Problem 1 (Numerical solution of a scalar ODE). First, for each of the three methods (and the Crank-Nicolson method you weren’t asked to use, but that we have seen since in class), here are short programs that compute the final value \(x(4)\) for a series of decreasing time step lengths \(h\). An interpretation follows below. Note that in all programs, rather than storing all intermediate values \(x_k\), we use only a single variable that denotes \(x_k\), and simply stop iterating when we get to \(t = 4\); we can do this, since at least for the present problem, we are certainly not interested in all the intermediate values, only the final one.

**First-order Taylor method**

For the first order Taylor method, the iteration formula was

\[
x_{k+1} = x_k + hf(t_k, x_k).
\]

Note that this is exactly the forward Euler method, which is why the latter’s name hasn’t been listed among the methods you were asked to investigate. Here’s a little code that computes things (including an outer loop that starts at \(h = 4\) and decreases it by factors of 2 until it reaches \(h = \frac{1}{512}\)):

```cpp
#include <iostream>
#include <iomanip>

int main ()
{
    std::cout << std::setprecision(16);

    for (double h = 4; h>1./1024; h/=2)
    {
        double x = 1./std::pow(10., 1./3.);

        double t = 0;
        do
        {
            // compute f(x(k))
            const double f = 1./(3*x*x);
            // compute x(k+1)
            x = x+h*f;

            t += h;

            // output the solution to x(t) for t = 0, h,..., 4
            std::cout << t << "\t" << x << std::endl;
        } while (t < 4);
    }
}
```

Answers for homework assignment 11
\( t \leftarrow h; \)
}
while (t<4);

```cpp
std::cout << "h=" << h << " x(4)=" << x
<< ", e=" << x-std::pow(4.1, 1./3)
<< std::endl;
```
}
}

The output from this program is this:

\[
\begin{align*}
\text{h=4, x(4)=6.652943994844984, e=5.05242330961829} \\
\text{h=2, x(4)=3.611197059144108, e=2.010676395260952} \\
\text{h=1, x(4)=2.240589712716079, e=0.640069048832937} \\
\text{h=0.5, x(4)=1.774246230152993, e=0.173725566269837} \\
\text{h=0.25, x(4)=1.656076526728398, e=0.0555586284524239} \\
\text{h=0.125, x(4)=1.623320542661419, e=0.0227998787782639} \\
\text{h=0.0625, x(4)=1.611128102959851, e=0.0106743907669615} \\
\text{h=0.03125, x(4)=1.605676479869782, e=0.005155815086626436} \\
\text{h=0.015625, x(4)=1.603066629432145, e=0.002545961888205261} \\
\text{h=0.0078125, x(4)=1.601786193290267, e=0.001265529407111332} \\
\text{h=0.00390625, x(4)=1.601151625769361, e=0.000630961886205261} \\
\text{h=0.001953125, x(4)=1.60083570132901, e=0.00031503744585426} \\
\end{align*}
\]

The behavior at the beginning is somewhat erratic, but towards the end, the error is reduced very reliably by a factor of 2 every time \( h \) is divided by 2. That's the hallmark sign of linear convergence, i.e. \( x(4) - x_N = \mathcal{O}(h) \).

As to why this reliable reduction doesn't happen in the first few steps: the statement of linear convergence only says that \( x(4) - x_N = \mathcal{O}(h) \), which in turn means that \( |x(4) - x_N| \leq Ch \) for some constant \( C \) of which we don't necessarily know the size. However, if you visualize this, then it only means that if you plot the error in \( y \)-direction on a plot where \( h \) is the \( x \)-direction, then the error must be below the line described by \( Ch \): it doesn't necessarily have to follow exactly this line, it just has to stay below this line. In many cases, it will get closer and close to this line as \( h \to 0 \), but will jump around while \( h \) is large. If you think about it, this shouldn't be too surprising, since one would expect the error to change in some rather reliable way if the mesh size is changed from \( \frac{1}{1000} \) to \( \frac{1}{2000} \), but it certainly matters a great deal more if we use mesh sizes \( h = 4 \) or \( h = 2 \) when integrating our ODE from \( t = 0 \) to \( t = T = 4 \).

