Yane Garcia

Mathematics 4670

Solve integrals using different methods

In the problem 15 of section 4.3, we have to find the value x of the following equation:

$$\int_0^x \frac{1}{\sqrt{2\pi}} e^{-t^2/2}\, dt = 0.45$$

We are going to use two numerical approaches. First, we need the Newton's method to find the zeros of the function f(x). Second, we can use (a) the Composite Simpson's rule or (b) the Composite Trapezoidal rule to estimate the integral.

To solve x using Newton's method, we can employ the following function:

$$f(x) = \int_0^x \frac{1}{\sqrt{2\pi}} e^{-t^2/2}\, dt - 0.45 = 0$$

Also, we have to derivate the previous function:

$$f'(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2}$$

The Newton's method is given by:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} = x_n - \frac{\int_0^{x_n} \frac{1}{\sqrt{2\pi}} e^{-t^2/2}\, dt - 0.45}{\frac{1}{\sqrt{2\pi}} e^{-x_n^2/2}}$$

In the part a of the problem 15 of section 4.3, we have to find a solution to f(x)=0 accurate to within 10^(-5) using Newton's method with p0=0.5, and the Composite Simpson's rule. We need to use the Composite Simpson's rule to find the value of that integral for the next pn in each iteration.

$$s = \int_a^b f(x) \approx \frac{h}{3}\left(f(a) + f(b) + 4\sum_{i=1}^{n/2} f(a+(2i-1)h) + 2\sum_{i=1}^{(n-2)/2} f(a+2ih)\right)$$

The code for this problem is:

```
!Newton's method
implicit none
double precision:: p0, p, fp,f2, TOL
integer:: i
p0=0.5
TOL= 10**(-5.0)
do i=1,500
```

```
p = p0-f2(p0)/fp(p0)
if(abs(p-p0) <TOL) then
print*, "p", i, "=",p
stop
end if
p0= p
end do
print*, p
stop
end

function f2(b)
implicit none
double precision:: simpson, b
real:: f2
f2=simpson(b) - 0.45
return
end

function fp(p)
implicit none
double precision::p
real::fp
real, parameter :: Pi = 3.1415927
fp=( (1/ (sqrt(2*Pi) ))*(EXP( (-p**2)/2)) )
return
end
!end Newton

!Simpson's rule
double precision function simpson(b) result(sim)
implicit none
double precision:: a, b, se, so, integrate, fv, dx
integer:: n, i
real::sim
a= 0.0d0
```

```
n= 20
sim = 0.0d0
dx=(b-a)/float(n)
se= 0.0d0
so= 0.0d0
do i= 1, n-1
fv=integrate(a+float(i)*dx)
if(mod(i,2)==0)then
se= se + fv
else
so= so + fv
end if
end do
sim= ((dx/3.0)*(integrate(a)+integrate(b)+2.0*se+4.0*so))
return
end

!f function
double precision function integrate(p) result (f)
implicit none
double precision::p
real::f
real, parameter :: Pi = 3.1415927
f=(1/(sqrt(2*Pi)))*(EXP((-p**2)/2))
return
end
!end Simpson's rule
```

I will show the output.

p6= 1.64485326590

The initial starting point is p0=0.5, and the accurate is 10−5. We get p6= 1.64485326590 with n=20.

In the part b of the problem 15 of section 4.3, we have to find a solution to f(x)=0 accurate to within 10^(-5) using Newton's method with p0=0.5, and the Composite Trapezoidal rule. We need to use the Composite Trapezoidal rule to find the value of that integral for the next pn in each iteration.

$$\int_a^b f(x)dx \approx \frac{h}{2}\left[f(a) + 2\sum_{k=1}^{n-1} f(x_k) + f(b) \right]$$

The code for this problem is:

```
!Newton's method
implicit none
double precision::p0, p, fp,f2, TOL
integer:: i
p0= 0.5
TOL= 10**(-5.0)
do i=1,500
p = p0-f2(p0)/fp(p0)
if( abs(p-p0)<TOL) then
print*,"p", i, "= ", p
stop
end if
p0=p
end do
print*,p
stop
end

function f2(p)
implicit none
double precision:: p, trapazoide
real:: f2
f2=trapazoide(p)-0.45
return
end

function fp(p)
implicit none
double precision::p
real::fp
real, parameter :: Pi = 3.1415927
```

```
fp=((1/(sqrt(2*Pi)))*(EXP((-p**2)/2)))
return
end
!end Newton's method

!Composite Trap
double precision function trapazoide(b) result(trap)
implicit none
integer:: i, n
double precision:: a, b, integrate, dx
real::trap
a= 0.0d0
n= 40
trap = 0.0d0
dx=(b-a)/dble(n)
do i=1, n-1
trap = trap + integrate(a+dble(i)*dx)
end do
trap= ((dx/2.0d0)*(2.0d0*trap + integrate(a) + integrate(b)))
return
end

!f function
double precision function integrate(p) result (f)
implicit none
double precision::p
real::f
real, parameter :: Pi = 3.1415927
f=(1/(sqrt(2*Pi)))*(EXP((-p**2)/2))
return
end
!end Composite Trap
```

