DES-TIME-2006

Dresden International Symposium on Technology and its Integration into Mathematics Education 2006

9th ACDCA Summer Academy
7th Int'l Derive & TI-CAS Conference

20-23 July 2006, Dresden, Germany

Numerical methods with the Voyage 200: to function or to program, That is the question!

Gilles PICARD and Chantal TROTTIER
Ecole de technologie supérieure
1100, Notre-Dame street West, Montreal (Quebec) Canada
gilles.picard@etsmtl.ca
chantal.trottier@etsmtl.ca

ABSTRACT

We teach a variety of math topics (Calculus of one and several variables, Differential Equations…) in a Technical Engineering School. The Voyage 200 is mandatory for all new full-time students. We encounter in these courses a few classical numerical methods (Newton's method, numerical integration, Euler or Runge-Kutta methods). Beside the theoretical aspects, these methods are usually illustrated using functions or programs already defined in a CAS or using files or worksheets where students only have to modify some parameters and observe the results. Having to teach a course where numerical methods would be the main subject for about 4 weeks, we decided to ask students to program some of these algorithms using their TI symbolic calculators.

We will explain briefly the differences between a function and a program on these calculators and show examples of the functions used normally in classrooms for the illustration of the classical methods. We will then see how to incorporate important aspects (number of iterations, tolerance, etc.) in the programming of these functions. Why should we want to create a program instead of a function? We will illustrate this with more complex algorithms, using examples from linear algebra (Cholesky and Gauss-Seidel methods) and even show examples of programs done by students.

The aim here was to give students the interest and the ability to create their own programs on their calculator, showing them the advantage of programming in an environment where a great number of math-functions already exist. We were also able to create exam questions based on these programs and methods; examples will be shown in our presentation. We have observed more interest from the students for this topic when we asked them to program their calculators with these algorithms. Many of them were surprised how easy it was. Perhaps, teachers could see with their students slightly more complex functions or programs illustrating these numerical methods.

1. Introduction

Teaching basic mathematics in an engineering curriculum, we are used to see some classic numerical methods, for example solutions of equations with Newton's method, numerical integration with Simpson's method, solutions of differential equations with Euler's or Runge-Kutta's method. Students learn early on how to put a function in memory of their Voyage 200 calculators (everyone has it on his desktop in the classrooms). This is one of the first thing they learn with the use of the symbolic calculator. We insist that the calculator does not merely put the algebraic expression in memory but it's more like a small program which, for the example below, will associate $t - \cos(t)$ to the input t. Changing the input will not change what the function has to do. Of course, if the variable used has an assigned value, this will reflect the output of the function.

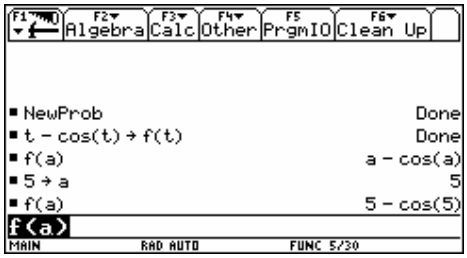

Figure 1

Combining the two preceding remarks will give, in the classrooms, usually a simple implementation of some numerical recipes. In the following screens we show Simpson's method which is implemented using the ability to create a one instruction function for this classic formula:

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots + 4f(x_{n-1}) + f(x_n)\right]$$

Here we use the fact that $\frac{b-a}{n} = h$ and we use the function mod(i,2) to insure the right multiplicative coefficients for the terms in the summation. The user must have memorized the function to be integrated in $f(x)$.

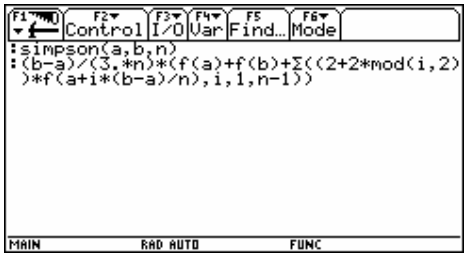

Figure 2

This is a straightforward implementation but it doesn't even check to see if the parameter n is even. In the figure above, we accessed the function with the Program Editor environment.

Another basic example is Newton's method which is usually seen as an application of derivatives for solving the equation $f(x)=0$ for a differentiable function. Let's look at finding a root of the equation $t-\cos(t)=0$ between 0 and 1. The classic iteration formula is $x_{n+1}=x_n-\dfrac{f(x_n)}{f'(x_n)}$ starting with an appropriate initial value x_0. If we want to keep things simple for the students, we'll put the initial value in a variable (*a* for this example) and ask the Voyage 200 to evaluate $a-\dfrac{f(a)}{f'(a)}$ and store the result again in variable *a*. Hitting the enter key calculates the new value each time, a process that we can repeat until the value calculated doesn't change anymore, which happens here after 5 iterations (see figures 3a and 3b).

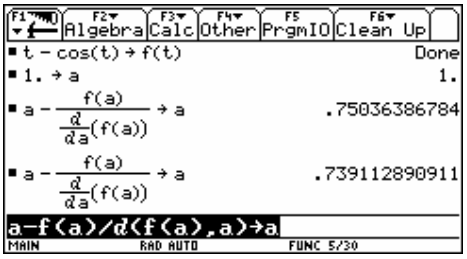

Figure 3a

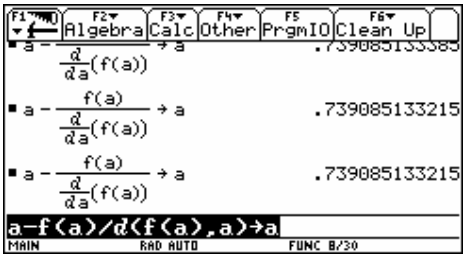

Figure 3b

We could also have defined a simple function named *newton*(*a*) which could do the same thing.

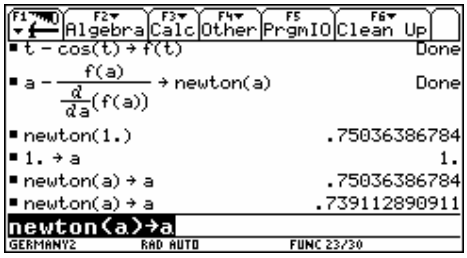

Figure 4

This is fine in a first Calculus course but it would be better if we could have more control on what happens, looking at the difference between two iterations or counting the number of iterations needed for a specified precision.

