# Bioinformatics and Bioengineering Summer Institute (2003)

Position-specific scoring matrices to search for repeated sequences

Implementing a PSSM program

## I. Arrays

You've resolved to find all repeated sequences in the genome of *Nostoc* and have decided that a PSSM is the way to go. You've collected all the sequences you can (see Notes from Monday), and now you're ready for a program that will make a PSSM out of the collection and run the *Nostoc* genome through it, looking for those sequences that best match the repeated sequence you already have. Fortunately, the guy in the next lab knows someone who picked up a PSSM program from somewhere, so it should be pretty good.

You get the program and run it... no, take a look at it first, a program entitled PSSM.tru (in fact, why not open it up now?). The file begins with an explanation of what it does. It seems to do what you want, which is good. Skimming through the program, you see that it uses functions. Since you're acquainted with these, they don't come as a shock. For example, you see a few declared in the LIBRARIES and DECLARATIONS block, and later on (in the Main Program) you see one of them, Inverse$, used:

```
CALL Analyze(Inverse$(genome$))
```

From your knowledge of functions, you can guess what this might mean. A function takes the value within the parentheses, works with it, and then replaces itself with the value it comes up with. So the statement probably means:

`CALL Analyze(`*inverted sequence of genome sequence*`)`

OK, moving on to the first subroutine (`Get_motifs`), however, you're stopped by what looks like a very peculiar use of a function:

```
LET motif$(motif) = field$(2)
```

You can't guess what these functions mean, but presumably `field$` takes the value 2 and replaces itself with something appropriate, and likewise motif$ takes the value contained in the variable `motif` and replaces itself with something else,... but then the statement makes no sense! How do you put one value (returned by `field$(2)`) into another value (returned by `motif$(motif)`)? You can put values into ***variables*** (e.g. `LET x = 2`), but putting values into ***values*** (e.g. `LET 3 = 2`) is nonsense!

You pursue this mystery, looking for a definition of the function `motif$` that might explain what it does, but nowhere in the program is there a `FUNCTION motif$` statement, nor is there a `DECLARE FUNCTION motif$` statement in the LIBRARIES and DECLARATIONS block. What kind of function is this? Investigating further, you find two kinds of statements you've never heard of before:

```
DIM motif$(1)      ! Array contains set of motifs
```
[in VARIABLES block]

and

```
MAT REDIM motif$(motif) ! Increase size of arrays…
```
[in `Get_motif`]

You seem to have run into a different beast altogether: not a function but an "array".

**SQ1. How does an array seem to differ from a function?**

### I.A. What are arrays?

Suppose you wanted for some reason to translated numbers into their English representations. You might then set up the following variables:

```
LET one_in_English$ = "one"
LET two_in_English$ = "two"
LET three_in_English$ = "three"
```

(Note that the variables all end in $, signifying that they store text, not numbers). Then, you could use these variables in the following way (Program **A**):

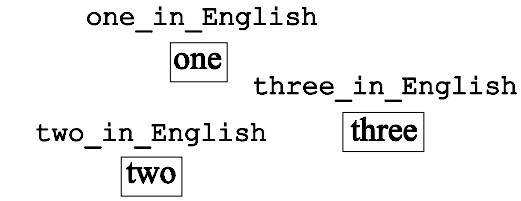
```
IF digit = 1 THEN
   PRINT one_in_English$
ELSE IF digit = 2 THEN
   PRINT two_in_English$
ELSE IF digit = 3 THEN ...
```

Clearly this would be a very tiring program to write! The problem is that you've set up ten separate boxes, and you have to decide by tedious logic which box you need at a specific time. Suppose instead that you set up a linked collection of boxes (Fig. 1). Then you could capture all of Program **A** by a single statement (Program **B**):

```
PRINT digit_in_English$(digit)
```

(where the number in parentheses indicates the box you want to access). This is quite an economy! Collections of boxes of this sort are called ***arrays***, but you could also think of them as tables. To exploit the utility of arrays, you need to tell True Basic three things:

Program **A**




Program **B**

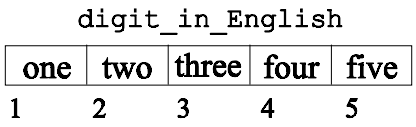



**Fig. 1: Organization of variables in two programs.** Program **A** stores words in separate variables. Program **B** stores the same words in a single array.

1. The name that will be interpreted as an array (so as not to confuse it with a function)
2. The type of information that will be stored within it (numeric or text)
3. The number of boxes to set up

All of this is accomplished by `DIM` statements (`DIM` is short for "dimension"). The following statement identifies the variable digit_in_English$ as an array (because it is defined by a DIM statement and not a DECLARE FUNCTION statement), specifies that it will store text (because it ends with a $), and sets aside 9 boxes:

```
DIM digit_in_English$(9)
```

You can stock the array in the usual (tedious) way:

```
LET digit_in_English(1) = "one"
LET digit_in_English(2) = "two"
```

and so forth.

