INT. J. MATH. EDUC. SCI. TECHNOL., 1994, VOL. 25, NO. 4, 481–490

Using MATLAB as a programming language for numerical analysis

by JOHN H. MATHEWS
Department of Mathematics, California State University Fullerton, USA

and KURTIS D. FINK
Department of Mathematics,
Northwest Missouri State University, USA

(*Received 11 February 1993*)

Today's software offers more for numerical analysis than just programming. The software MATLAB can be used to do things the traditional way; writing loops; branching using logical decisions and invoking subroutines. Now a larger programming environment is available; graphics and built in subroutine libraries. These features are influencing the way numerical analysis is taught. MATLAB is based on lists and many algorithms can be streamlined by taking advantage of this structure. Graphical output for interpolation, curve fitting and the solution of differential equations is easily produced by manipulating these data structures. This article illustrates how MATLAB can be used in a numerical analysis course to enhance the teaching of: Newton's method, Gaussian elimination, Chebyshev approximation, least squares polynomials, error analysis for numerical differentiation, adaptive quadrature, Runge–Kutta methods, and the solution of Laplace's equation. Our students have enjoyed MATLAB, and had a better experience computing with it. They were able to explore more algorithms given in the textbook and use several state of the art algorithms that are built into MATLAB.

1. 2-D Newton–Raphson method

To solve the nonlinear system $0=f_1(x,y)$, $0=f_2(x,y)$, given one initial approximation (p_0, q_0) and using Newton–Raphson iteration. The power of Matlab enables one to easily define the vector function $\mathbf{F(X)}=[f_1(x,y),\ f_2(x,y)]$ and its Jacobian (which is a matrix function).

Example 1. Solve the nonlinear system $0=x^2-2x-y+0{\cdot}5$ and $0=x^2+4y^2-4$. Place the above mentioned functions in the M-files; F.m and J.m, respectively.

```
function Z = F(X)
x = X(1);  y = X(2);
Z = [x.^2 - 2.*x - y + 0.5 , x.^2 + 4.*y.^2 - 4]';
```

```
function W = J(X)
x = X(1);  y = X(2);
W = [(2.*x - 2)  (-1);
     (2.*x)      (8.*y)];
```

Create the following subroutine in the M-file new2dim.m.

```
function [P0,Y0,err,P] = new2dim(F,J,P0,delta,epsilon,max1)
P = P0;  Y0 = feval(F,P0);
for k=1:max1,
  dF = feval(J,P0);
  if det(dF) == 0, dP = [0 0]; else dP = (dF\Y0)'; end
  P1 = P0 - dP;  Y1 = feval(F,P1);
  err = norm(dP);  relerr = err/(norm(P1)+eps);
  P0 = P1;  Y0 = Y1;  P = [P;P1];
  if (err<delta)|(relerr<delta)|(abs(Y1)<epsilon), break, end
end
Y0 = Y0';
```

To solve the nonlinear system execute the statements:

```
P0 = [2.0  0.25];  max1  = 40;
delta = 1e-12;  epsilon = 1e-12;
[P0,F0,err,P] = new2dim('F','J',P0,delta,epsilon,max1);
'     p(k)                q(k)', P
'The solution is P = ', P0
'F(P) = ', F0
'The error estimate for P is ±',disp(err)
```

Obtain the following results: (The graphics, Figure 1, is optional.)

p(k)	q(k)
2.0000000000000	0.2500000000000
1.9062500000000	0.3125000000000
1.9006905430712	0.3112125468165
1.9006767264649	0.3112185654047
1.9006767263671	0.3112185654193

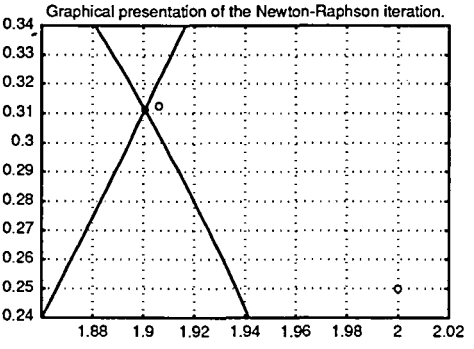


Figure 1.

```
The solution is:
P =          1.90067672636707          0.31121856541929

F(P) =       0.00000000000000          0.00000000000000

The error estimate for P is
             ± 9.896e-11               ± 9.896e-11
```

2. Gauss–Jordan method

To construct the solution to $AX = B$, by reducing the augmented matrix $[A, B]$ to diagonal form. Although MATLAB can solve linear systems, this subroutine is short and teaches the powerful row selection operation of MATLAB. A pivoting strategy can easily be added to make it more robust.

