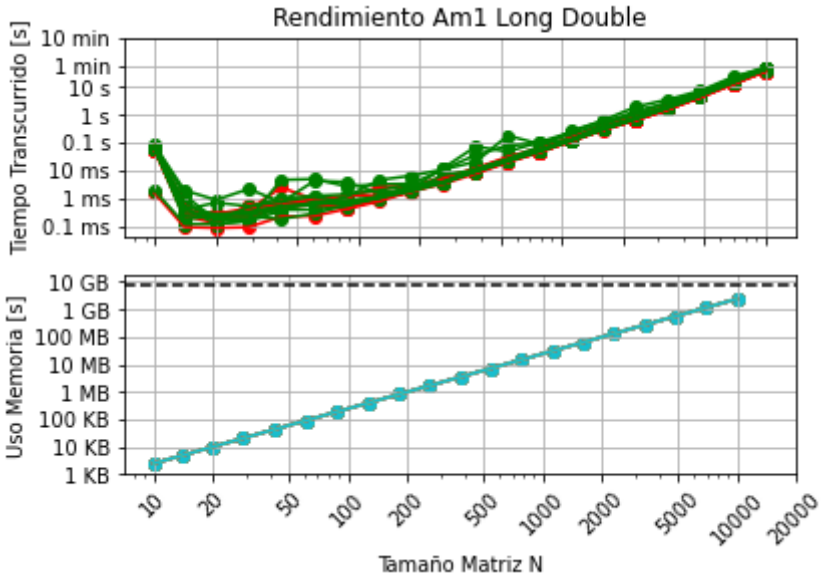
El objetivo es medir el tiempo transcurrido y el uso de memoria al invertir matrices de varias dimensiones usando numpy y scipy (con overwrite_a= True y False), es decir, 3 casos. Para cada uno, se usa 4 tipos de datos diferentes: half (el cual no es soportado por numpy), single, double y longdouble (tampoco lo soporta numpy). Con esto, se generan 12 archivos .txt con 10 corridas en cada uno.

A continuación, se presentan cuatro gráficos correspondientes al rendimiento de mi pc en invertir las matrices, donde cada uno corresponde a un tipo de dato, y contiene a los tres casos mencionados.









Es posible observar que las iteraciones realizadas por numpy (color cyan) se demoraron más que las realizadas por scipy (rojo y verde). Por otro lado, el uso de memoria es el mismo en todos los gráficos, debido a que se utilizaron las mismas dimensiones de matrices para cada caso.

