Adaptive Quadrature Methods

Adrian Alvarez
Dewayne Broome
Colin Cox
Emmanuel Vargas

Introduction

Adaptive quadrature, or adaptive numerical integration, refers to the process of approximating the integral of a given function to a specified precision by adaptively subdividing the integration interval into smaller sub-intervals over which a set of local quadrature rules are applied. The first publication of adaptive quadrature routines was around 50 years ago as proposed by G.F. Kuncir in 1962. Since then there are about 20 distinct algorithms that have been published. Adaptive simpson method is a method of numerical integration that could be considered the first recursive adaptive algorithm for numerical integration but thanks to the hardwork of future scholars many other methods have been creative. That uniqueness is the sole reason that mathematics has continued to be revolutionized for centuries to come. In addition to creating new algorithms we are also able to refine the technique of many methods especially the adaptive simpson method. We gathered motivation to pursue adaptive quadrature directly from its accuracy of giving the benefit of reducing error to below a threshold as needed to be able to obtain necessary accuracy, while also saving computational resources by only achieving the desired threshold. Throughout this text we plan to dwell deeper into adaptive quadrature in these key elements; mathematics, pseudo code, and comparing it different methods using iterations and complex functions. With this method we plan on understanding the usage of adaptive quadrature methods and applying our research to other topics in the future.

Mathematics

Adaptive quadrature is a numerical integration method, this method calculates an integral by dividing the integration domain into subintervals. Quadrature in this method stands for the region the integration is carried out on. To understand this, if the function defined as f(x) has values that are real and can be defined in a finite interval from a to b. Then the integral's function from the intval will look like:

$$\int_{a}^{b} F(x)\mathrm{dx}$$

Adaptive quadrature has the same approach as integration. Integration being the accumulation of subintervals for a given area under the curve. Additive property is a base of adaptive quadrature, this property is used to calculate the integral. The property will need to have a interval from a to b where c is the point between a and b, and the integral calculated, then you will find the integration to look like:

$$\int_a^b F(x)\mathrm{dx} = \int_a^c F(x)\mathrm{dx} + \int_c^b F(x)\mathrm{dx}$$

From this formula above, if the value of the integral is calculated, its sum could either have a desired output or something not desired. In the case where the output is not desired, you would apply the additive property on the intervals [a,c] and [c,b]. If the error is much bigger than anticipated within the interval from a to c, then this is where you would subdivide the interval into smaller subintervals to lessen the overall error. If the integrated function within the given limit is changing swiftly such that the curve is having abrupt changes. Then this tells us that the quadrature will be smaller therefore having less of a chance of having an error. But in the case of the curve not changing abruptly, then this is where you would have to select the regions of integration carefully so it does not change the function much to the point it is treated as a big quadrant.

Overall, Adaptive quadrature methods utilizes the composite quadrature rule with the error approximations. The interval is divided into smaller intervals, then using a rule called the quadrature rule to compute the integral of each subinterval, and after adding the results to approximate. A function f(x) will have different behaviors, one can change rapidly in one region and the other could change slowly in the other. Because the meshes cannot be the same, Fine and coarse meshes are used depending on the functions behavior. When the integrand is changing rapidly, and the mesh is uniform, a fine mesh is used. And if the integrand is changing very slowly, coarse mesh is used. when the entire integration domain is divided into sub-parts, the error output from composite integration should equal the overall error.

Procedure break down of method:

Approximate the integral $\int_a^b$ f(x) dx within some specified tolerance (ε)

say $\epsilon > 0$

Calculate the integral by applying an adaptive method on Simpson's rule.

Apply Simpson's rule with h = b - a

$$\int_a^b \mathrm{f(x)\ dx} \approx \mathrm{I}(h)$$

$$\mathrm{I(a,b)} = \frac{h}{6}[\mathrm{f(a)}+4\mathrm{f}(\frac{a+b}{2})+\mathrm{f(b)}]$$

Calculate the accuracy of the approximation, this can be done by dividing the mesh in half then solving for I(h)

$$Q(h) = I_{\frac{h}{2}} + E_{\frac{h}{2}}, \text{ Weddle's rule}$$

$$E_{\frac{h}{2}} = \frac{1}{15}[I_{\frac{h}{2}} - I_{h}]$$

From this process of using adaptive quadrature rule, next thing to do is to use $E_{\frac{h}{2}}$ to determine if the output of $I_{\frac{h}{2}}$ is accurate, in other words if the error is in the specific range it needs to be in.

