Notes on the FLAME APIs

Robert A. van de Geijn
The University of Texas at Austin
Austin, TX 78712

September 24, 2014

NOTE: I have not thoroughly proof-read these notes!!!

1 Motivation

In the course so far, we have frequently used the "FLAME Notation" to express linear algebra algorithms. In this note we show how to translate such algorithms into code, using various QR factorization algorithms as examples.

2 Install FLAME@lab

The API we will use we refer to as the "FLAME@lab" API, which is an API that targets the M-script language used by Matlab and Octave (an Open Source Matlab implementation). This API is very intuitive, and hence we will spend (almost) no time explaining it.

Download all files from `http://www.cs.utexas.edu/users/flame/Notes/FLAMEatlab/` and place them in the same directory as you will the remaining files that you will create as part of the exercises in this document. (Unless you know how to set up paths in Matlab/Octave, in which case you can put it whereever you please, and set the path.)

3 An Example: Gram-Schmidt Orthogonalization

Let us start by considering the various Gram-Schmidt based QR factorization algorithms from "Notes on Gram-Schmidt QR Factorization", typeset using the FLAME Notation in Figure 2.

3.1 The Spark Webpage

We wish to typeset the code so that it closely resembles the algorithms in Figure 2. The FLAME notation itself uses "white space" to better convey the algorithms. We want to do the same for the codes that implement the algorithms. However, typesetting that code is somewhat bothersome because of the careful spacing that is required. For this reason, we created a webpage that creates a "code skeleton.". We call this page the "Spark" page:

`http://www.cs.utexas.edu/users/flame/Spark/`.

When you open the link, you will get a page that looks something like the picture in Figure 3.

3.2 Implementing CGS with FLAME@lab

.

$[y^\perp, r] = \text{Proj\_orthog\_to\_Q}_{\text{CGS}}(Q, y)$ (used by classical Gram-Schmidt)	$[y^\perp, r] = \text{Proj\_orthog\_to\_Q}_{\text{MGS}}(Q, y)$ (used by modified Gram-Schmidt)
$y^\perp = y$	$y^\perp = y$
for $i = 0, \ldots, k-1$	for $i = 0, \ldots, k-1$
$\quad\rho_i := q_i^H y$	$\quad\rho_i := q_i^H y^\perp$
$\quad y^\perp := y^\perp - \rho_i q_i$	$\quad y^\perp := y^\perp - \rho_i q_i$
endfor	endfor

Figure 1: Two different ways of computing $y^\perp = (I - QQ^H)y$, the component of y orthogonal to $\mathcal{C}(Q)$, where Q has k orthonormal columns.

```
Algorithm: $[A, R] := \text{Gram-Schmidt}(A)$    (overwrites $A$ with $Q$)

Partition $A \rightarrow \left( A_L \,\|\, A_R \right)$, $R \rightarrow \left( \begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array} \right)$
    where $A_L$ has 0 columns and $R_{TL}$ is $0 \times 0$
while $n(A_L) \neq n(A)$ do
    Repartition
        $\left( A_L \,\|\, A_R \right) \rightarrow \left( A_0 \,\|\, a_1 \,|\, A_2 \right)$, $\left( \begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array} \right) \rightarrow \left( \begin{array}{c|c|c} R_{00} & r_{01} & R_{02} \\ \hline 0 & \rho_{11} & r_{12}^T \\ \hline 0 & 0 & R_{22} \end{array} \right)$
        where $a_1$ and $q_1$ are columns, $\rho_{11}$ is a scalar

    CGS                            | MGS                                                        | MGS (alternative)
    $r_{01} := A_0^H a_1$          |                                                            |
    $a_1 := a_1 - A_0 r_{01}$      | $[a_1, r_{01}] = \text{Proj\_orthog\_to\_Q}_{\text{MGS}}(A_0, a_1)$ |
    $\rho_{11} := \|a_1\|_2$       | $\rho_{11} := \|a_1\|_2$                                   | $\rho_{11} := \|a_1\|_2$
    $a_1 := a_1 / \rho_{11}$       | $q_1 := a_1 / \rho_{11}$                                   | $a_1 := a_1 / \rho_{11}$
                                   |                                                            | $r_{12}^T := a_1^H A_2$
                                   |                                                            | $A_2 := A_2 - a_1 r_{12}^T$

    Continue with
        $\left( A_L \,\|\, A_R \right) \leftarrow \left( A_0 \,|\, a_1 \,\|\, A_2 \right)$, $\left( \begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array} \right) \leftarrow \left( \begin{array}{c|c|c} R_{00} & r_{01} & R_{02} \\ \hline 0 & \rho_{11} & r_{12}^T \\ \hline 0 & 0 & R_{22} \end{array} \right)$
endwhile
```

