ADVANCED NUMERICAL METHODS – EXAM OF 5/12/2015 FULL ANSWER

N.B. The answers here are far more complete than could be possibly expected from the student in a real exam situation. Their main purpose is a teaching one, not an assessment-related one.

Part 1, Exercise 1

a) For a Lagrange interpolation polynomial of degree ≤ 3 we need 4 nodes. (The polynomial will be unique, so we will later check that its degree is indeed 3 as requested and not less.) It is usually good to choose the nodes as close as possible to the point x where we want to evaluate the polynomial. Therefore, here we will choose the two closest nodes on each side of $x = 1.37$, namely 1.2, 1.3 and 1.4, 1.5. This is mainly because the Runge effect (spurious oscillations) is stronger between the outer nodes than the near the centre, so it's good if one can choose one node on each side of x. And also because the roundoff errors will also tend to be smaller (which has nothing to do with the Runge effect, since it would take place exactly the same with exact arithmetic).

The nodes are evenly spaced, so we can write the Newton polynomial in terms of finite differences or of divided differences[1]. We will first use finite differences, which is probably what the examiner had in mind when providing evenly-spaced nodes. Another hint is that we are asked to use 5-*decimal*-digit arithmetic (not *significant*-digit), so the table of finite differences will automatically turn out exact (as opposed to the table of divided differences, in which the denominators may create decimals to the right of the fifth)[2].

Having said this, let's proceed. Accepting a little help from Matlab:

```
>> xi = 1.2:0.1:1.5                          % nodes
xi =     1.2     1.3     1.4     1.5
>> fi = [0.14358 0.13065 0.10998 0.08120];    % nodal ordinates
>> [Dky0,table] = anm_tablefd(xi,fi);         % subject's mfile
>> anm_prettyprintable(table,9)               % after a little formatting:
i    xi         yi          Dyi         D2yi         D3yi
0   1.2    0.14358
1   1.3    0.13065    -0.01293
2   1.4    0.10998    -0.02067    -0.00774
3   1.5    0.08120    -0.02878    -0.00811    -0.00037
```

The Newton polynomial in terms of t (using the coefficients in bold typeface) is:

$$q(t) = 0.14358 - 0.01293\frac{t}{1!} - 0.00774\frac{t(t-1)}{2!} - 0.00037\frac{t(t-1)(t-2)}{3!}$$

Its degree is indeed 3, not less, because the last coefficient (finite difference) is not 0.

Let us calculate $t(x)$ for $x = 1.35$ already:

---

[1] The Lagrange representation would also give the same results if the arithmetic were exact, but if there are perturbations it is not optimal with regard to stability.

[2] Using arithmetic with rounding to a fixed number of *decimal* digits (those to the right of the decimal point) is a bit weird, because modern computers always perform floating-point arithmetic with hardware and standards based on *significant* digits (those in the mantissa). Of course they also operate in base 2 rather than base 10; but if one wants to mimic the finite-precision arithmetic of a digital computer, it makes more sense to limit the number of significant digits than the number of decimal digits.

$$h = 0.1; \quad x_0 = 1.2; \quad x = x_0 + th = 1.2 + 0.1t \quad \Rightarrow \quad t = (x - x_0)/h = (1.37 - 1.2)/0.1 = 1.7$$

Substituting: $$q(1.7) = 0.14358 - 0.01293\frac{1.7}{1} - 0.00774\frac{1.7\times 0.7}{2\times 1} - 0.00037\frac{1.7\times 0.7\times(-0.3)}{3\times 2\times 1}$$

We now have to evaluate this "optimally", i.e. with the Hörner-like algorithm, or taking as many common factors as possible. Let's not forget to also take the common factors that we can find in the factorial denominators:

$$q(1.7) = 0.14358 + 1.7\left[-0.01293 + \frac{0.7}{2}\left(-0.00774 + \frac{-0.3}{3}(-0.00037)\right)\right]$$

