Perform QR Factorization Using CORDIC

This example shows how to write MATLAB® code that works for both floating-point and fixed-point data types. The algorithm used in this example is the QR factorization implemented via CORDIC (Coordinate Rotation Digital Computer).

Copy Command

A good way to write an algorithm intended for a fixed-point target is to write it in MATLAB using builtin floating-point types so you can verify that the algorithm works. When you refine the algorithm to work with fixed-point types, then the best thing to do is to write it so that the same code continues working with floating-point. That way, when you are debugging, then you can switch the inputs back and forth between floating-point and fixed-point types to determine if a difference in behavior is because of fixed-point effects such as overflow and quantization versus an algorithmic difference. Even if the algorithm is not well suited for a floating-point target (as is the case of using CORDIC in the following example), it is still advantageous to have your MATLAB code work with floating-point for debugging purposes.

In contrast, you may have a completely different strategy if your target is floating point. For example, the QR algorithm is often done in floating-point with Householder transformations and row or column pivoting. But in fixed-point it is often more efficient to use CORDIC to apply Givens rotations with no pivoting.

This example addresses the first case, where your target is fixed-point, and you want an algorithm that is independent of data type because it is easier to develop and debug.

In this example you will learn various coding methods that can be applied across systems. The significant design patterns used in this example are the following:

- Data Type Independence: the algorithm is written in such a way that the MATLAB code is independent of data type, and will work equally well for fixed-point, double-precision floating-point, and single-precision floating-point.
- Overflow Prevention: method to guarantee not to overflow. This demonstrates how to prevent overflows in fixed-point.
- Solving Systems of Equations: method to use computational efficiency. Narrow your code scope by isolating what you need to define.

The main part in this example is an implementation of the QR factorization in fixed-point arithmetic using CORDIC for the Givens rotations. The algorithm is written in such a way that the MATLAB code is independent of data type, and will work equally well for fixed-point, double-precision floating-point, and single-precision floating-point.

The QR factorization of M-by-N matrix A produces an M-by-N upper triangular matrix R and an M-by-M orthogonal matrix Q such that `A = Q*R`. A matrix is upper triangular if it has all zeros below the diagonal. An M-by-M matrix Q is orthogonal if `Q'*Q = eye(M)`, the identity matrix.

The QR factorization is widely used in least-squares problems, such as the recursive least squares (RLS) algorithm used in adaptive filters.

The CORDIC algorithm is attractive for computing the QR algorithm in fixed-point because you can apply orthogonal Givens rotations with CORDIC using only shift and add operations.

Setup

So this example does not change your preferences or settings, we store the original state here, and restore them at the end.

```
originalFormat = get(0, 'format'); format short
originalFipref = get(fipref);      reset(fipref);
originalGlobalFimath = fimath;     resetglobalfimath;
```

Defining the CORDIC QR Algorithm

The CORDIC QR algorithm is given in the following MATLAB function, where A is an M-by-N real matrix, and `niter` is the number of CORDIC iterations. Output Q is an M-by-M orthogonal matrix, and R is an M-by-N upper-triangular matrix such that `Q*R = A`.

```
function [Q,R] = cordicqr(A,niter)
  Kn = inverse_cordic_growth_constant(niter);
  [m,n] = size(A);
  R = A;
  Q = coder.nullcopy(repmat(A(:,1),1,m)); % Declare type and size of Q
  Q(:) = eye(m);                          % Initialize Q
  for j=1:n
    for i=j+1:m
      [R(j,j:end),R(i,j:end),Q(:,j),Q(:,i)] = ...
          cordicgivens(R(j,j:end),R(i,j:end),Q(:,j),Q(:,i),niter,Kn);
    end
  end
end
```

This function was written to be independent of data type. It works equally well with builtin floating-point types (double and single) and with the fixed-point `fi` object.

One of the trickiest aspects of writing data-type independent code is to specify data type and size for a new variable. In order to preserve data types without having to explicitly specify them, the output R was set to be the same as input A, like this:

```
R = A;
```

In addition to being data-type independent, this function was written in such a way that MATLAB Coder™ will be able to generate efficient C code from it. In MATLAB, you most often declare and initialize a variable in one step, like this:

```
Q = eye(m)
```

However, `Q=eye(m)` would always produce Q as a double-precision floating point variable. If A is fixed-point, then we want Q to be fixed-point; if A is single, then we want Q to be single; etc.

Hence, you need to declare the type and size of Q in one step, and then initialize it in a second step. This gives MATLAB Coder the information it needs to create an efficient C program with the correct types and sizes. In the finished code you initialize output Q to be an M-by-M identity matrix and the same data type as A, like this:

```
Q = coder.nullcopy(repmat(A(:,1),1,m)); % Declare type and size of Q
Q(:) = eye(m);                          % Initialize Q
```

The `coder.nullcopy` function declares the size and type of Q without initializing it. The expansion of the first column of A with `repmat` won't appear in code generated by MATLAB; it is only used to specify the size. The `repmat` function was used instead of `A(:,1:m)` because A may have more rows than columns, which will be the case in a least-squares problem. You have to be sure to always assign values to every element of an array when you declare it with `coder.nullcopy`, because if you don't then you will have uninitialized memory.

You will notice this pattern of assignment again and again. This is another key enabler of data-type independent code.

The heart of this function is applying orthogonal Givens rotations in-place to the rows of R to zero out sub-diagonal elements, thus forming an upper-triangular matrix. The same rotations are applied in-place to the columns of the identity matrix, thus forming orthogonal Q. The Givens rotations are applied using the `cordicgivens` function, as defined in the next section. The rows of R and columns of Q are used as both input and output to the `cordicgivens` function so that the computation is done in-place, overwriting R and Q.

```
[R(j,j:end),R(i,j:end),Q(:,j),Q(:,i)] = ...
    cordicgivens(R(j,j:end),R(i,j:end),Q(:,j),Q(:,i),niter,Kn);
```

Defining the CORDIC Givens Rotation

The `cordicgivens` function applies a Givens rotation by performing CORDIC iterations to rows `x=R(j,j:end)`, `y=R(i,j:end)` around the angle defined by `x(1)=R(j,j)` and `y(1)=R(i,j)` where `i>j`, thus zeroing out `R(i,j)`. The same rotation is applied to columns `u = Q(:,j)` and `v = Q(:,i)`, thus forming the orthogonal matrix Q.

```
function [x,y,u,v] = cordicgivens(x,y,u,v,niter,Kn)
  if x(1)<0
    % Compensation for 3rd and 4th quadrants
    x(:) = -x;  u(:) = -u;
    y(:) = -y;  v(:) = -v;
  end
  for i=0:niter-1
    x0 = x;
    u0 = u;
    if y(1)<0
      % Counter-clockwise rotation
      % x and y form R,         u and v form Q
      x(:) = x - bitsra(y, i);  u(:) = u - bitsra(v, i);
      y(:) = y + bitsra(x0,i);  v(:) = v + bitsra(u0,i);
    else
      % Clockwise rotation
      % x and y form R,         u and v form Q
      x(:) = x + bitsra(y, i);  u(:) = u + bitsra(v, i);
      y(:) = y - bitsra(x0,i);  v(:) = v - bitsra(u0,i);
    end
  end
  % Set y(1) to exactly zero so R will be upper triangular without round off
  % showing up in the lower triangle.
  y(1) = 0;
  % Normalize the CORDIC gain
  x(:) = Kn * x;  u(:) = Kn * u;
  y(:) = Kn * y;  v(:) = Kn * v;
end
```

The advantage of using CORDIC in fixed-point over the standard Givens rotation is that CORDIC does not use square root or divide operations. Only bit-shifts, addition, and subtraction are needed in the main loop, and one scalar-vector multiply at the end to normalize the CORDIC gain. Also, CORDIC rotations work well in pipelined architectures.

