Universidad Central de Venezuela

Facultad de Ciencias

Escuela de Computación

Asignatura: Cálculo Científico (6105)

Estudiante: Naranjo Sthory Alexanyer Antonio

Cédula de identidad: V – 26.498.600

Tarea -1: Repaso de Álgebra Lineal

En el presente informe se realiza una explicación detallada del código fuente desarrollado para la resolución de la correspondiente tarea, indicando además los algoritmos y métodos matemáticos utilizados según sea el caso a procesar. También se adjunta material visual (imágenes) que apoyen lo explicado en cada uno de las secciones del informe.

El informe se encuentra dividido en cinco (5) secciones diferentes, cada una abarcando los aspectos más importantes de la solución implementada utilizando el lenguaje de programación **Octave**. Las secciones son las siguientes, considerando que la solución implementada busca ser lo más modular posible y así facilitar la comprensión del código entregado:

1. Función principal.
2. Determinación de existencia de solución o no.
3. Cálculo de única solución.
    a. Caso **N x N.**
    b. Caso **N x M.**
4. Cálculo de infinitas soluciones.
5. Sistemas de Ecuaciones Lineales de pruebas.

1. Función principal:

Se desarrolla la función **main** como punto de partida para la ejecución del programa, en la cual, el objetivo de esta función es posibilitar la definición de los Sistemas de Ecuaciones Lineales a determinar si tienen o no solución, y en caso de tener, indicar su respectiva solución (o infinitas soluciones).

Esta función se apoya fundamentalmente de la función **hasSolution**, la cual explicaremos a detalle en la siguiente sección del informe, pero principalmente esta se encarga de recibir como parámetro una matriz **A** (matriz de coeficientes), un valor **n** (número de columnas presentes en la matriz **A**) y una matriz aumentada **Ag** (**[A|b]**, siendo **b** el vector de igualdad) y devuelve un valor numérico el cual, dependiendo del valor recibido, este permite identificar si el sistema indicado tiene o no solución y se procede a realizar las instrucciones necesarias según sea el caso en las que nos encontremos trabajando, donde podemos verificar que la función **hasSolution** puede retornar tres (**3**) posibles valores numéricos:

- Cero (**0**): Representa que el Sistema de Ecuaciones Lineales no posee solución alguna y se imprime directamente un mensaje indicando lo mencionado.
- Uno (**1**): Representa que el Sistema de Ecuaciones Lineales posee solución y además esta es única. Se invoca la función **getUniqueSolution**.
- Dos (**2**): Representa que el Sistema de Ecuaciones Lineales posee solución y además estas son infinitas. Se invoca la función **getInfiniteSolutions**.

A continuación, se adjunta una imagen que corresponde al código fuente de la función principal, la cual también fundamenta lo explicado anteriormente en esta sección del informe. También es importante destacar que esta imagen no contempla la versión final del desarrollo de esta función la cual contiene diversos casos de prueba que luego serán explicados en la sección cinco (**5**).

```
% MAIN FUNCTION
function main
  A = [-41 15 0; 109 -40 0; -3 1 0;2 0 1];
  b = [168; -447; 12; -1];

  Ag = [A b];
  result = hasSolution(A, Ag, size(A,2));

  disp('A =')
  disp(A)
  disp('b =')
  disp(b)

  if result == 0
    disp('System of Linear Equations Ax = b has no solution.')
  elseif result == 1
    disp('System of Linear Equations Ax = b has unique solution.')
    disp('Solution = ')
    disp(getUniqueSolution(A, b, Ag));
  else
    disp('System of Linear Equations Ax = b has infinites solutions.')
    getInfiniteSolutions(A,b);
  endif
endfunction
```

2. Determinación de existencia de solución o no:

Se implementó la función **hasSolution**, la cual ya ha sido mencionada en la sección anterior donde se hizo mención de que esta función tiene como tarea principal determinar si un Sistema de Ecuaciones Lineales tiene solución, ya sea única o infinitas, o no tiene solución. Esta recibe los siguientes parámetros como entrada:

- **A**: Una matriz de coeficiente que corresponde a **Ax = b.**
- **Ag**: Una matriz aumentada que corresponde a **[A|b]**.
- **n**: Un valor numérico que corresponde al número de columnas que se encuentran presentes en la matriz de coeficientes **A**.

hasSolution toma como inicio el cálculo del rango de la matriz **A** y de la matriz aumentada **[A|b]**, recordando que el rango de una matriz se define de como el *número de filas o columnas que son linealmente independientes*, esta definición es base para el desarrollo del método de **Eliminación Gaussiana.** Otra definición que se puede llegar a encontrar para el rango de una matriz es que es el orden de la mayor submatriz cuadrada no nula. Utilizando esta definición se puede calcular el rango usando determinantes.

