**Proceedings of the Third International DERIVE/TI-92 Conference**

# A Master Class in Advanced Programming with Derive For Windows™:

***The implementation of the QR algorithm for finding eigenvalues of large matrices and the solutions of high order polynomials.***

**Terence Etchells**
**School of Computing and Mathematical Sciences**
**Liverpool John Moores University, UK L3 3AF**
t.a.etchells@livjm.ac.uk
http://www.cms.livjm.ac.uk/etchells

## Abstract

In this paper we implement the QR algorithm for finding eigenvalues of large square matrices in the Derive for Windows (DFW) programming language. There are two major reasons for this: there is presently no implementation of this method in the DFW program or in the accompanying utility files; the implementation has required the development of DFW programming strategies that may help others achieve their particular programming goals. As an appendix to this work, the QR algorithm is used to find all the solutions to high order polynomials with real coefficients.

**Key Words**: QR algorithm, programming in Derive, eigenvalues, solving polynomials, stopping rules.

## Introduction

The prime motivation for this paper is the description of some programming strategies that have been developed in order to develop the QR algorithm in the Derive for Windows (DFW) programming language. DFW has within it a high level programming language that has many useful and computationally efficient tools that are available due its ability to algebraically manipulate objects and expressions. Whilst there are many advantages in solving mathematical programming problems in DFW, due to the mathematical tools already available, there are some apparent limitations in its ability to perform certain tasks. Some of these limitations and solutions are addressed in this paper. This paper may well have a limited shelf life as further versions of DFW may well address the difficulties and short comings presented here.

## The QR Algorithm

The QR algorithm is an iterative method that transforms a matrix $M$ into a diagonal matrix (or quasi-diagonal matrix) that has the same eigenvalues as $M$. As the diagonal elements of a diagonal matrix are its eigenvalues, we can find all the eigenvalues of any matrix $M$ (as long as we can get the method to converge).

A full and readable account of the method can be found in [1] and a rigorous treatment is in [5] but the bare bones of method are described here.

Take any matrix *M* and convert it into an upper Hessenberg Matrix with the same eigenvalues of *M*. An upper Hessenberg matrix is of the form

$$\begin{pmatrix} d_1 & a_{1,2} & \cdots & a_{1,n-1} & a_{1,n} \\ \delta_1 & d_2 & a_{2,3} & \cdots & a_{2,n} \\ 0 & \delta_2 & \ddots & \ddots & \vdots \\ \vdots & 0 & \ddots & d_{n-1} & a_{n,n-1} \\ 0 & \cdots & 0 & \delta_{n-1} & d_n \end{pmatrix}$$

i.e. all the elements below the sub diagonal ($\delta_1, \delta_2, \cdots, \delta_{n-1}$) are 0. This transformation is achieved through repeated Householder transformations which in effect annihilate (make zero) all the elements of the matrix *M* below the sub diagonal, whilst preserving the eigenvalues of *M*. This procedure is straightforward to program in DFW and a description of the method can be found in [1] section 6-7 and the DFW code is found below.

Once in upper Hessenberg form, we now wish to reduce the sub diagonal elements to zero. To do this we employ the technique of QR factorisation, i.e. we express the matrix as a product of an orthogonal matrix Q and an upper Right triangular matrix R.

The QR method for computing eigenvalues (Rutishauser [2], Francis [3] and Kublanovskaya [4]) employs the iterative procedure

$$M_n = Q_n R_n \text{ and } M_{n+1} = R_n Q_n .$$

If all the eigenvalues of *M* are real then this iterative procedure converges to a triangular matrix with the same eigenvalues as *M*, more typically some of the subdiagonal elements converge to some number other that 0 and hence a "quasi-triangular" matrix ensues. The quasi-triangular matrix implies that there are some complex eigenvalues (this is dealt with later).

Even with an upper Hessenberg transformation, the convergence of this method is quite slow and the computation times can be very high for large matrices. Consequently a shift technique (developed by Francis [3]) is employed to rapidly increase convergence. This technique involves the diagonal elements of the matrix *M* being shifted by the eigenvalue of the bottom right 2 x 2 submatrix $\begin{pmatrix} a_{n-1,n-1} & a_{n-1,n} \\ a_{n,n-1} & a_{n,n} \end{pmatrix}$ of *M* which is closest to the value $a_{n,n}$. Once the eigenvalues have been found these "shifts" are reversed.