What about zero? There's no zeroth box defined. By default, counting of array elements begins with one, but you can start with any number you like. Here are some examples:

```
DIM digit_in_English(0:9)          ! First box is labeled 0, last is labeled 9
DIM digit_in_English(1:9)           ! Equivalent to DIM digit_in_English(9)
DIM rainfall_in_Richmond(1999:2003)
```

The range of numbers must be contiguous, however.

**SQ2. Define an array variable that stores the number of instances that planes arrived at their destinations times relative to the scheduled time. Presume that the range of possible times in minutes is 60 minutes early to 120 minutes late.**

### I.B. Multidimensional arrays

Arrays can be used to organize more complicated information. Recall that the first step in construction of a PSSM is to count the different nucleotides at each position of a set of aligned sequences. The sequences themselves can be stored in an array (PSSM.tru stores them in the array called `motif$`). The count of nucleotides requires a box for each nucleotide and at each position (Fig. 2), which can be captured in a two-dimensional array. Two dimensional arrays can be visualized as tables, as shown (it's our choice which dimension we treat as columns and which as rows).

However, we needn't be limited by our ability to visualize tables. True Basic lets you define arrays with multiple dimensions. For example, the following 3-dimensional array could store the number of mutational changes in a set of sequences from several organisms, at different positions, to different nucleotides:

```
DIM mutations_in_motif(7,14,4)     ! 7 organisms, 14 positions, 4 nucleotides
```

**SQ3. How many boxes will be set aside for `mutations_in_motif` by the `DIM` statement above?**

**`motif$(5)`**

| motif | | |
|---|---|---|
| **1** | GTACAGTCTGTTAC |
| **2** | GTAGGCTCTGTTAC |
| **3** | GTAACAAAGACTAC |
| **4** | GTAACATAAGCTAC |
| **5** | GTAACATGAGATAC |

**`nucleotide_count(14,4)`**

| nuc | position | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **1** (A) | 0 | 0 | 5 | 3 | 1 | 3 | 1 | 2 | 2 | 1 | 1 | 0 | 5 | 0 |
| | **2** (C) | 0 | 0 | 0 | 1 | 3 | 1 | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 5 |
| | **3** (G) | 5 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 4 | 0 | 0 | 0 | 0 |
| | **4** (T) | 0 | 5 | 0 | 0 | 0 | 0 | 4 | 0 | 2 | 0 | 2 | 5 | 0 | 0 |

**Fig 2: Possible representation of two arrays.**

### I.C. Arrays of variable extents

If the `DIM` statement sets aside a specified number of boxes for an array, what do you do when you don't know how big the array should be? The program PSSM.tru reads in a list of sequences from which the PSSM will be constructed. But the program doesn't know how many sequences will be read (and neither do you – do you want to sit there ***counting*** a few hundred sequences?). What then?

There are three solutions:

1. Make up a number that's certain to be bigger than any conceivable number of sequences the program will ever need to read. For example:

   ```
   DIM motif$(1000)
   ```

   This has the advantage of simplicity, however there is a danger that the future will surprise you and the number you choose won't be high enough. You can't solve this problem by choosing a ridiculously high number (like `DIM motif$(10E100)`), because True Basic will dutifully attempt to set aside that many boxes, with dire results. Also, some operations (like some sorting routines) requires that the extent of the array accurately reflect what is actually used.

2. Follow the first strategy, but after all the sequences are read in, adjust the extent to make it reflect reality:

   ```
   DIM motif$(1000)
   DO WHILE MORE #motif_file          ! Go through the loop while there's still data
      [read a sequence into motif$]
      LET total_sequences = total_sequences + 1       ! count as you go
   LOOP
   MAT REDIM motif$(total_sequences)
   ```

   The latter statement redimensions the array `motif$` so that the total number of boxes set aside is given by the variable `total_sequences`.

3. Expand the array as you go:

   ```
   DIM motif$(1)      ! This statement just gets the array on the books with a minimal size
   DO WHILE MORE #motif_file
      LET total_sequences = total_sequences + 1        ! Count as you go
      MAT REDIM motif$(total_sequences)        ! Increases size of array by 1
      [read a sequence into motif$]
   LOOP
   ```

   The advantage of this method is that the array is always as big as need by the present data.

**SQ4. Why is the extent of motif$ set to 1 in solution #3?**

### I.D. Array operations and functions

Just as the statement to redimension an array began with MAT, so do array operations in general. Thus, the assignment of the values of one array to another is:

MAT array1 = array2

True Basic was written by two professors at Dartmouth, and as you might imagine, one of their big interests was academic use of the language. As a result, the language is very good for matrix operations (as would be useful in Linear Algebra). Built-in functions support things like matrix multiplication, inversion, transposition, and many others.

