Manual para uso de Matlab sobre él LNS.

Autor: Gerardo Flores Petlacalco.

Contenido

Introducción

En el laboratorio del LNS se ofrecen diferentes aplicaciones para sus usuarios, estas aplicaciones tienen, entre sus opciones, poder paralelizar sus aplicaciones con el objetivo de mejorar el rendimiento de las mimas.

Algunas razones para considerar un cómputo paralelo como el que se ofrece, destacan:

- Ahorrar tiempo por la distribución de tareas y ejecución de las mismas en forma simultánea.
- Resolver problemas con grandes cantidades de datos distribuyéndolos.
- Tomar ventaja de los recursos de su máquina local y escalar sus tareas de computo a un clúster para hacer un cómputo en la nube.

Entre las aplicaciones que permiten paralelizar su código está disponible Matlab mediante su "Parallel Computing Toolbox", que otorga las funciones necesarias. En este manual se plantarán las bases necesarias para entender el funcionamiento de esta toolbox y hacer un uso dentro del LNS.

Computo paralelo sobre Matlab

Matlab provee una serie de herramientas para realizar computo paralelo y usarlo para optimizar sus aplicaciones. Estas, se enfocan a resolver problemas con cantidades grandes de datos usando procesadores multinúcleo, GPU´s y Clúster.

Con ciclos *parfor*, tipos especiales de arreglos de datos y algoritmos de paralización, Matlab proporciona formas de optimizar sus tareas sin la necesidad de usar la programación en CUDA o MPI.

El "Parallel Computing Toolbox" le permitirá usar al máximo las capacidades de computadoras multinucleo, ejecutando las aplicaciones de forma paralela sobre *Workers* que trabajan de forma local en la máquina. Además, sin realizar cambio alguno en el código, podrás correr aplicaciones dentro de un clúster o una red de servicios de computo. Y podrás ejecutar tu aplicación de forma interactiva recuperando datos al momento o dejar un trabajo corriendo en background para regresar después a revisar los resultados.

Características de Matlab en el LNS

En el LNS se dispone de la Versión: 9.1.0.441655 (R2016b) de Matlab para realizar sus tareas de computo, las Toolbox instaladas se listan a continuación

---

MATLAB	Versión 9.1	(R2016b)
MATLAB Coder	Versión 3.2	(R2016b)
MATLAB Compiler	Versión 6.3	(R2016b)
Parallel Computing Toolbox	Version 6.9	(R2016b)

Si desea alguna otra Toolbox, favor de ponerse en contacto con los administradores del LNS, para determinar las condiciones de uso y definer si se puede o no adquirir la toolbox necesaria.

Conceptos básicos

Para usar el módulo de paralelismo en Matlab dentro de LNS, se deben conocer unos conceptos básicos que serán necesarios para comprender mejor el funcionamiento interno.

Entre estos conceptos se encuentra:

- Workers, es la unidad básica de procesamiento de tareas dentro de Matlab parallel toolbox.
- Matlab pool, es una colección de Workers y sirven para activar las características del cómputo paralelo en Matlab (Instrucciones *parfor y spdm).* Se puede configurar para que el usuario asigne los recursos de acuerdo a sus necesidades o dejar un perfil predefinido donde Matlab sea quien maneje los recursos disponibles.

Acceso a MATLAB dentro del clúster del LNS

El LNS ofrece el software MATLAB para su uso dentro del clúster, para cargar el módulo se requiere seguir los siguientes pasos. Dependiendo de la forma en la cual se conectó al clúster del LNS será como se muestre MATLAB al final.

Los pasos a seguir para cargar y hacer uso del módulo, se listan a continuación.

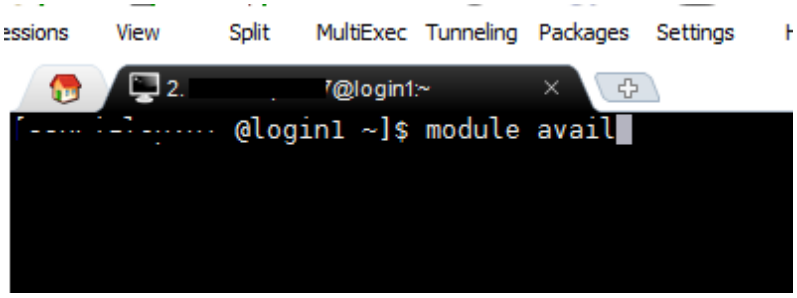
1. Conectarse al clúster del LNS, para ello hay dos formas de hacerlo.
    a. La primera, es la forma clásica, pasando por el nodo intermedio UI y de ahí hacer la conexión SSH.
        i. ssh nombre_de_usuario@ui.buap.mx
        ii. ssh nombre_de_usuario@192.168.170.213
    b. La segunda es conectarse usando una VPN y por medio del comando. Para más detalles puede consultar el manual “Conexión a aplicaciones gráficas del LNS”
        i. ssh -X -C nombre_de_usuario@192.168.170.213

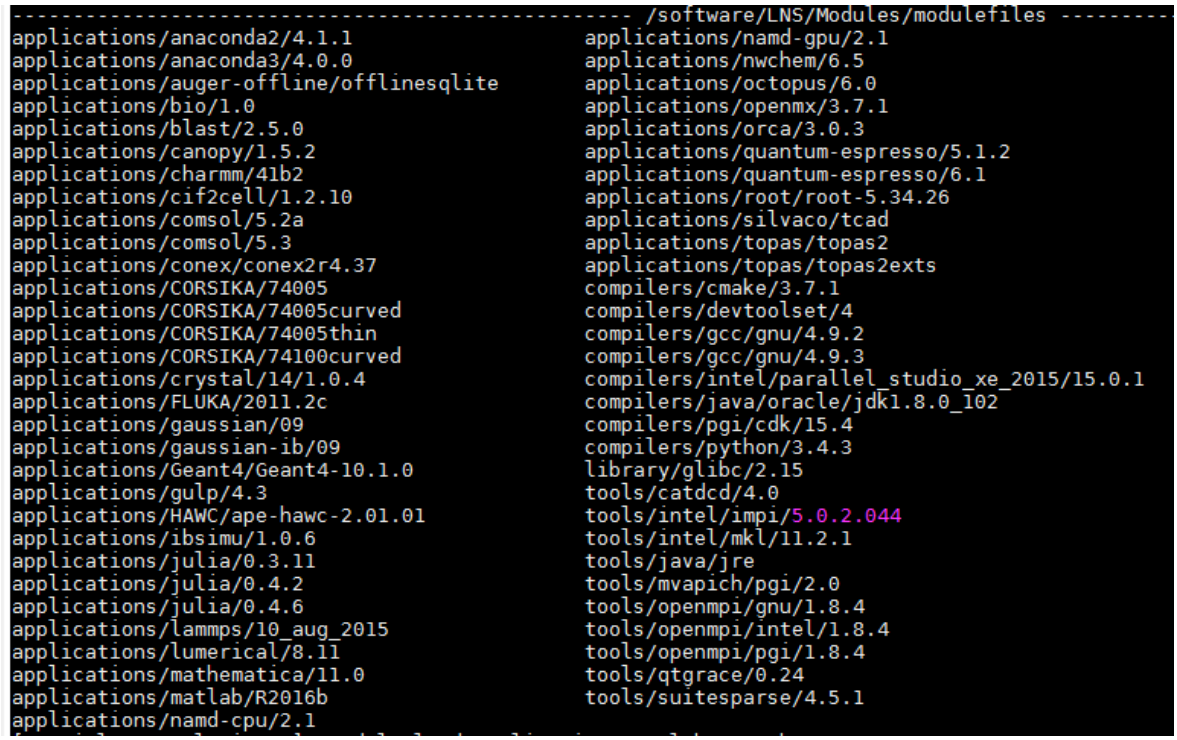
    Una vez mostrado el prompt que usemos, seguimos con los siguientes pasos

