Umeå Universitet
Department of Computing Science 09/01/2019

Project 3 - Accuracy in artillery computations

Aladdin Persson (alhi0008@student.umu.se)

Version 1, Username: tfy17api
Teknisk-vetenskapliga beräkningar
Supervisors: Carl Christian Kjelgaard Mikkelsen,
Niclas Börlin

Contents

1 Introduction

There are several pieces of relevant information that one could wish to calculate from a trajectory that models an artillery computation. With the ability to model and simulate very accurately, there exists the ability to test and improve within a computer only. In this project we wish to develop the ability to make accurate estimates of projectile motion in terms of trajectory range, time elapsed during motion and also find angles which corresponds to hitting a target at a specific location.

When performing calculations in computers and having the wish to do so accurately, because of finite arithmetic and therefore rounding errors, we need to find methods where we know when we can rely on the calculations.

All source code for functions and scripts in this project is included in appendix.

2 Richardson fraction and A.E.E

The method of choice to know when the calculations can be trusted to be accurate is to analyze the behavior of Richardsons fraction. This section will show the necessary results from this analysis, but will not go in too much theorethical depth of this analysis. This report is not primarily about theory behind this fraction but rather how we can use these known facts to make accurate error estimates. It is of importance to note that there exists an underlying assumption underneath using Richardson fraction which is emphasized because without realizing the faults of our assumptions, it can be difficult to debug when things aren't working. The idea behind Richardson fraction starts with comparing the true value compared to an approximation that is calculated with the use of a step size. This could be anything from numerically approximating derivatives, integrals or in this case artillery computations that are solved for numerically with Euler or Runge Kutta methods. All of these include the important part of having a step size h.

We wish to introduce this concept of Richardon's fraction and give intuition to the theory, which we will not do justice to but whose aim is to give the reader an intuition and the ability to use these facts. We start with assuming that the true value and the approximated value that depends on a step size h follow an asymptotic error expansion (A.E.E)

$$T - A_h = \alpha \cdot h^p + \beta h^q + \mathcal{O}(h^r), \tag{1}$$

where $0 < p < q < r$. Since the only option of change in the calculations is the step size h it follows naturally that we care to find a method that relies on h. By looking at

$$T - A_{2h} = \alpha \cdot (2h)^p + \beta (2h)^q + \mathcal{O}(h^r), \tag{2}$$

and then

$$(T - A_h) - (T - A_{2h}) = A_{2h} - A_h. \tag{3}$$

One can show through algebra and simplification from Eq. (3) that the primary error term is given by

$$\alpha \cdot h^p = \frac{A_h - A_{2h}}{2^p - 1} + \frac{(2^q - 1) \cdot \beta \cdot h^q}{2^p - 1} + \mathcal{O}(h^r). \tag{4}$$

For small h, and for known p we can estimate the primary error term by

$$\alpha \cdot h^p \approx \frac{A_h - A_{2h}}{2^p - 1}. \tag{5}$$

By utilizing the same trick as previously it is possible to find

$$A_{4h} - A_{2h}. \tag{6}$$

Hence, we can form the Richardson fraction to be

$$F_h = \frac{A_{2h} - A_{4h}}{A_h - A_{2h}} \tag{7}$$

after simplifying. One can also show with calculus that

$$F_h \to 2^p \text{ as } h \to 0_+. \tag{8}$$

This fraction has properties to also converge monotonically to 2^p from below or from above. When analyzing Richardson fraction we can see that if this is not the case but rather that it jumps between values of 2^p, we know that it does not follow what we would expect from an asymptotic error expansion. Mathematically an asymptotic error expansion requires a certain degree of differentiability and if this is sustained by the function we are analyzing, then we know that p, q, r will all be integers. If it is the case that it does not has the required amounts of differentiability necessary by the method then Richardson fraction will still converge to 2^p except that $p \notin \mathbb{N}$ but rather $p \in (0, \infty)$.

It might be interesting to know exactly how rapidly Richardsons fraction converges to 2^p which we can find by the following calculations.

$$2^p - F_h = \mathcal{O}(h^{q-p}) \implies \tag{9}$$

$$|2^p - F_h| \leq C \cdot h^{q-p}. \tag{10}$$

As we divide the step size by half forming A_{4h}, A_{2h}, etc, we know that $h = h_0 \cdot 2^{-k}$.

$$\begin{aligned} log_2(|2^p - F_h|) &\leq log_2(C) + (q-p)log_2(h) \\ &= log_2(C) + (q-p)log_2(h_0 \cdot 2^{-k}) \\ &= log_2(C) + (q-p)log_2(h_0) - (q-p) \cdot k \\ &= C_2 - (q-p) \cdot k \end{aligned} \tag{11}$$

where $C_2 = log_2(C) + (q-p)log_2(h_0)$. We notice that Eq. (11) is a straight line with slope $m = -(q-p)$ and hence with known p we can solve for q by looking at the slope of the line of $log_2(|2^p - F_h|)$.

To conclude with the relevant information necessary for doing analysis of accurate estimates with Richardson fraction we note that

1. Richardsons fraction converges to 2^p with p being an integer if the function is sufficiently smooth required by A.E.E.
2. Richardsons fraction converges monotonically to 2^p.
3. The fraction converges with rate h^{q-p} towards 2^p.

It is important to keep in mind that these conclusions are only true when h is sufficiently small and hence will usually not follow these traits in the early calculations. If one observes that it follows each of these criteria for several points then one has showed that it follows each of the traits characterized by having an asymptotic error expansion. This means that we can see when it stops behaving according to what we mathematically expect it to and therefore infer when rounding errors made by the machine has disrupted the calculations. This is of importance because then we have simple yet effective ways of knowing when our calculations and error estimates can be trusted.

3 Application of A.E.E and Richardson's fraction

1. Complete `MyRichardson.m` *according to the specifications and run minimal working example of this function in* `MyRichardsonMWE`*.*

The function `MyRichardson.m` uses the Richardson's fraction that was explained in the previous chapter in Eq. (7) as well as the approximation of the primary error term by Eq. (5). This is contingent on knowing the correct value of p which can be found by inspecting the behavior of Richardson's fraction, hence if the value of p is incorrect we will get error estimates that has no significance. A minimal working example for this, which finds an approximation of the numerical derivative of $f(x) = e^x$ at $x = 1$ is `MyRichardsonMWE`. The resulting table which displays the behavior of a typical Richardson's fraction is found in Table (1). Notice how Richardson's fraction behavior follows the three parts of relevant information, namely that it converges to 2^p, where $p = 1$. It does so monotonically and the fraction also converges with the same rate. The error is halved every time the step size is halved. This can be seen as the perfect scenario for the Richardson's scenario as everything behaves exactly as we except and also does it in a way that is easily noticeable. It does not always have this behavior at all values of h and specifically it usually does not resemble the expected behavior in the beginning and at the end. This has two reasons, in the beginning the step size is relatively large and it only converges to 2^p when only when h is sufficiently small. Secondly for very h too small we start to notice the effects of working in finite arithmetic which is that rounding errors will start to occur and subtractive cancellation. One way of using Richardson's fraction is therefore to know when we can form error estimates which are reliable, and also then find the optimal step size for where we have information we can trust together with error estimates that are reliable. Lastly one important part here is to notice that it converges to an integer, $p = 1$, which has significance that the function is sufficiently smooth. This is indeed the case as we are looking at $f(x) = e^x$.