Figure 2: Left: Classical Gram-Schmidt algorithm. Middle: Modified Gram-Schmidt algorithm. Right: Modified Gram-Schmidt algorithm where every time a new column of Q, q_1 is computed the component of all future columns in the direction of this new vector are subtracted out.

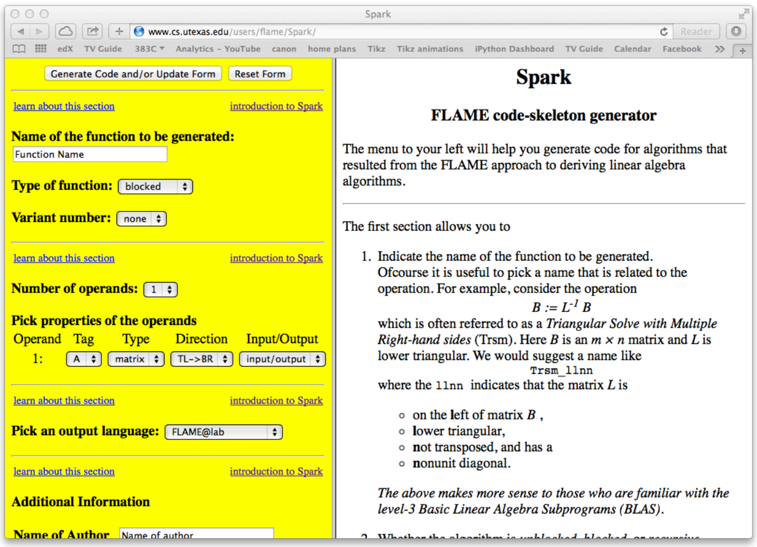
We will focus on the Classical Gram-Schmidt algorithm on the left, which we show by itself in Figure 4 (left). To its right, we show how the menu on the left side of the Spark webpage needs to be filled out.

Some comments:

Name: Choose a name that describes the algorithm/operation being implemented.

Type of function: Later you will learn about "blocked" algorithms. For now, we implement "unblocked" algorithms.

Variant number: Notice that there are a number of algorithmic variants for implementing the Gram-



Spark

www.cs.utexas.edu/users/flame/Spark/

Generate Code and/or Update Form | Reset Form

learn about this section | introduction to Spark

Name of the function to be generated: Function Name

Type of function: blocked

Variant number: none

learn about this section | introduction to Spark

Number of operands: 1

Pick properties of the operands

Operand	Tag	Type	Direction	Input/Output
1:	A	matrix	TL->BR	input/output

learn about this section | introduction to Spark

Pick an output language: FLAME@lab

learn about this section | introduction to Spark

Additional Information

Name of Author Name of author

Spark

FLAME code-skeleton generator

The menu to your left will help you generate code for algorithms that resulted from the FLAME approach to deriving linear algebra algorithms.