Sabiendo qué es el rango de una matriz, podemos apoyarnos de una de las funciones integradas que ofrece tanto Octave como Matlab en su conjunto de funciones, estamos haciendo referencia a la función **rank**, que recibe como parámetro de entrada una matriz cualquiera **A** y también se puede indicar un segundo parámetro **Tol** que es la tolerancia admitida, aunque este segundo parámetro puede ser omitido, y finalmente, esta devuelve un valor numérico que representa el rango de la matriz enviada.

Conociendo el rango de las matrices necesarias, es turno de realizar unas cuantas verificaciones para así saber con exactitud si el sistema que estamos procesando tiene o no tiene solución. Para ello, es importante tomar las siguientes consideraciones:

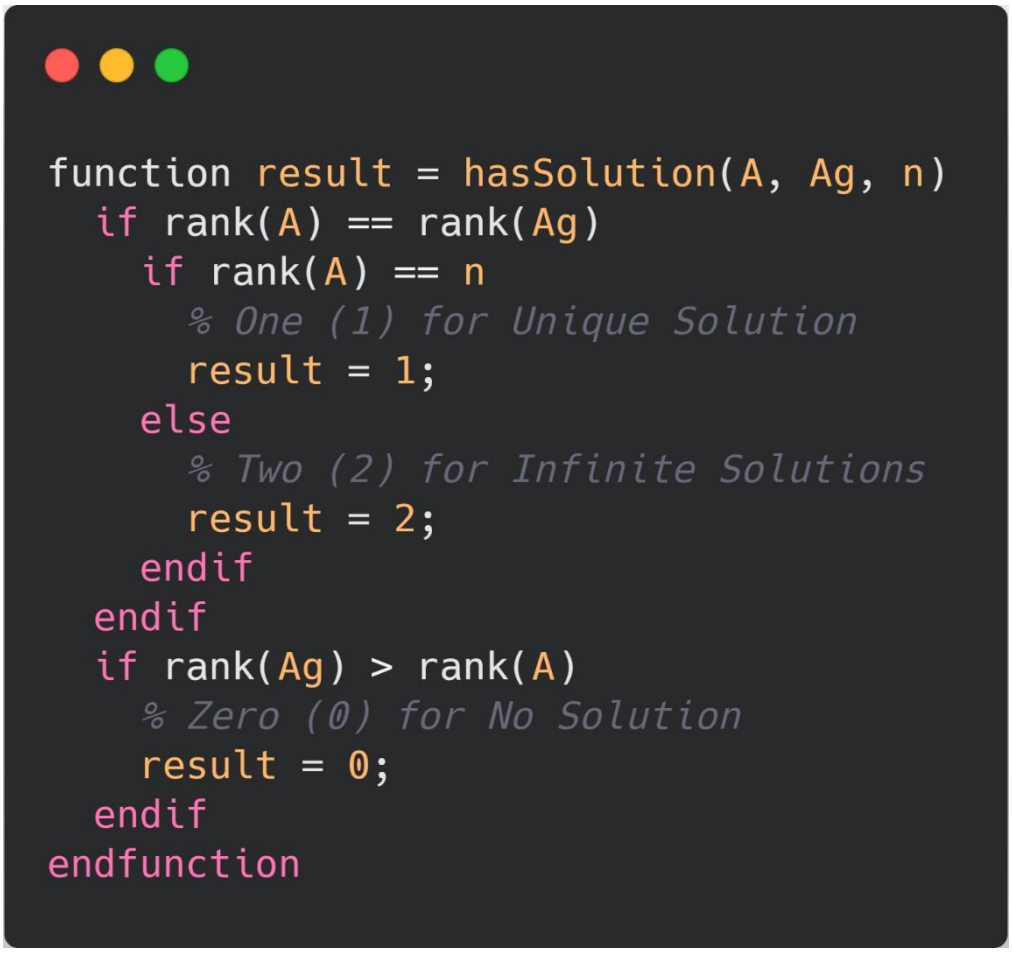
- **Si rank(A) == rank(Ag):** Esto implica que el Sistema de Ecuaciones Lineales tiene solución, pero no determina si es única o son infinitas, para ello pasamos al siguiente punto.
    - **Si rank(A) == n:** En caso de que el rango de la matriz **A** sea exactamente igual al número de columnas que esta tiene, podemos concluir que el sistema tiene **ÚNICA SOLUCIÓN** y el valor representado para este caso es el uno (**1**).
    - **Sino:** Se concluye que el Sistema de Ecuaciones Lineales tiene **INFINITAS SOLUCIONES** y el valor representado para este es el dos (**2**).

