

Tutorial for using TrueBasic to write computer programs

Prepared for the Course: Mathematical Models for the Study of Animal Behavior
at the Dept. of Life Sciences, Ben-Gurion University (1998)
Course instructor: Amos Bouskila

CONTENTS:

Introduction

1. Input and output methods:
 - 1a. Simple input in the program; output on the screen
 - 1b. Input in the program using the data command
 - 1c. External input, from a file
 - 1d. External input: interactive from the keyboard
 - 1e. External output: save results to a file (including the commas needed for opening it later with Excel)
2. Loops
 - 2a. For... next
 - 2b. Do until ... loop
 - 2c. Do while... loop
3. Arrays (and declaration of variables in a **dim** statement)
4. If statements
 - 4a. Simple If statements - one line only
 - 4b. If statements with several lines of commands to execute
 - 4c. If statements with several conditions
 - 4d. Select ... case
5. Exiting from a loop (exit for, exit do); Automatic indentation of your program
6. Printing complex output
 - 6a. Using a simple table
 - 6b. Using rows of different length
7. Numeric functions (built in functions)
8. Random number generator
 - 8a. Simulate events occurring at certain probabilities.
 - 8b. Random selection of events or items
 - 8c. Random selection of equally probable events or items – the easy way
9. User defined functions
10. Subroutines
11. Programming tips
 - 11a. Stages for writing a computer program
 - 11b. Tips and hints for good programming practices

INTRODUCTION

The tutorial is meant for students who have no programming experience and those whose programming experience was not with TrueBasic.

Rather than embarking in long explanation, this tutorial offers an opportunity to learn through experience. In order for this to work efficiently, users of the tutorial must run the program by themselves, and not just read them. It is recommended to write in a file, in a notebook, or even on these pages, the function of various commands to which you will be exposed while you are running the programs. Some of the material in the operation of TrueBasic is left to be learnt by the students, in order to make it an active learning experience. As you advance in the tutorial, the assignments require more complexity and some creativity.

Only those operations that were considered to be important for writing programs appear in the tutorial. These should be enough to write your programs. To learn more, you can consult the manual of TrueBasic. This tutorial was written with users of TrueBasic for Macintosh computers in mind. Nevertheless, users of IBM compatibles will find the TrueBasic for DOS identical in most respects. The programs that you write can be transferred from one type of computer to the other, although the details are not provided here.

To start typing in a new program, you double-click on the icon of TrueBasic, and you can start typing your statements in the clean screen you get. Remember to save your work, as well as making backups.

I thank the students who used previous versions of the tutorial and reported errors and corrections needed, and especially Shai Markman and Pablo Blinder who tested the operation of the programs.

Errors in the tutorial or suggestions for clarifications should be reported to my e-mail: bouskila@bgumail.bgu.ac.il

A. B.

In general, a computer program consists of three main parts:

1. Input (of data, parameter values, etc)
2. Operations performed on the input
3. Output (the end results of these operations, reported to the user of the program).

We will start by learning a few methods for input and output, and only then, describe various operations that can be used.

1. Input and output methods

In the following programs you will often encounter sentences starting with “!”; anything after the exclamation mark is a comment for the readers only, and in TrueBasic it has no effect on the operation of the program. Such comments may be very useful, though, to remind you, or somebody else, what you meant when you wrote a certain part of the program. Of course, when you copy the programs from here for your exercises there is no necessity to copy the comments.

Write a program that will have the following structure:

! Simple Program – addition1 ! it's very useful to have a meaningful title

! although the program will operate also without it

option nolet ! * See explanation below

x=5

y=7

sum=x+y

print sum !this format of print writes only on the screen

* If the program does not include the **option nolet** statement, you will need to use the longer method for assignment of values to variables (e.g., **let x=5**) throughout the whole program. Here we can use the shorter way of writing (**x=5**).