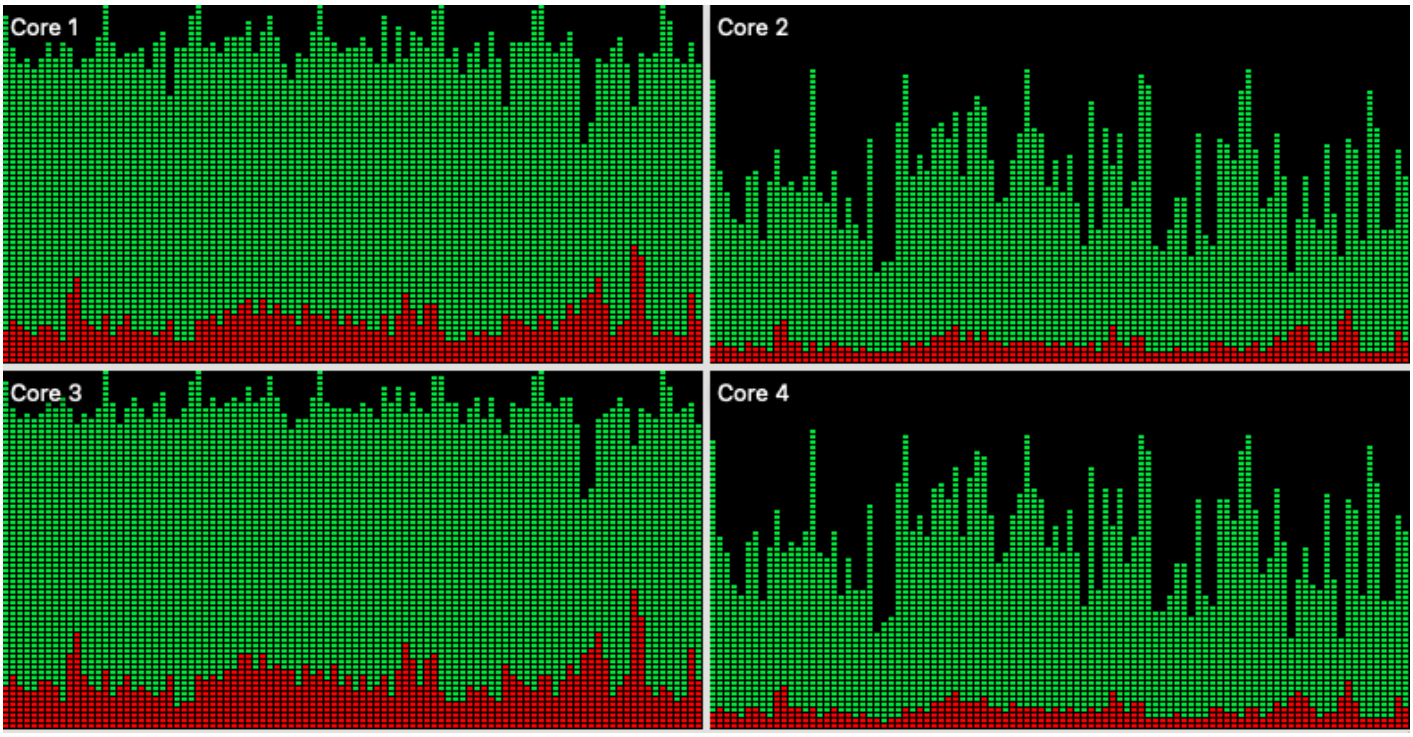
Además, se encuentra el porcentaje de memoria que fue usado durante las corridas, lo cual siempre se encontraba entre el 60% y 65%, y también la actividad del procesador.

```
0.0004165270001976751, 0.0004975570009264629, 0.00082896099956997 2,
0.0007650070001545828, 0.0015623489998688456, 0.0021347540005081 1,
0.0038349999995261896, 0.006857970000055502, 0.02182706900021003, 0.05210024000007252,
0.11670368500017503, 0.31806504799897084, 0.691835252000601, 1.7645910990013363,
4.80009174100087, 13.991642520999449, 49.81801568899937] s
Uso Memoria: [1600, 3136, 6400, 13456, 28224, 59536, 123904, 258064, 535824, 1106704,
2298256, 4752400, 9834496, 20358144, 42146064, 87235600, 180526096, 373571584,
773062416, 1600000000] bytes

Corrida Nº 4
Tiempo inversión: [0.0608873869987292, 0.0002446630005579209, 0.0002412729991192463 8,
0.0002184389995818492, 0.0002957270007755141 7, 0.000722131000657100 2,
0.0005696750013157725, 0.001293620000069495 3, 0.001715308000711957,
0.0032840889998624334, 0.006934715000170399 4, 0.02418611300163320 3, 0.05593043099906936
0.10607323600015661, 0.30139126500034763, 0.6726887699987856, 1.7208040449986584,
5.062993761999678, 16.785912879000534, 48.5863981980001] s
Uso Memoria: [1600, 3136, 6400, 13456, 28224, 59536, 123904, 258064, 535824, 1106704,
2298256, 4752400, 9834496, 20358144, 42146064, 87235600, 180526096, 373571584,
773062416, 1600000000] bytes

Corrida Nº 5
Tiempo inversión: [0.05727816199942026, 0.0001522839993413072, 0.000217360999158 7,
0.0002145039998140419, 0.0004217059995426098, 0.0005115589992783498,
0.000788755998655688, 0.001036281999404309 3, 0.001581795999300084,
0.0031608170011168113, 0.007570163001219043 5, 0.02045496599930629 6, 0.05231578900020395
0.10009033900132636, 0.3003365490003489, 0.6779345510003623, 2.208232841998324,
5.192332134000026, 15.591834895998545, 50.18148091200055] s
Uso Memoria: [1600, 3136, 6400, 13456, 28224, 59536, 123904, 258064, 535824, 1106704,
2298256, 4752400, 9834496, 20358144, 42146064, 87235600, 180526096, 373571584,
773062416, 1600000000] bytes

Corrida Nº 6
```



Preguntas:

1. ¿Qué algoritmo de inversión cree que utiliza cada método (ver wiki)? Justifique claramente su respuesta.

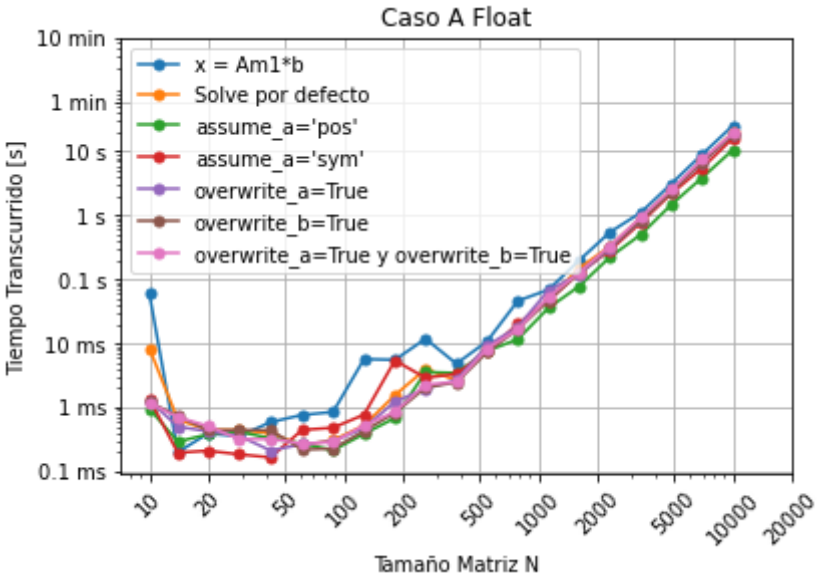
Tras realizar una investigación en internet, se llegó a que numpy llama a numpy.linalg.solve(A,I), donde A es la matriz a invertir e I es la matriz identidad. Y este solver resuelve usando la factorización LU de Lapack. En otras palabras, numpy llama a la descomposición LU de manera indirecta. Por otro lado, scipy llama de inmediato a LU, haciéndolo más rápido que numpy, lo cual se observó en los gráficos mostrados anteriormente.

