The GNU Basic Calculator (bc): a Quick-Start Guide for Mathematicians

Mike Pierce · Last tended 14 Jan 2023 · Hosted at coloradomesa.edu/~mapierce2/bc

The GNU basic calculator (bc) is a free and open source command-line program that performs arbitrary-precision calculations using the full capabilities of any computer. This tool is more robust than most OS's default calculator app, much faster than a TI calculator, and more convenient to use than the computational sledgehammers Mathematica and Jupyter/python. And it's only "basic" by default; bc is a full programming language too, extendable with user-written functions.

Professionally speaking, bc is the most elegant tool for mathematicians to attain the feeling of power that comes with wielding a computer to do arbitrarily precise computations. Pedagogically speaking, bc provides many benefits: a means to subvert the pervasiveness of proprietary hardware and software in classrooms, an easy way to demonstrate the computational capabilities of a computer to students, and also an accessible avenue for students to operate a computer as a calculator for themselves.

Contents

Installing on Linux/macOS

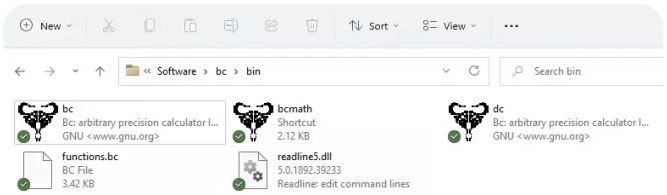
On macOS and most Linux operating systems bc will already be installed. If not, you can install it using your package manager. For example on a Debian-based OS like Ubuntu run the command `sudo apt install bc`. Similarly on macOS you can install bc via the homebrew package manager with the command `brew install bc`.

Running the command `bc` in your terminal will display bc's *copyright* banner and start bc interactively. You can suppress this banner with the `-q` option, and load bc's math library with the `-l` option. It's also helpful to keep a running collection of functions you write in a file, say *functions.bc*, which you load into bc on startup. All of this can be done automatically when you start bc by setting the environment variable `BC_ENV_ARGS`, by including this line in your shell's (bash, zsh, ...) configuration file:

```
export BC_ENV_ARGS="-lq /PATH/TO/functions.bc"
```

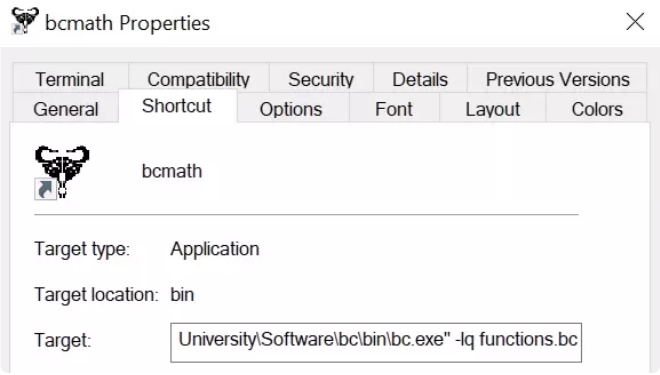
Installing on MS Windows

Although bc was originally written for unix-based operating systems, it's been ported to windows via GnuWin. If you have administrative privileges on your Windows machine just download the Setup files to install bc. If not —like if you're using a classroom computer locked down by your school's IT department—you can "portably" install bc. Download the Binaries and the Dependencies zip archives from the GnuWin page. Extract those zip archives, and move the files somewhere convenient. Copy the file *readline5.dll* from the dependencies archive into the same directory as *bc.exe*. You'll want to keep an auxiliary file of functions you write, *functions.bc*, in this same directory.



Screenshot: a directory containing the specified files in Windows

To suppress bc's *copyright* banner, and automatically load the math library and your auxiliary functions when you start bc, create a shortcut to *bc.exe* that includes the arguments `-lq functions.bc` to bc in the *Target* of the shortcut file. You could name the shortcut file something distinctive like *bcmath* and start it quickly by searching "bcmath" under the Windows Start menu.



Screenshot: how to edit the *Target* field in Windows

Getting Started Using bc

The bc manual comprehensively describes its usage and features, so this quick-start guide will be a simple distillation of the key ideas and quirks of using bc, and a few miscellaneous tips.

Since bc is a command-line calculator, no longer will you waste time commuting your hand between your mouse and keyboard. Running bc, simply type in an expression, press `Enter` to evaluate it, and the result will be printed on the next line. To stop any currently running computation (because it's taking longer than expected) press `Ctrl` + `C`. To quit and exit bc, either type the command `quit` or press `Ctrl` + `D`.

