Some Thoughts about Numerical Methods (part of the course 'Introduction to Flight Simulation')

Hans de Nivelle
Instytut Informatyki Uniwersytetu Wrocławskiego

October 27, 2010

1 Accuracy of Numerical Methods

Since it turned out that you do not understand much of floating point calculations, I will give some basic facts:

- Calculations on type `double` have a precision of approximately 2.10^{-16}. The precision of the data type `double` is a bit higher, but after some experimenting with library functions I concluded that more should not be expected.
- The speed of modern processors is insane. It is easy to do a numerical computation of 10^8 steps and have reasonable waiting times.

Suppose that you want to do some summation of the following form, in order to compute an integral or solve a differential equation:

```
Let h = some small number.
sum = 0.0
while( x0 < x1 )
{
   sum += G( x, sum, h )
   x0 += h
}
```

In most cases, G is not exactly equal to the function that you want to compute, but an approximation. In general, the difference between the exact function that you want to compute, and $G(x, sum, h)$, has form Ah^n, with $n > 1$, and A some fixed number. We will write that the error is $O(h^n)$, because in most cases we are too lazy to compute A.

In most cases, the simple algorithm will have $n = 2$, so that the total error in the computation decreases linearly with the step size h. You might think that this is good enough for most applications, because modern computers are very fast, and one can take a very small step size h. This turns out to be not the case.

For a single step precision $O(h^2)$, the method can be easily improved 10 times by making h 10 times smaller. The error in a single step will be 100 times better, but because the error is repeated 10 times, the result will be only 10 time better. Unfortunately, this will not go on forever. If h gets close to 10^{-8}, the accumulated floating point error will be close to $10^8.2.10^{-16} \approx 10^{-8}$. This means that with an accurracy of $O(n^2)$, one cannot reasonably expect more than 8 correct decimals. If one wants more decimals, it will be necessary to use a method with better convergence rate.

The picture with $n = 3$ is a bit better. With $h \approx 10^{-6}$, the floating point error will be 10^{-12}, and $h^2 \approx 10^{-12}$, so that one can hope for 12 correct decimals.

Unfortunately, from $n = 3$ onwards, it becomes difficult to get more decimals, but one can still reduce the cost of the computation. In order to get 14 decimals right, one would need a method of n^7, which in most cases is not practical.

So the message is: It seems that 12 decimals is about the best one can get with numerical methods based on summation.

If you want more than 8 decimals, you will need to put effort in improving your integration algorithm, because with a naive algorithm, 8 decimals is the best you can get. Note that with `float`, it would be only 5 decimals.

2 Approximation Methods

All approximation methods are based on the existence of Taylor polynomials.

If one wants to compute some functions $f(x)$, and one knows that $f(x)$ has form $B_0 + B_1x + B_2x^2 + B_3x^3 + \cdots + O(x^k)$, one can get a k-th order method. The easiest way of course would be to get the values of $B_0, B_1, B_2, \ldots$ from somewhere, but in practice this is often not possible.
In that case, there exist approximation methods, which are based on the fact that the Taylor approximation exists, but without the need to explicitly compute the $B_0, B_1, B_2, \ldots$. More precisely, these methods use the fact that the Taylor polynomial exists in the convergence proof, but the Taylor polynomial does not occur in the method itself.

Examples of such methods are *Heun's* method, and *Runge-Kutta* methods. We will eventually explain how to prove the fact that RK has an accurrancy of $O(h^5)$. The proof is conceptually simple, but very difficult in practice, due to the large polynomials involved. It is even a challence for modern computer algebra systems.

2.1 Taylor Polynomials

The fact that functions can be numerically approximated at all, is due to the existence of Taylor polynomials. We give Taylor's theorem for a single variable:

Theorem 2.1 *Let F be a function with signature $\mathcal{R} \to \mathcal{R}$.*

Let $x \in \mathcal{R}$ and assume that I is an open interval containing x. Assume that F is at least $(n+1)$ times differentiable in the internal I.