2. Dentro del clúster del LNS tecleamos el comando

    module avail

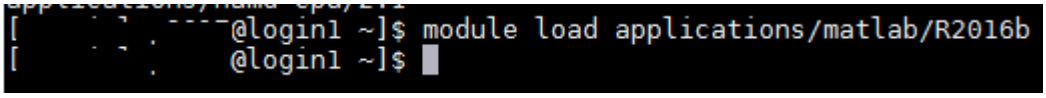
    Este comando nos listará los módulos disponibles para su uso dentro del LNS, recuerda que no tienes acceso a todos, eso dependerá de las directivas del grupo al que pertenezcas, para más información contacta con el administrador del LNS.





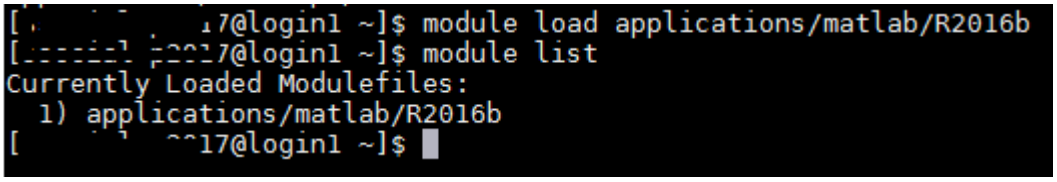
3. Carga el módulo, para cargar el módulo a nuestra sesión usaremos el comando

module load applications/matlab/R2016b



Se cargará el módulo a nuestra sesión y para confirmar la carga podemos teclear

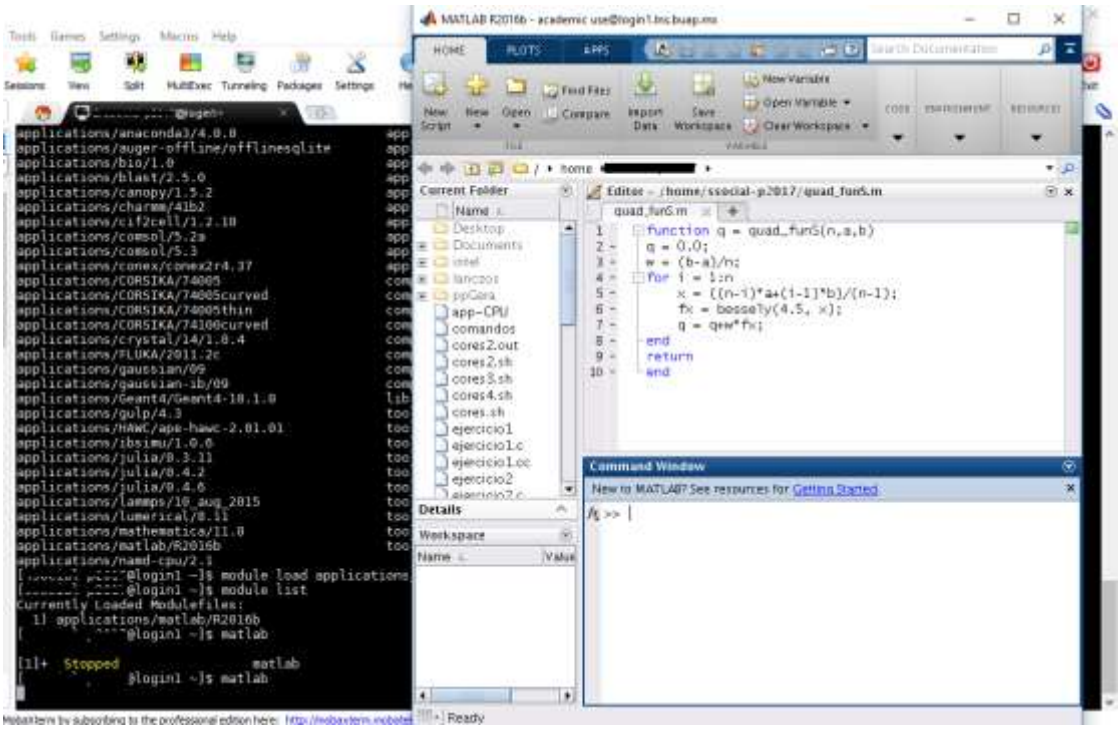
module list



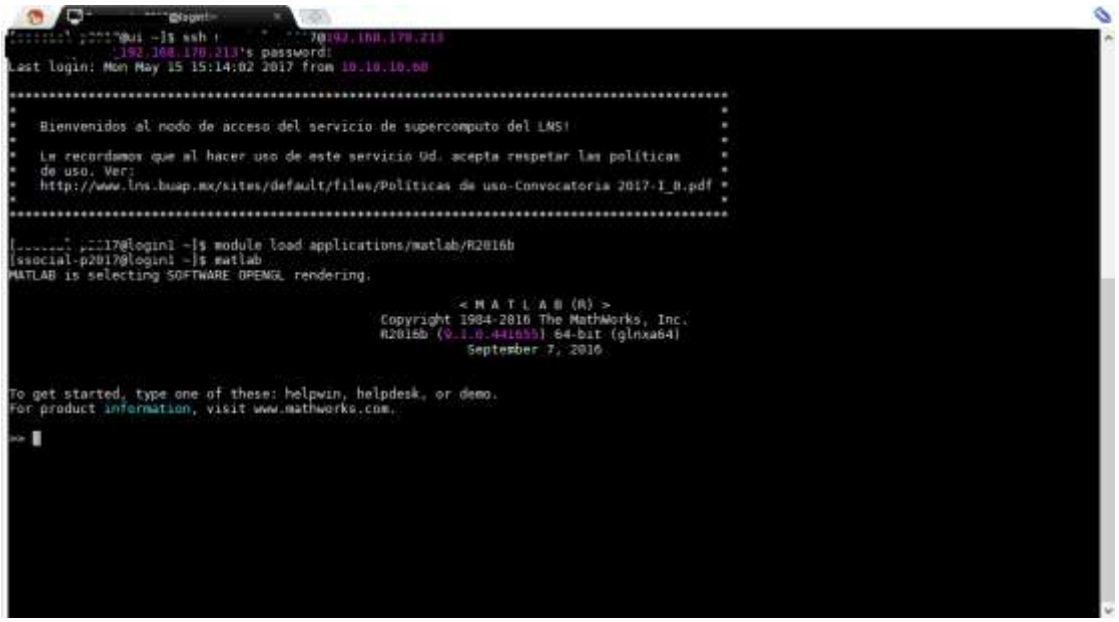
4. Ejecuta la aplicación, una vez realizado los pasos anteriores, podemos ejecutar MATLAB con la instrucción

Matlab

Esta captura es cuando te conectas para hacer uso de Matlab usando la interfaz gráfica



Esta es la interfaz de Matlab cuando no haces el uso de las aplicaciones gráficas disponibles dentro del LNS.