The terms **x,y** and **sum** in this program are variables – basically they are locations in the memory of the computer that are given these names, and are assigned a certain value.

Note that in TrueBasic variable names can be either a letter, or a whole word. There is an advantage in naming variables so that you will remember what they mean. Note however, that you cannot name a variable **time** or **date**, because these are reserved names for built-in functions of TrueBasic.

Now run the program (in programmers jargon, this means “let the program operate”) by going to the menu RUN, and pointing on the word Run from there.

1a. Simple input in the program; output on the screen

After running this small program, replace the last line to provide a more readable output (still on the screen only) by:

```
print “x=”; x, “y=”; y, “Sum of x+y=”; sum
```

Note the way TrueBasic treats statements in between “ ” and also how you have control over spaces by the different separators: , or ;

1b. Input in the program using the data command

So far, the input was an integral part of the program, originating from within the program.

The following is also an input from the program, but has a different structure:

! Simple Program – addition2

```
option nolet
```

```
data 5, 7
```

```
read x, y ! this will assign the first value in the data line to x, and the second to y
```

```
sum=x+y
```

```
print “x=”; x, “y=”; y, “Sum of x+y=”; sum
```

```
end
```

1c. External input: from a separate file

Create a new file (in TrueBasic) that has just one line:

```
5, 7
```

Save the file under the name mydata1 (or some other name that you will have to remember later)

Modify your program to read from the external file:

! Simple Program – addition3

```
option nolet
```

```
open #1: name “mydata1”, create newold, org text
```

```
! open the data file for reading from it, and assign to it the number 1
```

```
input #1: x, y ! this will assign the first value in the data file to x, and the second to y
```

```
sum=x+y
```

```
print “x=”; x, “y=”; y, “Sum of x+y=”; sum
```

The program knows in which file to look for the data by the number assigned to the file name, in this case #1.

Run the program, and make sure the program indeed reads the data from the file. (you can change the data in the file, whenever needed).

The advantage of reading from an external file is that often it is convenient to prepare/organize your data set in Excel, save it as a text file (using commas as separators) and then working on this data with your TrueBasic program.

1d. External input: interactive from the keyboard

! Simple Program – addition4

```
option nolet  
input prompt “What is the value of x?”: x  
input prompt “What is the value of y?”: y  
sum=x+y  
print “x=”; x, “y=”; y, “Sum of x+y=”; sum  
end
```

Run the program and you will see how you are prompted to enter the data by questions on the screen.

1e. External output: save results on a file

When you want to save the results on a file (for printing, further analysis by Excel, graphing, etc)

! Simple Program – addition5

```
option nolet  
open #1: name “mydata1”, create newold, org text  
open #2: name “results1”, create newold, org text  
reset #2: end ! this is important for files on which you write the output  
print #2: “Simple Program – addition5”  
! The previous line is useful - it will print a heading in the results file  
input #1: x, y ! input data from the external file  
sum=x+y  
print “x=”; x, “y=”; y, “Sum of x+y=”; sum  
print #2: “x=”; x, “y=”; y, “Sum of x+y=”; sum  
!print on output file with same format as on the screen  
close #2  
end
```

It's a good idea to close output files before the end of the program, because in some cases leaving such a file open will cause problems opening it later.

Note that in this case we left the line that prints also on the screen (to verify that the program is doing what we want). It is recommended, but not essential.

If we plan to open later the output file with Excel, we can plan the print line in such a way that will be easily opened by Excel (opened as CSV format, i.e., comma separated). To do this, replace the printing line with the following:

```
print #2: x; “,”; y; “,”; sum
```

or if you want to keep the explaining sections of this line:

```
print #2: “x=,”; x, “y=,”; y, “Sum of x+y=,”; sum
```

Try these and try to open the result file with Excel.

