Mathematics 152 — Spring 1999

Notes on the course calculator

1. The calculator VC

The web page

`http://gamba.math.ubc.ca/coursedoc/math152/docs/ca.html`

contains a generic version of the calculator VC and some related material. More precisely, on this page are (1) the calculator itself, which is a Java applet running there, and (2) instructions for using it. This note reproduces part of these instructions. This calculator will play a role in most if not all of the labs in several courses, and you will be able to access it from anywhere on the Internet to help you with nasty calculations on ordinary homework assignments (even perhaps, in other courses). From standpoint of the Mathematics Department, it means that we are able to assume that all of you have access to the same simple yet sufficiently capable calculator capabilities.

The calculator is programmable and relatively versatile, and imitates the well known and beautifully designed (but pricey) calculators from Hewlett-Packard in that it follows the conventions of Reverse Polish Notation (RPN). The calculator is called **VC** to stand for Vector Calculator.

The calculator is not very efficient and indeed rather slow. It has in fact been designed intentionally to be used in an undergraduate computer laboratory where a large number of people are working simultaneously, and where speed is less important than politeness. But in any event, this calculator is intended to demonstrate through simple examples how mathematical calculations can be made automatically, not to make complicated practical calculations. Have patience with it. Even if you already have your own programmable calculator, you will find before the end of the term that this one can handle a wider range of problems in linear algebra, and handle more efficiently many of the ones yours can.

Warning! If you reload a page containing one of these calculators in Netscape, editing in the calculator window will likely not work right. Resizing has exactly the same effect. (Can anyone explain to me the reason or the fix for this?) Therefore we suggest strongly:

- **Do not reload the page.**
- **Do not resize it after an initial adjustment.**

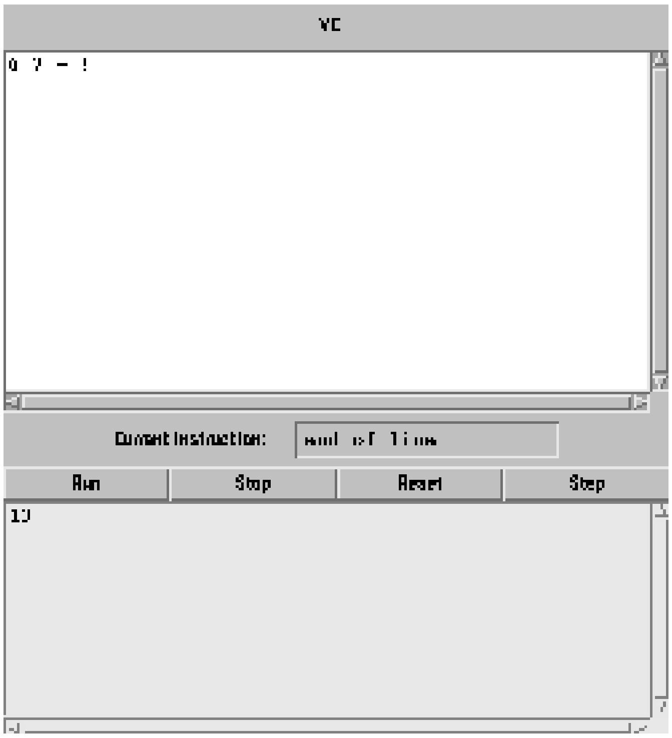
Also, if you exit this page and later return, your edited text may not still be there.

We have tried very hard to make this calculator foolproof and bug-free, but of course we cannot guarantee anything. *If you encounter bizarre behaviour of any kind, please report it to me, explaining in as much detail as you can what the circumstances were.*

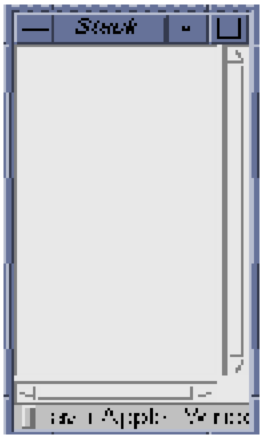
2. Details

What the calculator looks like is shown in the figure below. You type source code into the top window. Pressing **Run** will run the program and display its output in the lower window. Pressing **Stop** will halt the program while it is running, pressing **Reset** will reinitialize it, and pressing **Step** will do one step of calculation. This one-step button will be useful when you are learning how to use the calculator, but probably not later on. The text field in the middle displays the current instruction being processed. What that current instruction might not always be what you expect, since various instructions can change the flow of the calculation in non-intuitive ways. *There is one extra quirk to be patient with. The calculator takes a while to load. After a while it says it has loaded. If you then press a button, the calculator seems to freeze up. Fear not! In a few seconds all will be OK.*

Here is what the calculator itself looks like:



When the calculator come up, there is also a separate window labelled **Stack** floating somewhere on the computer screen. In this, the calculator's stack (explained shortly) is displayed (upside down). You cannot get rid of the stack window except by leaving the calculator page, but you can close it down, and the calculator will run more efficiently if you do so.