The algorithm that we employ in this paper is described here for a 4x4 matrix

1. Convert the matrix to upper Hessenberg

$$\begin{pmatrix} \oplus & \oplus & \oplus & \oplus \\ * & \oplus & \oplus & \oplus \\ 0 & * & \oplus & \oplus \\ 0 & 0 & * & \oplus \end{pmatrix}$$

here the * are the sub diagonal elements.

2. Shift the diagonals of the matrix by the eigenvalue of the sub matrix nearest the bottom right diagonal element.

   Now apply a special QR factorisation method called a Given's rotation [6] which annihilates each * and then post multiplies by the inverse (in this case also the transpose) of the annihilation matrix. The process of shifting and applying Given's rotations down the subdiagonal is repeated until the bottom right subdiagonal element is very small or it converges to a number other than 0.

   Eventually, what is left is a matrix of the form

$$\begin{pmatrix} \oplus & \oplus & \oplus & \oplus \\ * & \oplus & \oplus & \oplus \\ 0 & * & \oplus & \oplus \\ 0 & 0 & \varepsilon & \lambda \end{pmatrix},$$

   if ε is very small then λ + *total shift* is an eigenvalue of the matrix and the last row and column are deleted (deflated) and the process is repeated on the smaller 3x3 matrix.

   If ε converges to a number other than 0 then the eigenvalues of the bottom 2x2 sub matrix + *total shift* are eigenvalues of the matrix *M* and the last two rows and columns are deleted (deflated) and the process is repeated on the smaller 2x2 matrix

## The Problems of implementing the Algorithm

The problem of implementing this method is that it is iterative and hence we want to output results once some convergence has been achieved. So we need some method of stopping a sub process when convergence has been achieved, **without** terminating the global process. This is not straightforward in DFW as it acts as a "Turing Machine" and only outputs results once **all** computations are finished. That is intermediate results cannot be output or the user store values in variables during the computation for retrospectively use in later computations. As a consequence one must be very mindful of which variables will be needed at later stages of the computation and store them as elements of a vector which are then passed onto the next set of calculations.

In addition the programmer must also work efficiently, in that no calculation should be needlessly repeated. A simple example of needless repetition is

FEE(A) which multiplies the dimension of a matrix *A by* 1 less that the dimension of the matrix *A*:

```
FEE(A):=DIMENSION(A)*(DIMENSION(A)-1),
```

The inefficiency lies in the fact that the dimension of the matrix is calculated twice where we only need to calculate it once. The use of an auxiliary functions reduce the need for multiple calculations:

```
FEE_AUX(N):=N(N-1)
FEE(A):=FEE_AUX(DIMENSION(A))
```

Although an artificial example, this kind of use of auxiliary functions does drastically reduce computation times.

Stopping a convergent sub process can be achieved using recursively defined function via the IF command. For example imagine that we wish to stop an iterative process when the difference between the iterates is 0.1.

If the iterative process is $u_{n+1} = \cos(u_n)$ and we start at $u_0 = 1$, the series of iterates are:

```
ITERATES(COS(x), x, 1, 7)
```

[1, 0.5403023058, 0.8575532158, 0.6542897904, 0.7934803587, 0.7013687736, 0.7639596828, 0.7221024250]

So the difference between the iterates is less than 0.1 when the iterative process reaches 0.7013687736.

A strategy is:

Stop(s)
Iterate once start at s produce vector [s,t]
Is abs(t-s)<0.1 ? Yes -> output t No -> stop(t)

In DFW this is coded as

```
STOP_AUX(uuu, vect, xxx) :=
    IF(ABS(vect™2 - vect™1) < 10^(-1), vect™2,
        STOP_AUX(uuu, ITERATES(uuu, xxx, vect™2, 1), xxx))
```

Where uuu is the iterative scheme, vect is a 2 element vector of the last two iterates, xxx is the variable of the iterative scheme.

```
STOP(uuu, xxx, aaa) := STOP_AUX(uuu, ITERATES(uuu, xxx, aaa, 1), xxx)
```

The STOP()function passes the first two iterates into the STOP_AUX() recursive function and it keeps going until the condition

```
ABS(vect™2 - vect™1) < 10^(-1)
```

is achieved. Notice that the key aspect to this function is the bold typed **vect™2** in the STOP_AUX() definition above, this ensures that if the condition is not met the process starts again with the a new starting value, that being last value calculated. Approximating

```
STOP(COS(x), x, 1)
```

gives 0.7013687736

This technique will prove invaluable in the programming of the QR algorithm.