Nota: La recomendación es editar su código en haciendo uso de la interfaz gráfica y posteriormente ejecutarlo usando el SLURM y disponiendo de los datos en un archivo de salida. Esta configuración es la recomendaba puesto que se dispone de un mayor número de *workers* que ejecutar tu tarea que haciendo uso de la interfaz. Además de que los administradores podrán matar su proceso en caso de detectar un uso intensivo de procesamiento por largos intervalos de tiempo.

Instrucciones básicas de paralelismo en Matlab

Existen dos instrucciones para paralelizar en Matlab, la primera es *parfor*, esta instrucción convierte un bucle *for* de forma secuencial a un bucle *for* de forma paralela que se ejecuta sobre el conjunto de *workers* asignados.

La segunda instrucción es *SDPM*, esta instrucción sirve para ejecutar tareas de forma paralela, al contrario del *parfor* con esta sentencia se tiene mayor control sobre las acciones que se realizarán por el código en Matlab.

Para comenzar

Aquí se muestran las dos instrucciones básicas de paralelismo el MatLab, al ejecutar alguna de las dos se creará un espacio de trabajo que se conoce como MatLabpool, este contendrá el número de *workers* disponibles para un trabajo paralelo de MatLab. Por default, al hacer el llamando de una tarea paralela *parfor* o SPMD se crea un MatLabPool de manera automática, pero nosotros recomendamos que todos los programas y/o funciones paralelas que quiera crear sean iniciados con el siguiente bloque de instrucciones:

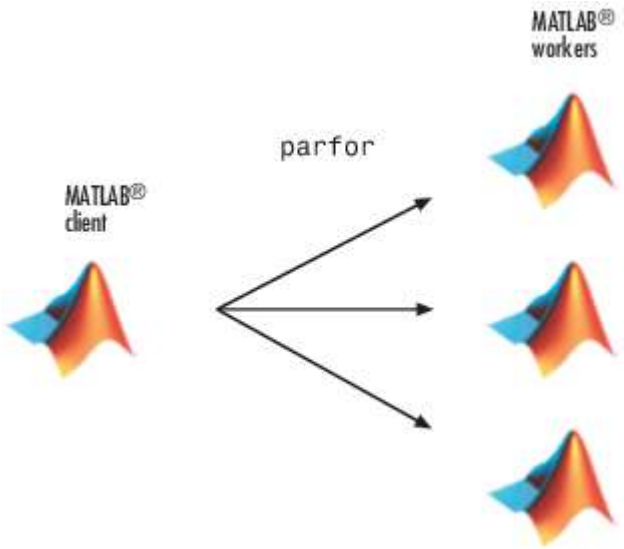
```
myCluster = parcluster('local');
myCluster.NumWorkers
parpool(myCluster.NumWorkers);
```

Las instrucciones son explicadas a continuación:

1. `myCluster = parcluster('local');`
    a. Carga la configuración local del clúster y la asigna a la variable myCluster.
2. `myCluster.NumWorkers`
    a. Esta instrucción nos muestra el número de Workers que tenemos disponibles y que formarán parte de nuestra MatLabPool.
3. `parpool(myCluster.NumWorkers);`
    a. Crea un MatLabPool de acuerdo al número de *workers* disponibles en nuestra configuración local. Este valor puede ser modificado de manera manual pasándole como variable un número entero, si ese número es 0, se ejecuta el código de manera secuencial, si es mayor al número de *workers* disponibles para trabajo se usarán todos los disponibles, si se deja en blanco crea la MatLabPool con la configuración por defecto.

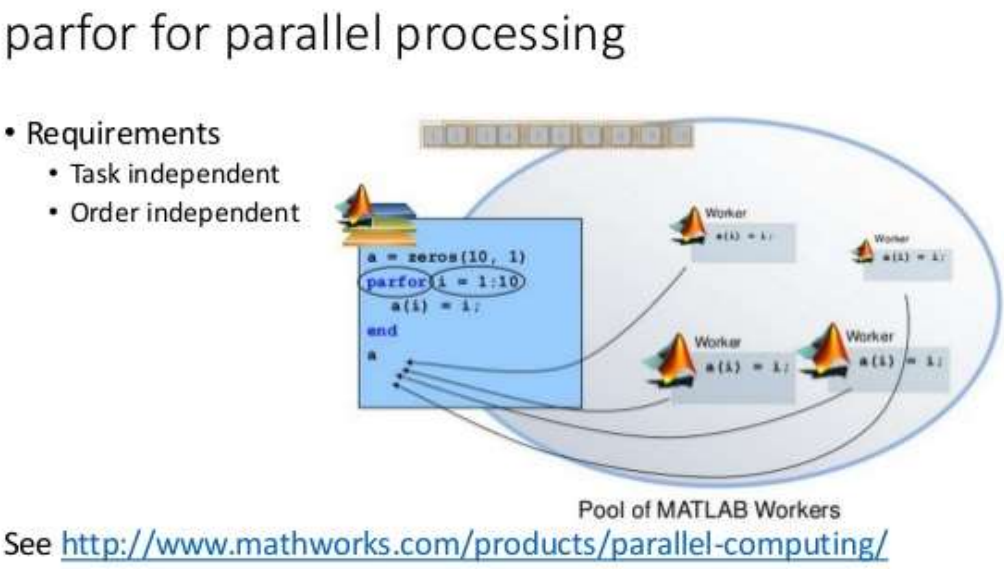
ParFor

El *parfor* es la forma más básica de paralelizar programas en MatLab, básicamente es un loop *for* donde todas sus interacciones son distribuidas y ejecutadas entre todos los *workers* disponibles dentro de un MatLabpool para disminuir en tiempo de ejecución de los cálculos.