Input preceded by a `#` character is ignored by bc, which is intended to be used to leave comments in code.

```
# Adding Three and Four
3 + 4
→ 7
```

Using Variables

You can assign values to variables using `=`, which is helpful for longer computations.

```
a = 1384600072955855577415429242O
b = 63050430913919525940448
hypotenuse_squared = a^2 + b^2
sqrt(hypotenuse_squared)
→ 1384600072970211191415664365 2.000000000000000000000
```

Variables don't have to be initialized like they do in most programming languages. GNU bc will just infer the data type of your variable: either a number or an array of numbers. Variable names must start with a lowercase letter, but otherwise can be any combination of lowercase alphanumerical characters and underscores. So using `h2` instead of `hypotenuse_squared` above would've been just fine. **Caution**: Capital letters are reserved for computations in bases higher than ten, and so are not allowed in variable names.

The special variable `last` contains the output of the previous command, allowing you to quickly feed calculation results into new calculations. Alternatively, a single period character `.` is a shorthand for `last`.

```
a = 1384600072955855577415429242O
b = 63050430913919525940448
a^2 + b^2
→ 191711736206911415591997907991571150760667877089711897104

sqrt(.)
→ 1384600072970211191415664365 2.000000000000000000000
```

Controlling the Scale of bc's Computations

The special variable `scale` gives you control over the fractional precision of bc's output, setting the number of digits to the right of the radix point (decimal point in base ten) that bc will store during computations. This simple control over the precision of real-number calculations is the shining feature of bc. Loading the math library sets `scale` to 20, but you can manually change this to whatever suits your needs:

```
sqrt(2)
→ 1.41421356237309504880

scale = 42
sqrt(2)
→ 1.414213562373095048801688724209698078569671
```

Conversely, there is a function `length()` that returns the number of an input's significant digits, and a function `scale()` that returns the number of significant digits to the right of the radix point.

```
gelfond = 23.1406926
length(gelfond)
→ 9

scale(gelfond)
→ 7
```

Caution: the value of `scale` is *not* the number of decimal places you may assume to be accurate. Rounding error can accumulate:

```
scale=2
9*(1/9)
→ .99
```

For a more stark example, see how bc could lead us astray from the fact that $\lim_{x\to 0} \frac{\ln(x+1)}{x} = 1$. Setting `scale=5` and using `l()` as the natural log function:

```
x=10^(-1); l(x+1)/x
→ .95310
```

```
  x=10^(-2); l(x+1)/x
→ .99500

  x=10^(-3); l(x+1)/x
→ .99000

  x=10^(-4); l(x+1)/x
→ .90000

  x=10^(-5); l(x+1)/x
→ 0
```

All you can say is that the difference between your calculated value and the "true value" is less than $10^{n-\text{scale}}$ for some n, depending on the algorithmic complexity of your calculation and magnitude of your parameters. Discovering exactly what n must be in a given case requires one dive into the world of numerical analysis.

Handling Fractional Exponents

Simply doing arithmetic in bc, the first thing you'll notice is that bc can't immediately handle non-whole numbers as exponents. Trying to calculate $2.71^{3.14}$ as a decimal number with the command `2.71^3.14` will throw an error. Instead you'll need to use a custom function for this, based on the functions `e(x)` $= e^x$ and `l(x)` $= \ln(x)$ from bc's math library, which do accept fractional parameters. Here's a `pow` function that evaluates b^x for an arbitrary base `b` and exponent `x`.

```
define pow (b,x) { return e(x*l(b)) }
```

Then you can compute $2.71^{3.14}$ as `pow(2.71, 3.14)`. That's a good function to keep in your auxiliary *functions.bc* file. Details about the math library and writing functions can be found later in this guide.

Getting the Integer Part of a Number

The modus operandi of bc is to work at some fixed scale, so it doesn't store the integer part of a number for easy access. You can calculate the integer part of a number by temporarily changing the scale though.

```
define int(x) {
    auto s;
    s=scale;
    scale=0;
    x/=1;
    scale=s;
    return x;
}
```

This is also a helpful function to keep in your auxiliary functions file.