En caso de que no se cumpla ninguna de las anteriores condiciones, se puede deducir que el sistema no posee solución alguna, pero para comprobar que esto es cierto, tenemos que hacer la siguiente verificación:

- **Si rank(Ag) > rank(A):** Este caso representa que el Sistema indicado **NO TIENE SOLUCIÓN** y el valor representado para este el cero (**0**).

De esta manera, evaluamos si un Sistema de Ecuaciones Lineales tiene o no solución y el resultado es devuelto a la función **main** quien se encargará de hacer los procedimientos necesarios según haya sido el valor retornado.

Se adjunta el correspondiente fragmento código de la función **hasSolution**, para así observar en código Octave, lo explicado anteriormente en esta sección.



```
function result = hasSolution(A, Ag, n)
  if rank(A) == rank(Ag)
    if rank(A) == n
      % One (1) for Unique Solution
      result = 1;
    else
      % Two (2) for Infinite Solutions
      result = 2;
    endif
  endif
  if rank(Ag) > rank(A)
    % Zero (0) for No Solution
    result = 0;
  endif
endfunction
```

3. Cálculo de única solución:

Dividiremos esta sección del informe en dos (2) partes donde abarcaremos en cada uno el caso cuando tenemos un Sistema de Ecuaciones Lineales **N x N** (Sistema cuadrado) y otra cuando nos encontramos en el caso de **N x M** (Sistema rectangular), además, se hará mención de los algoritmos y métodos utilizados en cada uno para así llegar a una correcta solución. Pero antes de entrar en detalle con cada caso, primero se hará una visualización general de la función implementada que se encarga de la búsqueda de solución para el sistema.

Se ha definido una función cuyo nombre es **getUniqueSolution**, la cual recibe como parámetros de entrada una matriz de coeficientes **A**, un vector columna **b** y una matriz aumentada **Ag** (la cual vendría siendo **[A|b]**). Dicha función inicia con la evaluación de si la matriz **A** es cuadrada (**N x N)**, es decir, si existe un mismo número de filas o columnas, dependiendo del resultado obtenido, aplicaremos alguno de los dos (2) métodos siguientes, donde el primero es aplicado para el caso de un Sistema de Ecuaciones Lineales cuadrada y el otro es para un Sistema de Ecuaciones Lineales rectangular respectivamente:

- Eliminación Gaussiana
- Factorización o Descomposición QR

En las siguientes subsecciones, se explicará al detalle cada uno de estos métodos y su respectiva implementación en el código fuente.

3.1. Caso N x N (*Sistema cuadrado*):

En caso de encontrarnos con un Sistema de Ecuaciones Lineales, aplicaremos el método de eliminación Gaussiana, la cual consiste de lo siguiente:

El método de **Eliminación Gaussiana** para la solución de sistemas de ecuaciones lineales consiste en convertir a través de operaciones básicas llamadas operaciones de filas (o columnas) un sistema en otro *equivalente* más sencillo cuya respuesta pueda leerse de manera directa. El método de eliminación Gaussiana es el mismo para sistemas de ecuaciones 2×2, 3×3, 4×4, y así sucesivamente siempre y cuando se respete la relación de al menos una ecuación por cada variable.

Es importante destacar que solamente contamos con las siguientes operaciones por filas (o columnas) que pueden ser aplicadas al momento de resolver un sistema de ecuaciones lineales a través de este método, estas operaciones son las siguientes:

- Intercambio: Podemos intercambiar el orden de las filas (o columnas).
- Multiplicación por una constante: Podemos multiplicar una fila (o columna) por un valor cualquier, siempre y cuando este sea diferente de cero (0).
- Suma o resta de filas (o columnas): Podemos tomar una fila y sumar o restar estar por otra fila multiplicada o no por cualquier número diferente de cero.