Sin embargo, el uso del *parfor* no tan sencillo, pues se necesitan considerar ciertas limitaciones que nos permitirán decidir si es la indicada para nuestro trabajo. Para hacer un adecuado uso de *parfor*, debe tomar en cuenta lo siguiente:

- Las tareas que se envíen a un *parfor* tendrán un orden independiente, es decir no hay forma de saber qué se ejecutará primero y cómo lo hará.
- Los incrementos dentro del bucle deben ser valores consecutivos.
- Las iteraciones de cada *parfor* deben ser independientes. Una iteración no debe depender de otra porque se ejecutan en un orden no determinado.
- No puede usar un *parfor* dentro de otro *parfor*.



Además, para iniciar un *parfor*, se debe crear un Matlab pool con el número de *workers* definidos para hacer la división del trabajo, en caso de que no exista un MatLab pool definido, la instrucción creará uno con la configuración que trae por defecto la instalación de MatLab

La ventaja principal de un *parfor* se ve en el tiempo de ejecución de un trabajo, tal como se muestra en la parte de abajo. Usted puede hacer la prueba con los siguientes códigos sobre MatLab

	Código	**Captura**	**Tiempo de ejecución**

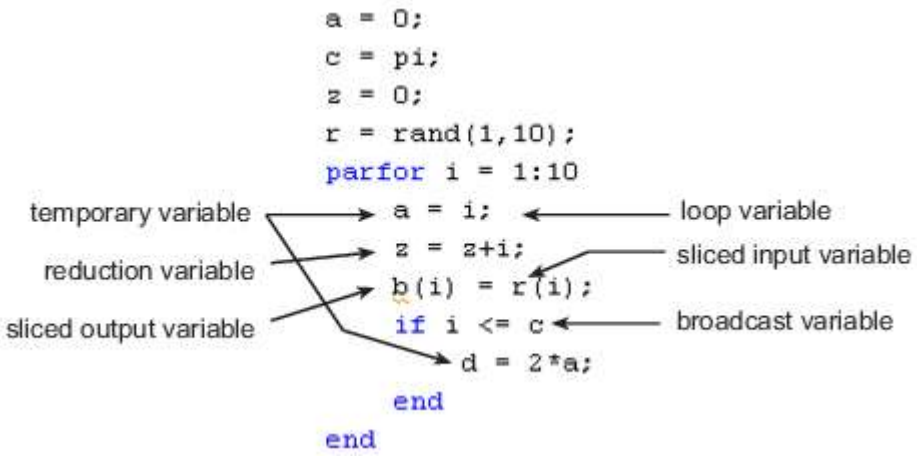
For	Tic
	n = 200;
	A = 500;
	a = zeros(n);
	for i = 1:n
	a(i) = max(abs(eig(rand(A))));
	end
	toc
		tic
		n = 200
		A = 500;
		a = zeros(n);
		for i = 1:n
		a(i) = max(abs(eig(rand(A))));
		end
		toc
			34.096831 seconds

Par for	Tic
	ticBytes(gcp);
	n = 200;
	A = 500;
	a = zeros(n);
		tic
		ticBytes(gcp);
		n = 200;
		A = 500;
		a = zeros(n);
		parfor i = 1:n
		a(i) = max(abs(eig(rand(A))));
		end
		tocBytes(gcp)
		toc
			23.052980 seconds

	parfor i = 1:n a(i) = max(abs(eig(rand(A)))); end tocBytes(gcp) toc		

Asimismo, el ciclo *parfor* contiene varios tipos de variables, estas variables son clasificadas dentro del programa al momento de ejecutar el código, por lo cual, si alguna de ellas no está dentro de la clasificación que se muestra, nos marcará un error. La lista de variables se muestra a continuación:

- Variables "Loop". Sirve como índice del bucle para los "array".
- Variables "Sliced". Es un array cuyos segmentos son operados por diferentes iteraciones del bucle.
- Variables "Broadcast". Es una variable definida antes del bucle, cuyo valor se utiliza dentro del bucle, pero no es asignado dentro del bucle.
- Variables "Reduction". Es una variable que acumula un valor a través de las iteraciones del bucle, independientemente del orden de la iteración.
- Variables "Temporary". Es una variable creada dentro del bucle, pero a diferencia de la variable del tipo "sliced", no estará disponible fuera del bucle



Sintaxis

La sintaxis del parfor es la siguiente:

```
parfor loopVar = initVal:endVal; statements; end
parfor (loopVar = initVal:endVal, M); statements; end
```

Donde los parámetros se manejan así:

- `loopVar: Variable del ciclo que tienen como valor iniciar el dado por initVal para tomar el valor final definido por endVal.` La variable puede ser cualquiera de tipo número y el valor debe ser un entero.
- `initVal: El valor iniciar de la variable loopVar. La variable puede ser cualquier variable numérica y el valor un entero.`
- `endVal: El índice final de la variable loopVar. La variable puede ser cualquier variable numérica y el valor un entero.`
- `Statements: Especifica las instrucciones que se ejecutarán dentro del parfor, recuerde: Un parfor no debe contener otro parfor`
- `M: Especifica el número máximo de workers para trabajar en la tarea for. De no definirse usará el perfil por default.`

Nota: En la configuración por default del clúster del LNS, el número máximo de *workers* para trabajar es de 16 si mandamos nuestro trabajo desde interfaz gráfica (NO RECOMENDABLE), si usamos el manejador de tareas SLURM se disponen de 24. En algunos casos pueden solo tener 12 *wokers*, pero depende la configuración del trabajo. Por ello se recomienda usarlo exclusivamente usando el planeador de tareas SLURM para disponer de todos los recursos necesarios y evitar perdida de datos por la cancelación de las tareas por un administrador.

