MATH 4220
Assignment # 6

Jarren Ralf

Due On: December 2, 2018

1 Prove the mean value theorem for integrals, as stated below:

Assume that $g \in \mathcal{C}[a,b]$ and that ψ is an integrable function that is either nonnegative or nonpositive throughout the interval $[a,b]$. Then there exists $\xi \in [a,b]$ such that

$$\int_a^b g(x)\psi(x)\,\mathrm{d}x = g(\xi)\int_a^b \psi(x)\,\mathrm{d}x.$$

Hint: use the Intermediate Value Theorem.

Proof. Assume that $g \in \mathcal{C}[a,b]$ and that ψ is an integrable function that is either nonnegative or nonpositive throughout the interval $[a,b]$. Suppose g is a bounded function on $[a,b]$. By definition of bounded, there exists $\hat{a}, \hat{b} \in [a,b]$ such that $g(\hat{a}) = \min g$ and $g(\hat{b}) = \max g$. Thus we have

$$\begin{aligned}
g(\hat{a}) \leq \quad & g(x) \quad \leq g(\hat{b}) \\
g(\hat{a})\psi(x) \leq \quad & g(x)\psi(x) \quad \leq g(\hat{b})\psi(x) \\
\int_a^b g(\hat{a})\psi(x)\,\mathrm{d}x \leq & \int_a^b g(x)\psi(x)\,\mathrm{d}x \leq \int_a^b g(\hat{b})\psi(x)\,\mathrm{d}x \\
g(\hat{a})\int_a^b \psi(x)\,\mathrm{d}x \leq & \int_a^b g(x)\psi(x)\,\mathrm{d}x \leq g(\hat{b})\int_a^b \psi(x)\,\mathrm{d}x \\
g(\hat{a}) \leq & \frac{\displaystyle\int_a^b g(x)\psi(x)\,\mathrm{d}x}{\displaystyle\int_a^b \psi(x)\,\mathrm{d}x} \leq g(\hat{b}).
\end{aligned}$$

Then, by the Intermediate Value Theorem, there exists $\xi \in [a,b]$, such that

$$g(\xi) = \frac{\displaystyle\int_a^b g(x)\psi(x)\,\mathrm{d}x}{\displaystyle\int_a^b \psi(x)\,\mathrm{d}x}.$$

This implies that

$$\int_a^b g(x)\psi(x)\,\mathrm{d}x = g(\xi)\int_a^b \psi(x)\,\mathrm{d}x.$$

□

2 Write a MATLAB program that computes an integral numerically, using the composite trapezoidal, midpoint and Simpson methods. Input for the program consists of the integrand, the ends of the integration interval, and the number of (uniformly spaced) subintervals. Apply your program to the two following integrals:

(a) $\int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x$

(b) $\int_0^1 \sqrt{x}\,\mathrm{d}x$

Use $r = 2, 4, 8, 16, 32$ subintervals. For each of the integrals, explain the behaviour of the error by observing how it is reduced each time the number of subintervals is doubled, and how large it is, compared to analytical bounds on the error. As much as you can, compare the performance of the methods to each other, taking into consideration both the error and the number of function evaluations. The exact values of the above integrals are π and $2/3$, respectively.