Example 2. Solve the linear system

$$\begin{pmatrix} 1 & 5 & 4 & -3 \\ 4 & 8 & 4 & 0 \\ 1 & 3 & 0 & -2 \\ 1 & 4 & 7 & 2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} -4 \\ 8 \\ -4 \\ 10 \end{pmatrix}.$$

Store the subroutine gaussj.m in an M-file.

```
function  X = gaussj(A,B)
[n n] = size(A);
A = [A';B']';  X = zeros(n,1);
for p = 1:n,
  for k = [1:p-1,p+1:n],
     if A(p,p)==0, break, end
     A(k,:) = A(k,:) - A(k,p)/A(p,p)*A(p,:);
  end
end
X = A(:,n+1)./diag(A);
```

To solve the linear system execute the statements:

```
A = [1   5   4  -3;
     4   8   4   0;
     1   3   0  -2;
     1   4   7   2];
B = [-4; 8; -4; 10];
X = gaussj(A,B);
'The solution to AX = B is displayed as X` =', X'
```

Obtain the results:

```
The solution to AX = B is X =
   3  -1  1  2
```

3. Least squares polynomial

To construct the least squares polynomial of degree M:

$$P_M(x) = c_1 x^M + c_2 x^{M-1} + \ldots + c_{M-2} x^3 + c_{M-1} x^2 + c_M x + c_{M+1}$$

that fits N data points.

Example 3. Find the least squares parabola that fits the points

$$(-3, 3), (-2, 2), (-1, 1{\cdot}5), (0, 1), (1, 1), (2, 1{\cdot}5), (3, 3), (4, 5).$$

The built-in MATLAB routine C=polyfit (X, Y, 2) will find the coefficients $C=[c_1, c_2, c_3]$, then polyval (C, x) evaluates $P_2(x)$. Type the commands:

```
x =  [-3  -2  -1   0  1   2   3  4];
y =  [ 3   2  1.5  1  1  1.5  3  5];
C = polyfit(X,Y,2);
F = polyval(C,X);  E = Y-F;
Xs =  -4:0.05:5;  Ys = polyval(C,Xs);
plot(X,Y,'o',Xs,Ys);
'y = P(x) = c(1)x^M + c(2)x^M-1 +...+ c(M)x + c(M+1)'
'The coefficients are stored in the array  C = '
disp(C')
'    x(k)      y(k)      P(x(k))   error'
disp([x;y;polyval(C,x);y-polyval(C,x)]')
```

Obtain the following results (Figure 2):

```
y = P(x) = c(1)x^M + c(2)x^M-1 +...+ c(M)x + c(M+1)
The coefficients are stored in the array  C =

    0.2500   -0.0238    0.8869

 x(k)    y(k)    P(x(k))     error
-3.00    3.00    3.2083    -0.2083
-2.00    2.00    1.9345     0.0655
-1.00    1.50    1.1607     0.3393
 0.00    1.00    0.8869     0.1131
 1.00    1.00    1.1131    -0.1131
 2.00    1.50    1.8393    -0.3393
 3.00    3.00    3.0655    -0.0655
 4.00    5.00    4.7917     0.2083
```

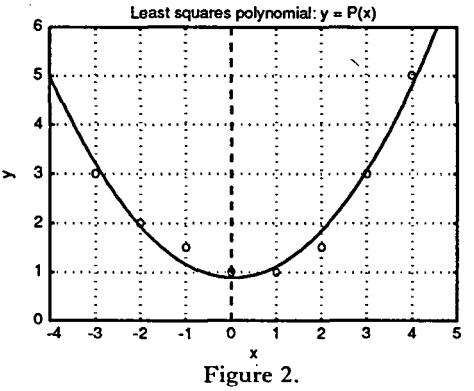


Figure 2.

4. Chebyshev approximation

To construct the Chebyshev interpolating polynomial $P_N(x)$ of degree N over the interval $[-1, 1]$, based on the nodes

$$x_k = \cos\left(\frac{(2k+1)\pi}{(2N+2)}\right), \text{ for } k = 0, 1, \ldots, N$$

MATLAB's polynomial fit routine is used to construct this polynomial. Since the abscissae must be supplied to the polyfit subroutine, this emphasizes the dependence on the Chebyshev nodes. A classroom exercise is to compare these results with Lagrange interpolation based on equally spaced nodes.