## The Shifted QR Algorithm Implemented

### Hessenberg Matrix

First we must transform our matrix into a Hessenberg Matrix. Our first step is to construct a Householder matrix. A Householder matrix $\mathbf{H}$ has the property that $\mathbf{H} = \mathbf{H}^{-1}$ and $\mathbf{H} = \mathbf{I} - 2\mathbf{w}\mathbf{w}^T$ for some column vector $\mathbf{w}$. By finding a special value for $\mathbf{w}$, we can build a Householder matrix that annihilates the last $n-k$ elements of a vector column $\mathbf{x}$ (size *n*) and leaves the first $k-1$ elements alone. The value for $\mathbf{w}$ that will do this is

$$\mathbf{w} = \frac{1}{\sqrt{2s\left(s+|x_k|\right)}} \begin{pmatrix} 0 \\ \vdots \\ 0 \\ x_k + \text{sign}(x_k)s \\ x_{k+1} \\ \vdots \\ x_n \end{pmatrix} \qquad (**)$$

where $s = \sqrt{x_k^2 + x_{k+1}^2 + \cdots + x_n^2}$ and $x_k$ is the $k^{th}$ element of $\mathbf{x}$.

It can be proven that the eigenvalues of the matrix $\mathbf{M}$ are equal to the eigenvalues of the matrix $\mathbf{A}^{-1}\mathbf{M}\mathbf{A}$. So in the case of a Householder matrix $\mathbf{HMH}$ has the same eigenvalues as $\mathbf{M}$. So we can annihilate the every element below each subdiagonal of each column of any matrix. The DFW code is

```
S__1(x_, k_, n_) := ¦VECTOR(x_ , t_, k_, n_)¦
                    ¦       t_              ¦
```

This function evaluates $s = \sqrt{x_k^2 + x_{k+1}^2 + \cdots + x_n^2}$ for a given column *k*

```
HH_1(x_, k_, s_, n_) :=
1/‹(2·s_·(s_ + ABS(x_™k_)))·
APPEND([VECTOR(0,p_, 1,k_ - 1,[x_™k_ + SIGN(x_™k_)·s_],
      VECTOR(x_™p_, p_, k_ + 1, n_)])
```

This function evaluates the expression (**) for a particular column *k*.

```
HH_2(i__, w__) := i__ - 2·w__`·w__
```

The general form of the Householder Matrix, notice that I_ is used for the identity matrix. This is to ensure that the identity matrix is only calculated once, inefficient programming can lead to this being calculated many times needlessly.

```
HH_3(x_, k_, i__, n_) :=
                HH_2(i__, [HH_1(x_, k_, S__1(x_, k_, n_), n_)])
```

Calculates the Householder matrix for the $K^{th}$ column.

```
HH_4(hh, matrix) := hh·matrix·hh
```

Auxiliary function to avoid over calculation

```
HH_5(matrix, matrixt, k_, i__, n_) :=
        HH_4(HH_3(matrixt™(k_ - 1), k_, i__, n_), matrix)

HH_6(matrix, c_, i__, n_) := HH_5(matrix, matrix`, c_ + 1, i__, n_)

HH_7(init_matrix, i__, n_) :=
(ITERATE([c_ + 1, HH_6(matrix, c_, i__, n_)], [c_, matrix], [1, init_matrix], n_ - 2))™2
```

The ITERATE command forces the Householder matrix to act on each column in turn preserving the Householder transformations as it progresses.

```
HESS_1(mat, n_) := HH_7(mat, IDENTITY_MATRIX(n_), n_)

HESS(mat) := HESS_1(mat, DIMENSION(mat))
```

The final HESS(mat) function to calculate the upper Hessenberg Matrix.

Example

```
   „ 450   75   -525   150 †   „  450   225   225   450 †
   ¦                   ¦  ¦                 ¦
   ¦ 75   253   380   -79 ¦  ¦ -225   225   450   225 ¦
HESS ¦                   ¦ = ¦                 ¦
   ¦ 150    5    325  -215 ¦  ¦   0   -225   225  -225 ¦
   ¦                   ¦  ¦                 ¦
   … 150  -604   160   322 ‡   …   0     0   -450   450 ‡
```

You will notice that the Trace of both these matrices is identical.