If the output is accurate, then $I_{\frac{h}{2}}$ should approximate $\int_a^b F(x)dx$ within ε.

If the output is not accurate, then repeat process individually on subintervals $[a,\frac{a+b}{2}]$ and $[\frac{a+b}{2},b]$

Where the intervals are divided again, each subinterval with a error should have a total error of $\frac{\varepsilon}{2}+\frac{\varepsilon}{2} = \varepsilon$.

Now, if the subinterval gives a error greater than $\frac{\varepsilon}{2}$ then it will be halved again. And if the calculation is more then $\frac{\varepsilon}{4}$ and so on, this process needs to be continued until the subintervals satisfy the error output.

Here is an example:

$$\int_0^{\frac{\pi}{4}} x^2 sin(x)dx$$

Approximating within $\varepsilon = 10^{-3}$

This will be function

$f(x) = x^2 sin(x)$

Interval from a to b

Calculating $I(0,\frac{\pi}{4})$ using 2 interval Simpson's rule

$I1(0,\frac{\pi}{4}) = \frac{h}{6}[f(0)+4f(\frac{\pi}{8})+f(\frac{\pi}{4})]$, plug in to function and solve.

$I1(0,\frac{\pi}{4}) = .0880$

Repeating process now with a 4 interval Simpson's rule

$I2(0,\frac{\pi}{4}) = \frac{h}{3}[f(0)+4f(\frac{\pi}{16})+2f(\frac{\pi}{8})+4f(\frac{3\pi}{16})+f(\frac{\pi}{4})]$, plug in to function and solve.

$I2(0,\frac{\pi}{4}) = .0887$

Checking error within 10^{-3}

$E = \frac{1}{15}[\mathrm{I2}(0,\frac{\pi}{4}) - \mathrm{I1}(0, \frac{\pi}{8})]$

$\frac{1}{15}|.0887 - .0880|$

Here is the approximation to within 10^{-3}

$.0000466 < .001\ (\frac{\pi}{4} - 0)$

$.0000466 < .000785$

Since the condition is met, there is no need to explore new intervals and the integral approximation is 0.0887.

<u>**Code**</u>

The algorithmic methods for computing Adaptive Quadratures are relatively simple with only a few nuances that can be discussed upfront. First, we will introduce a new data structure utilized by the algorithm, a stack. Stacks are actually quite simple; they are considered to be a Last In First Out (LIFO) structure. This data organization is accomplished with two main functions, pop() and push(x). We can think of the structure as a list of numbers that grows from the bottom up. The push(x) function will add a new element, x, to the top of the stack. The pop() function will return the element on the top of the stack and remove it from the stack. Thus, by inserting and removing data from the same spot in the structure, the top, we create the LIFO behavior we desire. This structure is what we will utilize to store the potential intervals for our algorithm. Thus, in this case each element in the stack will actually be an array of two numbers a, the left endpoint of the interval and b, the right endpoint of the interval. We need this LIFO behavior when storing our intervals as this will allow us to easily grab the leftmost interval and proceed to calculate the integral from left to right.

The second nuance of the algorithm we will discuss is a special termination condition. Since we are optimizing interval selection until a desired error tolerance is achieved, it is possible, but not likely, that we will not be able to achieve the tolerance. When this is the case, we do not want our algorithm to run forever, so we must mitigate that. We will do that by simply adding an iteration counter and ensuring that we do not exceed a maximum number of iterations. Because most of these problems will terminate in under 100 iterations, we will set the max iterations to be 10,000. This number will allow many iterations more than usually necessary to give difficult problems a fair chance, but will also not occupy the user's resources for more than a few seconds.