Extending bc with its Math Library

Besides support for basic arithmetic operations like addition, multiplication, integer exponentiation, etc, the only mathematical function built into bc is `sqrt()`. Loading bc's math library with the `-l` option defines the following additional functions:

- `s(x)`, the sine of x, for x in radians.
- `c(x)`, the cosine of x, for x in radians.
- `a(x)`, the arctangent of x, arctangent returns radians.
- `l(x)`, the natural logarithm of x.
- `e(x)`, the exponential function of raising `e` to the value x.
- `j(n,x)`, the bessel function of integer order n of x.

Only the trigonometric functions sine, cosine, and arctangent are included for historical reasons, but really that's all you need: all other trigonometric functions can be expressed in terms of these three. Doing this is left to the user as an exercise, … or you can find my implementations at the end of this guide.

Since the trigonometric functions in this library expect angles expressed in radian measure, it is helpful to have π stored as a constant. Define π as four times the arctangent of one, `pi = 4*a(1)`. You could keep this line in your auxiliary *functions.bc* file, along with these functions to convert between degree and radian measure:

```
define radtodeg (x) { return x*(45/a(1)) }

define degtorad (x) { return x*(a(1)/45) }
```

Programming: Functions, Arrays, & Limitations

A quick note: this guide is specific to **GNU** bc—there are other implementations of bc with tighter standards and more/less features. But GNU bc is the version you'd most likely encounter in the wild.

The syntax of bc is similar to the syntax of the C programming language, but much looser. If you have no programming experience you should read the *Statements* section of the bc manual for a more thorough treatment. Otherwise if you have some experience programming in C or in any other modern language, this section will help you translate that experience over to bc.

Defining Functions

The syntax for declaring a function in bc is

```
define NAME ( PARAMETERS ) {
    auto AUTO_LIST
    ...
    return OUTPUT
}
```

where `NAME` is the name of your function, `PARAMETERS` is the list of inputs to your function, `AUTO_LIST` is a list of local variables you define only for use in this function, and `OUTPUT` is the value the function returns. In the body of the function the curly braces `{}` group multiple independent commands together, and either semicolons `;` or newlines (or both) separate those independent commands. For example, here's a function that returns the n^{th} Fibonacci number using the classic recursive definition.

```
define fibonacci (n) {
    auto i
    if (n==1 || n==2) {
        return 1
    }
    return fibonacci(n-1) + fibonacci(n-2)
}
```

Printing Output & Reading Input

Within a function it may be helpful to print text to the screen besides just the return value, or to read user input while the function is running. For these needs we have respectively the `print` command and `read()` function. The `print` command takes a comma-separated list of things and prints them to the terminal, while the command `x = read()` will store a number entered interactively by a user to the variable `x`. For example:

```
define fibonacci_ask () {
    auto r
    print "Which Fibonacci number would you like?\n"
    r = read()
    return fibonacci(r)
}
```

That `\n` character prints a newline to the terminal; otherwise bc would prompt for input on that same line it printed the question.

Control-Flow Syntax & Arithmetic

All of the control-flow keywords `if`, `else`, `for`, `while`, `continue`, `break`, and `halt` are supported in GNU bc. Instead of the keywords "true" and "false" bc uses the numbers `1` and `0` respectively. To provide some illustrative examples of the use of these keywords: suppose you need a quick test to tell if two integers `a` and `b` such that $2 < a < b$ might be adjacent Fibonacci numbers:

```
if (int(10*(b/a)) == 16) {
    print "a and b may be adjacent Fibonacci numbers\n"
} else {
    print "a and b aren't adjacent Fibonacci numbers\n"
}
```

Or suppose you want to nicely display the first 42 Fibonacci numbers along with their index:

```
for (i=1; i<=42; ++i) {
    print i, " | ", fibonacci(i), "\n"
}
```

Or suppose you want to find the smallest power of 2 that has a leading digit of 7:

```
x = 1
while(1) {
    if (int(x/10^(length(x)-1)) == 7) {
        break
    }
    x*=2
}
print x
```

With the exception of C's bitwise operations and two other special cases, bc shares all of C's arithmetic, assignment, comparison, and logical operators. Those two special cases are these:

- The caret `^` is an *integer* exponential operator, not a bitwise operator. The command `a ^= b` does the same thing as `a = a^b`.
- The modulus operator `%` behaves the same in bc as in C only when scale is set to zero. Functionally `a % b` returns `a-(a/b)*b` regardless of the value of `scale`. In the language of the remainder theorem, you can think of this general modulus operator this way: for a fixed scale n, `a % b` returns the unique non-negative number r less than $b \times 10^{-n}$ for which there exists a number q such that $q \times 10^n$ is an integer and $a = q \times b + r$. For example, `scale=3; 8%7` returns `0.006` since $8 = 1.142 \times 7 + 0.006$.