2. Functions, from basics to more complex

Let's look again at the Newton example and consider adding a verification in the function: we want to verify if the tangent is horizontal and have a message announcing this when it happens (of course Newton's method fails in this case and will produce an undef answer). This will be

done be editing a copy of the function already in memory, using the Program Editor. The next two screens show the initial function and the modified one where we added the verification.

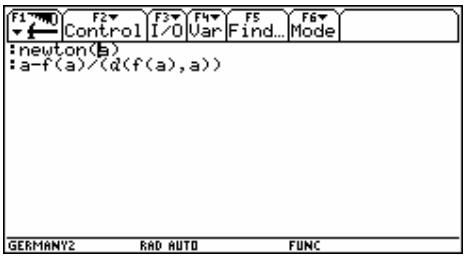

Figure 5a

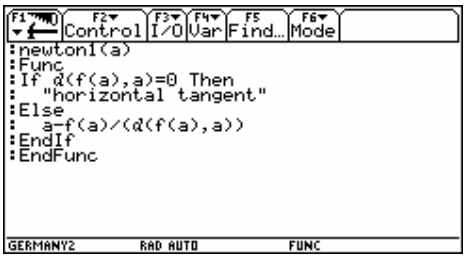

Figure 5b

Having a function with more than one instruction, we have to use the delimiters Func…EndFunc. The Voyage 200 uses classic and simple programming instructions, as showed in figure 5b where the function *newton1*(*a*) starts to look more like a small program. Using this new function still with the previous example of finding a root of $t-\cos(t)=0$, but starting with $t=\frac{3\pi}{2}$, so that the derivative has zero value, we see that the function gives the right warning.

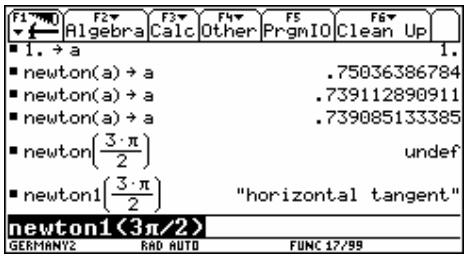

Figure 6

It would be natural next to try to control the precision obtained from one iteration to the next and to have a count of the number of iterations necessary to obtain a specified precision. We would like to implement the following rule: stop the iterations when the difference (in absolute value) of two consecutive iterations is less than a specified tolerance, let's call this value *tol*. We will add a maximum number of iterations (*maxit*) for convergence, to eliminate the possibility of a divergent process we would have to stop manually. Of course we would like the function to return two values: the result of the iterations and the number of steps needed to satisfy the tolerance specified. This challenges one limitation of working with functions, the output in the Home environment will be the result of the last line of code in the function. We will elude this problem by creating a list containing two elements: the resulting solution and the number of steps needed. Let's look in the following screens how the previous function is modified to incorporate these changes.

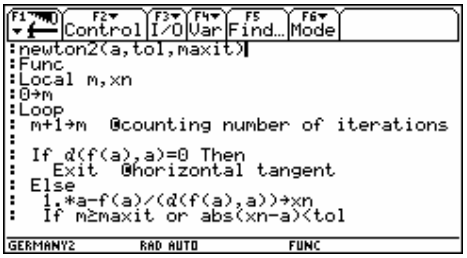

Figure 7a

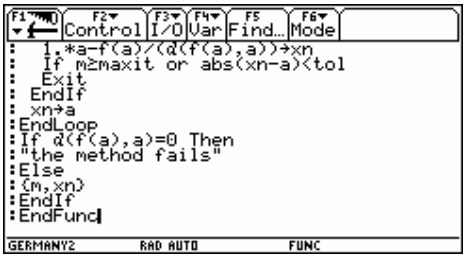

Figure 7b

We see that the function *newton2* now has 3 parameters, *a* being the initial value to start the iterations, *tol* the desired tolerance and *maxit* the maximum number of iterations. We have defined two local variables *m* and *xn*, *m* is for counting the number of iterations and *xn* is the new value obtained from the old one which is *a* (this is needed to evaluate the difference in absolute value between two consecutive iterations). We use a loop command to iterate the process, ending it when the tangent is horizontal or when the specified precision is obtained or when the maximum number of iterations is reached. The last line of code executed will be a message of error or a list containing the number of iterations (*m*), and the solution found (*xn*)

Let's solve again the initial problem of finding a root of the equation $t-\cos(t)=0$ between 0 and 1. We'll use a maximum number of iterations of 10 and a tolerance of 0.00005. We see in the following screen that if we start the process with $a=1$, we find in 3 steps a solution satisfying the desired tolerance.

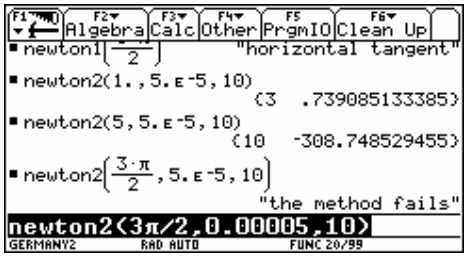

Figure 8

If we start with $a=5$, the calculator gives, after 10 steps, a false solution. Of course, when using Newton's method one must remember to use a starting point near the desired solution to insure convergence. When our function returns a number of steps equal to the maximum number of steps, one has to be suspicious of the result given. Finally, if the function encounters a null value for the derivative, we'll get an error message.

Looking at this last version of our Newton function, this really seems to be more a program than a function, using classic code for creating loops, using conditional testing… The user of this function still has to put the correct function to solve in a global memory, $f(x)$ in our example, and he has to decide a good starting value for the iterations to converge. An advantage of functions like this one is that the result is given in the Home environment and can be used in other calculations. Using the Voyage 200 for programming is quite interesting since one can use in their own functions or programs, system functions already implemented. We use in our function a command for the calculation and evaluation of the derivative of a function at a given value. So the big question, what is the difference between a function and a program, why use one instead of the other?