Then for every h, s.t. $(x+h) \in I$, there exists a j between 0 and h, s.t.

$$F(x+h) =$$

$$F(x)+hF'(x)+\frac{h}{2}F''(x)+\frac{h^3}{3!}F^{(3)}(x)+\frac{h^4}{4!}F^{(4)}(x)+\frac{h^5}{5!}F^{(5)}(x)+\cdots+\frac{j^{n+1}}{(n+1)!}F^{(n+1)}(x+j).$$

It is important to understand that the Taylor polynomial gives a precise bound. Theorem 2.1 does not tell 'the polynomial somehow gets close to the correct value if you compute enough terms', it give a very precise bound on how close it gets with how much computation. If you know the possible values of $F^{n+1}(j)$, then you have a hard guarantee how close the Taylor polynomial gets.

Proof

We use the fact that, for an arbitrary, differentiable function G,

$$G(b) = G(a) + \int_a^b G'(x)\ dx.$$

In order to avoid problems with dangling $dx's$, we use the notation:

$$G(b) = G(a) + \int_{x=a}^b G'(x).$$

Since F is $n+1$ times differentiable, we have

$$F(x+h_0) = F(x) + \int_{h_1=0}^{h_0} F'(x+h_1),$$

$$F'(x+h_1) = F'(x) + \int_{h_2=0}^{h_1} F''(x+h_2),$$

$$F''(x+h_2) = F''(x) + \int_{h_3=0}^{h_2} F'''(x+h_3),$$

$$F'''(x+h_3) = F'''(x) + \int_{h_4=0}^{h_3} F^{(4)}(x+h_4),$$

$$\cdots$$

$$F^{(n)}(x+h) = F^{(n)}(x) + \int_{j=0}^{h_n} F^{(n+1)}(x+j).$$

By substituting, one obtains

$$F(x+h_0) = F(x)+\int_{h_1=0}^{h_0} F'(x)+\int_{h_2=0}^{h_1} F''(x)+\int_{h_3=0}^{h_2} F'''(x)+\cdots+\int_{j=0}^{h_n} F^{(n+1)}(x+j).$$

(Note that each integral sign extends all the way to the end.) In this formula, each of $F(x), F'(x), F''(x), F'''(x), \ldots$ is a constant number. In addition,

$$\int_{h=a}^{b} G_1(h) + G_2(h) = \int_{h=a}^{b} G_1(h) + \int_{h=a}^{b} G_2(h),$$

so that we can get all $F^{(i)}(0)(x)$ out of the integrals. The result is

$$F(x+h_0) = F(x) + h_0.F'(x) + \frac{h_0^2}{2!}F'(x) + \frac{h_0^3}{3!}F''(x) + \frac{h_0^4}{4!}F'''(x) + \cdots$$

$$+\int_{h_1=0}^{h_0}\int_{h_2=0}^{h_1}\int_{h_3=0}^{h_2}\cdots\int_{j=0}^{h_n} F^{(n+1)}(x+j).$$

This is more or less what we need, but we still have to estimate the last integral. This we do in the following lemma:

Lemma 2.2 *Let $n \geq 0$ be a natural number, let G be a function of signature $\mathcal{R} \to \mathcal{R}$. Let $a \in \mathcal{R}$, let $h_0 \geq 0$. Assume that G is defined on the interval $[a, a+h_0]$. Then there exists a j, also in the interval $[a, a+h_0]$, s.t.*

$$\int_{h_1=0}^{h_0}\int_{h_2=0}^{h_1}\int_{h_3=0}^{h_2}\cdots\int_{h_{n+1}=0}^{h_n} G(h_{n+1}) = \frac{j^{n+1}}{(n+1)!}G(x+j).$$

Proof
I will not write completely write out the proof, but the intuition is like this: First try $j = 0$. If $j = 0$ is a solution, then the proof is complete. Otherwise, $\frac{j^{n+1}}{(n+1)!}G(x+j)$ is either too big or too small. Without loss of generality, assume that it is too small.

