Problem Set 3

18.330 Intro to Numerical Analysis (MIT, Spring 2019)

Due: March 7. To be submitted *online* on Stellar.

Problem 1. ODE integrators with adaptive stepsizing. (20 points)

Just like with numerical quadrature, often ODEs behave differently at different times, the solution may stay almost constant for a long time interval, and then it will suddenly jump or oscillate wildly. To handle such cases, just like you implemented an adaptive integration scheme in the previous PSET, here you will write an adaptive ODE integrator. The idea will be similar: at each time step we make *two* approximations with differing accuracy and and compare them to decide whether to locally decrease the step size.

Your function to adaptively integrate the IVP

$$\frac{d\mathbf{u}}{dt} = \mathbf{f}(t, \mathbf{u}), \quad \mathbf{u}(t_0) = \mathbf{u}_0 \tag{1}$$

should accept a (vector valued) function $\mathbf{f}(t, \mathbf{u})$, the initial condition $\mathbf{u}_0$, an initial stepsize h_0, and a final time t_f, until which to integrate. Your adaptive integrator should return a solution curve $\{(t_i, \mathbf{u}_i)\}_i$.

(a) Implement an algorithm somewhat like this:

1. Compute one step with the improved Euler method
2. Compute the same step with the fourth-order Runge-Kutta method
3. Compare the results. If they are very close in magnitude, conclude that the stepsize is too small and increase it slightly, retaining the RK4 result. If they are very far apart in magnitude, conclude that the stepsize is too large, decrease it, and recompute the solution until both methods are very close in magnitude. Then retain the RK4 result.

Values you might use for the vague descriptions above are “very far apart" = differing by more than 25%, “very close” = within 1% of each other, “increase stepsize” = multiply by 1.2, “decrease stepsize” = multiply by 0.8.

(b) Test your implementation on the IVP

$$\frac{du}{dt} = \cos(t\, u^2(t)), \quad u(0) = 3.$$

Plot both the solution curve $u(t)$ and the stepsizes $h(t)$ on top of each other in the same plot. Use a linear axis for the solution and a logarithmic axis for the stepsizes.

Problem 2. The Van der Pol oscillator. (20 points)

Consider the following second-order ODE:

$$\frac{d^2x}{dt^2} - \mu(1 - x^2)\frac{dx}{dt} + x = f(t),$$

which is a harmonic oscillator with non-linear damping term controlled by the parameter μ and a time-dependent forcing $f(t)$.

(a) In the case of $\mu = 0$ and $f(t) = 0$, obtain an analytic solution with initial conditions $x(0) = 0, \dot{x}(0) = 1$. Sketch the trajectory $(x(t), \dot{x}(t))$ traced out in phase-space (the $\dot{x}$–x plane).

(b) Set $\mu = 5$, and for the same initial conditions as in part (a), use the adaptive integrator you wrote to solve the IVP from $t = 0$ to $t = 40$. As before, plot the solution curves $x(t)$, $\dot{x}(t)$, and the stepsize $h(t)$ on top of each other in the same plot. Again use a logarithmic axis for the stepsizes. How does the shape of the trajectory in phase space change?

(c) Repeat what you did in part (b) with the forcing function

$$f(t) = \frac{\pi}{2} + \arctan(1000 \sin t).$$

Problem 3. Avoiding catastrophic loss of precision. (20 points)

Consider the following family of integrals, which appear in the solution of electromagnetic scattering problems:

$$I_n(x) = \frac{x^{n+1}e^x}{n!} \int_0^1 u^n e^{-ux}\, du,$$

with x a real number and n a non-negative integer.

(a) Evaluate the integral to obtain analytic formulas for $I_n(x)$ with $n = 0, 1, 2, 3$. (*Hint:* Integration by parts.)

(b) Write a program to evaluate these analytical formulas. Compute the results for $x = 10^{-4}$ and $n = 0, 1, 2, 3$.