The pseudo code for an Adaptive Quadrature method utilizing Simpson's Rule is shown below and will be briefly described. The algorithm will need to take in the left and right bound, the function to integrate, and the tolerance we want to achieve. The function will then return the approximation of the integral, which will be empty if it could not be computed. We will begin the algorithm by initializing our variables, the integral approximation and iterations to 0, and max iterations to our agreed 10,000 iterations. At this time we will also create a stack and push the original interval (a,b). We will then iterate until the stack is empty or we exceed 10,000 iterations. Within this loop we will pop an interval from the stack and then on that interval calculate a basic integral using a 2 interval Simpson's rule and a refined integral using a 4 interval Simpson's rule. With our integrals now approximated, we can then calculate our estimated error and compare it to the tolerance given. If the error is within the tolerance then we can proceed to add the refined integral to our approximation. If it wasn't within the tolerance then we will break the current interval into two subintervals and push them to the stack. Finally, after the loop we check to see if we exceeded the max number of iterations. If we did then we will discard our partially complete integral approximation and return an empty variable. Our algorithm can now terminate.

Pseudo Code for Adaptive Quadrature Using Simpson's Rule:

Note: // indicates a comment line

```
Inputs:                              Output:
f //function to integrate                 I //the approximation of
integral,   a //left endpoint                  //if empty, then
approximation
b //right endpoint                     //could not be computed within
tol //tolerance                        //reasonable time

Algorithm:
I = adaptive_quadrature_simpsons(f, a, b, tol)
     //Initializations
     I = 0;
     iterations = 0; //to count iterations done
     maxIterations = 10,000;
     create stack as a new empty stack data structure
     stack.push((a,b)) //push interval (a,b) to the stack

     while the stack is full and we don't exceed maxIterations
          //take the current interval
          (a,b) = stack.pop()
          //calculate basic and refined integral using midpoints
          basicIntegral = 2 interval Simpson's rule on (a,b)
          refinedIntegral = 4 interval Simpson's rule on (a,b)
          //check to see if our current integrals within tol
          if estimatedError < tol
               //add the refined integral to our answer
               I = I + refinedIntegral;
          else
               //if not within tol, push two new intervals
```

```
                //the order in which we push these is important
                stack.push((midPoint, b));
                stack.push((a, midpoint));
            end if
            //increment the number of iterations
            iterations = iterations + 1;
        end while
        //check if we got to maxIterations, if so empty I
        if iterations >= maxIterations
            I = empty;
        end if
    end adaptive_quadrature_simpsons()
```

Matlab Code:

We will now show a few snippets of code from our MatLab implementation of the above algorithm which can be tricky to implement. The entire code will be provided in the Appendix of this textbook. The first section we will discuss is the calculation of the basic and refined integrals. Below, first and last refer to the left and right endpoints of the current interval being selected. As discussed in other sections, to calculate the two interval Simpson's rule we must calculate the midpoint, and then compute as usual. This will be our basic integral. Next to calculate the four interval Simpson's rule we have to compute two more midpoints between first - midpoint and midpoint -last. We will call these subMidLeft and subMidRight to indicate they are the sub midpoints and to which side they correspond. We can now compute the four interval Simpson's rule as usual, resulting in the following code:

```
%now we will calculate a midpoint between the interval to refine
midpoint = (first + last) / 2;

%calculate integral using interval popped off the stack
%this interval will be done with a basic 2 interval rule
basicIntegral = ((last - first) / 6) * (f(first) +  4 * f(midpoint) + f(last));

%calculate the two sub midpoints necessary to compute 4 interval
%simpsons rule, this will be between the midpooint above and the
%endpoints
subMidLeft = (first + midpoint) / 2;
subMidRight = (midpoint + last) / 2;

%we will now calculate a refined integral using 4 intervals
refinedIntegral = (((midpoint - first) / 6) * (f(first) +  4 * f(subMidLeft) +
f(midpoint)) + (((last - midpoint) / 6) * (f(midpoint) +  4 * f(subMidRight) + f(last)));
```