3. From functions to programs

A function can be called in the Home environment and will return only one value or expression there. It cannot create or modify a global variable in memory, but can use a value already assigned in a global memory variable. A program on the other hand can create or modify global variables as well as work with local variables. The only initial difference in the code is the first and last line of code where Prgm and EndPrgm will replace the function delimiters Func and EndFunc. A program will not return a value in the Home environment but has more elaborate options for Input/Output using the "PrgmIO" screen accessed with F5 in the Home environment. Let's look again at the preceding example but we will solve it using a program instead of a function. The following screens show the preceding function modified to create a program. The main differences are at the beginning and the end of the code where we modified the way the program gets the information for the initial parameters and the way it gives the results.

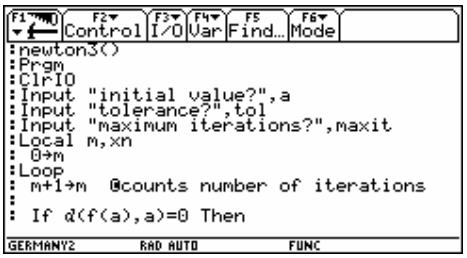

Figure 9a

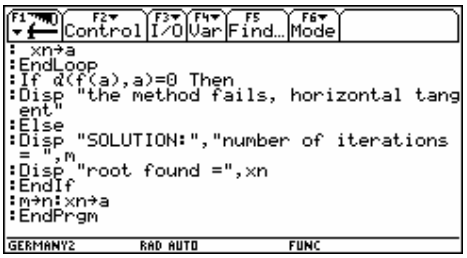

Figure 9b

We see here that the program *newton3*( ) has no parameters when called. We use the "Input" command to ask the users to input these values while the program is running. The "Disp"

command is used at the end to display the results in the PrgmIO environment. The next two screens show the result of entering the command newton3( ) in the Home environment.

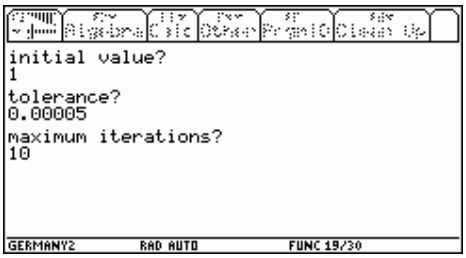

Figure 10a

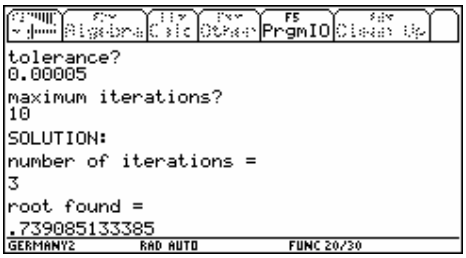

Figure 10b

If you look back at the last line of code in figure 9b, you'll see that the two results were put in global variables: *n* for the number of iterations done and *a* for the resulting root found for the equation. This is a big advantage when using a program; we may calculate more than one useful quantity or expression. You can recall these values in the Home environment, if you remember the names of the global variables that were used. But the user has to be aware that this will overwrite any value or expression already assigned to these variables. The next screen shows the recall of the two main values obtained in executing this program.

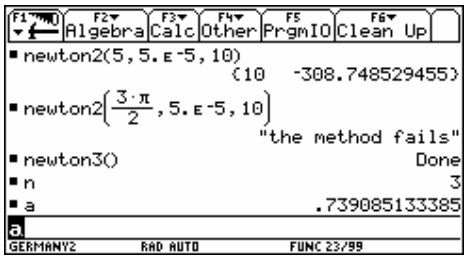

Figure 11

4. More complex examples

Let's look now at examples where the programming of the numerical methods is less obvious. Manipulating matrices is one area where students have to be careful. In figure 12a below, we see an example of a program using the command to create a new matrix; then the program manipulates these 2 new matrices. This example could be an exam question where students would be asked to describe what this program does given an input square matrix *a*. Even if the students type in the code instead of analyzing it, they still have to know how to find in the two global variables p and q what happens when executing this program.

The program in question *trans*(*a*) is illustrated below, in figure 12b, using the input matrix $a12 = \begin{bmatrix} 2 & -1 & 3 \\ -1 & 0 & 5 \\ -4 & 1 & 7 \end{bmatrix}$

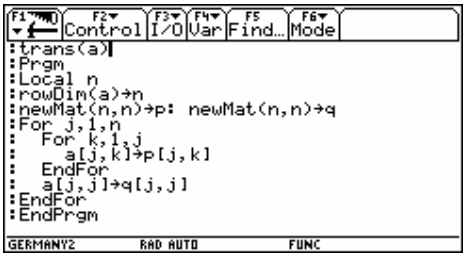

Figure 12a

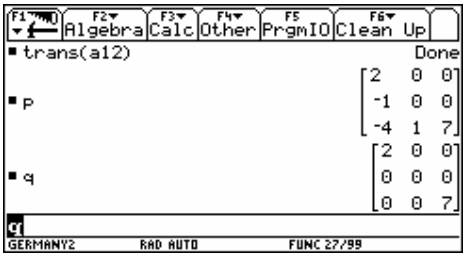

Figure 12b

The next examples deal with solving a system of *n* linear equations. Of course many of the classic algorithms for solving these systems are implemented in the Voyage 200: Gauss and Gauss-Jordan reduction as well as LU factorization. We will show two such methods, the Cholesky method and Gauss-Seidel iterative method. You will find, at the end of this text, in annex 1, copy of some pages of the book "Advanced Engineering Mathematics", 8th edition, from Erwin Kreyszig on this subject.

The basis for studying these methods is to try to keep the total count of arithmetic operations as low as possible in solving a system of *n* equations in *n* variables which has a unique solution. We can write the system to be solved as $Ax = b$ where A is a $n \times n$ matrix of coefficients, x is the column vector of variables and b is the column vector of constants of the equations.