The bit shifts in each iteration are performed with the bit shift right arithmetic (`bitsra`) function instead of `bitshift`, multiplication by 0.5, or division by 2, because `bitsra`

- generates more efficient embedded code,
- works equally well with positive and negative numbers,
- works equally well with floating-point, fixed-point and integer types, and
- keeps this code independent of data type.

It is worthwhile to note that there is a difference between sub-scripted assignment (`subsasgn`) into a variable `a(:) = b` versus overwriting a variable `a = b`. Sub-scripted assignment into a variable like this

```
x(:) = x + bitsra(y, i);
```

always preserves the type of the left-hand-side argument `x`. This is the recommended programming style in fixed-point. For example fixed-point types often grow their word length in a sum, which is governed by the `SumMode` property of the `fimath` object, so that the right-hand-side `x + bitsra(y,i)` can have a different data type than `x`.

If, instead, you overwrite the left-hand-side like this

```
x = x + bitsra(y, i);
```

then the left-hand-side x takes on the type of the right-hand-side sum. This programming style leads to changing the data type of x in fixed-point code, and is discouraged.

Defining the Inverse CORDIC Growth Constant

This function returns the inverse of the CORDIC growth factor after `niter` iterations. It is needed because CORDIC rotations grow the values by a factor of approximately 1.6468, depending on the number of iterations, so the gain is normalized in the last step of `cordicgivens` by a multiplication by the inverse Kn = 1/1.6468 = 0.60725.

```
function Kn = inverse_cordic_growth_constant(niter)
  Kn = 1/prod(sqrt(1+2.^(-2*(0:double(niter)-1))));
end
```

Exploring CORDIC Growth as a Function of Number of Iterations

The function for CORDIC growth is defined as

```
growth = prod(sqrt(1+2.^(-2*(0:double(niter)-1))))
```

and the inverse is

```
inverse_growth = 1 ./ growth
```

Growth is a function of the number of iterations `niter`, and quickly converges to approximately 1.6468, and the inverse converges to approximately 0.60725. You can see in the following table that the difference from one iteration to the next ceases to change after 27 iterations. This is because the calculation hit the limit of precision in double floating-point at 27 iterations.

niter	growth	diff(growth)	1./growth	diff(1./growth)
0	1.000000000000000	0	1.000000000000000	0
1	1.414213562373095	0.414213562373095	0.707106781186547	-0.292893218813453
2	1.581138830084190	0.166925267711095	0.632455532033676	-0.074651249152872
3	1.629800601300662	0.048661771216473	0.613571991077896	-0.018883540955780
4	1.642484065752237	0.012683464451575	0.608833912517752	-0.004738078560144
5	1.645688915757255	0.003204850005018	0.607648256256168	-0.001185656261584
6	1.646492278712479	0.000803362955224	0.607351770141296	-0.000296486114872
7	1.646693254273644	0.000200975561165	0.607277644093526	-0.000074126047770
8	1.646743506596901	0.000050252323257	0.607259112298893	-0.000018531794633
9	1.646756070204878	0.000012563607978	0.607254479332562	-0.000004632966330
10	1.646759211139822	0.000003140934944	0.607253321089875	-0.000001158242687
11	1.646759996375617	0.000000785235795	0.607253031529134	-0.000000289560741
12	1.646760192684695	0.000000196309077	0.607252959138945	-0.000000072390190
13	1.646760241761972	0.000000049077277	0.607252941041397	-0.000000018097548
14	1.646760254031292	0.000000012269320	0.607252936517010	-0.000000004524387
15	1.646760257098622	0.000000003067330	0.607252935385914	-0.000000001131097
16	1.646760257865455	0.000000000766833	0.607252935103139	-0.000000000282774
17	1.646760258057163	0.000000000191708	0.607252935032446	-0.000000000070694
18	1.646760258105090	0.000000000047927	0.607252935014772	-0.000000000017673
19	1.646760258117072	0.000000000011982	0.607252935010354	-0.000000000004418
20	1.646760258120067	0.000000000002995	0.607252935009249	-0.000000000001105
21	1.646760258120816	0.000000000000749	0.607252935008973	-0.000000000000276
22	1.646760258121003	0.000000000000187	0.607252935008904	-0.000000000000069
23	1.646760258121050	0.000000000000047	0.607252935008887	-0.000000000000017
24	1.646760258121062	0.000000000000012	0.607252935008883	-0.000000000000004
25	1.646760258121065	0.000000000000003	0.607252935008882	-0.000000000000001
26	1.646760258121065	0.000000000000001	0.607252935008881	-0.000000000000000
27	1.646760258121065	0	0.607252935008881	0
28	1.646760258121065	0	0.607252935008881	0
29	1.646760258121065	0	0.607252935008881	0
30	1.646760258121065	0	0.607252935008881	0
31	1.646760258121065	0	0.607252935008881	0
32	1.646760258121065	0	0.607252935008881	0

Comparing CORDIC to the Standard Givens Rotation

The `cordicgivens` function is numerically equivalent to the following standard Givens rotation algorithm from Golub & Van Loan, *Matrix Computations.* In the `cordicqr` function, if you replace the call to `cordicgivens` with a call to `givensrotation`, then you will have the standard Givens QR algorithm.

```
function [x,y,u,v] = givensrotation(x,y,u,v)
  a = x(1); b = y(1);
  if b==0
    % No rotation necessary.  c = 1; s = 0;
    return;
  else
    if abs(b) > abs(a)
      t = -a/b; s = 1/sqrt(1+t^2); c = s*t;
    else
      t = -b/a; c = 1/sqrt(1+t^2); s = c*t;
    end
  end
  x0 = x;             u0 = u;
  % x and y form R,   u and v form Q
  x(:) = c*x0 - s*y;  u(:) = c*u0 - s*v;
  y(:) = s*x0 + c*y;  v(:) = s*u0 + c*v;
end
```

The `givensrotation` function uses division and square root, which are expensive in fixed-point, but good for floating-point algorithms.