Next we will cover the portion of the code that will check if the approximation is within the tolerance. Mathematically, we know that the error of the Simpson's approximation for this subinterval will roughly equal 1/15 * | basicIntegral - refinedIntegral |. To avoid an extra calculation and variable, we included this calculation in the if state condition as such: abs(basicIntegral - refinedIntegral). Instead of dividing this by 15, we simply multiply 15 to the tolerance and ensure the error is less than that. There is one more nuance we must consider, because this iteration is calculating the integral for the current subinterval then the tolerance has to be updated to only consider the error for this subinterval. Thus, we have to multiply the tolerance by (last - first), thus scaling the tolerance to check only this interval. Putting all this together we get the following code:

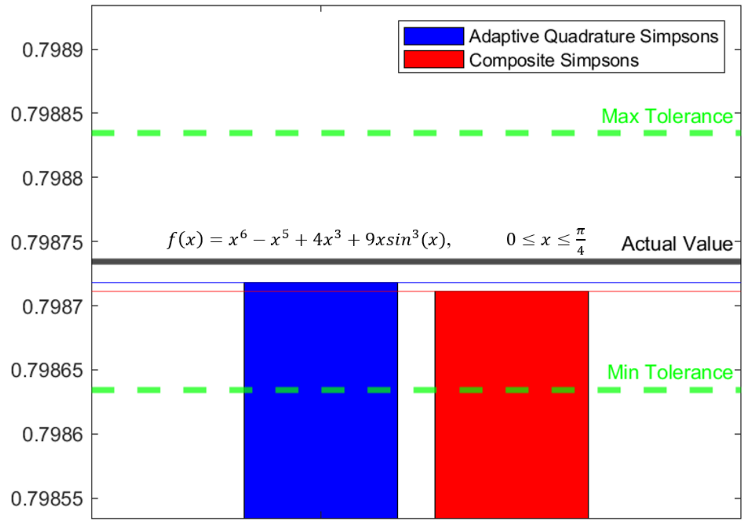
```
%check to see if we are within the tolerance
if abs(basicIntegral - refinedIntegral) < 15 * tol * (last - first)

    %add the refined integral to our final integral solution
    I = I + refinedIntegral;
```

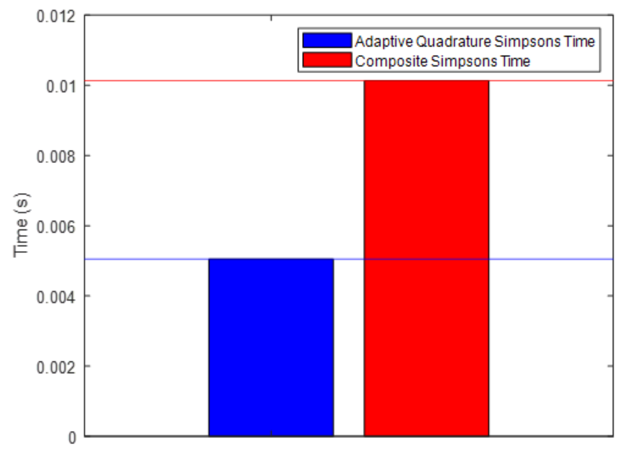
The remainder of the code is fairly straightforward and will not be discussed in more detail. We will now proceed to compare this algorithm to other numerical methods utilized to approximate integrals.

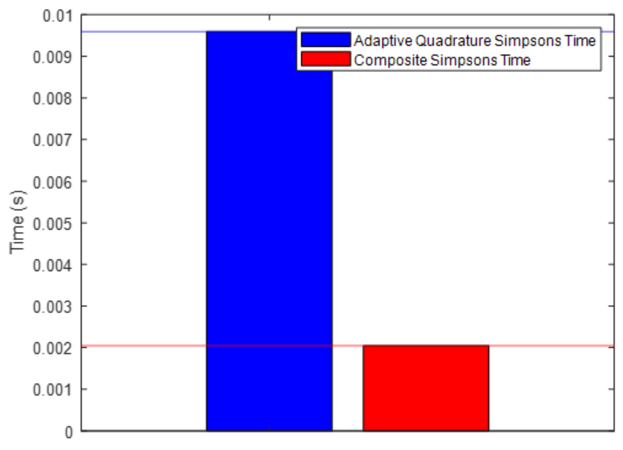
Comparisons

To determine whether the Adaptive Quadrature methods could perform better than the Compound Simpsons rule, we created a complex function, set the iterations to be equivalent, and plotted a bar graph to represent the accuracy of both methods. In many cases, the results were very similar to each other because the adaptive quadrature method of changing certain interval subsections had very little change in what the normal composite simpson's rule would calculate. However, when the equations become more complex and the interval approximations have larger errors, using the Adaptive Quadrature method reduces the error of the approximation.