Theorem 2.3 *Let F be a function with signature $\mathcal{R}^2 \to \mathcal{R}$. Let $(x, y) \in \mathcal{R}^2$, and assume that I is an open, circular interval containing (x, y). Assume that F is at least $(n+1)$ times (partially) differentiable on the interval I. Then for every h, k, s.t. $(x+h,\ y+k) \in I$, there exist h' and k', s.t. h' is between 0 and h, and k' is between 0 and k, and:*

$$F(x+h, y+k) =$$

$$F(x,y)+$$

$$h\frac{\partial F}{\partial x}(x,y) + k\frac{\partial F}{\partial y}(x,y)+$$

$$\frac{h^2}{2!}\frac{\partial F^2}{\partial x^2}(x,y) + hk\frac{\partial F^2}{\partial xy}(x,y) + \frac{k^2}{2!}\frac{\partial F^2}{\partial x^2}(x,y)+$$

$$\frac{h^3}{3!}\frac{\partial F^3}{\partial x^3}(x,y) + \frac{h^2k}{2!}\frac{\partial F^3}{\partial x^2 y}(x,y) + \frac{hk^2}{2!}\frac{\partial F^3}{\partial xy^2}(x,y) + \frac{k^3}{3!}\frac{\partial F^3}{\partial y^3}(x,y) +$$

$$\frac{h^4}{4!}\frac{\partial F^4}{\partial x^4}(x,y) + \frac{h^3k}{3!}\frac{\partial F^4}{\partial x^3 y}(x,y) + \frac{h^2k^2}{2!\,2!}\frac{\partial F^4}{\partial x^2y^2}(x,y) + \frac{hk^3}{3!}\frac{\partial F^4}{\partial xy^3}(x,y) + \frac{k^4}{4!}\frac{\partial F^4}{\partial y^4}(x,y) + \cdots$$

3 Approximation of Integrals by Simpson's Rule

Assume that $f(x_0) = y_0$. Assume that $f(x_0 + h) = \int_{k=0}^{h} F(x_0 + k)$. Then

$$f(x_0 + h) = \frac{h}{6}[\ F(x_0) + 4\,F(x_0 + \frac{1}{2}h) + F(x_0 + h)\ ] + \frac{h}{2880}G^{(5)}(j),$$

for a j between x_0 and $x_0 + h$.

4 First-Order Differential Equations

We study the problem of numericaly solving differential equations of the following form: Suppose that we are given a two place function $F(x, y)$, which we know how to compute, and a number $y_0 \in \mathcal{R}$. Let f be the one place function that is characterized by:

$$f(x_0) = y_0, \quad \text{and for all } x \quad f'(x) = F(x, f(x)). \tag{1}$$

How to compute f?
Solving differential equations of form 1 is harder than solving integrals, because with with an integral, the derivative of f depends only on x. In the differential equation, it also depends on values of $f(x)$, which we don't know yet. In case, $F(x, y)$ does not depend on y, the problem becomes equal to the problem of computing an integral.
In the next three sections, we will dicuss methods for solving first-order differential equations. The first method is the method that every computer scientist would invent. Unfortunately, it was already invented by Leonhard Euler and hence called *Euler's Method*. The second method, which has a better convergence rate, is called Heun's method and the last method is the Runge Kutta method, which is quite sophisticated.

4.1 Euler's Method

As said before, Euler's method is the algorithm that every computer scientist would come up with:

```
Assume that x0, y0 are given, and that y0 = f(x0).
Assume that we want to compute f(x1).

Let h = some small number.
```

```
while( x0 < x1 )
{
   y0 = y0 + h * F(x0,y0)
   x0 += h
}

Now x0 == x1 and y0 = f(x1).
```

The algorithm is slightly too simplistic, because it assumes that x_0 will become exactly equal to x_1. In practice, the algorithm will probably terminate with some $x_0 > x_1$, and it will have computed $f(x_0)$ for a number distinct from x_1. This may result in a rather large loss of accurracy. In order to avoid that, the algorithm should give special attention to the last step, so that it ends with $x_0 = x_1$.
The algorithm is supposed to maintain the invariant $y_0 = f(x_0)$, where f is the function that solves the original equation, and x_0 is a real number that moves slowly towards x_1.
In practice, the computed value y_0 will differ from $f(x_0)$, due to two reasons: **(1)** If $y_0 = f(x_0)$, then $f(x_0 + h)$ is probably not equal to $y_0 + h.F(x_0, y_0)$. **(2)** The algorithm accumulates floating point errors.