The first section allows you to

1. Indicate the name of the function to be generated. Ofcourse it is useful to pick a name that is related to the operation. For example, consider the operation
   $B := L^{-1} B$
   which is often referred to as a *Triangular Solve with Multiple Right-hand sides* (Trsm). Here B is an $m \times n$ matrix and L is lower triangular. We would suggest a name like
   `Trsm_llnn`
   where the `llnn` indicates that the matrix L is
   - on the left of matrix B ,
   - lower triangular,
   - not transposed, and has a
   - nonunit diagonal.

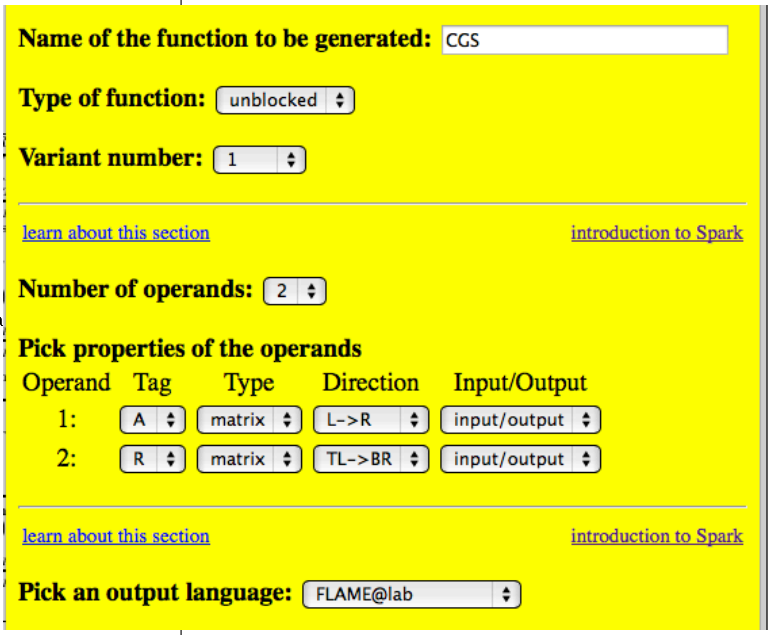
   The above makes more sense to those who are familiar with the level-3 Basic Linear Algebra Subprograms (BLAS).

Figure 3: The Spark webpage.

```
Algorithm: $[A, R] := \text{Gram-Schmidt}(A)$   (overwrites $A$ with $Q$)
Partition $A \rightarrow \left( A_L \,\middle|\, A_R \right)$,
  $R \rightarrow \left( \begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array} \right)$
  where $A_L$ has 0 columns and $R_{TL}$ is $0 \times 0$
while $n(A_L) \neq n(A)$ do
  Repartition
    $\left( A_L \,\middle|\, A_R \right) \rightarrow \left( A_0 \,\middle|\, a_1 \,\middle|\, A_2 \right)$,
    $\left( \begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array} \right) \rightarrow \left( \begin{array}{c|c|c} R_{00} & r_{01} & R_{02} \\ \hline 0 & \rho_{11} & r_{12}^T \\ \hline 0 & 0 & R_{22} \end{array} \right)$
    where $a_1$ and $q_1$ are columns, $\rho_{11}$ is a scala

    $r_{01} := A_0^H a_1$
    $a_1 := a_1 - A_0 r_{01}$
    $\rho_{11} := \|a_1\|_2$
    $a_1 := a_1 / \rho_{11}$

  Continue with
    $\left( A_L \,\middle|\, A_R \right) \leftarrow \left( A_0 \,\middle|\, a_1 \,\middle|\, A_2 \right)$,
    $\left( \begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array} \right) \leftarrow \left( \begin{array}{c|c|c} R_{00} & r_{01} & R_{02} \\ \hline 0 & \rho_{11} & r_{12}^T \\ \hline 0 & 0 & R_{22} \end{array} \right)$
endwhile
```



