Problem Set 2

Physics 305, Fall 2020
Due Sept 14 at 5.00pm

Problem 1. One-dimensional Trapezoidal Method [15 Points] The one-dimensional trapezoidal rule can be summarized as

$$\int_a^b f(x) = h \sum_{i=0}^{N-1} \frac{(f(x_i) + f(x_{i+1})}{2} = \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right] \tag{1}$$

where $x_i = a + i \cdot h$, $i = 0, 1, \ldots n$, with $h = (b-a)/n$, and where the form on the right hand side of Eq. (1) minimizes the number of function evaluations required. This can be implemented as follows:

(a) INPUT: $f(x)$, end points a, b, number of steps n (a positive integer), optional arguments for the integrand.
OUTPUT: approximation to the definite integral I.
(b) Set $h = (b-a)/n$.
(c) Set $I_{part1} = f(a) + f(b)$; $I_{part2} = 0$. #We will save $\sum f(x_i)$ in I_{part2}
(d) For $i = 1, \ldots n-1$ do steps 5 and 6.
  (e) Set $x = a + i \cdot h$
  (f) Set $I_{part2} += f(x)$
(g) Set $I = h/2(I_{part1} + 2 \cdot I_{part2})$.
(h) Return (I)

- Write a Python function `trapezoidalRule(func, a, b, *P)` to compute the definite integral of the given function `func` between the given limits `a` and `b`. Use this function to compute the definite integral

$$\int_0^1 xe^{-\alpha x}\, dx$$

for $\alpha = 2$. [4 Points]

- Run your code for $\mathbf{n} = \mathbf{50}, \mathbf{100}, \mathbf{200}, \mathbf{400}$. Evaluate the error in your approximation (by comparison with the analytic result), and plot the relative error as a function of $\mathbf{n}$ on a log-log plot, and compare with the expected order of convergence, which in this plot corresponds to a slope of $1/n^2$. [4 Points]
- Find an approximate n such that the numerical answer is accurate to at least five significant figures. [2 Points]
- Change step (d) of the algorithm to **For $i = 0, \ldots n$ do steps 5 and 6.**. Does the order of convergence match the expected order ($O(1/n^2)$) or do you obtain a slope in your plot which is equal to -1? Provide an analytic explanation. [5 Points]

Problem 2. Two-dimensional Trapezoidal Method [10 Points] Derive the trapezoidal method for double integrals in rectangular domains, which is given by comptrap2D where $x_i = a + ih_x,\ i = 0, 1, \ldots n$, with $h_x = (b-a)/n$, $y_j = c + jh_y,\ j = 0, 1, \ldots m$, with $h_y = (d-c)/m$, and where n and m can be either odd or even.

Note: You don't need to write a code for this problem. Show your work.

Problem 3. Total charge of an equal charge electric dipole [15 Points] An electric "dipole" consisting of two equal point charges q separated by a distance d has an electric field whose radial component far away from the source is given by

$$E_{\hat{r}} = \frac{1}{4\pi\epsilon_0}\left(\frac{2q}{r^2} + \frac{p\sin\theta\cos\phi}{r^3}\right). \tag{2}$$

where $p = qd$ is the magnitude of the dipole moment. Based on Gauss law's law the **total** charge inside a distant sphere of radius R is

$$Q = \epsilon_0 \int_{\mathcal{S}} E_{\hat{r}} R^2 \sin\theta d\theta d\phi = \frac{q}{4\pi}\int_0^{\pi}\int_0^{2\pi}\left(2 + \frac{d}{R}\cos\phi\sin\theta\right)\sin\theta d\theta d\phi \tag{3}$$

Build a code that implements the trapezoidal rule to perform the double integral appearing in the previous expression and find the total charge Q enclosed by the sphere assuming $d/R = 1/10$. Run your code for $n_\theta = n = n_\phi = m = 50, 100, 200, 400$. Does the value you obtain numerically converge to the expected result? If yes, at what order? Include a plot of the relative error vs n demonstrating the order of convergence in your solutions document.

Dealing with Extra Arguments

When writing mathematical software, it often occurs that one needs to pass a function as an argument to another function. You have already seen the need to do this in the previous problem set where you needed to pass a function to your root-finding routine. In this problem set you will need to pass a function to a numerical integration routine.

When the function depends only upon one argument this is easy. For example, to integrate

```
def cube(x):
    return x**3
```

we can simply pass it as an argument to the integration function

```
def integrator(f,a,b):
    # call the function as:
    q = f(x)
    # etc.
```

Using this looks like

```
answer = integrator(cube, a, b)
```

If the function takes more than one argument, the situation can get a bit more complicated. Take, for example the one-dimensional Gaussian distribution

$$G(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}}\, e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{4}$$

which we might code as

```
def Gaussian(x, mu, sig):
    return 1/(sqrt(2*pi)*sigma) * np.exp( -(x-mu)**2/(2*sigma**2) )
```

We might want to integrate this function over its first argument, but we need to be able to specify the remaining two arguments.

There are two ways to deal with this situation. The first is just to define another, intermediate function which sets the values of the unchanging parameters

```
def gaussvals(x):
    return Gaussian(x, 10, 0.1)
```

(of course, we could have specialized the Gaussian function itself). We can then pass `gaussvals` to our integration routine as before

```
ans=integrator(gaussvals, 0, 1)
```

The problem is that this requires us to modify the code of `gaussvals` every time we wish to use new arguments. As an example, suppose we wish to find the value of σ such that the integral of the Gaussian between certain limits was a given value. Our program would need to change the x argument of `Gaussian` to do the integral, and it would need to change the σ argument to do the root-find. We need a way to specify the extra arguments under program control.

A more elegant way of dealing with this is to use Python's argument-packing syntax. If `P` is a tuple, then `*P` in an argument list when calling a function unpacks the contents of the tuple into individual arguments. To specify the mean and standard deviation for our Gaussian function, we could thus write

```
P = (10, 0.1)
ans = Gaussian(x, *P)
```

and the `*P` unpacks into `10, 0.1` – the function is given three arguments in all.

When defining a function, the syntax `*P` means "pack all remaining arguments into the tuple `P`." Using this syntax, we might write our integrator function as

```
def integrator(func, a, b, *P):
    # and then use the integrand as
    blah = func(x,*P)
    # and so on...
```

We could then call the integrator using `Gaussian` as the integrand as

```
ans = integrator(Gaussian, 0, 1, 10, 0.1)
```

the last two arguments will be passed on to `Gaussian`. We can still integrate our `Cube` function as

```
ans = integrator(cube, 0, 1)
```

Note that to use this mechanism, the tuple-packing mechanism must be used *after* and mandatory arguments; in this case, these arguments are the function to be integrated and the limits of integration.

Use this tuple-packing syntax wherever you need to pass one function to another.