Math 56 Compu & Expt Math, Spring 2013: Homework 6

due 10am Thursday May 9th

Options for python/SAGE setup: 1) install python and the `mpmath` *and* `gmpy` *packages, or 2) install SAGE, or 3) work "in the cloud" by creating a notebook account at* `sage.dartmouth.edu`

This lets you do arbitrary precision; for now you use only the four basic arithmetic operations (although of course they have many more available).

For `mpmath` *usage see:* `http://mpmath.googlecode.com/svn/trunk/doc/build/basics.html`

1. Cementing the concept of super-algebraic convergence.
   (a) Write down a function $f(n)$ exhibiting super-algebraic convergence to zero as $n \to \infty$, but such that $e^{-c\sqrt{n}} = o(f(n))$ for each $c > 0$.
   (b) Fix $c > 0$. Let k be an algebraic convergence order. Find the smallest k-dependent "constant" C_k you can, such that, for each order $k = 1, 2, \ldots$, we have $e^{-c\sqrt{n}} \le C_k n^{-k}$ for all $n = 1, 2, \ldots$. What is $\lim_{k\to\infty} C_k$? This shows that the implied big-O constant[1] in the definition of super-algebraic need not be uniformly bounded with respect to order. [Hint: Fix k, put all upstairs in exp then maximize vs n.]

2. Fast multiplication, and I mean fast.
   (a) Write a Matlab function for Strassen's fast multiplication of arbitrary-precision integers in base 10, with an accompanying driver/test code. Test that the output is a vector of integers in the range 0 to 9. You may use the built-in FFT. [Hint: this is pretty quick if you adapt my code for addition of integers. Make sure to do something to the output of the convolution so that the carrying works reliably!]
   (b) Use your code to compute the ridiculously big number $2^{2^{22}}$ (note the top number is the decimal number 22). Taking 1 and doubling it 2^{22} times is impractical—instead devise a much faster scheme. How many decimal digits does this number have?! Print the last 10 digits. Dial them into a phone.[2] How long does the calculation take? Estimate (roughly, ie within a factor of two) how long it would take using naive arbitrary-precision multiplication at 10^9 flop/sec. [Hint: if `x` is a base-10 vector, `fprintf('%d',x)` is a useful way to display]

   BONUS Make conjectures about the last couple of decimal digits of 2^{2^j} for $j = 2, 3, \ldots$ and prove them.

3. Fast division.
   (a) In python (or SAGE) make a function `x = fastrecip(z,tol,xguess)` which implements the Newton method for the reciprocal of `z`, with initial guess `xguess`, stopping at relative tolerance `tol`. This being python, you can have the test driver and the function in the same file. Note `xguess` needs to be reasonably close to the answer (how near?)
   (b) Import `mpmath` and set working precision to 1000 decimal digits. Compute 1/97 using your above routine (don't use any division!). What is the repetition period of its decimal expansion? Hints:

```
from mpmath import *
z = mpf('97')                # the arb-precision representation of 97
```

[1] or this could instead be N_0 as in $n \ge N_0$, although here $N_0 = 1$ hence the constant has to take the hit.

[2] only joking.

(c) Assuming constant effort per flop[3] in the FFT, deduce the complexity in big-O notation for your fast reciprocal of an N-digit number.

(d) The wikipedia page `http://en.wikipedia.org/wiki/Division_algorithm` in the section **Newton–Raphson division**, in the paragraph starting "From a computation point of view..." claims that $x_{n+1} = x_n(2 - zx_n)$ is (much) less accurate than the mathematically equivalent $x_{n+1} = x_n + x_n(1 - zx_n)$. Evaluate this claim by using your skills in forward error analysis (rules of floating point), for just a single iteration. Ignore the initial rounding step, and you may assume that x_n is near its final (converged) value.

4. Algorithms for digits of π. Please work in python/SAGE/mpmath/gmpy and make use of all basic arbitrary-precision arithmetic and sqrt.

(a) Make a function `y = atantaylor(x,n)` which sums the `n`-term Taylor series about the origin (i.e. Maclaurin series) to approximate $y = \tan^{-1} x$, at whatever the current working precision is. Test against `mpmath`'s `atan` to check. [Hint: make sure all constants are multi-precision, e.g. `mpf('0')`]

(b) Use the above to evaluate any of the Machin–Euler type formulae for π to 10000 digits. Give the last 10 of those digits, state your convergence rate, how many terms n you used, and how many seconds it took.

Python tips: `print str(x)[-10:]` for printing, and the following for timing some chunk of code,

```
import time
t = time.time()
...  (do something) ...
print time.time()-t 'secs'
```

(c) Consider Gauss' arithmetic-geometric mean iteration that lies at the heart of the Brent–Salamin algorithm for π:

$$\begin{aligned} x_{n+1} &= (x_n + y_n)/2 \\ y_{n+1} &= \sqrt{x_n y_n} \end{aligned}$$

Prove that it is quadratically convergent to something. [Hints: bound $|x_{n+1} - y_{n+1}|$ in terms of its previous value. Play with the difference of two squares $x_{n+1}^2 - y_{n+1}^2$.]

(d) Code up Brent–Salamin using `mpmath` for all arithmetic including the square-root (to make your life easier). [Hint: debug at a "low" precision of only 100 digits, while printing $2x_n^2/\alpha_n$ each iteration.]

How many times faster is it for $N = 10^4$ digits than in (b)? Quote the 10 digits of π starting at the millionth (since the last few digits in your answer will be wrong, you'll need to go a few digits beyond what you need).

[3] This is not strictly true, since as N grows, more floating-point accuracy is needed in the FFT so that it rounds to the correct integers; see Crandall–Pomerance book Sec 9.5.