**Second-order Taylor method**

The iteration formula for the second order Taylor method reads

\[
x_{k+1} = x_k + hf(t_k,x_k) + \frac{h^2}{2}(f_t(t_k,x_k) + f_x(t_k,x_k)f(t_k,x_k)),
\]
where subscripts $f_t$ and $f_x$ denote partial derivatives of $f$ with respect to the indicated variable.

A program to implement this method is as follows:

```cpp
#include <iostream>
#include <iomanip>

int main ()
{
    std::cout << std::setprecision(16);

    for (double h = 4; h>1./1024; h/=2)
    {
        double x = 1./std::pow(10., 1./3.);
        double t = 0;
        do
        {
            // compute $f(x(k))$ as well as 
            // its partial derivatives
            const double f = 1./(3*x*x);
            const double f_t = 0;
            const double f_x = -2./(3*x*x*x);

            // compute $x(k+1)$
            x = x+h*f+h*h/2*(f_t+f_x*f);
            t += h;
        } while (t<4);
    std::cout << "h=", h << ", x(4)=", x
              << " , e=", x-std::pow(4.1, 1./3)
              << std::endl;
    }
}
```

Again, the output is as follows:

```
h=4, x(4)=75.86419082493778, e=-77.46471148882094
h=2, x(4)=17.06844423098321, e=-18.66896489486637
h=1, x(4)=3.041517329835609, e=-4.642037993718764
h=0.5, x(4)=76196.2707193759, e=76194.67019871202
h=0.25, x(4)=1.550175004359412, e=-0.05034565952374348
h=0.125, x(4)=1.593544329286291, e=-0.006976334596863865
h=0.0625, x(4)=1.599236871471157, e=-0.001283792411998164
h=0.03125, x(4)=1.6002495944623, e=-0.0002710694208549747
```
h=0.015625, x(4)=1.600458375475515, e=-6.228840763999877e-05
h=0.0078125, x(4)=1.600517003719129, e=-3.660164025864887e-06
h=0.00390625, x(4)=1.600519758032563, e=-9.058505927583838e-07

Here, the results vary even more during the first few time steps, but towards the end, the error is reliably reduced by a factor of 4 each time we halve $h$. That means that $x(4) - x_N = O(h^2)$, i.e. that the method converges quadratically.

**Implicit Euler method**

For the implicit Euler method, we iterate

$$x_{k+1} = x_k + hf(t_{k+1}, x_{k+1}),$$

where $t_{k+1} = t_k + h$. Here, $x_{k+1}$ appears both on the left and right hand side of the equation, and we can’t just evaluate the right hand side as before to assign it to the left hand side. Rather, we here have an implicit equation that we need to solve for $x_{k+1}$. We do so using Newton’s method; for this, let us reformulate the equation as one where we look for a root $g(x_{k+1}) = 0$ of the function

$$g(x_{k+1}) = x_{k+1} - x_k - hf(t_{k+1}, x_{k+1}).$$

Note that $x_k$ is of course already known by the time we want to compute $x_{k+1}$. We can solve for the latter using Newton’s method, where in Newton iteration $i + 1$ for time step $k + 1$ we would compute

$$(x_{k+1})_{i+1} = (x_{k+1})_i - \frac{g((x_{k+1})_i)}{g'((x_{k+1})_i)}.\]

For this, we need $g'$ (as a function of $x_{k+1}$), which is readily computed as

$$g'(x_{k+1}) = 1 - hf_x(t_{k+1}, x_{k+1}).$$

The following program implements this (the variables $x_k$ and $x_{k+1}$ are denoted by the names $xk$, $xkp1$):

```cpp
#include <iostream>
#include <iomanip>

double f (const double x) { return 1./(3*x*x); }

double f_prime (const double x) { return -2./(3*x*x*3); }
```