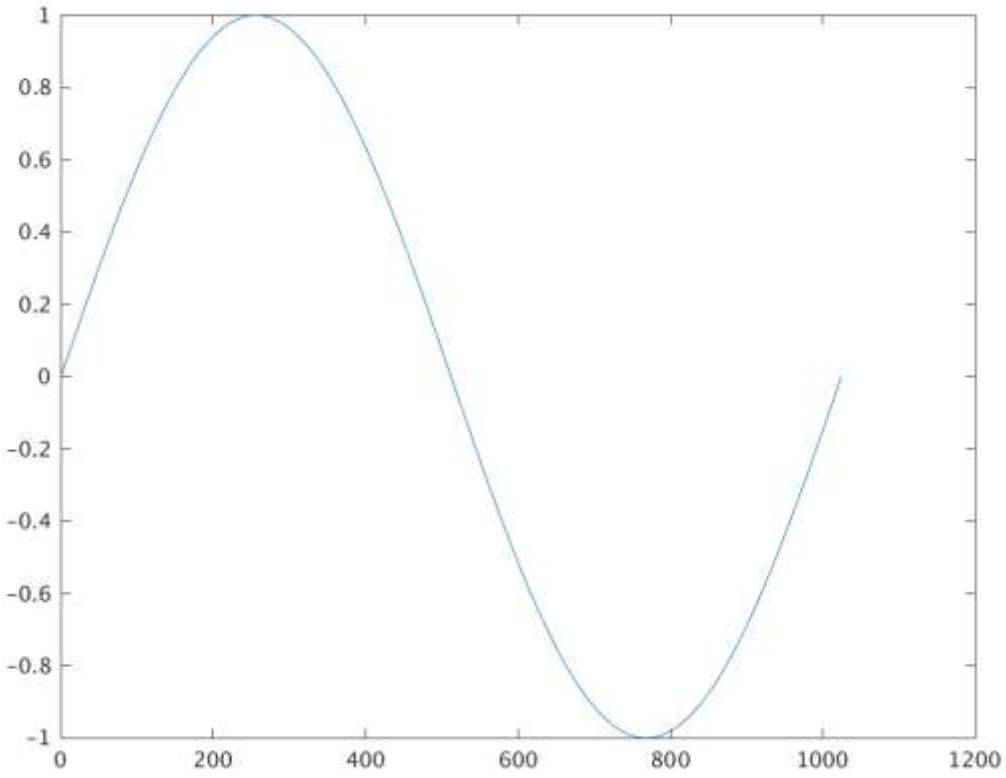
Ejemplos con *parfor*

Para comprender mejor el uso de *parfor* se muestran una serie de ejemplos que usted puede poner a prueba dentro del clúster y comprender mejor el uso del *parfor* para comprender su comportamiento.

Ejemplo 1

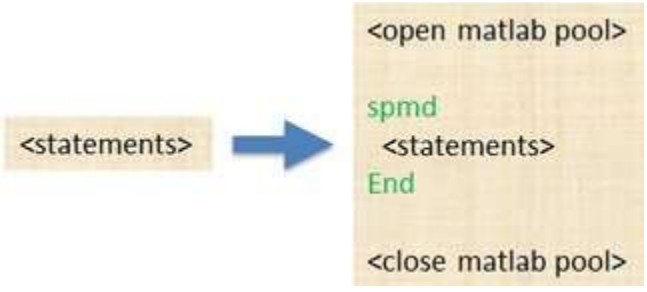
Este programa crea una gráfica de la función seno, usando el

```
myCluster = parcluster('local');
myCluster.NumWorkers
parpool(myCluster.NumWorkers);
tic
parfor i=1:1024
      A(i) = sin(i*2*pi/1024);
end
toc
plot(A)
saveas(gcf, strcat('pstfor','png'))
delete(gcp)
```



SPMD

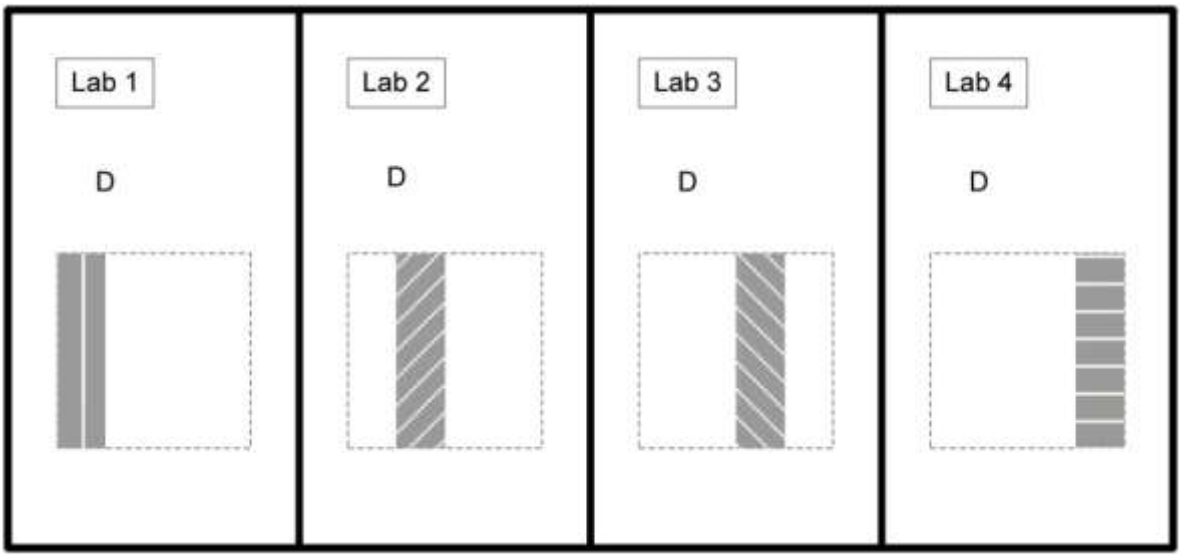
La instrucción “Simple program multiple data” define un bloque de instrucciones para ser ejecutadas simultáneamente sobre múltiples *workers*. Las variables asignadas dentro de una instrucción SPDM se acceden desde el cliente que lanzo la aplicación, esta instrucción permite tener un mayor nivel de control sobre las tareas ejecutadas como se verá más adelante.



La característica “Simple Program” de la instrucción SPMD implica que un código idéntico puede correr sobre multiples *workers*. Usted puede correr un programa en MATLAB, y sus partes pueden estar etiquetadas como bloques de la instrucción SPMD para que puedan correr sobre los *workers*

asignados. Cuando el bloque de instrucciones SPMD sea completado, el programa seguirá corriendo sobre el cliente que lanzo la instrucción.

La característica “Multilple Data” indica que cada bloque de instrucciones dentro de la sentencia SPMD tiene un único y exclusivo conjunto de datos para que el los procese. Esto indica que múltiples conjuntos de datos pueden ser acomodados sobre múltiples *workers*.



Las aplicaciones para SPDM están pensadas en aquellas que requieren la ejecución de un programa sobre muchos conjuntos de datos, cuando la comunicación y la sincronización es requerida entre los *workers*. Algunos casos comunes para usar esta instrucción son:

a. Programas que tomen un gran tiempo de ejecución
b. Programas que operen sobre un gran conjunto de datos

Sintaxis

Para hacer uso de la instrucción SPMD, se requiere la creación de un Matpool, sin embargo, nosotros usaremos la configuración por default.

La sintaxis de esta instrucción es la siguiente:

```
spmd
    <statements>
end
```

Donde:

- <statements> Son las instrucciones que se ejecutan simultáneamente en el espacio pararelo definido por el perfil por default.

Además, al igual que en un ciclo *parfor* podemos especificar el número de *workers* que queremos usar para esa instrucción, solo necesitamos modificar la instrucción como se muestra abajo:

```
spmd (n)
    <statements>
end
```

Donde:

- (n) Sirve para indicar el número de workers que queremos usar para esa tarea de paralelismo, el n debe estar entre 1 y el número de workers disponibles en la configuración por default del espacio de trabajo. Si n = 0, no se usarán workers y todo se ejecutará como código normal de MATLAB.

A diferencia del *parfor*, en la instrucción *spmd* cada *worker* tienen un valor conocido como *labindex*. Este valor, sirve para correr ciertos códigos sobre un numero de *workers* definido para personalizar la ejecución del programa. En el siguiente ejemplo, se crean diferentes *arrays*, dependiendo del número *labindex*.

