```
% display code
type('rectrap.m');
type('simpsons.m');
type('statError.m');

f = @statError;
a = 0;
b = 3;
n = 200;
err = 10^-5;
[S_t, error, evals] = rectrap(f, a, b, n, err);




fprintf(' Trapezoidal Rule\n');
fprintf('The value of the integral is %f\n', S_t);
fprintf('The number of function evaluations is %d\n', evals);
fprintf('The error is %f\n', error);


delta = 10^-5;
level_max = 50;
[numI, evals, error] = simpsons(f, a, b, delta, 0, level_max, 0);

fprintf(' Simpson''s Rule\n');
fprintf('The value of the integral is %f\n', numI);
fprintf('The number of function evaluations is %d\n', evals);
fprintf('The error is %f\n', error);



% quadrature
n = (evals + 1) / 2;
n = floor(n);

result = compguassquad(f, a, b, n);

fprintf(' Composite Gauss Quadrature\n');
fprintf('The value of the integral is %f\n', result);
fprintf('The number of function evaluations is %d\n', n);


function [S_T, error, evals] = rectrap(f, a, b, n, er)
  % RECTRAP Recursive trapezoid function with caching

  h_T = b - a;
  S_T = 0.5 * h_T * (f(b) - f(a));


  cache = containers.Map({'key'}, {0});
  remove(cache, 'key');
  cache(num2str(a)) = f(a);
```

```
  cache(num2str(b)) = f(b);
  evaluations = 2;

  for i = 1:n
      h_i = h_T / 2;
      X = a + h_i : h_T : b - h_i;


      for j = 1:length(X)
          x_str = num2str(X(j));
          if ~isKey(cache, x_str)
              cache(x_str) = f(X(j));
              evaluations = evaluations + 1;
          end
          X(j) = cache(x_str);
      end

      S_i = 0.5 * S_T + h_i * sum(X);
      error = (S_i - S_T) / S_i;
      S_T = S_i;
      h_T = h_i;
      fprintf('S_%d = %f, error = %f\n', i, S_T, error);
      if abs(error) < er
          break
      end
  end

  evals = evaluations;

  end

% function [S_T, error, evals] = rectrap(f, a, b, n, er)
%  %RECTRAP Recursive trapezoid function
% %


% h_T = b - a;
% S_T = 0.5 * h_T * (f(b) - f(a)) ;



% evaluations = 2;

% for i = 1:n
%     h_i = h_T / 2 ;
%     X = a + h_i : h_T : b - h_i;
%     X = arrayfun(f, X);
%     evaluations = evaluations + length(X);
%     S_i = 0.5 * S_T + h_i * sum(X);
%     error = (S_i - S_T) / S_i;
%     S_T = S_i;
%     h_T = h_i;
%     fprintf('S_%d = %f, error = %f\n', i, S_T, error);
%     if abs(error) < er
```

```
%         break
%     end
% end

% evals = evaluations;

% end


function [numI, eval_num, err] = simpsons(f, a, b, delta, level, level_max,
eval_num, cache)
  % SIMPSONS Adaptive Simpson's rule with caching

  if nargin < 8
      cache = containers.Map({'key'}, {0});
      remove(cache, 'key');
  end

  h = b - a;
  c = (a + b) / 2;
  points = [a, c, b];

  for point = points
      point_str = num2str(point);
      if ~isKey(cache, point_str)
          cache(point_str) = f(point);
          eval_num = eval_num + 1;
      end
  end

  I_1 = h * (cache(num2str(a)) + 4 * cache(num2str(c)) + cache(num2str(b))) /
6;
  level = level + 1;
  d = (a + c) / 2;
  e = (c + b) / 2;
  new_points = [d, e];

  for point = new_points
      point_str = num2str(point);
      if ~isKey(cache, point_str)
          cache(point_str) = f(point);
          eval_num = eval_num + 1;
      end
  end

  I_2 = h * (cache(num2str(a)) + 4 * cache(num2str(d)) + 2 *
cache(num2str(c)) + 4 * cache(num2str(e)) + cache(num2str(b))) / 12;

  err = abs(I_2 - I_1) / 15;

  if level >= level_max || err <= 15 * delta
      numI = I_2 + (I_2 - I_1) / 15;
  else
      [numI_1, new_eval_num_1, err_1] = simpsons(f, a, c, delta / 2, level,
```

```
level_max, eval_num, cache);
      [numI_2, new_eval_num_2, err_2] = simpsons(f, c, b, delta / 2, level,
level_max, new_eval_num_1, cache);
      numI = numI_1 + numI_2;
      eval_num = new_eval_num_2;
      err = max(err_1, err_2); % Take the maximum of the errors
  end

  end

% function [numI, eval_num, err] = simpsons(f, a, b, delta, level, level_max,
eval_num)
% %SIMPOSONS % adaptive simpsons rule
% %


% h = b - a;
% c = (a + b) / 2;
% I_1 = h * (f(a) + 4 * f(c) + f(b)) / 6;
% level = level + 1;
% d = (a + c) / 2;
% e = (c + b) / 2;
% I_2 = h * (f(a) + 4 * f(d) + 2 * f(c) + 4 * f(e) + f(b)) / 12;

% % count the number of function evaluations
% eval_num = eval_num + 3 + 5;

% err = abs(I_2 - I_1) / 15;

% if level >= level_max
%     numI = I_2;
% else
%     if err <= 15 * delta
%         numI = I_2 + (I_2 - I_1) / 15;
%     else
%         [numI_1, new_eval_num_1, err] = simpsons(f, a, c, delta / 2, level,
level_max, eval_num);
%         [numI_2, new_eval_num_2, err] = simpsons(f, c, b, delta / 2, level,
level_max, eval_num);
%         numI = numI_1 + numI_2;
%         eval_num = new_eval_num_1 + new_eval_num_2;
%     end
% end



% % fprintf('level: %d, eval_num: %d\n', level, eval_num);
% % fprintf('I_1: %f, I_2: %f, numI: %f\n', I_1, I_2, numI);
% end




function [result] = statError(x)
```

```
%STATERROR
%   this is the fn, for the error funciont
% 2/PI * (integral from 0 to t) of e^(-x^2) dt
% we want the non integral part, the integral part is handled seperately

persistent constant
if isempty(constant)
    constant = 2/sqrt(pi);
end

result = constant * exp(-x^2);

end
```

```
S_1 = -0.667785, error = -1.534290
S_2 = 0.153663, error = 5.345779
S_3 = 0.576827, error = 0.733607
S_4 = 0.788404, error = 0.268362
S_5 = 0.894192, error = 0.118305
S_6 = 0.947085, error = 0.055848
S_7 = 0.973531, error = 0.027166
S_8 = 0.986755, error = 0.013401
S_9 = 0.993366, error = 0.006656
S_10 = 0.996672, error = 0.003317
S_11 = 0.998325, error = 0.001656
S_12 = 0.999151, error = 0.000827
S_13 = 0.999565, error = 0.000413
S_14 = 0.999771, error = 0.000207
S_15 = 0.999875, error = 0.000103
S_16 = 0.999926, error = 0.000052
S_17 = 0.999952, error = 0.000026
S_18 = 0.999965, error = 0.000013
S_19 = 0.999971, error = 0.000006
 Trapezoidal Rule
The value of the integral is 0.999971
The number of function evaluations is 127478
The error is 0.000006
 Simpson's Rule
The value of the integral is 0.999978
The number of function evaluations is 17
The error is 0.000036
 Composite Gauss Quadrature
The value of the integral is 0.999978
The number of function evaluations is 9
```

Published with MATLAB® R2023b