In order to estimate **(1)**, one needs to assume that f is at least two times differentiable, everywhere between x_0 and x_1. In that case, $f(x_0 + h)$ can be represented by a Taylor polynomial of form $B_0 + B_1h + B_2h^2 + O(h^3)$. By substituting $h = 0$, we see that $B_0 = y_0$. Since B_1 is the first derivative of f, it must be the case that $B_1 = F(x_0, y_0)$. Using these facts, it can be seen that Euler's method computes $y_1 = y_0 + h.F(x_0, y_0) = B_0 + B_1.h$, so that the error of one step equals $B_2h^2 + O(h^3)$. Doing some handwaving, we assume that this error is made n times, where n is the number of steps of the algorithm. This results in an error of $O(h^2.n)$. Since n is equal to $(y_1 - y_0)/h$ (The total distance between y_1 and y_0 divided by the step size h.), we end up with an error of $O(h)$. The floating point error **(2)** can be estimated as $2n.10^{-16} = 2.10^{-16}/h$ for reasonable F.
If one assumes that the first error $O(h)$ is approximately equal to h, then it follows that the maximal accurracy obtainable by Euler's method is approximately 10^{-8}, which is obtained for the value $h \approx 10^{-8}$. That is not very good.

4.2 Heun's Method

Heun's method is obtained by replacing the while loop in the previous algorithm by

```
{
   k1 = F( x0, y0 )
   k2 = F( x0 + h, y0 + h * k1 )
   y1 = y0 + h * 0.5 * ( k1 + k2 )
```

```
    x0 += h;
}
```

Heun's method first uses Euler's method to estimate $f(x_0 + h)$. After that it uses the trapezium method to improve the estimation of y_1.
In the rest of this section, we prove that the error for Heun's method is $O(h^3)$ for a single step. The structure of the proof is quite simple, but writing out the details is tedious. The structure of the proof is as follows:

1. Approximate F near x_0, y_0 by a two dimensional Taylor polynomial.
2. Approximate f near x_0 by a Taylor polynomial.
3. Solve the coefficients of the Taylor polynomial of f in terms of the coefficients of the Taylor polynomial of F.
4. Using the Taylor polynomial for F, express k_1, k_2 and y_1^H by Taylor polynomials.
5. Compare the polynomials of step 3 and step 4, and agree that the first four coeffcients are equal.

Step 1: Using Theorem 2.3, values of F around x_0, y_0 can be approximated by

$$F(x_0{+}h,\ y_0{+}k) = \left\{ \begin{array}{lllllllll} A_{0,0} & + & & & & & & & \\ A_{1,0}h & + & A_{1,1}k & + & & & & & \\ A_{2,0}h^2 & + & A_{2,1}hk & + & A_{2,2}k^2 & + & & & \\ A_{3,0}h^3 & + & A_{3,1}h^2k & + & A_{3,2}hk^2 & + & A_{3,3}k^3 & & \\ O(h^4) & + & O(h^3k) & + & O(h^2k^2) & + & O(hk^3) & + & O(k^4) \end{array} \right.$$

Step 2: By Theorem 2.1, values of f near x_0 can be approximated by a polynomial of form:

$$f(x_0 + h) = B_0 + B_1h + B_2h^2 + B_3h^3 + B_4h^4 + O(h^5).$$

Note that $B_0 = y_0$, because $f(x_0) = y_0$.
Step 3: From the differential equation 1, we know that

$$f'(x_0 + h) = F(x_0 + h, f(x_0 + h)).$$

It is easy to compute the left hand side:

$$f'(x_0 + h) = B_1 + 2B_2h + 3B_3h^2 + 4B_4h^3 + O(h^4). \tag{2}$$

On the right hand side, things are a bit harder:

$$F(x_0 + h, f(x_0 + h)) = F(x_0 + h,\ B_0 + B_1h + B_2h^2 + B_3h^3 + B_4h^4 + O(h^5)) =$$

$$F(x_0 + h,\ y_0 + B_1h + B_2h^2 + B_3h^3 + B_4h^4 + O(h^5)).$$

This formula can be evaluated by substituting $k := (B_1h+B_2h^2+B_3h^3+B_4h^4+O(h^5)\ )$ in the polynomial of step 1. The result is the following, admittedly somewhat intimidating, polynomial:

$$\begin{array}{l}
h^0\ (A_{0,0}) \quad + \\
h^1\ (A_{1,1}B_1 + A_{1,0}) \quad + \\
h^2\ (A_{2,2}B_1^2 + A_{2,1}B_1 + A_{2,0} + A_{1,1}B_2) \quad + \\
h^3\ (A_{3,3}B_1^3 + A_{3,2}B_1^2 + A_{3,1}B_1 + A_{3,0} + 2A_{2,2}B_1B_2 + A_{2,1}B_2 + A_{1,1}B_3) \quad + \\
h^4\ (A_{4,4}B_1^4 + A_{4,3}B_1^3 + A_{4,2}B_1^2 + A_{4,1}B_1 + A_{4,0} + \\
\quad 3A_{3,3}B_2B_1^2 + 2A_{3,2}B_2B_1 + A_{3,1}B_2 + \\
\quad A_{2,2}B_2^2 + 2A_{2,2}B_1B_3 + A_{2,1}B_3 + A_{1,1}B_4) \quad + \\
O(h^5).
\end{array}$$

Although this polynomial is very big, its format is actually very convenient. By comparing it termwise to the polynomial in equation 2, we obtain the following equations:

$$B_1 = A_{0,0}$$

$$B_2 = \frac{A_{1,0} + B_1A_{1,1}}{2}$$

$$B_3 = \frac{A_{2,2}B_1^2 + A_{2,1}B_1 + A_{2,0} + A_{1,1}B_2}{3}$$

$$B_4 = \frac{A_{3,3}B_1^3 + A_{3,2}B_1^2 + A_{3,1}B_1 + A_{3,0} + 2A_{2,2}B_1B_2 + B_2A_{2,1} + B_3A_{1,1}}{4}$$

Although the equations are big, it is in principle easy to solve B_1, B_2, B_3, B_4 from them. It can be immediately seen that $B_1 = A_{0,0}$. After that, B_1 can be replaced by $A_{0,0}$ in the other equations. Now the second equation gives an explicit solution for B_2, which can again be substituted in the rest of the equations, etc.
When the values for $B_0, B_1, B_2, B_3, \ldots$ are known, they can be substituted in the polynomial of step 2. The result is the following polynomial, which is a bit smaller, but still very unpleasant:

$$\begin{array}{l}
h^0\ \ (B_0) \quad + \\
h^1\ \ (A_{0,0}) \quad + \\
h^2\ \ (\frac{A_{1,0}}{2} + \frac{A_{0,0}A_{1,1}}{2}) \quad + \\
h^3\ \ (\frac{A_{2.0}}{3} + \frac{A_{1,0}A_{1,1}}{6} + \frac{A_{0,0}A_{2,1}}{3} + \frac{A_{0,0}A_{1,1}^2}{6} + \frac{A_{0,0}^2A_{2,2}}{3}) \quad + \\
h^4\ \ (\frac{A_{3,0}}{4} + \frac{A_{1,1}A_{2,0}}{12} + \frac{A_{1,0}A_{2,1}}{8} + \frac{A_{1,0}A_{1,1}^2}{24} + \frac{A_{0,0}A_{3,1}}{4} + \frac{5A_{0,0}A_{1,1}A_{2,1}}{24} + \\
\quad \frac{A_{0,0}A_{1,1}^3}{24} + \frac{A_{0,0}A_{1,0}A_{2,2}}{4} + \frac{A_{0,0}^2A_{3,2}}{4} + \frac{A_{0,0}^2A_{1,1}A_{2,2}}{3} + \frac{A_{0,0}^3A_{3,3}}{4}) \quad + \\
O(h^5)
\end{array} \tag{3}$$

Step 4: We express y_1^H, (the formula computed by Heun's method) as polynomial. It is easy to see that

$$k_1 = F(x_0, y_0) = A_{0,0}.$$

Also,

$$k_2 = F(x_0 + h,\ y_0 + h.k_1),$$

which can be evaluated by substituting $k := h.k_1$ $(= h.A_{0,0})$ in the Taylor polynomial of step 1. The result equals

$$\begin{array}{ll} h^0 & (A_{0,0}) + \\ h^1 & (A_{1,1}A_{0,0} + A_{1,0} + \\ h^2 & (A_{2,2}A_{0,0}^2 + A_{2,1}A_{0,0}A_{2,0}) + \\ h^3 & (A_{3,3}A_{0,0}^3 + A_{3,2}A_{0,0}^2 + A_{3,1}A_{0,0} + A_{3,0}) + \\ h^4 & (A_{4,4}A_{0,0}^4 + A_{4,3}A_{0,0}^3 + A_{4,2}A_{0,0}^2 + A_{4,1}A_{0,0} + A_{4,0}) + \\ O(h^5). & \end{array}$$