Listing 1: Function that approximates integrals using Trapezoidal, Midpoint, and Simpson's Rule

```
function [trap, mid, simp] = numericalIntegration(f, a, b, r)
% The function numericalIntegration accepts an anonymous function f, left endpoint of an
% interval, a, right endpoint b, and the number of sub-intervals r, then calculates the
% approximate definite integral. The program sums the partitions of the interval using the
% techniques of the composite trapezoidal rule, composite midpoint rule, and the composite
% Simpson's rule. The function outputs 3 scalar values in the above order, i.e. the function
% needs to be pass these numbers to 3 separate variables. A non-zero positive integer is
% expected for r, however, if a negative integer is sent to the function, it's sign will be
% changed. If r is zero or non-integer, then the program will quit and assign NaN to the
% output variables. Additionally, the user of this function should notice that Simpson's
% rule is intended to be executed with an even number of sub-intervals. If an odd number is
% used, an approximating is still calculated, however, it is less accurate in general.
%
%          f = The anonymous function that the user wishes to integrate
%          a = The LEFT endpoint of the interval we want to integrate over (real number)
%          b = The RIGHT endpoint of the interval we want to integrate over (real number)
%          r = The number of equidistant points we want to partition the interval with
%               (non-zero integer)

    % If r is 0 or a non-natural number, assign NaN to the ouput variables
    if (r == 0 || r - floor(r) ~= 0)
        disp('Error: r must be a natural number.');
        trap = NaN;
        mid  = NaN;
        simp = NaN;
        return
    end

    % If r is negative make it positive
    if (r < 0)
        disp('Sign changed for variable r.');
        r = (-1)*r;
    end

    % Set the length of the sub-interval
    h = (b - a)/r;

    % Create a vector of sub-interval points and sub-interval midpoints
    subPts = a:h:b;
    midPts = (subPts(:, 1:end - 1) + subPts(:, 2:end)) / 2;

    % Create a vector of quadrature abscissae for evaluating Simpson's method
    t2k   = subPts(3:2:end - 2);
    t2k_1 = subPts(2:2:end);

    % Trapezoidal, Midpoint, and Simpson's Rule
    trap = h/2*(f(a) + sum(2*f(subPts(2:end - 1))) + f(b));
    mid  = h  *(sum(f(midPts)));
    simp = h/3*(f(a) + 2*sum(f(t2k)) + 4*sum(f(t2k_1)) + f(b));
end
```