## Given's Rotation

We now apply the QR algorithm with Given's Rotations. In this implementation we use the form

$$G_{i,i+1} = \begin{pmatrix} 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ \vdots & \ddots & 0 & 0 & 0 & \cdots & 0 \\ 0 & \cdots & c & -s & 0 & \cdots & \vdots \\ \vdots & \vdots & s & c & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & \ddots & 0 & 0 \\ 0 & 0 & \cdots & 0 & 0 & 1 & 0 \\ 0 & 0 & \cdots & 0 & 0 & 0 & 1 \end{pmatrix} \qquad (***)$$

where $c^2 + s^2 = 1$ and if we are annihilating the subdiagonal in column c_, then

$$s = \frac{a_{c\_+1,c\_}}{\sqrt{a_{c\_,c\_}{}^2 + a_{c\_+1,c\_}{}^2}} \text{ and } c = \frac{a_{c\_,c\_}}{\sqrt{a_{c\_,c\_}{}^2 + a_{c\_+1,c\_}{}^2}}$$

```
REPLACE_MAT_ELEMENT(aa, matr, ii, jj) := REPLACE_ELEMENT(REPLACE_ELEMENT(aa,
matr™ii, jj), matr, ii)
```

A function which replaces the element in the row ii and column jj in matrix matr with aa.

```
GIVEN_1(c__, s__, matr, c_) :=
REPLACE_MAT_ELEMENT(c__, REPLACE_MAT_ELEMENT(s__, REPLACE_MAT_ELEMENT(-
s__, REPLACE_MAT_ELEMENT(c__, matr, c_, c_), c_, c_ + 1), c_ + 1, c_), c_ + 1, c_ + 1)
```

```
GIVEN_2(i_, c_, a__, b__, c__) := GIVEN_1(a__/c__, - b__/c__, i_, c_)
```

```
GIVEN(matr_, i_, c_) :=
GIVEN_2(i_, c_, matr_™c_™c_, matr_™(c_ + 1)™c_,
                ‹(matr_™c_™c_^2 + matr_™(c_ + 1)™c_^2))
```

These functions efficiently produces the matrix described in (***).

```
QR_1(giv_temp, matrix_, c_) :=
                        [giv_temp™c_, giv_temp™(c_ + 1)]·matrix_
```

```
QR_2(mat_temp, matrix_, c_) := REPLACE_ELEMENT(mat_temp™2,
REPLACE_ELEMENT(mat_temp™1, matrix_, c_), c_ + 1)
```

Due to the nature of multiplying by these Given's matrices, there is a lot of multiplying by 0's and 1's in all but 2 of the rows, So these two functions just multiply by the active rows and then insert into the given identity matrix.

```
QR_3(matrix_, giv_temp, i_, c_) :=
    QR_2(QR_1(giv_temp, matrix_, c_), matrix_, c_)·giv_temp`
```

The giv_temp` at the end is the post mutliplication to preserve the eigenvalues of the original matrix.

```
QR_4(matrix_, i_, c_) := QR_3(matrix_, GIVEN(matrix_, i_,c_), i_, c_)
```

QR_4() will annihilate the sub diagonal in column c_ and then post multiply by the annihilating Given's rotation matrix transposed to preserve eigenvalues. This action takes the subdiagonal in column c_ closer to 0 or some convergent value.

```
QR_5(matrix_, i_, n_) := (ITERATE([c_ + 1, QR_4(j_, i_, c_)], [c_, j_], [1, matrix_], n_ - 1))^T^M2
```

The iterate here forces the QR_4() function to act on each column from 1 to $n-1$ where $n$ is the dimension of the matrix. Notice how we still have not used the IDENTITY_MATRIX(n) or DIMENSION(M) yet as this would cause unnecessary duplication of work .

Example

```
   ⎡ 450   225   225   450 ⎤                    ,
   ⎢                       ⎥
   ⎢-225   225   450   225 ⎥                    ,
QR_5⎢                       ⎥, IDENTITY_MATRIX(4), 4⎥
   ⎢  0   -225   225  -225 ⎥                    ⎥
   ⎢                       ⎥                    ⎥
   ⎣  0     0   -450   450 ⎦                    f

⎡      405          252.561      -68.7386  349.052 ⎤
⎢                                                  ⎥
⎢   -168.374        109.285      -7.87335  618.647 ⎥
⎢                                                  ⎥
⎢          -12                                     ⎥
⎢ - 6.46540·10       -393.667      565.714  168.374 ⎥
⎢                                                  ⎥
⎢          -12           -12                       ⎥
⎣… - 6.04782·10    - 7.44855·10    -252.561    270  ⎦
```

Notice that rounding error is creeping in and that two of the sub-diagonals have got closer to 0.

