Lab #6: Numerical Integration in *Matlab*

Introduction

When working with engineering mathematics, it is not unusual to encounter integrals that are difficult or impossible to evaluate manually (i.e., via a "pencil-and-paper" method). This is certainly the case when evaluating the coefficients that appear in weighted sums of orthogonal eigenfunctions that, as we will see, form all or part of the solutions of partial differential equations. For example, we have already seen that the eigenfunction solutions of a Sturm-Liouville problem can form a basis set to express any square-integrable function $f(x)$ on the interval $[a, b]$ as

$$f(x) = \sum_{n=1}^{\infty} a_n y_n(x) \quad \text{where} \quad a_n = \frac{\langle f(x), y_n(x) \rangle}{\langle y_n(x), y_n(x) \rangle}.$$

The inner products are determined by evaluating the integral

$$\langle f(x), g(x) \rangle = \int_a^b f(x)\, g(x)\, p(x)\, dx,$$

where $p(x)$ is a weighting function that comes from the self-adjoint form of the differential equation.

Inner products can often be evaluated manually if the functions are elementary types like trigonometric functions or polynomials. Even then, they can be challenging. If the functions are special types like Bessel functions or more esoteric varieties, then they can be extremely challenging or impossible to solve manually. In those cases, one can turn to numerical integration. Even if a manual solution is possible, numerical integration is a convenient way to check whether the manual evaluation of the integral was completed correctly.

Before beginning, download the *Matlab* script `Lab6start.m`, which is available at the course Moodle site in the "Lab Materials" section. You should set up a separate folder on your own computer and/or in your Bucknell private Netspace for your ENGR 695 lab activities.

Background

Matlab has many functions that can perform numerical integration. Most are based on some form of *adaptive quadrature*. Quadrature, simply speaking, refers to finding the area under a curve. The origin of the word is attributed to ancient Greek mathematicians, who tried to determine the lengths of the sides of a square that had the same area as a circle. The term *quadrature* is a reference to the four equal sides of a square. In modern adaptive quadrature methods, the area under a curve is broken up into small sections, typically trapezoidal in shape. The widths of the trapezoids under different parts of the curve are adjusted until the specified accuracy is obtained. The area under smooth and slowly varying parts of the curve might be accurately determined using trapezoids with relatively wide widths, whereas more numerous narrow trapezoids might be required under those parts of the curve that are rapidly changing or are oscillatory.

It is probably no surprise that a wide variety of adaptive quadrature algorithms have been proposed over the years, and they all have their advantages and disadvantages and claimed levels of accuracy. The *Matlab* function `quadgk` uses Gauss-Kronrod quadrature, which over time has proven to be a robust and accurate method. We will apply `quadgk` in this lab exercise. If you are interested in learning about the details of the Gauss-Kronrod method, you should be able to find good expositions in advanced textbooks on numerical analysis. There is also a Wikipedia page on the method that could serve as a starting point.

The *Matlab* quadrature functions fall into a special class of functions that take other functions as arguments. That is, they are "functions of functions." Care must be taken to let the function `quadgk` know what function to integrate. The basic syntax of the command is

```
q = quadgk(fun,a,b),
```

where `q` is the numerical value obtained when the integral is evaluated, `a` and `b` are the integration limits (explicit numerical values or variables that have been assigned values earlier), and `fun` is a function *handle* associated with the function to be evaluated.

What is a handle? Briefly, it is a numerical identifier that tells a computer program where in the computer's memory a block of code begins. Since the entity `fun` in this case is not a simple value but an entire function, the function `quadgk` needs to know where to find it. The function's handle specifies the location.

Fortunately, function handles are easy to specify in *Matlab*. A simple example showing how to numerically evaluate the definite integral

$$\int_2^4 x^2\,dx$$

should illustrate how to use handles. First, we need to define the function to be evaluated. One way to do that is to write a new m-file or to add a function definition to the bottom of the m-file in which `quadgk` is called. For example:

```
function y = xsquared(x)
y = x.^2;
end
```

Note that the vectorized power operator (`.^` instead of `^`) is used here. That is because, as stated in the *Matlab* help, the integrand function used in the `quadgk` command "should accept a vector argument X and return a vector result Y, the integrand evaluated at each element of X."