2. Develop a script `a3range.m` *which apply Richardon's technique to compute the range of the shot whose physical parameters are given by* `a3f3.m`*.*

- *Use methods 'rk1', 'rk2', 'rk3', 'rk4' to compute the trajectories*
- *Use time steps $h_k = 2^{-k}$ seconds for $k = 0, 1, 2, ...$*

For each of the four methods:

(a) *Determine the power of the primary error term.*

(b) *Determine the power of the secondary error term.*

(c) *Determine when the computed value of Richardson's fractions behaves in a manner consistent with an asymptotic error expansion of the type given by equation* (1).

(d) *Identify the best approximation of the range and explain why the error estimate is reliable.*

The trajectory which we wish to find range of is shown in Fig. (1).

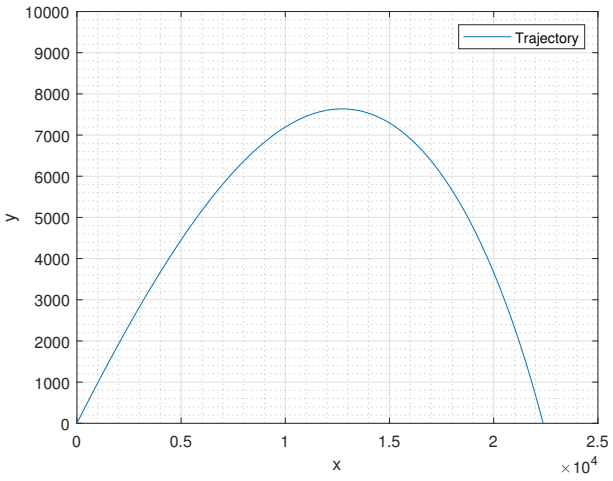

Figure 1 – Plot of the trajectory resembling an artillery shell.

From Table (2) we can see that method **rk1** approaches 2^1, hence $p = 1$. We see that in Fig. (2) that the slope is -1. Using previously shown in Eq. (11) we know that order of the second order term is

$$q - p = 1 \implies q = 2. \tag{12}$$

It is also apparent from the plot as well as the table that for $2 \leq k \leq 15$ it behaves what we deem to very closely resemble Richardson's fraction. There is a bit of wobbliness and it does not resemble the idea of a perfect straight line, but this is close enough as there will always be rounding errors. Therefore the best range we can estimate with a good error estimate is for $k = 15$ which gives

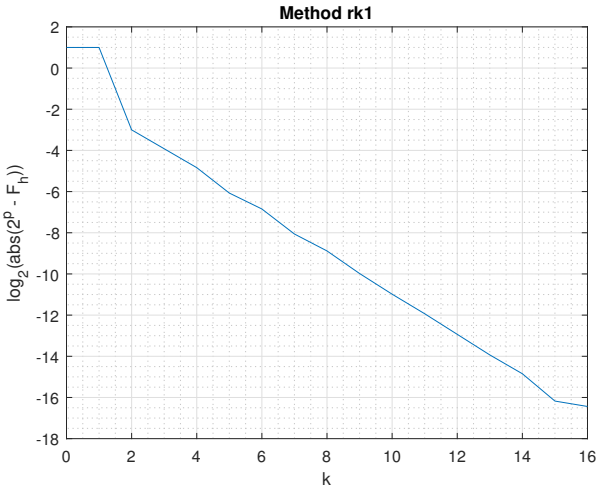

Figure 2 – Log_2 error of Richardon's fraction

$$range = 2.237297728231e + 04, \tag{13}$$
$$error\_est = 6.112062183092e - 03. \tag{14}$$

From Table (3) we can see that method **rk2** approaches 2^2, hence $p = 2$. We see that in Fig. (3) that the slope is -1 for $2 \leq k \leq 10$. Using previously shown in Eq. (11) we know that order of the second order term is

$$q - p = 1 \implies q = 3. \tag{15}$$

The best range we can estimate with a good error estimate is for $k = 10$ which gives

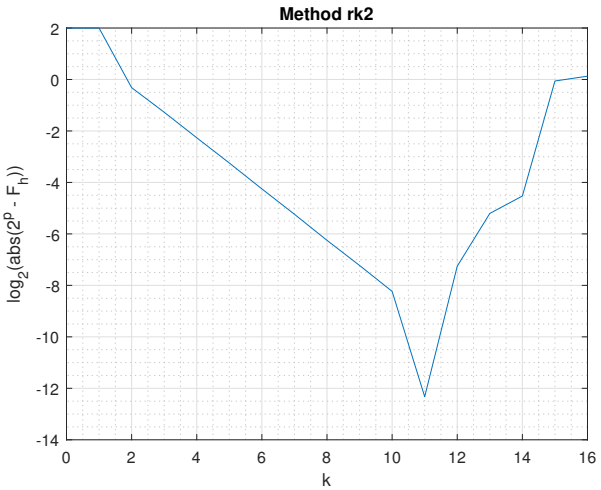

Figure 3 – Log_2 error of Richardon's fraction

$$range = 2.237298339187e + 04, \tag{16}$$
$$error\_est = 2.464193433601e - 06. \tag{17}$$

From Table (4) we can see that method **rk3** approaches 2^3, hence $p = 3$. We see that in Fig. (4) that the slope is about -1 for $2 \leq k \leq 7$. Using previously shown in Eq. (11) we know that order of the second order term is

$$q - p = 1 \implies q = 4. \tag{18}$$

The best range we can estimate with a good error estimate is for $k = 7$ which gives

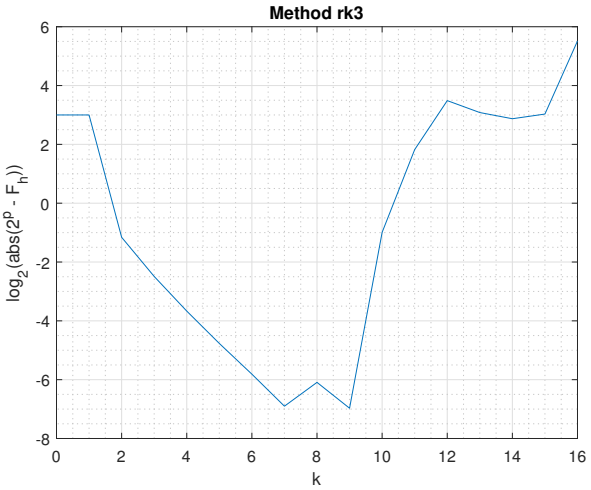

Figure 4 – Log_2 error of Richardon's fraction

$$range = 2.237298339448e + 04, \tag{19}$$
$$error\_est = -1.497511610588e - 07. \tag{20}$$

From Table (5) we can see that method **rk4** approaches 2^4, hence $p = 4$. We see that in Fig. (5) that the slope is about -1 for $2 \leq k \leq 5$. Using previously shown in Eq. (11) we know that order of the second order term is

$$q - p = 1 \implies q = 5. \tag{21}$$

The best range we can estimate with a good error estimate is for $k = 5$ which gives

