Numerical tools

There are a large number of tools available for Unix machines:

- ☞ Desktop tools such as `bc`, `dc`, and Pari/GP
- ☞ Computer Algebra Systems such as `maxima`
- ☞ Numerical tools library: GMP and Pari/GP
- ☞ Visualization via `gnuplot` and `graphviz`

bc and dc

bc is a calculator. Normally, it works with integers, but you can set it the number of decimal places with the scale variable:

```
[langley@sophie 2006-Fall]$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
1/6
0
scale=20
```

```
1/6
.1666666666666666666
```

`bc`

You can also do quick base conversions with `bc`:

```
$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type ‘warranty’.
obase=16
ibase=10
16
10
quit
$ bc
bc 1.06
```

```
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
ibase=10
obase=16
15
F
quit
```

bc

`bc` uses traditional infix notation:

```
$ bc
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type ‘warranty’.
12 + 34
46
12 * 34
408
34 / 12
2
99 - 12
```

```
87
56 % 14
0
3 ^ 3
27
```

bc

`bc` also allows small programs to be written:

```
a=0
while(a < 10)
{
  a = a+1;
  print a * a , "\n";
}

1
4
9
16
25
```

```
36
49
64
81
100
```

`bc`

`bc` supports the following statement types:

- Simple expressions, such as `3 * 5`
- Assignment, such `a = a - 1`
- `if/then`
- `while`

- ☞ Compound statements between `{ }`
- ☞ C-style `for`: `for(EXP1 ; EXP2 ; EXP3)`
- ☞ `break` and `continue`
- ☞ Function definition and return with `define` and `return`

bc

Math functions available when started with `-l`:

```
s(x)      # sine of x in radians
c(x)      # cosine of x in radians
a(x)      # arctangent of x in radians
l(x)      # natural logarithm of x
e(x)      # e to x
sqrt(x)   # square root of x (doesn't actually need -l option)
```

dc

The program `dc` is desk calculator much like `bc` in calculator mode, but is uses Reverse Polish Notation (RPN) rather than infix notation. Unlike `bc`, `dc` doesn't support complex statements and programming.

dc

```
[langley@sophie 2006-Fall]$ dc
34 99
f
99
34
55 88
f
88
55
99
34
+
*
*
```

```
f
481338
quit
```

dc

dc commands:

```
p      # print the top value from the stack
n      # print the top value from the stack and pop it off
f      # print the entire stack
+      # adds the top two values from the stack and pushes the result
-      # substracts the first value on the stack from the second, pops the
       # off, and pushes the result
*      # pops top two values from stack, pushes multiplication result onto
/      # pops top two values from stack, pushes division result back on st
~      # pops top two values from stack, pushes both division and remainde
       # back on stack
```

GP/Pari

GP/Pari is a much featureful calculator than `bc`. It handles integers, reals, exact rationals, complex numbers, vectors, and more. It does modular arithmetic natively. It can some equation simplification, and it has a number of number theoretical functions such as `gcd()`.

GP/Pari

Starting GP/Pari at a shell prompt is easy:

```
$ gp
                        GP/PARI CALCULATOR Version 2.1.7 (released)
                      i686 running linux (ix86 kernel) 32-bit version
                      (readline v4.3 enabled, extended help available)
                              Copyright (C) 2002 The PARI Group
PARI/GP is free software, covered by the GNU General Public License, and comes WITHOUT A
Type ? for help, \q to quit.
Type ?12 for how to get moral (and possibly technical) support.
   realprecision = 28 significant digits
   seriesprecision = 16 significant terms
   format = g0.28
parisize = 4000000, primelimit = 500000
? simplify((a+1)*(a-1))
%1 = a^2 - 1
? ??
```

You can also start it inside of Emacs with `M-x gp` if the appropriate pari.el file is available on your machine. The details are in the GP/Pari manual which you can pull up with `?? emacs`.

Using `gp`

`gp` also uses simple infix notation, like `bc`:

```
? 12 + 24
%2 = 36
?
```

Using gp

Notice that each result is numbered. You can use that notation to refer to a result:

```
? 12 + 24
%43 = 36
? %43 * 14
%44 = 504
?
```

(You can refer to just `%` for the previous result.)

Builtin functions in GP

There are a very large number of functions builtin to GP. You can them with ordinary prefix notation:

```
? gcd(1019986919288111313171891231912376299117891237171129910217,
2198699771571875111911119160590951112121701191107)
%42 = 319
? factor(1001)
%3 =
[7 1]

[11 1]

[13 1]
```

```
? factor(540)
%45 =
[2 2]

[3 3]

[5 1]
?
```

Some useful builtin functions in GP

```
gcd        # greatest common divisor
factor     # factorization
simplify   # simplify a one-variable polynomial
```