## II. Common irritations and quick fixes

You still haven't run the program, so resist no further and go for it.

You probably got a blank screen (note *Finished. Click mouse or press any key*). However, if you click out of the output window or go to the command window, you'll find the unwelcome news that an apparently critical file was not found.[1] Statement 121 is implicated, so scroll down to it (note that there's a very useful indicator at the bottom of the window telling you which line and character the cursor is sitting on). Statement 121 says:

```
OPEN #motif_file: NAME motif_file$, ORG text, ACCESS input, CREATE old
```

Add this knowledge to the error message, and you have to conclude that there is a file problem somehow. What can you find out about `motif_file` and `motif_file$`? To help you in this True Basic provides the locate command. Go to the command window and type:

```
locate motif_file
```

And then

```
locate motif_file$
```

**SQ5. What do you find is the value set for motif_file$? What problem do you see? What would be a better value for motif_file$ [hint: check the web page for Wednesday]. Make the appropriate change in the program.**

Try running the program again. Still no good, I bet.

**SQ6. What line number is the error this time? Go there and identify the offending statement. What section of the program is the statement in?**

If the program stopped at statement 131, then the motif file was evidently opened properly but we're still stuck in the subroutine called Get_motifs. Simple minded as I am, I would suppose that there's a problem in getting the motifs, i.e. reading the motif file. Why is that?

At this point, one is tempted to read the instructions.

**SQ7. From the comments at the beginning of the Get_motifs subroutine and the comments at the very beginning of the file, what does the program expect in the way of motifs?**

**SQ8. Are you providing the program with what it expects? (can't tell without looking)**

**SQ9. Find a way to fix the problem.**

In such situations there are two major classes of solutions: Change the input file or change the routine that reads it. You can go either way with this one. Take your pick.

[1] You may get an irritating white box in the source code window. To get rid of it, go to the error window and x out of it.

**SQ10. Try doing it the *other* way.**

Note that you don't have to understand everything about the program. Just from the description of what the program is ***trying*** to do, you can make a pretty good guess how to fix the problem.

Presuming you got past this problem, run it again… [crash]. Good grief! You begin to wonder if this program ***ever*** worked! What's the error this time? You're told (again after clicking out of the result window) that the program tried to take the log of zero or a negative number in statement 183. Going there, you find that there are two possible logs that could be at fault. Which is it?

To answer that question, insert just prior to statement 183 the following:

```
PRINT p, nucleotide, log_q(p,nucleotide),numerator, denominator
```

This will print out the values of all variables in the statement that caused the crash. Run the program again. Of course it crashed, but that's no surprise. The difference is that it told you what were its final thoughts. By matching the output with the list of variables printed (e.g. on the last line, p=1, nucleotide=2,…), you can see what the problem is.[2]

**SQ11. What values caused the program to attempt to calculate an illegal log?**

Use the same `locate` command to find where this variable is set and to what (or just scan the immediate vicinity to find out). You'll see that the variable is set to something called `nucleotide_count(p,nucleotide)`. This is an array of values related to nucleotide counts.

**SQ12. Is it possible for a nucleotide count to be zero? If you're unsure on this point, go back to the table of nucleotide counts we constructed on Monday.**

How in the world did the original programmer avoid this problem? Maybe he/she messed with the data by hand, replacing zeros with some nonzero number. Maybe the data set was so massive that there were no data values (every nucleotide will be present at any position, given a large enough data set). Who knows? What can YOU do about it?

**SQ13. Nothing's illegal about a taking the log of a number that's *almost* zero. Add a statement that checks to see if the variable is zero and, if so, changes it to 1E-47 (1 times 10 to the –47th).**

If you've accomplished this, then this time the program may run.

## III. Improving the program

The program may now work, but it's certainly not giving the answers you expected.

**SQ14. What sequences *did* you expect? (It might help to recall the scientific problem from Monday's notes)**

Here's another good time to read the instructions. How is it constructing the PSSM? Go to the Main Program and try to find the routine that does this. There's nothing named `Construct_PSSM`, but there's a lot of actions that should look to you as if they're involved in the construction. Go to these routines and read how they describe themselves.

---

[2] Programmers: Yes, True Basic includes a programming environment that allows you to monitor the contents of variables without adding extraneous statements.

**SQ15. How does the program calculate the PSSM? How does it differ from how you would *want* to calculate a PSSM?**

**SQ16. Find the section of the program where the difference ought to be and see if you can change it to do what you think is right.**

***Note well! In running this data myself I noticed that there is a mistake in the program. So don't expect to get reasonable results even after you put in all the fixes and improvements. We'll discuss the final resolution tomorrow.***