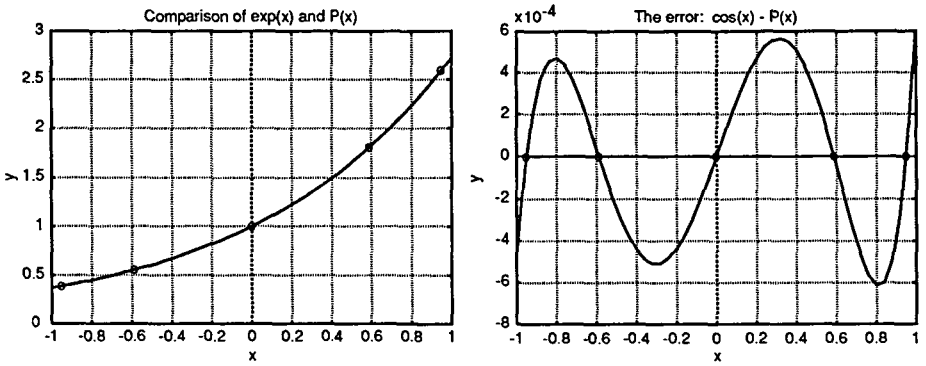
Example 4. Construct the Chebyshev approximation polynomial $P_4(x)$ for $f(x) = \exp(x)$ over $[-1, 1]$. Enter the commands:

```
n = 4;
K = 0:n;
X = cos((2*K+1)*pi/10);      % Generate the Chebyshev abscissas.
Y = exp(X);                  % Evaluate f(x) at each abscissa.
C = polyfit(X,Y,n);          % Compute the collocation polynomial.
X1 = -1:0.01:1;
Y1 = exp(X1);;
P = polyval(C,X1);
plot(X,Y,'or',X1,P,'-',X1,Y1,'--');
Z = zeros(1,n+1);
plot(X,Z,'o',X1,Y1-P,'-');
'The Chebyshev polynomial has been rearranged in ordinary form.';
'The coefficients of this ordinary polynomial are:', C
```

Obtain the following results (Figure 3 (*a*), (*b*))

```
The Chebyshev polynomial has been rearranged in ordinary form.
The coefficients of this ordinary polynomial are:

    0.0434    0.1773    0.4996    0.9973    1.0000
```



(*a*) (*b*)

Figure 3.

5. Error analysis for numerical differentiation

For the differentiation formula

$$f'(x)=\frac{f(x+h)-f(x-h)}{2h}+E(f,h)$$

the optimum step size is

$$h=\left(\frac{3\text{eps}}{m}\right)^{1/3}$$

and

$$|E(f,h)|\leqslant\frac{\text{eps}}{h}+\frac{mh^2}{6}$$

is the error bound, where eps is machine epsilon and $|f(3)x)|\leqslant m$. Investigation of the actual error and theoretical error bound is a useful exercise. Only a handful of MATLAB commands are necessary to carry out this investigation.

Example 5. Use $f(x)=\cos(x)$ and compute approximations for $f'(0{\cdot}8)$. Compare with the true value $f'(0{\cdot}8)=-\sin(0{\cdot}8)$. First, store the function:

```
function z = f(x)
z = cos(x);
```

To carry out the investigation type:

```
x = 0.8;  m = 1;
hopt = (3*eps/m)^(1/3); emin = eps/hopt + m*hopt^2/6;
a = hopt/10000;  b = 3.44*hopt; c = 0;  d = 5.25*emin;
H = a:(b-a)/150:b; B = eps./H + m*H.^2/6;
E = abs(- sin(x) - (f(x+H) - f(x-H))./(2*H));
axis([a b c d]);  plot(H,E,H,B);
```

Observe from the graph (Figure 4), that the optional step size is the minimum point of the smooth curve and the actual error incurred with step size h is the jagged curve. This shows how close the theoretical error bound matches the real situation.