Example of CORDIC Rotations

Here is a 3-by-3 example that follows the CORDIC rotations through each step of the algorithm. The algorithm uses orthogonal rotations to zero out the subdiagonal elements of R using the diagonal elements as pivots. The same rotations are applied to the identity matrix, thus producing orthogonal Q such that `Q*R = A`.

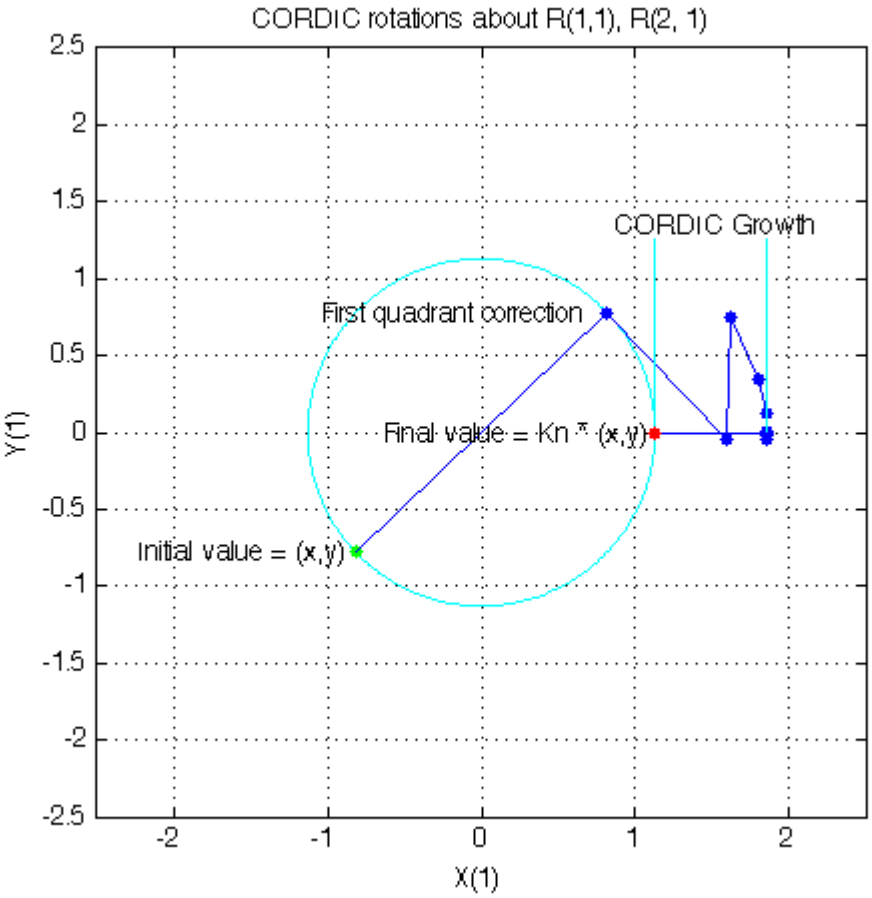
Let A be a random 3-by-3 matrix, and initialize `R = A`, and `Q = eye(3)`.

```
R = A = [-0.8201    0.3573   -0.0100
         -0.7766   -0.0096   -0.7048
         -0.7274   -0.6206   -0.8901]

    Q = [ 1         0         0
          0         1         0
          0         0         1]
```

The first rotation is about the first and second row of R and the first and second column of Q. Element `R(1,1)` is the pivot and `R(2,1)` rotates to 0.

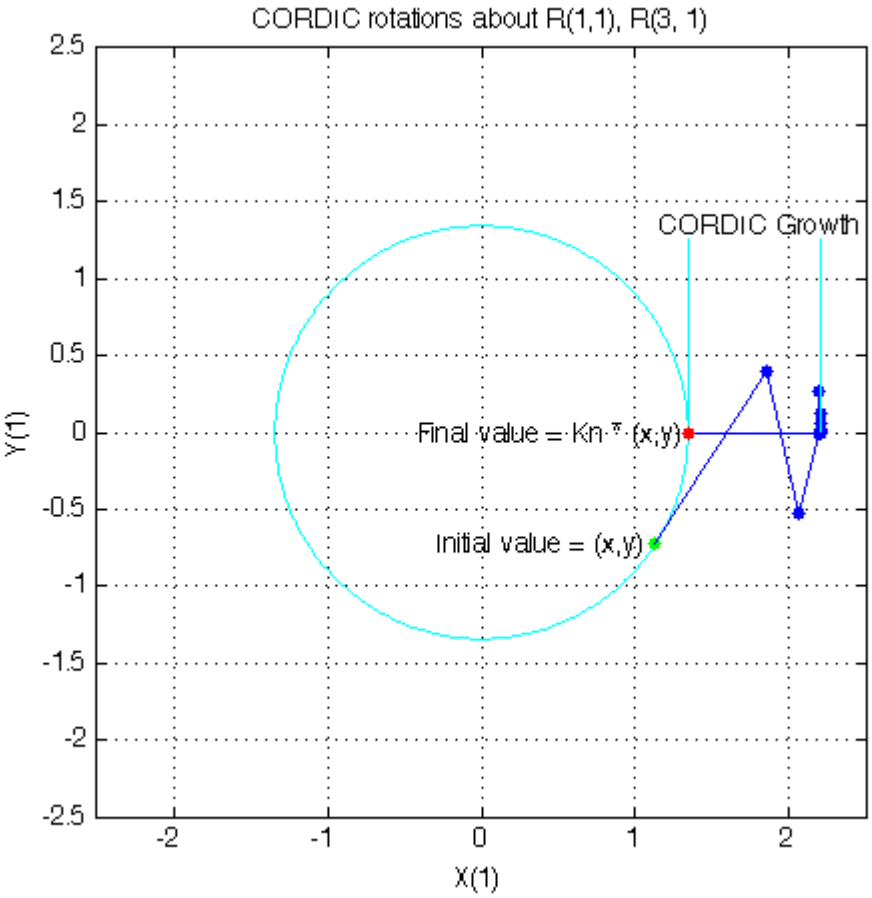
```
    R before the first rotation           R after the first rotation
x [-0.8201    0.3573   -0.0100]   ->    x [1.1294   -0.2528    0.4918]
y [-0.7766   -0.0096   -0.7048]   ->    y [     0    0.2527    0.5049]
   -0.7274   -0.6206   -0.8901            -0.7274   -0.6206   -0.8901

    Q before the first rotation           Q after the first rotation
    u         v                           u         v
   [1]       [0]        0               [-0.7261] [ 0.6876]        0
   [0]       [1]        0         ->    [-0.6876] [-0.7261]        0
   [0]       [0]        1               [      0] [      0]        1
```

In the following plot, you can see the growth in x in each of the CORDIC iterations. The growth is factored out at the last step by multiplying it by `Kn = 0.60725`. You can see that `y(1)` iterates to 0. Initially, the point `[x(1), y(1)]` is in the third quadrant, and is reflected into the first quadrant before the start of the CORDIC iterations.



