

# **IMPLEMENTING LINEAR ALGEBRA RELATED ALGORITHMS ON THE TI-92+ CALCULATOR**

John Alexopoulos and Paul Abraham

ADDRESS: Department of Mathematics and Computer Science, Kent State University Stark  
Campus, Canton, OH 44720, USA.

E-MAIL: [jalexopoulos@stark.kent.edu](mailto:jalexopoulos@stark.kent.edu), [pabraham@stark.kent.edu](mailto:pabraham@stark.kent.edu)

ABSTRACT: Perhaps the most common use of the TI-92+ calculator is routine symbolic computation. The purpose of this article, however, is to demonstrate a less utilized feature of the TI-92+: its natural and powerful programming language. Indeed programming is essential to complete many common symbolic computations that go beyond the calculator's built-in commands. In this article we show how to implement several linear algebra related algorithms including the Gram-Schmidt process, Least Squares Approximations, Wronskians, Cholesky Decompositions and Generalized Linear Least Square Approximations with QR Decompositions. For the authors the necessity to augment TI-92+ built-in capabilities with user-defined code represents an exciting way of introducing programming to mathematically oriented students from the perspective of high-level symbolic computation and list manipulation.

KEYWORDS: Symbolic Programming, TI-92+ Calculator, Linear Algebra, Gram-Schmidt Process, Least Squares Approximation, Wronskian, Cholesky Decomposition, Generalized Linear Least Square Approximation, QR Decomposition.

## INTRODUCTION

Perhaps the most common use of the TI-92+ calculator is routine symbolic computation. Though even greater and faster symbolic capabilities exist with the use of software packages like *Mathematica* or *Maple*, the TI-92+ offers a low-cost and high-quality means of incorporating symbolic computation into the teaching and learning of mathematics without the need for a computer.

The purpose of this article, however, is to demonstrate a less utilized feature of the TI-92+: its natural and powerful programming language. Unlike software programs that utilize extensive libraries of functions, programming is essential to complete many common symbolic computations that go beyond the TI-92+ built-in capabilities. For example compare the problems of orthonormalizing a collection of vectors in  $\mathbb{R}^n$  versus orthonormalizing a collection of functions over an interval. Orthonormalizing vectors in  $\mathbb{R}^n$  on the TI-92+ is simply a matter of forming a matrix  $A$  whose columns are the given vectors and then invoking the QR decomposition program. One of the outputs of the QR decomposition program is a matrix  $R$  whose non-zero columns form an orthonormal basis for the column space of  $A$ . In fact, in the case of symbolic calculation,  $R$  is computed using the Gram-Schmidt process for vectors in  $\mathbb{R}^n$  (Texas Instruments Inc., 1998). On the other hand, there is no ready-made mechanism to orthonormalize a collection of functions on the TI-92+ which is easily rectified in Section 1 with a minor amount of programming.

Besides increasing the power of one's calculator or computer, there are substantial gains in learning possible with programming in a language that naturally reflects mathematics. In his article on mathematics and ISETL programming, Dubinsky (Dubinsky, 1995) shows how programming can be effectively incorporated into a theory of learning to enhance the learning of undergraduate mathematics. In brief Dubinsky argues that programming provides a learning mechanism through which abstract mathematical concepts can be seen as concrete

objects which act on and interact with each other. Dubinsky's approach relies heavily on the ISETL programming language in which finite sets, functions and binary operations are easily and naturally translated into ISETL code and are used to investigate and internalize algorithms in abstract algebra and calculus.

In this article, lists and matrices are the objects of primary interest suiting well the strengths of the TI-92+ language. In particular we present:

- Section 1: The Gram Schmidt Process and Least Squares Approximations
- Section 2: The Wronskian
- Section 3: The Cholesky Decomposition
- Section 4: The Generalized Linear Least Squares Approximation Problem with QR Decomposition

In most cases, the progression from algorithm to code strongly reinforces the natural mathematical language used to describe the algorithm. Hence the construction and implementation of the algorithm represents the internalization and generalization of the specific computations students do in class and homework, and therefore offer worthwhile exercises, projects or classroom discussion items.