(c) Now evaluate the same integrals using numerical quadrature with your code for the composite Simpson's rule with $N = 1000$ subdivisions.

(d) Compare your results from parts (b) and (c) to the following values, which are accurate to all shown decimals:

n	$I_n(10^{-4})$
0	`1.0000500016667083e-4`
1	`5.0001666708334167e-9`
2	`1.6667083341666806e-13`
3	`4.1667500013877985e-18`

Which method is more accurate and why?

(e) *Extra credit:* Rewrite your code from part (a) to obtain more accurate results by avoiding catastrophic loss of precision.

Problem 4. Recursive Newton-Cotes quadrature. (20 points)

In class we discussed *pairwise summation* as a method to improve the accuracy of summations with many summands of similar magnitude. Here, you will use the same method to cure accumulation of rounding errors in Newton-Cotes methods.

Based on your existing implementation of the composite Simpson's rule with N subdivisions, implement the following algorithm for numerically evaluating

$$\int_a^b f(x)dx.$$

- If the number of subdivisions is less than some base case threshold (say, $N = 100$), call your old composite Simpson's rule function to evaluate it.
- Otherwise, split the integral into one on the subinterval $[a, (a+b)/2]$ and one on $[(a+b)/2, b]$, and integrate each of them recursively using a total of N subdivisions, returning the sum. (Be careful about the case where N is odd.)

Use your original and recursive Simpson's method to evaluate

$$\int_1^2 \log^3(x)\, dx$$

for the number of subdivisions N between $[1, 10^8]$. Plot the relative error of both methods compared to the exact value, `0.10109738718799412444`. Briefly explain what you see and why.

Problem 5. Kahan summation. (20 points)

There are more algorithms for improving accuracy of sums than pairwise summation. It can be shown that the error for pairwise summation of N summands grows like $\mathcal{O}(\log N)$, which is very slow, but not optimal. William Kahan (who also invented the IEEE 754 floating point standard which basically all modern computers use) came up with the following ingenious algorithm for summing whose error is $\mathcal{O}(1)$, so it's constant! The basic idea is to split the sum into two numbers, one storing the "large part," and a *running compensation* storing the lower digits.

The sum $S_N = \sum_{n=1}^{N} s_n$ is constructed iteratively from the summands s_n using the intermediate steps,

$$\begin{aligned} y_n &= s_n - c_{n-1} \\ t_n &= S_{n-1} + y_n \\ c_n &= (t_n - S_{n-1}) - y_n \\ S_n &= t_n, \end{aligned}$$

with initial conditions $S_1 = 0$ and $c_1 = 0$. (Note that the parentheses are crucial. The computations must be performed in exactly the given order).

(a) Show algebraically that $S_N = \sum_{n=1}^{N} s_n$ in exact arithmetic.

(b) Compute the sum $10000.0 + 4.14159 + 1.71828$ (i) exactly, and then round the result to 6 significant digits. (ii) in 6-digit decimal floating point arithmetic by naively summing all the summands one after the other, (iii) in 6-digit decimal floating point arithmetic using Kahan summation. Explain how Kahan summation is able to compensate for the loss of precision due to rounding errors to find the correctly rounded result.

(c) Implement Kahan summation in Julia (in the same function form as PSET 1, Problem 2), and evaluate

$$P(N) = \sum_{n=1}^{N} \frac{\pi}{N}$$

for N in $[10, 10^9]$. Plot the relative error $|P(N) - \pi|/\pi$ and compare the naive method from PSET 1, Problem 2, the pairwise summation algorithm from class, and Kahan summation.

Problem 6. *Extra credit:* The representable set. (10 points)

Visualize the representable sets for floating point number systems. For instance, use 2 decimal digits in the mantissa and one decimal digit in the exponent, and plot the representable set between $[0, 1]$, $[1, 2]$, $[1, 10]$, $[10, 100]$.