I will show the output.

p6= 1.64508522547

The initial starting point is p0=0.5, and the accurate is 10−5. We get p6= 1.64508522547 with n=40.

1. In this exercise, we have to use the composite Trapezoid rule, the composite Simpson's rule and Romberg quadrature to estimate the following function:

$$\int_0^2 f(x)\,dx$$

In the above function, if 0<=x<=1, f(x)=1-x^3 and if x>=1, f(x)=(x-1)^2. In this case, the function is continuous; nevertheless, it is not differentiable at x=1. For this reason, we have to use and else-if statement where we say, if x>=0 and x<=1, we use f(x)=1-x^3, else if x>=1, then we use f(x)=(x-1)^2. We can make an else statement to be sure that our number is not less than 0. If the number is less than 0, we can show a message saying the number is negative. The value is not in the range.

The code for the composite trapezoid rule is:

```
implicit none
double precision:: a, b, integrate, dx
integer:: i, n
real::trap
a=0.0d0
b=2.0d0
n=40
trap = 0.0d0
dx= (b-a)/dble(n)
do i=1, n-1
trap = trap + integrate(a+dble(i)*dx)
end do
trap= ((dx/2.0d0)*(2.0d0*trap + integrate(a) + integrate(b)))
print*,trap
end

!f function
double precision function integrate(p) result (f)
implicit none
double precision::p
real::f
```

```
if(p>=0 .AND. p<=1) then
f=(1-p**3)
else if (p>1) then
f=(p-1)**2
else
print*, "The p value, ", p, " is negative."
end if
return
end
```

The output using this code is:

1.08312

We get 1.08312 using the composite Trapezoid rule and having 40 subdivisions.

The code for the composite Simpson's rule is:

```
implicit none
integer:: i, n
double precision:: a, b, se, so, fv, integrate, dx
real::sim
a= 0.0d0
b= 2.0d0
n= 20
sim = 0.0d0
dx=(b-a)/float(n)
se= 0.0d0
so= 0.0d0
do i= 1, n-1
fv= integrate(a+float(i)*dx)
if(mod(i,2)==0)then
se= se + fv
else
so= so + fv
end if
end do
sim= ((dx/3.0)*(integrate(a)+integrate(b)+2.0*se+4.0*so))
print*, sim
end
```

```
!f function
double precision function integrate(p) result (f)
implicit none
double precision::p
real::f
if(p>=0 .AND. p<=1) then
f=(1-p**3)
else if (p>1) then
f=(p-1)**2
else
print*, "The p value, ", p, " is negative."
end if
return
end
```

The output using this code is:
1.08333

We get 1.08333 using the composite Simpson's rule and having 20 subdivisions.

The code for Romberg quadrature is:

```
implicit none
integer::i, n, j
double precision:: trapazoide
double precision, allocatable, dimension(:,:) :: T
real::botton
n=5
allocate( T(1:n,1:n))
do i=1, n
T(i,1)= trapazoide(i)
end do
do j=2,n
botton=((4.0d0**(j-1))-1.0d0)
do i=j,n
T(i,j)=(T(i,j-1)+(1.0d0/botton)*(T(i,j-1)-T(i-1,j-1)))
end do
end do
```

```
print*,T(n,n)
deallocate(T)
stop
end

!Composite Trap
double precision function trapazoide(i) result(trap)
implicit none
integer:: i, i0, n
double precision:: a, b, integrate, dx
real::trap
a= 0.0d0
b= 2.0d0
n=2**(i-1)
trap = 0.0d0
dx=(b-a)/dble(n)
do i0=1, n-1
trap = trap + integrate(a+dble(i0)*dx)
end do
trap= ((dx/2.0d0)*(2.0d0*trap + integrate(a) + integrate(b)))
return
end

!f function
double precision function integrate(p) result (f)
implicit none
double precision::p
real::f
if(p>=0 .AND. p<=1) then
f=(1-p**3)
else if (p>1) then
f=(p-1)**2
else
print*, "The p value, ", p, " is negative."
end if
return
```

end

The output using this code is:

1.08333506242

We get that T(5, 5)= 1.08333506242 using the Romberg quadrature and having 5 subdivisions.