Now, the Heun approximation can be computed:

$$y_1^H = y_0 + \frac{h}{2}(k_1 + k_2) =$$

$$\begin{array}{ll} h^0 & (B_0) + \\ h^1 & (A_{0,0}) + \\ h^2 & (\frac{A_{1,1}A_{0,0}}{2} + \frac{A_{1,0}}{2}) + \\ h^3 & (\frac{A_{2,2}A_{0,0}^2}{2} + \frac{A_{2,1}A_{0,0}}{2} + \frac{A_{2,0}}{2}) + \\ h^4 & (\frac{A_{3,3}A_{0,0}^3}{2} + \frac{A_{3,2}A_{0,0}^2}{2} + \frac{A_{3,1}A_{0,0}}{2} + \frac{A_{3,0}}{2}) + \\ O(h^5). & \end{array} \tag{4}$$

Step 5: By comparing the polynomials of equations 3 and 4, we see that the first three coefficients are equal, so that the error of Heun's method is indeed $O(h^3)$.
The proof that we gave was not difficult at all, but the polynomials involved were quite large. This is mostly due to the fact that we computed more coefficients than necessary. In principle, if one wants to establish that three coefficients are equal, there is no need to compute more than three coefficients. The reasons for computing more coefficients were: We wanted to prepare for the Runge Kutta method in the next section, for which we need five coefficients. We wanted to illustrate the mechanical nature of the convergence proof, and we established that Heun's method is not $O(h^4)$. (At least we made it likely.)

5 Runge Kutta Method of O(5)

The standard Runge-Kutte method (RK4) is obtained by replacing the while loop in the algorithm of Section 4.1 by

```
{
   k1 = F( x0, y0 )
   k2 = F( x0 + 0.5 * h, y0 + 0.5 * h * k1 )
   k3 = F( x0 + 0.5 * h, y0 + 0.5 * h * k2 )
   k4 = F( x0 + h, y0 + h * k3 )

   y1 = y0 + h * ( k1 + 2.0 * k2 + 2.0 * k3 + k4 ) / 6.0
}
```

The method of Section 4.2 can also be used for proving that RK4 has a single step accurracy of order $O(h^5)$. In order to do this, one needs to compute the polynomials for y_0^{RK} and $f(x_0 + h)$, and check that they are equal up to the fifth coefficient. Since the polynomials involved are too big to process by hand, I used a computer algebra to compute y_0^{RK} up to the 7-th coefficient. The result can be found on the course homepage.

5.1 How could they derive this in 1900?

After having seen the correctness proof for RK4, there remains one very big mystery: How could they manage to make this calculation in 1900? One possibility is that there exists a pattern, which I didn't see, and which makes it easy to write down the required polynomials and solve the corresponding equations. However, the remarks in the Wikipedia article on Runge-Kutta methods, that optimal methods for order 9 and 10 are not known, make this unlikely.

My theory is that Runge and Kutta derived the scheme by using rational coefficients. Instead of using generic coefficents, one can randomly construct a concrete Taylor polynomial for F, with rational coefficients, and follow the procedure of Section 4.2 with the concrete polynomial. In step 2, the coefficients $B_0, B_1, B_2 \ldots$, can be solved as rational numbers. Using the concrete $A_{i,j}$, it is tractable to compute the coefficients k_1, k_2, k_3, k_4 as rational polynomials up to sufficiently high degree.

Using the explicit polynomials, one can use Gauss elimination to find numbers $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ for which $k_1\alpha_1 + k_2\alpha_2 + k_3\alpha_3 + k_4\alpha_4$ agrees with the polynomial $B_0 + B_1 h + B_2 h^2 + B_3 h^3 + B_4 h^4 + B_5 h^5 + \cdots$ as far as possible.