2. Loops

Very useful in programs which have to repeat the same procedure over and over again. They can be used for reading data, processing data, or printing data. We will describe three types of loops

2a. For ... next

! Looping Program – 1

! This program will create a plot for the equation $y=a*x^2 - b*x + c$

option nolet

open #1: name “result1”, create newold, org text

reset #1: end

print #1: “Looping Program – 1”

data 1, 12, 27

data 11 ! this value will be assigned to the variable xmax below

read a, b, c

read xmax !this variable will determine how many times the loop will be repeated

for x = 1 to xmax

! the loop starts from here; x is a counter that determines where we are in the loops

y=a*x^2 – b*x + c

print “x=”; x, “y=”; y

print #1: x; “,”; y

next x ! this marks the end of the loop – from here the loop repeats itself, until x will reach xmax

close #1

end

Run this program, and open the results file with Excel. Plot the data that you received. You should get the typical U-shaped plot of a square root equation of this type. Save this plot – you will need it later.

2b. Do until ... loop

! Looping Program – 2

! This program will create a plot for the equation $y=a*x^2 - b*x + c$

option nolet

open #1: name “result1”, create newold, org text

reset #1: end

print #1: “Looping program – 2”

data 1, 12, 27

data 11

read a, b, c

read xmax

x=0 ! a starting value that will be modified later

do until x=xmax ! this is the condition for ending the loop

x=x+1 ! x will increase by 1 every time the loop passes through this stage; the first value used is 1

y=a*x^2 – b*x + c

print “x=”; x, “y=”; y

print #1: x; “,”; y

loop ! marks the end of the loop – from here the loop repeats itself, until x will reach the condition

close #1

end

2c. Do while ... loop

This is similar to the previous loop method, but the condition is tested at the end of the loop, and not at the beginning.

...
Do while x<xmax

and will end as before:

...
loop

TrueBasic executes any program you have written sequentially, i.e., according to the order it encounters the statements. A loop of any kind interferes in this process, but after the loop is repeated in the requested number of times, the sequential execution is resumed.

3. Arrays

An array is a list of numbers with one or more subscripts. For example, X(1), X(2)... is a one-dimensional array (a vector). A point in a two dimensional plane is part of a two-dimensional array: A(x1,y1), A(x2,y2)...

Arrays are very useful for storage of many values; these can be easily extracted later, using the subscripts (by using loops, for example). Arrays **MUST** be defined at the beginning of the program with a **dim** statement. Once you define a variable as an array, you cannot use the same name in the that program without the subscript.

! Looping Program – 3

! This program is based on Looping Program – 1, but uses a one-dimensional array

option nolet

dim y(20) ! here we reserve memory space for an array of up to 20 items

open #1: name "result1", create newold, org text

reset #1: end

print #1: "Looping program – 3"

data 1, 12, 27

data 20 ! note that this change will increase xmax from the previous version

read a, b, c

read xmax

for x = 1 to xmax

y(x)=a*x^2 – b*x + c

print "x="; x, "y(x)="; y(x)

next x

for i = 1 to 11 ! here we start a second loop that will print only part of the array created above

! the letter "i" is often used as a counter in loops. Here i takes the role of x

print "x="; i, "y(x)="; y(i)

print #1: i; ","; y(i)

next i

close #1

end

Note that y(x) is equivalent to y(i), because in fact, the array stores the values as y(1), y(2) etc. and in the first loop x is the variable that includes 1,2,3... and in the second – it's i that gets these values.

You may not see the importance of the array in the above program, but it's a very useful concept that will be used a lot in more complicated program.

4. If statements

It is often very useful to use conditional statements, which operate a set of commands only if a certain condition is fulfilled.

4a. Simple If statements – one line only

We will use now an if statement that will find the min value in the U-shaped function described by $y = a*x^2 - b*x + c$

Run the following program, and make sure that you understand what each line does. If you are not sure – don't hesitate to ask the staff.