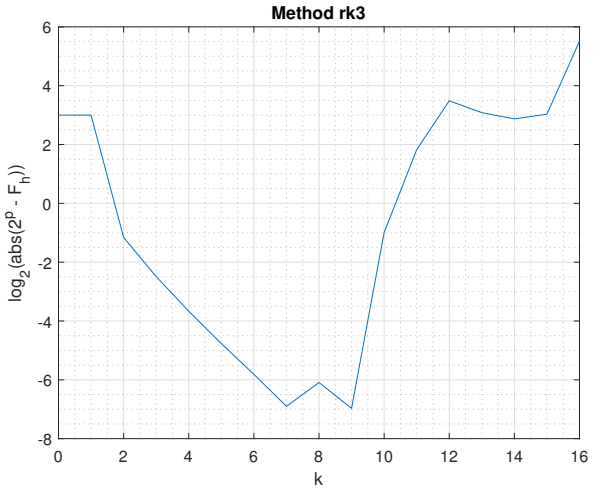

Figure 5 – Log_2 error of Richardon's fraction

$$range = 2.237298339432e + 04, \tag{22}$$
$$error\_est = 1.251998279865e - 08. \tag{23}$$

Notice how the pattern is that for more efficient methods, the faster the convergence and therefore also shorter resemblance of a straight line. It can be difficult for few data points to draw the conclusion that it follows A.E.E, and one solution is to first try a lower order method such as **rk1** to get a clear resemblance of a straight line. Another important idea is that if we were to go to quadruple precision rather than double it would also become clearer as the straight line would follow for a larger interval.

Our conclusion here is that our best estimate is using method **rk4**. This error estimate is accurate because it for these values of k follows an A.E.E, therefore we can estimate the error using Eq. (5) by deciding on the right value of p.

3. Develop a script ***a3time*** *which computes the flight time of the shot given by* ***a3f3.m*** *using your method of choice.*

(a) *You must include an error estimate.*

(b) *You must explain why your error estimate can be trusted.*

(c) *You must discuss the behavior of Richardson's fractions.*

Table. (6) shows the behavior for the Richardson's fraction and here we can see that it follows three important criteria

1. Converges to $2^2 \implies p = 2$

2. Converges monotonically to 2^2 from above

3. The same decay rate shown in Fig. (6)

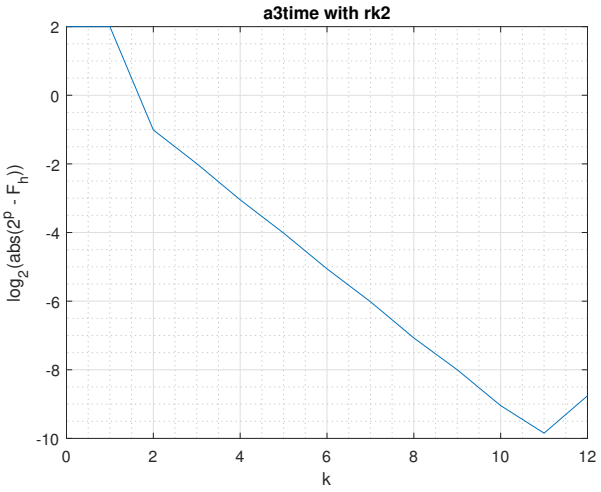

Figure 6 – Log_2 error of Richardon's fraction using method **rk2**.

Since it behaves as what we mathematically expect if it has an A.E.E we can in this case conclude that it does and therefore we can trust our error estimate. The best error estimate we can trust is for $k = 11$ for which we get

$$time\_approx = 7.841593888404e + 01, \tag{24}$$

$$error\_est = 8.920470880488e - 09. \tag{25}$$

4. Develop a script ***a3low.m*** *which computes the low firing solution for a target located at 15000 meters to the right of the gun given by* ***a3f3.m****.*

(a) *You must include an error estimate.*

(b) *You must explain why your error estimate can be trusted.*

(c) *You must discuss the behavior of Richardson's fractions.*

With the use of bisection and a clever residual function depending on the angle theta it is possible to form a Richardson's fraction. It is all primarily dependent on the value of the step size which generates the trajectory in the first place, hence the value of theta is dependent upon the step size. With this it is possible to create a table like we have done previously.

Table. (7) shows the behavior for the Richardson's fraction and here we can see that it follows three important criteria

1. Converges to $2^3 \implies p = 3$

2. Converges monotonically to 2^3 from below

3. The same decay rate shown in Fig. (6) for $2 \leq k \leq 6$

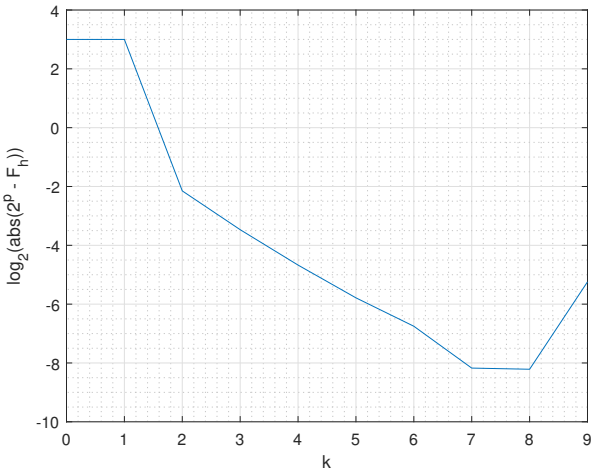

Figure 7 – Log_2 error of Richardon's fraction using method **rk3**.

Since it behaves as what we mathematically expect if it has an A.E.E we can in this case conclude that it does and therefore we can trust our error estimate. The best error estimate we can trust is for $k = 6$ for which we get

$$theta\_approx = 2.751102528731e - 01, \tag{26}$$

$$error\_est = 2.619157272018e - 11. \tag{27}$$

5. Develop a script ***a3range_g7.m*** *which computes the range of the shot defined by* ***a3f4.m****.*

(a) *For which methods do you retain the ability to estimate the range accurately?*

With methods **rk1** and **rk2** we find that there exists a A.E.E and Richardson's fraction behaves the way we expect, however for the higher order methods **rk3** and **rk4** there is no pattern. The reasoning for this is because with the higher order methods require more differentiability which is needed for the A.E.E and the function which generated this specific trajectory is not that many times differentiable. This results in that for these methods there doesn't exist an A.E.E.

6. Develop a script `a3range.m` *sabotage which uses* `range_rkx_sabotage.m` *to compute the range of the shot given by* `a3f3.m`*.*

(a) *For which methods do you retain the ability to estimate the range accurately?*

When the tolerance value has been made less strict as is the case in `range_rkx_sabotage.m` the approximated time for when the shell hits the ground will have a larger error than the original `range_rkx.m`. The range of the shell is dependent on the approximated time thus if the error for the approximated time is large the error for the range will therefore also be large. The major factor of the error in approximating the range will therefore not depend on the step size but rather the major error in finding the time of impact. This will cause the Richardson's fraction not to behave to what we expect, simple because the correlation between the approximated ranges using different time steps does not exist. To conclude, **none of the methods** will retain the ability to estimate the range accurately.