4
double g (const double xkp1,  
    const double xk,  
    const double h)  
{
    return xkp1 - xk - h*f(xkp1);
}

double g_prime (const double xkp1,  
    const double h)  
{
    return 1 - h*f_prime(xkp1);
}

double compute_xkp1 (const double xk,  
    const double h)  
{
    double xkp1 = xk;
    for (unsigned int it=0; it<4; ++it)
        xkp1 = xkp1 - g(xkp1, xk, h) / g_prime(xkp1, h);
    return xkp1;
}

int main ()
{
    std::cout << std::setprecision(16);

    for (double h = 4; h>1./1024; h/=2)
    {
        double x = 1./std::pow(10., 1./3.);
        double t = 0;
        do
        {
            x = compute_xkp1 (x, h);
            t += h;
        } while (t<4);
        std::cout << "h=" << h << " x(4)=" << x
    }
As seen in the function `compute` , we do only four Newton iterations. This seems few (and a more sophisticated implementation would also use as many iterations as it takes until the result does not change significantly any more), but you should note that we initialize the Newton iteration by using $x_{k+1} = x_k$ – and for small step sizes $h$, we certainly expect that our approximation $x_{k+1} \approx x(t_k + h)$ is close to $x_k \approx x(t_k)$; if you play a little bit with the code, you will realize that the results do not change at all if one uses more Newton iterations, which is certainly a good sign for its validity.

Be that as it may, the output of the code is this:

\[
\begin{array}{l}
\text{h=4, } x(4)=1.242050592530614, \ e=-0.3584700713525413 \\
\text{h=2, } x(4)=1.366433810669286, \ e=-0.2340868532138694 \\
\text{h=1, } x(4)=1.464780494534157, \ e=-0.1357401693489979 \\
\text{h=0.5, } x(4)=1.528073507701123, \ e=-0.07244715618203212 \\
\text{h=0.25, } x(4)=1.563287913967574, \ e=-0.0372327499155809 \\
\text{h=0.125, } x(4)=1.581500979101436, \ e=-0.01901968478171967 \\
\text{h=0.0625, } x(4)=1.590809276768069, \ e=-0.009711387115086234 \\
\text{h=0.03125, } x(4)=1.595587749966673, \ e=-0.004932913916582127 \\
\text{h=0.015625, } x(4)=1.598030503713434, \ e=-0.002490160169721589 \\
\text{h=0.0078125, } x(4)=1.599269097661273, \ e=-0.001251566221882161 \\
\text{h=0.00390625, } x(4)=1.599893193783135, \ e=-0.0006274701000201777 \\
\text{h=0.001953125, } x(4)=1.600206499454042, \ e=-0.0003141644291131396 \\
\end{array}
\]

Again, and conforming to what we had expected, the error is decreased by a factor of 2 in each time, i.e. the method is $O(h)$.

**Crank-Nicolson/trapezoidal method**

Finally, the iteration for the Crank-Nicolson scheme is

\[
x_{k+1} = x_k + \frac{h}{2} \left[ f(t_k, x_k) + f(t_{k+1}, x_{k+1}) \right].
\]

Again, $x_{k+1}$ appears both on the left and right hand side of the equation, and we have to solve an implicit equation for it. To do so, let us again write it as a root-finding problem, this time for the function

\[
g(x_{k+1}) = x_{k+1} - x_k - \frac{h}{2} \left[ f(t_k, x_k) + f(t_{k+1}, x_{k+1}) \right].
\]

The Newton iteration is again

\[
(x_{k+1})_{i+1} = (x_{k+1})_i - \frac{g((x_{k+1})_i)}{g'((x_{k+1})_i)}.
\]
but this time we have

\[ g'(x_{k+1}) = 1 - \frac{h}{2} f_x(t_{k+1}, x_{k+1}). \]