Now we are asked to use 5-decimal arithmetic, but not the order in which the operations must be executed. With this type of arithmetic, to maximize precision, when there is a choice one should do first the operations that increase the intermediate absolute values, because the digits to the left of the decimal point are "free of charge". In our case this does not make much of a difference, though:

$$q(1.7) = 0.14358 + 1.7\times[-0.01293 + 0.35\times(-0.00774 + \underbrace{0.1\times 0.00037}_{0.000037\,\to\,0.00004})] = \boxed{0.11701}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{0.11701}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{-0.026571\ \to\ -0.02657}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{-0.01563}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{-0.002695\ \to\ -0.00270}$$

$$\underbrace{\qquad\qquad\qquad\qquad}_{-0.00770}$$

Here, −0.002695 was rounded to −0.00270 rather than −0.00269 because the most common practice with endings in 5 (although not the only one) is to round away from 0, which is what we will use here (and Matlab does with its `round` function).

Also note that, instead of multiplying by 0.1 (on the very right), you can multiply by 0.3 and then divide by 3 like in the expression it comes from, or divide by 3 and then multiply by 0.3; as long as you round to 5 decimals after every step you take (pretending that you are a computer with that weird precision), the result turns out to be the same (in this case). As for multiplying –0.00770 by 0.35 (which is the 0.7/2 in the expression it comes from), you can also first multiply by 0.7, round and then divide by 2, or first divide by 2, round and then multiply by 0.7; again, as long as you round every intermediate result to 5 digits (and away from 0 in case of doubt), the same result –0.00270 is obtained.

Let us now use divided differences, which is the more general method. I am genuinely interested in seeing if the roundoff error propagation is worse with this arithmetic, since the table of differences will no longer be exact. Accepting again some help from Matlab, that table is:

```
>> [fii,zi,table]=anm_tableddr(xi,fi);        % subject's mfile
>> anm_prettyprintable(round(table*1e5)/1e5)  % after a little formatting:
i    xi        fi0        fi1        fi2         fi3
0   1.2     0.14358
1   1.3     0.13065    -0.1293
2   1.4     0.10998    -0.2067     -0.387
3   1.5      0.0812    -0.2878    -0.4055    -0.06167
```

This was done internally with double-precision arithmetic, only the final table being rounded off to 5 decimals. To operate with 5-decimal arithmetic throughout we have to do it "more manually":

```
>> format long g
>> f11 = (0.13065-0.14358)/(1.3-1.2)
f11 =              -0.1293
>> f21 = (0.10998-0.13065)/(1.4-1.3)
f21 =              -0.2067
```

```
>> f31 = (0.08120-0.10998)/(1.5-1.4)
f31 =              -0.2878
>> f22 = (-0.2067 - -0.1293)/(1.4-1.2)
f22 =               -0.387
>> f32 = (-0.2878 - -0.2067)/(1.5-1.3)
f32 =              -0.4055
>> f33 = (-0.4055 - -0.3870)/(1.5-1.2)
f33 =  -0.061666666666667
>> f33 = -0.06167   % only value needed of "manual" roundoff
f33 =             -0.06167
```

So, here, rounding off at the very end gave the same result as using 5-decimal-precision arithmetic.

The Newton polynomial is then (using the numbers in bold in the table):

$$p_3(x) = f_{00} + f_{11}\,(x-x_0) + f_{22}\,(x-x_0)(x-x_1) + f_{33}\,(x-x_0)(x-x_1)(x-x_2) =$$

$$= 0.14358 - 0.1293\,(x-1.2) - 0.387\,(x-1.2)(x-1.3) - 0.06167\,(x-1.2)(x-1.3)(x-1.4)$$

Its degree is again 3, of course. To evaluate it "optimally" in $x = 1.37$ we use the Hörner-like algorithm, or take as many common factors as possible:

$$p_3(1.37) = 0.14358 - 0.1293\times 0.17 - 0.387\times 0.17\times 0.07 - 0.06167\times 0.17\times 0.07\times(-0.03) =$$

$$= 0.14358 + 0.17[-0.1293 + 0.07(-0.387 + \underbrace{0.03\times 0.06167}_{0.0018501\,\to\,0.00185})] = \underline{0.11702}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{0.11702}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{-0.0265642\;\to\;-0.02656}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{-0.15626}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{-0.0269605\;\to\;-0.02696}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{-0.38515}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad}_{0.0018501\,\to\,0.00185}$$

