Math/CS 466/666: Programming Project 2

1. Consider the initial-value problem

$$y' = \frac{2}{t}y + t^2e^t \qquad \text{where} \qquad y(1) = 0$$

with exact solution $y(t) = t^2(e^t - e)$.

(i) Use Euler's method with $h = 0.1$ to approximate the solution and compare it with the actual values of y.

Recall the Euler's method is given by

$$\begin{cases} w_0 = 0 \\ t_n = t_0 + hn \\ w_{n+1} = w_n + hf(t_n, w_n) \end{cases} \qquad \text{where} \qquad f(t,y) = \frac{2}{t}y + t^2e^t.$$

My program was

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

double h=0.1;
double y(double t){
    return t*t*(exp(t)-exp(1));
}
double f(double t,double y){
    return 2*y/t+t*t*exp(t);
}
double euler(double t,double y){
    return y+h*f(t,y);
}

int main(){
    double wn=0,t0=1;
    printf("#%21s %21s %21s\n","tn","wn","|wn-y(tn)|");
    for(int n=0;;n++){
        double tn=t0+n*h;
        printf(" %21.14e %21.14e %21.14e\n",tn,wn,fabs(wn-y(tn)));
        if(n>=10) break;
        wn=euler(tn,wn);
    }
    return 0;
}
```

which generated the output

```
#                   tn                    wn            |wn-y(tn)|
  1.00000000000000e+00  0.00000000000000e+00  0.00000000000000e+00
  1.10000000000000e+00  2.71828182845905e-01  7.40916936938353e-02
  1.20000000000000e+00  6.84755577715406e-01  1.81886958044197e-01
  1.30000000000000e+00  1.27697834420870e+00  3.30236733972034e-01
  1.40000000000000e+00  2.09354768783769e+00  5.26811863398139e-01
  1.50000000000000e+00  3.18744512245892e+00  7.80221171768872e-01
  1.60000000000000e+00  4.62081784627951e+00  1.10014367931683e+00
  1.70000000000000e+00  6.46639637770960e+00  1.49747710013537e+00
  1.80000000000000e+00  8.80911968894342e+00  1.98450497154722e+00
  1.90000000000000e+00  1.17479965439625e+01  2.57508499192853e+00
  2.00000000000000e+00  1.53982356527792e+01  3.28486142910718e+00
```

Note the comparison of the errors given in the last column indicates by $t = 2$ the error is more than 17 percent. In particular, only one significant digit is correct.

(ii) Use the answers generated in part (i) and linear interpolation to approximate the values of $y(t)$ for $t \in \{1.04, 1.55, 1.97\}$ and compare that approximation to the actual values.

We use the Lagrange interpolating polynomial given as

$$\ell(t) = \frac{w_{n+1}(t - t_n) - w_n(t - t_{n+1})}{t_{n+1} - t_n} \qquad \text{for} \qquad t \in [t_n, t_n + 1]$$

and compute $\ell(t)$ for $t \in \{1.04, 1.55, 1.97\}$ using the program

```
1  #include <stdio.h>
2  #include <math.h>
3
4  double h=0.1;
5  double y(double t){
6      return t*t*(exp(t)-exp(1));
7  }
8  double f(double t,double y){
9      return 2*y/t+t*t*exp(t);
10 }
11 double euler(double t,double y){
12     return y+h*f(t,y);
13 }
14
15 double ts[] = { 1.04, 1.55, 1.97 };
16 const int tlen = sizeof(ts)/sizeof(double);
17 int main(){
18     double wn=0,t0=1;
19     printf("#%21s %21s %21s\n","tn","linear(tn)","|linear(tn)-y(tn)|");
20     for(int n=0;n<=10;n++){
```

```
21          double tn=t0+n*h;
22          double wnp1=euler(tn,wn),tnp1=t0+(n+1)*h;
23          for(int i=0;i<tlen;i++) if(tn<ts[i] && ts[i]<=tnp1){
24              double z=(wnp1*(ts[i]-tn)-wn*(ts[i]-tnp1))/(tnp1-tn);
25              printf(" %21.14e %21.14e %21.14e\n",ts[i],z,fabs(z-y(ts[i])));
26          }
27          wn=wnp1;
28      }
29      return 0;
30  }
```

which generated the output