También es importante destacar que no podemos combinar en la resolución de un sistema de ecuaciones, las operaciones por filas y las operaciones por columnas, es decir, si empezamos el desarrollo de la solución operando por filas, el resto debe continuar de esta manera o viceversa. Se acostumbra a trabajar por filas, pero existe la posibilidad de trabajar por columnas.

Este proceso debe aplicarse hasta que se obtenga una **Matriz en forma escalonada** (*Método de Gauss*) o en forma de **escalonada reducida** (*Método Gauss-Jordan*).

Recordando que una matriz en su forma escalonada cumple con lo siguiente:

1. Todos los renglones cero están en la parte inferior de la matriz.
2. El elemento delantero de cada reglón diferente de cero está a la derecha del elemento delantero diferente de cero del reglón anterior.
3. El primero elemento diferente de 0 y 1 de cada fila está a la derecha del primer elemento diferente de 0.

Por ejemplo, la siguiente matriz se encuentra en su forma escalonada:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & 4 \\ 0 & 0 & 1 \end{pmatrix}$$

Habiendo hallado una matriz equivalente a la original a través de las operaciones por reglones y esta se encuentra en su forma escalonada, podemos aplicar la **vuelta hacia atrás** o **sustitución regresiva**, la cual consiste de devolver al Sistema de Ecuaciones Lineales de su forma de matriz aumentada a su forma de ecuaciones e ir resolviendo cada ecuación con las variables conocidas, una a una.

También, a partir de la matriz equivalente calculada en su forma de escalonada, podemos calcular su forma escalonada reducida aplicando las operaciones por filas, pero en sentido inverso, de esta manera, obtener una matriz aumentada que podría verse de la siguiente manera:

$$[A|b] = \begin{pmatrix} 1 & 0 & 0 & | & 3 \\ 0 & 1 & 0 & | & 4 \\ 0 & 0 & 1 & | & 5 \end{pmatrix}$$

Ya conociendo la base teórica del método de Eliminación Gaussiana, la cual consiste de aplicar operaciones elementales de filas para obtener una matriz equivalente a la original en su forma escalonada o escalonada reducida, también es importante saber que existe un algoritmo para poder buscar estas formas de manera sencilla y dicho algoritmo es el utilizado para la implementación de este método.

El algoritmo es el siguiente:

1. Comenzamos por conseguir un elemento pivote **1** en la primera fila. Si en la primera columna hay algún elemento no nulo, este pivote lo conseguiremos en la posición **(1,1).** Si todos los elementos de la primera columna son cero, pasamos a intentarlo en la segunda posición, **(1,2)**, y así sucesivamente.
2. A cada una de las restantes filas se le hace **0** al elemento que cae bajo el pivote sumándole la primera multiplicada por el escalar conveniente. Una vez hecho esto, la matriz puede tener la siguiente forma, por ejemplo,

$$\begin{pmatrix} 1 & \dots & \dots & | & b1 \\ 0 & \dots & \dots & | & b2 \\ 0 & \dots & \dots & | & b3 \end{pmatrix}$$

3. Hemos de repetir el proceso (paso 1 y 2) para las siguientes filas, hasta que no haya más o todos los elementos de las filas que queden sean cero. Así, obtendremos una matriz escalonada.
4. Finalmente, con el pivote **1** de cada fila no nula se hace **0** el término correspondiente de todas las anteriores, con lo que la matriz resultante será escalonada reducida.

Partiendo de este algoritmo, nuestro código escrito en Octave para hallar la forma escalonada reducida de una matriz aumentada **[A|b]**, quedaría de la siguiente manera.

```
function solution = getUniqueSolution(A, b, Ag)
  % It's squared
  if size(A,1) == size(A,2)
    % Number of rows
    for i=1:size(Ag,1)
      % Make diagonal entry as 1
      Ag(i,:) = Ag(i,:)./Ag(i,i);
      for j = 1:size(Ag,1)
        if j~=i
          % Define PIVOT key element
          pivot = Ag(j,i)./Ag(i,i);
          % Apply elementary row operations
          Ag(j,:) = Ag(j,:)-pivot.*Ag(i,:);
        endif
      endfor
    endfor
    % return vector b with solution
    solution = Ag(:,end);
    ...
```

Se aprecia únicamente la solución implementada para un Sistema de Ecuaciones Lineales **N x N,** en la siguiente subsección explicaremos el método implementado en caso de tener un sistema rectangular.