Array Variables & Loops

In addition to simple variables, bc supports array variables, iterable collection of variables indexed by non-negative integers (indexing starts at zero). The index of an array you'd like to access is indicated by enclosing it in square brackets `[ ]` after the array's name. That is, the variable a_i for some $i \in \mathbf{Z}^+$ is denoted as `a[i]` in bc.

```
a[1] = 13846000729558555774154292420
a[2] = 63050430913919525940448
h2 = a[1]^2 + a[2]^2
sqrt(h2)
→ 13846000729702111914156643652.00000000000000000000
```

Simple variables and array variables of the same name can co-exist with no conflict: letting `a = a[1]^2 + a[2]^2` in that last example would have been

just fine. Like simple variables, arrays don't have to be initialized, and the size of an array never needs to be declared. The maximum size of an array in bc is set as `BC_DIM_MAX`, a value chosen at compile time. In my current version of bc, this value is 2^{24}.

Arrays are useful for looping over some enumeration of data. For example, we can write a more efficient `fibonacci()` function by using a for-loop and storing previously computed Fibonacci numbers in an array `fib`:

```
define fibonacci (n) {
    auto i, fib
    fib[1] = fib[2] = 1
    for (i=2; i<=n; ++i) {
        fib[i] = fib[i-1] + fib[i-2]
    }
    return fib[n]
}
```

Limitation of bc

There are a couple limitations of bc you should keep in mind when considering if you'd like to use a more robust programming language.

In bc you cannot pass functions as parameters to other functions. The only variable types—and so the only entities that may be a function's parameters—are simple variables and arrays. This limitation makes for messy code, where any function that morally *should* have a function as a parameter instead has to assume a globally-defined function of a fixed name already exists. I've been using the convention that such a function must be globally defined and named `f`, and its derivative named `ff`.

You cannot explicitly define the contents of an entire array at once. I.e. the syntax `array = {8, 6, 7}` is not accepted, and instead you'd have to use:

```
array[0] = 8; array[1] = 6; array[2] = 7;
```

Similarly, since bc doesn't keep track of the size of an array, there is no command to print the entire contents of an array. Instead you must keep track of the size of an array yourself and loop over the indices to individually print the entries. This makes working with data in bc cumbersome, and doing any sort of statistics impossible.

On a related note, you can't return an array from a function. Instead you have to pass your array by reference as a parameter, a feature of GNU bc not documented in the manual. The syntax is demonstrated in this example:

```
define fibonacci_next (*array[], n) {
    array[n] = array[n-1] + array[n-2]
    return array[n]
}


fib[42] = 267914296
fib[43] = 433494437
fibonacci_next(fib[], 44)
→ 701408733

  fib[42] = 267914296
  fib[44]
→ 701408733
```

Additionally there is not built-in support for matrices (arrays-of-arrays, 2D arrays).

Exercises & Challenges

The best way to learn a new tool is on the job; start using bc instead of whatever basic calculator you've been using, and if you teach be sure to install it on your classroom computer to have at hand during a lesson. In case you need some initial inspiration to dive into bc, here are some tasks you can complete as practice.

I've written functions to complete some of these tasks in my own *functions.bc* file, which you can find on GitHub, in case you'd like to compare solutions.

1. **The Quadratic Formula** · Write a function that takes the coefficients of a quadratic polynomial $ax^2 + bx + c$ as input and prints the roots of that polynomial. Since there are two "outputs" of this function you'll likely want to `print` one of the roots and return the other. Alternatively, using bc version ≥1.07.1 you could print both roots and have your function return nothing using the `void` keyword:

   ```
   define void f (a,b,c) { ... }
   ```

   For more of a challenge, try to (efficiently) write the function to compute the roots of a cubic or a quartic polynomial.

2. **Digit Manipulation** · What is the 10000^{th} digit in the decimal expansion of π?

   More generally, write a function that takes a decimal number x as input and for some $n <$ `scale` returns the n^{th} digit to the right of the decimal place in x.

3. **Decimal Digit as an Array** · Given a decimal number x create an array and write a loop that will store the first n digits to the right of the decimal point in x in the indices in the array.

4. **Newton's Method** · For a differentiable function `f` = f with a root and its derivative `ff` = f' implement Newton's method of approximating a root of f.