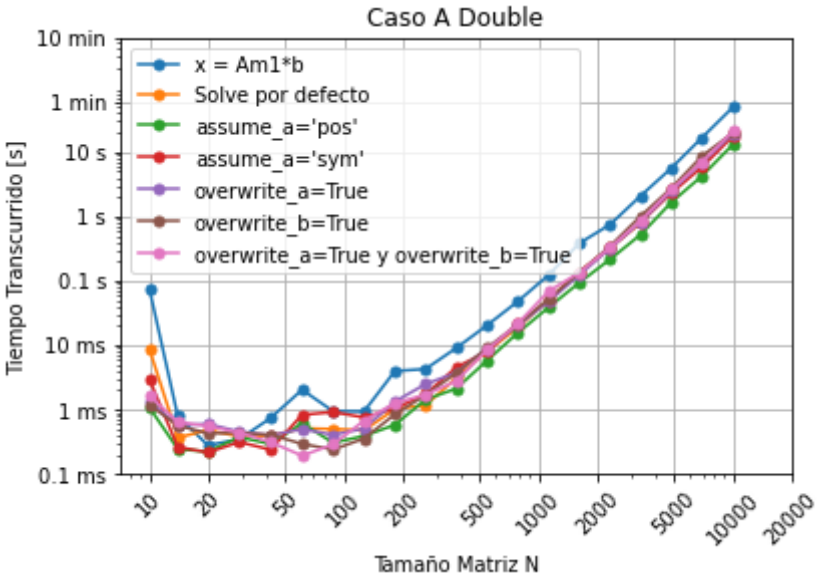
2. ¿Cómo incide el paralelismo y la estructura de caché de su procesador en el desempeño en cada caso? Justifique su comentario en base al uso de procesadores y memoria observado durante las corridas.

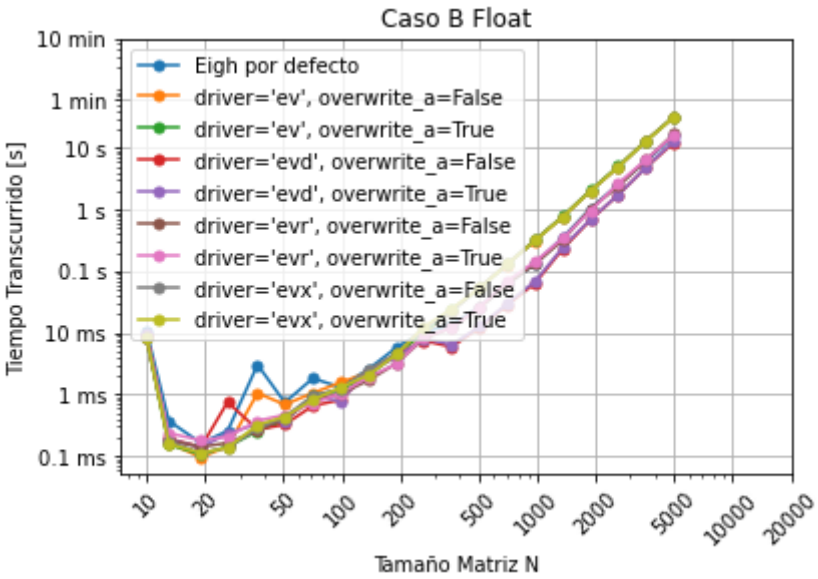
Observando los gráficos, se ve que el half toma menos tiempo en invertir la matriz, dado que tiene datos más pequeños que los single, double y longdouble. Estos cambios en el tiempo entre estos tipos es debido al paralelismo, en donde el procesador ejecuta varias tareas al mismo tiempo, realizando varios cálculos simultáneamente. Mirando el uso de memoria durante la ejecución del código, este oscilaba entre 60% y 65%, lo cual es relativamente bajo considerando las tareas que se estaban realizando, lo cual, nuevamente es debido al paralelismo y a la memoria caché. Además, los núcleos 2 y 4, los cuales se ven en la imagen de actividad del procesador, tienen menos dificultad durante la ejecución, dado que se ve menos actividad, lo cual tiene que ver con lo mencionado.

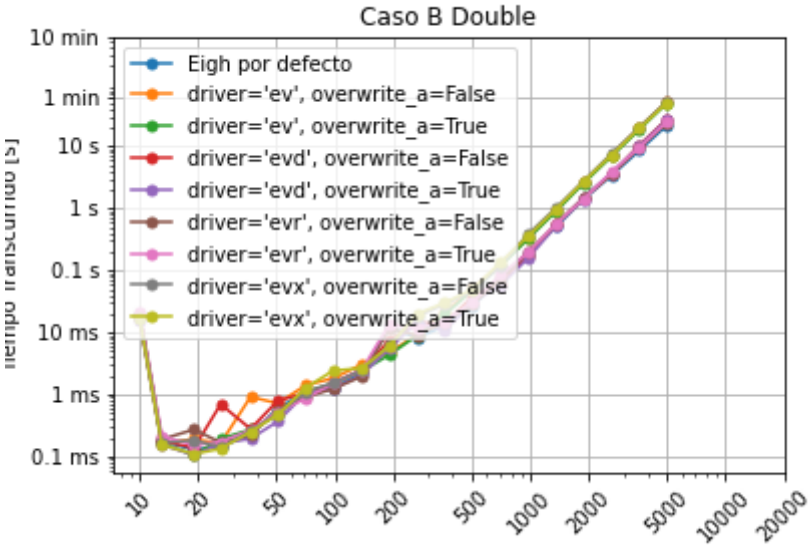
P0E4: Desempeño de SOLVE y EIGH

Se realizan dos procedimientos: "A" corresponde a la resolución de un sistema de ecuaciones con el solve de scipy y "B" a la obtención de valores y vectores propios de una matriz con eigh de scipy. Cada uno se desarrolla con diferentes casos de solve y eigh y con dos tipos de datos (float y double) y se mide el tiempo que demora el computador en ejecutar dichos cálculos. Los resultados son graficados y se presentan a continuación.