The first of our two examples occurs when the matrix A of the system is symmetric and positive definite. In this case, the matrix A can be expressed as a product of a lower and an upper triangular matrix: $A = LU$ where the matrix U can be chosen to be the transpose of matrix L. Thus $A = LL^T$ and the task of doing the LU decomposition is therefore simpler than usual. Once this decomposition is done, the solution of the system $Ax = b$ can be obtained in two stages solving the equivalent system $LL^T x = b$ by doing a forward substitution in the triangular system $Ly = b$ and a backward substitution in the system $L^T x = y$. The following equations express the coefficients of the lower triangular matrix L in terms of the matrix of coefficients of the system A.

$$l_{11} = \sqrt{a_{11}}$$

$$l_{j1} = \frac{a_{j1}}{l_{11}} \qquad j = 2, \cdots, n$$

$$l_{jj} = \sqrt{a_{jj} - \sum_{s=1}^{j-1} l_{js}^{\,2}} \qquad j = 2, \cdots, n$$

$$l_{pj} = \frac{1}{l_{jj}} \left(a_{pj} - \sum_{s=1}^{j-1} l_{js}\, l_{ps} \right) \qquad p = j+1, \cdots, n; \quad j \geq 2$$

The following screens show a function, implementing these equations, on the Voyage 200 which will be used to calculate the lower triangular matrix L of a symmetric matrix A. In some semester, this function was given to student to illustrate programming and manipulating matrices with the Voyage 200, in other semester, students were asked to program this decomposition and gives us back their code with documentation which was then graded. You may notice the code used below which is almost identical to the given formulas.

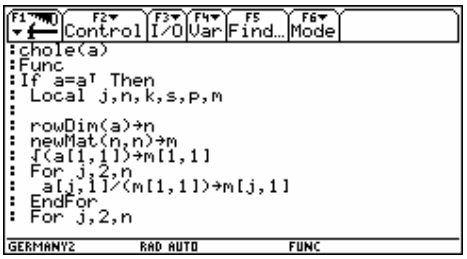

Figure 13a

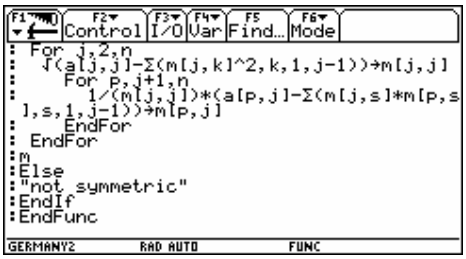

Figure 13b

Here is a numeric example with the matrix $\begin{bmatrix} 4 & 2 & 14 \\ 2 & 17 & -5 \\ 14 & -5 & 83 \end{bmatrix} = a13$ (see annex 1 for more details with this example):

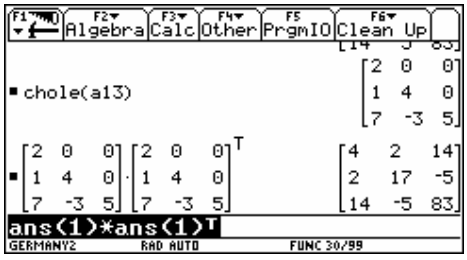

Figure 14

The last example is Gauss-Seidel method for solving the same linear system $Ax = b$, but using an iterative approach using an initial guess solution $x^{(0)}$ and calculating a sequence $x^{(1)}, x^{(2)}, \cdots, x^{(n)}$ where each column vector $x^{(i)}$ should be closer and closer to the solution of the system. We stop this process when we obtain a specified accuracy. This is kind of a vector-based

fixed point iteration process and it requires some conditions to be convergent and to provide a faster algorithm than the other classic methods. It can be interesting in large sparse system, with many 0 entries, with large diagonal values with respect to the other entries in each line of the matrix *A*. If we look at the example in annex 1, we see in fact the similarities with the fixed point method where we iterate this equation: $x_{n+1} = f(x_n)$. In some semester, we asked students to implement this method on their Voyage 200. This is the algorithm from Kreyszig's book (page 902):

Table 18.2
Gauss–Seidel Iteration

Algorithm Gauss–Seidel (A, b, $\mathbf{x}^{(0)}$, ϵ, N)

This algorithm computes a solution $\mathbf{x}$ of the system $\mathbf{Ax} = \mathbf{b}$ given an initial approximation $\mathbf{x}^{(0)}$, where $\mathbf{A} = [a_{jk}]$ is an $n \times n$ matrix with $a_{jj} \neq 0, j = 1, \cdots, n$.

INPUT: **A, b,** initial approximation $\mathbf{x}^{(0)}$, tolerance $\epsilon > 0$, maximum number of iterations N

OUTPUT: Approximate solution $\mathbf{x}^{(m)} = [x_j^{(m)}]$ or failure message that $\mathbf{x}^{(N)}$ does not satisfy the tolerance condition

For $m = 0, \cdots, N-1$, do:

For $j = 1, \cdots, n$, do:

1 $$x_j^{(m+1)} = \frac{1}{a_{jj}}\left(b_j - \sum_{k=1}^{j-1} a_{jk}x_k^{(m+1)} - \sum_{k=j+1}^{n} a_{jk}x_k^{(m)}\right)$$

End

2 If $\max_j |x_j^{(m+1)} - x_j^{(m)}| < \epsilon$ then OUTPUT $\mathbf{x}^{(m+1)}$. Stop

[*Procedure completed successfully*]

End

OUTPUT: "No solution satisfying the tolerance condition obtained after N iteration steps." Stop

[*Procedure completed unsuccessfully*]

End GAUSS–SEIDEL

One possible way of programming this method (it has to be a program if we want back the vector solution and the number of step since these two different object cannot be put in a list, like we did earlier with Newton's function) is to create a function, called *gauss*( ), for step 1 above, this calculates the next iteration, and do a program, *seidel*( ), which will call the function and control the accuracy and the number of steps required. The following screens illustrate this.

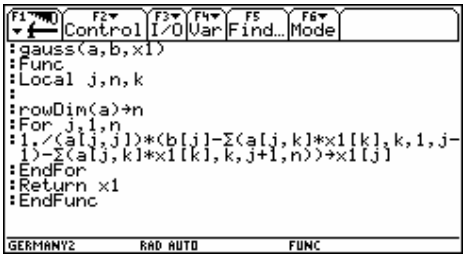

Figure 15a

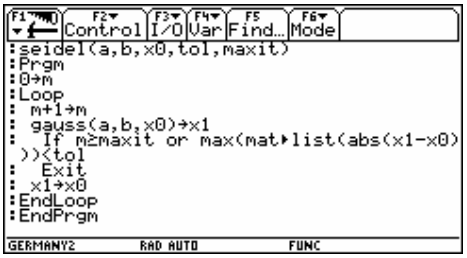

Figure 15b

You will note that iterations in the program are stopped when the maximum value in the vector column obtained by taking the absolute value of the difference between the old and the new iterate is less than the specified accuracy named *tol*.