In this case the exact value of $p_3(1.37)$ is 0.117015715 (as can be obtained with any polynomial representation, including the Lagrange one, at full precision). This rounds off to 0.11702, not 0.11701, so in this case the general Newton polynomial in terms of divided differences propagates roundoff errors a little bit better than the one with finite differences. However, in no way should this be considered to be a general rule. If the precision of the arithmetic were based on significant rather than decimal digits, both representations of the polynomial (with finite or with divided differences) would probably propagate errors very similarly, because precision is carried by the mantissas while order of magnitude is carried by the exponents all along. In any case, from this point we will use 0.11701 (the value obtained with finite differences) rather than 0.11702 (the slightly better value obtained with divided differences) because, the nodes being equally-spaced, that's what the examiner surely had in mind.

To estimate the error made with 0.11701 there are various possibilities. The conceptually clearest reasoning goes as follows. If the error made is, by definition, the exact value of $f(1.37)$ minus 0.11701, then our *best estimation* of the error made must logically be our *best estimation* of $f(1.37)$ minus 0.11701. And to look for our best estimation of $f(1.37)$, in this case it is reasonable to expect that the first and last nodes, not yet used, can both add some precision to it. Therefore we can calculate $p_5(1.37)$ using all 6 nodes provided (our best estimation of $f(1.37)$) and estimate the previous error as $p_5(1.37) - 0.11701$. One can compute $p_5(1.37)$ using either finite or divided differences. In both cases, with some foresight when writing the initial table we would have left room for additional numbers on top and below it (a new principal diagonal and a new bottom line), because all the rest is the same.

Another possibility to estimate that error is to use the well-known formula $E = f[x_0, \ldots, x_n, x]\,\Pi(x)$ and assume that the divided difference in it is close to some other divided differences of the same order in the table corresponding to all 6 points provided. That means assuming that those divided differences

do not change much if one of the points is x rather than some nearby node, which is often a reasonable assumption if one considers how divided differences are related with derivatives.

But this is not what we will do either. We will use the equivalent of $E = f[x_0, \ldots, x_n, x]\,\Pi(x)$ in terms of finite differences, because that is the formula we will be asked to prove in Exercise 2 below. To use it, let us first construct the full table of finite differences corresponding to all 6 points. In it, only the numbers in bold are new:

```
>> xi = 1.1:0.1:1.6                                          % all nodes
xi =     1.1     1.2     1.3     1.4     1.5     1.6
>> fi = [0.14927 0.14358 0.13065 0.10998 0.08120 0.04406];  % all ordinates
>> [Dky0,table] = anm_tablefd(xi,fi);                        % our mfile
>> anm_prettyprintable(table,9)               % after a little formatting:
i    xi         yi          Dyi         D2yi         D3yi        D4yi     D5yi
0   1.1     0.14927
1   1.2     0.14358     -0.00569
2   1.3     0.13065     -0.01293     -0.00724
3   1.4     0.10998     -0.02067     -0.00774      -0.0005
4   1.5      0.0812     -0.02878     -0.00811     -0.00037     0.00013
5   1.6     0.04406     -0.03714     -0.00836     -0.00025     0.00012    -1e-5
```

Both finite differences of order 4 are similar (and hence the one of order 5 is close to 0). This suggests that the original function $f(x)$ might have been a polynomial of degree 4, later affected by noise. And it certainly means that using the last node alone, 1.6, to estimate the error, will give a similar result to using the first one, 1.1, or to using both. We will first use 1.6, since this allows us to maintain the initial numbering of the nodes ($x_0 = 1.2, \ldots, x_3 = 1.5$) with the new one being $x_4 = 1.6$, and to apply the formula of Exercise 2 "as is" with $n = 3$:

$$e(x) \approx \frac{\Delta^{n+1} f(x_0)}{(n+1)!} t(t-1)(t-2)\cdots(t-n) \quad \underset{n=3}{\rightarrow} \quad e(x) \approx \frac{\Delta^4 f(x_0)}{4!} t(t-1)(t-2)(t-3) \tag{1}$$

$$e(1.37) \approx \frac{0.00012}{4!} 1.7 \times 0.7 \times (-0.3) \times (-1.3) = \boxed{2.3205\text{e-}6}$$

which, if one insists in using 5-decimal-arithmetic, would round off to $\boxed{0}$. Of course it is not reasonable to estimate the error incurred by $f(1.37) \approx 0.11701$ as 0, which would be as much as saying that we think 0.11701 is the exact value of $f(1.37)$, unless we interpret it as saying "0.11701 contains basically all the precision allowed by our 5-decimal arithmetic", which is quite true.