Caso A Float
Tiempo Transcurrido [s]
10 min
1 min
10 s
1 s
0.1 s
10 ms
1 ms
0.1 ms
x = Am1*b
Solve por defecto
assume_a='pos'
assume_a='sym'
overwrite_a=True
overwrite_b=True
overwrite_a=True y overwrite_b=True
10
20
50
100
200
500
1000
2000
5000
10000
20000
Tamaño Matriz N


Caso A Double
Tiempo Transcurrido [s]
10 min
1 min
10 s
1 s
0.1 s
10 ms
1 ms
0.1 ms
x = Am1*b
Solve por defecto
assume_a='pos'
assume_a='sym'
overwrite_a=True
overwrite_b=True
overwrite_a=True y overwrite_b=True
10
20
50
100
200
500
1000
2000
5000
10000
20000
Tamaño Matriz N


Caso B Float
Tiempo Transcurrido [s]
10 min
1 min
10 s
1 s
0.1 s
10 ms
1 ms
0.1 ms
Eigh por defecto
driver='ev', overwrite_a=False
driver='ev', overwrite_a=True
driver='evd', overwrite_a=False
driver='evd', overwrite_a=True
driver='evr', overwrite_a=False
driver='evr', overwrite_a=True
driver='evx', overwrite_a=False
driver='evx', overwrite_a=True
10
20
50
100
200
500
1000
2000
5000
10000
20000
Tamaño Matriz N



Además, se encuentra el porcentaje de memoria que fue usado durante las corridas, lo cual siempre se encontraba entre el 60% y 65%, y también la actividad del procesador que fue relativamente constante en todas las corridas realizadas.

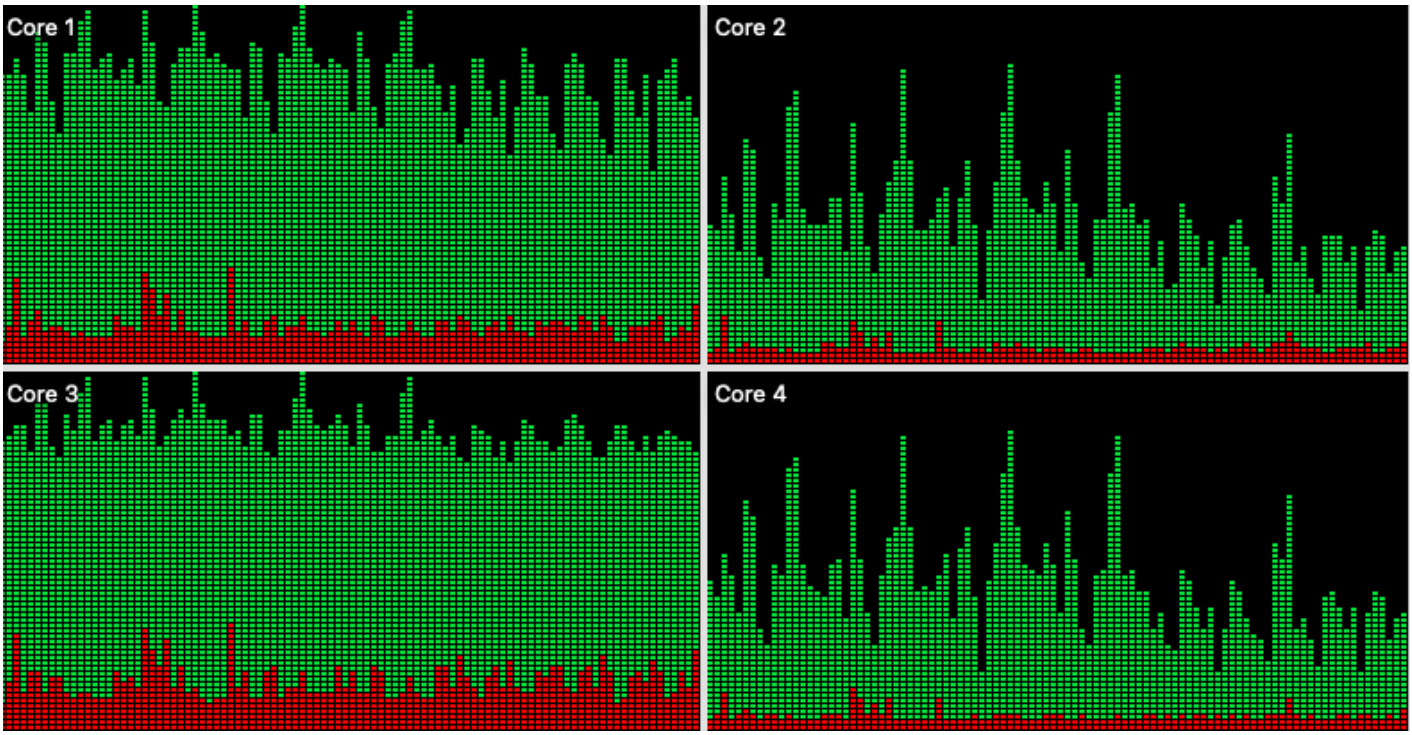
```
Tiempo Transcurrido: [0.0011061489994972362, 0.000368343999980425,
0.0005323630002749269, 0.00010932700024568476, 0.00010437499986437615,
0.00012427700039552292, 0.00022655500015389407, 0.00031523599955107784,
0.0005954230000497773, 0.0027489489993968164, 0.0018361140000706655,
0.008512839999639255, 0.01356922300055885, 0.042855576999500045, 0.10186264000003575,
0.37091526899985183, 0.7173119250001037, 1.9908750340000552, 6.983395570000539,
18.114055000000008] s

Corrida A7 Nº 1
/Users/georgechammas/Desktop/Universidad/2021-20/MCOC/MCOC2021-P0/A_float.py:175:
LinAlgWarning: Ill-conditioned matrix (rcond=4.13819e-08): result may not be accurate.
  x = solve(A,b, overwrite_a=True, overwrite_b=True)
Tiempo Transcurrido: [0.0010128399999302928, 0.0003070439997827634,
0.0006718309996358585, 0.0004932339998049429, 0.0007102850004230277,
0.0001779650001481059, 0.00017371699959767284, 0.00029271199946379056,
0.0006251909999264171, 0.0029223820001789136, 0.0024619589994472335,
0.0060120389998701285, 0.013044858000284876, 0.03722038099931524, 0.09956776400031231,
0.26161090999994485, 0.6894538480000847, 1.9029626370002006, 5.318658950000099,
15.898672067000007] s

Corrida A7 Nº 2
/Users/georgechammas/Desktop/Universidad/2021-20/MCOC/MCOC2021-P0/A_float.py:175:
LinAlgWarning: Ill-conditioned matrix (rcond=1.9996e-08): result may not be accurate.
  x = solve(A,b, overwrite_a=True, overwrite_b=True)
Tiempo Transcurrido: [0.0011048500000470085, 0.0002668839997568284,
0.0005096200002299156, 0.0003404630006116349, 0.0001370159998259623 5,
0.0001405940001859562 5, 0.000174396999682358, 0.0003026530002898653,
0.0008554300002288073, 0.002849354999852949, 0.003286797000328079,
0.00803302700023778, 0.013348231999771087, 0.04595012800018594, 0.10617685299985169,
0.26799935799954255, 0.741665185999409, 1.987661735999609, 5.960254298999644,
16.422021614000187] s

Corrida A7 Nº 3
```