Using the matrix $mata = \begin{bmatrix} 1 & -0.25 & -0.25 & 0 \\ -0.25 & 1 & 0 & -0.25 \\ -0.25 & 0 & 1 & -0.25 \\ 0 & -0.25 & -0.25 & 1 \end{bmatrix}$ and the vectors $matb = \begin{bmatrix} 50 \\ 50 \\ 25 \\ 25 \end{bmatrix}$ and $x0 = \begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \end{bmatrix}$ to illustrate the example seen in annex 1, where we solve the system of 4 equations $mata\,X = matb$ with initial guess solution $x0$ and column vector of variables $X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$, we get with the Voyage 200 applying the gauss( ) function and using a dummy variable *iter*:

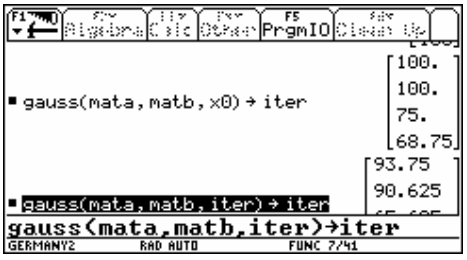

Figure 16a

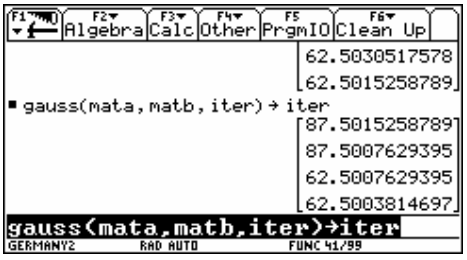

Figure 16b

After hitting the enter key a few times, we get a good sense of where the iterations are going and we have a solution that is precise to a few decimal places. Of course, like the Newton examples seen before, using the *seidel*( ) program, which calls this function will enable us to specify a desired accuracy, to count the number of iterations done and to specify a maximum number of iterations to be performed. The following screen shows that to obtain an accuracy of 0.005 , we need 8 iterations and the resulting approximate solution is given by *x1*.

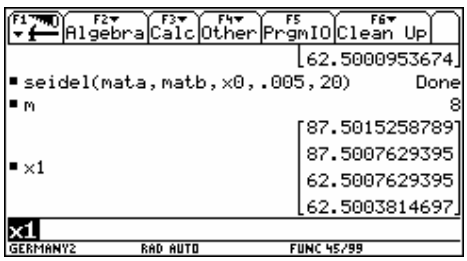

Figure 17

5. Examples from students

The next examples will show briefly work done by students where the assignment was to program and document either Gauss-Seidel's or Cholesky's method on their symbolic calculators. They had to give us the programs so we could test them.

Example 1:

This is a simple program which does Gauss-Seidel iterations. Here is the code:

```
seidelt(a,b,x0,err,maxloop)
Prgm
Local n,j
rowDim(a)->n
newMat(n,1)->xn

For c,1,maxloop

 For j,1,n
  1/(a[j,j]) * (b[j] - ∑(a[j,k]*xn[k],k,1,j-1) - ∑(a[j,k]*x0[k],k,j+1,n)) -> xn[j]
 EndFor

 If max(abs(xn-x0))[1,1] < err Then
  Exit
 Else
  xn->x0
 EndIf
EndFor
EndPrgm
```

This is quite similar to what we showed in figure 15, but simpler, everything is done inside the program; the number of iterations is in variable *c* and the solution is in the variable *xn*.

Example 2:

This next example, also with Gauss-Seidel, shows a function that could (may be) return the right solution but there is no way of knowing the number of iterations done. Furthermore, there is a problem if you don't give the right value for the parameter *n* which is the number of equations in the system (this should be done directly in the program with the appropriate command).

Figure 18a

Figure 18b

As we can see in the following screens, this last function does the right thing for our example if you give the right value for *n* (here we have 4 equations); but the function fails if you try $n = 5$ and even gives you a false answer if you use $n = 3$.

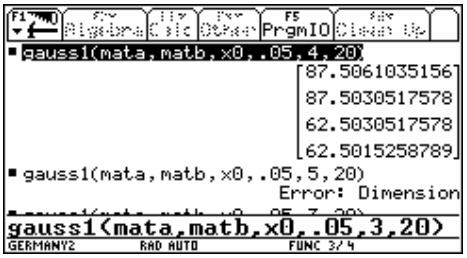

Figure 19a

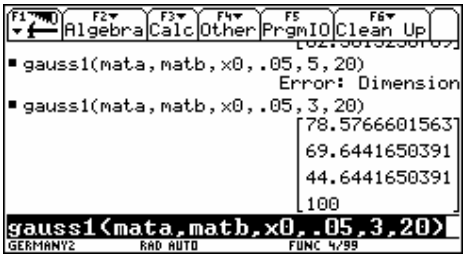

Figure 19b

Example 3:

Here is another program for Gauss-Seidel. This student had some problems with his program, in fact it worked with the numeric example he had in the assignment but it failed even with the system of equations we have seen earlier, which is the one used in showing this method to the students in this course. In the first screen below, we see that this student doesn't use the summation function available in his Voyage 200. In the second screen we see a bad test for the desired accuracy which will fail since it is not comparing the differences between variables in the same position.