Finally, we conclude this article with a brief discussion of why we believe programming on the TI-92+ may also be viewed as a valuable addition to the computer science training that mathematically-oriented students normally receive.

## **SECTION 1: THE GRAM-SCHMIDT PROCESS AND LEAST SQUARES APPROXIMATIONS**

Given a linearly independent collection  $V = \{v_1, \dots, v_n\}$  of square-integrable functions on some interval  $[a, b]$ , the Gram-Schmidt Orthonormalization Process can be used to obtain an

orthonormal basis for the span of  $V$ . Using standard notation (Larson and Edwards, 1996, 273-274) for square-integrable functions  $f$  and  $g$  on  $[a,b]$ ,

$$\langle f, g \rangle = \int_a^b f(t) \cdot \overline{g(t)} dt \text{ and } \|f\| = \sqrt{\langle f, f \rangle} = \sqrt{\int_a^b |f(t)|^2 dt},$$

the major programming steps of the Gram-Schmidt Process are as follows:

Step 1: Create a new list  $W$  that is the same size (dimension) as  $V$ .

Step 2: Set  $w_1 = \frac{v_1}{\|v_1\|}$ .

Step 3: For  $1 < i \leq n$ , set  $w_i = \frac{v_i - \sum_{j=1}^{i-1} \langle v_i, w_j \rangle \cdot w_j}{\|v_i - \sum_{j=1}^{i-1} \langle v_i, w_j \rangle \cdot w_j\|}$ .

The collection  $W = \{w_1, \dots, w_n\}$  is then an orthonormal basis for the span of  $V$ . The implementation of the Gram-Schmidt process requires only simple list handling which includes creating a list  $w$  whose dimension is determined at run time.

```
grsmdt(v,x,a,b)
Func
Local m,w,i,j
```

© Given a list  $v$  of linearly independent functions of variable  $x$  on an interval  $[a,b]$ , this function returns an orthonormal list of functions  $w$  whose span is identical to that of  $v$ .

```
dim(v)→m
newList(m)→w
For i,1,m
    v[i]-Σ(f(v[i]*conj(w[j]),x,a,b)*w[j],j,1,i-1)→w[i]
    w[i]/(√(f(w[i]*conj(w[i]),x,a,b)))→w[i]
EndFor
Return w
EndFunc
```

A sample run of this Gram-Schmidt function is given below with the functions 1,  $x$  and  $x^2$  on the interval  $[-1,1]$ . As you may notice, the function returns the first three normalized Legendre polynomials.

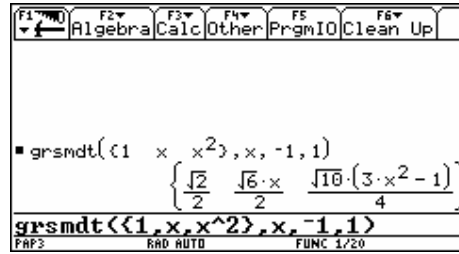


Figure 1. A Run of the Gram-Schmidt Function

The Gram-Schmidt process is commonly used to determine the least squares approximation for a given function (Larson and Edwards, 1996, 295-304). Recall the least squares approximation problem is as follows:

Given a square-integrable function  $f$  and a collection  $L = \{f_1, \dots, f_n\}$  of linearly independent, square-integrable functions on an interval  $[a, b]$ , find a function  $g$  in the span of  $L$  so that  $\|f - g\|^2 = \int_a^b |f(t) - g(t)|^2 dt$  is minimal.

We solve the problem as follows:

Step 1: Use the Gram-Schmidt process to create a list  $V = \{v_1, \dots, v_n\}$  which is an orthonormal basis for the span of  $L$ .

Step 2: Create the list  $fc$  of the Fourier coefficients of  $f$  relative to the basis  $V$ , where

$$fc_i = \langle f, v_i \rangle = \int_a^b f(t) \cdot \overline{v_i(t)} dt.$$

Step 3: Set  $g(t) = \sum_{i=1}^n fc_i \cdot v_i(t)$ .