```
#                    tn             linear(tn)    |linear(tn)-y(tn)|
  1.04000000000000e+00  1.08731273138362e-01  1.12562239229821e-02
  1.55000000000000e+00  3.90413148436922e+00  8.84503536432187e-01
  1.97000000000000e+00  1.43031639201342e+01  2.97613451542345e+00
```

(iii) Compute the value of h necessary of $\big|y(t_i) - w_i\big| \leq 0.1$ using the error estimate

$$\big|y(t_i) - w_i\big| \leq \frac{hM}{2L}\Big[e^{L(t_i-a)} - 1\Big].$$

Here $a = 1$ and $t_i \in [1, 2]$. Further recall that M is a bound such that $|y''(t)| \leq M$ for all $t \in [1, 2]$ and L is a Lipschitz constant such that

$$\big|f(t, y) - f(t, w)\big| \leq L|y - w| \qquad \text{for} \qquad i = 0, 1, \ldots, N$$

where y and w are contained in a some suitable neighborhood of $y(t)$ which is large enough to contain the approximation given by $w_i \approx y(t_i)$.

Now, since $y = t^2(e^t - e)$ it follows that

$$\begin{aligned} y' &= 2t(e^t - e) + t^2e^t = (t^2 + 2t)e^t - 2te \\ y'' &= (2t + 2)e^t + (t^2 + 2t)e^t - 2e = (t^2 + 4t + 2)e^t - 2e \\ y''' &= (2t + 4)e^t + (t^2 + 4t + 2)e^t = (t^2 + 6t + 6)e^t. \end{aligned}$$

The maximum of $|y''(t)|$ on $[1, 2]$ occurs either at the endpoints of the interval or at a critical point where $y'''(t) = 0$. Solving for the critical points yields

$$t^2 + 6t + 6 = 0 \qquad \text{or equivalently} \qquad t = \frac{-6 \pm \sqrt{6^2 - 4 \cdot 6}}{2} = -3 \pm \sqrt{3}.$$

As both values of t are outside $[1, 2]$, then only the endpoints need to be studied. Since

$$\begin{aligned} y''(1) &= (1 + 4 + 2)e - 2e = 5e \approx 13.5914 \\ y''(2) &= (4 + 8 + 2)e^2 - 2e = (14e - 2)e \approx 98.0102, \end{aligned}$$

taking $M = 98.0103$ implies $|y''(t)| \leq M$ for all $t \in [1, 2]$.

To find L we take partial derivatives of f with respect to the second variable to obtain

$$f_y(t, y) = \frac{2}{t} \qquad \text{so that} \qquad \big|f_y(t, y)\big| \leq 2 \quad \text{for} \quad t \in [1, 2].$$

Note that we are lucky in this case that $f_y(t, y)$ does not, in fact, depend on y, so there is no need to choose or make use of a suitable neighborhood of $y(t_i)$ in the estimates.

Substituting for M, L and a and then maximizing over $t \in [0, 1]$ we obtain that it is sufficient for

$$\frac{h \cdot 98.0103}{2 \cdot 2}\Big[e^{2(2-1)} - 1\Big] \leq 0.1$$

or equivalently that

$$h \leq \Big(\frac{4}{98.0103}\Big)\Big(\frac{0.1}{e^2 - 1}\Big) \approx 0.000638780$$

to ensure that $\big|y(t_i) - w_i\big| \leq 0.1$ for all i.

2. Again consider the initial-value problem

$$y' = \frac{2}{t}y + t^2 e^t \qquad \text{where} \qquad y(1) = 0$$

with exact solution $y(t) = t^2(e^t - e)$.

(i) Use Taylor's method of order two with $h = 0.1$ to approximate the solution, and compare it with the actual values of y.

The Taylor's method of order n in general requires differentiating the function $f(t, y)$ to obtain higher order derivatives of y. For the second order method we shall need

$$\frac{d^n}{dt^n}y(t) = \frac{d^{n-1}}{dt^{n-1}}f\big(t, y(t)\big) \qquad \text{for} \qquad n = 1, 2.$$

In anticipation of constructing Taylor's method of order four later, a Maple script was constructed that takes $f(t, y)$ as input and then uses symbolic differentiation and the built-in code generation facilities to form a jet of C functions contained in the file `jet.i` which compute the needed derivatives. The Maple script `jet.mpl` is