Listing 2: MATLAB code that approximates the integrals of given functions

```
% Set the integrands up as anonymous functions
f1 = @(x, y) 4./(1 + x.^2);
f2 = @(x) sqrt(x);

% Set the endpoints
a = 0;        b = 1;

% Choose the number of sub-intervals, r, and calculate sub-interval length, h
e = 1:5;      r = 2.^e;      h = (b - a)./r;

% Initialize the matrices of solutions
A1 = zeros(length(r), 3);
A2 = zeros(length(r), 3);

% Execute the integral approximations and fill the matrices
for i = 1:length(r)
    [t1, m1, s1] = numericalIntegration(f1, a, b, r(i));
    [t2, m2, s2] = numericalIntegration(f2, a, b, r(i));
    A1(i, :) = [t1 m1 s1];
    A2(i, :) = [t2 m2 s2];
end

% Find the appropriate derivatives for calculating the error estimations
d2f1 = matlabFunction( (-1)*diff( diff( f1(x)))); % Second derivative for trap and midpoint
d2f2 = matlabFunction( (-1)*diff( diff( f2(x))));
d4f1 = matlabFunction( (-1)*diff( diff( diff( diff( f1(x)))))); % Fourth deriv for Simpson's
d4f2 = matlabFunction( (-1)*diff( diff( diff( diff( f2(x))))));

% Find the maximum of the derivatives in order to use the error estimation formulas
[x1, fval_d2_1] = fminbnd(d2f1, a, b); % Notice that this function finds the minimum value,
[x2, fval_d2_2] = fminbnd(d2f2, a, b); % so that is why the functions above are multiplied
[x3, fval_d4_1] = fminbnd(d4f1, a, b); % by -1. We minimize -f which is equivalent to
[x4, fval_d4_2] = fminbnd(d4f2, a, b); % finding the max.

% Use matlab's built in function to determine an accurate value of each integral
q1 = integral(f1, a, b);
q2 = integral(f2, a, b);

% Print the results of function 1
fprintf('For function 1:\nThe true value of the integral is %0.17f\n\n', q1);
fprintf('r \t trapezoidal \t\t midpoint \t\t\t Simpson''s\n');
for i = 1:length(r)
    fprintf('%g \t %0.12f \t %0.12f \t %0.12f \n', r(i), A1(i, 1), A1(i, 2), A1(i, 3));
end
fprintf('\nERROR ESTIMATES:\n');
fprintf('r \t trapezoidal \t\t midpoint \t\t\t Simpson''s\n');
for i = 1:length(r)
    fprintf('%g \t %0.12f \t %0.12f \t %0.12f \n', r(i), abs(fval_d2_1)/12*(b - a)*h(i)^2, ...
                        abs(fval_d2_1)/24*(b - a)*h(i)^2, abs(fval_d4_1)/180*(b - a)*h(i)^4);
end
fprintf('\nABSOLUTE ERROR:\n');
fprintf('r \t trapezoidal \t\t midpoint \t\t\t Simpson''s\n');
for i = 1:length(r)
    fprintf('%g \t %0.12f \t %0.12f \t %0.12f \n', r(i), ...
                                abs(q1 - A1(i, 1)), abs(q1 - A1(i, 2)), abs(q1 - A1(i, 3)));
end

% Print the results for function 2
fprintf('\n\nFor function 2:\nThe true value of the integral is  %0.16f\n\n', q2);
fprintf('r \t trapezoidal \t\t midpoint \t\t\t Simpson''s\n');
for i = 1:length(r)
    fprintf('%g \t %0.12f \t %0.12f \t %0.12f \n', r(i), A2(i, 1), A2(i, 2), A2(i, 3));
end
fprintf('\nERROR ESTIMATES:\n');
fprintf('r \t trapezoidal \t\t midpoint \t\t\t Simpson''s\n');
for i = 1:length(r)
    fprintf('%g \t %0.12f \t %0.12f \t %0.12f \n', r(i), abs(fval_d2_2)/12*(b - a)*h(i)^2, ...
                        abs(fval_d2_2)/24*(b - a)*h(i)^2, abs(fval_d4_2)/180*(b - a)*h(i)^4);
end
fprintf('\nABSOLUTE ERROR:\n');
fprintf('r \t trapezoidal \t\t midpoint \t\t\t Simpson''s\n');
for i = 1:length(r)
    fprintf('%g \t %0.12f \t %0.12f \t %0.12f \n', r(i), ...
                                abs(q2 - A2(i, 1)), abs(q2 - A2(i, 2)), abs(q2 - A2(i, 3)));
end
```

```
For function 1:
The true value of the integral is 3.14159265358979312

r    trapezoidal      midpoint         Simpson's
2    3.100000000000   3.162352941176   3.133333333333
4    3.131176470588   3.146800518394   3.141568627451
8    3.138988494491   3.142894729592   3.141592502459
16   3.140941612041   3.141918174309   3.141592651225
32   3.141429893175   3.141674033796   3.141592653553

ERROR ESTIMATES:
r    trapezoidal      midpoint         Simpson's
2    0.166666662297   0.083333331148   0.014062500000
4    0.041666665574   0.020833332787   0.000878906250
8    0.010416666394   0.005208333197   0.000054931641
16   0.002604166598   0.001302083299   0.000003433228
32   0.000651041650   0.000325520825   0.000000214577

ABSOLUTE ERROR:
r    trapezoidal      midpoint         Simpson's
2    0.041592653590   0.020760287587   0.008259320256
4    0.010416183002   0.005207864804   0.000024026139
8    0.002604159099   0.001302076002   0.000000151131
16   0.000651041548   0.000325520719   0.000000002365
32   0.000162760415   0.000081380207   0.000000000037
```

```
For function 2:
The true value of the integral is  0.6666666666666666

r    trapezoidal      midpoint         Simpson's
2    0.603553390593   0.683012701892   0.638071187458
4    0.643283046243   0.672977397006   0.656526264793
8    0.658130221624   0.669032172130   0.663079280085
16   0.663581196877   0.667536675681   0.665398188628
32   0.665558936279   0.666982686478   0.666218182746

ERROR ESTIMATES:
r    trapezoidal      midpoint         Simpson's
2    0.005208684312   0.002604342156   0.000325569217
4    0.001302171078   0.000651085539   0.000020348076
8    0.000325542770   0.000162771385   0.000001271755
16   0.000081385692   0.000040692846   0.000000079485
32   0.000020346423   0.000010173212   0.000000004968

ABSOLUTE ERROR:
r    trapezoidal      midpoint         Simpson's
2    0.063113276073   0.016346035226   0.028595479209
4    0.023383620424   0.006310730339   0.010140401874
8    0.008536445042   0.002365505463   0.003587386582
16   0.003085469789   0.000870009014   0.001268478039
32   0.001107730388   0.000316019811   0.000448483920
```

