Computer Assignment 4 – Solutions

1. Matlab function `fastft.m`:

```
function y = fastft(x)
n = length(x);      % Determine the length of x
w = exp(-2i*pi/n);  % Calculate the primitive n-th root of 1
if n == 1
  y = x;
else
  nhalf = round(n/2);  % Note: round() is not needed if n is even,
                       % but it allows us to apply the following check:
  if 2*nhalf ~= n
    error('Length of input vector must be a power of two');
  end
  p = x(1:2:end);  % These are 'even' elements of x
  s = x(2:2:end);  % These are 'odd' elements of x
  q = fastft(p);   % Recursive use of fastft
  t = fastft(s);
  y = zeros(n,1);  % Allocate memory for the output vector y
  for k = 0:n-1    % Combine the even and odd parts
    y(k+1) = q(mod(k,nhalf)+1) + w.^k.*t(mod(k,nhalf)+1);
  end
% The above loop can be replaced by Matlab vector indexing:
%    y = q([1:nhalf 1:nhalf]) + w.^(0:n-1).*t([1:nhalf 1:nhalf]);
end
```

Test the function by computing the DFT of a given vector:

```
>> x = [0 3 3 3 1 -2 1 -3]';
>> fastft(x)
ans =
   6.0000 + 0.0000i
  -1.7071 - 9.7782i
  -3.0000 - 1.0000i
  -0.2929 - 5.7782i
   4.0000 - 0.0000i
  -0.2929 + 5.7782i
  -3.0000 + 1.0000i
  -1.7071 + 9.7782i
```

Compare the result to that obtained with the built-in Matlab function `fft`:

```
>> fastft(x)-fft(x)
ans =
   1.0e-14 *
   0.0000 + 0.0000i
   0.1332 + 0.0000i
   0.0000 - 0.0444i
   0.2331 + 0.1776i
   0.0000 - 0.0444i
  -0.4552 + 0.0000i
  -0.0888 - 0.0444i
  -0.7327 - 0.3553i
```

2. a) Matlab function `simpson.m`

```
function I = simpson(f, a, b, n)
% Integrate f(x) from a to b using composite Simpson rule with n intervals
h = (b-a)/n;
I = h/6*(f(a) + f(b) + 4*sum(f(a+(0.5:n)*h)) + 2*sum(f(a+(1:n-1)*h)));
```

Test:

```
>> format long
>> S = simpson(@(x)1./(1-sin(x)),0,pi/3,16)
S =
   2.732058440959821
```

b) Matlab function `gaussquad.m`

```
function G = gaussquad(f, a, b, k)
% Integrate f(x) from a to b using Gaussian quadrature rule with k nodes.
% Note: k = 1, 2, or 3.

% Define nodes and weights for different number of nodes
switch k
  case 1
    xi=0;  wi=2;
  case 2
    xi = [-1; 1]/sqrt(3);  wi=[1; 1];
  case 3
    xi=[-1; 0; 1]*sqrt(3/5);  wi=[5; 8; 5]/9;
  otherwise
    error('Number of nodes must be 1, 2, or 3')
end
% Rescale the values of nodes and weights for use in the interval [a, b]
xi = xi*(b-a)/2 + (a+b)/2;  wi = wi*(b-a)/2;
G = 0;
for i=1:k
    G = G + wi(i).*f(xi(i));
end
% The above loop can be replaced with the sum (provided function f can take
% a vector argument):
%   G = sum(wi.*f(xi));
```

Test the function:

```
>> G = gaussquad(@(x)1./(1-sin(x)),0,pi/3,1)
G =
   2.094395102393195
>> G = gaussquad(@(x)1./(1-sin(x)),0,pi/3,2)
G =
   2.647865728063311
>> G = gaussquad(@(x)1./(1-sin(x)),0,pi/3,3)
G =
   2.723239447875524
```

c) Matlab function `compgaussquad.m`:

```
function G = compgaussquad(f, a, b, k, n)
% Integrate f(x) from a to b using composite Gaussian quadrature rule
% with k nodes and n intervals
h = (b-a)/n;
G = 0;
for i = 1:n
  G = G + gaussquad(f, a+(i-1)*h, a+i*h, k);
end
```