Name of the function to be generated: CGS

Type of function: unblocked

Variant number: 1

learn about this section | introduction to Spark

Number of operands: 2

Pick properties of the operands

Operand	Tag	Type	Direction	Input/Output
1:	A	matrix	L->R	input/output
2:	R	matrix	TL->BR	input/output

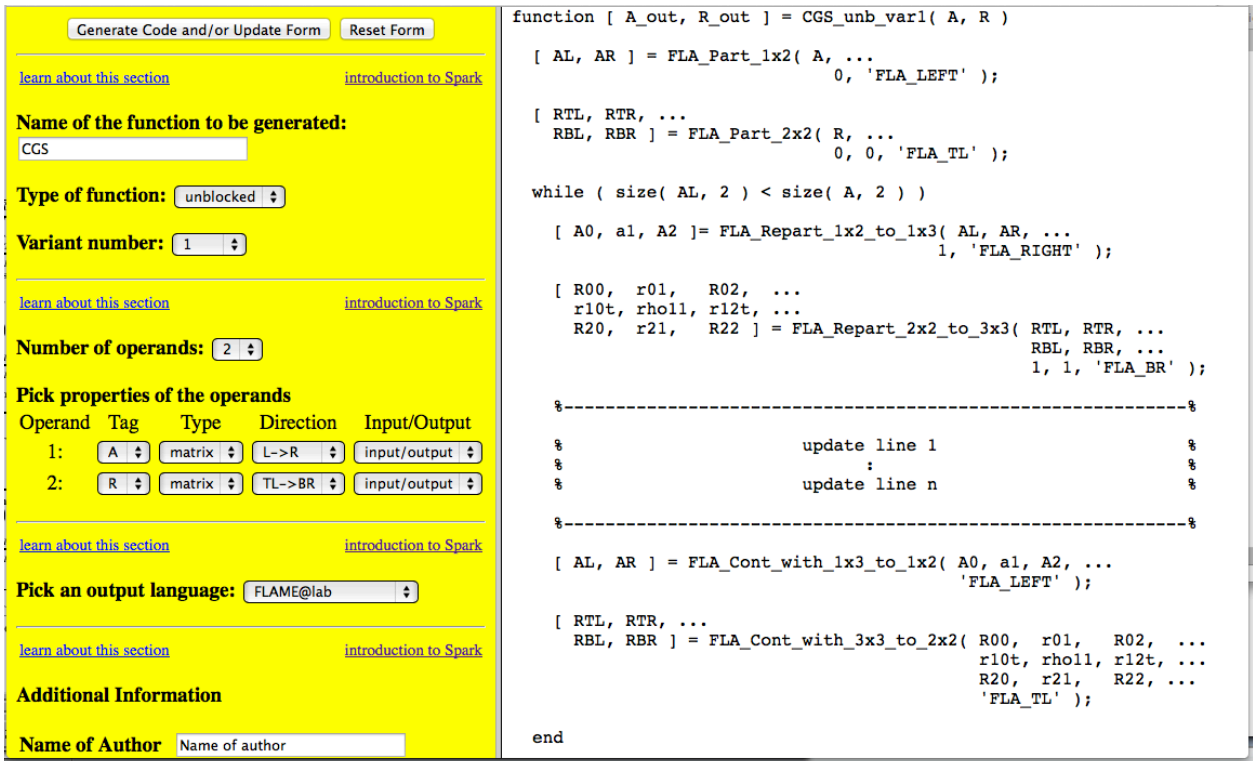
learn about this section | introduction to Spark

Pick an output language: FLAME@lab

Figure 4: Left: Classical Gram-Schmidt algorithm. Right: Generated code-skeleton for CGS.

Schmidt algorithm. We choose to call the first one "Variant 1".