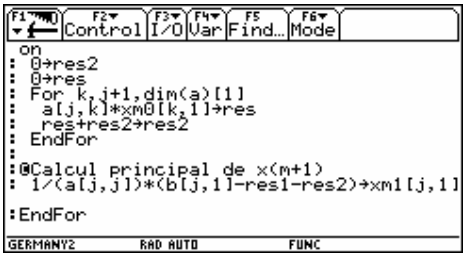

Figure 20a

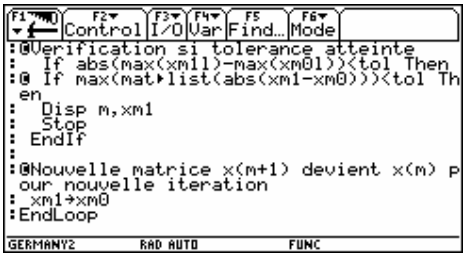

Figure 20b

You can see in the second line of code in figure 20b the problem mentioned above. If we execute this program with the previous numeric values of our example, it stops after one iteration since 100 is the maximum value in the initial vector and this value also appears in the next vector iterate, so the difference between these two maximum is 0, which is smaller than *tol.* We did put just below this line of code, in a comment, what could have been the right code for this test.

Example 4:

This last example shows a function doing Cholesky's decomposition.

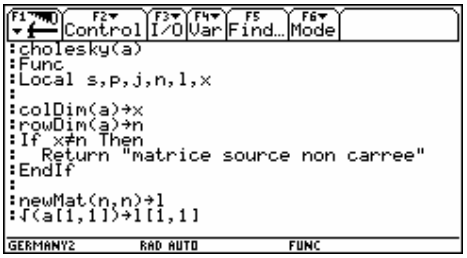

Figure 21a

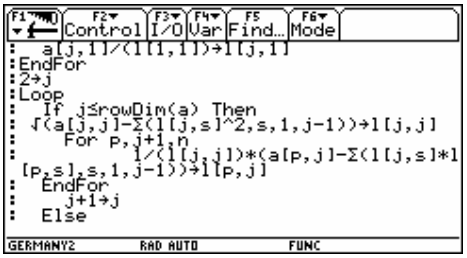

Figure 21b

This student had the good idea to test to see if the matrix is square; it would have been better to test also to see if it is symmetric.

6. Conclusion

Although we've illustrated only a few algorithms in this paper, students in our course had to write programs or functions for every numerical method seen in class (polynomial interpolation, numerical integration, fixed point iteration etc.). For most of our students, this was their first time in programming their Voyage 200 symbolic calculator. As we did discover in preparing for this course (we had never programmed either our Voyage 200), it is quite simple to do with this CAS system. Students discovered a new way of working with their calculators and, as usual, loved the direct feedback when working in classrooms with the Voyage 200. One colleague of us, who first gave this new course, chose to create a MAPLE worksheet with all the numeric methods and he gave this file to all the students. He used all the available functions or subroutines available in this software which left students with just about no programming to do. Since every student in our university has a symbolic calculator, we decided it would be more appropriate to ask them to do the programming themselves and to use this tool they all have on their desks.

We believe that even in the basic math courses, we could try to give students a glimpse of what they could do with small functions or programs. As for the initial question, "To function or to program", it depends on what you want to do! But if you need to generate more than one answer or expression you would be better off with a program, as long as you know how to get these values or expressions in global memory.

Reference:

Advanced Engineering Mathematics, 9th edition, *Erwin Kreyszig*
Wiley
(pages 897-898-898-901-902 are in annex 1, see next page)

Thus the factorization (2) is

$$\begin{bmatrix} 3 & 5 & 2 \\ 0 & 8 & 2 \\ 6 & 2 & 8 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & -1 & 1 \end{bmatrix} \begin{bmatrix} 3 & 5 & 2 \\ 0 & 8 & 2 \\ 0 & 0 & 6 \end{bmatrix}.$$

We first solve $\mathbf{Ly} = \mathbf{b}$, determining y_1, then y_2, then y_3, that is,

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 2 & -1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 8 \\ -7 \\ 26 \end{bmatrix}. \quad \text{Solution} \quad \mathbf{y} = \begin{bmatrix} 8 \\ -7 \\ 3 \end{bmatrix}.$$

Then we solve $\mathbf{Ux} = \mathbf{y}$, determining x_3, then x_2, then x_1, that is,

$$\begin{bmatrix} 3 & 5 & 2 \\ 0 & 8 & 2 \\ 0 & 0 & 6 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 8 \\ -7 \\ 3 \end{bmatrix}. \quad \text{Solution} \quad \mathbf{x} = \begin{bmatrix} 4 \\ -1 \\ 1/2 \end{bmatrix}.$$

This agrees with the solution in Example 1 of Sec. 18.1. ◀

Our formulas in Example 1 suggest that for general n the elements of the matrices $\mathbf{L} = [m_{jk}]$ (with main diagonal $1, \cdots, 1$ and m_{jk} suggesting "multiplier") and $\mathbf{U} = [u_{jk}]$ in the ***Doolittle method*** are computed from

$$\begin{aligned} u_{1k} &= a_{1k} && k = 1, \cdots, n \\ m_{j1} &= \frac{a_{j1}}{u_{11}} && j = 2, \cdots, n \\ \textbf{(4)} \qquad u_{jk} &= a_{jk} - \sum_{s=1}^{j-1} m_{js}u_{sk} && k = j, \cdots, n; \quad j \geqq 2 \\ m_{jk} &= \frac{1}{u_{kk}} \left(a_{jk} - \sum_{s=1}^{k-1} m_{js}u_{sk} \right) && j = k+1, \cdots, n; \quad k \geqq 2. \end{aligned}$$

Row Interchanges. Matrices, such as

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

have no LU-factorization (try!). This indicates that for obtaining an LU-factorization, row interchanges of $\mathbf{A}$ (and corresponding interchanges in $\mathbf{b}$) may be necessary.