## The Francis Shift

```
FRANCIS_SHIFT_AUX_1(matr_1, n_) :=
RE(RHS(EIGENVALUES([[matr_1^T^M(n_ - 1)^T^M(n_ - 1), matr_1^T^M(n_ - 1)^T^Mn_],
                    [matr_1^T^Mn_^T^M(n_ - 1), matr_1^T^Mn_^T^Mn_]])))
```

This function calculates the eigenvalues of the bottom right sub matrix of the matrix matr_1 of dimension n_.

```
FRANCIS_SHIFT_AUX_2A(eig, ann) :=
IF(eig™1 = 0 •  eig™2 = 0, -1, IF(ABS(eig™1 - ann) < ABS(eig™2 - ann),
                                  eig™1, eig™2, eig™1))
```

These functions find the eigenvalue nearest the bottom right diagonal, also if the real par of the two eigenvalues are 0, this will result in a shift of 0 which is useless. If this should happen we shift by an arbitrary –1 to help the Francis shifting to get started.

```
FRANCIS_SHIFT_AUX_2(eig, matr_1, n_) :=
FRANCIS_SHIFT_AUX_2A(eig, matr_1™n_™n_, n_)

FRANCIS_SHIFT_AUX_3(matr_1, sigmak, i_) := matr_1 - sigmak·i_
```

sigmak is the calculated Francis shift

```
FRANCIS_SHIFT_AUX_4(matr_1, i_, n_) :=
  FRANCIS_SHIFT_AUX_3(matr_1,
    FRANCIS_SHIFT_AUX_2(FRANCIS_SHIFT_AUX_1(matr_1, n_), matr_1, n_,
                                                                    i_)
```

More passing into auxiliary functions to avoid repetitive calculations.

## The Shifted QR Algorithm

We apply a Francis shift to our Hessenberg matrix and apply Given's rotation annihilation and post multiplication down the sub diagonal elements. We inspect the bottom right sub diagonal, if it less that say 10^-8 then we can estimate one of the eigenvalues as the bottom right diagonal element. If not we need to apply the whole shifted process again (but we need to remember the previous shift(s) so that they can all be reversed when convergence or closeness to zero has been achieved. If convergence of the bottom sub diagonal element to number other than zero has been achieved then we find the eigenvalues of the bottom right 2x2 sub matrix and reverse all the shifts.

Here's how we do this:

```
QR_SHIFTED_AUX_1(matr_1, total, sigmak, i_, n_) :=
[total + sigmak, QR_5(FRANCIS_SHIFT_AUX_3(matr_1, sigmak, i_),i_,n_)]
```

This produces a 2 element vector of the shift +total and the shifted matrix after the Given's process.

```
QR_SHIFTED_AUX_2(matr_1, total, i_, n_) :=
  QR_SHIFTED_AUX_1(matr_1, total,
    FRANCIS_SHIFT_AUX_2(FRANCIS_SHIFT_AUX_1(matr_1, n_), matr_1, n_),
                                        i_, n_)
```

A computation saving function, this saves the Francis shift being calculated twice.

```
QR_SHIFTED_AUX_3A(matr_1, n_, i_) :=
```

```
ITERATES(QR_SHIFTED_AUX_2(matr_2, total, i_, n_),
        [total, matr_2], [matr_1™1, matr_1™2], 1)
```

This function calculates two iterations of the Francis shift, with the accumulated shift in the first column of the ensuing matrix and the second column having the successive shifted and Given's matrices . Matr_1 must be of the form [0,Hessenberg]

