1. In this exercise you will be asked to write code to evaluate a function, and then apply
the trapezoidal method and Simpson’s rule to evaluate its integral over the interval
(0, 1). The main challenge here will be writing the code to evaluate the function. This
function is a classical construction of a very rough continuous function (there are
infinitely many points where the function’s derivative does not exist).

I am purposefully describing the function in graphical terms, rather than giving you an
explicit formula or algorithm, since developing such an algorithm is the main challenge
of this problem.

The function is constructed by summing up an infinite sequence of piecewise linear
functions. The first three such functions are shown on the next page. Each of the
triangles depicted is an isosceles triangle. The triangles at a given level all have the
same height, and the height at level \( k \) is \( \frac{1}{k^{q-1}} \), where \( q \) is a parameter to be specified.
The graphs on the next page depict levels \( k = 1, 2, 3 \).

(a) Derive a formula for the exact value of the integral. The formula may take the
form of an infinite series. Write code to evaluate this series (truncating once the
terms become negligible). Your formula and code should work for a general value
of \( q \).

**Solution:** There are \( 2^{k-1} \) triangles at level \( k \), each of which has area \( \frac{1}{(2 \cdot 3^k \cdot k^{q-1})} \).
Therefore the exact value of the integral is

\[
\sum_{k>0} 2^{k-2}/(3^k \cdot k^{q-1}).
\]

For code, see the `main` function in my solution to part b.