Cholesky's Method

For a ***symmetric, positive definite*** matrix $\mathbf{A}$ (thus $\mathbf{A} = \mathbf{A}^T$, $\mathbf{x}^T\mathbf{Ax} > 0$ for all $\mathbf{x} \neq \mathbf{0}$) we can in (2) even choose $\mathbf{U} = \mathbf{L}^T$, thus $u_{jk} = m_{kj}$ (but impose no conditions on the main

diagonal entries). For example,

$$(5)\qquad \begin{bmatrix} 4 & 2 & 14 \\ 2 & 17 & -5 \\ 14 & -5 & 83 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 1 & 4 & 0 \\ 7 & -3 & 5 \end{bmatrix} \begin{bmatrix} 2 & 1 & 7 \\ 0 & 4 & -3 \\ 0 & 0 & 5 \end{bmatrix}.$$

The popular method of solving $\mathbf{Ax} = \mathbf{b}$ based on this factorization $\mathbf{A} = \mathbf{LL}^T$ is called **Cholesky's method.** In terms of the entries of $\mathbf{L} = [l_{jk}]$ the formulas for the factorization are

$$l_{11} = \sqrt{a_{11}}$$

$$l_{j1} = \frac{a_{j1}}{l_{11}} \qquad j = 2, \cdots, n$$

$$(6)\qquad l_{jj} = \sqrt{a_{jj} - \sum_{s=1}^{j-1} l_{js}^2} \qquad j = 2, \cdots, n$$

$$l_{pj} = \frac{1}{l_{jj}} \left(a_{pj} - \sum_{s=1}^{j-1} l_{js} l_{ps} \right) \qquad p = j+1, \cdots, n; \quad j \geqq 2.$$

If $\mathbf{A}$ is symmetric but not positive definite, this method could still be applied, but then leads to a *complex* matrix $\mathbf{L}$, so that it becomes impractical.

EXAMPLE 2 Cholesky's method

Solve by Cholesky's method:

$$\begin{aligned} 4x_1 + 2x_2 + 14x_3 &= 14 \\ 2x_1 + 17x_2 - 5x_3 &= -101 \\ 14x_1 - 5x_2 + 83x_3 &= 155. \end{aligned}$$

Solution. From (6) or from the form of the factorization

$$\begin{bmatrix} 4 & 2 & 14 \\ 2 & 17 & -5 \\ 14 & -5 & 83 \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{bmatrix}$$

we compute, in the given order,

$$l_{11} = \sqrt{a_{11}} = 2 \qquad l_{21} = \frac{a_{21}}{l_{11}} = \frac{2}{2} = 1 \qquad l_{31} = \frac{a_{31}}{l_{11}} = \frac{14}{2} = 7$$

$$l_{22} = \sqrt{a_{22} - l_{21}^2} = \sqrt{17 - 1} = 4$$

$$l_{32} = \frac{1}{l_{22}} (a_{32} - l_{31} l_{21}) = \frac{1}{4} (-5 - 7 \cdot 1) = -3$$

$$l_{33} = \sqrt{a_{33} - l_{31}^2 - l_{32}^2} = \sqrt{83 - 7^2 - (-3)^2} = 5.$$

This agrees with (5). We now have to solve $\mathbf{Ly} = \mathbf{b}$, that is,

$$\begin{bmatrix} 2 & 0 & 0 \\ 1 & 4 & 0 \\ 7 & -3 & 5 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 14 \\ -101 \\ 155 \end{bmatrix}. \qquad \text{Solution} \qquad \mathbf{y} = \begin{bmatrix} 7 \\ -27 \\ 5 \end{bmatrix}.$$

As the second step, we have to solve $\mathbf{Ux} = \mathbf{L}^T\mathbf{x} = \mathbf{y}$, that is,

$$\begin{bmatrix} 2 & 1 & 7 \\ 0 & 4 & -3 \\ 0 & 0 & 5 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 7 \\ -27 \\ 5 \end{bmatrix}. \qquad \text{Solution} \qquad \mathbf{x} = \begin{bmatrix} 3 \\ -6 \\ 1 \end{bmatrix}. \quad \blacktriangleleft$$

THEOREM 1 (Stability of the Cholesky factorization)

The Cholesky $\mathbf{LL}^T$*-factorization is numerically stable.*

PROOF. We have $a_{jj} = l_{j1}^2 + l_{j2}^2 + \cdots + l_{jj}^2$ by squaring the third formula in (6) and solving it for a_{jj}. Hence for all l_{jk} (note that $l_{jk} = 0$ for $k > j$) we obtain (the inequality being trivial)

$$l_{jk}^2 \leqq l_{j1}^2 + l_{j2}^2 + \cdots + l_{jj}^2 = a_{jj}.$$

That is, l_{jk}^2 is bounded by an entry of $\mathbf{A}$, which means stability against round-off. $\blacktriangleleft$

The two methods that we have discussed were particularly popular in desk-computer times because of their relatively small number of intermediate results. They are still attractive since they can be implemented with accumulated inner products (see [E17]).

Gauss–Jordan Elimination. Matrix Inversion

Another variant of the Gauss elimination is the **Gauss–Jordan elimination,** introduced by W. Jordan in 1920, in which back substitution is avoided by additional computations that reduce the matrix to diagonal form, instead of the triangular form in the Gauss elimination. But this reduction from the Gauss triangular to diagonal form requires more operations than back substitution does, so that the method is *disadvantageous* for solving systems $\mathbf{Ax} = \mathbf{b}$. But it may be used for matrix inversion, where the situation is as follows.

The **inverse** of a nonsingular square matrix $\mathbf{A}$ may be determined in principle by solving the n systems

$$\mathbf{Ax} = \mathbf{b}_j \qquad (j = 1, \cdots, n) \tag{7}$$

where $\mathbf{b}_j$ is the jth column of the $n \times n$ unit matrix.

However, it is preferable to produce $\mathbf{A}^{-1}$ by operating on the unit matrix $\mathbf{I}$ in the same way as the Gauss–Jordan algorithm, reducing $\mathbf{A}$ to $\mathbf{I}$. A typical illustrative example of this method is given in Sec. 6.7.

equation in a potential problem of 10^4 equations in 10^4 unknowns with typically only 5 nonzero terms per equation (more on this in Sec. 19.4).

Gauss–Seidel Iteration Method

This is an iterative method of great practical importance, which we can simply explain in terms of an example.