To discuss a little bit more about error estimation, let us now drop the 5-decimal and use double-precision arithmetic. As said before, the "exact" value corresponding to 0.11701 or 0.11702 is $p_3(1.37) = 0.117015715$. Because of the way (1) is derived, the error estimation it provides, 2.3205e-6, is identically equal to $p_4(1.37) - p_3(1.37)$ when $p_4(x)$ is the interpolation polynomial by the last 5 points. Similarly, the error estimation obtainable with the first-node information only is equal to $p_4(1.37) - p_3(1.37)$ when $p_4(x)$ is calculated with the first 5 nodes. That turns out to be 2.513875e-6, if you do it. One can even use (1) to obtain the same result by renaming 1.5, 1.4, 1.3, 1.2 as x_0, x_1, x_2, x_3 respectively, and use the finite difference 0.00013.

Finally, if one wants to use *both* the first and the last node to estimate the error, one could take the average of 2.3205e-6 and 2.513875e-6, which is about 2.4172e-6, but, as said above, the conceptually best idea is to calculate $p_5(x)$ with all the nodes and then do $p_5(1.37) - p_3(1.37)$. Doing that gives 2.4094525e-6, which could be our best estimation of the error made if the data and the arithmetic were of higher precision and $f(x)$ behaved well.

b) Once $p_3(x)$ is known from part a), we just have to solve $p_3(x) = 0.12$ for x (or $q(t) = 0.12$ for t), and the first thing that comes to mind is trying to avoid solving a cubic equation. We will do that later approximately using inverse interpolation. But the cubic equation can actually be solved exactly with an existing closed formula (which is possible too for the general quartic equation, but not quintic), or iteratively (e.g. with the Newton-Raphson method, which works very well with polynomials). Any of those methods will give $x = 1.356246312$, which will later be nice to have for comparison purposes.

However, the expected way to go in this exercise is to use inverse interpolation, i.e. to interchange abscissas and ordinates and evaluate a cubic polynomial rather than solve it for x. When both the abscissas and the ordinates are monotonic, like here, one can normally do that safely:

```
>> xi = [0.14358 0.13065 0.10998 0.08120];  % new abscissas = old ordinates
>> yi = 1.2 : 0.1 : 1.5;                    % new ordinates = old abscissas
>> [fii,zi,table] = anm_tableddr(xi,yi);    % because nodes unevenly spaced
>> table = signif(table,5);                 % round to 5 significant digits
>> anm_prettyprintable(table)               % after a little formatting:
i         xi     fi0         fi1         fi2         fi3
0    0.14358     1.2
1    0.13065     1.3      -7.734
2    0.10998     1.4     -4.8379     -86.191
3     0.0812     1.5     -3.4746     -27.569     -939.76
>> [y012,coefs,chars] = anm_newton(0.12,xi,yi);
>> chars{:}
p3(x) = 1.2 + -7.734*(x-0.14358) + -86.191*(x-0.14358)*(x-0.13065) + -
939.76*(x-0.14358)*(x-0.13065)*(x-0.10998)
p3(x) = ((-939.76*(x-0.10998) + -86.191)*(x-0.13065) + -7.734)*(x-0.14358)
+ 1.2
p3(0.12) = ((-939.76*(0.12-0.10998) + -86.191)*(0.12-0.13065) + -
7.734)*(0.12-0.14358) + 1.2
p3(0.12) = ((-939.76*0.01002 + -86.191)*-0.01065 + -7.734)*-0.02358 + 1.2
>> y012, y012 = signif(y012,5)
y012 =    1.35835696196193
y012 =              1.3584
```