The code required to accomplish least squares approximations and some sample runs follow. The powerful and convenient ability to apply an operator across a list, in particular to integrate a list of functions, is used in computing the Fourier coefficients in one step.

```
l sapx(f,l,x,a,b)
Func
Local v,fc
```

© This function computes the least squares approximation of a function  $f$  of variable  $x$  on an interval  $[a,b]$  with respect to the space spanned by a list  $l$  of linearly independent functions.

```
grsmdt(l,x,a,b)→v
∫(f*conj(v),x,a,b)→fc
Return expand(sum(fc*v),x)
EndFunc
```

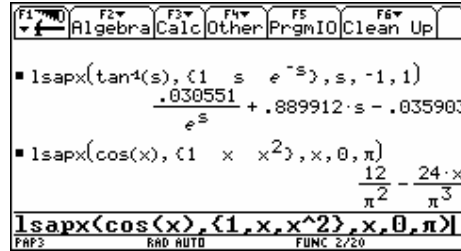


Figure 2. Sample Runs of the Least Squares Approximation Function

It is worth noting at this stage that the Gram-Schmidt and the Least Squares Approximation processes can be generalized to the space of square-integrable functions  $L^2([a,b], \mu)$  where  $\mu$  is the measure induced by a *weight function*  $\omega$  by setting

$$\langle f, g \rangle = \int_a^b \omega(t) \cdot f(t) \cdot \overline{g(t)} dt \text{ and } \|f\| = \sqrt{\langle f, f \rangle} = \sqrt{\int_a^b \omega(t) \cdot |f(t)|^2 dt}.$$

Recall that a non-negative measurable function  $\omega$  defined on  $[a,b]$  is called a *weight function*

if  $\int_a^b \omega(t) dt > 0$  and all moments  $\int_a^b t^k \omega(t) dt$  exist and are finite for all non-negative integers

$k$  (Stoer and Burlisch, 1993, 150-151). The code that implements these generalizations follows:

```
grsmdt(v,w,x,a,b)
Func
Local m,w,i,j

dim(v)→m
newList(m)→w
For i,1,m
  v[i]-Σ(∫(ω*v[i]*conj(w[j]),x,a,b)*w[j],j,1,i-1)→w[i]
  w[i]/(√(∫(ω*w[i]*conj(w[i]),x,a,b)))→w[i]
EndFor
Return w
EndFunc
```

```
lsapx(f,l,w,x,a,b)
```

```

Func
Local v,fc

grsmdt(1,w,x,a,b)→v
f(w*f*conj(v),x,a,b)→fc
Return expand(sum(fc*v),x)
EndFunc

```

## SECTION 2: THE WRONSKIAN

Recall the Wronskian *wronsk* of a collection of  $n$   $(n-1)$ -times-differentiable functions  $L = \{f_1, \dots, f_n\}$  of a single variable  $x$  on some interval  $(a, b)$ , is defined by:

$$wronsk(\{f_1, \dots, f_n\}, x) = \begin{vmatrix} f_1(x) & f_2(x) & \cdots & f_n(x) \\ f_1'(x) & f_2'(x) & \cdots & f_n'(x) \\ \vdots & \vdots & \ddots & \vdots \\ f_1^{(n-1)}(x) & f_2^{(n-1)}(x) & \cdots & f_n^{(n-1)}(x) \end{vmatrix}$$

The Wronskian has several uses in linear algebra and differential equations, most notably to establish whether or not a collection of functions is linearly independent (Larson and Edwards, 1996, p. 226). Our pseudocode for calculating the Wronskian is:

Step 1: Create an  $n$  by  $n$  matrix *mtx* to hold the rows on the Wronskian matrix.

Step 2: Fill in the first row of the Wronskian matrix with  $L$ .  $L$  must first be converted to a row matrix.

Step 3: Fill in the rest of Wronskian matrix row by row.

Step 4: Compute the determinant of the Wronskian matrix.

The code for the algorithm follows and again reflects the pure reality of calculation done by hand and provides a nice illustration of constructing a matrix row by row at run time.

```

wronsk(1,x)
Func
Local n,mtx,i

© This function computes the Wronskian of a list 1 of functions in
variable x.

dim(1)→n
newMat(n,n)→mtx
list→mat(1)→mtx[1]
For i,2,n
    d(mtx[i-1],x)→mtx[i]

```

```

EndFor
Return  det(mtx)
EndFunc

```

Sample calls of the Wronskian function are displayed next.

