Computer intensive statistical methods

Introduction

Alexander Ploner
alexander.ploner@ki.se

Medical Epidemiology & Biostatistics
Karolinska Institutet

2009-09-01

Lecture: Numerical concepts and linear equation systems

Numbers, errors, computation

Linear equation systems

Linear least squares

Computation & Errors I

Many problems requiring computation can be formulated as

$$G(X) = Y$$

for some quantities X, Y, and some function G.
We can distinguish situations where

- X, G known and Y unknown: *direct* problem,
- Y, G known and X unknown: *indirect* problem,
- X, Y known and G unknown: *functional* problem.

Computation & Errors II

In any kind of non-trivial computation, we have distinct sources of error:

Rounding error due to finite precision of the computations

Truncation error due to approximation of functions involved (think Taylor expansion)

Termination of iterations due to iterative algorithms

Statistical error due to algorithmic sampling (think Monte Carlo integration)

Computation & Errors III

Other obvious error sources:

- data errors
- model errors

due to stochastic variation and/or model misspecification

$\Rightarrow$ not usually the subject of numerical analysis, unless they interact with other error sources

(And the number one error source in using software of any kind. . . ?)

Terminology I

Exact algorithms do not suffer from truncation error; if run with full (i.e. hypothetical) precision, they will deliver exact solutions (practically: linear problems).

Stable algorithms are insensitive to small changes in the data of the problem or rounding errors.

Forward error measures how a change in the data of the problem or due to rounding impacts the solution of a problem.

Backward error measures how a change in the solution of a problem corresponds to a perturbation in the original problem.

Number Representations

In digital computers, numbers are recorded as binary digits (bits), usually in groups of eight bits (bytes). Usually: four to eight bytes for integers, eight or more bytes for real numbers. In a 4 byte representation, the set of integers can be expressed as the numbers

$$I = \sum_{i=0}^{31} x_i 2^i - 2^{31}$$

Details can vary if e.g. some bit patterns are reserved for special codes.

Exercise: Determine the range of integers in your favorite statistical analysis package!

Floating Point Representation

Real numbers are approximated by floating point numbers consisting of

- a sign bit,
- an integer exponent (usually for base 2) that scales the magnitude of the number,
- a mantissa of t bits that gives the binary digits of the number.

$$f = (-1)^{x_0} \left(\sum_{i=1}^{t} x_i 2^{-i} \right) 2^k$$

- Traditionally, single precision is 4 bytes and double precision is 8 bytes.
- DP means 1 sign bit, 11 bits for the exponent, and 52 bits for the mantissa. IEEE standard reserves some bit patterns for special codes like `NaN`.

Machine Precision

The precision of the floating point implementation on a specific system is usually expressed as one of

- distance from 1 to the next biggest number in the floating point representation,
- the smallest number that can be added to 1 that results in a number bigger than 1 (not quite the same, see `[Gray]`, p.9)
- the smallest number that can be subtracted from one that results in a number smaller than 1,
- some variant thereof.

The precision is referred to as ϵ_M and assumed to limit the *relative rounding error*:

$$\frac{[x - f(x)]}{|x|} \leq \epsilon_M$$

For an eight byte implementation, $\epsilon_M \approx 2.22e-16$. This is *not* the smallest representable number!

Floating Point Operations

Fairly safe operations:

- addition,
- multiplication,
- division.

The relative error of the result is a small multiple of ϵ_M (in first approaximation). The relative error of a sequence of N operations will grow like $O(N\epsilon_M)$.

Dangerous:

- subtraction of two numbers of comparable magnitude

The leading bits of the mantissae cancel out, and everything after a few trailing bits is just truncated. Worst case: the relative error can be arbitrarily large, no correct digits! See `[Gray]`, p.11)

Linear Equation Systems (LES)

Much computation is concerned with LES in p variables and n equations:

$$\begin{array}{ccccccc} a_{11}x_1 & + & \ldots & + & a_{1p}x_p & = & b_1 \\ \vdots & & & & \vdots & = & \vdots \\ a_{n1}x_1 & + & \ldots & + & a_{np}x_p & = & b_n \end{array}$$

Which we traditionally write in matrix notation:

$$Ax = b$$

In statistics usually

- written as $y = X\beta$
- $n > p$ (overdetermined system)

For now we want to look at solving LES with $n = p$ and therefore use a more numerical notation.