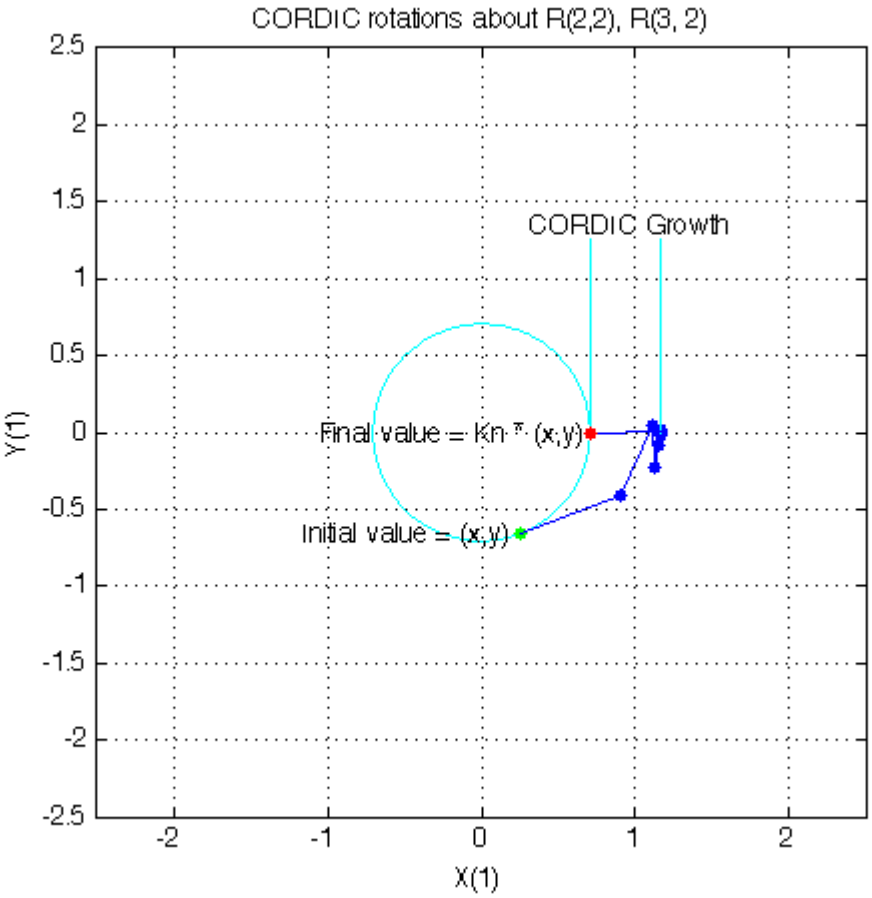
The second rotation is about the first and third row of R and the first and third column of Q. Element `R(1,1)` is the pivot and `R(3,1)` rotates to 0.

```
     R before the second rotation           R after the second rotation
x  [1.1294   -0.2528    0.4918]   ->    x [1.3434    0.1235    0.8954]
         0    0.2527    0.5049                   0    0.2527    0.5049
y [-0.7274]  -0.6206   -0.8901    ->    y [     0   -0.6586   -0.4820]

     Q before the second rotation           Q after the second rotation
     u                      v               u                    v
  [-0.7261]   0.6876       [0]           [-0.6105]   0.6876  [-0.3932]
  [-0.6876]  -0.7261       [0]    ->     [-0.5781]  -0.7261  [-0.3723]
  [      0]        0       [1]           [-0.5415]        0  [ 0.8407]
```



The third rotation is about the second and third row of R and the second and third column of Q. Element `R(2,2)` is the pivot and `R(3,2)` rotates to 0.

```
   R before the third rotation            R after the third rotation
   1.3434    0.1235    0.8954             1.3434    0.1235    0.8954
x       0  [ 0.2527    0.5049]   ->   x       0   [0.7054    0.6308]
y       0  [-0.6586   -0.4820]   ->   y       0   [     0    0.2987]

   Q before the third rotation            Q after the third rotation
             u         v                          u         v
  -0.6105  [ 0.6876] [-0.3932]          -0.6105  [ 0.6134] [ 0.5011]
  -0.5781  [-0.7261] [-0.3723]   ->     -0.5781  [ 0.0875] [-0.8113]
  -0.5415  [      0] [ 0.8407]          -0.5415  [-0.7849] [ 0.3011]
```



This completes the QR factorization. R is upper triangular, and Q is orthogonal.

```
R =
   1.3434    0.1235    0.8954
        0    0.7054    0.6308
        0         0    0.2987

Q =
  -0.6105    0.6134    0.5011
  -0.5781    0.0875   -0.8113
  -0.5415   -0.7849    0.3011
```

You can verify that Q is within roundoff error of being orthogonal by multiplying and seeing that it is close to the identity matrix.

```
Q*Q' =  1.0000    0.0000    0.0000
        0.0000    1.0000         0
        0.0000         0    1.0000

Q'*Q =  1.0000    0.0000   -0.0000
        0.0000    1.0000   -0.0000
       -0.0000   -0.0000    1.0000
```

You can see the error difference by subtracting the identity matrix.

```
Q*Q' - eye(size(Q)) =           0   2.7756e-16   3.0531e-16
                      2.7756e-16   4.4409e-16            0
                      3.0531e-16            0   6.6613e-16
```

You can verify that Q*R is close to A by subtracting to see the error difference.

```
Q*R - A =  -3.7802e-11  -7.2325e-13  -2.7756e-17
           -3.0512e-10   1.1708e-12  -4.4409e-16
            3.6836e-10  -4.3487e-13  -7.7716e-16
```

Determining the Optimal Output Type of Q for Fixed Word Length

Since Q is orthogonal, you know that all of its values are between -1 and +1. In floating-point, there is no decision about the type of Q: it should be the same floating-point type as A. However, in fixed-point, you can do better than making Q have the identical fixed-point type as A. For example, if A has word length 16 and fraction length 8, and if we make Q also have word length 16 and fraction length 8, then you force Q to be less accurate than it could be and waste the upper half of the fixed-point range.

The best type for Q is to make it have full range of its possible outputs, plus accommodate the 1.6468 CORDIC growth factor in intermediate calculations. Therefore, assuming that the word length of Q is the same as the word length of input A, then the best fraction length for Q is 2 bits less than the word length (one bit for 1.6468 and one bit for the sign).

Hence, our initialization of Q in `cordicqr` can be improved like this.

```
if isfi(A) && (isfixed(A) || isscaleddouble(A))
      Q = fi(one*eye(m), get(A,'NumericType'), ...
             'FractionLength',get(A,'WordLength')-2);
else
  Q = coder.nullcopy(repmat(A(:,1),1,m));
  Q(:) = eye(m);
end
```

A slight disadvantage is that this section of code is dependent on data type. However, you gain a major advantage by picking the optimal type for Q, and the main algorithm is still independent of data type. You can do this kind of input parsing in the beginning of a function and leave the main algorithm data-type independent.