```
1 # Run this script with maple -q jet.mpl to create output file jet.i
2 restart;
3 kernelopts(printbytes=false);
4 interface(warnlevel=0);
5 # Change the following function for different Taylor Methods
6 jet1:=2*y(t)/t+t^2*exp(t);
7 n:=4;
8 for i from 2 to n do
9     t1:=diff(jet||(i-1),t);
```

```
    t2:=subs(diff(y(t),t)=jet1,t1);
    jet||i:=simplify(t2);
end do;
with(CodeGeneration);
writeto("jet.i");
for i from 1 to n do
    printf("/*\n");
    d||i||y:=unapply(subs(y(t)=y,jet||i),(t,y));
    printf("*/\n");
    C(d||i||y,deducetypes=false,defaulttype=float,optimize=true);
end do;
writeto(terminal);
```

When run with the command `maple -q jet.mpl` this script produces the output file `jet.i` that contains

```
/*
                                        2 y    2
                      d1y := (t, y) -> --- + t  exp(t)
                                         t


*/
#include <math.h>

double d1y (double t, double y)
{
  double t6;
  double t7;
  t6 = t * t;
  t7 = exp(t);
  return(0.2e1 * y / t + t6 * t7);
}
/*
                                          3           4
                                  2 y + 4 t  exp(t) + t  exp(t)
                d2y := (t, y) -> -----------------------------
                                               2
                                              t


*/
#include <math.h>

double d2y (double t, double y)
{
  double t4;
```

```
  double t6;
  double t9;
  t4 = t * t;
  t6 = exp(t);
  t9 = t4 * t4;
  return((0.2e1 * y + 0.4e1 * t4 * t * t6 + t6 * t9) / t4);
}
/*
                                                          2
                    d3y := (t, y) -> exp(t) (6 + 6 t + t )

*/
#include <math.h>

double d3y (double t, double y)
{
  double t3;
  double t5;
  t3 = exp(t);
  t5 = t * t;
  return(t3 * (0.6e1 + 0.6e1 * t + t5));
}
/*
                                                           2
                    d4y := (t, y) -> exp(t) (12 + 8 t + t )

*/
#include <math.h>

double d4y (double t, double y)
{
  double t3;
  double t5;
  t3 = exp(t);
  t5 = t * t;
  return(t3 * (0.12e2 + 0.8e1 * t + t5));
}
```

It is now possible to modify the code which implements the Euler method for the previous problem to use Taylor's second order method. The resulting C code is

```
#include <stdio.h>
#include <math.h>
#include "jet.i"

```

```
5  double h=0.1;
6  double y(double t){
7      return t*t*(exp(t)-exp(1));
8  }
9  double taylor2(double t,double y){
10     return y+h*(d1y(t,y)+h*d2y(t,y)/2);
11 }
12
13 int main(){
14     double wn=0,t0=1;
15     printf("#%21s %21s %21s\n","tn","wn","|wn-y(tn)|");
16     for(int n=0;;n++){
17         double tn=t0+n*h;
18         printf(" %21.14e %21.14e %21.14e\n",tn,wn,fabs(wn-y(tn)));
19         if(n>=10) break;
20         wn=taylor2(tn,wn);
21     }
22     return 0;
23 }
```

Note that the file `jet.i` is included at the top of the file and the `taylor2` subroutine calls the functions `d1y` and `d2y` from that file. Rather than including `jet.i` the functions `d1y` and `d2y` could have been compiled separately into an object file linked at build time. Aside from simplicity, the advantage of including `jet.i` over linking is that this allows an optimizing compiler to inline calls of the functions `d1y` and `d2y` from the `taylor2` subroutine. Inlining allows further data-flow analysis to perform common subexpression elimination between the two functions so that, for example, only one library call to exponential function is made for each time step.

Running the resulting executable produces the following output.