! If-statement Program – 1

! This program is based on Looping Program – 3, but locates the minimum value with a condition

option nolet

dim y(20)

open #1: name "result1", create newold, org text

reset #1: end

print #1: "If-statement Program – 1"

data 1, 12, 27

data 20

read a, b, c

read xmax

ymin=0 !just a starting value; will be replaced by lower numbers, if encountered by the program

for x = 1 to xmax

y(x)=a*x^2 - b*x + c

print "x="; x, "y(x)="; y(x)

if y(x) < ymin then ymin = y(x) ! this updates the value of ymin, and stores the lowest value

next x

print ! this command followed by nothing prints an empty line

print "The lowest value of y =";ymin

print #1: The lowest value of y =";ymin

close #1

end

4b. If statements with several lines of commands to execute

We will use now an if statement that will store the solutions of the equation $a*x^2 - b*x + c=0$ in a small array with two items x0(1) and x0(2).

! If-statement Program – 2

! This program is based on If-statement Program – 1, but locates the solutions with a condition

option nolet

dim y(20), x0(5) !we now store space also for the small array.

! Even though we need only 2 cells we may ask for more

open #1: name "result1", create newold, org text

reset #1: end

print #1: "If-statement Program – 2"

data 1, 12, 27

data 20

read a, b, c

read xmax

i=0 ! this is a counter that will be used later

c

```

y(x)=a*x^2 - b*x + c
print "x="; x, "y(x)="; y(x)

```

if y(x) = 0 then ! we start a series of statements; will be performed if the condition is fulfilled
i = i +1 ! here we use the counter to mark the two possible solutions as 1 and 2
x0(i) = x ! here we store the x of the solution in our little array of solutions
end if ! you always need this when you have an if statement that is longer then one line

```

next x

```

```

print
print "The solutions for the equation are:"
for i =1 to 2
print "x0(i)=";x0(i)
print #1: "x0(i)="; x0(i)
next i

```

```

close #1
end

```

Run this program and check with the U-shaped plot you plotted in Excel (in section 2a) that the program indeed finds the correct values.

4c. If statements with several conditions

In the previous program, we basically separated the values of y(x) that were equal to 0, and we asked what are the x's that fulfil this condition.

Imagine that you wanted to separate the values to three groups: those x values that cause y(x) < 0, those that cause values of y(x) above 0, and those where y(x) equal to 0. You would have to use a complex if-statement with several conditions. Create a program based on If-statement program – 2 and call it If-statement Program – 3. Replace in it the lines of the if statement with the following:

```

if y(x) < 0 then
n=n+1
xnegative(n) = x
else if y(x) = 0 then
i=i+1
x0(i) = x
else ! This means – all other options, which in our case, are all the positive values
p=p+1
xpositive(p) = x
end if

```

If you just introduce these changes, the program will not work. BEFORE you run the program, go over this paragraph and see what other changes in the program are required (all these essential changes need to be made somewhere before this paragraph).

In addition, you need to have a print statement, that will print the end result of the program.

Once this program is running and you are happy with it, save it under the name If statement program -3, and print it, to hand it in with your lab assignment.

4d. Select ... case

If you have several conditions that depend on different values of a single expression, you can use a command that simplifies and avoids having several if statements. The

following example makes use of the **select ... case** in a version of the If-statement Program – 3.

Save the modified program under a different name, so that you leave your previous program, If-statement Program 3 untouched. (You will need it in section 6a).

Imagine that we wanted to single out 4 group of x's: one that lead to a $y(x)=0$, the second to values between 1 and 5 (we will call this group smallposi), the third, called smallnega, includes all the values between 0 and -5 , and the last one (called others), will include all other values. What we want to know is just how many values there are of each type.