With this, we have to change above code only slightly (in the definitions of \( g \) and \( g' \)):

```cpp
#include <iostream>
#include <iomanip>

double f (const double x)
{
    return 1./(3*x*x);
}

double f_prime (const double x)
{
    return -2./(3*x*x*3);
}

double g (const double xkp1,
         const double xk,
         const double h)
{
    return xkp1 - xk - h/2*f(xkp1) - h/2*f(xk);
}

double g_prime (const double xkp1,
                const double h)
{
    return 1 - h/2*f_prime(xkp1);
}

double compute_xkp1 (const double xk,
                      const double h)
{
    double xkp1 = xk;
    for (unsigned int it=0; it<4; ++it)
        xkp1 = xkp1 - g(xkp1, xk, h) / g_prime(xkp1, h);
    return xkp1;
```
```cpp
int main ()
{
    std::cout << std::setprecision(16);

    for (double h = 4; h>1./1024; h/=2)
    {
        double x = 1./std::pow(10., 1./3.);

        double t = 0;
        do
        {
            x = compute_xkp1 (x, h);
            t += h;
        } while (t<4);

        std::cout << "h=" << h <<", x(4)=" << x
        <<", e=" << x-std::pow(4.1, 1./3)
        << std::endl;
    }
}
```

The output is:

- h=4, x(4)=3.60971452692846, e=2.009193863045305
- h=2, x(4)=2.231250697851338, e=0.6307300339681825
- h=1, x(4)=1.752636363085053, e=0.1521156992018977
- h=0.5, x(4)=1.634152497171556, e=0.0336183328840114
- h=0.25, x(4)=1.608231525053629, e=0.00771086117047148
- h=0.125, x(4)=1.602377777209178, e=0.001857108326022594
- h=0.0625, x(4)=1.600980400103171, e=0.0004597362200158273
- h=0.03125, x(4)=1.600635441345253, e=0.0001147774620982034
- h=0.015625, x(4)=1.600549358648323, e=2.869476516731417e-05
- h=0.0078125, x(4)=1.600527838065252, e=7.174182096614246e-06
- h=0.00390625, x(4)=1.60052245747632, e=1.793593164434171e-06
- h=0.001953125, x(4)=1.600521112284994, e=4.48401833266175e-07

As for the second-order Taylor method, this is clearly a quadratically convergent method.

**Fourth order Runge-Kutta method**

Just for comparison, we also include the fourth order Runge-Kutta (RK4) method. There, we had to compute the following terms (explicitly, i.e. no
solution of an implicit equation is necessary):

\[ F_1 = hf(t_k, x_k), \]
\[ F_2 = hf\left(t_k + \frac{h}{2}, x_k + \frac{F_1}{2}\right), \]
\[ F_3 = hf\left(t_k + \frac{h}{2}, x_k + \frac{F_2}{2}\right), \]
\[ F_4 = hf(t_k + h, x_k + F_3), \]
\[ x_{k+1} = x_k + \frac{1}{6}(F_1 + 2F_2 + 2F_3 + F_4). \]

This is implemented in this program:

```cpp
#include <iostream>
#include <iomanip>

int main ()
{
    std::cout << std::setprecision(16);
    for (double h = 4; h>1./1024; h/=2)
    {
        double x = 1./std::pow(10., 1./3.);

        double t = 0;
        do
        {
            const double F1 = h * 1./(3*x*x);
            const double F2 = h * 1./(3*(x+F1/2)*(x+F1/2));
            const double F3 = h * 1./(3*(x+F2/2)*(x+F2/2));
            const double F4 = h * 1./(3*(x+F3)*(x+F3));

            // compute x(k+1)
            x = x+(F1+2*F2+2*F3+F4)/6;
            t += h;
        } while (t<4);

        std::cout << "h=" << h << " , x(4)=" << x << " , e=" << x-std::pow(4.1, 1./3) << std::endl;
    }
}
```