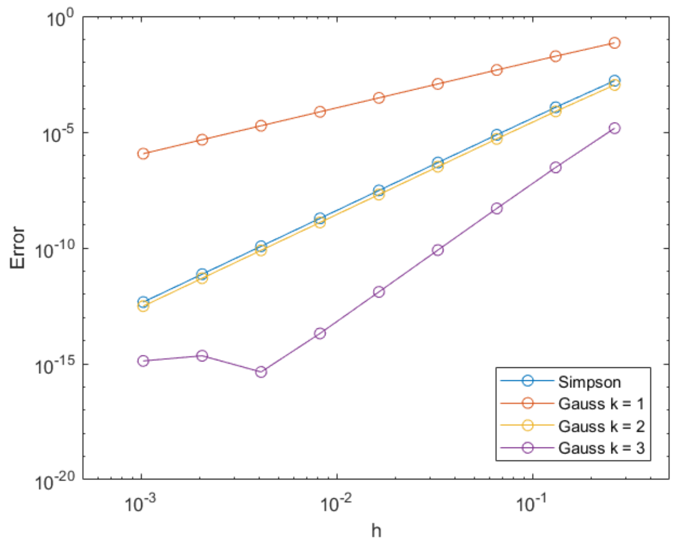
Test the function:

```
>> G = compgaussquad(@(x)1./(1-sin(x)),0,pi/3,3,16)
G =
    2.732050802516621
```

d) Matlab script to compute integration errors and generate the plot:

```
  f = @(x)1./(1-sin(x)); a = 0; b = pi/3;  % Define function and interval
  Iexact = 1 + sqrt(3);  % Exact result
  m = 2:10;
  % Allocate memory for outputs
  S = zeros(length(m),1); G1 = S; G2 = S; G3 = S; h = S;
  for i = 1:length(m)
    n = 2^m(i);  h(i) = (b-a)/n;
    S(i) = simpson(f, a, b, n);
    G1(i) = compgaussquad(f, a, b, 1, n);
    G2(i) = compgaussquad(f, a, b, 2, n);
    G3(i) = compgaussquad(f, a, b, 3, n);
  end
```

```
% Compute errors
ES = abs(S-Iexact);   EG1 = abs(G1-Iexact);
EG2 = abs(G2-Iexact); EG3 = abs(G3-Iexact);
loglog(h, ES,'o-'); hold all;
loglog(h, EG1,'o-'); loglog(h, EG2,'o-'); loglog(h, EG3,'o-');
legend({'Simpson','Gauss k = 1','Gauss k = 2','Gauss k = 3'},...
    'location','southeast');
xlim([5e-4 5e-1]);  xlabel('h'); ylabel('Error');
```



We observe linear dependence between $\log(h)$ and $\log(Error)$, which is consistent with the expectation that $Error = Ch^p$, i.e. $\log(Error) = \log(C) + p\log(h)$, so p is the slope of the straight lines. The deviation of the 3-node Gaussian quadrature from the straight line at small h is due to the round-off error.

One possible way to determine the slopes of the lines in the above plot is from the slopes of lines between successive points: $p \approx \frac{\log(Error_{i+1})-\log(Error_i)}{\log(h_{i+1})-\log(h_i)}$.

```
>> p = (log(ES(2:end))-log(ES(1:end-1)))./(log(h(2:end))-log(h(1:end-1)))
p =
    3.8089
    3.9436
    3.9851
    3.9962
    3.9991
    3.9998
    4.0002
    4.0036
>> p = (log(EG1(2:end))-log(EG1(1:end-1)))./(log(h(2:end))-log(h(1:end-1)))
p =
    1.9162
    1.9766
    1.9940
    1.9985
    1.9996
    1.9999
    2.0000
    2.0000
>> p = (log(EG2(2:end))-log(EG2(1:end-1)))./(log(h(2:end))-log(h(1:end-1)))
p =
    3.7955
    3.9398
    3.9841
    3.9960
    3.9990
    3.9997
    3.9994
    4.0016
```

```
>> p = (log(EG3(2:end))-log(EG3(1:end-1)))./(log(h(2:end))-log(h(1:end-1)))
p =
    5.6244
    5.8836
    5.9686
    5.9901
    5.9572
    5.5236
   -2.3219
    0.7370
```

We observe that $p \approx 4$ for the Simpson rule, while $p \approx 2, 4,$ and 6 for the Gaussian quadrature with $k = 1$, 2, and 3 nodes, respectively. Note that the Simpson and 2-node Gaussian quadrature have the same convergence rate, but the 2-node Gaussian quadrature is more accurate.

If the quadrature rule is exact for polynomials of degree n, then the integration error of the composite quadrature rule with the interval size h is $O(h^{n+1})$. For the Simpson rule $n = 3$ so the error scales as $O(h^4)$. For the k-node Gaussian rule $n = 2k - 1$, so the error scales as $O(h^{2k})$. Therefore, the above observations are consistent with the theory.