Debugging

You can turn on copious debugging in GP with `\g20`:

```
? \g20
   debug = 20
? factor(120940129401294019203490124901249012401421241412410241124111l)
Miller-Rabin: testing base 1000288896
IFAC: cracking composite
        34338877624535303177265598981012930047607660148829727
IFAC: checking for pure square
OddPwrs: is 34338877624535303177265598981012930047607660148829727
        ...a 3rd, 5th, or 7th power?
        modulo: resid. (remaining possibilities)
           211:   79   (3rd 1, 5th 0, 7th 0)
```

```
           209:   98   (3rd 0, 5th 0, 7th 0)
IFAC: trying Pollard-Brent rho method first
Rho: searching small factor of 175-bit integer
Rho: using X^2-11 for up to 4770 rounds of 32 iterations
Rho: time =    100 ms,        768 rounds
Rho: fast forward phase (256 rounds of 64)...
Rho: time =     50 ms,        1028 rounds, back to normal mode
Rho: time =     30 ms,        1280 rounds
Rho: time =     40 ms,        1536 rounds
Rho: fast forward phase (512 rounds of 64)...
Rho: time =    120 ms,        2052 rounds, back to normal mode
Rho: time =     30 ms,        2304 rounds
Rho: time =     30 ms,        2560 rounds
Rho: time =     40 ms,        2816 rounds
Rho: time =     30 ms,        3072 rounds
Rho: fast forward phase (1024 rounds of 64)...
Rho: time =    230 ms,        4100 rounds, back to normal mode
Rho: time =     40 ms,        4352 rounds
Rho: time =     40 ms,        4608 rounds
Rho: time =     20 ms,        Pollard-Brent giving up.
```

```
IFAC: trying Shanks' SQUFOF, will fail silently if input
      is too large for it.
IFAC: trying Lenstra-Montgomery ECM
ECM: working on 8 curves at a time; initializing for up to 3 rounds...
ECM: time =      0 ms
ECM: dsn =  4,        B1 =  700,        B2 =  77000,        gss =  128*42
ECM: time =    200 ms, B1 phase done, p = 701, setting up for B2
        (got [2]Q...[10]Q)
        (got [p]Q, p = 709 = 79 mod 210)
        (got initial helix)
ECM: time =     10 ms, entering B2 phase, p = 913
ECM: finishing curves 4...7
        (extracted precomputed helix / baby step entries)
        (baby step table complete)
        (giant step at p = 27799)
ECM: finishing curves 0...3
        (extracted precomputed helix / baby step entries)
        (baby step table complete)
        (giant step at p = 27799)
ECM: time =    140 ms
```

```
ECM: dsn =  6,         B1 =  900,         B2 =  99000,         gss =  128*42
ECM: time =    260 ms, B1 phase done, p = 907, setting up for B2
        (got [2]Q...[10]Q)
        (got [p]Q, p = 911 = 71 mod 210)
        (got initial helix)
ECM: time =      0 ms, entering B2 phase, p = 1117
ECM: finishing curves 4...7
        (extracted precomputed helix / baby step entries)
        (baby step table complete)
        (giant step at p = 28001)
        (giant step at p = 81761)
ECM: finishing curves 0...3
        (extracted precomputed helix / baby step entries)
        (baby step table complete)
        (giant step at p = 28001)
        (giant step at p = 81761)
ECM: time =    190 ms
ECM: dsn =  8,         B1 = 1150,         B2 = 126500,         gss =  128*42
ECM: time =    320 ms, B1 phase done, p = 1151, setting up for B2
        (got [2]Q...[10]Q)
```

```
        (got [p]Q, p = 1153 = 103 mod 210)
        (got initial helix)
ECM: time =     10 ms, entering B2 phase, p = 1361
ECM: finishing curves 4...7
        (extracted precomputed helix / baby step entries)
        (baby step table complete)
        (giant step at p = 28277)
        (giant step at p = 82003)
ECM: finishing curves 0...3
        (extracted precomputed helix / baby step entries)
        (baby step table complete)
ECM: time =    110 ms,        p <=  28229,
        found factor = 31705445367881
IFAC: cofactor = 10830593049899902997180130267987274657 67
Miller-Rabin: testing base 768462011
Miller-Rabin: testing base 892785826
Miller-Rabin: testing base 739165157
Miller-Rabin: testing base 1874708212
Miller-Rabin: testing base 1732294655
Miller-Rabin: testing base 1648543222
```

```
Miller-Rabin: testing base 659912585
Miller-Rabin: testing base 370113064
Miller-Rabin: testing base 670592259
Miller-Rabin: testing base 481073162
IFAC: factor 10830593049899902997180130267987274657 67
        is prime
Miller-Rabin: testing base 1340817133
Miller-Rabin: testing base 353959964
Miller-Rabin: testing base 1730244551
Miller-Rabin: testing base 1484512990
Miller-Rabin: testing base 1728249361
Miller-Rabin: testing base 22662352
Miller-Rabin: testing base 905839691
Miller-Rabin: testing base 2098523762
Miller-Rabin: testing base 1062164725
Miller-Rabin: testing base 1715475524
IFAC: factor 31705445367881
        is prime
IFAC: prime 31705445367881
        appears with exponent = 1
```

```
IFAC: main loop: 1 factor left
IFAC: prime 1083059304989990299718013026798727465767
        appears with exponent = 1
IFAC: main loop: this was the last factor
IFAC: found 2 large prime (power) factors.
%4 =
[5441 1]

[6473 1]

[31705445367881 1]

[1083059304989990299718013026798727465767 1]

?
```

GP/Pari

Getting help is easy. The most comprehensive help comes from firing up the manual pages with `??`. You can choose a specific topic with `?? TOPIC` such as `?? gcd`.