3.2. Caso N x M (*Sistema rectangular*):

Para llevar a cabo una implementación capaz de hallar la solución única de un Sistema de Ecuaciones Lineales **N x M**, se optó por desarrollar la *Factorización o Descomposición QR*, el cual, antes de visualizar el código fuente correspondiente a este, entraremos primero en una breve teoría de qué consiste, cuándo aplicarlo y cómo trabaja exactamente este método propio del álgebra lineal.

3.2.1. Factorización QR:

La descomposición QR (también llamada factorización QR) de una matriz es una descomposición de la matriz en una matriz ortogonal y una matriz triangular. La descomposición QR de una matriz real A es una descomposición de A como,

$$A = QR,$$

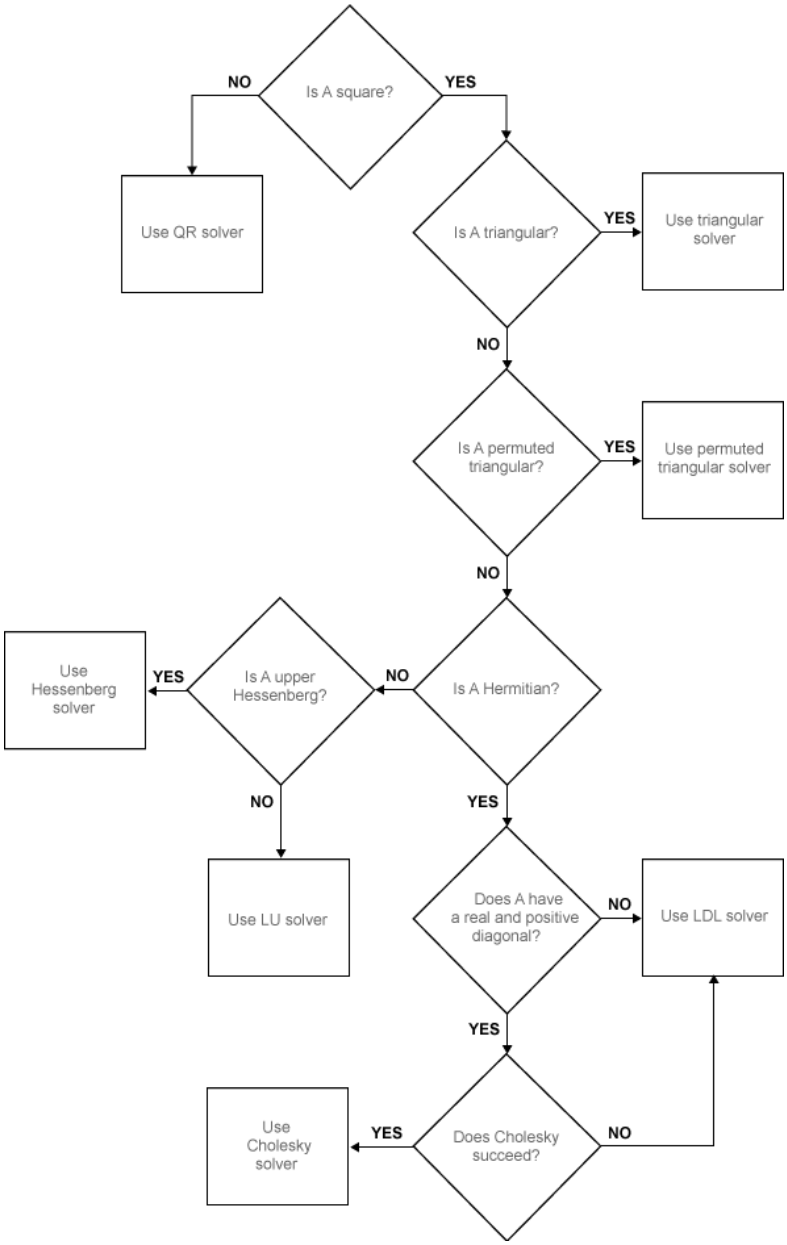
donde Q es una matriz ortogonal (es decir, $Q^T\ Q\ =\ I$) y R es una matriz triangular superior. Si A es no singular, esta factorización es única. Existen varios métodos para calcular la descomposición QR, entre los más conocidos tenemos:

- Las rotaciones de Givens
- Los reflectores de Householder
- El proceso de ortogonalización de Gramm-Schmidt

Es importante conocer que la factorización de una matriz en álgebra lineal consiste de técnicas que permiten escribir una matriz como producto de dos o tres matrices con una estructura especial. La factorización de matrices es importante por ejemplo cuando se quiere resolver sistemas de ecuaciones con un número muy grande tanto de variables como de ecuaciones, pero también cuando se quieren resolver sistemas simultáneos de ecuaciones.