Preventing Overflow in Fixed Point R

This section describes how to determine a fixed-point output type for R in order to prevent overflow. In order to pick an output type, you need to know how much the magnitude of the values of R will grow.

Given real matrix A and its QR factorization computed by Givens rotations without pivoting, an upper-bound on the magnitude of the elements of R is the square-root of the number of rows of A times the magnitude of the largest element in A. Furthermore, this growth will never be greater during an intermediate computation. In other words, let `[m,n]=size(A)`, and `[Q,R]=givensqr(A)`. Then

```
max(abs(R(:))) <= sqrt(m) * max(abs(A(:))).
```

This is true because the each element of R is formed from orthogonal rotations from its corresponding column in A, so the largest that any element `R(i,j)` can get is if all of the elements of its corresponding column `A(:,j)` were rotated to a single value. In other words, the largest possible value will be bounded by the 2-norm of `A(:,j)`. Since the 2-norm of `A(:,j)` is equal to the square-root of the sum of the squares of the m elements, and each element is less-than-or-equal-to the largest element of A, then

```
norm(A(:,j)) <= sqrt(m) * max(abs(A(:))).
```

That is, for all j

```
norm(A(:,j))  = sqrt(A(1,j)^2 + A(2,j)^2 + ... + A(m,j)^2)
             <= sqrt( m * max(abs(A(:)))^2)
              = sqrt(m) * max(abs(A(:))).
```

and so for all i,j

```
abs(R(i,j)) <= norm(A(:,j)) <= sqrt(m) * max(abs(A(:))).
```

Hence, it is also true for the largest element of R

```
max(abs(R(:))) <= sqrt(m) * max(abs(A(:))).
```

This becomes useful in fixed-point where the elements of A are often very close to the maximum value attainable by the data type, so we can set a tight upper bound without knowing the values of A. This is important because we want to set an output type for R with a minimum number of bits, only knowing the upper bound of the data type of A. You can use `fi` method `upperbound` to get this value.

Therefore, for all i,j

```
abs(R(i,j)) <= sqrt(m) * upperbound(A)
```

Note that `sqrt(m)*upperbound(A)` is also an upper bound for the elements of A:

```
abs(A(i,j)) <= upperbound(A) <= sqrt(m)*upperbound(A)
```

Therefore, when picking fixed-point data types, `sqrt(m)*upperbound(A)` is an upper bound that will work for both A and R.

Attaining the maximum is easy and common. The maximum will occur when all elements get rotated into a single element, like the following matrix with orthogonal columns:

```
A = [7    -7     7     7
     7     7    -7     7
     7    -7    -7    -7
     7     7     7    -7];
```

Its maximum value is 7 and its number of rows is `m=4`, so we expect that the maximum value in R will be bounded by `max(abs(A(:)))*sqrt(m) = 7*sqrt(4) = 14`. Since A in this example is orthogonal, each column gets rotated to the max value on the diagonal.

```
niter = 52;
[Q,R] = cordicqr(A,niter)
```

```
Q =

    0.5000   -0.5000    0.5000    0.5000
    0.5000    0.5000   -0.5000    0.5000
    0.5000   -0.5000   -0.5000   -0.5000
    0.5000    0.5000    0.5000   -0.5000


R =

   14.0000    0.0000   -0.0000   -0.0000
         0   14.0000   -0.0000    0.0000
         0         0   14.0000    0.0000
         0         0         0   14.0000
```

Another simple example of attaining maximum growth is a matrix that has all identical elements, like a matrix of all ones. A matrix of ones will get rotated into `1*sqrt(m)` in the first row and zeros elsewhere. For example, this 9-by-5 matrix will have all `1*sqrt(9)=3` in the first row of R.

```
m = 9; n = 5;
A = ones(m,n)
niter = 52;
[Q,R] = cordicqr(A,niter)
```

```
A =

     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1
     1     1     1     1     1


Q =

  Columns 1 through 7

    0.3333    0.5567   -0.6784    0.3035   -0.1237    0.0503    0.0158
    0.3333    0.0296    0.2498   -0.1702   -0.6336    0.1229   -0.3012
    0.3333    0.2401    0.0562   -0.3918    0.4927    0.2048   -0.5395
    0.3333    0.0003    0.0952   -0.1857    0.2148    0.4923    0.7080
    0.3333    0.1138    0.0664   -0.2263    0.1293   -0.8348    0.2510
    0.3333   -0.3973   -0.0143    0.3271    0.4132   -0.0354   -0.2165
    0.3333    0.1808    0.3538   -0.1012   -0.2195         0    0.0824
    0.3333   -0.6500   -0.4688   -0.2380   -0.2400         0         0
    0.3333   -0.0740    0.3400    0.6825   -0.0331         0         0

  Columns 8 through 9

    0.0056   -0.0921
   -0.5069   -0.1799
    0.0359    0.3122
   -0.2351   -0.0175
   -0.2001    0.0610
   -0.0939   -0.6294
    0.7646   -0.2849
    0.2300    0.2820
         0    0.5485


R =

    3.0000    3.0000    3.0000    3.0000    3.0000
         0    0.0000    0.0000    0.0000    0.0000
         0         0    0.0000    0.0000    0.0000
         0         0         0    0.0000    0.0000
         0         0         0         0    0.0000
         0         0         0         0         0
         0         0         0         0         0
         0         0         0         0         0
         0         0         0         0         0
```

As in the `cordicqr` function, the Givens QR algorithm is often written by overwriting A in-place with R, so being able to cast A into R's data type at the beginning of the algorithm is convenient.

In addition, if you compute the Givens rotations with CORDIC, there is a growth-factor that converges quickly to approximately 1.6468. This growth factor gets normalized out after each Givens rotation, but you need to accommodate it in the intermediate calculations. Therefore, the number of additional bits that are required including the Givens and CORDIC growth are `log2(1.6468* sqrt(m))`. The additional bits of head-room can be added either by increasing the word length, or decreasing the fraction length.

A benefit of increasing the word length is that it allows for the maximum possible precision for a given word length. A disadvantage is that the optimal word length may not correspond to a native type on your processor (e.g. increasing from 16 to 18 bits), or you may have to increase to the next larger native word size which could be quite large (e.g. increasing from 16 to 32 bits, when you only needed 18).

A benefit of decreasing fraction length is that you can do the computation in-place in the native word size of A. A disadvantage is that you lose precision.

Another option is to pre-scale the input by right-shifting. This is equivalent to decreasing the fraction length, with the additional disadvantage of changing the scaling of your problem. However, this may be an attractive option to you if you prefer to only work in fractional arithmetic or integer arithmetic.

Example of Fixed Point Growth in R

If you have a fixed-point input matrix A, you can define fixed-point output R with the growth defined in the previous section.