Number of operands: This routine requires two operands: one each for matrices A and R. (A will be



Name of the function to be generated: CGS

Type of function: unblocked

Variant number: 1

Number of operands: 2

Pick properties of the operands

Operand	Tag	Type	Direction	Input/Output
1:	A	matrix	L->R	input/output
2:	R	matrix	TL->BR	input/output

Pick an output language: FLAME@lab

Additional Information

Name of Author Name of author

```
function [ A_out, R_out ] = CGS_unb_var1( A, R )

  [ AL, AR ] = FLA_Part_1x2( A, ...
                               0, 'FLA_LEFT' );

  [ RTL, RTR, ...
    RBL, RBR ] = FLA_Part_2x2( R, ...
                               0, 0, 'FLA_TL' );

  while ( size( AL, 2 ) < size( A, 2 ) )

    [ A0, a1, A2 ]= FLA_Repart_1x2_to_1x3( AL, AR, ...
                                        1, 'FLA_RIGHT' );

    [ R00,  r01,   R02,  ...
      r10t, rho11, r12t, ...
      R20,  r21,   R22 ] = FLA_Repart_2x2_to_3x3( RTL, RTR, ...
                                                  RBL, RBR, ...
                                                  1, 1, 'FLA_BR' );

    %------------------------------------------------------------%

    %                      update line 1                         %
    %                            :                               %
    %                      update line n                         %

    %------------------------------------------------------------%

    [ AL, AR ] = FLA_Cont_with_1x3_to_1x2( A0, a1, A2, ...
                                           'FLA_LEFT' );

    [ RTL, RTR, ...
      RBL, RBR ] = FLA_Cont_with_3x3_to_2x2( R00,  r01,   R02,  ...
                                             r10t, rho11, r12t, ...
                                             R20,  r21,   R22, ...
                                             'FLA_TL' );

  end
```

Figure 5: The Spark webpage filled out for CGS Variant 1.

overwritten by the matrix Q.)

Operand 1: We indicate that A is a matrix through which we "march" from left to right (`L->R`) and it is both input and output.

Operand 2: We indicate that R is a matrix through which we "march" from to-left to bottom-right (`TL->BR`) and it is both input and output. Our API requires you to pass in the array in which to put an output, so an appropriately sized R must be passed in.

Pick and output language: A number of different representations are supported, including APIs for M-script (FLAME@lab), C (FLAMEC), LaTeX(FLaTeX), and Python (FlamePy). Pick FLAME@lab.