Replace the paragraph between the **if ... end if** with the following

```
select case y(x) ! declaring that the selection is based on the value of y(x)  
case 0 ! equivalent to saying: "if y(x)=0 then..."  
i = i +1  
case 1,2,3,4,5 ! equivalent to saying: "if y(x)=1 or 2 or 3 or 4 or 5, then..."  
smallposi=smallposi+1  
case -1,-2,-3,-4,-5  
smallnega=smallnega+1  
case else  
others=others+1  
end select ! you must have such a statement at the end
```

! print the results on the screen

```
print "zeros=";i,"smallposi="; smallposi, "smallnega=";smallnega, "others="; others  
! print on file  
print #1: " Using Select Case"  
print #1:"zeros=";i,"smallposi="; smallposi,"smallnega=";smallnega,"others="; othe
```

5. Exiting from a loop

When you have a loop of some sort, apart from the conditions that regulate the loop (for, while or until), you can have additional conditions that will stop the operation of the loop, using an if statement.

Here is an example based on If-statement Program – 2:

! If-statement Program – 4

! This program finds the solutions with a condition, and stops once they have been found

! it is based on the fact that we know there are only up to 2 solutions to a square root equation

```
option nolet
```

```
dim y(20), x0(5)
```

```
data 1, 12, 27
```

```
data 20
```

```
read a, b, c
```

```
read xmax
```

```
i =0
```

```
for x = 1 to xmax
```

```
  y(x)=a*x^2 - b*x + c
```

```
  if y(x) = 0 then
```

```
    i = i +1
```

```
    x0(i) = x
```

```
    print "x0(i)=";x0(i)
```

```
  end if
```

```
  if i=2 then exit for ! stops the execution of the loop if the second solution is found
```

```
next x
```

In this case, we abandon the loop before we reach the $x=x_{\max}$ condition. If we were using a do loop of some sort (do while or do until) the equivalent statement would be:

```
if i=2 then exit do
```

These exit option may turn to be very useful when you have very long loops, and you can find reasons to exit before the end of the loop: it can shorten considerably the time programs run.

Note how indentation of loops and if statements can help in reading and checking your program. From now on, it is recommended that you start using this practice. There is a way to have the indentation done for the whole program automatically, using the DO FORMAT function under the menu "Custom".

6. Printing complex output

6a. Using a simple table

After you have made the required changes in If statement Program – 3, the program hopefully ran, but it did not present the data that you wanted on the screen or in a file. Here is one option for presenting the three groups of values:

Replace in the program the loop for printing with the following loop:

```
print  
print "count", "negat.", "zero", "positives" !this is a line of titles for the  
table below  
print  
for k=1 to 15 ! counter is named k, in order not to confuse with the i used in one of the 3 groups  
print k, xnegative(k), x0(k), xpositive(k)  
next k
```

Some of the values in the table will be 0's because we used a large k (15). You might need to change your array definitions, if you had arrays smaller than that. If the program runs, you will see the advantage of commas in print statements for tables, because they are used as tabs.

6b. Using rows of different length

Another, maybe more elegant in this case, way for printing the results is based on knowing how many values you have for each of the three groups. Maybe you don't know a-priori how many you will have in each group, but after the program completed the sorting through the complex if-statement, it already knows how many – the current value of n, i and p are the sizes of the negative, the zero group, and the positives. If you see why, you can replace the loop for printing with the following:

```
print  
print "Negatives:",  
! comma at the end of line will cause the next item to be printed on the same line  
for k=1 to n ! remember that n is the number of values in the group of negatives  
print xnegative(k);", ";  
! all these will appear in one line, due to the ; at the end of the line  
next k  
print  
print "Zeros:",  
for k=1 to i  
print x0(i);", ";  
next k  
  i
```

Try to run the program at this stage. You should get an output with the two first groups, each one in a separate line.
 Now that you got the general idea, complete the paragraph with the commands needed to print also the third group (the positives) in the same format.

7. Numeric functions

There are several functions built in TrueBasic, and it is useful to know they exist, in case you will need them.

The following is a simple program to demonstrate how some of these functions operate.