Start with a random matrix X.

```
X = [0.0513   -0.2097    0.9492    0.2614
     0.8261    0.6252    0.3071   -0.9415
     1.5270    0.1832    0.1352   -0.1623
     0.4669   -1.0298    0.5152   -0.1461];
```

Create a fixed-point A from X.

```
A = sfi(X)
```

```
A =

    0.0513   -0.2097    0.9492    0.2614
    0.8261    0.6252    0.3071   -0.9415
    1.5270    0.1832    0.1352   -0.1623
    0.4669   -1.0298    0.5152   -0.1461

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 14
```

```
m = size(A,1)
```

```
m =

     4
```

The growth factor is 1.6468 times the square-root of the number of rows of A. The bit growth is the next integer above the base-2 logarithm of the growth.

```
bit_growth = ceil(log2(cordic_growth_constant * sqrt(m)))
```

```
bit_growth =

     2
```

Initialize R with the same values as A, and a word length increased by the bit growth.

```
R = sfi(A, get(A,'WordLength')+bit_growth, get(A,'FractionLength'))
```

```
R =

    0.0513   -0.2097    0.9492    0.2614
    0.8261    0.6252    0.3071   -0.9415
    1.5270    0.1832    0.1352   -0.1623
    0.4669   -1.0298    0.5152   -0.1461

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 14
```

Use R as input and overwrite it.

```
niter = get(R,'WordLength') - 1
[Q,R] = cordicqr(R, niter)
```

```
niter =

    17


Q =

    0.0284   -0.1753    0.9110    0.3723
    0.4594    0.4470    0.3507   -0.6828
    0.8490    0.0320   -0.2169    0.4808
    0.2596   -0.8766   -0.0112   -0.4050

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 16

R =

    1.7989    0.1694    0.4166   -0.6008
         0    1.2251   -0.4764   -0.3438
         0         0    0.9375   -0.0555
         0         0         0    0.7214

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 14
```

Verify that `Q*Q'` is near the identity matrix.

```
double(Q)*double(Q')
```

```
ans =

    1.0000   -0.0001    0.0000    0.0000
   -0.0001    1.0001    0.0000   -0.0000
    0.0000    0.0000    1.0000   -0.0000
    0.0000   -0.0000   -0.0000    1.0000
```

Verify that Q*R - A is small relative to the precision of A.

```
err = double(Q)*double(R) - double(A)
```

```
err =

   1.0e-03 *

   -0.1048   -0.2355    0.1829   -0.2146
    0.3472    0.2949    0.0260   -0.2570
    0.2776   -0.1740   -0.1007    0.0966
    0.0138   -0.1558    0.0417   -0.0362
```

Increasing Precision in R

The previous section showed you how to prevent overflow in R while maintaining the precision of A. If you leave the fraction length of R the same as A, then R cannot have more precision than A, and your precision requirements may be such that the precision of R must be greater.

An extreme example of this is to define a matrix with an integer fixed-point type (i.e. fraction length is zero). Let matrix X have elements that are the full range for signed 8 bit integers, between -128 and +127.

```
 X = [-128  -128  -128   127
      -128   127   127  -128
       127   127   127   127
       127   127  -128  -128];
```

Define fixed-point A to be equivalent to an 8-bit integer.

```
A = sfi(X,8,0)
```

```
A =

  -128  -128  -128   127
  -128   127   127  -128
   127   127   127   127
   127   127  -128  -128

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 8
        FractionLength: 0
```

```
m = size(A,1)
```

```
m =

     4
```

The necessary growth is 1.6468 times the square-root of the number of rows of A.

```
bit_growth = ceil(log2(cordic_growth_constant*sqrt(m)))
```

```
bit_growth =

     2
```

Initialize R with the same values as A, and allow for bit growth like you did in the previous section.

```
R = sfi(A, get(A,'WordLength')+bit_growth, get(A,'FractionLength'))
```

```
R =

  -128  -128  -128   127
  -128   127   127  -128
   127   127   127   127
   127   127  -128  -128

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 10
        FractionLength: 0
```

Compute the QR factorization, overwriting R.

```
niter = get(R,'WordLength') - 1;
[Q,R] = cordicqr(R, niter)
```

```
Q =

   -0.5039   -0.2930   -0.4062   -0.6914
   -0.5039    0.8750    0.0039    0.0078
    0.5000    0.2930    0.3984   -0.7148
    0.4922    0.2930   -0.8203    0.0039

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 10
        FractionLength: 8


R =

   257   126    -1    -1
     0   225   151  -148
     0     0   211   104
     0     0     0  -180

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
```

```
              WordLength: 10
          FractionLength: 0
```

Notice that R is returned with integer values because you left the fraction length of R at 0, the same as the fraction length of A.

The scaling of the least-significant bit (LSB) of A is 1, and you can see that the error is proportional to the LSB.

```
err = double(Q)*double(R)-double(A)
```

```
err =

   -1.5039   -1.4102   -1.4531   -0.9336
   -1.5039    6.3828    6.4531   -1.9961
    1.5000    1.9180    0.8086   -0.7500
   -0.5078    0.9336   -1.3398   -1.8672
```

You can increase the precision in the QR factorization by increasing the fraction length. In this example, you needed 10 bits for the integer part (8 bits to start with, plus 2 bits growth), so when you increase the fraction length you still need to keep the 10 bits in the integer part. For example, you can increase the word length to 32 and set the fraction length to 22, which leaves 10 bits in the integer part.

```
R = sfi(A, 32, 22)
```

```
R =

  -128  -128  -128   127
  -128   127   127  -128
   127   127   127   127
   127   127  -128  -128

           DataTypeMode: Fixed-point: binary point scaling
             Signedness: Signed
             WordLength: 32
         FractionLength: 22
```

```
niter = get(R,'WordLength') - 1;
[Q,R] = cordicqr(R, niter)
```

```
Q =

   -0.5020   -0.2913   -0.4088   -0.7043
   -0.5020    0.8649    0.0000    0.0000
    0.4980    0.2890    0.4056   -0.7099
    0.4980    0.2890   -0.8176    0.0000

           DataTypeMode: Fixed-point: binary point scaling
             Signedness: Signed
             WordLength: 32
         FractionLength: 30

R =

  255.0020  127.0029    0.0039    0.0039
         0  220.5476  146.8413 -147.9930
         0         0  208.4793  104.2429
```

```
           0         0         0 -179.6037

            DataTypeMode: Fixed-point: binary point scaling
              Signedness: Signed
              WordLength: 32
          FractionLength: 22
```

Now you can see fractional parts in R, and `Q*R-A` is small.

```
err = double(Q)*double(R)-double(A)
```

```
err =

   1.0e-05 *

   -0.1234   -0.0014   -0.0845    0.0267
   -0.1234    0.2574    0.1260   -0.1094
    0.0720    0.0289   -0.0400   -0.0684
    0.0957    0.0818   -0.1034    0.0095
```

The number of bits you choose for fraction length will depend on the precision requirements for your particular algorithm.