7. The script `a3length.m` *computes the length of the trajectory of the shot given by* `a3f3.m`*.*

(a) *For each method what is the order of the primary error term?*

We see that with method **rk1** Richardson's fraction approaches 2^1, hence $p = 1$. For the higher orders methods **rk2**, **rk3** and **rk4** they all approach 2^2, $p = 2$. This is initially surprising because we normally see that with higher order methods the primary error term will be of higher order. The reason for why this is not the case is that the length of the trajectory is approximated with the trapezoidal rule which we know has $\mathcal{O}(h^2)$ and this will bottleneck the operations with higher order.

4 Conclusions

It is often the case that we wish to numerically approximate derivatives, integrals and also more advanced calculations such as approximating the range of an artillery shell. Mathematically the solution to finding more accurate estimates is simple as we only need to decrease the step size. In practice it is much more difficult as we have finite arithmetic and because of this we need to find other methods for making a more complete analysis for where the approximations and the errors are to be trusted.

In this project we have seen that the Richardson's fraction can be used with a variety of problems and gives us a reliable way to find error estimates. The underlying theory behind A.E.E was not developed in this project but rather how we can confirm that it exists using Richardson's fraction.

5 References

[1] Carl Christian Kjelgaard Mikkelsen: *An Introduction to Scientific Computing*, Department of Computing Science, Umeå University (2018)

6 Appendix

6.1 Task 1 tables

k	Approximation A_h	Fraction F_h	Error estimate E_h
0	2.895480163672	0.00000000	0.000000000000
1	2.805025851403	0.00000000	-9.045431226844e-02
2	2.761200888902	2.06399064	-4.382496250165e-02
3	2.739629445828	2.03161941	-2.157144307422e-02
4	2.728927822736	2.01571695	-1.070162309156e-02
5	2.723597892360	2.00783544	-5.329930376490e-03
6	2.720938129638	2.00391198	-2.659762721350e-03
7	2.719609546672	2.00195456	-1.328582965698e-03
8	2.718945579511	2.00097692	-6.639671619268e-04
9	2.718613676976	2.00048837	-3.319025345263e-04
10	2.718447745963	2.00024413	-1.659310128161e-04
11	2.718364785520	2.00012206	-8.296044325107e-05
12	2.718323306559	2.00006078	-4.147896106588e-05
13	2.718302567373	2.00002842	-2.073918585666e-05
14	2.718292197853	2.00001403	-1.036952016875e-05
15	2.718287013122	2.00001123	-5.184730980545e-06
16	2.718284420669	1.99993264	-2.592452801764e-06
17	2.718283124268	1.99973060	-1.296401023865e-06
18	2.718282476068	2.00000000	-6.482005119324e-07
19	2.718282151967	2.00000000	-3.241002559662e-07

Table 1 – Richardson's fraction for `MyRichardsonMWE` which finds numerical approximation for $f'(x) = e^x$ at point $x = 1$.

6.2 Task 2 tables

k	Approximation A_h	Fraction F_h	Error estimate E_h
0	2.215475800698e+04	0.00000000	0.000000000000e+00
1	2.226834769726e+04	0.00000000	1.135896902761e+02
2	2.232180737277e+04	2.12477328	5.345967551322e+01
3	2.234768231668e+04	2.06607890	2.587494390464e+01
4	2.236039746277e+04	2.03497024	1.271514609780e+01
5	2.236670811603e+04	2.01487003	6.310653262091e+00
6	2.236984977888e+04	2.00869844	3.141662846803e+00
7	2.237141767501e+04	2.00374425	1.567896123950e+00
8	2.237220079351e+04	2.00211861	7.831185015530e-01
9	2.237259215891e+04	2.00099062	3.913654031312e-01
10	2.237278779330e+04	2.00049388	1.956343916754e-01
11	2.237288559799e+04	2.00025573	9.780469011821e-02
12	2.237293449721e+04	2.00012783	4.889921965878e-02
13	2.237295894604e+04	2.00006392	2.444882848795e-02
14	2.237297117025e+04	2.00003391	1.222420699560e-02
15	2.237297728231e+04	2.00001352	6.112062183092e-03
16	2.237298033832e+04	2.00001127	3.056013869355e-03

Table 2 – Richardson's fraction for a3range using method **rk1**

k	Approximation A_h	Fraction F_h	Error estimate E_h
0	2.236937113617e+04	0.00000000	0.000000000000e+00
1	2.237221489000e+04	0.00000000	9.479179419698e-01
2	2.237280705137e+04	4.80232914	1.973871250727e-01
3	2.237294120066e+04	4.41419684	4.471643019982e-02
4	2.237297307937e+04	4.20811609	1.062623493393e-02
5	2.237298084435e+04	4.10544520	2.588327064586e-03
6	2.237298276044e+04	4.05251784	6.386960327897e-04
7	2.237298323630e+04	4.02651976	1.586223515915e-04
8	2.237298335488e+04	4.01313863	3.952575934818e-05
9	2.237298338448e+04	4.00668004	9.864965250017e-06
10	2.237298339187e+04	4.00332422	2.464193433601e-06
11	2.237298339372e+04	3.99980513	6.160783717254e-07
12	2.237298339418e+04	4.00655352	1.537676628989e-07
13	2.237298339429e+04	4.02712230	3.818301289963e-08
14	2.237298339432e+04	3.95664740	9.650345115612e-09
15	2.237298339433e+04	3.03972498	3.174742838989e-09
16	2.237298339433e+04	5.09338521	6.233070356150e-10

Table 3 – Richardson's fraction for a3range using method **rk2**

k	Approximation A_h	Fraction F_h	Error estimate E_h
0	2.237327039413e+04	0.00000000	0.000000000000e+00
1	2.237302121565e+04	0.00000000	-3.559692690968e-02
2	2.237298822139e+04	7.55217806	-4.713464993464e-03
3	2.237298400328e+04	7.82203872	-6.025877862287e-04
4	2.237298347078e+04	7.92134046	-7.607144133155e-05
5	2.237298340391e+04	7.96329830	-9.552755467926e-06
6	2.237298339553e+04	7.98220289	-1.196756784339e-06
7	2.237298339448e+04	7.99163610	-1.497511610588e-07
8	2.237298339435e+04	7.98533976	-1.875326103930e-08
9	2.237298339433e+04	7.99202658	-2.346496330574e-09
10	2.237298339433e+04	8.50282486	-2.759666780808e-10
11	2.237298339433e+04	4.46218487	-6.184563972056e-11
12	2.237298339433e+04	-3.21621622	1.922931655177e-11

Table 4 – Richardson's fraction for a3range using method **rk3**

k	Approximation A_h	Fraction F_h	Error estimate E_h
0	2.237296894265e+04	0.00000000	0.000000000000e+00
1	2.237298253527e+04	0.00000000	9.061748710034e-04
2	2.237298334202e+04	16.84861431	5.378334705407e-05
3	2.237298339111e+04	16.43565957	3.272357086341e-06
4	2.237298339413e+04	16.22025681	2.017450848750e-07
5	2.237298339432e+04	16.11384681	1.251998279865e-08
6	2.237298339433e+04	16.01675458	7.816803796838e-10
7	2.237298339433e+04	18.10674157	4.317068184415e-11