We can see in the bar graph above, when computing the integral of f(x) from 0 to pi/4, the Adaptive Quadrature method for Simpson's rule has a closer approximation than the Composite Simpson's rule. The reason this is the case is due to the following reasons. With the Adaptive Quadrature methods, the portions of the approximation that are within the tolerance are untouched, and ranges where the approximation is outside the tolerance are adjusted. Alternatively, the Composite Simpson's rule adjusts all of the ranges if any portion of the approximation is outside the tolerance. In this scenario, the Adaptive Quadrature method is more accurate than the Composite Simpson's method, however this is not always true. For some of the example questions, both methods reached the same answer. Since this occurs, we can conclude that Adaptive Quadrature method for Simpson's rule is very similar to Composite Simpson's rule, however the biggest advantage for the Adaptive Quadrature Method is the time it takes to perform calculations compared to the Composite method for Simpson's rule.





For the Adaptive Quadrature method, the time to perform the calculations for the figure on the right was about 0.0050 seconds, while the Composite Simpson's rule function took about 0.0101 seconds. However, if we look at the right figure which is performed for a different equation, the time for the Adaptive Quadrature is longer. This leads us to the idea that the time performance of the two methods vary based on the iterations and function size. The trend is that the Adaptive Quadrature method takes slightly more time than the Composite Simpson's method. The Adaptive Quadrature method is a great way to make function approximations more precise, however in doing so it sacrifices a tiny bit of its processing speed.

Conclusion

As we dwell deeper into understanding the uses of Adaptive quadrature method and analysis, we find ourselves gaining more admiration for research and breaking barriers. The more we find ourselves understanding the subject the more we relate and foster a love for understanding complex algorithms. Throughout the process we dismantled the method of calculation for this numerical method. Understanding the procedure breakdown provides room for improvement and creation. Following the mathematical process we converted our physical knowledge into digital knowledge and observed the algorithm by writing pseudo code for Adaptive quadrature method utilizing the Simpsons rule. We provide examples of that in our screenshots of matlab code then we compared those methods with compound simpson's rule. All in all we came to understand that the Adaptive Quadrature method is an amazing tool to make function approximations more precise but it will take more time than the Composite Simpson's Method.

Resources

All Matlab source code discussed in the above chapter and some additional code for demonstrations and testing can be found at the google drive below:

https://drive.google.com/drive/folders/1dx7vBdSPFBzE9p9WbbIF-KF5PcLPhSuW?usp=sharing

Sources

Anonymous.“Adaptive Numerical Integration.” *Learn About Adaptive Numerical Integration Chegg.com*, 2003, Retrieved from www.chegg.com/learn/calculus/calculus/adaptive-numerical-integration?trackid=3d3bffe6a6d7&strackid=23128ea70a72

Gonnet, P. (n.d.). *A Review of Error Estimation in Adaptive Quadrature*. Retrieved from https://arxiv.org/pdf/1003.4629.pdf

Lambers, J. (2009). Adaptive Quadrature. In *math.usm.edu*. Retrieved May 6, 2021, from https://www.math.usm.edu/lambers/mat460/fall09/lecture30.pdf

A lecture from professor Jim Lambers that introduces the concept of Adaptive Quadrature and provides intuition to the creation of the algorithm for trapezoidal rule, as well as an in depth example of an integral utilizing Simpson’s adaptive quadrature. This example was utilized to assess the correctness of our own algorithm.

Zhang, Zhiqiang. (May 5, 2011). List, queue, stack. Version 1.1.0.0. Source Code. Retrieved from https://www.mathworks.com/matlabcentral/fileexchange/28922-list-queue-stack