```
#                    tn                    wn            |wn-y(tn)|
  1.00000000000000e+00  0.00000000000000e+00  0.00000000000000e+00
  1.10000000000000e+00  3.39785228557381e-01  6.13464798235924e-03
  1.20000000000000e+00  8.52143449276347e-01  1.44990864832564e-02
  1.30000000000000e+00  1.58176950519471e+00  2.54455729860259e-02
  1.40000000000000e+00  2.58099664973816e+00  3.93629014976722e-02
  1.50000000000000e+00  3.91098455934566e+00  5.66817348821296e-02
  1.60000000000000e+00  5.64308103583302e+00  7.78804897633192e-02
  1.70000000000000e+00  7.86038160386642e+00  1.03491873978549e-01
  1.80000000000000e+00  1.06595144803927e+01  1.34110180097901e-01
  1.90000000000000e+00  1.41526820903769e+01  1.70399445514105e-01
  2.00000000000000e+00  1.84699944825563e+01  2.13102599330085e-01
```

The comparison of errors indicates Taylor's second order method is more accurate than Euler's method. While better, at $t = 2$ only two significant digits are correct.

(ii) Use the answers generated in part (i) and linear interpolation to approximate the values of $y(t)$ for $t \in \{1.04, 1.55, 1.97\}$ and compare that approximation to the actual values.

Modifications similar to those needed to solve the above problem performed for the corresponding Euler code that does linear interpolation to obtain

```
#include <stdio.h>
#include <math.h>
#include "jet.i"

double h=0.1;
double y(double t){
    return t*t*(exp(t)-exp(1));
}
double taylor2(double t,double y){
    return y+h*(d1y(t,y)+h*d2y(t,y)/2);
}

double ts[] = { 1.04, 1.55, 1.97 };
const int tlen = sizeof(ts)/sizeof(double);
int main(){
    double wn=0,t0=1;
    printf("#%21s %21s %21s\n","tn","linear(tn)","|linear(tn)-y(tn)|");
    for(int n=0;n<=10;n++){
        double tn=t0+n*h;
        double wnp1=taylor2(tn,wn),tnp1=t0+(n+1)*h;
        for(int i=0;i<tlen;i++) if(tn<ts[i] && ts[i]<=tnp1){
            double z=(wnp1*(ts[i]-tn)-wn*(ts[i]-tnp1))/(tnp1-tn);
            printf(" %21.14e %21.14e %21.14e\n",ts[i],z,fabs(z-y(ts[i])));
        }
        wn=wnp1;
    }
    return 0;
}
```

with output

```
#                   tn            linear(tn)    |linear(tn)-y(tn)|
  1.04000000000000e+00  1.35914091422952e-01  1.59265943616083e-02
  1.55000000000000e+00  4.77703279758934e+00  1.16022232120603e-02
  1.97000000000000e+00  1.71748007649025e+01  1.04497670655157e-01
```

(iii) Use Taylor's method of order four with $h = 0.1$ to approximate the solution, and compare it with the actual values of y.

The code to compute the derivatives

$$\frac{d^n}{dt^n}y(t) = \frac{d^{n-1}}{dt^{n-1}}f\big(t, y(t)\big) \qquad \text{for} \qquad n = 1, 2, 3, 4$$

has already been written and appears as `jet.i` above. We include the same file and modify second order method to contain the next two terms of the Taylor series to obtain the fourth order method. The resulting code is

```
1  #include <stdio.h>
2  #include <math.h>
3  #include "jet.i"
4
5  double h=0.1;
6  double y(double t){
7      return t*t*(exp(t)-exp(1));
8  }
9  double taylor4(double t,double y){
10     return y+h*(d1y(t,y)+h*(d2y(t,y)/2+h*(d3y(t,y)/6+h*d4y(t,y)/24)));
11 }
12
13 int main(){
14     double wn=0,t0=1;
15     printf("#%21s %21s %21s\n","tn","wn","|wn-y(tn)|");
16     for(int n=0;;n++){
17         double tn=t0+n*h;
18         printf(" %21.14e %21.14e %21.14e\n",tn,wn,fabs(wn-y(tn)));
19         if(n>=10) break;
20         wn=taylor4(tn,wn);
21     }
22     return 0;
23 }
```

with output

```
#                    tn                    wn            |wn-y(tn)|
  1.00000000000000e+00  0.00000000000000e+00  0.00000000000000e+00
  1.10000000000000e+00  3.45912688845699e-01  7.18769404112196e-06
  1.20000000000000e+00  8.66625729278685e-01  1.68064809182855e-05
  1.30000000000000e+00  1.60718588643574e+00  2.91917449999346e-05
  1.40000000000000e+00  2.62031484281613e+00  4.47084197023884e-05
  1.50000000000000e+00  3.96760253888109e+00  6.37553467068841e-05
  1.60000000000000e+00  5.72087475559039e+00  8.67700059474785e-05
  1.70000000000000e+00  7.96375924414548e+00  1.14233699487265e-04
```