For integral (a), $\int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x$, the error in each approximation is reduced when the number of sub-intervals is doubled. For the trapezoidal method, there is not quite one decimal place of accuracy gained with each iteration we performed, but rather maybe an average of half a decimal. The midpoint approximation actually behaves similarly, but, it is clearly a little bit better than the trapezoidal rule. Lastly, simpson's method clearly dominates the first two. Not only does it start with an approximation that is much closer than the previous two, each time the number of sub-intervals is doubled, more that two full decimal places are gained in accuracy each time. It is also worth noting that all of the methods outperform their own error estimations. What could be a possible explanation? Each of the error terms is based on either $f''(\xi)$ or $f''''(\xi)$, but since these values of ξ are unknown, we bounded the error by taking the maximum of the function. This clearly lead to an error estimate that is a bit too large. That being said, the estimates for trapezoidal and midpoint are only within one decimal place of the calculated absolute error. The estimate for Simpson's error is much different than the calculated version.

For integral (b), $\int_0^1 \sqrt{x}\,\mathrm{d}x$, increasing the number of sub-intervals indeed decreases the error. The trapezoidal and midpoint rules perform as well as they did for integral (a), however, simpson's rule under performs stunningly. It gains about half a decimal of accuracy, juxtaposed with it's performs of two full decimals for the first integral. In fact, simpson's method is a worse approximation of the integral in (b) than the midpoint rule. It does marginally outperform the trapezoidal rule however. The trapezoidal method gains less than half a digit of accuracy each time the sub-intervals are double. Again, it gives the worse performance of the three approximations. Each of the error estimates are different by a more significant amount compared to part (a). The same argument for why this is could be applied. For both trapezoidal and midpoint, the estimates seem to be less than one order of accuracy, we could postulate above one half. The midpoint error estimate is a couple digits off compared to the calculated absolute error. The trapezoidal rule, however, has an increasing disparity between the estimate and calculated error. It begins approximately one decimal different, and by the time the interval has been double four times, the absolute error is less than the estimated error by about two decimal places.

Let's break down how many function evaluations there are for each method. Starting with the trapezoidal rule, the formula is as follows

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{r-1} f(a+ih) + f(b)\right].$$

Observe that the sum in the middle of the formula will have $r-1$ terms to add up and for each of these, there is one function evaluation. Additionally, the left and right endpoints are evaluated, therefore we are left with a total of $r+1$ function evaluations. Next, for the composite midpoint rule, defined as

$$\int_a^b f(x)\,\mathrm{d}x \approx h\sum_{i=1}^{r} f(a+(i-1/2)h),$$

there are r terms in the sum, and thus r, function evaluations. Lastly, Simpson's method is

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{h}{3}\left[f(a)+2\sum_{k=1}^{r/2-1} f(t_{2k})+4\sum_{k=1}^{r/2} f(t_{2k-1})+f(b)\right].$$

The first sum yields $r/2-1$ function evaluations, while the second sum yields $r/2$. Along with the evaluations at the endpoints, we observe that the total function evaluations is $r+1$.