6. Adaptive quadrature using Simpson's rule

To approximate the integral

$$\int_A^B f(x)\,dx\approx\sum_{k=1}^{M}\frac{h_k}{3}[f(x_{4k-4})+4f(x_{4k-3})+2f(x_{4k-2})+4f(x_{4k-1})+f(x_{4k})]$$

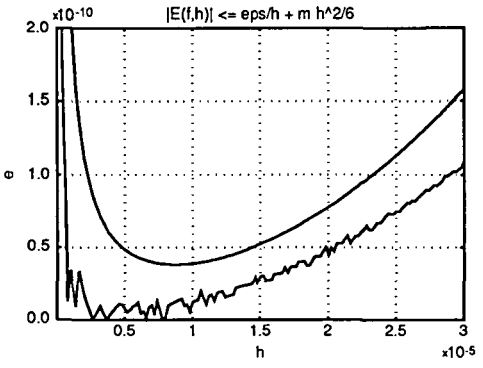


Figure 4.

Simpson's rule is applied with $4M$ subintervals $x_{4k-4+j}=x_{4k-4}+jh_k$ for each $k=1,2,\ldots,M$ and $j=1,\ldots 4$.

Example 6. Use adaptive quadrature to integrate $f(x)=13\,(x-x^2)\exp(-3x/2)$ over $[0,4]$.

```
function y = f(x)
y = 13.*(x - x.^2).*exp(-3.*x./2);
```

Store the first subroutine in the M-file; aqustep.m

```
function [quad,errb,cnt] = aqustep(f,a,c,b,fa,fc,fb,sr0,tol,lev)
if lev > max1,
  disp('Beware, recursion level exceeded!');
  quad = sr0;
else
  h   = (b - a)/2;
  c1  = (a + c)/2;                    % Determine midpoints c1, c2
  c2  = (c + b)/2;                    % for  [a c]  and  [c b].
  f1  = feval(f,c1);                  % Compute new function values
  f2  = feval(f,c2);                  % f(c1)  and  f(c2).
  sr1 = h*(fa + 4*f1 + fc)/6;         % Simpson's rule for [a,c].
  sr2 = h*(fc + 4*f2 + fb)/6;         % Simpson's rule for [c,b].
  quad = sr1 + sr2;
  errb = abs(sr1 + sr2 - (h*(fa + 4*fc + fb)/3))/10;
  cnt = 2;
  err = abs(quad - sr0)/10;
  % Recursively refine the subintervals if necessary.
  if err > tol,
     tol2 = tol/2;
     [sr1,errb1,cnt1] = ...
                    AQuStep(f,a,c1,c,fa,f1,fc,sr1,tol2,lev+1);
     [sr2,errb2,cnt2] = ...
                    AQuStep(f,c,c2,b,fc,f2,fb,sr2,tol2,lev+1);
     quad = sr1 + sr2;
     errb = errb1 + errb2;
     cnt = cnt + cnt1 + cnt2;
  end
end
```

Store the second subroutine in the M-file; aquad.m

```
function [quad,errb,cnt] = aquad(f,a,b,tol)
c   = (a + b)/2;              % Starting initialization
fa  = feval(f,a);             % which is required before
fb  = feval(f,b);             % recursively calling the
fc  = feval(f,c);             % subroutine  Aquadstep.
lev = 1;  sr0 = inf;  errb = 0;
% Now perform adaptive quadrature by recursive
% programming and using the subroutine  aqustep.
[quad,errb,cnt] = aqustep(f,a,c,b,fa,fc,fb,sr0,tol,lev);
cnt = cnt + 3;
```

To integrate $f(x)$ in Example 6 execute the statements:

```
a = 0;  b = 4;  toler = 0.00001;
[quad1,err,n] = Aquad('f',a,b,toler);
Mx2 = '   quadrature value   +- error bound', [quad1 err]
'The number of subintervals required was  m = ', (n-1)/4
Mx4 = 'The number of function evaluations  was  n = ', n
```

Obtain the results:

```
   quadrature value    +- error bound
  -1.54878823412532    0.00000296808616

The number of subintervals required was  m = 20

The number of function evaluations  was  n = 81
```

The program can be modified to obtain a graph of the subintervals used (Figure 5):

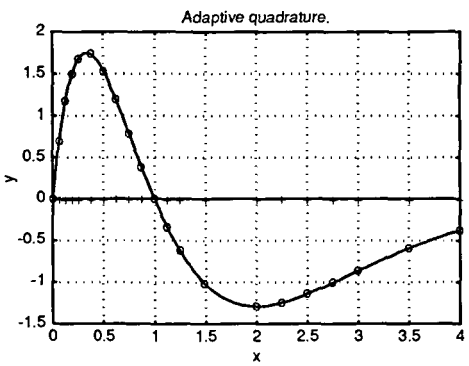


Figure 5.