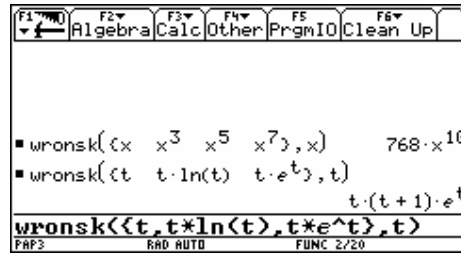


Figure 3. Sample Runs of the Wronskian Function

### SECTION 3: THE CHOLESKY DECOMPOSITION

Recall a matrix  $A$  is called *positive definite* if and only if it is Hermitian and  $x^T A x > 0$ , for all non-zero column vectors  $x$  of compatible dimension. If  $A$  is an  $n \times n$  positive definite matrix then  $A$  has a unique decomposition of the form  $L \cdot L^T$  for some lower triangular matrix  $L$ . This decomposition is called the Cholesky decomposition (Stoer and Burlisch, 1993, 180-183) of  $A$ . The Cholesky decomposition differs from the common LU decomposition in that  $U$  must equal  $L^T$ . Like LU decompositions, Cholesky decompositions are useful in solving systems of equations. The existence of Cholesky decompositions can be proved with induction.

For an  $n \times n$  matrix  $B$ , let  $B_{n-1}$  be the square matrix obtained by deleting the last row and column entries of  $B$ . With this notation in mind,  $L$  ( $= L_n$ ) is normally computed recursively, where the last row and column entries of  $L_n$  non-trivially depend on  $L_{n-1}$ ,  $A_{n-1}$  and the entries in the last row and column of  $A_n$ , until  $n=1$  is reached. Specifically the pseudocode for computing the Cholesky decomposition is as follows:



Step 1: Given an  $n$  by  $n$  matrix  $A$ , if  $n=1$  return the singleton matrix  $(\sqrt{a_{11}})$ . Else do the following steps.

Step 2: Noting that  $A_{n-1}$  is positive definite and  $A_{n-1} = L_{n-1} \cdot L_{n-1}^T$ , determine the Cholesky decomposition  $L_{n-1}$  of  $A_{n-1}$ . Once  $L_{n-1}$  is computed, Step 3 is based on the following observations, where  $b$  and  $c$  are  $(n-1)$ -dimensional column vectors and  $\alpha$  is a scalar.

- $A$  is of the form  $\begin{bmatrix} A_{n-1} & b \\ b^T & a_{nn} \end{bmatrix}$ .
- $L$  is of the form  $\begin{bmatrix} L_{n-1} & 0 \\ c^T & \alpha \end{bmatrix}$ .
- Since  $A = L \cdot L^T$ ,  $(L_{n-1} \cdot c)^T = b^T$  and  $c^T \cdot c + \alpha^2 = a_{nn}$ .

Step 3: Obtain  $c$  by solving the triangular system  $L_{n-1} \cdot c = b$  and then compute  $\alpha$ .

Step 4: Construct  $L_n$  by augmenting  $L_{n-1}$  appropriately.

Our Cholesky decomposition function and a sample run follow.

```
cholesky(a)
Func
Local  n,l,b,c,alpha

© This function computes the Cholesky decomposition of a positive
definite matrix a.

rowDim(a)->n
If n=1 Then
    Return  [[sqrt(a[1,1])]]
Else
    cholesky(subMat(a,1,1,n-1,n-1))->l
    subMat(a,1,n,n-1,n)->b

    © simlt solves a system using row-reduction

    simlt(l,b)->c
    sqrt(a[n,n]-(norm(c))^2)->alpha
    augment(c;[[alpha]])->c
    newMat(n-1,1)->b
    augment(l,b)->l
    augment(l;c^T)->l
    Return  l
EndIf
EndFunc
```