Some useful notation

- Vector norms: 1, 2, ∞
- Matrix norms: the same with

$$||A|| = \sup_{x \neq 0} \frac{||Ax||}{||x||}$$

  which yields
  - $||A||_1 = \max_j \sum_i |a_{ij}|$
  - $||A||_\infty = \max_i \sum_j |a_{ij}| = ||A^T||_1$
- Condition number:

$$\kappa = ||A||||A^{-1}||$$

  with $\kappa \geq 1$ A singular implies $\kappa = \infty$
- In practice:
  - $1/\kappa$ less dangerous
  - Estimation, not calculation, without calculating A^{-1}.

Condition of LES I

Let's assume that A can be read into the computer without loss of precision (no rounding error when converting to floating point representation). Regardless of algorithm, we will not get the exact solution x_0, but a numerical solution x^*.
The x^* is the exact solution of a LES with the same matrix A, but a perturbed right hand side:

$$Ax^* = b^*$$

with

$$A(x + \delta x) = b + \delta b \quad \text{and} \quad \delta b = b^* - b = Ax^* - b$$

so δb is the exact residual of the numerical solution.
The most stringent equation linking the size of the residual to the size of the actual error in x:

$$\frac{||x - x^*||}{||x||} \leq \kappa(A) \frac{||b - b^*||}{||b||}$$

Condition of LES II

The relative error on the rhs is magnified by the condition number of A
The best we can expect for the the size of the relative error is ϵ_M. The best case scenario is

$$\frac{||x - x^*||}{||x||} \leq \kappa(A)\epsilon_M$$

regardless of the actual algorithm involved!

How to solve them, take one I

Gauss eliminationation:

- the LES and its solution are not changed by adding or subtracting multiples of linear equations
- We can zero out everything under the diagonal until we have a system in upper triangular form.

Example:

$$\begin{pmatrix} 10 & -7 & 0 \\ -3 & 2 & 6 \\ 5 & -1 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 7 \\ 4 \\ 6 \end{pmatrix}$$

How to solve them, take one II

Performing Gaussian elimination on the combined matrix of coefficients and rhs:

$$\left[\begin{array}{ccc|c} 10 & -7 & 0 & 7 \\ -3 & 2 & 6 & 4 \\ 5 & -1 & 5 & 6 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|c} 10 & -7 & 0 & 7 \\ 0 & -0.1 & 6 & 6.1 \\ 0 & 2.5 & 5 & 2.5 \end{array}\right]$$

$$\Rightarrow \left[\begin{array}{ccc|c} 10 & -7 & 0 & 7 \\ 0 & -0.1 & 6 & 6.1 \\ 0 & 0 & 155 & 155 \end{array}\right]$$

How to solve them, take one III

We can solve this triangular system easily from the bottom up:

$$\begin{array}{rcll} x_3 & = & 155/155 & = & 1 \\ x_2 & = & (6.1 - 6)/(-0.1) & = & -1 \\ x_1 & = & (7 - 7 - 0)/10 & = & 0 \end{array}$$

This is *back substitution*. Obviously, forward substitution on a lower triangular system works the same.

How to solve them, take one IV

```
GaussElim1 = function(A,b)
{
    # Gaussian elimination WITHOUT pivoting
    # no checks, no prisoners, no mercy
    n = nrow(A)
    X = cbind(A,b)
    for (i in 1:(n-1)) {
        for (j in (i+1):n){
            fac = X[j,i]/X[i,i]
            for (k in i:(n+1)){
                X[j,k] = X[j,k] - X[i,k]*fac
            }
        }
    }
```

How to solve them, take one V

```
    # back substitution
    x = rep(0,n)
    x[n] = X[n, n+1]/X[n,n]
    for (i in (n-1):1) {
        dummy = 0
        for (j in n:(i+1)) {
            dummy = dummy + x[j]*X[i,j]
        }
        x[i] = (X[i, n+1] - dummy)/X[i,i]
    }
    x
}
```

Pivoting to avoid zeros I

The algorithm breaks down when we hit a zero on the diagonal (overflow). We have to get rid of the zero in a way that does not change the properties of the LES – we can always

- swap rows (change the order of the equations)
- swap columns (change the order of the variables)

We do not want to mess up the part of the matrix that we have already processed. We can therefore choose only

- rows below,
- columns to the right

of the offending zero element.