Therefore $\boxed{f^{-1}(0.12) \approx 1.3584}$

The value obtained, 1.3584, is not too far from the theoretically exact one $x = 1.3562\ldots$

Part 1, Exercise 2

We start from: $e(x) = f[x_0,\ldots,x_n,x]\Pi(x) = f[x_0,\ldots,x_n,x](x-x_0)(x-x_1)\cdots(x-x_n)$

Substituting $x = x_0 + th \quad \rightarrow \quad \begin{cases} x - x_0 = th \\ x - x_1 = x_0 - x_1 + th = -h + th = (t-1)h \\ x - x_2 = x_0 - x_2 + th = -2h + th = (t-2)h \\ \vdots \\ x - x_n = (t-n)h \end{cases}$

we get: $e(x) = f[x_0,\ldots,x_n,x]\, th\,(t-1)h\,(t-2)h\cdots(t-n)h = f[x_0,\ldots,x_n,x]\, h^{n+1} t(t-1)(t-2)\cdots(t-n)$

Now if the divided difference $f[x_0, \ldots, x_n, x]$ can be estimated by means of another divided difference of the same order, for instance $f[x_0, \ldots, x_n, x_{n+1}]$, where $x_{n+1} = x_n + h$ is a new equally-spaced node:

$$e(x) \approx f[x_0, \ldots, x_n, x_{n+1}]\, h^{n+1} t(t-1)(t-2)\cdots(t-n)$$

where t, of course, still corresponds to x (not to x_{n+1}).

Finally, writing the divided difference on a set of evenly-spaced nodes in terms of the finite difference of the same order:

$$e(x) \approx \frac{\Delta^{n+1} f(x_0)}{(n+1)!\, h^{n+1}} h^{n+1} t(t-1)(t-2)\cdots(t-n) = \Delta^{n+1} f(x_0) \frac{t(t-1)(t-2)\cdots(t-n)}{(n+1)!}$$

as we were asked to prove. ■

The "weak link" in this "proof" is assuming that we can approximate $f[x_0, \ldots, x_n, x]$ by $f[x_0, \ldots, x_n, x_{n+1}]$ even if x is not another evenly-space node. Can we? How reasonable is that? Well, of course we can do it if we want to, but a way to justify that it is reasonable is that both are related with $f^{n+1)}$ at some point between the nodes or between the nodes and x:

$$f[x_0, \ldots, x_n, x] = \frac{f^{n+1)}(\xi_1)}{(n+1)!} \quad \text{for some } \xi_1 \text{ between } x_0, \ldots, x_n, x$$

$$f[x_0, \ldots, x_n, x_{n+1}] = \frac{f^{n+1)}(\xi_2)}{(n+1)!} \quad \text{for some } \xi_2 \text{ between } x_0, \ldots, x_n, x_{n+1}$$

Of course ξ_1 and ξ_2 are different points in general, but if $f^{n+1)}$ behaves well enough, $f^{n+1)}(\xi_1)$ will not be too different from $f^{n+1)}(\xi_2)$. And, at any rate, as they say in Spain, "el movimiento se demuestra andando" ("movement is proved by moving"), so, in order to, as requested, "prove" that that error "can" be approximated in that way, it is enough to do it. I do it, so in that way I prove it can be done.

A better way to justify the expression we are asked to prove is probably to note that it is equal to $h_{n+1}(x(t))$:

$$\Delta^{n+1} f(x_0) \frac{t(t-1)(t-2)\cdots(t-n)}{(n+1)!} = \Delta^{n+1} f(x_0) \binom{t}{n+1} = q_{n+1}(t) - q_n(t)$$

In other words, this is precisely the term that must be added to $q_n(t)$ in order to obtain $q_{n+1}(t)$. If we assume that $q_{n+1}(t(x))$ is a better approximation to $f(x)$ than $q_n(t(x))$ (because it uses information from one more interpolation point), then it is reasonable to estimate the error made by $q_n(t(x))$ as our best estimation of $f(x)$ minus $q_n(t(x))$: $e(x(t)) \approx q_{n+1}(t) - q_n(t) = \Delta^{n+1} f(x_0) \binom{t}{n+1}$.