Ejemplos

Ejemplo 1

En este ejemplo se muestra un ejemplo básico donde se multiplica una matriz de unos. Abajo se muestra la descripción del ejemplo para comprender su funcionamiento. Se introduce los datos distribuidos, que se explican más abajo.

```
myCluster = parcluster('local');
myCluster.NumWorkers
parpool(myCluster.NumWorkers);
W = ones(6,100); % Crea una matriz de 6 x 100 de unos.
W = distributed(W); % Esta instrucción convierte el arreglo en un dato distribuido
spmd
       T = W*2; % Calculamos la matriz, el paralelo.
end
T % Esperamos el resultado en T.
delete(gcp)
```

Salida

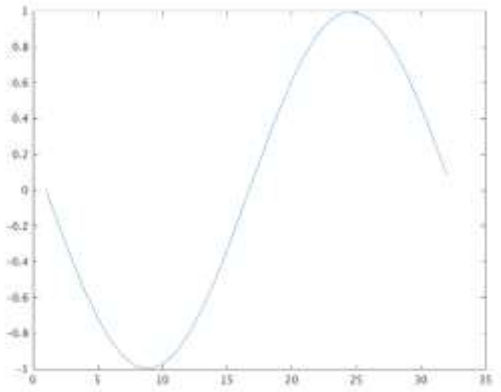
```
*Resultados.txt

                            < M A T L A B (R) >
                  Copyright 1984-2016 The MathWorks, Inc.
                   R2016b (9.1.0.441655) 64-bit (glnxa64)
                              September 7, 2016


To get started, type one of these: helpwin, helpdesk, or demo.
For product information, visit www.mathworks.com.

>> >>
ans =

    24

>> Starting parallel pool (parpool) using the 'local' profile ... Warning: The system time zone setting, 'Mexico/General', does not specify a
connected to 24 workers.
>> >> >> >>
T =

  Columns 1 through 13

     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2

  Columns 14 through 26

     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2

  Columns 66 through 78

     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2

  Columns 79 through 91

     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2     2     2     2     2

  Columns 92 through 100

     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2
     2     2     2     2     2     2     2     2     2

>> Parallel pool using the 'local' profile is shutting down.
>>
```

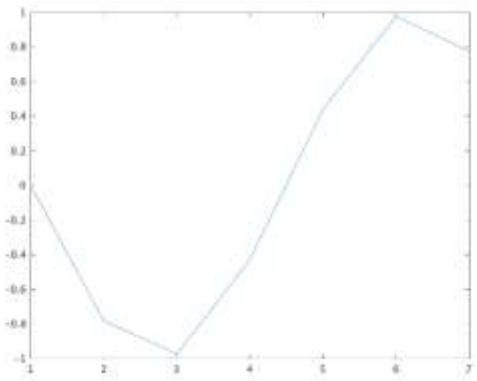
Ejemplo 2

En este ejemplo se realizarán graficas de seno, los valores y la forma de las gráficas estarán determinados por el número de *worker*, en este ejemplo se nota la distribución de tareas en un MatLabpool por medio del atributo *labindex*. Al final, las gráficas obtenidas se guardan en archivos de imagen que pueden descargarse para su estudio.

```
myCluster = parcluster('local');
myCluster.NumWorkers
parpool(myCluster.NumWorkers);
spmd
  % build magic squares in parallel
  switch(labindex)
      case labindex > 5
          aux = -pi:((labindex-5) *0.1): pi
          q = sin(aux)
      case labindex < 5
          aux = -pi:((labindex+5) * 0.1): pi
          q = sin(aux)
      otherwise
          aux = -pi:(labindex * 0.1): pi
          q = sin(aux)
  end
end
for ii=1: length(q)
  % plot each magic square
  %figure, imagesc(q{ii});
  plot(q{ii})
  saveas(gcf, strcat('ejemplo', int2str(ii)),'png');
end
delete(gcp)
```

Salidas





Tipos de Datos distribuidos

Para mejorar el manejo de grandes cantidades de datos entre *workers* de MatLab se implementan un nuevo tipo de datos llamados *Distributed Array*. Su función es segmentar un *array* y asignarlo a cada *worker* cuando se le pida esto con ayuda de la instrucción SPMD. Puedes particionar un *array* de dos dimensiones horizontalmente, asignando columnas a los *workers* o particionarlo verticalmente para asignar columnas.

Por ejemplo, para distribuir un *array* de 80 * 100 entre 4 *workers*, puedes particionarlo por columnas de 80*25 o por filas donde cada *worker* recibe un segmento de 20*100. Si el arreglo no es simétrico, MatLab distribuirá de la mejor manera posible los segmentos de *array*.

Este es un ejemplo donde se crea una matriz de 80*1000 que se distribuye entre el número de *workers*.

```
myCluster = parcluster('local');
myCluster.NumWorkers
parpool(myCluster.NumWorkers);


spmd
  A = zeros(80, 1000);
  D = codistributed(A)
end
```

Variables:

- La instrucción zeros crea un arreglo en su espacio de trabajo y la asigna a la variable A
- El segundo comando distribuye el array D en 4 *workers*. Los resultados arrojados se muestran abajo.

•

```
Lab  1:

  This worker stores D(:,1:63).

          LocalPart: [80×63 double]
      Codistributor: [1×1
codistributor1d]

Lab  2:

  This worker stores D(:,64:126).

          LocalPart: [80×63 double]
      Codistributor: [1×1
codistributor1d]

Lab  3:

  This worker stores D(:,127:189).

          LocalPart: [80×63 double]
      Codistributor: [1×1
codistributor1d]

Lab  4:

  This worker stores D(:,190:252).

          LocalPart: [80×63 double]
      Codistributor: [1×1
codistributor1d]

Lab  5:

  This worker stores D(:,253:315).

          LocalPart: [80×63 double]
      Codistributor: [1×1
codistributor1d]

Lab  6:

  This worker stores D(:,316:378).

          LocalPart: [80×63 double]
      Codistributor: [1×1
codistributor1d]

Lab  7:

  This worker stores D(:,379:441).

          LocalPart: [80×63 double]
      Codistributor: [1×1
codistributor1d]

Lab  8:

  This worker stores D(:,442:504).

          LocalPart: [80×63 double]
      Codistributor: [1×1
codistributor1d]

Lab  9:

  This worker stores D(:,505:566).

          LocalPart: [80×62 double]
      Codistributor: [1×1
codistributor1d]

Lab 10:

  This worker stores D(:,567:628).

          LocalPart: [80×62 double]
      Codistributor: [1×1
codistributor1d]

Lab 11:

  This worker stores D(:,629:690).

          LocalPart: [80×62 double]
      Codistributor: [1×1
codistributor1d]

Lab 12:

  This worker stores D(:,691:752).

          LocalPart: [80×62 double]
      Codistributor: [1×1
codistributor1d]

Lab 13:

  This worker stores D(:,753:814).

          LocalPart: [80×62 double]
      Codistributor: [1×1
codistributor1d]

Lab 14:

  This worker stores D(:,815:876).

          LocalPart: [80×62 double]
      Codistributor: [1×1
codistributor1d]
```

```
Lab 15:

  This worker stores D(:,877:938).

         LocalPart: [80×62 double]
      Codistributor: [1×1
codistributor1d]

Lab 16:

  This worker stores
D(:,939:1000).

          LocalPart: [80×62
double]
      Codistributor: [1×1
codistributor1d]
```

Para declarar un conjunto de datos distribuido, se utiliza en constructor

codistributed ()

Por ejemplo:

```
A = zeros(80, 1000);
D = codistributed(A)
```

En este ejemplo, se crea una matriz de ceros, para luego ser particionada por la instrucción codistributed en segmentos que se asignan a *workers* más adelante.