Playing with these three programs and having 6 subdivisions, the composite trapezoid rule produces the value 1.07407. The composite Simpson's rule gives us 1.02881, and the Romberg quadrature produces 1.08333333164. An interesting behavior that we can report is that while the composite trapezoid rule and the composite Simpson's rule give us a value with five decimal places, the Romberg quadrature produces a value with eleven decimal places.

I am going to compare the Trapezoidal, Simpson's rule, and the Romberg quadrature approximations to

a) $\int_0^1 \left(1 - x^3\right) dx$

b) $\int_1^2 (x-1)^2 dx$

c) $\int_0^2 f(x)\, dx$

f(x)	a	b	c
Exact value	0.75	1/3	1.0833
Trapezoidal	0.743056	0.337963	1.07407
Simpson's	0.750000	0.333333	1.02881
Romberg	0.750000000000	0.333333333333	1.08333333164

After this table, we can support that the Romberg quadrature is the more accurate method because its values are closest ones to the exact values.

3. I modified the Composite Simpson's rule in a straightforward manner for use it in the approximation of multiple integrals. The double integral is:

$$\iint_R f(x, y)\, dA$$

where R is a rectangular region in the plane:

R= {(x, y) | a<=x<=b, c<=y<=d}

Using the Composition Simpson's rule, we suppose that nx and ny are chosen to determine the step size dx= (b-a)/nx and dy= (d-c)/ny. We first write the double integral as an iterated integral:

$$\int_a^b \left(\int_c^d f(x,y)\,dy \right) dx$$

We use the Composition Simpson's rule to approximate treating x as a constant.

$$\int_c^d f(x,y)\,dy$$

The use of approximation methods for double integrals is not limited to integral with rectangular regions of integration. We can do approximate double integrals with variable inner limits:

$$\int_a^b \left(\int_{\text{alpha}(x)}^{\text{beta}(x)} f(x,y)\,dy \right) dx$$

I developed a two dimensional Simpson's rule approximation, and I implemented it in a code. I selected an example from the textbook. The code is the following:

```
!main program
implicit none
integer:: nx, ny
double precision:: simpson
double precision:: a, b
a= 0.1d0
b= 0.5d0
nx= 10
ny= 10
print*, "nx= ", nx, " ny= ", ny
print*," "
print*, "double integral  ", simpson(nx,ny,a, b)
stop
end

!simpons
double precision function simpson(nx, ny,a, b) result(sim)
implicit none
integer:: i, nx, ny
double precision:: a, b, se, so, inner, fv, dx
real::sim
```

```
sim=0.0d0
dx=(b-a)/float(nx)
se=0.0d0
so=0.0d0
do i= 1, nx-1
fv= inner(a+float(i)*dx, ny)
if(mod(i,2)==0)then
se= se + fv
else
so= so + fv
end if
end do
sim=((dx/3.0d0)*(inner(a, ny)+inner(b, ny)+2.0d0*se+4.0d0*so))
end

!inner function
double precision function inner(x, ny) result(isim)
implicit none
double precision:: x, dy, alpha, beta, f, fuv, se, so
integer:: ny, j
real::isim
isim=0.0d0
dy=((beta(x)-alpha(x))/float(ny))
se=0.0d0
so=0.0d0
do j= 1, ny-1
fuv= f(x, alpha(x) + dy * float(j))
if(mod(j,2)==0)then
se= se + fuv
else
so= so + fuv
end if
end do
isim= ((dy/3.0d0)*(f(x,alpha(x))+f(x,beta(x))+2.0d0*se +4.0d0*so))
return
end
```

```
!f function
double precision function f(x, y)
implicit none
double precision:: x, y
f= EXP(y/x)
return
end

!alpha function
double precision function alpha(x)
implicit none
double precision::x
alpha=x**3
return
end

!beta function
double precision function beta(x)
implicit none
double precision::x
beta=x**2
return
end
```

I will show the output.

nx=10 ny=10

double integral 3.330546128190E-02

Using Composite Simpson's rule with nx=ny= 1, the function f(x, y)=e^(y/x) produces the value 3.330546128190E-02. In this example, a=0.1, b=0.5, alpha(x)=x^3, and beta(x)=x^2.

