```
1x{Float64}:
0  3.0
0  6.0
0  9.0
```

practice if m is too big...

$H_k = I - 2v_k v_k^T$ instead store the v_k's

$v_k \in \mathbb{R}^m$ for $k = 1, \ldots, n$

Instead of Qx do $H_1(H_2 \cdots (H_n x))$ which is efficient

also instead of $Q^T b$ do $H_n(H_{n-1} \cdots (H_1 b))$ which is efficient

why is that efficient?

Idea: Rather than constructing the matrix $Q = H_1 H_2 \cdots H_n$ or $Q^T = H_n H_{n-1} \cdots H_2 H_1$

$A \in \mathbb{R}^{m \times n}$ so $A = Q R$ ($m \times n$ = $m \times m$ $m \times n$)

$H_1 = I - 2v_1 v_1^T$, $H_2 = I - 2v_2 v_2^T$, ..., $H_n = I - 2v_n v_n^T$

one H for each column.

$Qx = H_1 H_2 \cdots H_n x = H_1(H_2(\cdots(H_n x)\cdots))$

($m \times m$ m = $m \times m$ $m \times m$ $m \times 1$)

first compute $H_n x$

scalar-vector mult = m mult

$H_n x = (I - 2v_n v_n^T)x = x - 2v_n v_n \cdot x =$

$v_n \cdot x$: dot product = m mult + $(m-1)$ additions

$x - (\cdots)$: vector subtraction; m subtractions (count as Add)

Total: mult = $m + m = 2m$

add/sub = $2m - 1$

Now do that n times: mult = $2mn$

$H_1 H_2 \cdots H_n x$ add/sub = $(2m-1)n$

m scalar vector mult = total of m^2 mult

$Qx = q_1 x_1 + q_2 x_2 + \cdots + q_m x_m$

$m(m-1)$ vector additions

mult m^2

add: $m(m-1)$

If $m \gg n$ then $2mn \ll m^2$

and also $(2m-1)n \ll m(m-1)$

Alternatively after finding $v_1, v_2, \ldots, v_n$ one could compute the reduced QR factorization

$$\underset{m\times n}{A} = \underset{m\times n}{\tilde{Q}}\ \underset{n\times n}{\tilde{R}}$$

$$Q = H_1 H_2 \cdots H_n \underset{m\times m}{I}$$

$$\tilde{Q} = H_1 H_2 \cdots H_n \begin{bmatrix} I_{n\times n} \\ 0 \end{bmatrix} \in \mathbb{R}^{m\times n}$$

$m \times n$

to get the reduced QR from the v_i's.

To compute $\tilde{Q}$ change line 8 so $Q^T = \begin{bmatrix} I_{n\times n} & 0 \end{bmatrix} \in \mathbb{R}^{n\times m}$

```
5 """
6 function qrfact(A)
7     m,n = size(A)
8     Qt = diagm(ones(m))
9     R = float(copy(A))
10    for k in 1:n                    ← k^th column
11        z = R[k:m,k]
12        w = [ -sign(z[1])*norm(z) - z[1]; -z[2:end] ]
13        nrmw = norm(w)   ↖ choose ± for c except problem if
14        if nrmw < eps() continue; end    # skip this iteration
15        v = w / nrmw;
16        # Apply the reflection to each relevant column of A and
17        for j in 1:n
18            R[k:m,j] -= v*( 2*(v'*R[k:m,j]) )   ← Mult R by H
19        end
20        for j in 1:m
21            Qt[k:m,j] -= v*( 2*(v'*Qt[k:m,j]) )
22        end                          ↑ Mult H_{k-1} by H_k
23    end
24    return Qt',triu(R)
25 end
```

Chapter 4 root finding.

Example finding $\sqrt{2}$.

Guess & Check. (Bisection Method)

$f(x) = x^2 - 2$

Suppose $x > 0$ then

if $f(x) > 0$ then $x > \sqrt{2}$

if $f(x) < 0$ then $x < \sqrt{2}$

Start with two points a, b such that $f(a) < 0$ and $f(b) > 0$

Thus $\sqrt{2} \in (a, b)$.

Guess the midpoint $c = \frac{a+b}{2}$ midpoint.