Part 1, Exercise 3

The Chebyshev polynomial of the first kind of degree n, $T_n(t)$, is defined as the polynomial whose values in the interval [–1,1] are:

$$T_n(t) = \cos(n \operatorname{acos}(t)) \quad (n = 0, 1, 2, \ldots)$$

Unless your name is Chebyshev, these do not immediately look like polynomials in t, but they are. Here comes the proof.

For $n = 0$ it is trivially true: $T_0(t) = \cos(0) \equiv 1$
which is a polynomial of degree 0 in [–1,1].

For $n = 1$ it is also trivially true: $T_1(t) = \cos(\operatorname{acos}(t)) = t$
which is a polynomial of degree 1 in [–1,1].

For higher degrees we will prove the recurrence relation:

$$T_{n+1}(t) = 2t\, T_n(t) - T_{n-1}(t)$$

We will call $\theta = \text{acos}(t)$, $\cos(\theta) = t$

On the left: $T_{n+1}(t) = \cos((n+1)\,\text{acos}(t)) = \cos(n\theta+\theta) = \cos(n\theta)\cos(\theta) - \sin(n\theta)\sin(\theta)$

On the right, 1st term: $2t\,T_n(t) = 2t\cos(n\theta) = 2\cos(\theta)\cos(n\theta)$

On the right, 2nd term: $T_{n-1}(t) = \cos((n-1)\theta) = \cos(n\theta-\theta) = \cos(n\theta)\cos(\theta) + \sin(n\theta)\sin(\theta)$

Full right-hand side: $2t\,T_n(t) - T_{n-1}(t) = \cos(n\theta)\cos(\theta) - \sin(n\theta)\sin(\theta)$

which coincides with the left-hand side, as we wanted to prove. ■

Let us particularize the recurrence relation until $T_4(t)$:

$$T_2(t) = 2t\,T_1(t) - T_0(t) = 2t\,t - 1 = 2t^2 - 1$$

$$T_3(t) = 2t\,T_2(t) - T_1(t) = 2t\,(2t^2-1) - t = 4t^3 - 3t$$

$$T_4(t) = 2t\,T_3(t) - T_2(t) = 2t\,(4t^3-3t) - (2t^2-1) = 8t^4 - 8t^2 + 1$$

To be sure let me check with `anm_chebyshev1`, which implements this same recurrence relation:

```
>> [cc,cccc,chch] = anm_chebyshev1(4); disp(chch)
      1
      t
      2*t^2 - 1
      4*t^3 - 3*t
      8*t^4 - 8*t^2 + 1
```

All is fine.

Continuing like this always gives another polynomial of degree one unit higher (the leading coefficient being 2^{n-1}), which proves that $T_n \in \text{P}_n\ \forall n \in \text{Z}$, so they can be called polynomials.

As for the roots t_i: $T_n(t) = \cos(n\,\text{acos}(t)) = 0 \quad \Rightarrow \quad n\,\text{acos}(t) = \pi/2 + k\pi \quad \Rightarrow$

$$\text{acos}(t) = \frac{\pi/2 + k\pi}{n} \quad \rightarrow \quad \boxed{t_i = \cos\left(\frac{\pi/2 + k\pi}{n}\right) \quad (k = 0,1,2,3,n-1)}$$

because after $k = n$ values start to repeat.

Particularizing for $n = 4$, the roots of $T_4(t)$ are:

$$\boxed{t_i = \cos\left(\frac{\pi/2 + k\pi}{4}\right) \quad (k = 0,1,2,3)}$$

With Matlab:

```
>> k = 0:3; ti = cos( (pi/2+k*pi)/4 )
ti =  0.92388      0.38268     -0.38268     -0.92388
```

