Mathematics 307—October 11, 1995

A vector calculator

In my section of this course I expect you to use a calculator program that runs on just about any computer, including PC's but perhaps not Macs. It will do all sorts of standard operations with numbers, but it will also implement certain vector and matrix operations. The base calculator is part of the public domain progarm `ghostscript`, which is an implementation of `PostScript`, and I have added a certain number of routines to it to make it easy to do the vector and matrix computations. There is one drawback to using `ghostscript`, namely that it deals with real numbers of only very limited precision, about 6 digits. Therefore it is not a very practical calculator. I wouldn't contemplate doing serious calculations with it. But the limited precision will turn out to be an advantage for teaching purposes; it will force us to think about how limited precision will affect what we are doing.

`PostScript` is a widely used programming language, although most computer users never see it—it is the language in which computers communicate with laser printers, at least the ones at the high end. It is capable of producing graphics and text of extremely high quality, although in this course most of you will probably never see that capability. Nonetheless, it is a fairly cleverly designed language capable of just about everything other programming languages are. It was designed, you might be amused to know, by a graduate student in mathematics at the University of Utah who quit graduate school to found Adobe.

You can program in complicated ways in `PostScript`, but for most purposes in this course the necessity will not arise. I have tried to write for you most of the complicated routines you will need. The advantage of using a computer in this course is that you will be able to deal with matrices of a size that would otherwise overwhelm you, and begin to look at real-life problems rather than the toy ones you would otherwise be restricted to. I should mention, however, that `PostScript` is by no means an orthodox programming language for our purposes, since it is not only of limited precision but also somewhat slow. Production programs are usually written in C or Fortran, and run faster by a factor of at least several hundred.

How to use it

`PostScript` has one feature that will bother you at first. Expressions are calculated in what will seem a backwards manner. If I want to calculate $4 + 3$ for example I type

$$4\ 3\ +$$

and will get the answer 7. In this it is like the calculators sold by Hewlett-Packard, as opposed to most of the rest. This way of handling expressions is called *Reverse Polish Notation* or RPN for short. There are several reasons why it is used in HP calculators, and also several but probably different reasons why it is used in `PostScript`, and yet other reasons why we are using it.

HP uses it because it uses fewer keystrokes than the usual method, and seems far more convenient once you get used to it. PostScript uses it because expressions can be calculated much more efficiently than with any other scheme of program interpretation. We are using it because it is far simpler for me to configure for vector algebra, and for you to use, than anything else available.

There are a number of technical things you will need to know to use the calculator efficiently. (1) *It is a stack-based calculator.* What this means is that whatever you type into the computer gets pushed into an array of stuff as though you were stacking trays in a cafeteria, and stuff that you last put on is the most accessible. (2) *Operations will generally affect the things at the top of the stack only.* Thus in the course of the calculation above the stack looks like thus:

4
4 3
7

In other words, data is entered on top of the stack, which grows as you do this, but the operation + replaces the top two elements by their sum. *Operations will generally remove their arguments and replace them by the result.* (3) *Results will not appear unless you request them to.* The actual appearance of your screen in the course of this calculation is this:

```
GS>4 3 add
GS<1>
```

The `<1>` here means that there is a single thing on the stack, but isn't telling you what it is. Using the macros I give you, you can then type `print` to see what is there:

```
GS<1>print
7
GS<1>
```

so the effect of `print` is to show you what is there without affecting it. If you type `print` when nothing is actually there you will get a string of very intimidating stuff:

```
GS>print
Error: /stackunderflow in dup
Operand stack:

Execution stack:
    %interp_exit --nostringval-- --nostringval-- --nostringval-- %loop_continue --
nostringval-- --nostringval-- false --nostringval-- --nostringval-- --nostringval--
--nostringval--
Dictionary stack:
   511/547 0/20 27/200
Current file position is 5
GS>
```

which you should learn to ignore.

`PostScript` has most of the usual calculator functions to do arithmetic. For example, to find the length of the vector $(3, 4)$ you can go

```
GS>3 3 mul 4 4 mul add sqrt
GS<1>print
5.0
GS<1>
```

and to finds its angle

```
GS>4 3 atan
GS<1>print
53.1301
GS<1>
```

Note that `PostScript` *works with degrees!*

To give you an idea of how the stack works, here is a picture of the stack as the first calculation proceeds:

3
3 3
9
9 4
9 4 4
9 16
25
5

Commands you will use

The first category is made up of arithmetic operations and functions built into `PostScript`. I write the stack left to right, bottom to top.

(•) `add` replaces a b by $a + b$.

(•) `sub` replaces a b by $a - b$.

(•) `mul` replaces a b by ab.

(•) `div` replaces a b by a/b. If $b = 0$ you will get a long error message.

```
GS>4 0 div
Error:  /undefinedresult in --div--
Operand stack:
    4  0
Execution stack:
     %interp_exit  --nostringval--  --nostringval--  --nostringval--  %loop_continue  --
nostringval--  --nostringval--  false  --nostringval--  --nostringval--  --nostringval--
Dictionary stack:
    511/547  0/20  25/200
Current file position is 7
GS<2>pstack
0 4
GS<2>
```

As you can see, after this error the original data 4 0 are left on the stack. (The command `pstack` shows you everything on the stack, whereas `print` just shows you what's on top. Note that it lists the stack from top on down.)