Pivoting to avoid zeros II

Example: as before, but with minimal modifications (underlined)

$$\left[\begin{array}{ccc|c} 10 & -7 & 0 & 7 \\ -3 & \underline{2.1} & 6 & \underline{3.9} \\ 5 & -1 & 5 & 6 \end{array}\right] \Rightarrow \left[\begin{array}{ccc|c} 10 & -7 & 0 & 7 \\ 0 & 0 & 6 & 6 \\ 0 & 2.5 & 5 & 2.5 \end{array}\right]$$

$$\Rightarrow \left[\begin{array}{ccc|c} 10 & -7 & 0 & 7 \\ 0 & 2.5 & 5 & 2.5 \\ 0 & 0 & 6 & 6 \end{array}\right]$$

Pivoting to avoid zeros III

Note:

1. The matrix is not singular – the zero diagonal pivot is purely bad luck.
2. Swapping rows 2 and 3 terminates here – IRL, we just continue from element (3,3) onwards.
3. As we only swapped rows, we can back substitute exactly as before.
4. This modification preserves the solution (0, -1, 1).

Numerical stability I

Example: Modification as before, but this time adding/subtracting $\delta = 0.1 - 1000 \times 2^{-52}$

$$\left[\begin{array}{ccc|c} 10 & -7 & 0 & 7 \\ -3 & 2+\delta & 6 & 4-\delta \\ 5 & -1 & 5 & 6 \end{array}\right]$$

$$\Rightarrow \left[\begin{array}{ccc|c} 10 & -7 & 0 & 7 \\ 0 & -2.22e-13 & 6 & 6 \\ 0 & 2.5 & 5 & 2.5 \end{array}\right]$$

$$\Rightarrow \left[\begin{array}{ccc|c} 10 & -7 & 0 & 7 \\ 0 & -2.22e-13 & 6 & 6 \\ 0 & 0 & 6.75e+13 & 6.75e+13 \end{array}\right]$$

When we do the back substitution, we get the solution (0.0056, -0.992, 1), which is clearly wrong.

Numerical stability II

What happened is that

1. due to a very small pivot in the second step, we got a huge pivot in the third step.
2. We therefore had already lost some digits when transforming the last equation.
3. This relatively small error in the solution for the third variable (which NB still rounds to 1) was magnified again when back substituting by the small divisor in the second equation!

Gaussian elimination without numerical pivoting is an unstable algorithm!

Numerical pivoting I

Gaussian elimination can be made numerically stable by pivoting

- the absolutely largest element in the current column to the diagonal through row swapping (*partial pivoting*),
- the absolutely largest element in the unprocessed rest of the matrix to the diagonal through row- and column swapping (*full pivoting*).

Gaussian elimination with partial pivoting is generally sufficiently stable for most problems!
(Note that this takes implicitly care of zeros on the diagonal!)

Numerical pivoting II

```
GaussElim2 = function(A,b)
{
    # Gaussian elimination w. partial pivoting
    n = nrow(A)
    X = cbind(A,b)
    for (i in 1:(n-1)) {
        # Find maximum pivot
        imax = i
        pmax = abs(X[i,i])
        for (j in (i+1):n){
            if (abs(X[j,i]) > pmax) {
                imax = j
                pmax = abs(X[j,i])
            }
        # Check for singularity
        # Not really kosher
        if (pmax==0) {
            stop("matrix singular")
        }
```

Numerical pivoting III

```
        # Swap if required
        if (imax != i) {
            for (j in i:(n+1)) {
                dummy = X[i,j]
                X[i,j] = X[imax,j]
                X[imax,j] = dummy
            }
        }
        # Proceed with elimination as before
        for (j in (i+1):n){
            fac = X[j,i]/X[i,i]
            for (k in i:(n+1)){
                X[j,k] = X[j,k] - X[i,k]*fac
            }
        }
    }
```

Numerical pivoting IV

```
    # Back substitution as before
    x = rep(0,n)
    x[n] = X[n, n+1]/X[n,n]
    for (i in (n-1):1) {
        dummy = 0
        for (j in n:(i+1)) {
            dummy = dummy + x[j]*X[i,j]
        }
        x[i] = (X[i, n+1] - dummy)/X[i,i]
    }
    x
}
```