El motivo de haber optado por investigar, indagar e implementar la factorización QR es debido a la forma en cómo los lenguajes de programación, por ejemplo, tanto Matlab como Octave, buscan resolver sistemas de ecuaciones lineales a través de funciones integradas de estos.

Si revisamos la documentación de parte de Matlab, este trae consigo la función **mldivide**, la cual permite la resolución de Sistemas de Ecuaciones Lineales, pero, además, ofrece el siguiente diagrama de flujo,



Donde observamos que la primera decisión que toma es en base a si la matriz de coeficientes **A** es cuadrada o no, de no ser cuadrada, recurre a la Factorización QR para, como se ha mencionado anteriormente, obtener una descomposición de la matriz **A** en una multiplicación de las matrices **QR**.

Partiendo de esto, se buscó aplicar alguno de los métodos conocidos para lograr esta factorización, donde se finalmente se implementó el método de los *reflectores de Householder,* la cual explicaremos en la siguiente subsección y se mostrará el código fuente correspondiente a este.

3.2.3. Reflectores de Householder:

El método de Householder aplica una sucesión de matrices unitarias Qk por la izquierda de A:

$$Q_n \dots Q_2 Q_1 A = R$$

$$Q_n \dots Q_2 Q_1 A = Q^*$$

$$Q = Q *_1 Q *_2 \dots Q *_n$$

Y la matriz resultando R, es una matriz triangular superior. El producto de Q^*, es una matriz unitaria, por lo que se obtiene la factorización QR completa de A.

Este método fue propuesto por Alston Householder en el año 1958, y es una forma de diseñar matrices unitarias Q_k, que realicen las siguientes operaciones.

$$\underset{A}{\begin{bmatrix} x & x & x \\ x & x & x \\ x & x & x \\ x & x & x \\ x & x & x \end{bmatrix}} \xrightarrow{Q_1} \underset{Q_1 A}{\begin{bmatrix} * & * & * \\ 0 & * & * \\ 0 & * & * \\ 0 & * & * \\ 0 & * & * \end{bmatrix}} \xrightarrow{Q_2} \underset{Q_2 Q_1 A}{\begin{bmatrix} x & x & x \\ 0 & * & * \\ 0 & 0 & * \\ 0 & 0 & * \\ 0 & 0 & * \end{bmatrix}} \xrightarrow{Q_3} \underset{Q_3 Q_2 Q_1 A}{\begin{bmatrix} x & x & x \\ 0 & x & x \\ 0 & 0 & * \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}}$$

Donde,

- **x**: Elemento de la matriz, no necesariamente cero.
- *****: Elemento que se ha modificado recientemente.

En general, Q_k, opera sobre las filas k, …, m.

Sabiendo que, Q_k, tiene la siguiente forma:

$$Q_k = \begin{pmatrix} I & 0 \\ 0 & F \end{pmatrix},$$

Considerando que:

- I es la matriz identidad y tiene dimensión (k -1) x (k -1).
- F es el reflector de Householder y tiene dimensión (m – k +1) x (m – k +1), y es una matriz unitaria.

Acotación: Ha sido excelente leer la sección del libro Análisis Numérico, redactado por usted y sus compañeros, sobre factorización QR y factorización LU. Realmente no tengo conocimiento alguno si serán temas tratados a futuro, pero me encantaría que fuese así. Quisiera comprender en su totalidad estos temas, dado a que solamente investigué e implementé todo en base a la asignación, aunque sé que hay muchas más utilidades para esta parte tan interesante.

