Approximating iteratively: CSC270, Assignment 1

September 24, 2002

There is an HTML version of this document available at
http://www.cs.utoronto.ca/~heap/270F02/a1/a1.html

Assignment 1 is due Thursday October 10 by midnight. Late assignments are not normally accepted, and are never accepted without written explanation. What you hand in should be your own work, and you should re-read the section in the Course Information Sheet if you're unsure.

In this assignment you'll explore a technique for approximating a function, $f(x)$. What makes this exploration a little tricky, at first, is that you aren't given a definition of $f(x)$, but rather of its derivative, $f'(x)$.

A warm-up function

Suppose you're building a reflector to concentrate the sun's rays. After a bit of thought you conclude that the reflector should be circular when viewed from above, and that its cross-section in any vertical plane should always have a slope equal to twice its distance from the reflector's centre. You've got a knack for mathematical notation, so you say that if the distance from the centre is called x, then your reflector should have vertical cross-sections that correspond to a function $f(x)$ with derivative $2x$. How do you find the function $f(x)$ that satisfies $f'(x) = 2x$?

Since you've recently studied some calculus, you might recognize that $f(x) = x^2$ will work (as will $f(x) = x^2 + 37$, and many other similar functions). Or you might haul out the Fundamental Theorem of Calculus and integrate both sides of $f'(x) = 2x$ to come up with the same result. Or you might decide to use the following iterative approximation: Since you can always place your reflector at ground level, you know that (at least) $f(0) = 0$ — so you know at least one pair of coordinates $(0, f(0)) = (0, 0)$. Plugging this into your formula for f', you can see that $f'(0)$ is also 0. You reason that you can approximate $f(h)$ (if h is pretty small) by drawing the tangent line through $f(0)$ and then calling the point where this tangent line has horizontal component h (h, f_h) — a pretty good approximation of $(h, f(h))$. In fact, you don't even need to draw the tangent line if you calculate $f_h = f(0) + f'(0) \times h$. You then **iterate** your approach — you use your approximation $f_h \approx f(h)$ to calculate a new tangent, and with this you come up with a second approximation $(2h, f_{2h})$. If $h = 0.1$, the result is shown in Figure 1.

Clearly our approximate function (which we're denoting f_x to distinguish it from $f(x)$) is not a perfect parabola. But it might turn out that if you build a reflector based on it, you'll be able to heat a can of soup. Otherwise, you'll need to think about how to improve your approximation.

A more challenging function

In the previous section, you may have wondered "why bother with an iterative solution when I can easily see that $f(x) = x^2$ is a solution?" However, what if you were modelling a problem where things were a bit

x	f_x	$f(x)$	x	f_x	$f(x)$
0.0	0.0	0.0	0.6	0.3	0.36
0.1	0.0	0.01	0.7	0.42	0.49
0.2	0.02	0.04	0.8	0.56	0.64
0.3	0.06	0.09	0.9	0.72	0.81
0.4	0.12	0.16	1.0	0.9	1.0
0.5	0.2	0.25	1.1	1.1	1.21

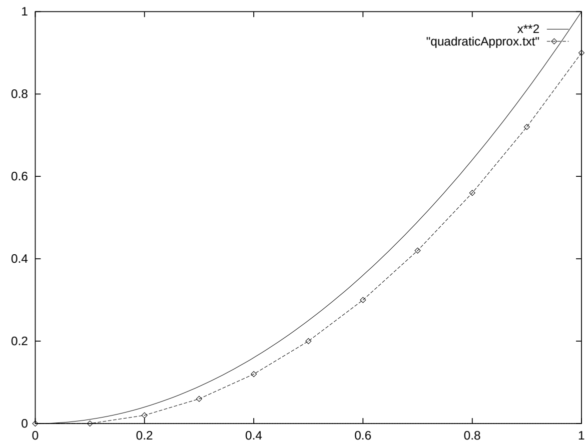

Figure 1: The variable x is incremented by 0.1 at each step. Use the known value of $f(x)$ at the initial point, $x = 0$), and then use the approximation $f_x = f'(x - 0.1) \times 0.1 + f_{x-0.1}$.

messier? Suppose you knew that for some constant K you had:

$$f'(x) = -f(x) - K(f(x) - \exp(-x)) \quad (1)$$

$$f(0) = 2. \quad (2)$$

Once again you have an unknown function, $f(x)$, defined in terms of its derivative.[1] However, our current definition is more complicated, since the right-hand side is an expression in both x and our unknown function $f(x)$. You might have studied differential equations (which Equation 1 is) and be able to use a well-studied bag of tricks (integrals, integrating factors, etc.)[2] to work out with paper and pencil that

$$f(x) = \exp(-(K+1)x) + \exp(-x) \quad (3)$$

However, if what you need is an approximation, you can proceed as in the previous section:

1. Decide on a step size that you'll increment x by (we used 0.1 in the last section, but your mileage may vary), and call it h.
2. Denote your approximate function as f_x to distinguish it from the exact function $f(x)$. Also denote the approximate derivative as f'_x, since it won't always be identical to $f'(x)$.
3. Since you are given the exact value $f(0) = 2$, start things out on the right foot by setting $f_0 = 2$, and the initial value of $x = 0$.
4. f_h is $f'_0 \times h + f_0$, and f'_0 is calculated by by substituting f_0 for $f(0)$ in Equation 1.
5. f_{2h} is $f'_h \times h + f_h$, substituting f_h for $f(h)$ in Equation 1.
6. And so on...

Since you've been given the exact expression for $f(x)$, you can always compare your approximation f_x to $f(x)$.

[1] An equation that defines a function in terms of its derivative(s) is called a Differential Equation (DE). If it has a single independent variable (e.g. x in our case), it's called an Ordinary Differential Equation (ODE).

[2] There are lots of DEs that resist all the tricks in this well-studied bag.

Your job

In a file called `diffeq.c` you will implement the function `delist` specified in diffeq.h. Starter files can be found in the directory ~ajr/a1 on the CDF machines. Your function will use Equation1 and the steps outlined in the previous section to approximate $f(x)$ as x increases from 0 to 1.0 in steps of 0.01. Since you have the initial value $f(0) = 2$, you can fill in the zeroth entry of the table with no calculation:

```
{0.0   2.0   2.0   0.0}
```

For the remaining 99 entries, you have to use the approach described in the last section to calculate f_x.

You may define any helper methods you choose in `diffeq.c`. You may modify the constants `TABLELINES` and `STEPSIZE` in diffeq.h, but you shouldn't alter any of the declarations there. You are welcome to use testdiffeq.c to test drive your solution, and Makefile to build `testdiffeq`.

Briefly comment on your results for $K == 10$ in a file called `report`. What happens if you change K to 500? What happens to the relative error?

Although we don't generally have a nice analytical solution for $f(x)$ that we have derived here, you can use the one provided (Equation 3) to analyze truncation error. In the course notes, page 14–15, there is an expression for the truncation error introduced by truncating a Taylor series (Equation 1.5.4). Substitute our solution into this expression to see whether you can come up with a way to improve our approximation. Explain your suggestion in `report`.

What to hand in

The deadline is Thursday October 10 at midnight. Late submissions are not generally accepted, and never accepted unless you have a written explanation. If you do have an explanation, please do submit your assignment late and send me an explanation by email.

You are to submit files `diffeq.c` and `report`. Each of these files **must** begin with a prologue comment stating your name, student number, and TA (or tutorial time/place), as well as the purpose of the program and any overall notes. Make sure your files are in plain text by using the `cat` command on a CDF machine to display them on the console.

`diffeq.c` contains your implementation of `delist` and any helper functions it needs to do its job. `report` contains your comments on the results of running `DE` with $K == 10$ and $K == 500$, plus your error analysis and suggestions for improving the algorithm. `report` must not exceed 300 words in length.

Do **not** submit diffeq.h or testdiffeq.c, since the marking TA will use their own version.

Once you are satisfied with your files, you can submit them for grading as follows:

```
submit -c csc270h -a a1 diffeq.c report
```

Don't submit your compiled program, just `diffeq.c` and `report`. You may change your files and resubmit them any time up to the due time (the latest copy overwrite an earlier one). You can check that your assignment has been submitted with the command:

```
submit -l -c csc270h -a a1
```

To resubmit a file that has already been submitted, use `-f`:

```
submit -c csc270h -a a1 -f diffeq.c
```

Hand in your own work. A mark of 0/10 is better than the academic penalty for plagiarism.

Marking scheme

You are graded on the quality and organization of your C code. A program that "works" but is extremely disorganized might receive a failing grade.

We may test your implementation of `delist` using automated testing, and these tests may play a role in how the correctness of your code is evaluated. To make this possible, your code should compile and allow `testdiffeq` to run when your `diffeq.c` is present in the same CDF directory as Makefile, `testdiffeq.c`, and diffeq.h.

Your `diffeq.c` must be written in C (not C++) and must compile and run on the CDF Unix system using `gcc -Wall -ansi -pedantic`.