Python 3.8.5) main [6] Line 5, Col 1 UTF-8 LF RW Mem 63%



Comentarios:

En cuanto al procedimiento "A", el caso que más se demoró fue el primero, en que se calcula la inversa de la matriz A y se multiplica por el vector b, mientras que el caso que menos se demoró fue el del solve con assume_a='pos'. En ambos casos, y en el resto también, el tiempo de ejecución del tipo de dato double fue mayor al float, debido a que tiene datos más pesados que este último. Además, todos los subcasos tienen casi los mismos tiempos de ejecución (existe poca variabilidad).

En cuanto al procedimiento "B", se redujo el tamaño máximo de la matriz laplaciana a 5000 (en el procedimiento A era 10000) porque se demoraba más de dos minutos por iteración (aquí es posible ver la relación exponencial entre el tamaño de la matriz y el tiempo de ejecución del programa). El caso que más se demoró fue el driver='evx' con overwrite_a=True, mientras que el que se demoró menos fue el driver='evd' con overwrite_a=False para el tipo de dato float, y el eigh por defecto para el double, aunque todos los casos tienen casi los mismos tiempos.

A diferencia de las entregas anteriores, no se traficó el uso de memoria. Sin embargo, se sabe que tiene una relación lineal con el tamaño de las matrices, por lo que el gráfico, al igual que las entregas anteriores, sería una simple recta lineal.

P0E5-E6: Matrices Dispersas y Complejidad Computacional

1. Complejidad Algorítmica de MATMUL

Se realizó un archivo py con el cual se ejecuta la operación MATMUL entre dos matrices laplacianas llenas y dispersas (tipo CSR), con un tipo de dato double. Al correr el código, se observó que, el uso de matrices dispersas es más eficiente en cuanto al tiempo de ensamblaje y de solución de la operación, debido a que solo guarda los números de las matrices que son diferentes de cero, es decir, trabaja con menos datos (los más importantes) y se ahorra sumas de 0 que no tienen efecto. Esto se nota en los gráficos de tiempo vs N que se muestran a continuación, en donde el tamaño máximo de matriz usado para las llenas es de 10.000, mientras que para las dispersas es de 1.000.000.

Código de Ensamblaje

A continuación, se presentan los dos códigos usados para armar las matrices laplacianas llenas y dispersas.