A continuación, se presenta el código fuente implementado para el desarrollo de la factorización QR donde se desarrolló la función **myQr**, observando que este método devuelve dos matrices, **Q** y **R,** los cuales son los resultados de realizar este método sobre la matriz enviada como parámetro, **A.** Luego de recibir estos resultados dentro de la función **getUniqueSolution**, se procede a realizar el correspondiente cálculo para la búsqueda de la solución del Sistema de Ecuaciones Lineales que estemos tratando actualmente al momento de invocar la función.

```
function [Q, R] = myQr(A)
  [m,n] = size(A);
  if (m > n)
    % Initialization
    R = zeros(m,n);
    Q = eye(m,m);
    z = zeros(m);
    for j = 1:n
      % Reflection of column vector of A
      y = A(j:end,j);
      w = y +sign(A(j,j))*norm(y)*eye(size(y,1),1);
      v = w/norm(w);
      d = 2*(v*v');
      z(j:end,j:end) = d;
      % Generating Householder Matrix
      H = eye(m) - z;
      % Calculating new matrix A using H*A
      A = H*A;
      % Calculating orthogonal matrix, Q using Q=H1*H2*....*Hn
      Q = Q*H;
      z = zeros(m);
    endfor
  else
    % Initialization
    R = zeros(m,n);
    Q = eye(m,m);
    z = zeros(m);
    for j = 1:m-1
      % Reflection of column vector of A
      y = A(j:end,j);
      w = y +sign(A(j,j))*norm(y)*eye(size(y,1),1);
      v = w/norm(w);
      d = 2*(v*v');
      z(j:end,j:end) = d;
      % Generating Householder Matrix
      H = eye(m) - z;
      % Calculating new matrix A using H*A
      A = H*A;
      % Calculating orthogonal matrix, Q using Q=H1*H2*....*Hn
      Q = Q*H;
      z = zeros(m);
    endfor
  endif
  % Forming the R matrix, R = A
  for i = 1:m
    for j = i:n
      R(i,j) = A(i,j);
    endfor
  endfor
endfunction
```

3.3. Función getUniqueSolution:

Ya habiendo explicado cada uno de los métodos utilizados y observado el código fuente de cada uno, a continuación, se adjunta por completo el cuerpo de la función definida **getUniqueSolutión,** que retorna finalmente un vector *solution* que contiene las soluciones correspondientes al Sistema de Ecuaciones Lineales pasado por parámetro en la invocación de dicha función.

```
function solution = getUniqueSolution(A, b, Ag)
  % It's squared
  if size(A,1) == size(A,2)
    for i=1:size(Ag,1)
      Ag(i,:) = Ag(i,:)./Ag(i,i);
      for j = 1:size(Ag,1)
        if j~=i
          pivot = Ag(j,i)./Ag(i,i);
          Ag(j,:) = Ag(j,:)-pivot.*Ag(i,:);
        endif
      endfor
    endfor
    solution = Ag(:,end);
  else
  % It is not square
    [Q,R] = myQr(A);
    solution = R \ (Q' * b);
  endif
endfunction
```

4. Cálculo de infinitas soluciones:

Finalmente, en caso de procesar un Sistema de Ecuaciones Lineales que tenga infinitas soluciones, se optó por implementar la *Eliminación Gaussiana* apoyándonos de las funciones para notación simbólica que trae consigo Octave. Nos estamos haciendo referencia a las funciones **sym** y **syms**, las cuales nos ayudarán a indicar la solución del sistema de ecuaciones en su forma paramétrica. Además, funciones también se encuentran disponibles en el conjunto de funciones integradas que trae consigo Matlab.

> **NOTA IMPORTANTE:** Es importante ejecutar el comando **pkg load symbolic**, en la ventana de comandos de Octave para así poder cargar el paquete correspondiente del lenguaje y así poder hacer uso de las funciones anteriormente mencionadas para realizar representaciones simbólicas.

Como se ha explicado en anteriores secciones del informe, la *Eliminación Gaussiana* consiste en operar sobre la matriz ampliada del sistema hasta hallar la forma escalonada (una matriz triangular superior). Así, se obtiene un sistema fácil de resolver por sustitución hacía atrás. Si finalizamos las operaciones al hallar la forma escalonada reducida (forma lo más parecida a la matriz identidad), entonces el método se denomina eliminación de *Gauss-Jordan*.

Para este caso, se determina la cantidad de variables *libres* que el sistema tendrá disponibles al momento de su resolución y se procede a resolver las ecuaciones necesarias en base a las variables libres que estén presentes, de esta manera podemos hallar una forma general para representar las infinitas soluciones que el sistema de ecuaciones pueda llegar a tener.