Example

```
          • „  „ 450   225   225   450 ††                ,
          ¦¦  ¦                ¦¦              ¦
          ¦¦  ¦ -225   225   450   225 ¦¦             ¦
QR_SHIFTED_AUX_3A¦¦0, ¦                ¦¦, 4, IDENTITY_MATRIX(4)¦
          ¦¦  ¦   0   -225   225  -225 ¦¦             ¦
          ¦¦  ¦                ¦¦              ¦
          ¦¦  ¦                ¦¦              ¦
          • …   …   0     0   -450   450 ‡‡              f

„           „  450   225   225   450 †          †
¦           ¦                ¦               ¦
¦           ¦                ¦               ¦
¦           ¦ -225   225   450   225 ¦         ¦
¦  0        ¦                ¦               ¦
¦           ¦   0   -225   225  -225 ¦         ¦
¦           ¦                ¦               ¦
¦           ¦                ¦               ¦
¦           …   0     0   -450   450 ‡         ¦
¦                                    ¦
¦    „    406.159        252.439     -68.8425  349.321 † ¦
¦    ¦                               ¦¦
¦    ¦   -168.364        110.391     -7.25956  618.760 ¦¦
¦    ¦                               ¦¦
¦ -1 ¦          -11                     ¦¦
¦    ¦ - 3.21748·10      -393.193      567.074  167.864 ¦¦
¦    ¦                               ¦¦
¦    ¦          -11          -11         ¦¦
…    …  - 3.00681·10     1.13966·10     -252.671  270.374 ‡ ‡
```

In this example the nearest eigenvalue of the bottom right 2x2 sub matrix to the bottom right diagonal is 0 ,so the arbitrary shift of –1 was employed to offset this.

```
QR_SHIFTED_AUX_3B(matr_4, n_, i_) :=
  IF(0.9999 < ABS(matr_4™2™1/matr_4™1™1) < 1.0001 •
                  ABS(matr_4™2™2™n_™(n_ - 1)) < 10^(-8),
    matr_4™2,
    QR_SHIFTED_AUX_3B(QR_SHIFTED_AUX_3A(matr_4™2, n_, i_), n_, i_))
```

This tests either the convergence of the accumulated Francis shifts **or** that bottom right sub diagonal is less than 10^-8 (this can be changed to suite accuracy needs), if not carry on until it does (recursively), using the last accumulated shift and Given's matrix thus far.

```
REDUCE_MAT(matr_5, n_, sw_) :=
IF(sw_ = 0, DELETE_ELEMENT(DELETE_ELEMENT(matr_5, n_)`, n_)`,
DELETE_ELEMENT(DELETE_ELEMENT(DELETE_ELEMENT(DELETE_ELEMENT(matr_5, n_), n_ - 1)`, n_),
n_ - 1)`)
```

If the argument sw_ (switch) is 0, delete last row and last column else delete the last two rows and last two columns. This function **deflates** the matrix once an eigenvalue or a pair of eigenvalues have been found.

```
QR_SHIFTED_AUX_4A(tmat_1, totshift, redmat1, redmat2, id_1, id_2, n_) := IF(ABS(tmat_1™n_™(n_ - 1)) <
10^(-6),
[[totshift + tmat_1™n_™n_], redmat1 + totshift·id_1],
[[totshift, totshift] + RHS(EIGENVALUES([[tmat_1™(n_ - 1)™(n_ - 1), tmat_1™(n_ - 1)™n_],
[tmat_1™n_™(n_ - 1), tmat_1™n_™n_]])), redmat2 + totshift·id_2])
```

At this stage the we have a vector of two elements, an accumulated shift and a reduced matrix. If the bottom right sub diagonal of the reduced matrix is <10^-6 then add the total accumulated shift to the bottom right sub diagonal and then add the total shift back to the diagonals of the matrix and delete the last row and last column (deflate). If the bottom right sub diagonal of the reduced matrix is not <10^-6, then find the eigenvalues of the bottom right 2x2 sub matrix and add the total accumulated shift to each eigenvalue, add the accumulated shift to the diagonals of the reduced matrix and delete the last two rows and columns. Quite an important function.

```
QR_SHIFTED_AUX_4B(tmat_1, totshift, redmat1, redmat2, n_) :=
 QR_SHIFTED_AUX_4A(tmat_1, totshift, redmat1, redmat2,
  IDENTITY_MATRIX(DIMENSION(redmat1)),
  IDENTITY_MATRIX(DIMENSION(redmat2)), n_)

QR_SHIFTED_AUX_4C(tmat_1, totshift, n_) :=
 QR_SHIFTED_AUX_4B(tmat_1, totshift, REDUCE_MAT(tmat_1, n_, 0),
 REDUCE_MAT(tmat_1, n_, 1), n_)

QR_SHIFTED_AUX_4(matr_9, n_) :=
 QR_SHIFTED_AUX_4C(matr_9™2, matr_9™1, n_)

QR_SHIFTED_AUX_5(matr_1, n_, i_) :=
  QR_SHIFTED_AUX_4(QR_SHIFTED_AUX_3(matr_1, n_, i_), n_)


QR_SHIFTED_AUX_6(matr_1, n_) :=
  QR_SHIFTED_AUX_5(matr_1, n_, IDENTITY_MATRIX(n_))

QR_SHIFTED_AUX_7(matr_1) :=
  QR_SHIFTED_AUX_6(matr_1, DIMENSION(matr_1))
```

Computation saving auxiliary functions.