7. Runge–Kutta method of order 4 for systems

To approximate the solution of the initial value problem $Z' = F(t, Z)$ with $Z(a) = Z_0$ over $[a, b]$. MATLAB vector functions enables us to use the Runge–Kutta formula that is taught in the one dimensional case.

Example 7. Solve

$$x' = x - xy - \frac{x^2}{10} \quad \text{and} \quad y' = xy - y - \frac{y^2}{20}$$

with $x(0)=2$ and $y(0)=1$ over the interval $[0, 15]$. First, place the above formulas in the vector function $F(t, Z)$:

```
function W = Fn(t,Z)
x = Z(1);  y = Z(2);
W = [(x - x*y - x^2/10), (x*y - y - y^2/20)];
```

Then store the following subroutine in the M-file; rks4.m

```
function [T,Z] = rks4(Fn,a,b,Za,m)
h =  (b - a)/m; T = zeros(1,m+1);
Z =  zeros(m+1,length(Za)); T(1) = a; Z(1,:) = Za;
for j=1:m,
  tj = T(j);  Zj = Z(j,:);
  K1 = h*feval(Fn,tj,Zj);
  K2 = h*feval(Fn,tj+h/2,Zj+K1/2);
  K3 = h*feval(Fn,tj+h/2,Zj+K2/2);
  K4 = h*feval(Fn,tj+h,Zj+K3);
  Z(j+1,:) = Zj + (K1 + 2*K2 + 2*K3 + K4)/6;
  T(j+1) = a + h*j;
end
```

To solve the system of differential equations type:

```
a  =  0;  b = 15;  m = 150;
Za =  [2  1];
[T,Z] = rks4('Fn',a,b,Za,m);
P =  [T;Z']';  X = Z(:,1);  Y = Z(:,2);
points = P(1:10:length(P),:);
plot(X,Y,'g');
'Runge-Kutta solution.'
't(k)     x(k)     y(k)'
disp(points)
```

Obtain the results (Figure 6):

```
Runge-Kutta solution.
```

t(k)	x(k)	y(k).
0.0	2.0000	1.0000
1.0	1.1116	1.6907
2.0	0.6005	1.2786
3.0	0.5621	0.7808
4.0	0.7586	0.5293
5.0	1.1486	0.4864
6.0	1.5550	0.6836
7.0	1.4591	1.1518
8.0	0.9643	1.3293
9.0	0.7257	1.0429
10.0	0.7547	0.7560
11.0	0.9582	0.6257
12.0	1.2397	0.6697
13.0	1.3669	0.8953
14.0	1.1673	1.1349
15.0	0.9122	1.1039

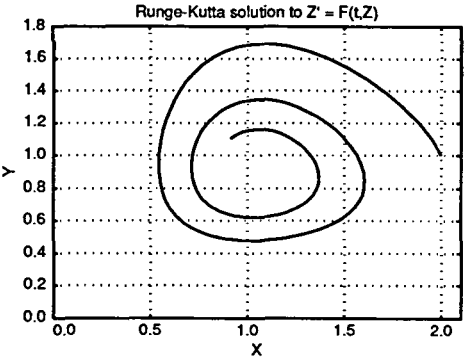


Figure 6.

8. Laplace's equation

To approximate the solution of $u_{xx}(x,y)+u_{yy}(x,y)=0$ over $R=\{(x,y): 0\leqslant x\leqslant a, 0\leqslant y\leqslant b\}$ with boundary conditions $u(x,0)=f_1(x)$, $u(x,b)=f_2(x)$ for $0\leqslant x\leqslant a$ and $u(0,y)=f_3(y)$, $u(a,y)=f_4(y)$ for $0\leqslant y\leqslant b$. Here the three-dimensional capabilities of MATLAB come into play. We no longer need to settle for a table of values for our numerical experiment, but can easily obtain a 3-D representation of the solution.