Practical considerations

Operation & memory count:

- decomposition: $N^3/3$
- back substitution: $N^2/2$
- no extra memory required

Multiple right hand sides:

- can be run simultaneously BUT
- also have to be run simultaneously,

because matrix and rhs are processed together.
The inverse matrix can be calculated by having the identity matrix as multiple rhs (naively: $4N^3/3$ operations, but can be reduced to N^3)

How to solve them, take two I

LU-decomposition is really the smart way of doing Gauss elimination: by running it on the matrix A alone and recording the pivot elements, we can decompose the matrix as follows:

$$A = LU$$

where U is the upper triangular form from before, and L is lower triangular with diagonal 1, with the factors that were used to zero out the elements in A in the corresponding place.
For any given rhs b, we therefore have:

$$LUx = b$$

which we solve in in two steps:

1. $Ly = b$ and forward solving,
2. $Ux = y$ and backward solving.

How to solve them, take two II

In case of partial pivoting, we can write this as

$$PULx = b$$

where P is a permutation matrix (i.e. identity matrix with permuted rows). We then solve $ULx = Pb$ as above (and apply to the solution to correct the order).
Example: the matrix from the first system can be written as

$$\begin{pmatrix} 10 & -7 & 0 \\ -3 & 2 & 6 \\ 5 & -1 & 5 \end{pmatrix} =$$

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0.5 & 1 & 0 \\ -0.3 & -0.04 & 1 \end{pmatrix} \begin{pmatrix} 10 & -7 & 0 \\ 0 & 2.5 & 5 \\ 0 & 0 & 6.2 \end{pmatrix}$$

Practical considerations I

Operation & memory count:

- decomposition: $N^3/3$
- back substitution: $N^2/2 + N^2/2 = N^2$ for back + forward substitution
- little extra memory required

Multiple right hand sides: YES! Note that if we have

- a large number of rhs,
- which are known at the time of computation,
- and we're sure there won't be any other,

then the elimination procedure is cheaper due to the fact that we have only one back substitution. If there are only a few rhs, or if we need to solve the system repeatedly instead of only once, the LU-decomposition is either equivalent or better.

Practical considerations II

Again the inverse matrix can be calculated by having the identity matrix as multiple rhs, with some ingenuity in N^3 operations.
The point is though that unless we explicitly need the coefficients of A^{-1}, we do not need to do this: we can always compute $A^{-1}b$ as the solution to

$$Ax = b$$

in N^2 steps using the LU-decomposition – this is as expensive as multiplying the vector b with an explicitly known A^{-1}!
Note that we get the determinant of A as the product of the diagonal elements of U.

Numerical pivoting I

What happens? This can be explained by a result due to Wilkinson: if we solve the LES by computing (perturbed) $\tilde{L}$, $\tilde{U}$, and $\tilde{x}$, then there exists a δA so that

$$(A + \delta A)\,\tilde{x} = b$$

with

$$\frac{||\delta A||_\infty}{||A||_\infty} \leq n\epsilon_M \left(3 + \frac{5||\tilde{L}||_\infty ||\tilde{U}||_\infty}{||A||_\infty}\right)$$

The problem here is one of big intermediate results that cancel out during the multiplication of LU, but loose significant digits in the process. The solution here is again to pivot away diagonal elements that are absolutely small.
Partial pivoting only garantuees only a bound on the lower triangular matrix:

$$||\tilde{L}||_\infty \leq 1$$

Numerical pivoting II

In practice, this is sufficient for most problems, though in the rare worst cases the fraction on the right hand side can grow exponentially with n.
Full pivoting OTOH additionally ensures that

$$\frac{||\tilde{U}||_\infty}{||A||_\infty} \leq n^{log(n)/2}$$

In practice, the bound is usually closer to n.

Cholesky decomposition I

One more triangular decomposition! For positive definite A, we can get the special decomposition

$$A = LL^T$$

where L is lower triangular. This can be considered the square root of A.
Cholesky can be computed in the same manner as a the LU decomposition, but we only ever have to do the lower half (the L, but with non-1 diagonal). This means that we need half the operation count of LU ($N^3/6$), and we can either

- only ever use the lower half of the matrix, overwrite it successively, and save half the memory, or
- store L below the diagonal and return the new diagonal returned in an extra vector; this means we have both the original matrix and the decomposition with minimum extra memory.