If one finds $\alpha_1, \ldots, \alpha_4$ that work for different random choices of the $A_{i,j}$, then one can be pretty sure that they will work for every $A_{i,j}$. Unless somebody tells me something different, I will assume that this is how the Runge-Kutta method was originally obtained.

6 Simulation of Planets

Planets and rockets surprisingly easy to understand, and it is easy to simulate them. For an object with mass m in three dimensional space, we have

$$\overline{F} = m.\overline{a}, \tag{5}$$

where $\overline{F}$ is the sum of all forces working on the object. The gravity attraction of an object with mass m_1 at position $\overline{x}_1$ towards an object with mass m_2 at position $\overline{x}_2$ is given by the following formula:

$$\overline{F}_{1,2} = \frac{Gm_1m_2(\overline{x}_2 - \overline{x}_1)}{|\overline{x}_2 - \overline{x}_1|^3}, \tag{6}$$

where G denotes the gravity constant $6.67428.10^{-11}$.

In order to simulate a solor system, one first computes all gravity forces using the positions and the masses of the planets, together with equation 6. After that, equation 5 can be used to compute the accelerations, which are the second derivative of the positions. We explain in the next section how to deal with second order differential equations.

6.1 Runge-Kutta methods for Second Order Differential Equations

We have seen in the previous section that the differential equations that we need in mechanics, are all of form

$$f(x_0) = y_0, \quad f'(x_0) = y'_0, \text{ and } \forall x\ f''(x) = F(x, f(x)).$$

In case there is resistance (which depends on speed), the equation takes form

$$f(x_0) = y_0, \quad f'(x_0) = y'_0, \text{ and } \forall x\ f''(x) = F(x, f(x), f'(x)) \tag{7}$$

with a 3-place function F.
Higher-order differential equations can be easily transformed into first-order differential equations on tuples. First observe that Runge-Kutta style methods can be used for finding functions f of any type $\mathcal{R} \to \mathcal{V}$. In that case f' also has type $\mathcal{R} \to \mathcal{V}$, $\quad F$ has type $\mathcal{R} \times \mathcal{V} \to \mathcal{V}$, and k_1, k_2, k_3, k_4 have type $\mathcal{V}$.
The only requirement on the type $\mathcal{V}$ is that it is closed under addition, and that elements of $\mathcal{V}$ can be multiplied with elements of $\mathcal{R}$, the result of which must be in $\mathcal{V}$ again. (This means that $\mathcal{V}$ has to be a *vector space*.)
In order to deal with Equation 7, we introduce another function $g(x)$, for which we will ensure that always $g(x) = f'(x)$. From the fact that $g(x) = f'(x)$, it follows that $g'(x) = f''(x)$. Equation 7 becomes:

$$\left\{ \begin{array}{lcl} \forall x\ \ f'(x) & = & g(x) \\ \forall x\ \ g'(x) & = & F(x, f(x), g(x)). \end{array} \right. \tag{8}$$

Using vector notation, this can be written as

$$\forall x,\ (\ f(x), g(x)\ )' = (\ g(x),\ F(x, f(x), g(x))\ ).$$

If we now define

$$\left\{ \begin{array}{lcl} \overline{f}(x) & = & (\ f(x),\ g(x)\ ), \text{ and} \\ \overline{F}(x, \overline{y}) & = & (\ \ \overline{y}[0],\ F(x, \overline{y}[0], \overline{y}[1]\ )\ \ ) \end{array} \right.$$

then the result is the following equation over $\mathcal{R}^2$,

$$\overline{f}(x) = (y_0, y_0'), \qquad \overline{f}'(x) = \overline{F}(x, \overline{f}(x)\ ),$$

which is first order and which can be solved by the Runge Kutta method in the usual way.

The physical meaning (in case of planet simulation) is that $\overline{f}(t)$ is a vector of positions and speeds of the simulated planets at time t. $\overline{F}(t, \overline{x})$ takes such a vector of positions and speeds, and must construct a vector of speeds and accelerations. The speeds can be copied from the input, and the accelerations have to be computed from the gravity interactions and the masses.

In order to implement this in C^{++}, one should define a class `planet_state`, and define addition for this class and multiplication with `double`. See `planets_rk.cpp` on the course homepage for an implementation.