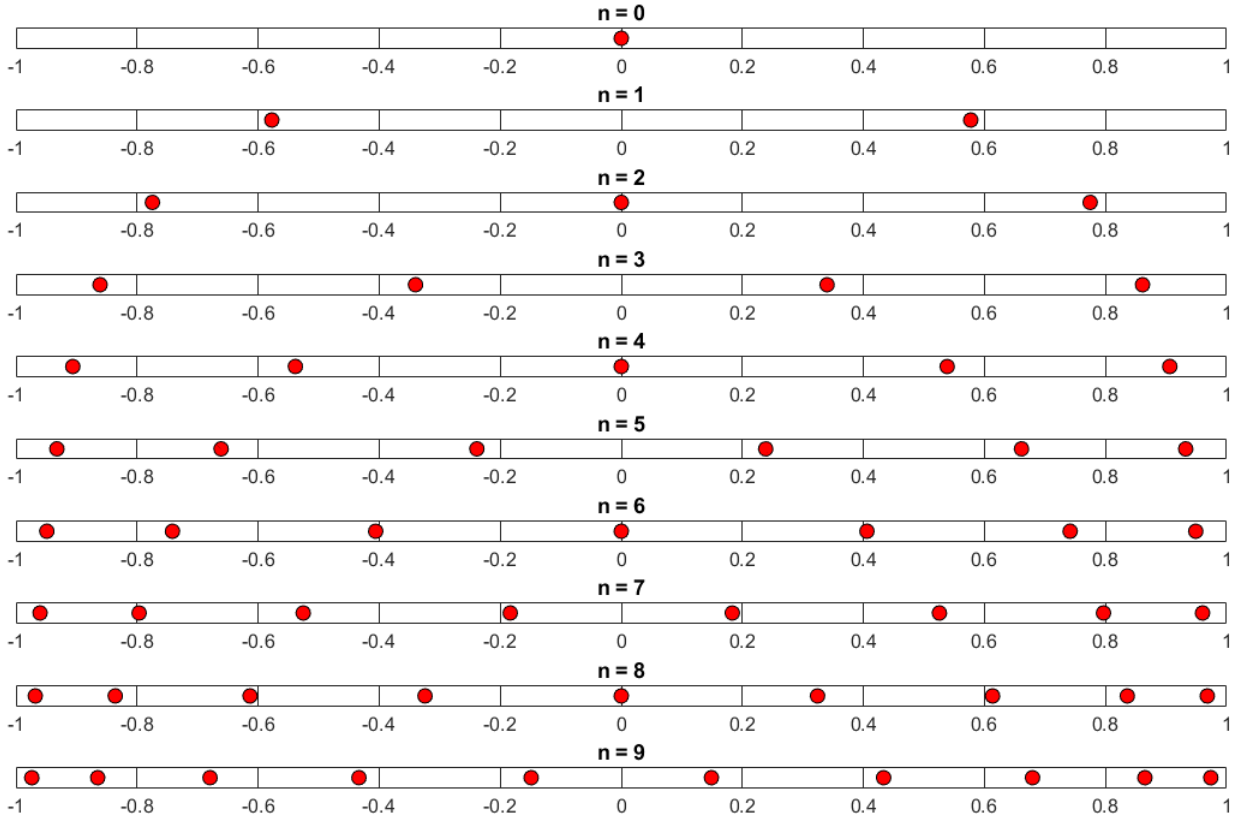
We know that however computationally expensive a calculation becomes, that function evaluations are more costly than elementary operations. Strictly on this criteria, the midpoint rule would be less computationally expensive than the other two. Observing the formula above, simpson's method is the most expensive of the given three methods because of the presence of a few more elementary operations. In conclusion, as long as simpson's rule is not significantly slower than the other two methods, then it is the best choice because of its high order of accuracy. Although, just like the example of integral (b), there would be some cases such that the midpoint approximation would be a more accurate choice, while simultaneously being the fastest method. The user should also consider whether they want to run the methods until a certain tolerance level is met. Because if this is the case, the fact that Simpson's method usually converges to the answer rather quickly suggest a fewer number of function evaluation would be executed. Of course, if the scenario was such that the midpoint rule was more accurate, then this method would have much fewer function evaluations because it already has one less per number of sub-intervals.

3 Write a short MATLAB program that will find the $n+1$ Gauss points on the interval $[-1,1]$ for each $n=0,1,\ldots,9$. You may use MATLAB's roots function. Display these points in one (nice looking) plot. Describe a composite quadrature method of order 20.

Listing 3: MATLAB code used to find $n+1$ the Gauss points on the gieven interval

```
% Set the values of n
n = 0:9;

% Initialize a square matrix of NaNs
% The advantage of this is that many functions ignore NaNs and if zeros() was used
% for example, the zeros in the matrix may be mistaken for actual Gauss points
R = NaN(length(n));

for i = n

    % Find the coefficients of the legendre polynomial up to order i + 1
    p = legendrepol(i + 1);

    % Find the roots of the legendre polynomial using the coefficients
    R(i + 1, 1:i + 1) = roots(p(i + 2, :));

    % Graph the interpolation for each value of epsilon
    subplot(length(n) + 1, 1, i + 1);
    plot(R(i + 1, :), 0, 'ok', 'MarkerFaceColor', 'r'); % plot() ignores NaNs by default
    set(gca, 'YTick',      []); % Remove YTicks and YTickLabels in order
    set(gca, 'YTickLabel', []); % to clean up the display of the graph
    axis([-1 1 -0.00000000001 0.00000000001])
    title(['n = ', num2str(i)]);
end
```

Figure 1: Graph of Gauss Points for various values of n



First, consider that our intuition for creating a higher order rule is using more subintervals. In particular, if we choose more Gauss points, then maybe we have a solution. Let's explore this strategy.