Esta función recibe como argumento el nombre del arreglo que se desea convertir a un conjunto de datos distribuidos. El resultado regresado por esta función se puede combinar con demás funciones para determinar las características que posea el conjunto de datos distribuidos.

Para mayor información puede recurrir a la siguiente referencia:

https://www.mathworks.com/help/distcomp/working-with-codistributed-arrays.html

Planificación de trabajos

Como es requerida una planeación de tareas dentro del clúster del LNS un trabajo en MatLab no se puede enviar directamente desde la línea de comandos o la interfaz gráfica, ya que es probable que un administrador del sistema localice la tarea y proceda a "matarla" por estar haciendo uso sin autorizar de recursos del sistema.

Por lo anterior, para mandar tareas de cálculo en MatLab se recomienda hacerlo a través de SLURM el planeador de tareas instalado dentro del LNS. La desventaja radica en la necesidad de usar un script para meter la tarea a la cola, además de configurar ciertos parámetros para conocer la salida de tu programa y no poder ver en "tiempo real" los resultados. Sin embargo, es la mejor manera para lanzar sus aplicaciones y hacen un completo uso de todas las características del clúster.

El primer paso es realizar el script de lanzamiento para la tarea, a continuación, dejamos un ejemplo que se puede utilizar, se indican los campos que se le debe cambiar a gusto del usuario, pero servirá completamente para lanzar tareas al planeador.

Script de Ejemplo

```
#!/bin/bash

#SBATCH -J Work_Name #Nombre del trabajo
#SBATCH -o SalidaDeResultados #Archivo de Salida del trabajo
#SBATCH -p comp #SLURM QUEUE
#SBATCH -N 1
module load applications/matlab/R2016b

matlab -nodisplay < Script_Matlab.m
```

Las partes marcadas en amarillo son datos que se pueden cambiar a gusto del usuario, para adecuarlo a sus necesidades, se recomienda dejarlo tal cual esta y solo modificar estas partes. Para mayor referencia puedes consultar https://slurm.schedmd.com/quickstart.html

Los campos marcados en amarillo y se pueden cambiar, se explican a continuación:

- `#SBATCH -J Work_Name #Nombre del trabajo`
  - En esta línea se define el nombre con el cual puedes detectar tu trabajo en la cola de SLURM.

- `#SBATCH -o SalidaDeResultados #Archivo de Salida del trabajo`
    - Esta línea indica el archivo donde se guardarán los datos de salida del trabajo y todo lo que MatLab realice, este archivo es muy importante pues si existe algún error en su programa será indicado en este lugar, se va actualizando cada vez que MatLab vaya realizando una acción por lo que tiene una función de monitor de la aplicación. Su cuerpo es texto completamente plano, por lo cual no muestra elementos gráficos.
- `matlab -nodisplay < Script_MatLab.m`
    - Aquí se determina que el programa Matlab se ejecute y la salida no se muestre en la terminar que tienes abierta mediante el atributo, -nodisplay. Al final se coloca el nombre del archivo con el código de trabajo de Matlab y puede ser cambiado de acuerdo al nombre del archivo que quiera lanzar. Lamentablemente no se le podrán añadir atributos por lo cual debe definir todas sus variables dentro del código pues no hay manera de introducirlos después.

Hecho el script de SLURM se debe enviar a la cola de tareas para posteriormente hacer uso de los recursos de computo. Enviar una tarea a la cola de SLURM, es a través de los siguientes comandos.

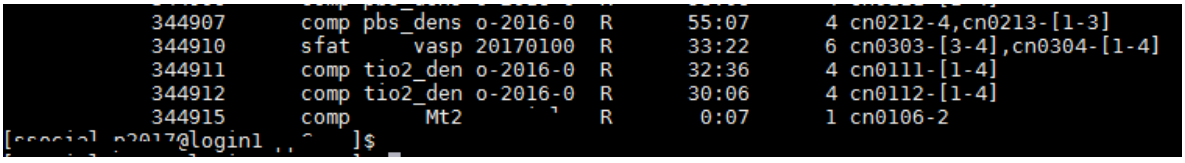
sbatch NombreDelBatch.batch

Donde la parte marcada con amarillo es el nombre de nuestro script de SLURM, al enviar este comando la tarea de MatLab descrita en el cuerpo del script será enviada a la cola para su ejecución dentro del clúster.

Si desea conocer los procesos que tiene en cola o si su proceso de añadió correctamente puede introducir la siguiente instrucción.

squeue

Y buscar sus procesos en la lista que les regrese dicho comando.



En la imagen de arriba de puede ver la tarea Mt2 en la cola de tareas, fue lanzada usando el script de ejemplo colocado en la parte de arriba.

Visualización de Resultados

Aunque existe una manera de usar la interfaz gráfica de MatLab y esta permite enviar a evaluar los scripts, no se recomienda hacerlo pues se están usando recursos del clúster de forma no autorizada lo que ocasiona que un administrador pueda matar su proceso sin una advertencia previa. La solución es enviar los trabajos mediante SLURM como lo explicamos en la parte de arriba, sin embargo, esto nos quita la posibilidad de poder visualizarlos en tiempo real y nos obliga a almacenarlos en algún archivo que podamos recuperar más adelante.

La solución a este problema está indicada en el script para planificar la tarea por SLURM

```
#!/bin/bash

#SBATCH -J Work_Name #Nombre del trabajo
#SBATCH -o SalidaDeResultados #Archivo de Salida del trabajo
#SBATCH -p comp #SLURM QUEUE
#SBATCH -N 1
module load applications/matlab/R2016b

matlab -nodisplay < Script_Matlab.m
```

Más específicamente en la línea resaltada, esta línea nos indica que todos los resultados de nuestro programa en MatLab serán escritos en el archivo que ahí le indiquemos, si el archivo no existe, SLURM lo creará de forma automática y hará la escritura de resultados en ese lugar.