EXAMPLE 1 **Gauss–Seidel iteration**

We consider the linear system

$$
\begin{aligned}
x_1 - 0.25x_2 - 0.25x_3 &= 50 \\
-0.25x_1 + x_2 - 0.25x_4 &= 50 \\
-0.25x_1 + x_3 - 0.25x_4 &= 25 \\
-0.25x_2 - 0.25x_3 + x_4 &= 25.
\end{aligned} \tag{1}
$$

(Equations of this form arise in the numerical solution of partial differential equations and in spline interpolation.) We write the system in the form

$$
\begin{aligned}
x_1 &= 0.25x_2 + 0.25x_3 + 50 \\
x_2 &= 0.25x_1 + 0.25x_4 + 50 \\
x_3 &= 0.25x_1 + 0.25x_4 + 25 \\
x_4 &= 0.25x_2 + 0.25x_3 + 25.
\end{aligned} \tag{2}
$$

We use these equations for iteration, that is, we start from a (possibly poor) approximation to the solution, say, $x_1^{(0)} = 100$, $x_2^{(0)} = 100$, $x_3^{(0)} = 100$, $x_4^{(0)} = 100$, and compute from (2) a presumably better approximation

Use "old" values

("New" values here not yet available)

$$
\begin{aligned}
x_1^{(1)} &= 0.25x_2^{(0)} + 0.25x_3^{(0)} + 50.00 = 100.00 \\
x_2^{(1)} &= 0.25x_1^{(1)} + 0.25x_4^{(0)} + 50.00 = 100.00 \\
x_3^{(1)} &= 0.25x_1^{(1)} + 0.25x_4^{(0)} + 25.00 = 75.00 \\
x_4^{(1)} &= 0.25x_2^{(1)} + 0.25x_3^{(1)} + 25.00 = 68.75.
\end{aligned} \tag{3}
$$

Use "new" values

We see that these equations are obtained from (2) by substituting on the right the ***most recent*** approximations. In fact, corresponding elements replace previous ones as soon as they have been computed, so that in the second and third equations we use $x_1^{(1)}$ (not $x_1^{(0)}$), and in the last equation of (3) we use $x_2^{(1)}$ and $x_3^{(1)}$ (not $x_2^{(0)}$ and $x_3^{(0)}$). The next step yields

$$
\begin{aligned}
x_1^{(2)} &= 0.25x_2^{(1)} + 0.25x_3^{(1)} + 50.00 = 93.75 \\
x_2^{(2)} &= 0.25x_1^{(2)} + 0.25x_4^{(1)} + 50.00 = 90.62 \\
x_3^{(2)} &= 0.25x_1^{(2)} + 0.25x_4^{(1)} + 25.00 = 65.62 \\
x_4^{(2)} &= 0.25x_2^{(2)} + 0.25x_3^{(2)} + 25.00 = 64.06.
\end{aligned}
$$

In practice, one would do further steps and obtain a more accurate approximate solution.

The reader may show that the exact solution is $x_1 = x_2 = 87.5$, $x_3 = x_4 = 62.5$. ◀

To obtain an algorithm for the Gauss–Seidel iteration, let us derive the general formulas for this iteration.

We assume that $a_{jj} = 1$ for $j = 1, \cdots, n$. (Note that this can be achieved if we can rearrange the equations so that no diagonal coefficient is zero; then we may divide each equation by the corresponding diagonal coefficient.) We now write

$$\mathbf{A} = \mathbf{I} + \mathbf{L} + \mathbf{U} \qquad (a_{jj} = 1) \tag{4}$$

where $\mathbf{I}$ is the $n \times n$ unit matrix and $\mathbf{L}$ and $\mathbf{U}$ are respectively lower and upper triangular matrices with zero main diagonals. If we substitute (4) into $\mathbf{Ax} = \mathbf{b}$, we have

$$\mathbf{Ax} = (\mathbf{I} + \mathbf{L} + \mathbf{U})\mathbf{x} = \mathbf{b}.$$

Taking $\mathbf{Lx}$ and $\mathbf{Ux}$ to the right, we obtain, since $\mathbf{Ix} = \mathbf{x}$,

$$\mathbf{x} = \mathbf{b} - \mathbf{Lx} - \mathbf{Ux}. \tag{5}$$

Remembering from our computation in Example 1 that below the main diagonal we took "new" approximations and above the main diagonal "old" approximations, we obtain from (5) the desired iteration formulas

$$\boxed{\mathbf{x}^{(m+1)} = \mathbf{b} - \mathbf{L}\mathbf{x}^{(m+1)} - \mathbf{U}\mathbf{x}^{(m)}} \qquad (a_{jj} = 1) \tag{6}$$

where $\mathbf{x}^{(m)} = \left[x_j^{(m)}\right]$ is the mth approximation and $\mathbf{x}^{(m+1)} = \left[x_j^{(m+1)}\right]$ is the $(m+1)$st approximation. In components this gives the formula in line 1 in Table 18.2. The matrix

Table 18.2
Gauss–Seidel Iteration

Algorithm Gauss–Seidel $(\mathbf{A}, \mathbf{b}, \mathbf{x}^{(0)}, \epsilon, N)$

This algorithm computes a solution $\mathbf{x}$ of the system $\mathbf{Ax} = \mathbf{b}$ given an initial approximation $\mathbf{x}^{(0)}$, where $\mathbf{A} = [a_{jk}]$ is an $n \times n$ matrix with $a_{jj} \neq 0, j = 1, \cdots, n$.

INPUT: **A, b,** initial approximation $\mathbf{x}^{(0)}$, tolerance $\epsilon > 0$, maximum number of iterations N

OUTPUT: Approximate solution $\mathbf{x}^{(m)} = \left[x_j^{(m)}\right]$ or failure message that $\mathbf{x}^{(N)}$ does not satisfy the tolerance condition

For $m = 0, \cdots, N-1$, do:

  For $j = 1, \cdots, n$, do:

1 $$x_j^{(m+1)} = \frac{1}{a_{jj}}\left(b_j - \sum_{k=1}^{j-1} a_{jk}x_k^{(m+1)} - \sum_{k=j+1}^{n} a_{jk}x_k^{(m)}\right)$$

  End

2 If $\max_j \left|x_j^{(m+1)} - x_j^{(m)}\right| < \epsilon$ then OUTPUT $\mathbf{x}^{(m+1)}$. Stop

  [*Procedure completed successfully*]

End

OUTPUT: "No solution satisfying the tolerance condition obtained after N iteration steps." Stop

[*Procedure completed unsuccessfully*]

End GAUSS–SEIDEL