The output, again, is this:

h=4, x(4)=3.202228877071321, e=1.601708213818866
h=2, x(4)=1.980996611970487, e=0.3804759480873319
h=1, x(4)=1.657623838525743, e=0.05710772204258818
h=0.5, x(4)=1.606809685024954, e=0.006283468216557008
h=0.25, x(4)=1.60149010704811, e=0.0005283468216557008
h=0.125, x(4)=1.60055529083791, e=3.462695475464628e-05
h=0.0625, x(4)=1.600522610806136, e=1.94692981053666e-06
h=0.03125, x(4)=1.600520770128423, e=1.062452681388493e-07
h=0.015625, x(4)=1.6005206669895427, e=6.01227179153408e-09
h=0.0078125, x(4)=1.60052066423761, e=3.544549098677458e-10
h=0.00390625, x(4)=1.600520663894623, e=2.146771649361888e-11
h=0.001953125, x(4)=1.600520663884473, e=1.317834730230612e-12

It is not quite as easy to see the convergence order at first glance here, but a closer look reveals that for the last few step sizes, the error is reduced by a factor of roughly 16 each time \( h \) is reduced by a factor of 2. That means, the method is \( O(h^4) \), or fourth-order accurate.

**Interpretation**

We can plot the errors of all these methods in a single graph against the mesh width \( h \):

As can be seen, the lines representing the errors as functions of \( h \) all tend to zero as \( h \) goes to zero (i.e. as we move from right to left). However, some decay faster and some slower. The two with the slowest decay are the two methods of first order. The second group, again one explicit and one implicit one, are the two second-order methods. They are more expensive to compute than the explicit Euler scheme, but for \( h = \frac{1}{512} \), the error is also smaller by a factor of about 600, not to be snuffed at.

The RK4 method was included for comparison, and it becomes immediately clear why it is so popular: it is an explicit method requiring only four function
evaluations per iteration, i.e. it is four times as expensive as the cheapest method (the first order Taylor expansion, i.e. the forward Euler method), but at the same $h$ it yields an error that is a whopping 8 orders of magnitude more accurate!

**Ways to botch it**

The codes above seem simple enough, but there are plenty of ways to mess up. Here’s a particularly hideous one, one that I know at least some of you encountered. Take, for example, the RK4 method, and change the main loop to this:

```cpp
do{
    const double F1 = h * 1./(3*x*x);
    const double F2 = h * 1./(3*(x+F1/2)*(x+F1/2));
    const double F3 = h * 1./(3*(x+F2/2)*(x+F2/2));
    const double F4 = h * 1./(3*(x+F3)*(x+F3));
    // compute x(k+1)
    x = x+(F1+2*F2+2*F3+F4)/6;
    t += h;
} while (t<=4);
```

The only change is that we changed the end condition to say: compute as long as the time is less than or equal to $4$. That’s certainly an error easy to make.

Here’s what happens:

- $h=4$, $x(4)=3.327306946760034$, $e=1.726786282876879$
- $h=2$, $x(4)=2.13809054246543$, $e=0.53756987853388$
- $h=1$, $x(4)=1.771009064659026$, $e=0.1704884007758711$
- $h=0.5$, $x(4)=1.668930611718625$, $e=0.06840994785354939$
- $h=0.25$, $x(4)=1.632919816124533$, $e=0.03239915224137735$
- $h=0.125$, $x(4)=1.616657503192022$, $e=0.0161368393086656$
- $h=0.0625$, $x(4)=1.608614339843623$, $e=0.00809367596046817$
- $h=0.03125$, $x(4)=1.604576845384881$, $e=0.004056181501725931$
- $h=0.015625$, $x(4)=1.60255127427033$, $e=0.002030610387174558$
- $h=0.0078125$, $x(4)=1.601536610088162$, $e=0.001015946205006779$
- $h=0.00390625$, $x(4)=1.6001028797999287$, $e=0.000508134116134867$
- $h=0.001953125$, $x(4)=1.6000747771256075$, $e=0.0002541073729200871$