   Then use this function to compute the Dottie number, the constant defined to be the single real solution to the equation $\cos(x) = x$, accurate to 42 decimal places.

5. **Numerical Integration** · Write a function that, given a globally defined function `f`, approximates the value of the definite integral

$$\int_a^b f(x)\,\mathrm{d}x\,.$$

Can you write this function to accurately compute the value of the integral up to `scale`?

6. **Prime Factorization** · Write a function that prints the prime factorization of its input.

Links & References

See the GNU bc manual to learn *all* of bc's functionality. If you have a question and you can't find the answer in the manual, you can ask general questions about bc on the Unix & Linux Stack Exchange, or programming-related questions about bc on Stack Overflow.

The maintainer of `phodd.net` has authored a huge library of bc functions and written a helpful bc FAQ page. Additionally, Keith Matthews has written some number theory programs for bc. So if you need a specific algorithm you don't want to implement yourself, these are the first places you should check.

And if you're ever on Twitter having an argument that could be settled with a computation, there is a Twitter bot @bc_1 that will execute bc code.

My File of Auxiliary Functions

In case you're not excited by the romantic notion of implementing mathematical functions yourself as you need them, here are two options: copy any of the "basic" functions below into your own *functions.bc* auxiliary file to give you a more usable base to start using bc from, or feel free to fork/clone my personal working file of bc functions from GitHub.

Helpful Constants & Functions

```
pi=4*a(1)
ex=e(1)
define abs(x) { if (x>0) return x; return -x }
define sgn(x) { if (x>0) return 1; return -1 }

# Return the integer-part of a number (not the floor)
define int(x) {
    auto s;
    s=scale;
    scale=0;
    x/=1;
    scale=s;
    return x;
}
```

Logarithms & Exponentials

```
define log(x) { return l(x) }
define exp(x) { return e(x) }
define logb(x,b) { return l(x)/l(b) }
define pow(x,n) { return e(n*l(x)) }
```

Trigonometry

```
define radtodeg(x) { return x*(45/a(1)) }
define degtorad(x) { return x*(a(1)/45) }

define cos(x) { return c(x) }
define sin(x) { return s(x) }
define tan(x) { return s(x)/c(x) }
define sec(x) { return 1/c(x) }
define csc(x) { return 1/s(x) }
define cot(x) { return c(x)/s(x) }
define arccos(x) {
    if(x == 1) return 0
    if(x == -1) return pi
    return pi/2-a(x/sqrt(1-(x^2)))
}
define arcsin(x) {
    if(x == 1) return pi/2
    if(x == -1) return -pi/2
    return sgn(x)*a(sqrt((1/(1-(x^2)))-1))
}
define arctan(x) { return a(x) }
define arcsec(x) { return arccos(1/x) }
define arccsc(x) { return arcsin(1/x) }
define arccot(x) { return pi/2-a(x) }

# Display the Pythagorean triple
# generated by integers m and n,
# and return the hypotenuse
define pythagtriple(m,n) {
    print abs(n*n-m*m), "\n"
    print abs(2*m*n), "\n"
    return m*m+n*n
}
```

Combinatorics

Note these functions don't bother to check that their parameters are integers.

```
define factorial(n) {
    if (n<1)
        return 1
    return n*factorial(n-1)
}

# nCk, the number of ways to choose k of n objects
```

```
define choose(n,k) {
    auto c
    c = factorial(n)/(factorial(n-k)*factorial(k))
    return int(c)
}

# nPk, the number of ways to permute k of n objects
define pick(n,k) {
    auto i, r
    r = n-k
    for (i=1; n > r; --n)
        i*=n
    return i
}
```

Fork this Guide on GitHub

In keeping with the ethos of digital gardening, and of the internet being a living, community-tended library of humanity's collective knowledge, I don't want this page to become a corpse of my sole authorship cluttering the stacks of our library. What a tragic abuse of the internet that would be. No, I'd rather simply be the … maintainer? custodian? groundskeeper? … of this living guide. Please feel free to become a co-author and tend to this page with me.

CONTRIBUTING · The semantic HTML for this page is hosted on GitHub, licensed as CC BY-NC-SA 4.0. If you'd like to suggest an edit or addition to my instance of this guide, create an issue on GitHub, fork the repository, make your changes, and submit a pull request. My only requirement of co-authors of this instance is that you keep with the spirit of the page as a *quick-start guide* written with an audience of mathematicians, instructors, and students in mind—professional programmers and Unix administrators are already well-served by the GNU bc manual itself.