The general form of a weighted Gaussian quadrature rule is

$$\int_a^b f(x)w(x)\,\mathrm{d}x = \sum_{j=0}^{n} a_j f(x_j)$$

where the parameters x_j and a_j are determined by maximizing the precision of the integral. In order to to use this quadrature as a composite method, we evaluate f at the following distinct points and using the following chosen weights,

$$t_{i,k} = t_{i-1} + \frac{t_i - t_{i-1}}{2}(x_k + 1) \qquad \text{and} \qquad a_j = \frac{2(1-x_j^2)}{[(n+1)\phi_n(x_j)]^2}, \quad j = 0, 1, \ldots, n.$$

Notice that the $t_{i,k}$ are the x_j for the original Gaussian quadrature formula. Recall that this notation is used to describe the abscissae for composite methods. As long as the mesh is uniform, the error in obtaining the composite Gaussian rule is estimated by

$$E_{n,h} = \frac{(b-a)((n+1)!)^4}{(2n+3)((2n+2)!)^2} f^{(2n+2)}(\xi) h^{2n+2}.$$

We know that if we want a quadrature method of order 20, the exponent on the h needs to be 20. Thus, $2n+2 = 20 \implies 2n = 18 \implies n = 9$. This suggests that the first step for creating a composite Gaussian quadrature method of order 20, we need to find the Gauss points corresponding to the zeros of the degree 10 Legrende polynomial. Then the 11 Gauss points are scaled and used in each subinterval to form a set of $11r$ distinct points for r subintervals.

4 Suppose that the interval of integration $[a, b]$ is divided into equal subintervals of length h each such that $r = (b - a)/h$ is even. Denote by R_1 the result of applying the composite trapezoidal method with step size $2h$ and by R_2 the result of applying the same method with step size h. Show that one application of Richardson extrapolation, reading

$$S = \frac{4R_2 - R_1}{3}$$

yields the composite Simpson's method.

Proof. Suppose that the interval of integration $[a, b]$ is divided into equal subintervals of length h each such that $r = (b-a)/h$ is even. Denote by R_1 the result of applying the composite trapezoidal method with step size $2h$ and by R_2 the result of applying the same method with step size h. Then by using one application of Richardson extrapolation, we get

$$\begin{aligned}
S &= \frac{4R_2 - R_1}{3} \\
&= \frac{1}{3}\left(4 \cdot \frac{h}{2}\left[f(a) + 2\sum_{i=1}^{r-1} f(a+ih) + f(b)\right] - \frac{(2h)}{2}\left[f(a) + 2\sum_{i=1}^{\frac{r-1}{2}} f\Big(a + i(2h)\Big) + f(b)\right]\right) \\
&= \frac{h}{3}\left(2\left[f(a) + 2\sum_{i=1}^{r-1} f(a+ih) + f(b)\right] - \left[f(a) + 2\sum_{i=1}^{\frac{r-1}{2}} f(a+2ih) + f(b)\right]\right) \\
&= \frac{h}{3}\left[2f(a) + 4\sum_{i=1}^{r-1} f(a+ih) + 2f(b) - f(a) - 2\sum_{i=1}^{\frac{r-1}{2}} f(a+2ih) - f(b)\right] \\
&= \frac{h}{3}\left[f(a) + 4\sum_{i=1}^{r-1} f(a+ih) - 2\sum_{i=1}^{\frac{r-1}{2}} f(a+2ih) + f(b)\right] \\
&= \frac{h}{3}\left[f(a) + 4\sum_{i=1}^{\frac{r}{2}} f\Big(a + (2i-1)h\Big) + 2\sum_{i=1}^{\frac{r}{2}-1} f(a+2ih) + f(b)\right] \\
&= \frac{h}{3}\left[f(a) + 2\sum_{i=1}^{\frac{r}{2}-1} f(a+2ih) + 4\sum_{i=1}^{\frac{r}{2}} f\Big(a + (2i-1)h\Big) + f(b)\right].
\end{aligned}$$