So lo and behold, the method is suddenly only of first order again!?! One can spend a long time figuring out what exactly is going on, but here’s the reasoning: with this change, when we are at $t = 4$, rather than exiting the loop, we do one more iteration of the loop. This means, that rather than computing $x_N = x(4) + O(h^4)$, the final value of our variable $x$ is $x_{N+1} = x(4+h) + O(h^4) = x(4) + hx'(4) + O(h^2) + O(h^4) = x(4) + O(h)$. In other words, the high order
term $O(h^4)$ is completely subsumed by the low order term $O(h)$ that we get from the Taylor expansion $x(4 + h) = x(4) + O(h)$. Little mistake, big effect!

Another possibility to get wrong results opens up if your time step doesn’t divide the end time exactly. For example, if we tried to compute up to time $T = 4$ with a time step of $h = 1.5$, the (correct, as shown in the previous subsections) loop would actually run until time 4.5. One way to guard against this would be to write the codes like this:

```c
    do
        { 
            // do whatever it takes to compute x for this time step 
            t += h;
        }
    while (t<4-h);

    // t is now less than h away from the end time, so do 
    // one time step of (shorter than before) length h=4-t 
    h = 4-t;
    // ... do whatever it takes to compute x on this 
    // final time step
```

**Problem 2 (Numerical solution of a vector-valued ODE).** The problem is stated as a differential equation of order 2, i.e. in the equation we have second derivatives of the distance variable $d(t)$. This is not according to the standard form, so we have to introduce an auxiliary variable. We define this variable, which is a velocity, through $v(t) = d'(t)$. The full ODE system in standard form then reads

$$
d'(t) = v(t), \hspace{1cm} d(0) = 0,
$$

$$
v'(t) = f(t,d(t)), \hspace{1cm} v(0) = 0,
$$

where $f(t,d(t)) = \frac{F(t)}{m(t)}$, with $F(t)$ depending on the distance $d(t)$ again.

The following short, and mostly self-explanatory program solves this problem:

```c
#include <iostream>

double m(double t)
{
    return (t<600 ? 1.-0.9*t/600 : 0.1);
}

double T(double t)
{
    return (t<600 ? 12 : 0);
}
```
double G(double h, double t)
{
    return -6.371e6*6.371e6*10*m(t)/h/h;
}

double F(double t, double h)
{
    return T(t) + G(h,t);
}

double a(double t, double h)
{
    return F(t,h) / m(t);
}

int main ()
{
    double h = 6.371e6;
    double v = 0;
    double dt = .001;
    double t = 0;
    do
    {
        const double f_v = a(t,h);
        const double f_h = v;
        v = v+dt*f_v;
        h = h+dt*f_h;
        t += dt;
        std::cout << t << ' ' << h << ' ' << v << std::endl;
    }
    while (t<36000);
}
If we plot the distance from earth, we get the following graphs (left an inset up to time $t = 800$, at the right for the full time span):

What can be seen from these graphs is that the rocket really starts to gain height only as the time approaches $t = 600$. That is not particularly surprising: at the beginning, the thrust has to accelerate the entire rocket including all the leftover fuel. As time progresses, the mass of the rocket becomes smaller and smaller as fuel is burnt; since the thrust stays the same, this means that the acceleration gets larger and larger. After the rocket’s engines shut down at $t = 600$, the rocket flies on a ballistic trajectory, i.e. its path is changed only by gravity. Because it is already fast enough, it is slowed down by gravity but continues its journey upward (and will eventually escape earth’s gravity field).