! Program Numeric Functions – 1

! provides examples for some of the commonly used numeric functions.

option nolet

x = -4

y = 2

z=5.63 ! These three lines provided values that will be used below

print "x="; x ; "abs(x)="; abs(x) ! absolute value

print "y="; y ; "exp(y)="; exp(y) ! exponent function

print "y="; y ; "sqr(y)="; sqr(y) ! square root

print "y="; y ; "log(y)="; log(y) ! natural logarithm

print "y="; y ; "log10(y)="; log10(y) ! base 10 logarithm

print "z="; z ; "int(z)="; int(z) ! integer

print "z="; z ; "round(z)="; round(z) ! round to an integer

print "z="; z ; "round(z,1)="; round(z,1) ! round to 1 decimal place

! (the 1 can be replaced by any number, if you need more)

print "x="; x ; "y="; y ; "sgn(x)="; sgn(x) ; "sgn(y)="; sgn(y)

! what's the sign of a number: negative or positive?

print "x="; x ; "y="; y ; "min(x,y)="; min(x,y) ! find which one is smaller

print "x="; x ; "y="; y ; "max(x,y)="; max(x,y) ! find which one is larger

print "z="; z ; "y="; y ; "mod(z,y)="; mod(z,y) ! find remainder when z is divided by y

end

8. Random number generator

A very important function for our applications is rnd, a function that generates a "random" number between 0 and 1.

8a. Simulate events occurring at certain probabilities

Let us assume that the probability of finding a food item during 1 minute of foraging is 0.33 and that each minute's foraging is independent of previous events ; let's write a program that will simulate how many food items are expected to be found in a 30 minutes-long foraging bout. Every time that the program encounters the function rnd, it will seek and obtain a new "random" number. The sequence of numbers that we will get by the **rnd** command is not truly random, but for most applications, they are independent enough to be considered random.

! Program Random function – 1

option nolet

foragend=30 ! defining the time limit of foraging

items=0 ! initializing to 0 the number of collected food items

for i = 1 to foragend

```

    if rnd <0.33 then items = items+1
next timeunit

print "The forager collected "; items; "items during"; foragend; "time
units."

end

```

We are likely to have about 33% of the generated numbers that fit this condition, so we should get "about" 10 items. Of course, in a random process you don't get exactly the expected number, and we are likely to see some deviation. If we will run this little program again and again, we will always get the same total, because by default, the sequence of numbers generated is always the same. If we want to change this default, we have to add the statement

randomize

in a separate line, near the beginning of the program (before the loop starts). Each run will be different now, and some of them are likely to produce different results. The average of many such runs is expected to be 10.

This program may seem a bit unnecessary in this case because you can guess the expected result without running it. But here is an example where simulating the probabilities may be faster than calculating them:

Let us assume that our forager has energy reserves equivalent to the content of 3 food items. Lets assume further that for each minute of foraging the forager incurs an energetic cost equivalent to 0.4 of the energetic value of a food item. If the forager reaches a level of energy reserves = 0, it dies.

Based on the program Random function – 1, write a flowchart, and then a program that will simulate 100 such runs, and will calculate what is the probability of starvation for this forager. (each run includes our 30 minutes as before, only we want it to be repeated for 100 times). Then change energy expenditure to 0.5 and see how it affects the results.

Print the program, and add on the same page the final results you received (by pen or pencil; there is no need to save the results on a file). Hand it in with your lab assignment.

Hopefully, you can see here the advantage of simulating stochastic events.

8b. Random selection of events or items

Random number generators can be used to select randomly one item out of several available.

For example, assume that we have a forager who has an equal probability to find one of 4 types of food, each with a different energetic value. Here is a small program that will find out which food item is found.

