Photo by Ray Reyes (https://unsplash.com/@rareyesphoto) on Unsplash (https://unsplash.com/)

Numerical Integration

3 minute read

In this lesson we will learn how to use `QuadGK` to perform numerical integration of uni-dimensional integrals. `QuadGK` performs an adaptive Gauss-Kronrod quadrature. You can find more information on the package here (https://github.com/JuliaMath/QuadGK.jl).

First of all we need to install `QuadGK`, to do it type in the REPL:

```julia
using Pkg
Pkg.add("QuadGK")
```

Once you have installed the library you are ready to perform numerical integration.

Calculations

The function used to perform numerical integration is `quadgk`. The first argument of `quadgk` is the function to be integrated, while the second and third are the integration boundaries. Let's compute a Gaussian integral:

```julia
using QuadGK

func1(x) = exp(-x^2)
res, err = quadgk(func1, -Inf, Inf)

>>> abs(res-sqrt(π))/sqrt(π)
1.25e-12
```

`quadgk` returns two values: the result of the integration and the estimated error. We know that the result of the Gaussian integral is:

$$\int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi}$$

If we compare the result of the integration with `sqrt(π)`, as it is done at line 6, we get a relative error of `1.25e-12`, which is a good approximation of the true result.

It is possible to get a better estimation of the value of the integral increasing the relative tolerance `rtol`. Basically, the integration continues until the relative difference between two successive refinement steps of the integration routine is less than `rtol`.

```
res, err = quadgk(func1, -Inf, Inf, rtol=1e-15)

>>> abs(res-sqrt(π))/sqrt(π)
2.51e-16
```

The Gauss-Kronrod formula is a modified version of the Gaussian quadrature (https://en.wikipedia.org/wiki/Gaussian_quadrature). This kind of algorithms have a parameter called the **order** of the quadrature rule which is linked to how complex the integral approximation scheme is. With a higher order integration rule, it is possible to integrate "exactly" polynomials of higher degree, so if we expect the argument of the integral to be pretty complex, a higher order of the integration rule may help achieving a better accuracy. That being said, the default order is 7 and it is generally enough to compute almost all the integrals you will face. In case you are having difficulty in achieving convergence, try increasing the order of the integration rule.

In our case, if we increase the order of the quadrature we can achieve a result coinciding with the analytical solution (with double-precision):

```
res, err = quadgk(func1, -Inf, Inf, order=12)

>>>abs(res-sqrt(π))/sqrt(π)
0.00
```

Increasing the order of the quadrature usually leads to longer integration time.

Functions with multiple arguments

Let's suppose we have a function which takes more than one argument:

```
func2(x, y, z) = x + y^3 + sin(z)
```

If we want to integrate the function in `y` from 1 to 3, and we want `x` and `z` to be fixed, we need to define an "argument" function in this way:

```julia
x = 5
z = 3
arg(y) = func2(x, y, z)

quadgk(arg, 1, 3)
```

In this case `arg` takes only one argument, which is the integrated variable.

Another option is to use anonymous functions:

```julia
quadgk(y -> func2(x, y, z), 1, 3)
```

Although the result is the same, in my opinion the first option is clearer and is particularly handy if we use a `let` block:

```julia
res, err = let x=5; z=3
    arg(y) = func2(x, y, z)

    quadgk(arg, 1, 3)
end
```

In this way we avoid fixing `x` and `z` outside of the `let` block (remember, a `let` block introduces a local scope and thus `x` and `z` won't be accessible outside of the `let` block) and we only get the result of the integration.

Another case where this notation is useful is when the integration appears inside a function:

```julia
func3(x,y) = x^2*exp(y)

function test_int(x, ymin, ymax)
    arg(y) = func3(x, y)
    return quadgk(arg, ymin, ymax)[1]
end

test_int(3, 1, 5)
```

For multidimensional integration, see the HCubature.jl (https://github.com/stevengj/HCubature.jl), Cubature.jl (https://github.com/stevengj/Cubature.jl), and Cuba.jl (https://github.com/giordano/Cuba.jl) packages.

Conclusions

In this lesson we have learnt how to compute uni-dimensional integrals numerically. Furthermore we have seen how it is possible to integrate functions which take multiple arguments.

If you liked this lesson and you would like to receive further updates on what is being published on this website, I encourage you to subscribe to the **newsletter** (https://techytok.com/newsletter/)! If you have any **question** or **suggestion**, please post them in the **discussion below**!

Thank you for reading this lesson and see you soon on TechyTok!

Tags: Julia

Categories: Lessons Tutorial

Updated: December 25, 2019

/