As for determining the altitude of the rocket at $t = 36000$ accurately: with a time step of $h = 1$ as used in the program shown above, the distance $d(36000)$ from the earth core after 10 hours of flight is $3.2534 \cdot 10^8$, i.e. roughly 325,000km, or almost as far away from earth as the moon is. In order to determine how accurate this number is, let us compute it with a sequence of decreasingly smaller time steps:

<table>
<thead>
<tr>
<th>$h$</th>
<th>$d(36000)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>$-5.29531 \cdot 10^9$</td>
</tr>
<tr>
<td>10</td>
<td>$2.91001 \cdot 10^8$</td>
</tr>
<tr>
<td>1</td>
<td>$3.21895 \cdot 10^8$</td>
</tr>
<tr>
<td>0.1</td>
<td>$3.24974 \cdot 10^8$</td>
</tr>
<tr>
<td>0.01</td>
<td>$3.25341 \cdot 10^8$</td>
</tr>
<tr>
<td>0.001</td>
<td>$3.25317 \cdot 10^8$</td>
</tr>
</tbody>
</table>

(The first line is obviously nonsensical, but so is the time step of 100 seconds.) For an accuracy of 100 meter, we need to know $d(36000)$ up to 6 digits beyond the decimal point. This is obviously hard to achieve using this method, given that for $h = 0.001$ we seem to have at best four digits beyond the decimal point accurately. Because we reduce the error by a factor of 10 each time we reduce $h$ by a factor of 10 (this method is linearly convergent, after all), we gain one digit of accuracy in each such step; comparing with the accuracy we have with $h = 0.001$, this means that to determine $d(36000)$ accurately to within 100 meters would require a step size $h \approx 10^{-5}$; to use such a step size would make the
program run for approximately 8 hours on my, already pretty fast, desktop – an exercise I leave to you. It may serve as an illustration that linearly convergent methods are usually not particularly fast, and that it is usually beneficial to use a higher order method; for example, the quadratically convergent Crank-Nicolson method gains 2 digits of accuracy with each reduction of \( h \) by a factor of 10, and the fourth order Runge-Kutta method gains 4 digits each time. It is apparent that we can get away with a significantly larger \( h \) for these methods than what is required for the explicit Euler method used here.

**Problem 3 (Some parameter determination with ODEs).** Let us first try to get a feel for the range in which we have to expect the parameter to be. For this, assume that the 50 m/s is the terminal velocity, i.e. the velocity at which air friction and gravity exactly cancel each other. In that case, the skydiver will not get any faster anymore, i.e. \( v' = 0 \). Since \( v' = 10 - av^2 \), this would imply that \( av^2 = 10 \), and with assumed velocity of 50 m/s, that \( a = 10/2500 = 0.004 \).

With this rough guess for \( a \), we can numerically approximate the ODE. Here’s the most trivial program I could come up with for this particular purpose that implements the forward Euler method:

```cpp
#include <iostream>

int main ()
{
    double v = 0;
    double a = 0.004;

double dt = .01;
for (double t=0; t<=20; t+=dt)
{
    const double f_v = 10-a*v*v;
    v = v+dt*f_v;
    std::cout << t << ' ' << v << std::endl;
}
}
```

With it, we get the following graph for the velocity:
So we see that the velocity at time $t = 10$ hasn’t yet increased to 50 m/s, but that it is getting close. Apparently we have overestimated the air friction coefficient (the velocity is smaller than expected), but have guessed close.

So how do we compute $a$ more accurately so that $v(10) = 50$? The answer to this lies in taking the right viewpoint of the problem, namely that we can consider the velocity at time 10 as a function of the coefficient $a$: for each value of $a$, we can compute $v(10)$ using the forward Euler method. Let us denote this by indexing $v_a(10)$ to express the fact that $v_a(10)$ is really a function of $a$. We are then interested in finding a value of $a$ such that $v_a(10) = 50$, or in other words: we are looking for a zero of the function