Table 5 – Richardson's fraction for a3range using method **rk4**

6.3 Task 3 tables

k	Approximation A_h	Fraction F_h	Error estimate E_h
0	7.840444726979e+01	0.00000000	0.000000000000e+00
1	7.841334458859e+01	0.00000000	2.965772931357e-03
2	7.841532345978e+01	4.49615863	6.596237306222e-04
3	7.841578893805e+01	4.25126439	1.551594231444e-04
4	7.841590188824e+01	4.12109312	3.765006481634e-05
5	7.841592969611e+01	4.06180657	9.269290449273e-06
6	7.841593659629e+01	4.03002319	2.300058833763e-06
7	7.841593831469e+01	4.01545956	5.728008953080e-07
8	7.841593874350e+01	4.00743369	1.429345909780e-07
9	7.841593885059e+01	4.00390580	3.569878970211e-08
10	7.841593887736e+01	4.00189521	8.920470880488e-09
11	7.841593888404e+01	4.00108570	2.229512574559e-09
12	7.841593888572e+01	3.99768971	5.577002563465e-10

Table 6 – Richardson's fraction for a3rtime using method **rk2**

6.4 Task 4 tables

k	Approximation A_h	Fraction F_h	Error estimate E_h
0	2.751036855308e-01	0.00000000	0.000000000000e+00
1	2.751094101286e-01	0.00000000	8.177996903637e-07
2	2.751101464406e-01	7.77469108	1.051874193065e-07
3	2.751102395311e-01	7.90963482	1.329864421759e-08
4	2.751102512247e-01	7.96076979	1.670522395017e-09
5	2.751102526898e-01	7.98187485	2.092894746397e-10
6	2.751102528731e-01	7.99071812	2.619157272018e-11
7	2.751102528960e-01	7.99653775	3.275364106920e-12
8	2.751102528989e-01	7.99663117	4.095929944421e-13
9	2.751102528993e-01	8.02641803	5.103060831045e-14

Table 7 – Richardson's fraction for a3low using method **rk3**

6.5 Appendix Code - MyRichardson.m

Source code for `MyRichardson.m`.

```
function data=MyRichardson(a,p,t)

% MyRichardson Computational kernel for Richardson's technique
%
% Does Richardson extrapolation for a set of values assuming that the
% user has determined the order of the primary error term correctly
%
% CALL SEQUENCE: data=richardson(val,p);
%
% INPUT:
%   a       array of m approximations of t, such that if a(i) corresponds
%           stepsize h, then a(i+1) corresponds to stepsize h/2
%   p       the order of the primary order term
%   t       (optional) the target value of the approximations
%
% OUTPUT:
%   data   an array of information such that
%            data(i,1) = i
%            data(i,2) = a(i)
%            data(i,3) = Richardson's fraction for i > 2
%            data(i,4) = Richardson's error estimate for i > 1
```

```
%          if the exact target value is supplied, then
%            data(i,5) = exact error
%            data(i,6) = comparision of error estimate to exact error
%
% MINIMAL WORKING EXAMPLE: A3F2

% PROGRAMMING by Carl Christian K. Mikkelsen (spock@cs.umu.se)
%   2015-12-10  Initial programming amd testing
%   2018-12-09  Printing moved to minimal working example
%   2018-12-09  Skeleton extracted from working code

% PROGRAMMING by Aladdin Persson (alhi0008@student.umu.se)
%   2018-12-19 Initial programming
%   2019-01-08 Minor changes

% Reshape the input array as a colum vector
m=numel(a); a=reshape(a,m,1);

% Is the target value known?
if ~exist('target','var')
    % Set a flag to indicate that the target value is unknown
    flag=0;
    % Allocate space for the table used to print the results
    data=zeros(m,4);
else
    % Set a flag to indicate that the the target value is known
    flag=1;
    % Allocate space for the table used to print the results
    data=zeros(m,6);
end

% Initialize the first and the second columns of data
for i=1:m
   data(i,1)=i-1;
   data(i,2)=a(i);
end

% Process the data, computing Richardson's fractions
for i=3:m
    data(i,3)=(data(i-1,2)-data(i-2,2))/(data(i,2)-data(i-1,2));
```

```
end

% Compute Richardson's error estimates assuming order p is correct!
for i=2:m
   data(i,4) = (data(i,2)-data(i-1,2))/(2^p - 1);
end

% If possible, then compute the error and compare it to the error estimate
if (flag==1)
    for i=1:m
        % Compute the exact error
        data(i,5)=t-data(i,2);
        % Compare the error estimate to the true error
        % i.e. log10(abs(relative error))
        data(i,6)=log2(abs((data(i,5)-data(i,4))./data(i,5)));
    end
end
```

6.6 Appendix Code - MyRichardsonMWE.m

Source code for `MyRichardsonMWE.m`, or in specification a3f2.

```
% MyRichardsonMWE Minimal working example for MyRichardson

% PROGRAMMING by Aladdin Persson (alhi0008@student.umu.se)
%   2018-12-19 Initial programming
%   2019-01-08 Minor changes

clear all; close all; clc;

% Set trial function
f=@(x)exp(x);

% Set point where we want to estimate the derivative
x=1;

% Define the finite difference approximation
D1=@(g,x,h)(g(x+h)-g(x))./h;

% Set the theoretical order of the method
p=1;
```

```
% Set the basic stepsize
h0=0.125;

% Set the number of times we will reduce the stepsize by a factor of 2
kmax=21;

% Define the real derivative
df=@(x)exp(x);

% Allocate space for approximations
a=zeros(kmax,1);

% Construct the different approximations
h=h0;
for i=1:kmax
    % Compute approxmation
    a(i)=D1(f,x,h);
    % Reduce step size
    h=h/2;
end

% Compute target value
t=df(x);

% Apply Richardson's techniques
data=MyRichardson(a,p,t);

% Display results
rdifprint(data,p);
```

6.7 Appendix Code - a3range.m

Source code for `a3range.m`

```
% This script uses range_rkx.m and MyRichardson.m using different
% runge_kutta and euler methods to calculate the range of a simulated
% shell. It prints the table for richardson and then also prints
% error_quotients simply to make looking for ratio changes in the error
% more easily visible to the user.
```

```
% PROGRAMMING by Aladdin Persson (alhi0008@student.umu.se)
%   2018-12-19 Initial programming
%   2019-01-08 Minor changes

clear all; close all; clc;

% run a3f3 which has param, theta, v0, etc needed for our shell simulation
a3f3

% kmax set to only 8, to be able to run relatively fast. This was increased
% to much higher when wanted to get the most accurate result stated in
% report.

kmax=8;

a=zeros(kmax,1);

m=["rk1","rk2","rk3","rk4"];
P = [1,2,3,4];

for rk = 1:4
    method = m(rk);
    dt=1; maxstep=1e7;

    for i=1:kmax
        % Compute approxmation
        [r, flag, t, tra] = range_rkx(param,v0,theta,method,dt,maxstep);
        a(i)=r;
        % Reduce step size
        dt=dt/2;
    end

    p = P(rk);

    % Apply Richardson's techniques
    data=MyRichardson(a,p);

    % Display results
    figure(rk)
    rdifprint(data,p)
```

```
    title(strcat('Method rk',num2str(rk)))

    a = abs((data(3:end-1,3)) - 2^p);
    b = abs((data(4:end,3)) - 2^p);

    % display how error of Richardon's fraction changes, simply for easier
    % visible inspection of the error decay rate
    error_quotient = a ./ b
end

figure(rk+1)
% Plot the trajectory of the shell.
plot(tra(1,:),tra(2,:));

% Turn of the major grid lines and set the axis
grid ON; axis([0 25000 0 10000]); grid MINOR;

xlabel('x'); ylabel('y'); legend('Trajectory')
print('a3rangetrajectory','-depsc2');
```