3. a) The analytical solution (i.e, the flow map) of the IVP $du/dt = u(1 - u^2)$, $u(0) = u_0$ is

$$u(t) = \Phi_t(u_0) = \frac{e^t}{\sqrt{e^{2t} - 1 + u_0^{-2}}}$$

Matlab function `flowmap.m` implements the evaluation of this flow map:

```
function u = flowmap(t,u0)
%  Computes the flow map of the IVP du/dt = u*(1-u^2), u(0) = u0.
%  Note: If t is a vector, the function returns a vector u(t).
u = exp(t)./sqrt(exp(2*t)-1+u0^-2);
```

b) The local and global errors of the numerical method Ψ_h for solving the initial value problem (IVP)

$$du/dt = f(u), \; u(0) = u_0,$$

are defined as follows:

$$\mathrm{le}(h,u) = \Phi_h(u) - \Psi_h(u)$$
$$\mathrm{ge}_n(h,u) = \Phi_{nh}(u) - \Psi_{nh}(u)$$

So, the local and global errors along the numerical solution $u_{n+1} = \Psi_h(u_n)$ are given by

$$\mathrm{le}(h,u_n) = \Phi_h(u_n) - u_{n+1}$$
$$\mathrm{ge}_n(h,u_0) = \Phi_{nh}(u_0) - u_n$$

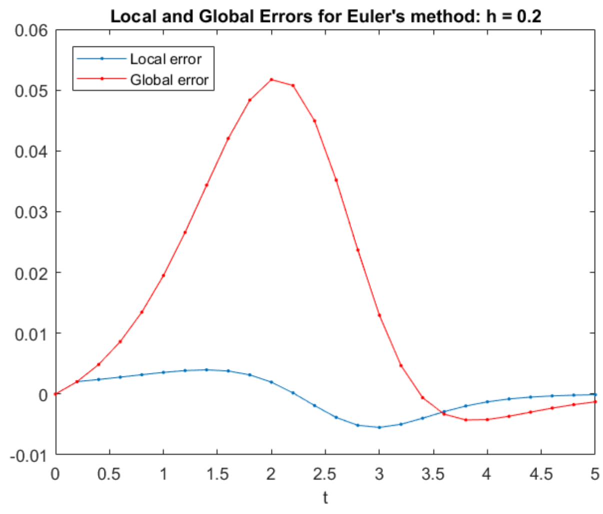
$n = 0, 1, 2, \dots$.

The following function calculates the local and global errors of Euler's method:

```
function [t, u, le, ge] = errors_euler(u0, T, h)
%  Compute local and global error for the solution of the IVP
%  du/dt = u*(1-u^2), u(0) = u0,
%  using Euler's method with step size h up to time T.
f = @(u) u.*(1-u.^2);
N = round(T/h);
t = zeros(N+1,1); u = t; le = t; % Initialise variables
u(1) = u0;  % Initial condition
for n = 1:N
  t(n+1) = t(n) + h;
  u(n+1) = u(n) + h * f(u(n));  % Euler's method
  le(n+1) = flowmap(h,u(n)) - u(n+1);
end
ge = flowmap(t,u0) - u;
```

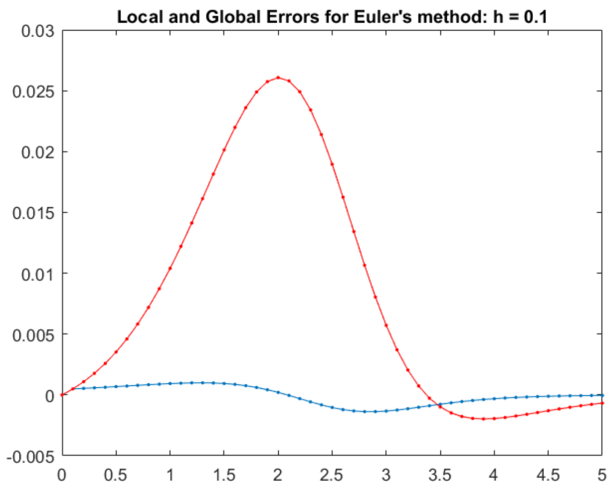
Here is how we use it to produce the plot of local and global errors along the numerical solution with step size h = 0.2:

```
>> u0 = 0.1;  T = 5.0;  h = 0.2;
>> [t1, u1, le1, ge1] = errors_euler(u0, T, h);
>> plot(t1,le1,'.-',t1,ge1,'r.-'); xlim([0 T]);
>> title('Local and Global Errors for Euler''s method: h = 0.2');
>> xlabel('t'); legend('Local error','Global error','location','northwest');
```



The calculation for h = 0.1:

```
>> u0 = 0.1;  T = 5.0;  h = 0.1;
>> [t2, u2, le2, ge2] = errors_euler(u0, T, h);
>> plot(t2,le2,'.-',t2,ge2,'r.-'); xlim([0 T]);
>> title('Local and Global Errors for Euler''s method: h = 0.1');
>> xlabel('t'); legend('Local error','Global error','location','northwest');
```