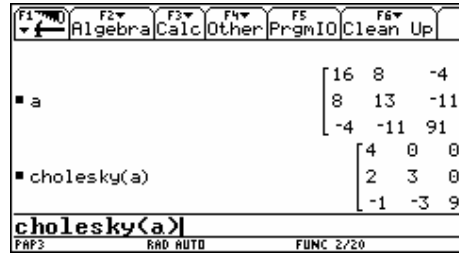


Figure 4. A Sample Run of the Cholesky Function

Even though the code for the Cholesky function naturally reflects the mathematics involved, it is nonetheless somewhat inefficient. The use of recursion, together with vector and matrix instructions may hide quite a bit of *unnecessary* looping. In addition to this, a more efficient algorithm should take advantage of the triangular nature of the system of equations involved into computing the last row of the Cholesky matrix. That is, explicit looping can be used to solve the system:

$$\left[ \begin{array}{ccc|c} l_{11} & \cdots & 0 & a_{1n} \\ \vdots & \ddots & \vdots & \vdots \\ l_{(n-1)1} & \cdots & l_{(n-1)(n-1)} & a_{(n-1)n} \end{array} \right]$$

by forward-substitution  $c_i = \frac{a_{in} - \sum_{j=1}^{i-1} l_{ij} \cdot c_j}{l_{ii}}$ , which unlike in Maple is not a built-in process

on the TI-92+. The more efficient code is next and runs significantly faster than the first Cholesky function, even for small matrices. The orders of `cholesky()` and `cholesk2()` are  $O(n^4)$  and  $O(n^3)$  respectively.

```
cholesk2(a)
Func
Local n,prev,l,i,j

rowDim(a)→n
If n=1 Then
  Return [[√(a[1,1])]]
Else
  newMat(n,n)→l
  cholesk2(subMat(a,1,1,n-1,n-1))→prev

  © explicit loops are used to solve [l|b] with back
  substitution. Subsequently calculate ||c|| and augment l.
```

```

For i,1,n-1
  prev[i,i]→l[i,i]
  a[i,n]→l[n,i]
  For j,1,i-1
    prev[i,j]→l[i,j]
    l[n,i]-prev[i,j]*l[n,j]→l[n,i]
  EndFor
  l[n,i]/(l[i,i])→l[n,i]
  l[n,n]+(l[n,i])^2→l[n,n]
EndFor
√(a[n,n]-l[n,n])→l[n,n]
Return l
EndIf
EndFunc

```

## SECTION 4: THE GENERALIZED LINEAR LEAST SQUARES APPROXIMATION PROBLEM WITH QR DECOMPOSITION

Even though the TI-92+ has several built-in regressions (e.g., polynomial up to degree 4, exponential, logistic, etc.), it does not offer a general solution to the linear least squares approximation problem:

Given a set of points  $\{(x_1, y_1), \dots, (x_n, y_n)\}$  and a collection of functions  $\{f_1, \dots, f_m\}$

find scalars  $a_1, \dots, a_m$  so that if  $g(x) = \sum_{i=1}^m a_i \cdot f_i(x)$  then the quantity

$\sum_{j=1}^n (y_j - g(x_j))^2$  is minimized.

Typically,  $a_1, \dots, a_m$  are found by solving the *normal equations*:

$$(M^T \cdot M) \cdot X = M^T \cdot Y, \text{ where } M = \begin{bmatrix} f_1(x_1) & \cdots & f_m(x_1) \\ \vdots & \ddots & \vdots \\ f_1(x_n) & \cdots & f_m(x_n) \end{bmatrix} \text{ and } Y = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}.$$

The normal equations can be solved by making use of standard techniques such as Gaussian elimination, LU-decomposition and Cholesky decomposition (Stoer and Burlisch, 1993, 207-209). There are drawbacks though such as numerical instability and even failure when the square matrix  $M^T \cdot M$  is singular or nearly singular. In the case where  $n > m$  which is typically expected, a better technique for determining  $a_1, \dots, a_m$  that utilizes the QR

decomposition of a matrix (Stoer and Burlisch, 1993, 209-210) can be used to circumvent the difficulties that arise with the normal equations approach. Further, the QR decomposition technique is a built-in program on the TI-92+.