Función Laplaciana Matrices Llenas

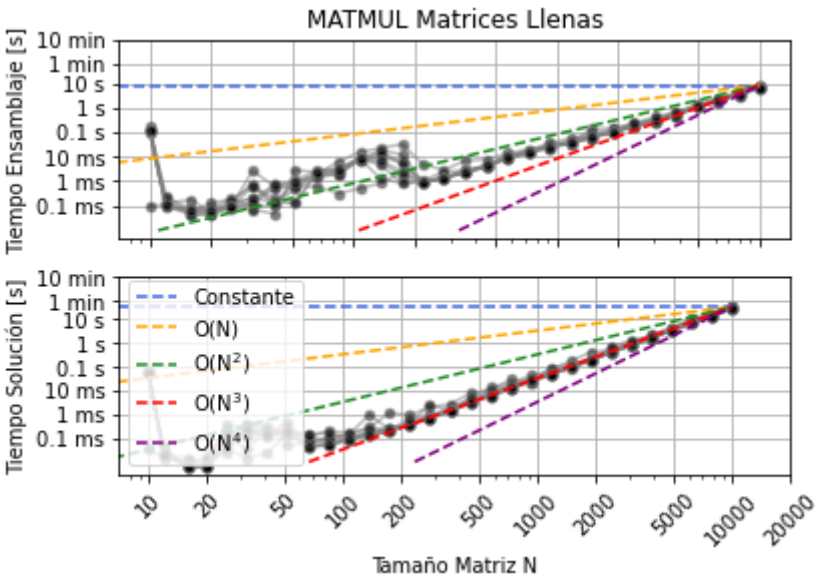
```
def laplaciana(N, t=double):
    e = eye(N) - eye(N,N,1)
    return t(e+e.T)
```

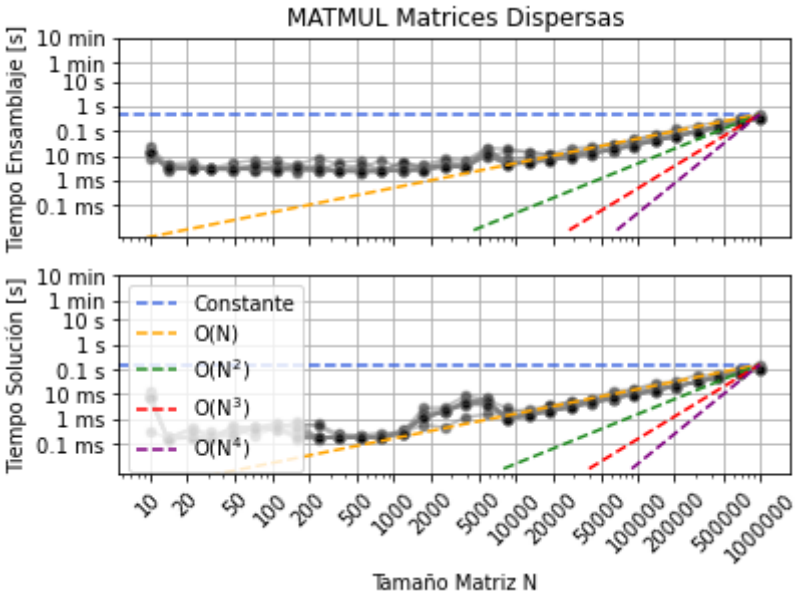
Función Laplaciana Matrices Dispersas

```
def laplaciana_dispersa(N, t=double):
    return 2*sparse.eye(N, dtype = double) - sparse.eye(N, N, 1, dtype = double) - s
```

Gráficos

Se presentan los gráficos obtenidos de tiempo de ensamblaje y de solución vs tamaño de matriz a continuación:





Las líneas discontinuas que aparecen en los gráficos muestran comportamientos y relaciones entre el tiempo transcurrido y el tamaño de las matrices. Dado que los gráficos tienen una escala bi-logarítmica, las relaciones exponenciales se muestran como lineales.

En el caso del tiempo de ensamblaje de matrices llenas, este presentó un comportamiento sub-lineal y sobre la relación cuadrática con matrices pequeñas, pero asintótico a la linea cúbica con tamaños mayores. Mientras que el tiempo de solución del MATMUL mantenía un comportamiento asintótico a la relación cúbica con casi todo tamaño de matriz.

En cuanto al tiempo de ensamblaje y el tiempo de solución MATMUL de las matrices dispersas, estos presentaron un comportamiento sobre lineal hasta N=10.000, luego del cual pasaron a ser asintóticos con respecto a la misma recta.

2. Complejidad Algorítmica de SOLVE

Se realizó un archivo py con el cual se ejecuta la operación SOLVE del sistema de ecuaciones Ax=b, donde la matriz A es laplaciana (se usaron matrices llenas y matrices dispersas del tipo CSR) y el vector b es un vector de unos, ambos con un tipo de dato double. Al correr el código, se observó que, el uso de matrices dispersas es más eficiente en cuanto al tiempo de ensamblaje y de solución de la operación, debido a que solo guarda los números de las matrices que son diferentes de cero, es decir, trabaja con menos datos (los más importantes) y se ahorra sumas de 0 que no tienen efecto. Esto se nota en los gráficos de tiempo vs N que se muestran a continuación, en donde el tamaño máximo de matriz usado para las llenas es de 10.000, mientras que para las dispersas es de 1.000.000.

Código de Ensamblaje

A continuación, se presentan los dos códigos usados para armar las matrices laplacianas llenas y dispersas.