2. In this program, we let R be the region described by 0<=x<=π/4 and sin(x)<=y<=cos(x). Our mass density function is:

$$f(x, y) = \sqrt{x}\, e^{x^2+y^2}$$

The center of mass(x,y), where x=My/M, and y=Mx/M.

Using our two dimensional Simpson's rule from the previous question, we can get the three double integrals: M, My, and Mx.

$$M = \int\int_R f(x,y)\,dA$$

The code for M using 12 subdivisions in the x variable and 14 in the y variable is the following:

```
!main program
implicit none
integer:: nx, ny
double precision:: simpson
double precision:: a, b
real, parameter :: Pi = 3.1415927
a= 0.0d0
b= Pi/4.00
nx= 12
ny= 14
print*, "M= ", simpson(nx,ny,a, b)
stop
end

!simpon's
double precision function simpson(nx, ny,a, b) result(sim)
implicit none
integer:: i, nx, ny
double precision:: a, b, se, so, inner, fv, dx
real::sim
sim= 0.0d0
dx=(b-a)/float(nx)
se= 0.0d0
so= 0.0d0
do i= 1, nx-1
fv=inner(a+float(i)*dx, ny)
if(mod(i,2)==0)then
se= se + fv
```

```
else
so= so + fv
end if
end do
sim=((dx/3.0d0)*(inner(a, ny)+inner(b, ny)+2.0d0*se+4.0d0*so))
end

!inner function
double precision function inner(x, ny) result(isim)
implicit none
integer:: ny, j
double precision:: x, se, so, alpha, beta, f, fuv, dy
real:: isim
isim= 0.0d0
dy=((beta(x)-alpha(x))/float(ny))
se= 0.0d0
so= 0.0d0
do j= 1, ny-1
fuv= f(x, alpha(x) + dy * float(j))
if(mod(j,2)==0)then
se= se + fuv
else
so= so + fuv
end if
end do
isim= ((dy/3.0d0)*(f(x,alpha(x))+f(x,beta(x))+2.0d0*se +4.0d0*so))
return
end

!f function
double precision function f(x,y)
implicit none
double precision::x,y
f=sqrt(x)*EXP(x**2+y**2)
return
end
```

```
!alpha function
double precision function alpha(x)
implicit none
double precision::x
alpha=sin(x)
return
end

!beta function
double precision function beta(x)
implicit none
double precision::x
beta=cos(x)
return
end
```

I will show the output.

M= 0.366926240966

The code for M using 12 subdivisions in the x variable and 14 in the y variable is like the code for Mx and My. There are some differences such as the output, while in the M program appears “M =”, it appears a “Mx= “ in the Mx program. However, the most important difference is in the f(x,y) function.

In My, we need to multiply f(x,y) by x:

$$M = \int\int_R x f(x,y)\, dA$$

I am only going to show the code of the f function.

```
!f function
double precision function f(x,y)
implicit none
double precision::x,y
f=x*(sqrt(x)*EXP(x**2+y**2))
return
end
```

I will show the output.

My= 0.137387544117

In Mx, we need to multiply f(x,y) by y:

$$M = \int\int_R yf\,(x,y)\,dA$$

I am only going to show the code of the f function.

```
!f function
double precision function f(x,y)
implicit none
double precision:: x, y
f=y*(sqrt(x)*EXP(x**2+y**2))
return
end
```

I will show the output.

Mx= 0.247464609803

We have M= 0.366926240966, My= 0.137387544117, and Mx= 0.247464609803. So, we can calculate x with the formula x=My/M; we get x= 0.374428233. We can find y with the formula y=Mx/M; we get y=0.674426035. To conclude, the center of mass using 12 subdivisions in the x variable and 14 in the y variable is (0.374428233, 0.674426035).

To estimate the center of mass using one hundred subdivisions for each variable, we only have to change the number subdivisions in the x variable(nx), and the number subdivisions in the y variable(ny) to one hundred. We can find nx and ny at the beginning of the program in the main function.

In this case, we have M= 0.368853071902, My= 0.137373586743, and Mx= 0.248585870494. Again, we can calculate x with the formula x=My/M; we get x= 0.3724344386. We can find y with the formula y=Mx/M; we get y=0.6739427957. To conclude, the center of mass using 100 subdivisions in the x variable and 100 in the y variable is (0.3724344386, 0.6739427957).