Our solution to the Generalized Linear Least Squares Approximation Problem is based on an honor's project (Petrovic, 1999) supervised by the first author. The solution will need to be coded as a program since functions can not call programs (e.g. the QR decomposition program), receive functions as parameters or manipulate system graphing variables and settings. It is designed to accept multiple data sets simultaneously as a single matrix  $dmtx$  whose first column contains the  $x$ -values of the data while the  $k$  subsequent columns of  $dmtx$  contain corresponding  $y$ -values. The list of approximating functions must be stored as a row vector field under the external name  $f(x)$ . The program computes the approximating functions of the different data sets and stores them into graphing variables  $y_1, \dots, y_k$ . It plots the data sets using different plotting marks and then graphs the approximating functions.

Like the preceding examples, the translation of the mathematics to TI-92+ is natural and the major steps are listed below. On the other hand, the desire to produce graphs within the program poses technical issues. But even these are not too difficult to overcome and provide us with an opportunity to demonstrate some powerful evaluation and addressing features which can be utilized within programs.

The major steps in the program are:

Step 1: Form an  $n \times (m + k)$  augmented matrix  $A$  by setting  $A = [M | Y]$ . A function `mkmtx` is used to construct  $M$ , one row at a time. Then  $Y$ , the matrix of successive  $y$ -coordinates, is appended to  $M$ .

Step 2: Find the QR decomposition of  $A$ . That is, write  $A = Q \cdot R$  where  $Q$  is an  $n \times n$  unitary matrix and  $R$  is of the form  $R = \begin{bmatrix} U \\ T \end{bmatrix}$  where  $U$  is an  $m \times m$  upper-triangular

matrix augmented with  $k$  column vectors and  $T$  is the  $(n-m) \times m$  zero-matrix augmented with  $k$  column vectors. Hence,

$$U = \left[ \begin{array}{ccc|ccc} r_{11} & \cdots & r_{1m} & h_{11} & \cdots & h_{1k} \\ \vdots & \ddots & \vdots & & & \\ 0 & \cdots & r_{mm} & h_{m1} & \cdots & h_{mk} \end{array} \right] \text{ and } T = \left[ \begin{array}{ccc|ccc} 0 & \cdots & 0 & h_{m+1,1} & \cdots & h_{m+1,k} \\ \vdots & \ddots & \vdots & & & \\ 0 & \cdots & 0 & h_{n1} & \cdots & h_{nk} \end{array} \right]$$

Step 3: Extract the matrix  $U$  from  $R$  since the solutions to the upper triangular system represented by the matrix  $U$  yields the desired coefficient lists  $\{a_{1j}, \dots, a_{mj}\}$ ,  $j = 1, \dots, k$  corresponding to the  $k$  data sets.

Step 4: Solve the  $k$  systems corresponding to  $U$ .

Step 5: Plot the  $k$  data sets and linear least square fits. This part of the program requires programming "trickery" to set up the graphing variables and settings and is explained in comments.

The code for the generalized linear least squares program is next, followed by a sample run.

```
linls(dmtx)
Prgm
Local mkmtx,qm,rm,m,n,s,a,xi,j

© This program is an implementation of the linear least squares
algorithm, using QR decomposition.

© dmtx is a matrix containg the data. The first column contains the
x-values, while the subsequent columns contain y-values. The
approximating functions need to be externally defined by a function
named f which must return a row matrix.

Define mkmtx(dat)=Func
Local m,a,i

© This function evaluates the approximating functions at each x-
value. It returns a matrix.

rowDim(dat)→m
newMat(m,colDim(f(x)))→a
For i,1,m
    f(dat[i,1])→a[i]
EndFor
Return a
EndFunc

rowDim(dmtx)→m
colDim(dmtx)→n
colDim(f(x))→s
```

- © Set up the plot data.
- © Augment the matrix of values of  $f$ , with the  $y$ -values of the data and store the result in matrix  $a$ .

```
NewData sysData,dmtx
augment(mkmtx(dmtx),subMat(dmtx,1,2,m,n))→a
```

