

bash & bc

In the Shell Calculator

It doesn't always have to be a GUI-based calculator like *xcalc* or *Kcalc* – *bash* can handle simple arithmetical operations, and for more advanced math you can always try *bc*.

BY HEIKE JURZIK

The Bourne Again Shell), or *bash* for short is the default shell on Linux. It accepts commands and interprets them, launches programs, and manages processes. In addition to this, *bash* stores environment variables and allows other programs to use them. Besides a variety of programming features, *bash* can also handle the four basic mathematical operations (addition, subtraction, multiplication, and division), and the modulo operator, and is thus capable of evaluating mathematical expressions.

However, as *bash* can only handle integer values (whole numbers without decimals) of up to 32 bits, it isn't exactly a mathematical genius. If you need to work with floating point numbers, why not try *bc*, the "Basic Calculator"? The program can be used interactively or scripted.

A Tool to Reckon with

Bash requires a special syntax for mathematical operations. Older versions need to add a dollar sign prefix and enclose the mathematical expression in square brackets. *Bash* 2.0 or later encloses the expression in double round brackets. Even though later *bash* versions can handle the older notation, we will be sticking to the double round brackets in the following examples.

Bash performs calculations with "normal" numbers and variables, for example:



```
hj@asteroid:~$ echo $((1+1))
2
```

Alternatively, you can define variables, and assign values to them, before using them in calculations:

```
hj@asteroid:~$ a=4
hj@asteroid:~$ b=5
hj@asteroid:~$ echo $((a*b))
20
hj@asteroid:~$ echo $((a-b))
-1
```

Use a semicolon to fit these statements and calculations in a single line. The semicolon connects the commands. The previous example can also be expressed in a single line, as follows:

```
a=4; b=5; echo $((a*b))
```

The other operators are the minus sign (-) for subtractions, the asterisk (*) for multiplications, and the forward slash (/) for integral divisions.

A double asterisk (**) is used as the exponential operator, for example, *echo \$((2**16))*, and the percent sign represents the modulo operator, which ascertains the remainder in integral divi-

sions. For example, *echo \$((5%3))* equals 2, as there is a remainder of 2 when you divide 5 by 3.

What About the Decimal Point?

If you type something like *echo \$((1/3))* you will note that this is not really an integral division, but *bash* can't handle floating point numbers. Enter *bc*. Type *bc* to launch the calculator.

First time users may be confused by the fact that the command does not display a prompt, but don't panic, *bc* is waiting for your input all the same. You can either type *quit* or press [Ctrl-d] to quit the calculator.

As you would expect, arithmetical expressions can be formulated with the

Command Line

Although GUIs such as KDE or GNOME are useful for various tasks, if you intend to get the most out of your Linux machine, you will need to revert to the good old command line from time to time. Apart from that, you will probably be confronted with various scenarios where some working knowledge will be extremely useful in finding your way through the command line jungle.

normal operators: +, -, *, and /, and brackets:

```
(1.1+2)*2 [Return]
6.2
```

Press [Return] after typing an expression, and *bc* will return the result. Type a semicolon to separate operations:

```
1.1+2; 3.1-2 [Return]
3.1
1.1
```

bc stores a history of all your commands, allowing you to use the arrow keys to scroll back and forth through the most recent commands, just like in the shell.

The four basic mathematical operations in *bc* are identical to those provided by *bash*, but the exponential operators are different. *bc* uses a circumflex (^) instead of a double asterisk.

Variables

If you repeatedly use the same number in a calculation, it makes sense to declare it as a variable. The equals sign is used to assign a value to a variable, for example:

```
a=201; b=300
```

Unfortunately, *bc* does not tell you that it has stored the variables. To display the values of *a* and *b*, you need to type the variable name and press [Enter] – in fact, *bc* is extremely terse.

Any variables you define can be used as numbers in calculations:

```
a*b
60300
a+b
501
```

You might be in for a surprise if you attempt to divide these two variables: *bc* returns a result of 0. There is a simple answer, however. To support floating point operations, you need to specify the *-l* option when launching the program. Fortunately, there is also a way of adding floating point support at runtime. Enter *scale=value* (where value is an integer between 0 and 99) to specify the number of decimal places *bc* displays (and calculates):

```
scale=23 [Return]
a/b [Return]
.670000000000000000000000
```

When performing divisions, you may note that *bc* suffers from the same problem as any programs that calculate floating points: rounding errors. For example, *1/3*3* returns a value of 0.99999999 (the number of nines reflects the number of decimal places you specified), whereas the mathematically equivalent expression, *1/(3/3)*, is calculated correctly, returning 1.

On the upside, *bc* handles negative numbers really well. Simply prefix a minus sign to a number:

```
a + -b
-99
-a
-201
```

Other Worlds

bc has quite a few tricks up its sleeve, mathematical functions such as the square root, for example:

```
sqrt(144)
12
```

By default, *bc* will use the decimal (base 10) system. However, it is quite simple to assign a different base:

```
ibase=16
A
10
```

Anything entered after this *ibase* command will be interpreted by *bc* as hexadecimal input – including the command to switch back to base 10, *ibase=10*. Of course, *bc* expects hexadecimal input, and correctly interprets 10 as 16 in decimal, so why should it change? The correct syntax in this case is *ibase=A*. To check which base the program is working in at present, type *ibase* without any additional parameters.

While *ibase* sets the input format, *obase* defines the base for output. If you are interested in discovering the hexadecimal value for 15, type the following:

```
obase=16 [Return]
15 [Return]
```

```
F
```

You should not have any difficulty resetting *obase* to base 10. *obase=10* will do the trick, assuming that you have not changed the *ibase*, i.e. the input format.

Check it out

bc is actually a programming language, as a quick review of the manpage confirms. This allows the program to interpret control structures, similar to the ones used by C or C++, for more complex tasks. To output the squares of the numbers 1 through 10, use a simple *for* loop. This can save a lot of typing:

```
for (i=1; i<=10; i++) print "
i^2, "\n";
1
4
9
16
25
36
49
64
81
100
```

The syntax of the loop is easy to understand. There are three expressions in the brackets, followed by a statement. The first expression (*i=1*) is performed exactly once before the loop commences. The statement is performed until the second expression becomes true, i.e. until *i <= 10*. The third expression is executed on each iteration, i.e. *i* is incremented.

There is no need to launch *bc*, enter a command, and terminate the program, each time you need to perform a simple calculation. You can use a pipe to pass a calculation to *bc*, which is in fact a sneaky way of getting *bash* to do floating point calculations:

```
hj@asteroid:~$ echo 1/3 | bc -l
.333333333333333333333333
```

Using backticks (`), you can even store calculations in shell variables:

```
x=`echo "$a*$b" | bc -l`
```

multiplies the values of the shell variables *a* and *b*, and stores the result in *x*. ■