$$f(a) = v_a(10) - 50.$$

We have learned a number of methods that find the root of such functions. One of the problems we face here is that we don’t quite know what $f'(a)$ is (one can define this derivative properly, but this is beyond the scope of this class), so we can only use methods that don’t require the derivative. The bisection method is one of these. What we then need to do is define a function that computes $f(a)$ for a given value of $a$ by integrating the ODE to $t = 10$ and subtracting 50. Next we need an initial interval for where to find the solution; we already know that $a = 0.004$ is too large, so let’s take 0.002 and 0.004. With this, we can use the program for question 3a) of homework 2, which then reads as follows (the left and right end points of the present interval are denoted by $a_1$ and $a_2$):

```cpp
#include <iostream>
#include <iomanip>
#include <cmath>

// define f(a)
double f(double a) {
  
}
double v = 0;
double dt = .01;
for (double t=0; t<=10; t+=dt)
{
    const double f_v = 10-a*v*v;
    v = v+dt*f_v;
}
return v-50;
}

int main ()
{
    // set output precision to all 16 valid
    // digits of double precision floating
    // point numbers
    std::cout << std::setprecision(16);
    // specify the initial interval
    double a1 = 0.002;
    double a2 = 0.004;
    // then iterate:
    for (unsigned int n=0; n<10; ++n)
    {
        // check that the function values at
        // the endpoints of the present
        // interval have different signs
        if (f(a1) * f(a2) >= 0)
        {
            std::cout << "Invalid interval endpoints!" << std::endl;
            abort ();
        }
        // evaluate the midpoint of the
        // interval
        const double c = (a1+a2)/2;
        // the best guess for where the zero is
        // was at the middle of the interval
        std::cout << "a_" << n << " = " << c << std::endl;
        // then set interval for the next
        // iteration
}
if (f(a1) * f(c) < 0)
    a2 = c;
else
    a1 = c;
}
}

Within 10 iterations, we get the following output:

<table>
<thead>
<tr>
<th>i</th>
<th>a_0</th>
<th>a_1</th>
<th>a_2</th>
<th>a_3</th>
<th>a_4</th>
<th>a_5</th>
<th>a_6</th>
<th>a_7</th>
<th>a_8</th>
<th>a_9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.003</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.0035</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.00375</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.003625</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.0036875</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.00365625</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.003671875</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0.0036640625</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0.00366796875</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0.003669921875</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This gives us confidence that the sought parameter has the value \( a = 0.00367 \), with more accuracy easily achievable. If we take this value back into our original ODE solver, we get the following graph:

As can be seen, the velocity at times \( t = 10 \) hits the 50m/s pretty exactly, at least to within the “eyeball-norm” (as good as one can distinguish by looking at a graph; the value for \( v(10) \) for this particular value of the parameter \( a \) is in fact 50.0035, close enough for our purposes).

In the early part of this class, we have seen that the bisection method is rather robust, but doesn’t converge particularly quickly (it has only linear convergence). On the other hand, the Newton method converges very quickly.
(quadratically, to be exact), so it would have been nice if we could have used it here – we can’t immediately, however, because it requires $f'(a)$ and we don’t know how to approximate it. One way to do so would have been to approximate every appearance of $f'(a)$ in the Newton formulas by $\frac{f(a+\delta a) - f(a-\delta a)}{2\delta a}$, i.e. a second order finite difference approximation of the derivative, with some small $\delta a$, say 0.0001.

As a remark: If one continues to compute the solution beyond $t = 10$, one will find that the velocity only increases to about 52.2m/s $\approx 117$ mph, before becoming essentially constant. This is actually behavior for skydiving velocity: one reaches almost terminal velocity after about 10 seconds.

As a second remark: In the program above, I have approximated $v_a(10)$ using the forward Euler method with a step size of $h = 0.01$. You should note that this is only a numerical approximation to the exact solution. If we had chosen a different value of $h$, we may well have gotten something else than 50.0035 for $v(10)$ and the value of $a$ computed above. What that means is that if you choose a different $h$, then your $v_a(10)$ changes, and the parameter $a$ you compute such that $v_a(10) = 50$ may be different than the one above. In particular, if $h$ were significantly too large, say $h = 1$, then $v(10)$ would have a rather large error and your computed value for $a$ would be wrong as well; only if you choose $h$ small enough can you be certain that the $a$ you computed actually implies that the exact solution (no numerical approximation) $v_a(10) = 50$. 