(•) `abs` replaces a by $|a|$.

(•) `neg` replaces a by $-a$.

(•) `sqrt` replaces a by $\sqrt{a}$; error if $a < 0$.

(•) `sin` replaces a by $\sin a$, interpreting a in degrees (after all, the founder of Adobe didn't get his Ph. D.).

(•) `cos` replaces a by $\cos a$, interpreting a in degrees.

(•) `atan` replaces a b by the angle (in degrees) that the vector (b, a) makes with respect to the positive x-axis. Note the reversed ordering.

(•) `exp` replaces a b by a^b. Keep in mind that $e = 2.718281828$ to more than enough accuracy.

(•) `ln` replaces a by $\log_e a$.

The next group of commands display and rearrange the stack.

(•) `==` shows you what's on top of the stack *but destroys it at the same time*.

(•) `dup` makes an extra copy of the stack top: a becomes a a. Thus `print` is equivalent to `dup ==`.

(•) `pstack` exhibits the whole stack without changing it.

```
GS>1 2 3 pstack
3
2
1
GS<3>
```

(•) `exch` exchanges the top two items: a b becomes b a.

(•) `n m roll` rolls around the top n items by a shift up of m:

```
GS>1 2 3 pstack
3
2
1
GS<3>3 1 roll
GS<3>pstack
2
1
3
GS<3>
```

(•) `clear` takes away everything on the stack.

```
GS<3>clear
GS>
```

The next group has to do with arrays. An array is written in square brackets, for example `[1 2 3]`. Its elements are numbered from 0 up. Items are retrieved from an array and placed into one by `get` and `put`.

(•) `get` replaces `[1 2 3] 1` by `2`.

(•) `[1 2 3] 1 4 put` puts `[1 4 3]` on the stack.

`PostScript` uses variables, although it may seem awkward at first. Assignments are made with the command `def`.

(•) `/X 3 def`

means that X is a variable and sets it to 3. The term `/X` here means that we are referring to X as a variable rather than to its current value. Variables can be almost anything. In particular **routines** in `PostScript` are defined by `def`. The operation I call `print` is defined by the line

```
/print {dup ==} def
```

You can read from files:

```
(homework) run
```

will run the file `homework` as a `PostScript` program. Using files rather than on-line typing will usually be more efficient, especially if errors are made.

In addition to the commands above, which are built into `PostScript`, I have added a few. They are stored in the file `calc.inc` so that in order to use them your calculations must first do

```
(calc.inc) run
```

A **vector** in my scheme is an array of numbers. A **matrix** is an array of vectors, interpreted as its rows.

(•) `vectoradd` replaces two vectors by their sum.

(•) `vectorsub` replaces two vectors u v by their difference $u - v$.

(•) `dotproduct` replaces two vectors by their dot product (a number).

(•) `matrixadd` replaces two vectors by their sum.

(•) `transpose` replaces a matrix by its transpose.

(•) `vectorscale` replaces v c by the scalar product cv.

(•) `matrixmul` replaces two matrices by their product.

(•) `matrixscale` replaces m c by the matrix cm.

(•) `elementswap` removes u i j from the stack, swapping u_i and u_j. It swaps the rows of matrices as well.

(•) `rowscale` removes m i c from the stack, replacing the i-th row m_i of m by cm_i.

(•) `rowsubscale` removes m i j c from the stack, replacing m_j by $m_j - cm_i$.

(•) `identity` replaces n by the $n \times n$ identity matrix.

(•) `tworotation` replaces θ by the matrix

$$[[\cos\theta \quad -\sin\theta] \quad [\sin\theta \quad \cos\theta]]$$

(•) `twodet` replaces the 2×2 matrix m by its determinant.

(•) `threedet` replaces the 3×3 matrix m by its determinant.

(•) `twoinverse` replaces the 2×2 matrix m by its inverse.

(•) `threeinverse` replaces the 3×3 matrix m by its inverse.

(•) `gauss` replaces a matrix by three other matrices by means of a version of Gaussian elimination. Exactly what happens is complicated and will be explained in class.

In many of these routines, sizes have to agree, and if they don't a lot of junk will come spilling out onto your screen:

```
GS>[1 2] [3 4 5] vectoradd
GS<1>clear
```

is OK but this is trouble:

```
GS>[3 4 5] [1 2] vectoradd
Error:  /rangecheck in --get--
Operand stack:
    -marktype-  4  6  5  [1 2]  2
Execution stack:
     %interp_exit  --nostringval--  --nostringval--  --nostringval--  %loop_continue  --
nostringval--  --nostringval--  false  --nostringval--  --nostringval--  --nostringval--  0
--nostringval--  %repeat_continue  --nostringval--  --nostringval--  --nostringval--
Dictionary stack:
    511/547  0/20  25/200  4/4
```

```
Current file position is 23
GS<6>
```

Error handling in various versions of `PostScript` behave rather differently, and none are very informative. Best to clear the stack and start over when errors appear, although in time you will get to know some kinds of simple errors well.