3. **How to use the calculator**

Ihis calculator is a **stack-based calculator**. The **stack** is an array of indefinite length into which data is fed, and on which all operations are performed. Generally, you have access only to the top (or growing end) of the stack (like a stack of dishes in a cafeteria). Commands are applied to top items. For example, `6 7 +` puts 6, then 7, on the stack and then replaces them by their sum. Roughly speaking, the data you can use in this calculator are integers, real numbers, vectors, and strings. You can apply built-in procedures or define your own. You can also use both global and local variables.

Normally, results are not displayed when calculated. If you want to display them, you can type **=**, which will display the item at the top of the stack without removing it. If you use **!** instead you will both display and remove it. (So there are two ways to output the item at the top of the stack: **!** which is destructive and **=** which is non-destructive.) Thus you would type `6 7 + =` to calculate 6+7=13 and display the result, leaving it on the stack.

This 'backwards' behaviour may seem peculiar at first, but it is extremely efficient in a chain of complicated calculations, and you should get comfortable with it in time.

4. Examples

Task: Calculate $6 + (7 * 8)$**.** It is in fact an interesting problem to figure out how to evaluate a general algebraic expression in RPN terms. The general rule is to **lay down the data left to right**, but **apply operators from the inside out**, or from the lowest levels up. Here the inner expression is (7 * 8), so multiplication is the first operator we apply.

Input:

```
6 7 8 * + !
```

Output:

```
62
```

Remark: What is going on here inside the stack? First we enter 6, 7, and 8. At this point the stack has three items on it. Then we replace 7 and 8 by 7*8=56, leaving 6 and 56 on the stack. Finally, we replace 6 and 56 by 6+56 = 62 and display the result destructively. At the end, the stack is empty. Here is a sequence of pictures of the stack as the calculation proceeds:

```
6
6 7
6 7 8
6 56
62
```

Task: Calculate the sum of vectors [1 2] and [1 -3].

Input:

```
[ 1 2 ][ 1 -3 ] + =
```

Output:

```
[ 2 -1 ]
```

Remark: Here the result is left on the stack.

Task: Define variables $x = 6$**,** $y = 7$**, set** $z = x + y$**, and display** z**.**

Input:

```
6 @x def
7 @y def
x y + @z def
z !
```

Output:

```
13
```

Remark: The term **@x** is the **name** of the varaible **x**, which is not at all the same as the variable **x**. If the calculator encounters **x**, it will substitute the value of **x**. But **@x** is much like a pointer in some programming languages. In the calculator, the names of varaibles are mostly used as they are here, to assign values to the variable. **The only way to assign a value to a variable is to define a value for its name.**

Task: Calculate and display the sum of the first 10 squares 1 + 4 + 9 + ...

Input:

```
0 @n def
0 @s def
10 {
1 n + @n def
n dup * s + @s def
} repeat
s !
```

Output:

```
385
```

Task: Construct a procedure called `average` which has just one argument, a vector, and returns the average of its coordinates.

Input:

```
{
  @v def
  v dim @n def
  0 @s def
  0 @i def
  n {
    v(i) s + @s def
    i 1 + @i def
  } repeat
  s n /
} @average def

# sample usage:
[ 4 5 6 7 ] average =
```

Remark: This is not as efficient as it might be. Cleverer stack manipulations could do better. Note that $v(i)$ is the i-th coordinate of v.

Task: Calculate numerically the integral of $y = e^{-x^2}$ from 0 to 1 by applying the trapezoidal rule with 10 intervals.

Input:

```
# Define the function to be integrated
# Here f(x) = exp(-x*x)
{
   dup * -1 * exp
} @f def

# Do the sums for the trapezoidal rule
# Each term = (f(x) + f(x+h))*0.5*h
10 @N def
0 @x def
1 N / @h def
0 @s def

N {
  x f
  x h + @x def
  x f + 0.5 *
  h *
  s + @s def
} repeat

# display the result
s !
```

Output:

```
 0.7462
```

Remark: This is more complicated than other examples. First we define the variable f to be the **procedure** or function which takes the variable x off the stack and then places e^{-x^2} on the stack. Just to be sure you get the point, we'll make it more explicit: **you can define variables to be equal to procedures as well as ordinary constants**. And almost always functions defined in the calculator will do something like this one—remove some items on the stack as its arguments, and place something on the stack as its return value. Incidentally, the command **dup** used here just makes an extra copy of what is on top of the stack. Also, this function is not as efficient as it might be. With a little care you can get away with only one function evaluation in each loop.

Task: Construct a function called `length` which takes a single argument which is a vector, and returns its length.

Left as an exercise.

Task: Construct a function called `angleBetween` which takes two arguments which are vectors and returns the angle between them.