The function handle is simply the function name with the "at" symbol (@) appended to the beginning. In the example above, the function handle would be `@xsquared`. Thus, the complete function call to evaluate the definite integral would be

```
q = quadgk(@xsquared, 2, 4)
```

In this particular case, we can easily evaluate the integral manually:

$$\int_2^4 x^2\,dx = \frac{1}{3}x^3\Big|_2^4 = \frac{1}{3}\left[(4)^3 - (2)^3\right] = \frac{1}{3}(64-8) = \frac{56}{3} = 18.667\,,$$

where the result is given to five digits of accuracy. The actual answer is 18 2/3. We can use this result to check whether we have correctly applied the `quadgk` command. The result given by *Matlab* is 18.667 (again to five digits of accuracy).

In many cases, like the example above, defining a separate function just to evaluate an integral is unnecessarily complicated. Fortunately, *Matlab* provides a way to define simple functions in a single line, thus avoiding the need to write a separate block of code. *Matlab* calls them *anonymous functions*, and their syntax is probably best described by providing an example. The following line of code defines the function x^2 using an anonymous function:

```
xsquared2 = @(x) x.^2;
```

The variable `xsquared2`, of course, is the function name, but it is also the function's handle. This distinction will become apparent soon. The variable(s) listed in parentheses after the @ symbol are the inputs, that is, the independent variable(s) on which the function operates. Only one independent variable is used in this simple example, but there could be almost any number. If there are more than one, they are separated by commas. The code following the parentheses defines the function, the vectorized form of x^2 in this case. Again, we define the function in vectorized form because that is what the `quadgk` command requires.

After it has been defined, the function `xsquared2` can be used like any other function. Besides providing it to the `quadgk` command, we could use it by itself as a standalone function. For example, if we were to type `xsquared(16)` at the *Matlab* command prompt, we would get the result `ans = 256` because 256 is equal to 16^2.

The function call to evaluate the definite integral using the anonymous function `xsquared2` would be

```
q = quadgk(xsquared2, 2, 4)
```

Note that in this case the @ symbol is not added to the beginning of `xsquared2` because the anonymous function's name is already a function handle. That is an important difference that you need to keep in mind. If you add the @ symbol to the name of an anonymous function, *Matlab* will produce an error message.

More information on anonymous functions is available in both the online and the built-in *Matlab* documentation. The latter feature is obtained by selecting the "Documentation" option in the "Help" pull-down menu in the ribbon at the top of the *Matlab* main window.

Procedure

The *Matlab* script `Lab6start.m` is very short. The first few lines implement the simple examples described in the "Background" section. You should run the unmodified code once to verify that the examples work on your computer. The remaining sections of the script consist of two comments that indicate where you should add new code as described in the steps listed below. There is also a function (`xsquared`) at the end that is used in one of the examples above.

Take some time to familiarize yourself with the script `Lab6start.m` and understand how it works, and then complete the following steps:

1. Find the text "Your Name" immediately following the date in the header of the script, and add your name after the colon.

2. Add code after the first comment line marked by three asterisks (***) to evaluate the inner product of the eigenfunctions that form the solution to the boundary value problem

   $y'' + a^2 y = 0 \quad$ with $\quad y'(0) = 0 \quad$ and $\quad y(\pi) = 0$.