Add early in the program Random function – 1 the following statement:

Dim ntype(10)

Replace the loop in that program with the following:

(Note that in TrueBasic we can use <= as an expression for "small or equal to".)

```

! initialize the number of collected items from each food type
for f= 1 to 4 ! cycle over the four food types
ntype(f)=0 ! initially – none of them was collected. ntype(f) means number of items of type f

```

```

for timeunit = 1 to foragend
    selector = rnd !in each loop, a new random number is assigned to a variable named
    selector
    if selector <= 0.25 then
        f=1 ! the value of the selector that we got determines food type
    else if selector >0.25 and selector <= 0.5 then
        f=2
    else if selector >0.5 and selector <= 0.75 then
        f=3
    else ! which is equivalent here to saying if selector >0.75 then
        f=4
    end if
ntype(f)=ntype(f)+1 ! the number of items from the chosen type is incremented by 1

next timeunit

! Print the number of collected items from each food type
for f= 1 to 4 ! cycle over the four food types
print ntype(f); "items of type"; f
next f

```

Have the forager go out and forage 3 times, to see the variation you get in the results. There are at least two different ways to do this correctly. Do one of them, but try to think what would be the other.

8c. Random selection of equally probable events or items – the easy way

When the probabilities of the events happen to be equal, as in the previous example, there is a much shorter trick to find f; you replace the 10 indented lines above with the following line:

```
f= int(rnd*4)+1
```

Make sure you remember from section 7. above what the function int(...) means.

If you are not sure why this magic line is doing the whole complex operation that it replaced, try a few possible values of rnd (remember it is always between 0 and 1) either with pencil and paper, or on a small calculator, until you are convinced.

If so, why did we go through the previous if-based method? It is still necessary when you have unequal probabilities. Modify the program in section 8b for probabilities of 0.4, 0.3, 0.2 and 0.1 for food items 1,2,3 and 4, respectively.

9. User defined functions

TrueBasic lets you define your own functions; once they are defined, they can be used exactly as the built-in functions. A definition of a function can be one line long, such as:

```
def root4(x) = x^0.25
```

here we calculate the 4th root of x. After defining this at the beginning of a program, anytime that we will write

```
number=81 ! number is the name of the variable
a=root4(number)
```

our program will take the 4th root of 81, and will evaluate the expression as a=3.

We can define functions with more than one argument in the parenthesis:

```
def flexiroot(a,x) = x^(1/a)
```

Which gives us a flexible root – not only the 4th, but any other type that we will want: to have the 3rd and the square root of 64, for example, we will write

```
number=64  
b=flexiroot(3,number)  
c=flexiroot(2,number)
```

The definition of a function is not limited to one line, and can be quite complex, if needed. In this case we will have to add a statement that marks the end of the definition:

```
def chop(x) ! a function that chops the value of x to be between an upper and a lower  
boundary  
if x>xmax then chop=xmax  
if x<=xmax then chop=x  
if x<xmin then chop=xmin  
end def
```

9a.

Use the program that you wrote and submitted as an assignment at the end of the last lab; use the version where the energy expenditure was 0.4 items per time unit. Somewhere in the beginning, add to it a section that defines gut capacity, by using a function similar to the chop function on the number of items collected (set the minimum capacity to 0, and the maximum to 7 items). Then modify the finding food section, so that the number of items will never exceed the capacity, by using the function you defined.

See how it affects the proportion or the number of animals that die of starvation.

9b.

Now run the same program, but first you have to modify it to show you the effect of different maximum gut capacities - from 7 to 1. Prepare the output so that it CAN be easily plotted by Excel. (no need to plot it, just prepare it in the right format). Run this program twice, to see the effects of stochasticity on the results, and submit the two lists of results.

10. Subroutines

The end result of a function is always a single value that is returned by the computer, and in the previous case we called it chop(x), or before that flexiroot(a,x). Even though this last function had 2 arguments in the parenthesis, the function resulted in one value - flexiroot, which depends on a and on x.

Unlike functions, subroutines may return as many variables as necessary, thus subroutines are often used when a calculation returns more than one result.

Whenever you need the subroutine to do some calculations for you, you call the subroutine, and after performing the calculations, it will return to the same place, right after the call statement (you will soon see an example below).