Example 8. Use finite difference method to solve $u_{xx}(x,y)+u_{yy}(x,y=0$ over $R=\{(x,y): 0\leqslant x\leqslant 4, 0\leqslant y\leqslant 4\}$ with $u(x,0)=f_1(x)=20$, $u(x,b)=f_2(x)=180$ for $0\leqslant x\leqslant 4$ and $u(0,y)=f_3(y)=80$, $u(a,y)=f_4(y)=0$ for $0\leqslant y\leqslant 4$. First store the functions:

```
function z = f1(x)
z = 20;
```

```
function z = f2(x)
     z = 180;
```

```
function z = f3(y)
z = 80;
```

```
function z = f4(y)
     z = 0;
```

Then store the following subroutine in the M-file; dirich.m

```
function U = dirich(f1,f2,f3,f4,a,b,h,tol,max1)
n =  fix(a/h)+1;  m = fix(b/h)+1;
ave = (a*(feval(f1,0)+feval(f2,0)) ...
     +  b*(feval(f3,0)+feval(f4,0)))/(2*a+2*b);
U = ave*ones(n,m);
for j=1:m,
  U(1,j) = feval(f3,h*(j-1));  U(n,j) = feval(f4,h*(j-1));
end
for i=1:n,
  U(i,1) = feval(f1,h*(i-1));  U(i,m) = feval(f2,h*(i-1));
end
U(1,1) = (U(1,2)+U(2,1))/2;     U(1,m) = (U(1,m-1)+U(2,m))/2;
U(n,1) = (U(n-1,1)+U(n,2))/2;  U(n,m) = (U(n-1,m)+U(n,m-1))/2;
w = 4/(2+sqrt(4-(cos(pi/(n-1))+cos(pi/(m-1)))^2));
err = 1;  cnt = 0;
while ((err>tol)&(cnt<=max1))
  err = 0;
  for j=2:(m-1),
     for i=2:(n-1),
        relx = ...
        w*(U(i,j+1)+U(i,j-1)+U(i+1,j)+ U(i-1,j)-4*U(i,j))/4;
        U(i,j) = U(i,j) + relx;
        if (err<=abs(relx)), err=abs(relx); end
     end
  end
  cnt = cnt+1;
end
```

To solve the problem PDE execute the statements:

```
a = 4.0;  b = 4.0;  h = 0.5;
tol = 0.001;  max1  = 25;
U = dirich('f1','f2','f3','f4',a,b,h,tol,max1);
'The solution to Laplace`s equation.'
W = rot90(U); disp(W)
mesh(U);
```

Obtain the following results (the graph of which is shown in Figure 7):

```
The solution to Laplace's equation.

130  180.0000  180.0000  180.0000  180.0000  180.0000  180.0000  180.0000   90
 80  125.8208  141.1717  145.4135  144.0043  137.4780  122.6423   88.6070    0
 80  102.1116  113.4527  116.4780  113.1256  103.2653   84.4842   51.7855    0
 80   89.1730   94.0492   93.9203   88.7548   77.9734   60.2438   34.0509    0
 80   80.5314   79.6509   76.3993   69.9998   59.6298   44.4665   24.1743    0
 80   73.3017   67.6235   62.0262   55.2154   46.0794   33.8182   18.1798    0
 80   65.0524   55.5153   48.8665   42.7563   35.6540   26.5471   14.7265    0
 80   51.3928   40.5192   35.1688   31.2895   27.2333   21.9899   14.1791    0
 50   20.0000   20.0000   20.0000   20.0000   20.0000   20.0000   20.0000   10
```

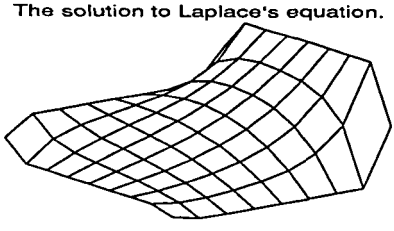


Figure 7.

9. Conclusions

MATLAB has widespread accessibility for educational instruction and is available in an inexpensive student version. Its structured programming style is easy for students to learn and resembles FORTRAN or Pascal. Graphical capabilities gives students visual understanding of their problems. This portrays an invaluable sense of realism and enjoyment in numerical mathematics. Students are empowered by this new tool of graphical presentation and can explore details that were previously only possible by looking at pictures in a textbook.

A complete set of algorithms are available which supplement the text 'Numerical Methods; for Mathematics, Science and Engineering'. Inquiries can be made directly to the author by surface mail or the E-mail address mathews@fullerton.edu.