6.8 Appendix Code - a3time.m

Source code for `a3time.m`.

```
% a3time.m computes flight time using MyRichardson, range_rkx and the shell
% given by a3f3.m.

% PROGRAMMING by Aladdin Persson (alhi0008@student.umu.se)
%   2018-12-19 Initial programming
%   2019-01-08 Minor changes

clear all; close all; clc;

% run a3f3 which has param, theta, v0, etc needed for our shell simulation
a3f3

method='rk2';

% kmax chosen = 13 because this is exact time when method rk2 stops being
% reliable (we can't trust error estimates anymore)
```

```
kmax=13;

a=zeros(kmax,1);
dt=1; maxstep=1e7;

for i=1:kmax
    % Compute approxmation
    [r, flag, t, tra] = range_rkx(param,v0,theta,method,dt,maxstep);
    a(i)=t(end);
    % Reduce step size
    dt=dt/2;
end


% Necessary commands for plotting the trajectory of the shell below. Not
% necessary for computations but can be good for intuition about
% simulation.

% plot(tra(1,:),tra(2,:));
% % Turn of the major grid lines and set the axis
% grid ON; axis([0 25000 0 10000]); grid MINOR;
% title('Trajectory of shell')
% xlabel('range x'); ylabel('height y')

% Observed order for MyRichardson, p=2 since it converges to 2^2
p=2;

% Apply Richardson's techniques
data=MyRichardson(a,p);

% Display results
rdifprint(data,p)
title('a3time with rk2')

% Dummy variables for error quotient calculations. Not necessary either but
% help (at least me) in seeing error decay rate.
a = abs((data(3:end-1,3)) - 2^p);
b = abs((data(4:end,3)) - 2^p);
error_quotient = a./b
```

6.9 Appendix Code - a3low.m

Source code for `a3low.m`.

```
% a3low.m computes the low firing solution (i.e, the angle) to hit a target
% at 15000 meters.

% PROGRAMMING by Aladdin Persson (alhi0008@student.umu.se)
%   2018-12-19 Initial programming
%   2019-01-08 Minor changes

clear all; close all; clc;
tic

% run a3f3 which has param, theta, v0, etc needed for our shell simulation
a3f3

%choose method
method='rk3';

%stepsize and maximum number of steps for range_rkx
dt=1; maxstep=1e7;

% maxit for bisection
maxit=1e7;

maxk=10;
a=zeros(maxk,1);

for k = 1:maxk

    % Define the range function, let everything be constant except we choose
    % the elevation of fire
    range=@(theta)range_rkx(param,v0,theta,method,dt,maxit);

    % Distance to target from gun
    d=15000;

    % residual function, we want res=0 for perfect hit
    res=@(theta)range(theta)-d;

    % Specify elevation in degress and convert to radian
```

```
    deg=0:10:90; theta=deg*pi/180;

    [table, flag]=compute_range(param,v0,theta,method,dt,maxstep,0);


    % Will be two angles which hit target at 15000 meters, this is why we
    % use the min() function in finding alow and blow
    idx=find(table(2,:)>d);

    % goes too short
    alow=theta(min(idx)-1);
    % goes too far
    blow=theta(min(idx));

    eps=10^(-10); delta=0;

    %theta for low trajectory
    [thetalow, lflag]=bisection(res,alow,blow,res(alow),res(blow),delta,eps,maxit,

    a(k)=thetalow;
    dt=dt/2;

end

% guess for p
p = 3;

% Apply Richardson's techniques
data=MyRichardson(a,p);

% Display results
rdifprint(data,p)
```

6.10 Appendix Code - a3range_g7.m

Source code for `a3range_g7.m`.

```
% a3low.m computes the low firing solution (i.e, the angle) to hit a target
% at 15000 meters.

% PROGRAMMING by Aladdin Persson (alhi0008@student.umu.se)
```

```
%   2018-12-19 Initial programming
%   2019-01-08 Minor changes

clear all; close all; clc;

% run a3f4 which has param, theta, v0, etc needed for our shell simulation
a3f4

kmax=10;
a=zeros(kmax,1);
dt=1/4; maxstep=1e6;

m=["rk1","rk2","rk3","rk4"];
P = [1,2,3,4];

% Loop through each of the methods in m and perform the calculations using
% each one.

for rk = 1:4
    method = m(rk);
    dt=1; maxstep=1e7;

    for i=1:kmax
        % Compute approxmation
        [r, flag, t, tra] = range_rkx(param,v0,theta,method,dt,maxstep);
        a(i)=r;
        % Reduce step size
        dt=dt/2;
    end

    p = P(rk);

    % Apply Richardson's techniques
    data=MyRichardson(a,p);

    % Display results
    figure(rk)
    rdifprint(data,p)

    title(strcat('Method rk',num2str(rk)))
```

```
    a = abs((data(3:end-1,3)) - 2^p);
    b = abs((data(4:end,3)) - 2^p);

    % display how error of Richardon's fraction changes, simply for easier
    % visible inspection of the error decay rate
    error_quotient = a ./ b
end
```

6.11 Appendix Code - a3range_sabotage.m

Source code for `a3range_sabotage.m`.

```
% This script uses range_rkx_sabotage.m and MyRichardson.m using different
% runge_kutta and euler methods to calculate the range of a simulated
% shell. It prints the table for richardson and then also prints
% error_quotients simply to make looking for ratio changes in the error
% more easily visible to the user.

% PROGRAMMING by Aladdin Persson (alhi0008@student.umu.se)
%   2018-12-19 Initial programming
%   2019-01-08 Minor changes


clear all; close all; clc;

% load model a3f3 which has param, theta, v0, etc needed for our shell
% simulation
a3f3

kmax=8;
a=zeros(kmax,1);

m=["rk1","rk2","rk3","rk4"];
P = [1,2,3,4];

% Loop through each of the methods in m and perform the calculations for
% range using each method.
for rk = 1:4
    method = m(rk);
    dt=1; maxstep=1e7;
```

```
    for i=1:kmax
        % Compute approxmation
        [r, flag, t, tra] = range_rkx_sabotage(param,v0,theta,method,dt,maxstep);
        a(i)=r;
        % Reduce step size
        dt=dt/2;
    end

    p = P(rk);

    % Apply Richardson's techniques
    data=MyRichardson(a,p);

    % Display results
    figure(rk)
    rdifprint(data,p)

    title(strcat('Method rk',num2str(rk)))

    a = abs((data(3:end-1,3)) - 2^p);
    b = abs((data(4:end,3)) - 2^p);

    % display how error of Richardon's fraction changes, simply for easier
    % visible inspection of the error decay rate
    error_quotient = a ./ b
end
```