Cholesky decomposition II

The Cholesky decomposition is numerically stable even without pivoting! Positive definitness garantuees that the diagonal elements are large enough throughout the process.

Example: In order to evaluate $b^T A^{-1} b$ repeatedly, we decompose A, compute $L^T x = b$ in $N^2/2$ steps and compute squared vector norm of the solution: effectively, almost half the operations for evaluating a quadratic form than for multiplying a vector and matrix!

Cholesky decomposition III

Example in pseudo-Fortran code from [NR]:

```
SUBROUTINE choldc(a,n,np,p)
INTEGER n,np
REAL a(np,np),p(n)
    Given a positive-definite symmetric matrix a(1:n,1:n), with physical dimension np, this
    routine constructs its Cholesky decomposition, A = L·L^T. On input, only the upper triangle
    of a need be given; it is not modified. The Cholesky factor L is returned in the lower triangle
    of a, except for its diagonal elements which are returned in p(1:n).
INTEGER i,j,k
REAL sum
do 13 i=1,n
    do 12 j=i,n
        sum=a(i,j)
        do 11 k=i-1,1,-1
            sum=sum-a(i,k)*a(j,k)
        enddo 11
        if(i.eq.j)then
            if(sum.le.0.)pause 'choldc failed'       a, with rounding errors, is not
            p(i)=sqrt(sum)                             positive definite.
        else
            a(j,i)=sum/p(i)
        endif
    enddo 12
enddo 13
return
END
```

Special Matrices and Tactics

Special properties of matrices like positive definitness often allow modification of standard algorithms that

- run faster, and/or
- run in less memory.

Examples: matrix A with distinct pattern or a larger proportion of zero entries (patterned and sparse matrices, e.g. tridiagonal or banded matrices). Often more efficiently storage/faster decomposed than general matrices by

- modifying standard algorithms like the LU decomposition
- using iterative methods, e.g. Gauss-Seidel
- by explicitly solving the linear equations as a minimization problem where

$$||Ax - b|| \rightarrow \min$$

Summary: LES

- The condition of the matrix A limits the achievable precision inherently.
- There is no optimal all purpose LES solver.
- However, LU decomposition with partial pivoting offers a reasonable tradeoff between stability and performance for general matrices.
- Decomposition is usually better than elimination.
- Pivoting can improve the stability of an algorithm.
- Exploiting special properties and patterns of the matrix A can offer huge advantages in memory and speed.

LES & Linear Least Squares I

If we write an over determined system ($n > p$) as usually

$$X\beta = y$$

than this will generally have no solution. The usual approach is therefore to chose β so as to minimize the sum of squared distances from the rhs (aka residual sum of squares):

$$||X\beta - y||_2^2 \rightarrow \min$$

This is an obvious generalization and guaranteed to have a solution (though not necessarily unique).

LES & Linear Least Squares II

Mathematically, this is equivalent to solving the normal equations

$$X^T X\beta = X^T y$$

using e.g. the Cholesky decomposition.

Practically, this is **not** the most stable way of proceeding, as the computation of $X^T X$ adds both operations and rounding errors.

QR Decomposition I

We try again a triangular reduction. Assuming that we can write

$$X = QR$$

we can then try to solve the new problem

$$||QR\beta - y||_2^2 = |R\beta - Q^{-1}y||_2^2 \rightarrow \min$$

provided that Q

- preserves the Euclidean norm,
- is easy to invert.

QR Decomposition II

This suggests that Q has to be orthogonal, because this garantuees

$$||x||_2 = ||Qx||_2 \quad \text{and} \quad Q^{-1} = Q^T$$

Note that R cannot be not purely upper triangular due to dimension constrains, but if we aim for a $n \times p$ matrix that is $p \times p$ upper triangular, augmented by 0, we get

$$X = QR = (Q_1 Q_2) \begin{pmatrix} \tilde{R} \\ 0 \end{pmatrix} = Q_1 \tilde{R}$$

We can solve the LLS problem then via back substitution in

$$\tilde{R}\beta = Q_1^T y$$

As it happens, this decomposition can be arranged in several ways.