   The solution consists of the eigenvalues and eigenfunctions

   $$a_n = \frac{2n-1}{2} \qquad y_n(x) = \cos(a_n x) \qquad \text{for } n = 1, 2, 3, \ldots$$

   The inner product evaluates the integral of $\cos(a_m x)\cos(a_n x)$ for two integer values m and n over the appropriate interval. One way to write the code that defines the anonymous function needed by the `quadgk` command is

   ```
   m = 1;
   n = 2;
   am = (2*m – 1)/2;
   an = (2*n – 1)/2;
   integrand = @(x) cos(am*x).*cos(an*x);
   ```

   Writing the anonymous function in this way allows you to change the eigenvalues and eigenfunctions corresponding to different indices m and n easily. You need only change the values of m and n, and the code takes care of the rest. The variables `am` and `an` are parameters rather than independent variables, but anonymous functions can include previously defined parameter values as shown in the example above. The parameter values `am` and `an` are passed to the function `quadgk` along with the function `integrand`.

3. Use the new code that you have just written and a call to `quadgk` to confirm that the inner product of two cosine eigenfunctions with $m \neq n$ equals zero. You might not actually get a result of zero, but if it is on the order of 10^{-17} or something similarly small, that result is essentially zero to within round-off error. Next, try computing a few self-products. All self-products should be equal to $\pi/2$, regardless of the value of n.

4. On your own, add code after the second comment line marked by three asterisks (***) to evaluate the inner product of the eigenfunctions that form the solution to the Bessel equation-based boundary value problem

   $$x^2y'' + xy' + \lambda x^2 y = 0 \quad \text{with} \quad y'(0) = 0 \quad \text{and} \quad y(5) = 0,$$

   which is similar to a problem that we solved as an example in class. The solution consists of the eigenvalues and eigenfunctions

   $$\lambda_n = \left(\frac{r_n}{5}\right)^2 \qquad y_n(x) = J_0\left(\sqrt{\lambda_n}x\right) \qquad \text{for } n = 1, 2, 3, \ldots,$$

   where r_n is the nth root of $J_0(x)$. The first four roots are $r_1 = 2.4048$, $r_2 = 5.5201$, $r_3 = 8.6537$, and $r_4 = 11.7915$. Consider defining a vector `r` to contain the roots so that you do not have to retype them every time you change index numbers. Be careful when you define the integrand for the inner product. Remember the weighting function $p(x)$.

5. Use the new code that you have just written to confirm that the inner product of two eigenfunctions with $m \neq n$ equals zero. (It will actually be around 10^{-5} or 10^{-6} because the roots above have only five or six-digit accuracy.) Check your code further by computing the self-products for the first four eigenfunctions. The results should be

   $$\left\langle J_0\left(\sqrt{\lambda_1}x\right), J_0\left(\sqrt{\lambda_1}x\right)\right\rangle = 3.36900 \qquad \left\langle J_0\left(\sqrt{\lambda_3}x\right), J_0\left(\sqrt{\lambda_3}x\right)\right\rangle = 0.92109$$
   $$\left\langle J_0\left(\sqrt{\lambda_2}x\right), J_0\left(\sqrt{\lambda_2}x\right)\right\rangle = 1.44724 \qquad \left\langle J_0\left(\sqrt{\lambda_4}x\right), J_0\left(\sqrt{\lambda_4}x\right)\right\rangle = 0.67547$$

6. Remove the weighting function $p(x)$ from the inner product integrand for the Bessel function (that is, set $p(x) = 1$), and verify that the result is not zero when $m \neq n$. Restore the weighting function before submitting the final version of your *Matlab* m-file.

7. Save a copy of the m-file with your code added and with your name added under the header. It should be obvious how to change the index numbers in the code to evaluate the various inner products. Add comments to explain anything that is not obvious. Change the name of your edited m-file to `LName_Lab6_fa23.m`, where `LName` is your last name, and then e-mail the file to me.

Lab Scoring and Submission Deadline

Your score will be based primarily on your submitted *Matlab* script and will be assigned according to the rubric posted on the Laboratory page at the course web site.

If you do not complete the exercises during the lab session, then you may submit your *Matlab* m-file as late as 11:59 pm on Friday, October 27. If the file is submitted after the deadline, a 5% score deduction will be applied for every 24 hours or portion thereof that the item is late (not including weekend or fall break days) unless extenuating circumstances apply. No credit will be given five or more days after the deadline.