Example

```
            „  450   225   225   450 †
            ¦                      ¦
            ¦ -225   225   450   225 ¦
QR_SHIFTED_AUX_7 ¦                      ¦
            ¦   0   -225   225  -225 ¦
            ¦                      ¦
            …   0     0   -450   450 ‡

„                                  „ -77.9142  -337.743 ††
¦[570.844 + 78.0216·î, 570.844 - 78.0216·î], ¦                 ¦¦
```

```
…                                   …  477.503   286.225 ‡‡
```

Here the function has extracted two complex eigenvalues (as the bottom right sub diagonal has not reduced to less than 10^-8) and has deflated the matrix to a 2x2.

```
QR_SHIFTED_AUX_8(eigenv, temp_mat) :=
[APPEND(eigenv, temp_mat™1), temp_mat™2]
```

We want to save the found eigenvalues as we deflate the matrix further and further so we append each found eigenvalue or pair of eigenvalues to a vector called eigvenv, which starts at [].

```
QR_SHIFTED_AUX_9(mm_, eigenv) :=
     QR_SHIFTED_AUX_8(eigenv, QR_SHIFTED_AUX_7(mm_))

QR_SHIFTED_AUX_10(mm_) :=
IF(DIMENSION(mm_™2) = 0, APPEND(mm_™1),
  IF(DIMENSION(mm_™2) = 1, APPEND([mm_™1, mm_™2™1]),
     QR_SHIFTED_AUX_10(QR_SHIFTED_AUX_9(mm_™2, mm_™1))))
```

This recursively defined function continues extracting eigenvalues and appending them to the vector eigenv until the deflated matrix has null size or is of dimension 1. We then append the final eigenv to tidy up the elements of eigenv.

```
QR(mat) := QR_SHIFTED_AUX_10([[], HESS(mat)])
```

This final function in this section calculates the Hessenberg of the matrix and sets eigenv to [] and the passes into all the previously defined functions.

Example

```
  „ 450   75   -525   150 †
  ¦                      ¦
  ¦ 75    253   380   -79 ¦
QR¦                      ¦
  ¦ 150    5    325  -215 ¦
  ¦                      ¦
  … 150  -604   160   322 ‡
```

```
[570.844 + 78.0216·î, 570.844 - 78.0216·î, 104.155 + 357.944·î, 104.155 - 357.944·î]
```

Computation time 2.2s second 133Mhz Pentium 32 MB RAM Windows 95

A bigger example

```
  „ 2  4  1  4  7  7  4  7 †
  ¦                        ¦
  ¦ 7  4  6  3  6  4  7  0 ¦
  ¦                        ¦
  ¦ 7  5  1  0  5  1  5  7 ¦
```

```
  ¦              ¦
  ¦3 2 1 7 3 0 3 4¦
QR¦              ¦
  ¦5 1 3 2 3 4 6 6¦
  ¦              ¦
  ¦7 5 6 5 3 3 0 3¦
  ¦              ¦
  ¦5 3 2 2 5 2 4 0¦
  ¦              ¦
  …1 2 1 3 0 4 7 1‡
```

[1.28313 + 2.84106·î, 1.28313 - 2.84106·î, 5.34450, -3.73242 + 3.29570·î, -3.73242 - 3.29570·î, -2.12266 + 4.54225·î, -2.12266 - 4.54225·î, 28.7993]

Computation time 29.7 seconds 133Mhz Pentium 32 MB RAM Windows 95

## Solving real coefficient polynomials

As a corollary to this work, if we can find the matrix whose characteristic equation is the polynomial we wish to solve. Then we can use the Shifted QR method to find the eigenvalues of this **companion matrix** and hence the solutions to the polynomial.