Se destaca que la función desarrollada, **getInfiniteSolutions**, se encuentra intradocumentada en su totalidad para así detallas el *paso a paso* que se ha hecho para obtener los resultados deseados.

```
function getInfiniteSolutions(A, b)
  [m,n] = size(A);
  n = min(m,n);

  Au = [A,b];
  Au = sym(Au);
  kA = rank(A);
  kAu = rank(Au);

  % the matrix A is irregular, SLE has infinitely many solution
  % in this case we need compute the base of solution and a
  %general solution as a linear
  % combination of the base vectors.

  R=rref(Au) % create augumented matrix

  % create n variables, number of columns in matix A=number of variables
  x = sym('x',[1 n])
  a = sym('a', [1 n])
  r = rank(R);
  p = n-r; %find out number of free variables
  t = sym('t', [1 p])

  % create empty symbolic arrays to store equations and basic and free variables
  eqn = sym(zeros(r,n));
  eqns = sym(zeros(r,1));
  vrb = sym(zeros(1,r));
  bsc = sym(zeros(1,n));

  % create symbolic equations from matrix so it can be solved by the
  % 'solve' matlab function
  for i=1:r
    for j=1:n
      %create symbollic equations from reducer row echelon matrix
      eqn(i,j) = R(i,j)*a(j);
      eqns(i) = eqns(i)+eqn(i,j);
      %find out basic variables
      if R(i,j) ~= 0 && vrb(i) ==0
        vrb(i)= x(j);
        bsc(j)= a(j);
      endif
    endfor
    eqns(i)=eqns(i)==R(i,end); %add right side to the equations
  endfor
  frees = a-bsc; %find out free variables by substracting basic from all variables
  free = nonzeros(frees).'; %store free variables
  bsc = nonzeros(bsc).'; %store basic variables

  t = flip(t);
  eqnss = subs(eqns,bsc,vrb); %replace basic variables with x
  eqnss = subs(eqnss,free,t); %replace free variables with parameter t

  S = solve(eqnss,vrb); %solve the equations

  %create vector with solution
  tt = subs(frees,free,t);
  idx = find(tt == 0);

  fprintf('%s',"Parametric solution of a SLE:");
  disp(tt.');
  fprintf('%s',"Where ");
  disp(t);

  if length(t) == 1
    fprintf('%s'," is arbitrary real number");
  else
    fprintf('%s'," are arbitrary real numbers");
  endif
endfunction
```

5. Sistemas Lineales de Ecuaciones de pruebas:

En la función **main**, se encuentran en forma de comentarios, algunos ejemplos utilizados de Sistemas de Ecuaciones Lineales y así realizar un proceso de *testing* de cada una de las funciones definidas en el código fuente de la asignación, tratando de abarcar los diferentes casos que puede llegar a presentarse al momento el programa. Varios ejemplos han sido obtenidos de las diferentes bibliografías ofrecidas por el grupo docente.

Se destaca que se han definido dos (2) variables principales para así ejecutar el programa correctamente:

- **A**: La cual es la correspondiente matriz de coeficientes del sistema de ecuaciones lineales a procesar.
- **b**: Un vector columna que representa las igualdades de cada una las ecuaciones presentes en el sistema.

A continuación, se presenta una visualización de la función **main** final junto con los comentarios anteriormente mencionados en esta sección.

```
% MAIN FUNCTION
function main
  % UNIQUE SOLUTION - CASE #1
  A = [-41 15 0; 109 -40 0; -3 1 0;2 0 1];
  b = [168; -447; 12; -1];

  % UNIQUE SOLUTION - CASE #2
  %A = [1 0; 0 1];
  %b = [1; 1];

  % NO SOLUTION - CASE #3
  %A = [1 0; 0 1; 0 0];
  %b = [1; 1; 1];

  % Note: run "pkg load symbolic" from the Octave prompt
  % INFINITE SOLUTIONS - CASE #4
  %A = [1 0; 0 0];
  %b = [1; 0];

  Ag = [A b];
  result = hasSolution(A, Ag, size(A,2));

  disp('A =')
  disp(A)
  disp('b =')
  disp(b)

  if result == 0
    disp('System of Linear Equations Ax = b has no solution.')
  elseif result == 1
    disp('System of Linear Equations Ax = b has unique solution.')
    disp('Solution = ')
    disp(getUniqueSolution(A, b, Ag));
  else
    disp('System of Linear Equations Ax = b has infinites solutions.')
    getInfiniteSolutions(A,b);
  endif
endfunction
```