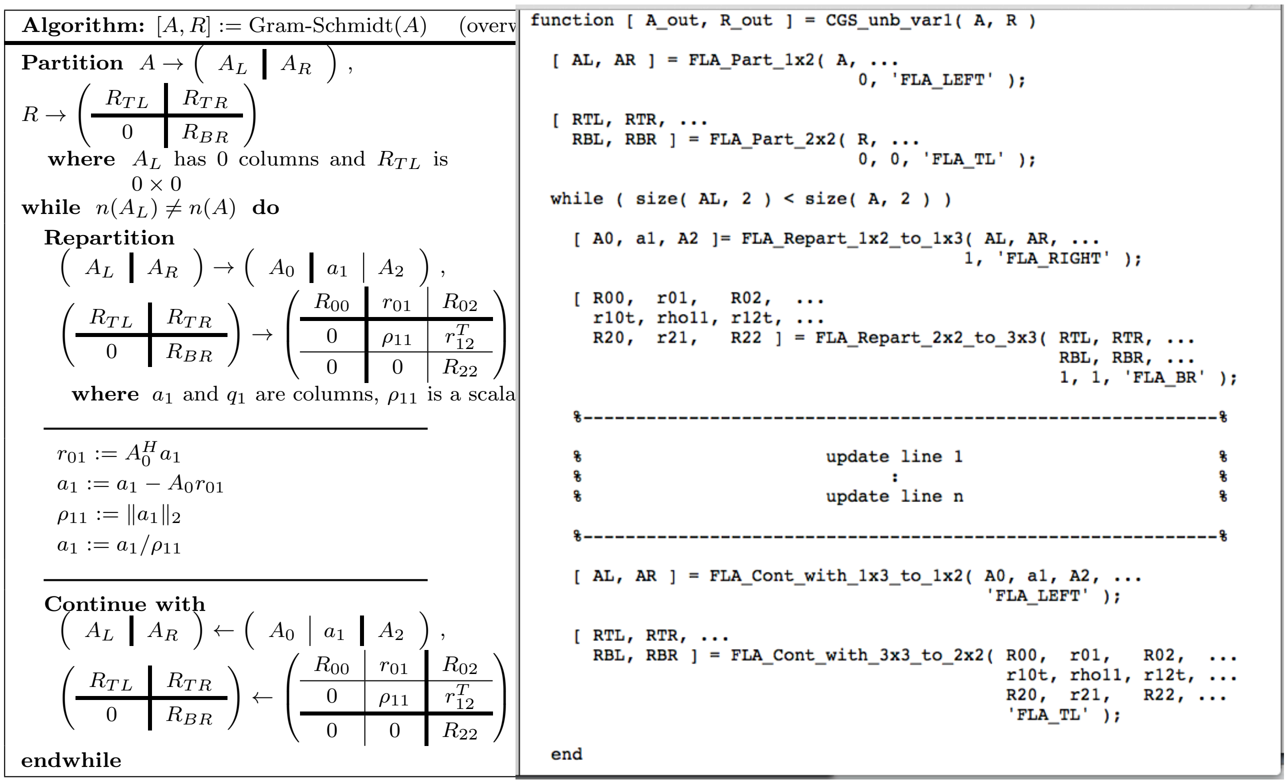
To the left of the menu, you now find what we call a code skeleton for the implementation, as shown in Figure 5. In Figure 6 we show the algorithm and generated code skeleton side-by-side.

3.3 Editing the code skeleton

At this point, one should copy the code skeleton into one's favorite text editor. (We highly recommend emacs for the serious programmer.) Once this is done, there are two things left to do:

Fix the code skeleton: The Spark webpage "guesses" the code skeleton. One detail that it sometimes gets wrong is the "stopping criteria". In this case, the algorithm should stay in the loop as long as $n(A_L) \neq n(A)$ (the width of A_L is not yet the width of A). In our example, the Spark webpage guessed that the column size of matrix A is to be used for the stopping criteria:

```
while ( size( AL, 2 ) < size( A, 2 ) )
```



```
Algorithm: $[A, R] := \text{Gram-Schmidt}(A)$    (overv
Partition $A \rightarrow \left( A_L \,\middle|\, A_R \right)$,
  $R \rightarrow \left( \begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array} \right)$
    where $A_L$ has 0 columns and $R_{TL}$ is $0 \times 0$
while $n(A_L) \neq n(A)$ do
  Repartition
    $\left( A_L \,\middle|\, A_R \right) \rightarrow \left( A_0 \,\middle|\, a_1 \,|\, A_2 \right)$,
    $\left( \begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array} \right) \rightarrow \left( \begin{array}{c|c|c} R_{00} & r_{01} & R_{02} \\ \hline 0 & \rho_{11} & r_{12}^T \\ \hline 0 & 0 & R_{22} \end{array} \right)$
      where $a_1$ and $q_1$ are columns, $\rho_{11}$ is a scala

    $r_{01} := A_0^H a_1$
    $a_1 := a_1 - A_0 r_{01}$
    $\rho_{11} := \|a_1\|_2$
    $a_1 := a_1 / \rho_{11}$

  Continue with
    $\left( A_L \,\middle|\, A_R \right) \leftarrow \left( A_0 \,|\, a_1 \,\middle|\, A_2 \right)$,
    $\left( \begin{array}{c|c} R_{TL} & R_{TR} \\ \hline 0 & R_{BR} \end{array} \right) \leftarrow \left( \begin{array}{c|c|c} R_{00} & r_{01} & R_{02} \\ \hline 0 & \rho_{11} & r_{12}^T \\ \hline 0 & 0 & R_{22} \end{array} \right)$
endwhile
```

```
function [ A_out, R_out ] = CGS_unb_var1( A, R )

  [ AL, AR ] = FLA_Part_1x2( A, ...
                              0, 'FLA_LEFT' );

  [ RTL, RTR, ...
    RBL, RBR ] = FLA_Part_2x2( R, ...
                              0, 0, 'FLA_TL' );

  while ( size( AL, 2 ) < size( A, 2 ) )

    [ A0, a1, A2 ]= FLA_Repart_1x2_to_1x3( AL, AR, ...
                                        1, 'FLA_RIGHT' );

    [ R00,  r01,   R02,  ...
      r10t, rho11, r12t, ...
      R20,  r21,   R22 ] = FLA_Repart_2x2_to_3x3( RTL, RTR, ...
                                                  RBL, RBR, ...
                                                  1, 1, 'FLA_BR' );

    %------------------------------------------------------------%

    %                      update line 1                         %
    %                            :                               %
    %                      update line n                         %

    %------------------------------------------------------------%

    [ AL, AR ] = FLA_Cont_with_1x3_to_1x2( A0, a1, A2, ...
                                          'FLA_LEFT' );

    [ RTL, RTR, ...
      RBL, RBR ] = FLA_Cont_with_3x3_to_2x2( R00,  r01,   R02,  ...
                                             r10t, rho11, r12t, ...
                                             R20,  r21,   R22,  ...
                                             'FLA_TL' );

  end
```