I'll leave this as an exercise, too. It will use ***** to calculate the dot product of two vectors, the **length** function from the previous exercise, and the function **acos** (inverse cosine). You'll have to recall a formula from linear algebra relating the dot product to angles.

5. Description of basic commands and operations

- **+** replaces the previous two items on the stack by their sum. Can add integers, real numbers, or vectors. Thus

```
7 6 +
```

calculates 7+6=13.

+ can also be used to build **strings**. A string is a phrase inside quotes. The sum of a string and any item tacks on a string representation of the item to the original string. Thus

```
"x = " 3 +
```

produces the string "x = 3". Using this feature is good for explaining in output exactly what displayed data means.

• **–** replaces the previous two items on the stack by their difference. Can subtract integers, real numbers, or vectors. Thus

```
7 6 -
```

calculates 7-6=1.

• ***** replaces the previous two items on the stack by their product. Can multiply integers or real numbers. Also calculates the dot product of two vectors, or the scalar product of a vector and a scalar. Thus

```
7 6 *
```

calculates 7*6=42.

• / replaces the previous two items on the stack by their quotient. Can divide integers or real numbers. Thus

```
14 2 /
```

calculates 14/2 = 7.

• **fix** requires a non-negative integer on the stack. It sets the number of decimal figures displayed, and does not leave anything on the stack. Thus

```
5 fix
4.0 =
```

displays `4.00000`.

• **def** defines the previous item to be the item below it. The previous item must be a **variable name** such as `@x` or `@longVariableName`. A variable name is what you get by putting `@` before the variable itself. Thus `x` is a variable and `@x` is its name. (We have to distinguish between the variable and its name because the results of putting them in a program are very different. When the calculator comes across the variable, it attempts to make a substitution. This is similar to the difference between a variable and a pointer to the variable in some programming languages.) Thus

```
5 @x0 def
```

defines the variable `x0` to be 5. Subsequent occurrences of `x0` (with some exceptions to be explained some other time) will be replaced by 5.

If v is a vector then `4 @v(i) def` defines $v(i)$ to be 5.

• **dim** replaces a vector by its dimension.

• **cross** replaces the previous two items by their cross product, if they are both three dimensional vectors.

• **floor** replaces a number by the largest integer less than or equal to it. Thus 6.7 gets replaced by 6, while -6.7 gets replaced by -7.

sqrt replaces the previous item by its square root, if it is a non-negative number.

• **exp** replaces the previous item x by e^x. Similarly for **cos**, **acos**, **sin**, **log** (which is the natural log).

• **atan2** has two arguments y and x **in that order**, and returns the angle coordinate of the point (x, y). (This odd and unfortunate choice of the order in which x and y are written conforms with that of most programming languages.)

• ˆ has two arguments x and y, and returns x^y. Here either $x > 0$ or y must be an integer.

• **pi** is a constant equal to 3.14159 ...

• **dup** makes an extra copy of the item at the top of the stack.

• **pop** just removes the item at the top of the stack.

• **exch** swaps the top two items on the stack.

• **lt**, **le**, **gt**, **ge**, **eq** are tests on the previous two items, which should be numbers. The names stand for **less than**, **less than or equals to**, etc. The effect is to place either a **true** or a **false** on the stack.

• **ifelse** uses the top three items on the stack, which should be **true**/**false** and two procedures. If **true**, it executes the first procedure, while if **false** it executes the second.

• **repeat** can be used to perform loops. It requires an integer and a **procedure** immediately preceding it. A procedure is a sequence of instructions inside brackets **{** and **}**. Thus

```
1000
10 { 1 - = } repeat
```

will output

```
999
998
997
996
995
994
993
992
991
990
```

• **break** will break out of an enclosing loop. This should be used together with conditionals in order to halt a **repeat** loop. Thus the following program will print out only the numbers 10, 9, 8, 7, 6.

```
10 @x def
10 {
x 5 eq { break } { x = x 1 - @x def } ifelse
} repeat
```

• Any error will be signalled by displaying an error message. You should never ignore one of these messages. It is possible that it is caused by a bug in the program, in which case you should make a bug report.

6. **Obtaining a version for yourself**

You can run this calculator within most Internet browsers, but a more efficient version can be run on any computer with a Java interpreter installed. The calculator page contains a link to a file **rpn.zip** which you should download and unzip. It will unpack everything, including Java source files as well as class files, and a copy of the calculator's html file, into a directory **rpn** (and a few subdirectories). If Java is in your execution path and the directory above

rpn is in your Java class path, you can run the calculator through standard input in any UNIX terminal or MSDOS window by typing **java rpn.vc.vc**. Typing

```
java rpn.vc.vc x
```

will run the calculator with the file **x** as input. You can run the window version by running **appletviewer** on the file **ca.html** which is unpacked in the collection.