- © Find the QR decomposition of  $a$  and
- © store the useful part of the  $R$ -matrix back in  $a$ .

```
QR a,qm,rm
subMat(rm,1,1,s,s+n-1)→a
```

- © Solve the resulting system and put the solutions row-wise back in  $a$ . The command "rref" stands for "Reduced Row Echelon form".

```
rref(a)→a
(subMat(a,1,s+1,s,s+n-1))T→a
```

- © Output:

```
ClrGraph
For j,2,n
```

- © Plot the  $(j-1)$ th data set.

```
"c"&string(j)→ξ
```

- © Create a new plot in plot variable  $(j-1)$ , based upon the  $x$ -value obtained from the 1st column of the system variable  $sysData$  and  $y$ -values obtained from the  $j$ th column of the same data variable. Note that  $\#ξ$  is called an indirection. When  $\#$  is followed by a string, the machine is instructed to interpret the string as a variable name.

```
NewPlot j-1,1,c1,#ξ,,,j
```

- © Store the approximating function for the  $(j-1)$ th data set in the  $(j-1)$ th graphing variable.

- © What follows appears to be little strange but it is nonetheless very powerful. The problem lies with the fact that local variables cease to exist beyond the execution of the program. By design, if an expression is stored into a graphing variable (which of course belongs to the system and is thus global) then variables within the expression are not replaced by their current value. So if such an expression involves a local variable, the graphing variable will not contain what was intended. The problem is remedied by the use of the command `expr`. This command takes a string as a parameter and proceeds into converting it into an expression which it subsequently executes. This process is similar to indirection but it is far more general.

```
string(dotP(a[j-1],f(x)))→ξ
ξ&"→y"&string(j-1)&"(x)"→ξ
expr(ξ)
EndFor
```

- © Adjust the graph screen to the data and return to the home screen.

ZoomData  
DispHome  
EndPrgm

Here are several screens demonstrating how to set up and run the program with two data sets and the function field 1,  $x$  and  $\sin x$ .

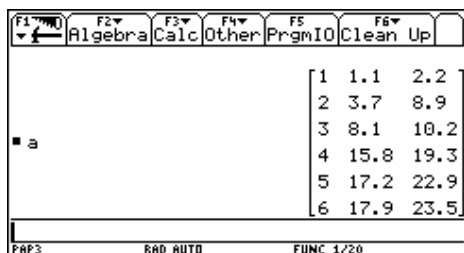


Figure 5a. Storage of Two Data Sets

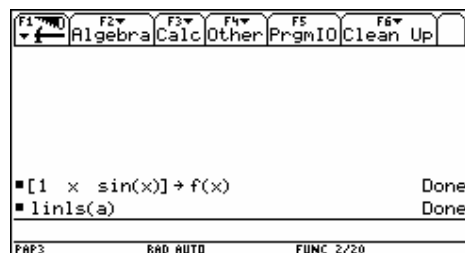


Figure 5b. Input of Three Functions

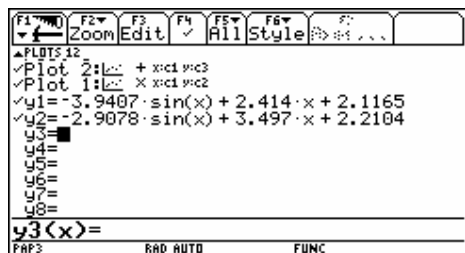


Figure 5c. Resulting Y-Window

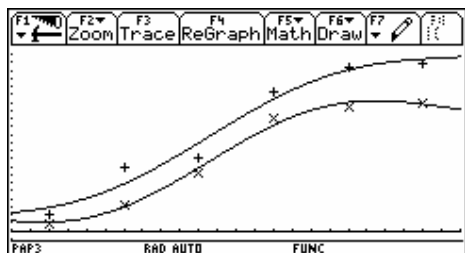


Figure 5d. Resulting Graph Window

## SOME FINAL REMARKS

In the last four sections we have illustrated that a small amount of TI-92+ programming can be used to solve several common and important linear algebra related problems. In most cases, the programming is straightforward enough for exercises, projects or classroom discussions which enhance learning of mathematics rather than detract from it. Recently there have been other articles, for example (Mann and Zehavi, 1998), (Cappuccio, 1997) and (Gonzalez-Vega, 1999), which demonstrate how programming on a computer algebra system can be used to investigate problems in calculus, linear algebra, numerical analysis, modern algebra and elementary number theory. Hence, there seems to be some agreement that students can discover valuable mathematics by creating and implementing programs.