Condition of the LLS problem I

Regardless of the precise algorithm, we will not get the exact solution β_0 of the problem, but a numerical solution β^*. The β^* is the exact solution of a LLS with a perturbed matrix $X + E$ and a perturbed right hand side $y + \delta$:

$$(X + E)x^* = (y + \delta)$$

In the same manner we can define the exact residuals of the original system and the perturbed system:

$$r_0 = X\beta_0 - y \quad \text{and} \quad r^* = (X + E)\beta^* - (y + \delta)$$

Condition of the LLS problem II

The the following relationships between the exact and numeric quantities can be made precise for Euclidean norm and condition number (see `[Gray]`):

$$\frac{||\beta_0 - \beta^*||}{||\beta_0||} \leq O\left(\kappa_2(X)^2 \epsilon\right)$$

and

$$\frac{||r_0 - r^*||}{||(X + E)\beta^*||} \leq O\left(\kappa_2(X)\epsilon\right)$$

where ϵ is an upper limit for the relative magnitude of the perturbation:

$$\epsilon = \max\left(\frac{||E||}{||X||}, \frac{||\delta||}{||y||}\right)$$

QR vs. normal equations I

The *QR* decomposition can be implemented so that the above relationships hold with

$$\epsilon \approx 6np^{3/2}\epsilon_M$$

When solving the normal equations, errors will be produced both during matrix multiplication and equation solving, both in the order of

$$np\kappa_2(X^T X)\epsilon_M = np\kappa_2(X)^2\epsilon_M$$

This means that while the bound on the relative error of the solution is not necessarily worse, we have a much weaker bound on the residual of the system.

QR vs. normal equations II

For badly conditioned X the matrix $X^T X$ may well be numerically singular for the purpose of the *LU*-decomposition.

Example: the Longley data at
`http://lib.stat.cmu.edu/datasets/longley`
a simple data set with 16 cases and six independent variables.

Householder Transformations

Given any vector $v \in \mathcal{R}^n$, we can define the corresponding Householder transformation

$$H(v) = I_n - 2\frac{vv^T}{v^Tv}$$

As can be easily shown, these matrices are

- symmetric,
- orthogonal,
- independent of the scale of v (so we can assume $||v|| = 1$).

Furthermore, given any vector x, we can define a corresponding Householder transform that maps x into a multiple of the first elementary base vector $e_1 = c(1, 0, \ldots, 0) \in \mathcal{R}^n$. This is done for

$$v(x) = x \pm ||x||e_1$$

Building the *QR* from Householder transforms

Step1: take x_1 as the first column of X, $x_1 = X_{-1}$. Compute H_1 so that $H_1x_1 = \pm||x_1||e_1$. Then

$$Q = H_1 \quad \text{and} \quad R = H_1X$$

Step 2: take x_2 as the second column of X, starting at row 2: $x_2 = X_{2:n,1}$. Compute H_2 so that $H_2x_2 = \pm||x_2||e_2 \in \mathcal{R}^{n-1}$. Then we set

$$\tilde{H}_2 = \begin{pmatrix} 1 & 0 \\ 0 & H_2 \end{pmatrix}$$

and

$$Q = \tilde{H}_2Q \quad \text{and} \quad R = \tilde{H}_2R$$

and so on until we have Q and R of the desired shape after p steps.

Numerical Aspects

- The *QR* decomposition is significantly more stable than solving the normal equations.
- In borderline situations, column pivoting can boost the stability of the *QR* additionally: at each step, the column whose not-yet-eliminated lower part has the largest Euclidean distance is swapped to the front and use to define the Householder transformation.
- The sign of the Householder transform can be chosen to avoid subtractions.
- Orthogonal matrices always have Euclidean norm and condition 1 and are therefore fairly safe.
- The *QR* decomposition is also a stable way of solving LES.

Practical aspects

Operations & memory count:

- decomposition: $O(np^2)$ operations
- back substitution: $O(np)$ operations
- storage: when overwriting X, only p additional numbers need to be stored

The trick is that Q is not explicitly computed during the decomposition: instead, we keep the vectors $v_1, \ldots v_p$ that define the individual Householder tranformations – except for the first element, these can be stored below the diagonal. Applying the Housholder transformations to any rhs can be done in $O(pn)$ – same as a full matrix multiplication Q_1^Ty.