Función Laplaciana Matrices Llenas

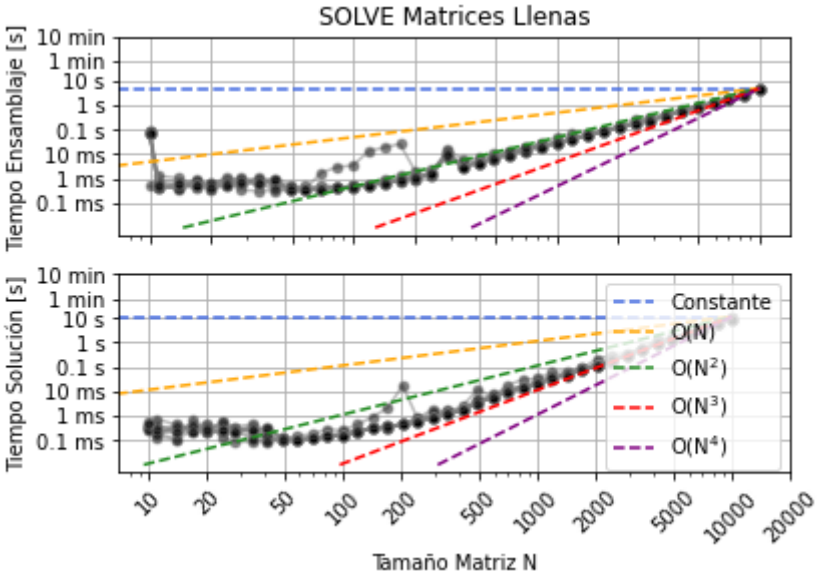
```
def laplaciana(N, tipo=double):
    e = eye(N)-eye(N,N,1)
    return tipo(e+e.T)
```

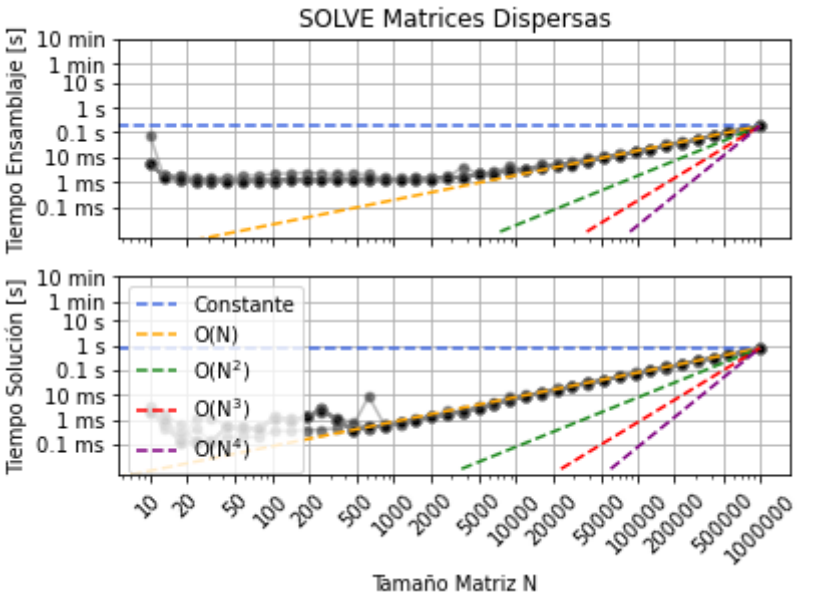
Función Laplaciana Matrices Dispersas

```
def laplaciana_dispersa(N, t=double):
    return 2*sp.eye(N, dtype = double) - sp.eye(N, N, 1, dtype = double) - sp.eye(N,
```

Gráficos

Se presentan los gráficos obtenidos de tiempo de ensamblaje y de solución vs tamaño de matriz a continuación:





En el caso del Solve con matrices llenas, el tiempo de ensamblaje es asintótico a la relación cuadrática entre el tiempo transcurrido y el tamaño de la matriz, es decir, O(N^2), mientras que el tiempo de solución es asintótico a la relación cúbica O(N^3).

En cuanto al Solve con matrices dispersas, tanto el tiempo de ensamblaje como el de solución son asintóticos a la relación lineal O(N).

Con esto, se puede ver cómo afecta el tamaño de las matrices al comportamiento aparente, en donde, en todos los casos, al tener matrices más grandes, siempre se tiene un comportamiento asintótico con una de las relaciones O(N^n), mientras que con matrices pequeñas no siempre sucede, sino que las corridas se encuentran entre dos rectas O(N^n).

Analizando la estabilidad de las corridas, es decir, cuánto varía una con la otra, las del tiempo de ensamblaje de matrices llenas, y los tiempos de solución de las llenas y dispersas coinciden todas menos una corrida, que se desvía alrededor de N=200. Mientras que las corridas del tiempo de solución de las matrices llenas son todas prácticamente iguales.

3. Complejidad Algorítmica de INV

Se realizó un archivo py con el cual se ejecuta la operación INV en donde se invierte una matriz A laplaciana (se usaron matrices llenas y matrices dispersas del tipo CSC) con un tipo de dato double. Al correr el código, se observó que, el uso de matrices dispersas es más eficiente en cuanto al tiempo de ensamblaje y de solución de la operación, al igual que MATMUL y SOLVE. Sin embargo, para INV, el tamaño máximo de las matrices fue de 10.000 para matrices llenas y dispersas, dado que la reducción en el tiempo en el caso de las dispersas no fue notable, y se demoraba más de dos minutos por corrida con matrices de mayor tamaño, por lo que no se llegó a matrices de N=1.000.000 como MATMUL y SOLVE.