6.12 Appendix Code - a3length.m

Source code for `a3range_sabotage.m`.

```
% a3length.m computes the length of the trajectory

% PROGRAMMING by Aladdin Persson (alhi0008@student.umu.se)
%   2018-12-19 Initial programming
%   2019-01-08 Minor changes

clear all; close all; clc

% Load parameters describing shot
```

```
a3f3

% Set initial time step
h0=1;

% Methods
m=["rk1","rk2","rk3","rk4"];

% Number of rows in table
kmax=10;

% Define the function needed for arc lenght
g=@(z)sqrt(z(3,:).^2+z(4,:).^2);

% Loop over methods
for i=1:4
    % Select method
    method=m(i);

    % Initialize time step
    dt=h0;

    % Initialize maxstep
    maxstep=200;

    % Loop over approximations
    for k=1:kmax
      % Compute range
      [r, flag, t, tra]=range_rkx(param,v0,theta,method,dt,maxstep);
      % Save information
      a(k)=a3int(g,t,tra);
      % Decrease time step
      dt=dt/2;
      % Increase maxstep
      maxstep=maxstep*2;
    end

    % Run Richardsons techniques
    data=MyRichardson(a,2);
```

```
    % New figure
    h(i)=figure();

    % Print to screen
    rdifprint(data,i);
end
```

6.13 Appendix Code - a3f3.m

Source code for `a3f3.m`.

```
% A3F3  Physical parameters for firing a projectile

% Set parameters.
param.mass=10;
param.cali=0.088;

% Constant drag coefficient
param.drag=@(x)0.1873;

param.atmo=@(x)atmosisa(x);
param.grav=@(x)9.82;

% Select muzzle velocity and elevation
v0=780; theta=45*pi/180;
```

6.14 Appendix Code - a3f4.m

Source code for `a3f4.m`.

```
% A3F4  Physical parameters for firing a projectile

% Load standard shell models
load shells

% Set parameters.
param.mass=10;
param.cali=0.088;

% This drag coefficient is not constant
param.drag=@(x)mcg7(x);
```

```
param.atmo=@(x)atmosisa(x);
param.grav=@(x)9.82;

% Select muzzle velocity and elevation
v0=780; theta=45*pi/180;
```

6.15 Appendix Code - a3int.m

Source code for `a3int.m`.

```
function s=a3int(g,t,tra)

% a3int  Integrates a function of the trajectory using trapezoidal rule.
%
% CALL SEQUENCE: s=a3int(g,t,tra)
%
% INPUT:
%   g    a function of such that g(tra) is defined
%   t    the time instances corresponding to tra
%   tra  the trajectory in question
%
% OUTPUT:
%   s    an approximation of the the integral from t(1) to t(end)
%        of g(gamma(t)) where t -> gamma(t) is the true trajectory
%
% MINIMAL WORKING EXAMPLE: TODO

% PROGRAMMING by Carl Christian Kjelgaard Mikkelsen (spock@cs,umu.se)
%   2016-12-04 Initial programming and testing

% Determine the number of points on the trajectory
n=size(tra,2);

% Apply g to every point of the trajectory
aux=g(tra);

% Initialize
s=0;

% Loop over the points, trapezoidal style
```

```
for i=1:n-1
    % Compute time step
    dt=t(i+1)-t(i);
    % Add the contribution from the ith interva
    s=s+0.5*dt*(aux(i)+aux(i+1));
end
```

6.16 Appendix Code - range_rkx.m

Source code for `range_rkx.m`.

```
function [r, flag, t, tra]=range_rkx(param,v0,theta,method,dt,maxstep)

% RANGE_RKX Computes the range of a shell using a Runge-Kutta method
%
% All time steps have the same size, except the last which is adjusted to
% put the shell exactly on the ground.
%
% CALL SEQUENCE: [r, flag, t, tra]=range_rkx(param,v0,theta,method,dt,maxstep)
%
% INPUT:
%   param    a structure describing of the environment and the shell
%                param.mass   the mass of the shell
%                param.cali   the caliber of the shell
%                param.drag   a function computing the drag coeffient
%                param.atmo   a function computing the atmosphere
%                param.grav   a function computing gravity
%                param.wind   a function computing the wind
%   v0       the muzzle velocity of the shell
%   theta    the elevation of the gun in radians
%   method   a string describing the method, see "help RK" for options
%   dt       the standard time step, the last step will be shorter
%   maxstep  the maximum number of time steps allowed, a safety valve.
%
% OUTPUT:
%   r        the computed range if flag=1;
%   t        the time instances where the trajectory was approximated
%   tra      the computed trajectory, y(:,i) corresponds to time t(i)
%               tra(1,i) is the x-component of the shells position
```

```
%               tra(2,i) is the y-component of the shells position
%               tra(3,i) is the x-component of the shells velocity
%               tra(4,i) is the y-component of the shells velocity
%    flag    flag=0 if the shell did not hit the ground
%            flag=1 if the shell hit the ground
%
% MNIMAL WORKING EXAMPLE: range_rkx_mwe1
%
% See also: COMPUTE_RANGE, COMPUTE_ELEVATION, FIRE, RANGE_RK1, TARGET

% PROGRAMMING by Carl Christian Kjelgaard Mikkelsen (spock@cs.umu.se)
%  Fall 2014   Initial programming and testing
%  2015-09-22  Globals m, k, and g integrated into structure CONST
%  2015-10-31  Replaced structure CONST with mandatory PARAM
%  2015-10-31  Minor error in the inline comments fixed during yearly review
%  2015-11-01  Extended the description of the initial condition
%  2015-12-08  Added support for wind to shell4.m
%  2016-06-22  Adapted routine to improved bisection method
%  2016-06-23  Added logical check for bad elevations
%  2016-09-09  Adapted to more flexible SHELL4A

% TODO: Prior to HT-2016
%    a) Add support for wind to shell8.m and target.m
%    b) Add shell data to the simulation using interpolation from tables
%    c) Investigate if a military grade 6 DOF model is feasible

%////////////////////////////////////////////////////////////////////////
% Select the relative tolerance being use by the nonlinear solver
%////////////////////////////////////////////////////////////////////////

% The default value of tol is the double precision unit round off.
%   ... change this value at your own peril
%   ... or if you are told to do so
%   ... or if you are feeling adventurous
tol=2^-53;

% Select a shell model, feeding it the parameters of the simulation
shell=@(t,x)shell4a(param,t,x);

% Define the initial condition; must be compatible with the simple shell model
```

```
% There are four coordinates:
%  1st coor. is the x coordinate of the muzzle of the gun
%  2nd coor. is the y coordinate of the muzzle of the gun
%  3rd coor. is the x coordinate of the velocity of the shell when it exits
%  4th coor. is the y coordinate of the velocity of the shell when it exits
tra0=[0; 0; v0*cos(theta); v0*sin(theta)];

% Allocate space for trajectory
tra=zeros(4,maxstep+1);

% Initialize the trajectory
tra(:,1)=tra0;

% Allocate space to record the time instances
t=zeros(1,maxstep+1);

% Anticipate failure or bad input.
r=NaN; flag=0;

% Check for bad elevation
if (sin(theta)<=0)
    % The shell is fired into the ground
    r=0; flag=1; t=0; tra=tra(:,1);
    % Quick return
    return;
end

% Pickup the method to use.
switch lower(method)
    case 'rk1'
        phi=@phi1;
    case 'rk2'
        phi=@phi2;
    case'rk3'
        phi=@phi3;
    case 'rk4'
        phi=@phi4;
    otherwise
        fprintf('Invalid method specified! Aborting\n'); return;
end
```

```
% Loop over the time steps
for it=1:maxstep
    % Advance the clock a single time step
    t(it+1)=it*dt;

    % Advance the solution a single step using the selected method.
    tra(:,it+1)=phi(shell,t(it),tra(:,it),dt);

    % Test to see if we are below ground level, tra(2,it+1)<0
    if (tra(2,it+1)<0)
        %-----------------------------------------------------------------
        % We passed through the ground! Go back and compute the time step
        % which will put the shell exactly on the ground.
        %-----------------------------------------------------------------

        % Isolate the last point above ground level.
        z0=tra(:,it); t0=t(it);

        % Define the function psi(x) which isolates the y coordinate
        % of the shell after a step of size x*dt.
        psi=@(x)[0 1 0 0]*phi(shell,t0,z0,x*dt);

        % Find the 'exact' timestep which will put the shell on the ground
        rho=bisection(psi,0,1,tra(2,it),tra(2,it+1),tol*t(it+1),tol*tra(1,it),60);

        % Calculate the 'exact' point of impact;
        aux=phi(shell,t0,z0,rho*dt);

        % Save the time and point of impact
        t(it+1)=t0+rho*dt; tra(:,it+1)=aux;

        % The range has now been computed, signal succes ...
        flag=1;

        % ... and break from the for-loop.
        break;
    end
end
```

```
% Remove any trailing zeros from output arrays
tra=tra(:,1:it+1); t=t(1,1:it+1);

% Only define the range if the shell hit the ground !!!
if flag==1
  % Isolate the range
    r=tra(1,it+1);
end
```