Re-index the sums with k. Finally, denote the even points, $a + 2ih$ by, t_{2k} and the odd points, $a + (2i - 1)h$ by t_{2k-1}. Thus, we have derived

$$S = \frac{h}{3}\left[f(a) + 2\sum_{k=1}^{r/2-1} f(t_{2k}) + 4\sum_{k=1}^{r/2} f(t_{2k-1}) + f(b)\right].$$

Therefore, one application of Richardson extrapolation yields the composite Simpson's method. □

5 Using Romberg integration, compute to 8 digits (i.e. with a precision of $t = 8$) by obtaining approximations of the integral

$$\pi = \int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x$$

with MATLAB. Describe your solution approach and provide the appropriate Romberg table.

Listing 4: MATLAB code used to perform Romberg integration

```
% Set the integrands up as anonymous functions
f = @(x) 4./(1 + x.^2);

% Set the desired accuracy
s = 5;

% Set the endpoints
a = 0;
b = 1;

% Choose the number of sub-intervals
r = 1;

% Calculate the sub-interval length
h = (b - a)/r;

% Initialize the Lower Triangular Romberg Matrix (table)
R = NaN(s);

% Assigning the output of the numericalIntegration function to one variable
% will give us the trapezoidal rule
R(1, 1) = numericalIntegration(f, a, b, r);

for j = 1:s - 1

    % Set the element in the first column of the table
    % as you loop through starting with the second row
    h = h/2;
    R(j + 1, 1) = (1/2)*R(j, 1) + h*sum(f(a + h*(2*(1:r*2^(j - 1)) - 1)));

    % Calculate the rest of the elements in the row
    for k = 2:j + 1
        R(j + 1, k) = R(j + 1, k - 1)+(R(j + 1, k - 1) - R(j, k - 1))/(4^(k - 1) - 1);
    end
end
```

Table 1: Romberg Table

$\mathcal{O}(h^2)$	$\mathcal{O}(h^4)$	$\mathcal{O}(h^6)$	$\mathcal{O}(h^8)$	$\mathcal{O}(h^{10})$
3.0000000000				
3.1000000000	3.1333333333			
3.1311764706	3.1415686275	3.1421176471		
3.1389884945	3.1415925025	3.1415940941	3.1415857838	
3.1409416120	3.1415926512	3.1415926611	3.1415926384	3.1415926653

My solution approach was to copy and paste Louis Saumier's Math4220_Lesson_7_Code for implementing Romberg integration, and make minimal changes to suit my example. The first step in coding this question is initializing the first term in the table. This value is obtained by deploying an instance of definite integral approximation using the trapezoidal rule. For my code, I was able to obtain this approximation from a function I created called `numericalIntegration()`. Next an iterative process is defined which populates the rest of the table based on previous elements. The algorithm deployed her is outlined in detail on page 471 of our textbook.

The first change that needed to be made from Louis' in-class example was that not enough digits were being displayed in the output. This means the accuracy could not be observed by printing the result from matlab. Changing the format to `long` at the top of the code was the solution. According to MathWorks when you set the format to `long` the display will print "with 15 digits after the decimal point for double values." Recall that our goal to print an answer that is accurate to a precision of $t = 8$. From the definition of floating point representation of a number, we know that we want our approximation to match the first 8 digits, in the case of π, we want to see the first digit, 3, and then 7 more identical digits after the decimal place. When using the positive integer s to be equal to 4 in the code, our result was not accurate enough. This was determined simply by observing the values printed in the table and counting how many digits were similar to the correct representation of π (at least more accurate). Next we tried incrementing s by one to achieve a more accurate approximation. This led to an acceptable answer. Observe in Table 1, that the entry in the bottom right of the table has 8 digits of accuracy. Notice that the integer s is representative of the number of rows and columns of the Romberg table. The relationship between this number and the order of accuracy is, $\mathcal{O}(h^{2s})$. Hence for our example, since $s = 5$, then $\mathcal{O}(h^{2s}) \implies \mathcal{O}(h^{10})$, which is an order of accuracy of 10.