More routines will come along as the course goes on, such as `crossproduct`, `inverse`, `solve`, and some stuff for finding eigenvalues and eigenvectors.

Exercise. *Try to write down a sequence of steps that replace a $3D$ vector by its length.*

Exercise. *Calculate the length of the* 10 *dimensional vector* `[1 2 3 4 5 6 7 8 9 10]`.

Example. *If*

$$A = \begin{bmatrix} 1 & -2 \\ 2 & 1 \end{bmatrix}, \quad M_F = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

then we calculate and exhibit AM_FA^{-1} by the sequence

```
/A [[1 -2][2 1]] def
/MF [[1 0][0 -1]] def

A
MF matrixmul
A twoinverse
matrixmul
print
```

Exercise. *Calculate AM_FA^{-1} when*

$$A = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -1 & 1 \\ 1 & 0 & -2 \end{bmatrix}, \quad M_F = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Subroutines

I have said that you will not have to do any complicated programming with this calculator, but nonetheless I imagine you will want to write your own subroutines to make calculations more efficient. Subroutines in `PoscTscript` are essentially just a way of replacing a long and often repeated sequence of steps by a word or two. We have seen an example above:

```
/print { dup == } def
```

means that whenever you type `print` the computer will immediately replace it by the two commands `dup` `==` in succession—in other words, `print` becomes a single word for a two-word sequence. To see something a bit more complicated and more useful, let's write a routine to use in calculating cross-products—that is to say, it will replace two $3D$ vectors on the stack by their cross product.

So we expect to start out with a template

```
/crossprod { } def
```

and start to fill in between the brackets `{ }`. There are a few tricky items to handle, but you can just copy them without understanding everything about why you are doing them.

First of all, since operations nearly always remove their arguments from the stack, if you want to reuse data you must save it. So we begin our routine by naming the two vectors, say as U and V. It may happen that somewhere else in your calculations you already have vectors named U and V which you don't want to lose track of, so you want some **local variables** U and V. You make local variables in `PostScript` by using a temporary **dictionary**. This is done by two lines at the beginning and end of the routine:

```
/crossprod {
2 dict begin

end
} def
```

which means that you are using a temporary dictionary which will hold two items.

We then want to assign the variables U and V to be the vectors which are sitting on the stack when the routine begins. Thus at the beginning of the routine the stcak looks like this

```
[1 2 3][1 1 1]
```

We want to say something like `/V [1 1 1] def` but of course using what's at the top of the stack. This is done in two lines like this

```
/crossprod {
2 dict begin
/V exch def
/U exch def

end
} def
```

The only tricky thing is that we have to use `exch` to get the assignments in the right format, and note that since we place U and then V on the stack we define them in apparent reverse order. At the end of these definitions the two vectors have been removed from the stack.

Now we start to build up the cross product. It is going to be a vector so we surround the numbers we put on by brackets `[]`.

```
/crossprod {
2 dict begin
/V exch def
/U exch def
[

]
end
} def
```

Now we calculate the numbers themselves. Recall that the arrays U and V are (u_0, u_1, u_2) and (v_0, v_1, v_2) since the numbering in arrays starts at 0. The first entry in the cross product is therefore $u_1v_2 - u_2v_1$ so we write

```
/crossprod {
2 dict begin
/V exch def
/U exch def
[
U 1 get
V 2 get
mul
U 2 get
V 1 get
mul
sub

]
end
} def
```

and to get all the entries, remembering to reverse sign in the middle:

```
/crossprod {
2 dict begin
/V exch def
/U exch def
[
U 1 get
V 2 get
mul
U 2 get
V 1 get
mul
sub

U 0 get
V 2 get
mul
U 2 get
V 0 get
mul
sub
neg

U 0 get
V 1 get
U 1 get
V 0 get
mul

]
end
} def
```

Finally, we can add some comments to make our routine easier to read—not by the machine, but by us! Comments are begun by a `%`.

```
% this routine replaces two 3D vectors on the stack by their cross product

/crossprod {
% use a local dictionary of two variables
2 dict begin
% call U and V the two vectors we are working with
/V exch def
/U exch def
% the first bracket in the answer
[
% the first entry
U 1 get
V 2 get
mul
U 2 get
V 1 get
mul
sub

% the second entry
U 0 get
V 2 get
mul
U 2 get
V 0 get
mul
sub
% not forgetting sign reversal
neg

% the third entry
U 0 get
V 1 get
mul
U 1 get
V 0 get
mul
sub
% final bracket
]
end
% what is now on the stack is [W0 W1 W2]
% which is the cross product U x V
} def
```

Exercise. *Add this definition to your copy of the file* `calc.inc`*. Use it to calculate the cross product of* $(1, 1, 1)$ *and* $(1, -1, 0)$.