Figure 6: Left: Classical Gram-Schmidt algorithm. Right: Generated code-skeleton for CGS.

which happens to be correct. (When you implement the Householder QR factorization, you may not be so lucky...)

The "update" statements: The Spark webpage can't guess what the actual updates to the various parts of matrices A and R should be. It fills in

```
%                      update line 1                         %
%                            :                               %
%                      update line n                         %
```

Thus, one has to manually translate

$$\boxed{\begin{array}{l} r_{01} := A_0^H a_1 \\ a_1 := a_1 - A_0 r_{01} \\ \rho_{11} := \|a_1\|_2 \\ a_1 := a_1 / \rho_{11} \end{array}}$$

into appropriate M-script code:

```
r01 = A0' * a1;
a1 = a1 - A0 * r01;
rho11 = norm( a1 );
a1 = a1 / rho11;
```

(Notice: if one forgets the ";", when executed the results of the assignment will be printed by Matlab/Octave.)

At this point, one saves the resulting code in the file `CGS_unb_var1.m`. The ".m" ending is important since the name of the file is used to find the routine when using Matlab/Octave.

3.4 Testing

To now test the routine, one starts octave and, for example, executes the commands

```
> A = rand( 5, 4 )
> R = zeros( 4, 4 )
> [ Q, R ] = CGS_unb_var1( A, R )
> A - Q * triu( R )
```

The result should be (approximately) a 5×4 zero matrix.

(The first time you execute the above, you may get a bunch of warnings from Octave. Just ignore those.)

4 Implementing the Other Algorithms

Next, we leave it to the reader to implement

- Modified Gram Schmidt algorithm, (`MGS_unb_var1`, corresponding to the **right-most** algorithm in Figure 2), respectively.
- The Householder QR factorization algorithm and algorithm to form Q from "Notes on Householder QR Factorization".

  The routine for computing a Householder transformation (similar to Figure 1) can be found at

  `http://www.cs.utexas.edu/users/flame/Notes/FLAMEatlab/Housev.m`

  That routine implements the algorithm on the left in Figure 1). Try and see what happens if you replace it with the algorithm to its right.

Note: For the Householder QR factorization and "form Q" algorithm how to start the algorithm when the matrix is not square is a bit tricky. Thus, you may assume that the matrix *is* square.