A companion matrix to the polynomial $x^n + c_{n-1}x^{n-1} + \cdots + c_1 x + c_0$ is

$$\begin{pmatrix} 0 & 0 & 0 & 0 & -c_0 \\ 1 & 0 & 0 & \vdots & -c_1 \\ 0 & 1 & 0 & & \vdots \\ \vdots & 0 & 1 & 0 & -c_{n-2} \\ 0 & \cdots & 0 & 1 & -c_{n-1} \end{pmatrix}$$

The Derive functions that convert a polynomial into its companion matrix are

```
                  1     • d ,n
POLY_COEFF(u, x, n) := ————————·lim ¦——¦ u
                       n!  x˜0 • dxƒ
POLY_DEGREE_AUX(u, x, n) :=
IF(u = 0, n, POLY_DEGREE_AUX(DIF(u, x), x, n + 1),
          POLY_DEGREE_AUX(DIF(u, x), x, n + 1))


POLY_DEGREE(u, x) := POLY_DEGREE_AUX(u, x, -1)
```

These 3 functions written by SWH Inc.

```
POLY_COEFF_VECT(u_, x_, n_) :=
VECTOR(POLY_COEFF(u_, x_, p_), p_, 0, n_ - 1)
```

Produces a vector of the coefficients of the given polynomial equation.

```
COMPANION_MATRIX_AUX(u_, x_, n_) :=
APPEND(DELETE_ELEMENT(IDENTITY_MATRIX(n_)`, 1), - [POLY_COEFF_VECT(u_, x_,
n_)/POLY_COEFF(u_, x_, n_)])`
```

```
COMPANION_MATRIX(u_) := COMPANION_MATRIX_AUX(LHS(u_) - RHS(u_),
(VARIABLES(u_))™1, POLY_DEGREE(u_, (VARIABLES(u_))™1))
```

Finds the companion matrix of the polynomial equation u_.

Example

```
            7     3
COMPANION_MATRIX(x  - 3·x  + 3 = 0)

„ 0  0  0  0  0  0  -3 †
 ¦                     ¦
 ¦ 1  0  0  0  0  0   0 ¦
 ¦                     ¦
 ¦ 0  1  0  0  0  0   0 ¦
 ¦                     ¦
 ¦ 0  0  1  0  0  0   3 ¦
 ¦                     ¦
 ¦ 0  0  0  1  0  0   0 ¦
 ¦                     ¦
 ¦ 0  0  0  0  1  0   0 ¦
 ¦                     ¦
… 0  0  0  0  0  0  1   0 ‡
```

```
ROOTS_AUX(u_) := QR(COMPANION_MATRIX(u_))
```

Finds good approximations to the solutions using the eigenvalues of the companion matrix.

```
ROOTS_AUX_1(u_, x_, scheme, start) :=
VECTOR(ITERATE(scheme, x_, q_, 5), q_, start)

ROOTS_AUX_2(u_, x_) :=
ROOTS_AUX_1(u_, x_, x_ - u_/DIF(u_, x_), ROOTS_AUX(u_))

ROOTS_AUX_3(u_) := ROOTS_AUX_2(LHS(u_) - RHS(u_), (VARIABLES(u_)) )
                                                    1
ROOTS(u_) := ROOTS_AUX_2(LHS(u_) - RHS(u_), (VARIABLES(u_)) )
                                           1
```

Uses the Newton Raphson process to find the correct solutions to the accuracy set in the Algebra State Menu, using the eigenvalues as starting points.

Example

```
     7     3
ROOTS(x  - 3·x  + 3 = 0)

[-1.41867, -0.515085 + 0.789532·î, -0.515085 - 0.789532·î, 1.11066 + 0.203548·î, 1.11066 - 0.203548·î,
0.113754 + 1.36138·î, 0.113754 - 1.36138·î]
```

Computation time 12.0 seconds 133Mhz Pentium 32 MB RAM Windows 95

## References

[1] Hager, W,W. "Applied Numerical Linear Algebra". Prentice Hall International, (1998), ISBN 0-13-041369-0

[2] Rutishauser, solution of eigenvalue problems with the LR factorisation, Nat. Bur. Stand. App. Math. Ser. 49(1958), pp 47-81.

[3] Francis, J.G.F The QR transformation: A unitary analogue to the LR transformation, Computing J., 4(1961), Part I pp. 265-272 and Part II pp. 332-345.

[4] Kublanovskaya, V.N., On some algorithms for the solution of the complete eigenvalue problem Zh. Vycisl. i Mat. Fiz., 1(1961),pp. 555-570.

[5] Wilkinson, J.H., The Algebraic Eigenvalue Problem, Oxford University Press, London, 1965.

[6] Given, W. Computation of plane unitary rotations transforming a general matrix to triangular form, SIAM J. Appl. Math., 6(1958),pp 26-50.