According to numerical analysis, the local error of Euler's method should be proportional to h^2, while the global error should be proportional to h. Whether this is the case is not clear from the above plots, but we can check, for example, the ratio of maximum local and global errors for the two methods:

```
>> max(abs(le1))/max(abs(le2))
ans =
    4.0005
>> max(abs(ge1))/max(abs(ge2))
ans =
    1.9840
```

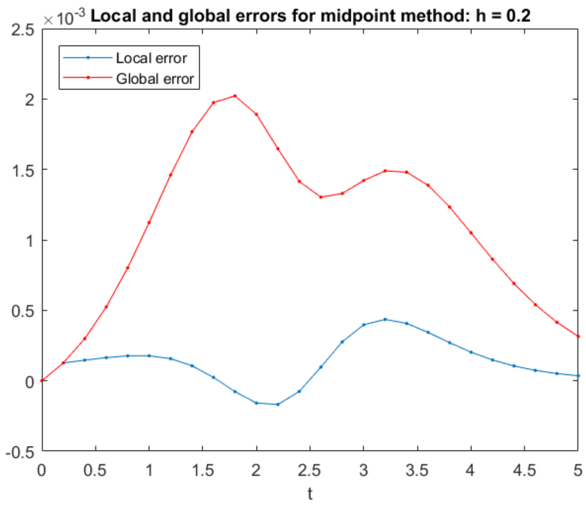
These results are consistent with our expectation: the maximum local error with h = 0.2 is four times larger than with h = 0.1, and the maximum global error is approximately two times larger.

c) The following function calculates the local and global errors of the explicit midpoint method:

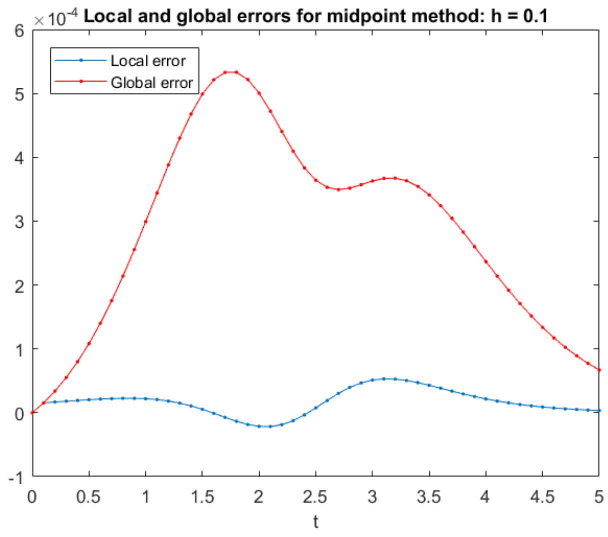
```
function [t, u, le, ge] = errors_midpoint(u0, T, h)
%  Compute local and global error for the solution of the IVP
%  du/dt = u*(1-u^2), u(0) = u0,
%  using explicit midpoint method with step size h up to time T.
f = @(u) u.*(1-u.^2);
N = round(T/h);
t = zeros(N+1,1); u = t; le = t; % Initialise variables
u(1) = u0;  % Initial condition
for n = 1:N
  t(n+1) = t(n) + h;
  ut = u(n) + (h/2) * f(u(n));   % compute the 'mid-point'
  u(n+1) = u(n) + h * f(ut);     % Explicit midpoint rule
  le(n+1) = flowmap(h,u(n)) - u(n+1);
end
ge = flowmap(t,u0) - u;
```

Here are the results for the explicit midpoint method:

```
>> u0 = 0.1;  T = 5.0;  h = 0.2;
>> [t1, u1, le1, ge1] = errors_midpoint(u0, T, h);
>> plot(t1,le1,'.-',t1,ge1,'r.-'); xlim([0 T]);
>> title('Local and global errors for midpoint method: h = 0.2');
>> xlabel('t'); legend('Local error','Global error','location','northwest');
```



```
>> u0 = 0.1;  T = 5.0;  h = 0.1;
>> [t2, u2, le2, ge2] = errors_midpoint(u0, T, h);
>> plot(t2,le2,'.-',t2,ge2,'r.-'); xlim([0 T]);
>> title('Local and global errors for midpoint method: h = 0.1');
>> xlabel('t'); legend('Local error','Global error','location','northwest');
```



We see that explicit midpoint method is much more accurate than Euler's method, but it is again difficult to determine the scaling of errors from the above figures. So, we again calculate the ratio of maximum local and global errors:

```
>> max(abs(le1))/max(abs(le2))
ans =
    8.2040
>> max(abs(ge1))/max(abs(ge2))
ans =
    3.7925
```

According to numerical analysis, explicit midpoint method is 2nd order, so the local error should scale as h^3 and the global error as h^2. So, the ratios which we calculate above should be 8 for the local error and 4 for the global error. The above results are consistent with this analysis.