6.17 Appendix Code - range_rkx_sabotage.m

Source code for `range_rkx_sabotage.m`.

```
function [r, flag, t, tra]=range_rkx_sabotage(param,v0,theta,method,dt,maxstep)

% RANGE_RKX_SABOTAGE  Sloppy computation of final stepsize
%
% All time steps have the same size, except the last which is adjusted to
% put the shell exactly on the ground.
%
% CALL SEQUENCE: [r, flag, t, tra]=range_rkx(param,v0,theta,method,dt,maxstep)
%
% INPUT:
%   param    a structure describing of the environment and the shell
%                param.mass   the mass of the shell
%                param.cali   the caliber of the shell
%                param.drag   a function computing the drag coeffient
%                param.atmo   a function computing the atmosphere
%                param.grav   a function computing gravity
%                param.wind   a function computing the wind
%   v0       the muzzle velocity of the shell
%   theta    the elevation of the gun in radians
%   method   a string describing the method, see "help RK" for options
%   dt       the standard time step, the last step will be shorter
%   maxstep  the maximum number of time steps allowed, a safety valve.
%
% OUTPUT:
%   r        the computed range if flag=1;
%   t        the time instances where the trajectory was approximated
%   tra      the computed trajectory, y(:,i) corresponds to time t(i)
%               tra(1,i) is the x-component of the shells position
```

```
%               tra(2,i) is the y-component of the shells position
%               tra(3,i) is the x-component of the shells velocity
%               tra(4,i) is the y-component of the shells velocity
%    flag    flag=0 if the shell did not hit the ground
%            flag=1 if the shell hit the ground
%
% MNIMAL WORKING EXAMPLE: range_rkx_mwe1
%
% See also: COMPUTE_RANGE, COMPUTE_ELEVATION, FIRE, RANGE_RK1, TARGET

% PROGRAMMING by Carl Christian Kjelgaard Mikkelsen (spock@cs.umu.se)
%  Fall 2014   Initial programming and testing
%  2015-09-22  Globals m, k, and g integrated into structure CONST
%  2015-10-31  Replaced structure CONST with mandatory PARAM
%  2015-10-31  Minor error in the inline comments fixed during yearly review
%  2015-11-01  Extended the description of the initial condition
%  2015-12-08  Added support for wind to shell4.m
%  2016-06-22  Adapted routine to improved bisection method
%  2016-06-23  Added logical check for bad elevations
%  2016-09-09  Adapted to more flexible SHELL4A

% TODO: Prior to HT-2016
%    a) Add support for wind to shell8.m and target.m
%    b) Add shell data to the simulation using interpolation from tables
%    c) Investigate if a military grade 6 DOF model is feasible

%////////////////////////////////////////////////////////////////////////
% Select the relative tolerance being use by the nonlinear solver
%////////////////////////////////////////////////////////////////////////

% The default value of tol is the double precision unit round off.
%   ... change this value at your own peril
%   ... or if you are told to do so
%   ... or if you are feeling adventurous
tol=0.125;

% Select a shell model, feeding it the parameters of the simulation
shell=@(t,x)shell4a(param,t,x);

% Define the initial condition; must be compatible with the simple shell model
```

```
% There are four coordinates:
%  1st coor. is the x coordinate of the muzzle of the gun
%  2nd coor. is the y coordinate of the muzzle of the gun
%  3rd coor. is the x coordinate of the velocity of the shell when it exits
%  4th coor. is the y coordinate of the velocity of the shell when it exits
tra0=[0; 0; v0*cos(theta); v0*sin(theta)];

% Allocate space for trajectory
tra=zeros(4,maxstep+1);

% Initialize the trajectory
tra(:,1)=tra0;

% Allocate space to record the time instances
t=zeros(1,maxstep+1);

% Anticipate failure or bad input.
r=NaN; flag=0;

% Check for bad elevation
if (sin(theta)<=0)
    % The shell is fired into the ground
    r=0; flag=1; t=0; tra=tra(:,1);
    % Quick return
    return;
end

% Pickup the method to use.
switch lower(method)
    case 'rk1'
        phi=@phi1;
    case 'rk2'
        phi=@phi2;
    case'rk3'
        phi=@phi3;
    case 'rk4'
        phi=@phi4;
    otherwise
        fprintf('Invalid method specified! Aborting\n'); return;
end
```

```
% Loop over the time steps
for it=1:maxstep
    % Advance the clock a single time step
    t(it+1)=it*dt;

    % Advance the solution a single step using the selected method.
    tra(:,it+1)=phi(shell,t(it),tra(:,it),dt);

    % Test to see if we are below ground level, tra(2,it+1)<0
    if (tra(2,it+1)<0)
        %-----------------------------------------------------------------
        % We passed through the ground! Go back and compute the time step
        % which will put the shell exactly on the ground.
        %-----------------------------------------------------------------

        % Isolate the last point above ground level.
        z0=tra(:,it); t0=t(it);

        % Define the function psi(x) which isolates the y coordinate
        % of the shell after a step of size x*dt.
        psi=@(x)[0 1 0 0]*phi(shell,t0,z0,x*dt);

        % Find the 'exact' timestep which will put the shell on the ground
        rho=bisection(psi,0,1,tra(2,it),tra(2,it+1),tol*t(it+1),tol*tra(1,it),60);

        % Calculate the 'exact' point of impact;
        aux=phi(shell,t0,z0,rho*dt);

        % Save the time and point of impact
        t(it+1)=t0+rho*dt; tra(:,it+1)=aux;

        % The range has now been computed, signal succes ...
        flag=1;

        % ... and break from the for-loop.
        break;
    end
end
```

```
% Remove any trailing zeros from output arrays
tra=tra(:,1:it+1); t=t(1,1:it+1);

% Only define the range if the shell hit the ground !!!
if flag==1
  % Isolate the range
    r=tra(1,it+1);
end
```