Lamentablemente la explicación de arriba solo es válida para aquellos resultados en texto plano, dejando a un lado elementos gráficos. Tampoco podemos visualizar las gráficas cuando mandemos una tarea a la cola, pues el atributo *–nodisplay* del script impiden que se creen ventanas de visualización, añadiendo que en una sesión normal de trabajo dentro del clúster las opciones graficas están desactivadas y nuestro programa marcará error en la ejecución.

Para saltar esta desventaja y salvar datos gráficos, la mejor manera de hacerlo es usar los métodos nativos propios de MatLab que permiten salvar estos elementos en archivos. Este proceso no se realiza en el script de SLURM pues al ser instrucciones nativas del programa deben ser colocadas dentro del código de ejecución. MatLab provee un manual específico para salvar datos, sin embargo, en los ejemplos de este documento hay dos que indican la creación de gráficas y se muestran las salidas de esta. Para lograr este resultado, en el código expuesto arriba se muestran las siguientes instrucciones.

```
for ii=1: length(q)
  % plot each magic square
  %figure, imagesc(q{ii});
  plot(q{ii})
  saveas(gcf, strcat('ejemplo', int2str(ii)),'png');
end
delete(gcp)
```

La instrucción *saveas(gcf, strcat('ejemplo', int2str(ii)),'png');* tiene la función de guardar figuras creadas en el proceso de ejecución por la instrucción plot(q{ii}) que se encuentra inmediatamente arriba de ella. La instrucción plot crea una figura y es guardada en la variable reservada gcf, es importante advertir que esta variable se actualiza cada vez que se ejecuta una instrucción que genera una salida gráfica por lo cual es importante que la variable se guarde en un archivo al momento de ser creado o salvado en una variable aparte para realizar una manipulación posterior. Para más información en como manipular la variable gcf se muestra en la siguiente referencia:

https://www.mathworks.com/help/matlab/ref/gcf.html

Su sintaxis es la siguiente:

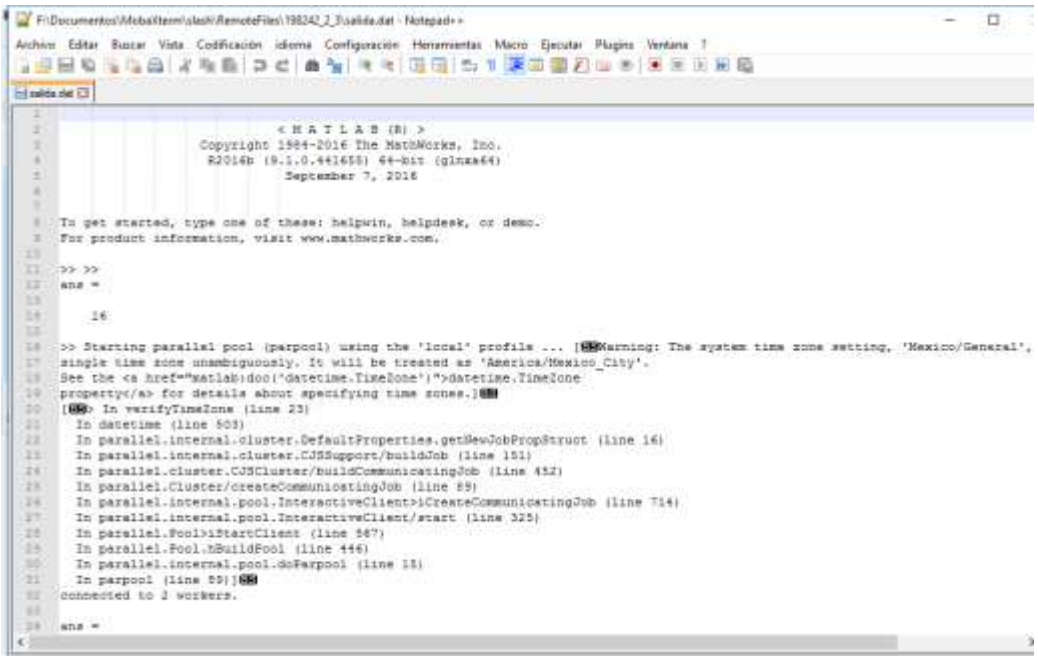
```
saveas(fig,filename)
saveas(fig,filename,formattype)
```

Donde los argumentos son los siguientes:

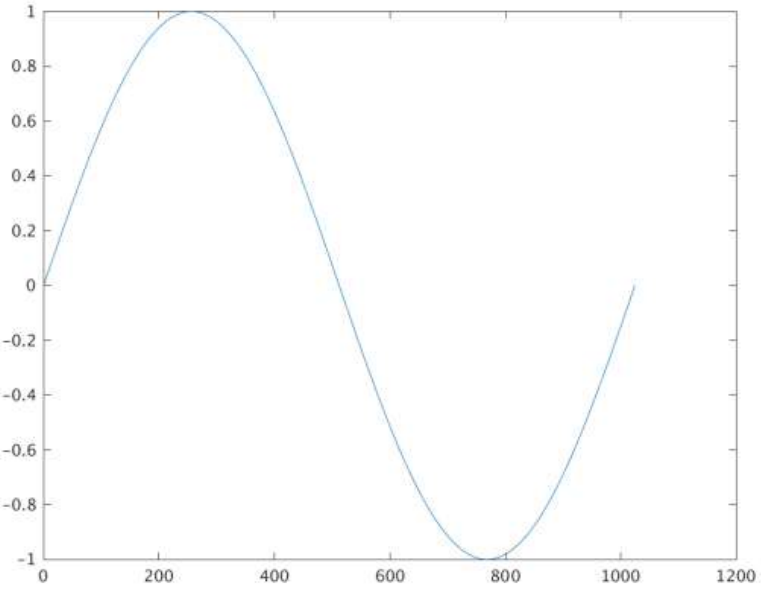
- fig — Indica la figura a guardar. La cual puede ser un figure object | Simulink block diagram.
- filename — Donde puedes especificar el nombre del fichero
- formattype — (Opcional) Que indica el formato de imagen que quieres guardar. Para más información de los archivos soportados puede consultar el manual en:

https://www.mathworks.com/help/matlab/ref/saveas.html#inputarg_fig

Este es un ejemplo de una salida en archivo de texto:



Este es un ejemplo de salida en gráfico



Referencias

VVAA. (16 de Marzo de 2016). *Quick Start User Guide.* Obtenido de SLURM Workload Manager: https://slurm.schedmd.com/quickstart.html

VVAA. (2017). *Distributed Arrays.* Obtenido de MathWorks: https://www.mathworks.com/help/distcomp/distributed-arrays.html

VVAA. (Marzo de 2017). *Parallel Computing Toolbox™ User's Guide.* Obtenido de MatLab: https://www.mathworks.com/help/pdf_doc/distcomp/distcomp.pdf

VVAA. (2017). *saveas.* Obtenido de MathWorks: https://www.mathworks.com/help/matlab/ref/saveas.html