```
 1.80000000000000e+00  1.07934779832196e+01  1.46677271066764e-04
 1.90000000000000e+00  1.43228968484455e+01  1.84687445532461e-04
 2.00000000000000e+00  1.86828681680090e+01  2.28913877396764e-04
```

Switching to Taylor's fourth-order method yields a substantial improvement in accuracy– the approximations for $t_i \in [1, 2]$ are all accurate to within 0.0012 percent.

(iv) Use the answers generated in part (iii) and piecewise cubic Hermite interpolation to approximate the values of $y(t)$ for $t \in \{1.04, 1.55, 1.97\}$ and compare that approximation to the actual values.

The routines from class which use Newton's divided differences to create an interpolating polynomial were modified to make Hermite polynomials as described in Chapter 3.4 of our textbook *Numerical Analysis* by Burden and Faires. The resulting code is

```
#include <stdio.h>
#include <math.h>
#include "jet.i"

double h=0.1;
double y(double t){
    return t*t*(exp(t)-exp(1));
}
double taylor4(double t,double y){
    return y+h*(d1y(t,y)+h*(d2y(t,y)/2+h*(d3y(t,y)/6+h*d4y(t,y)/24)));
}
double hpolyeval(int n,double dd[2*n][2*n+1],double t){
    double s=dd[0][2*n];
    for(int j=2*n-1;j>=1;j--){
        s=dd[0][j]+(t-dd[j-1][0])*s;
    }
    return s;
}
void hpolymake(int n,double dd[2*n][2*n+1],
    double x[n],double y[n],double dy[n]){
    for(int i=0;i<n;i++){
        int j=2*i;
        dd[j][0]=x[i]; dd[j+1][0]=x[i];
        dd[j][1]=y[i]; dd[j+1][1]=y[i];
        dd[j][2]=dy[i];
    }
    for(int i=0;i<n-1;i++){
        int j=2*i+1;
        dd[j][2]=(dd[j+1][1]-dd[j][1])
            /(dd[j+1][0]-dd[j][0]);
    }
```

```
    for(int j=2;j<2*n;j++){
        for(int i=0;i<2*n-j;i++) {
            dd[i][j+1]=(dd[i+1][j]-dd[i][j])
                     /(dd[i+j][0]-dd[i][0]);
        }
    }
}

double ts[] = { 1.04, 1.55, 1.97 };
const int tlen = sizeof(ts)/sizeof(double);
int main(){
    double wn=0,t0=1;
    printf("#%21s %21s %21s\n","tn","hermite(tn)","|hermite(tn)-y(tn)|");
    for(int n=0;n<=10;n++){
        double tn=t0+n*h;
        double wnp1=taylor4(tn,wn),tnp1=t0+(n+1)*h;
        for(int i=0;i<tlen;i++) if(tn<ts[i] && ts[i]<=tnp1){
            double dd[4][5];
            hpolymake(2,dd,(double [2]){tn,tnp1},(double [2]){wn,wnp1},
                (double [2]){d1y(tn,wn),d1y(tnp1,wnp1)});
            double z=hpolyeval(2,dd,ts[i]);
            printf(" %21.14e %21.14e %21.14e\n",ts[i],z,fabs(z-y(ts[i])));
        }
        wn=wnp1;
    }
    return 0;
}
```

with output

```
#                   tn           hermite(tn)   |hermite(tn)-y(tn)|
  1.04000000000000e+00  1.19970383518573e-01  1.71135427707858e-05
  1.55000000000000e+00  4.78852715568361e+00  1.07865117796813e-04
  1.97000000000000e+00  1.72790404208027e+01  2.58014754983549e-04
```

Note that the Hermite polynomials preserves the error level of 10^{-4} for the interpolation. To see how important the higher-order interpolation is, it is possible to use linear interpolation as before. With `taylor4` and only linear interpolation the results are

```
#                   tn            linear(tn)    |linear(tn)-y(tn)|
  1.04000000000000e+00  1.38365075538280e-01  1.83775784769356e-02
  1.55000000000000e+00  4.84423864723574e+00  5.56036264343369e-02
  1.97000000000000e+00  1.73748767721400e+01  9.55783365822960e-02
```

which is much worse than the Hermite interpolation. The linear case further confirms that the code for the Hermite polynomials is working correctly.