(b) Write code to evaluate the function described above, and implement the trape-
zoidal method and Simpson’s rule to numerically approximate the value of the
integral.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* Evaluate the function at x. */
double fun(const double x, const double q)
{
    double y=x, h=1;
```
int iy, k=1;

while (1)
{
    /* Figure out which segment we’re in. If it’s the middle one, 
     * return the correct height. */
    iy = floor(3*y);
    if (iy == 1) return h*(1 - 2*fabs(3*y-1.5));

    /* Otherwise scale up to the next level. */
    ++k;
    h = exp(-(q-1)*log(k));
    y = 3*y - iy;

    /* Just in case. */
    if (h < 1e-12) return 0;
}

/* Never reach here. */
return 0;
}

/* Apply the trapezoidal method to approximate the value of the integral. */
void Trapezoid(const double T,
               const double q)
{
    double A, h, x;
    int k;

    /* Starting approximation. */
    A = (fun(0, q) + fun(1, q)) / 2;

    /* Starting mesh. */
    h = 1;

    /* Double the number of quadrature points 20 times. Here we are 
     * updating the approximation rather than starting from scratch 
     * at each iteration. */
    for (k=0; k<20; ++k)
    {
        /* The mesh shrinks by a factor of two each time. */
        h /= 2;
        A /= 2;
    }
/ * Cycle through the new grid points. */
x = h;
while (x <= 1)
{
    A += h*fun(x, q);

    /* Advance to the next point, skipping one point which was included in the previous rule. */
x += 2*h;
}

/* Print out the log_10 absolute error and the log_10 ratio of absolute error to squared mesh. */
printf("Trapezoidal: \%5.1f \%5.1f\n", log(fabs(A-T))/log(10),
   log(fabs(A-T)) - 2*log(h))/log(10);
}

/* Apply Simpson’s method to approximate the value of the integral. */
void Simpson(const double T,
    const double q)
{
    double h, A, x;
    int k, ii;

    /* Starting mesh. */
h = 0.5;

    /* Double the number of quadrature points 20 times. */
for (k=0; k<20; ++k)
{
    h /= 2;

    A = h*(fun(0, q)/3.0 + fun(1, q)/3.0);

    x = h;
    ii = 0;
    while (x < 1)
    {
        A += h*fun(x, q) * ((ii % 2 == 0) ? 4/3.0 : 2/3.0);
        ++ii;
        x += h;
    }
printf("Simpson: %5.1f %5.1f\n", log(fabs(A-T))/log(10),
       (log(fabs(A-T)) - 3*log(h))/log(10));

}
}

int main(int argc, char** argv)
{
    double T, u;
    int k;
    double q = 3;

    /* Get a direct approximation to the exact value. */
    T=0, k=1;
    while (1)
    {
        u = exp(k*log(2.0/3.0) - (q-1)*log(k))/4;
        if (fabs(u) < 1e-15) break;
        T += u;
        ++k;
    }
    printf("T=%f\n", T);
    Trapezoid(T, q);
    Simpson(T, q);
    return 0;
}

(c) Study the convergence behavior of the trapezoidal method and Simpson’s rule to the direct numerical approximation from part a. Compare the two methods, and assess whether the theoretical convergence behavior derived in class for smooth integrands is operating in this case. Consider a small value of $q$, say $q = 1.5$ and a larger value, say $q = 10$. Don’t try $q = 1$ since the approximation in part a converges too slowly.

**Solution:** Both the trapezoidal method and Simpson’s rule are convergent. When $q = 3$, the trapezoidal method requires around half a million grid points to reach seven figures of accuracy. When $q = 3$, neither method converges as fast as the theory for smooth integrands predicts. The following tables show the results for twenty iterations. The first column is the log$_{10}$ absolute error, and the second column is the ratio of this error to the square mesh (trapezoidal rule) or the cubed...
mesh (Simpson’s method). Also note that Simpson’s rule and the trapezoidal method perform equivalently. For larger values of \( q \), the performance is more consistent with the theory for smooth integrands.

<table>
<thead>
<tr>
<th>Trapezoidal rule</th>
<th>Simpson’s rule</th>
<th># grid points</th>
</tr>
</thead>
<tbody>
<tr>
<td>-0.5</td>
<td>0.2</td>
<td>2</td>
</tr>
<tr>
<td>-1.4</td>
<td>0.4</td>
<td>4</td>
</tr>
<tr>
<td>-2.3</td>
<td>0.6</td>
<td>8</td>
</tr>
<tr>
<td>-2.4</td>
<td>0.7</td>
<td>16</td>
</tr>
<tr>
<td>-2.3</td>
<td>0.8</td>
<td>32</td>
</tr>
<tr>
<td>-3.0</td>
<td>0.9</td>
<td>64</td>
</tr>
<tr>
<td>-3.1</td>
<td>1.0</td>
<td>128</td>
</tr>
<tr>
<td>-3.3</td>
<td>1.0</td>
<td>256</td>
</tr>
<tr>
<td>-4.0</td>
<td>1.1</td>
<td>512</td>
</tr>
<tr>
<td>-4.5</td>
<td>1.1</td>
<td>1024</td>
</tr>
<tr>
<td>-4.8</td>
<td>1.2</td>
<td>2048</td>
</tr>
<tr>
<td>-5.1</td>
<td>1.2</td>
<td>4096</td>
</tr>
<tr>
<td>-5.1</td>
<td>1.3</td>
<td>8192</td>
</tr>
<tr>
<td>-5.1</td>
<td>1.3</td>
<td>16384</td>
</tr>
<tr>
<td>-5.4</td>
<td>1.3</td>
<td>32768</td>
</tr>
<tr>
<td>-6.6</td>
<td>1.3</td>
<td>65536</td>
</tr>
<tr>
<td>-6.9</td>
<td>1.4</td>
<td>131072</td>
</tr>
<tr>
<td>-7.4</td>
<td>1.4</td>
<td>262144</td>
</tr>
<tr>
<td>-6.9</td>
<td>1.4</td>
<td>524288</td>
</tr>
<tr>
<td>-7.2</td>
<td>1.4</td>
<td>1048576</td>
</tr>
</tbody>
</table>
2. In this problem you will use Gauss-Hermite quadrature to numerically approximate the expected values of certain random variables. Suppose that $X$, $Y$ and $Z$ are independent and have standard normal distributions. Let $A = (X + Y)^2$ and $B = 1/(X^2 + Y^2 + Z^2)$.

(a) Derive the expected values of $A$ and $B$ (hint: $B$ has an inverse gamma distribution).

**Solution:** Since $X + Y$ is normal with mean 0 and variance 2, $(X + Y)^2/2$ has a $\chi^2_1$ distribution, hence has expected value 1. Therefore $(X + Y)^2$ has expected value 2.

Since $X^2 + Y^2 + Z^2$ has a $\chi^2_3$ or $\Gamma(1, 3)$ distribution, it follows that $B$ has an inverse gamma $\mathcal{IG}(3/2, 2)$ distribution (may also be written $\mathcal{IG}(3/2, 1/2)$ depending on the parameterization), which has expected value 1.

(b) Use Gauss-Hermite quadrature to approximate these values, using 2, 4, 8, 16, and 32 quadrature points. Discuss the accuracy of the approximation in the context of the theoretical properties about Gaussian quadrature discussed in class.

**Solution:** Octave code for part b:

```octave
%% Integrate the function $2*(x+y)^2/pi$ with respect to the weight function $\exp(-x^2-y^2)$ using Gauss-Hermite quadrature. The input array qp
%% contains the quadrature points (abscissas in column 1, weights in column 2).
function v = IG1(qp)
    v = 0;
    for i=1:size(qp,1)
        for j=1:size(qp,1)
            fv = (qp(i,1) + qp(j,1))^2;
            v = v + fv*qp(i,2)*qp(j,2);
        end
    end
    v = 2*v/pi;
endfunction