Picking Default Number of Iterations

The number of iterations is dependent on the desired precision, but limited by the word length of A. With each iteration, the values are right-shifted one bit. After the last bit gets shifted off and the value becomes 0, then there is no additional value in continuing to rotate. Hence, the most precision will be attained by choosing `niter` to be one less than the word length.

For floating-point, the number of iterations is bounded by the size of the mantissa. In double, 52 iterations is the most you can do to continue adding to something with the same exponent. In single, it is 23. See the reference page for eps for more information about floating-point accuracy.

Thus, we can make our code more usable by not requiring the number of iterations to be input, and assuming that we want the most precision possible by changing `cordicqr` to use this default for `niter`.

```
function [Q,R] = cordicqr(A,varargin)
  if nargin>=2 && ~isempty(varargin{1})
     niter = varargin{1};
  elseif isa(A,'double') || isfi(A) && isdouble(A)
    niter = 52;
  elseif isa(A,'single') || isfi(A) && issingle(A)
    niter = single(23);
  elseif isfi(A)
    niter = int32(get(A,'WordLength') - 1);
  else
    assert(0,'First input must be double, single, or fi.');
  end
```

A disadvantage of doing this is that this makes a section of our code dependent on data type. However, an advantage is that the function is much more convenient to use because you don't have to specify `niter` if you don't want to, and the main algorithm is still data-type independent. Similar to picking an optimal output type for Q, you can do this kind of input parsing in the beginning of a function and leave the main algorithm data-type independent.

Here is an example from a previous section, without needing to specify an optimal `niter`.

```
A = [7    -7     7     7
     7     7    -7     7
```

```
     7    -7    -7    -7
     7     7     7    -7];
[Q,R] = cordicqr(A)
```

```
Q =

    0.5000   -0.5000    0.5000    0.5000
    0.5000    0.5000   -0.5000    0.5000
    0.5000   -0.5000   -0.5000   -0.5000
    0.5000    0.5000    0.5000   -0.5000


R =

   14.0000    0.0000   -0.0000   -0.0000
         0   14.0000   -0.0000    0.0000
         0         0   14.0000    0.0000
         0         0         0   14.0000
```

Example: QR Factorization Not Unique

When you compare the results from `cordicqr` and the QR function in MATLAB, you will notice that the QR factorization is not unique. It is only important that Q is orthogonal, R is upper triangular, and `Q*R - A` is small.

Here is a simple example that shows the difference.

```
m = 3;
A = ones(m)
```

```
A =

     1     1     1
     1     1     1
     1     1     1
```

The built-in QR function in MATLAB uses a different algorithm and produces:

```
[Q0,R0] = qr(A)
```

```
Q0 =

   -0.5774    0.8165   -0.0000
   -0.5774   -0.4082   -0.7071
   -0.5774   -0.4082    0.7071


R0 =

   -1.7321   -1.7321   -1.7321
         0   -0.0000   -0.0000
         0         0    0.0000
```

And the `cordicqr` function produces:

```
[Q,R] = cordicqr(A)
```

```
Q =

    0.5774    0.7495    0.3240
    0.5774   -0.6553    0.4871
    0.5774   -0.0942   -0.8110


R =

    1.7321    1.7321    1.7321
         0    0.0000    0.0000
         0         0   -0.0000
```

Notice that the elements of Q from function `cordicqr` are different from Q0 from built-in QR. However, both results satisfy the requirement that Q is orthogonal:

```
Q0*Q0'
```

```
ans =

    1.0000    0.0000   -0.0000
    0.0000    1.0000   -0.0000
   -0.0000   -0.0000    1.0000
```

```
Q*Q'
```

```
ans =

    1.0000    0.0000    0.0000
    0.0000    1.0000   -0.0000
    0.0000   -0.0000    1.0000
```

And they both satisfy the requirement that `Q*R - A` is small:

```
Q0*R0 - A
```

```
ans =

   1.0e-15 *

   -0.1110   -0.1110   -0.1110
   -0.1110   -0.1110   -0.1110
   -0.1110   -0.1110   -0.1110
```

```
Q*R - A
```

```
ans =

   1.0e-15 *

   -0.2220    0.2220    0.2220
```

```
    0.4441         0         0
    0.2220    0.2220    0.2220
```

Solving Systems of Equations Without Forming Q

Given matrices A and B, you can use the QR factorization to solve for X in the following equation:

```
A*X = B.
```

If A has more rows than columns, then X will be the least-squares solution. If X and B have more than one column, then several solutions can be computed at the same time. If `A = Q*R` is the QR factorization of A, then the solution can be computed by back-solving

```
R*X = C
```

where `C = Q'*B`. Instead of forming Q and multiplying to get `C = Q'*B`, it is more efficient to compute C directly. You can compute C directly by applying the rotations to the rows of B instead of to the columns of an identity matrix. The new algorithm is formed by the small modification of initializing `C = B`, and operating along the rows of C instead of the columns of Q.

```
function [R,C] = cordicrc(A,B,niter)
  Kn = inverse_cordic_growth_constant(niter);
  [m,n] = size(A);
  R = A;
  C = B;
  for j=1:n
    for i=j+1:m
      [R(j,j:end),R(i,j:end),C(j,:),C(i,:)] = ...
          cordicgivens(R(j,j:end),R(i,j:end),C(j,:),C(i,:),niter,Kn);
    end
  end
end
```

You can verify the algorithm with this example. Let A be a random 3-by-3 matrix, and B be a random 3-by-2 matrix.

```
A = [-0.8201    0.3573   -0.0100
     -0.7766   -0.0096   -0.7048
     -0.7274   -0.6206   -0.8901];

B = [-0.9286    0.3575
      0.6983    0.5155
      0.8680    0.4863];
```

Compute the QR factorization of A.

```
[Q,R] = cordicqr(A)
```

```
Q =

   -0.6105    0.6133    0.5012
   -0.5781    0.0876   -0.8113
   -0.5415   -0.7850    0.3011


R =
```

```
    1.3434    0.1235    0.8955
         0    0.7054    0.6309
         0         0    0.2988
```

Compute `C = Q'*B` directly.

```
[R,C] = cordicrc(A,B)
```

```
R =

    1.3434    0.1235    0.8955
         0    0.7054    0.6309
         0         0    0.2988


C =

   -0.3068   -0.7795
   -1.1897   -0.1173
   -0.7706   -0.0926
```

Subtract, and you will see that the error difference is on the order of roundoff.

```
Q'*B - C
```

```
ans =

   1.0e-15 *

   -0.0555    0.3331
         0         0
    0.1110    0.2914
```

Now try the example in fixed-point. Declare A and B to be fixed-point types.

```
A = sfi(A)
```

```
A =

   -0.8201    0.3573   -0.0100
   -0.7766   -0.0096   -0.7048
   -0.7274   -0.6206   -0.8901

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

```
B = sfi(B)
```

```
B =

   -0.9286    0.3575
    0.6983    0.5155
```

```
    0.8680    0.4863

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 16
        FractionLength: 15
```

The necessary growth is 1.6468 times the square-root of the number of rows of A.