If $f(c) > 0$ then $c > \sqrt{2}$ so c is too big but not as big as b

thus $\sqrt{2} \in (a, c)$

If $f(c) < 0$ then $c < \sqrt{2}$ so c is too small but not as small as a

thus $\sqrt{2} \in (c, b)$

Now repeat the process with the midpoint of the new interval...

```
julia> f(x)=x^2-2
f (generic function with 1 m

julia> a=0
0

julia> b=2
2

julia> c=(a+b)/2
1.0

julia> f(c)
-1.0

julia> a=c
1.0

julia> [a,b]
2-element Vector{Float64}:
 1.0
 2.0
```

length [a,b] = 2

this approx √2 to within 1 unit.

neg. too small

new lower bound

length [a,b] = 1

```
julia> c=(a+b)/2
1.5

julia> f(c)
0.25

julia> b=c
1.5

julia> [a,b]
2-element Vector{Float64}:
 1.0
 1.5
```

this approximates √2 to within ±.5

too big

new upper bound

length [a,b] = 0.5

```
julia> c=(a+b)/2
1.25

julia> f(c)
-0.4375

julia> a=c
1.25

julia> [a,b]
2-element Vector{Float64}:
 1.25
 1.5
```

approximates √2 to within ± 0.25

length [a,b] = 0.25

Advantages

- simple
- guaranteed to converge
- easy bound on the error

Disadvantage

- slow convergence. to determine 52 bits of precision (double precision) one has to make 52 guesses ... also depends on the length of the initial interval.

```
f(x)=x^2-2
a=0
b=2
for n=1:52
    c=(a+b)/2
    println("sqrt(2)=",c," p/m ",(b-a)/2)
    if f(c)>0
        global b=c
    else
        global a=c
    end
end
```

```
julia> include("bisect.jl")
sqrt(2)=1.0 p/m 1.0
sqrt(2)=1.5 p/m 0.5
sqrt(2)=1.25 p/m 0.25
sqrt(2)=1.375 p/m 0.125
sqrt(2)=1.4375 p/m 0.0625
sqrt(2)=1.40625 p/m 0.03125
sqrt(2)=1.421875 p/m 0.015625
sqrt(2)=1.4140625 p/m 0.0078125
sqrt(2)=1.41796875 p/m 0.00390625
sqrt(2)=1.416015625 p/m 0.001953125
sqrt(2)=1.4150390625 p/m 0.0009765625
sqrt(2)=1.41455078125 p/m 0.00048828125
sqrt(2)=1.414306640625 p/m 0.000244140625
sqrt(2)=1.4141845703125 p/m 0.0001220703125
sqrt(2)=1.41424560546875 p/m 6.103515625e-5
```

```
1 f(x)=x^2-2
2 a=0
3 b=2
4 cold=0
5 while true
6     c=(a+b)/2
7     if cold==c
8         break
9     end
10    global cold=c
11    println("sqrt(2)=",c," p/m ",(b-a)/2)
12    if f(c)>0
13        global b=c
14    else
15        global a=c
16    end
17 end
```

check for when guesses repeat...
no more precision possible after that...

ends like this

```
sqrt(2)=1.414213562372879 p/m 4.547473508864641e-13
sqrt(2)=1.4142135623731065 p/m 2.2737367544323206e-13
sqrt(2)=1.4142135623729928 p/m 1.1368683772161603e-13
sqrt(2)=1.4142135623730496 p/m 5.684341886080802e-14
sqrt(2)=1.414213562373078 p/m 2.842170943040401e-14
sqrt(2)=1.4142135623730923 p/m 1.4210854715202004e-14
sqrt(2)=1.4142135623730994 p/m 7.105427357601002e-15
sqrt(2)=1.4142135623730958 p/m 3.552713678800501e-15
sqrt(2)=1.414213562373094 p/m 1.7763568394002505e-15
sqrt(2)=1.414213562373095 p/m 8.881784197001252e-16
sqrt(2)=1.4142135623730954 p/m 4.440892098500626e-16
sqrt(2)=1.4142135623730951 p/m 2.220446049250313e-16
sqrt(2)=1.414213562373095 p/m 1.1102230246251565e-16
```

Same value we ended at... when checking for finding repeat guesses by hand...

WARNING → Note: Bisection is one of the only methods where you can end a loop by checking for exact equality between floating point numbers.