%% Integrate the function $1/(2*pi^3/2*(x^2+y^2+z^2))$ with respect to the weight function $\exp(-x^2-y^2-z^2)$ using Gauss-Hermite quadrature. The input array qp contains the quadrature points (abscissas in column 1, weights in column 2).
function v = IG2(qp)
```
\[ v = 0; \]

\[
\text{for } i=1:\text{size}(qp,1) \\
\quad \text{for } j=1:\text{size}(qp,1) \\
\quad \quad \text{for } k=1:\text{size}(qp,1) \\
\quad \quad \quad \text{fv} = 1/(qp(i,1)^2 + qp(j,1)^2 + qp(k,1)^2); \\
\quad \quad \quad \quad v = v + fv*qp(i,2)*qp(j,2)*qp(k,2); \\
\quad \quad \text{end} \\
\quad \text{end} \\
\text{end} \\
\]

\[ v = v/(2*\pi^{1.5}); \]

\text{endfunction}

You’ll see that the results are exact for $A$, but not for $B$, as predicted by theory. To get a better idea about the convergence for $B$, we can suppose that $E_k \sim k^m$, where $E_k$ is the absolute error using $k$ quadrature points, and $m$ is an unknown constant that can be estimated via regression on the log/log scale, since $\log E_k \sim m \log k$. The correlation between $\log E_k$ and $k$ over $k = 2, 4, 8, 16, 32$ is $-0.99$, showing excellent agreement to the hypothesized convergence pattern. The estimated value of $m$ is $-0.45$. From this value it can be derived that the number of grid points must be increased by a factor of $2^{1/0.45} \approx 4.66$ to cut the error in half. This relatively poor performance is not unexpected since $B$ is very far from any low-degree polynomial.

Octave code for carrying out the regression analysis described above.

\[
\text{ii} = 1; \\
\text{for } k=[2,4,8,16,32] \\
\quad \text{eval(sprintf('load -force qp%d;', k));} \\
\quad \text{eval(sprintf('qp = qp%d;', k));} \\
\quad V(ii) = IG2(qp); \\
\quad ii = ii+1; \\
\text{endfor} \\
\]

\# Regress $\log(|\text{error}|)$ against the number of quadrature points.
\[
\text{E} = \log(\text{abs}(1-V)); \\
\text{h} = \log([2,4,8,16,32]); \\
\text{m} = \text{cov(E, h)} / \text{var(h)}; \\
\]

8
(c) Use Simpson’s method to approximate the two expected values, also using 2, 4, 8, 16 and 32 grid points (per axis). Compare to the performance to that of Gauss-Hermite quadrature.

Solution: The following Octave code calculates approximations to $EA$ and $EB$ using Simpson’s rule.

```octave
## Calculate the weight under Simpson’s rule for the ith quadrature
## point out of a total of nqp points when the mesh is h.
function w = sw(i, nqp, h)
    ii = (i==1) || (i==nqp);
    jj = (rem(i,2) == 1);
    w = ii*h/3 + 2*(1-ii)*jj*h/3 + 4*(1-ii)*(1-jj)*h/3;
endfunction

## Integrate the function $2e^{-(x^2+y^2)}(x+y)^2/\pi$ using Simpson’s rule
## with nqp quadrature points spaced evenly between -3 and 3.
function v = IG1_simpson(nqp)
    v = 0;
    h = 6/(nqp-1);
    for i=1:nqp
        x = -3 + (i-1)*h;
        wx = sw(i, nqp, h);
        for j=1:nqp
            y = -3 + (j-1)*h;
            wy = sw(j, nqp, h);
            fv = 2*exp(-x^2-y^2)*(x+y)^2/\pi;
            v = v + wx*wy*fv;
        end
    end
endfunction

## Integrate the function $f(x,y,z) = 1/(2\pi^{3/2}(x^2+y^2+z^2))$ using
## Simpson’s rule with nqp quadrature points spaced evenly between -3
## and 3 where h is the mesh.
function v = IG2_simpson(nqp)
    v = 0;
    h = 6/(nqp-1);
endfunction
```

```
for i=1:nqp
    x = -3 + (i-1)*h;
    wx = sw(i, nqp, h);
    for j=1:nqp
        y = -3 + (j-1)*h;
        wy = sw(j, nqp, h);
        for k=1:nqp
            z = -3 + (k-1)*h;
            wz = sw(k, nqp, h);

            fv = (2*pi)^-1.5*exp(-x^2/2-y^2/2-z^2/2)/(x^2+y^2+z^2);
            v = v + wx*wy*wz*fv;
        end
    end
end
endfunction

Again we can suppose that the error follows $E_k \sim k^m$ and $m$ can be estimated via regression. I get $m_1 \approx -3.1$ for integrand $A$ and $m_2 \approx -1$ for integrand $B$. This means that it takes roughly a 25% increase in the number of grid points to cut the error in half for integrand $A$, and roughly a doubling in the number of grid points to cut the error in half for integrand $B$. Here is the Octave code I used to calculate these values.

ii = 1;
E = [];
for k=[2,4,8,16,32]
    E(ii,:) = [abs(IG1_simpson(k)-2), abs(IG2_simpson(k)-1)];
    ii = ii+1;
endfor

F = [log([2,4,8,16,32]’), log(E)];
m1 = cov(F(:,1), F(:,2)) / var(F(:,1));
m2 = cov(F(:,1), F(:,3)) / var(F(:,1));