Código de Ensamblaje

A continuación, se presentan los dos códigos usados para armar las matrices laplacianas llenas y dispersas.

Función Laplaciana Matrices Llenas

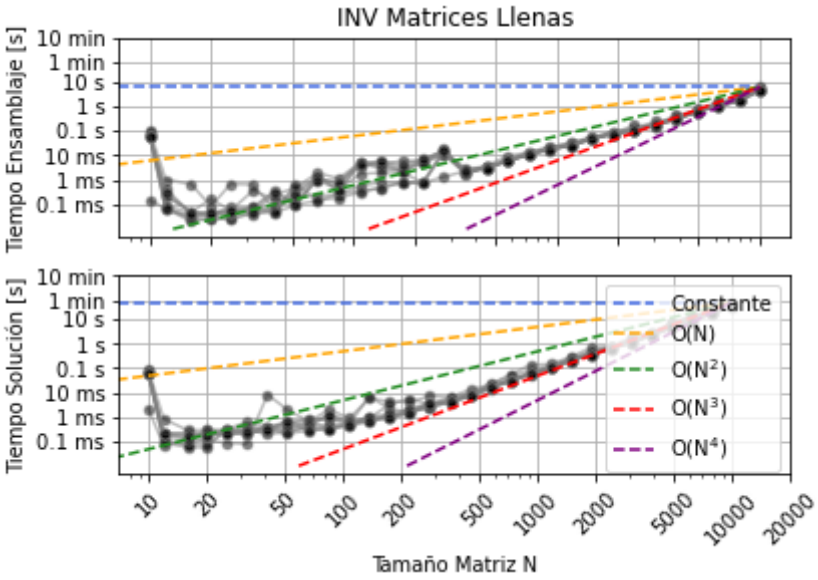
```
def laplaciana(N, tipo=double):
    e = eye(N)-eye(N,N,1)
    return tipo(e+e.T)
```

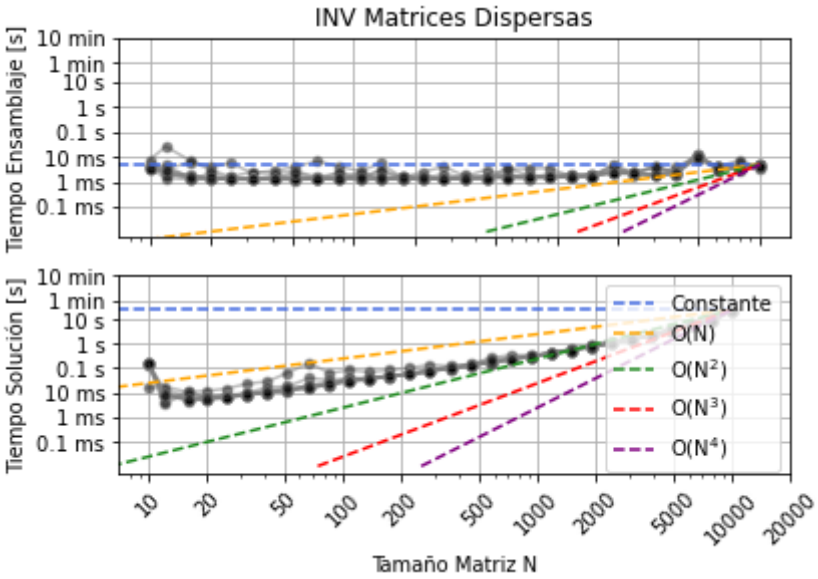
Función Laplaciana Matrices Dispersas

```
def laplaciana_dispersa(N, t=double):
    return 2*sp.eye(N, dtype = double) - sp.eye(N, N, 1, dtype = double) - sp.eye(N,
```

Gráficos

Se presentan los gráficos obtenidos de tiempo de ensamblaje y de solución vs tamaño de matriz a continuación:





En el caso del Inv con matrices llenas, el tiempo de ensamblaje es asintótico a la relación cuadrática entre el tiempo transcurrido y el tamaño de la matriz, es decir, O(N^2), pero solo hasta un N=200 aproximadamente, luego es asintótico a la relación cúbica O(N^3), mientras que el tiempo de solución solo es asintótico a la relación cúbica O(N^3).

En cuanto al Inv con matrices dispersas, el tiempo de ensamblaje es asintótico a la relación constante, mientras que el tiempo de solución es asintótico a la relación cuadrática O(N^2) después de N=1000.

Con esto, se puede ver cómo afecta el tamaño de las matrices al comportamiento aparente, en donde, en todos los casos, al tener matrices más grandes, siempre se tiene un comportamiento asintótico con una de las relaciones O(N^n), mientras que con matrices pequeñas no siempre sucede, sino que las corridas se encuentran entre dos rectas O(N^n).

Analizando la estabilidad de las corridas, se notó que en los cuatro gráficos (tiempos de ensamblaje y solución de las matrices llenas y dispersas) existe variaciones entre una corrida y la otra, pero todas en un rango pequeño, en donde se sigue notando que tienen la misma tendencia.