Example for a subroutine:

Imagine that in the last program that you wrote for 9b, you wanted not only to limit the gut content after every food item was found, but at the same time you wanted also to keep a record of how much of the stomach was still empty. We could write a subroutine that will perform both the function chop and monitoring the empty space in

the gut, and will return two values - the updated amount of items, and the empty space in the guts.

In the program, you should have a line that calculates whether a food item was found:

```
if rnd <0.33 then items = items+1
```

Right after that we should call the subroutine by the call command:

```
call guts(items, space)
```

This means: call a subroutine named guts, which has two arguments that will serve as its input and output, if needed (their names are items and space).

Then we should define the subroutine somewhere in the program (Unlike functions, that need to be defined before they are used, subroutine definition can be anywhere; some prefer to place them just before the **end** statement, but subroutines can be anywhere in the program.

In the definition of the subroutine, we must mention the same number of arguments as in the call command (two in our case); they don't have to be with the same name - the first one will correspond to the first, and the second, to the second:

```
sub guts(x, space)  
if x>xmax then x=xmax  
if x<=xmax then x=x  
if x<xmin then x=xmin
```

```
space=xmax-x ! here we check how much empty space we have
```

```
end sub
```

Why would we want to use parameters with a different name in the **call** and in the **sub** statements? Sometimes we copy subroutines from other programs we wrote for different purposes; if the structure is right, we can use the old subroutine as-is and keep the names as they were before; all we have to do, is transfer the variable names as we did in the example between the **call** and the **sub** statement. Another case where you would do this name exchange, is when you **call** a subroutine at several places in your program, and in different places it requires different variable names.

Note that you can always write a program without any subroutines - you simply put the commands in the body of the program. However, they contribute to the organization of the program, and they can save a lot of writing, when you use them in cases where you need the same operation done in several places.

11. Programming tips

11a. Stages for writing a computer program

1. Decide what you want the program to do
2. See if you can start with a simplified version of the problem
3. Plan the stages of execution in a pseudocode, or a flow chart
4. Choose how you want the input and output
5. Check if you have an older problem that can be used as a base for program.
6. Start writing the program in stages, and check/debug each stage before next one.
7. Add remarks in the program to remind you what you meant in different stages
8. Check if you can improve the efficiency of the program, the clarity, etc.
9. Increase the sophistication of the program - from the simplified problem you wanted to solve.

11b. Tips and hints for good programming practices

1. Always use a pseudocode or a flowchart when planning your program
2. Use "structured programming": your program should be made of separate segments that you can easily recognize where they start and where they end. Use sub-routines and functions whenever they make sense.
3. Try to concentrate all the DIM statements in one area, all the DIM parameter values in one place etc.
4. Use indentation for loops and for IF-statements (the best is to use the FORMAT command in the appropriate menu), to see that they are done properly.
5. Do not use complicated formulas - break them down to several sub-separate lines.
6. When you have parameters in your equations - give them a name and value somewhere at the beginning of the program. Avoid using numbers in the equations themselves. This will make it easy to change the values if needed.
7. Initialize counters and totals, even if they are equal to 0.
8. If you are dividing by a variable or an expression, make sure the denominator is not 0. If they can be 0 - find a solution that will not cause a problem (e.g., with a statement such as "if variable = 0 then variable = 1")
9. Do not use GOTO commands, but other ways to EXIT out of loops (e.g., FOR or EXIT DO)
10. When you are introducing changes in a program that already runs, make a copy of the older version under a different name - in case you run into difficulties finding the bug, you will be happy to go back to the older version that runs.

11. When debugging, try to use temporary PRINT lines or the "command and ask it to print parameter values to check that the program is expect it to do.
12. Don't get stuck on one bug toooo much time - you should either fi systematic search, or avoid it by going back to a previous versic somebody to help you - they might be able to look at the program see it for you.

Good Luck! (you might need it...)