```
bit_growth = ceil(log2(cordic_growth_constant*sqrt(m)))
```

```
bit_growth =

     2
```

Initialize R with the same values as A, and allow for bit growth.

```
R = sfi(A, get(A,'WordLength')+bit_growth, get(A,'FractionLength'))
```

```
R =

   -0.8201    0.3573   -0.0100
   -0.7766   -0.0096   -0.7048
   -0.7274   -0.6206   -0.8901

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 15
```

The growth in C is the same as R, so initialize C and allow for bit growth the same way.

```
C = sfi(B, get(B,'WordLength')+bit_growth, get(B,'FractionLength'))
```

```
C =

   -0.9286    0.3575
    0.6983    0.5155
    0.8680    0.4863

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
            WordLength: 18
        FractionLength: 15
```

Compute C = Q'*B directly, overwriting R and C.

```
[R,C] = cordicrc(R,C)
```

```
R =

    1.3435    0.1233    0.8954
         0    0.7055    0.6308
         0         0    0.2988

          DataTypeMode: Fixed-point: binary point scaling
            Signedness: Signed
```

```
              WordLength: 18
          FractionLength: 15

  C =

     -0.3068   -0.7796
     -1.1898   -0.1175
     -0.7706   -0.0926

            DataTypeMode: Fixed-point: binary point scaling
              Signedness: Signed
              WordLength: 18
          FractionLength: 15
```

An interesting use of this algorithm is that if you initialize B to be the identity matrix, then output argument C is Q'. You may want to use this feature to have more control over the data type of Q. For example,

```
A = [-0.8201    0.3573   -0.0100
     -0.7766   -0.0096   -0.7048
     -0.7274   -0.6206   -0.8901];
B = eye(size(A,1))
```

```
  B =

       1     0     0
       0     1     0
       0     0     1
```

```
[R,C] = cordicrc(A,B)
```

```
  R =

      1.3434    0.1235    0.8955
           0    0.7054    0.6309
           0         0    0.2988


  C =

     -0.6105   -0.5781   -0.5415
      0.6133    0.0876   -0.7850
      0.5012   -0.8113    0.3011
```

Then C is orthogonal

```
C'*C
```

```
  ans =

      1.0000    0.0000    0.0000
      0.0000    1.0000   -0.0000
      0.0000   -0.0000    1.0000
```

and `R = C*A`

```
R - C*A
```

```
ans =

   1.0e-15 *

    0.6661   -0.0139   -0.1110
    0.5551   -0.2220    0.6661
   -0.2220   -0.1110    0.2776
```

Links to the Documentation

Fixed-Point Designer™

- `bitsra` Bit shift right arithmetic
- `fi` Construct fixed-point numeric object
- `fimath` Construct `fimath` object
- `fipref` Construct `fipref` object
- `get` Property values of object
- `globalfimath` Configure global `fimath` and return handle object
- `isfi` Determine whether variable is `fi` object
- `sfi` Construct signed fixed-point numeric object
- `upperbound` Upper bound of range of `fi` object
- `fiaccel` Accelerate fixed-point code

MATLAB

- `bitshift` Shift bits specified number of places
- `ceil` Round toward positive infinity
- `double` Convert to double precision floating point
- `eps` Floating-point relative accuracy
- `eye` Identity matrix
- `log2` Base 2 logarithm and dissect floating-point numbers into exponent and mantissa
- `prod` Product of array elements
- `qr` Orthogonal-triangular factorization
- `repmat` Replicate and tile array
- `single` Convert to single precision floating point
- `size` Array dimensions
- `sqrt` Square root
- `subsasgn` Subscripted assignment

Functions Used in this Example

These are the MATLAB functions used in this example.

CORDICQR computes the QR factorization using CORDIC.

- `[Q,R] = cordicqr(A)` chooses the number of CORDIC iterations based on the type of A.
- `[Q,R] = cordicqr(A,niter)` uses `niter` number of CORDIC iterations.

CORDICRC computes R from the QR factorization of A, and also returns `C = Q'*B` without computing Q.

- `[R,C] = cordicrc(A,B)` chooses the number of CORDIC iterations based on the type of A.
- `[R,C] = cordicrc(A,B,niter)` uses `niter` number of CORDIC iterations.

CORDIC_GROWTH_CONSTANT returns the CORDIC growth constant.

- `cordic_growth = cordic_growth_constant(niter)` returns the CORDIC growth constant as a function of the number of CORDIC iterations, `niter`.

GIVENSQR computes the QR factorization using standard Givens rotations.

- `[Q,R] = givensqr(A)`, where A is M-by-N, produces an M-by-N upper triangular matrix R and an M-by-M orthogonal matrix Q so that A = Q*R.

CORDICQR_MAKEPLOTS makes the plots in this example by executing the following from the MATLAB command line.

```
load A_3_by_3_for_cordicqr_demo.mat
niter=32;
[Q,R] = cordicqr_makeplots(A,niter)
```

References

1. Ray Andraka, "A survey of CORDIC algorithms for FPGA based computers," 1998, ACM 0-89791-978-5/98/01.
2. Anthony J Cox and Nicholas J Higham, "Stability of Householder QR factorization for weighted least squares problems," in Numerical Analysis, 1997, Proceedings of the 17th Dundee Conference, Griffiths DF, Higham DJ, Watson GA (eds). Addison-Wesley, Longman: Harlow, Essex, U.K., 1998; 57-73.
3. Gene H. Golub and Charles F. Van Loan, *Matrix Computations,* 3rd ed, Johns Hopkins University Press, 1996, section 5.2.3 Givens QR Methods.
4. Daniel V. Rabinkin, William Song, M. Michael Vai, and Huy T. Nguyen, "Adaptive array beamforming with fixed-point arithmetic matrix inversion using Givens rotations," Proceedings of Society of Photo-Optical Instrumentation Engineers (SPIE) -- Volume 4474 Advanced Signal Processing Algorithms, Architectures, and Implementations XI, Franklin T. Luk, Editor, November 2001, pp. 294--305.
5. Jack E. Volder, "The CORDIC Trigonometric Computing Technique," Institute of Radio Engineers (IRE) Transactions on Electronic Computers, September, 1959, pp. 330-334.
6. Musheng Wei and Qiaohua Liu, "On growth factors of the modified Gram-Schmidt algorithm," Numerical Linear Algebra with Applications, Vol. 15, issue 7, September 2008, pp. 621-636.

Cleanup

```
fipref(originalFipref);
globalfimath(originalGlobalFimath);
close all
set(0, 'format', originalFormat);
%#ok<*MNEFF,*NASGU,*NOPTS,*ASGLU>
```