Let us conclude with the additional thought that mathematically-oriented students may also gain valuable experience in computer science that might otherwise be missed through programming on the TI-92+ (see also (Abraham, 2000) and (Alexopoulos, 2000)). Indeed, although many mathematically oriented students will complete a beginning computer science course that utilizes a major language like Basic, Fortran or C++, meaningful high-level symbolic computation in those languages remains unrealistic for novice programmers. Even dynamic allocation and high-level list manipulations, which are natural and vital in many programming applications, may not be covered in beginning computer science courses required for mathematics, science and engineering students. To date, all of our linear algebra and multivariable calculus students—even those who have owned a symbolic programmable calculator for some time and have completed the computer science classes required for mathematics, science and engineering students—have had no clue at the start of the semester about dynamic allocation, high-level list manipulation, or symbolic programming.

## REFERENCES

- [1] Abraham, P. (2000). Using the TI-92+ Calculator as a Tool for Illustrating Programming Concepts. *Proceedings of the ICTCM-12*. <http://archives.math.utk.edu/ICTCM/EP-12.html>, October 10, 2000.
- [2] Alexopoulos, J. (2000). Implicit Differentiation on the TI-92+ Calculator as an Illustration of Some Powerful Programming Features. *Proceedings of the ICTCM-12*. <http://archives.math.utk.edu/ICTCM/EP-12.html>, October 10, 2000.
- [3] Cappuccio, S. (1997). TI-92 Programming Language: Two Examples of Application in Classroom. *International Journal of Computer Algebra in Mathematics Education*, **4 No. 3**, 273-288.



- [4] Cohoon, J.P. and Davidson, J.W. (1999). *C++ Program Design: An Introduction to Programming and Object-Oriented Design*, 2<sup>nd</sup> edition. McGraw-Hill.
- [5] Dubinsky E. (1995). ISETL: A Programming Language for Learning Mathematics. *Communications on Pure and Applied Mathematics*, **48**, 1027-1051.
- [6] Gonzalez-Vega, L. (1999). Using Linear Algebra to Introduce Computer Algebra, Numerical Analysis, Data Structures and Algorithms. *International Journal of Computer Algebra in Mathematics Education*, **6 No. 3**, 209-219.
- [7] Larson, R. and Edwards, B. (1996). *Elementary Linear Algebra*, 3rd edition. D.C. Heath Company.
- [8] Mann, G. and Zehavi, N. (1998). A Programming Approach to Extrema Problems. *International Journal of Computer Algebra in Mathematics Education*, **5 No. 4**, 269-277.
- [9] Petrovic, J. (1999). The Linear Least Squares Problem on the TI-89 and TI-92<sup>+</sup> Calculators. Honors project submitted to the Honors College of Kent State University.
- [10] Stoer, J. and Burlisch, R. (1993). *Introduction to Numerical Analysis*, 2<sup>nd</sup> edition. Springer-Verlag.
- [11] Texas Instruments Inc. (1998). *TI-92 Plus Module: A Supplement to the TI-92 Guidebook*. Texas Instruments Inc.

## **BIOGRAPHICAL SKETCH**

John Alexopoulos received his Ph.D. in mathematics from Kent State University in 1992. He formerly held full time faculty positions at Flagler College and the Illinois Mathematics and Science Academy. Currently, he is an assistant professor of mathematics at Kent State University Stark Campus. His research interests include functional and abstract analysis, measure theory and the teaching of undergraduate mathematics.

Paul Abraham received his Ph.D. in mathematics from Kent State University in 1993 under the supervision of Joe Diestel. His research interests include Banach spaces, martingale theory and teaching-related issues like course assessment and the use of technology in the classroom. He joined the KSU Stark faculty in 1996 after teaching at the College of the Ozarks in Missouri for four years.