Part 1, Exercise 4

a) I will use the method of indeterminate coefficients. I have to determine 3 coefficients (A_0, B_0, B_1), so I will impose the exact integration of the first three monomials (1, x, x^2):

$$\int_0^h 1dx = h = hA_0 1 + h^2\left[B_0 0 + B_1 0\right] \quad \Rightarrow \quad A_0 = 1$$

$$\int_0^h x dx = \frac{h^2}{2} = hA_0\frac{h}{2} + h^2\left[B_0 1 + B_1 1\right] = \frac{h^2}{2}1 + h^2\left[B_0 1 + B_1 1\right] \quad \Rightarrow \quad B_1 = -B_0$$

$$\int_0^h x^2 dx = \frac{h^3}{3} = hA_0\left(\frac{h}{2}\right)^2 + h^2\left[B_0 2\cdot 0 + B_1 2h\right] = \frac{h^3}{4} + 2h^3 B_1 \quad \Rightarrow \quad B_1 = \frac{1}{2}\left(\frac{1}{3} - \frac{1}{4}\right) = \frac{1}{24}$$

Hence $A_0 = 1, \quad B_0 = \frac{-1}{24}, \quad B_1 = \frac{1}{24} \quad \Rightarrow$ $\boxed{\int_0^h f(x)dx \approx h\, f\left(\frac{h}{2}\right) + \frac{h^2}{24}\left[-f'(0) + f'(h)\right]}$

The polynomial degree of exactitude N of the formula will therefore be 2 at least (because if it integrates 1, x and x^2 exactly it will also integrate exactly any linear combination of them); but it may be higher. I will know N as soon as I find the first monomial that is not integrated exactly. Trying with $f(x) = x^3$: $\int_0^h x^3 dx = \frac{h^4}{4} = h\left(\frac{h}{2}\right)^3 + \frac{h^2}{24}\left[-3\cdot 0^2 + 3h^2\right] + E = \frac{h^4}{8} + \frac{h^4}{8} + E \Rightarrow E = 0 \Rightarrow N \geq 3$

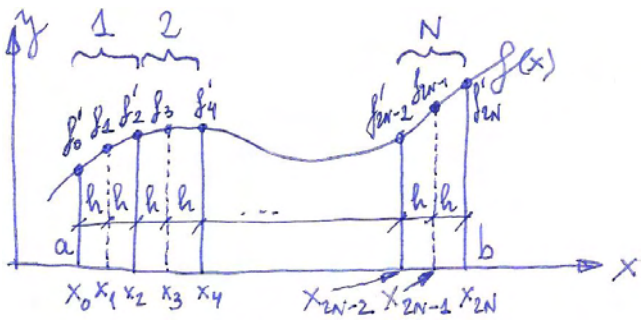
With $f(x) = x^4$:

$$\int_0^h x^4 dx = \frac{h^5}{5} = h\left(\frac{h}{2}\right)^4 + \frac{h^2}{24}\left[-0 + 4h^3\right] + E = \frac{h^5}{16} + \frac{h^5}{6} + E \quad \Rightarrow \quad \frac{E}{h^5} = \frac{1}{5} - \frac{1}{16} - \frac{1}{6} = \frac{-7}{240} \neq 0$$

Therefore[3] $N = 3$ and the error term has the form $E = Kf^{N+1)}(\xi) = Kf^{4)}(\xi)$ for some $\xi \in (0,h)$. In the case of $f(x) = x^4$, $f^{4)}(\xi) = 4! = 24$ which, being a constant, does not depend on ξ, and that allows us to isolate the constant K in the error term:

$$\frac{K\cdot 24}{h^5} = \frac{-7}{240} \quad \Rightarrow \quad K = \frac{-7h^5}{5760} \quad \Rightarrow \quad \boxed{E = \frac{-7f^{4)}(\xi)}{5760}h^5 \quad \text{for some } \xi \in (0,h)}$$

b) A little scheme always helps. Here N is the number of subintervals, not the polynomial degree of exactitude:



Calling: $h = \frac{b-a}{2N}; \quad x_i = a + ih \quad (i = 0,...,2N); \quad f_i = f(x_i); \quad f_i' = f'(x_i)$

[3] One might have expected the polynomial degree of the rule to be at least 4 based on this: a node where f' is an interpolation datum is like a double node (two infinitely close interpolation points that define f'). Hence we have two nodes on $x = 0$, one node on $x = h/2$, and another two nodes on $x = h$, which makes 5, and hence minimum degree of exactitude 4. The failure in this reasoning is, of course, that only f', but not f, is used on both end nodes, so the rule does not use all the possible information in 5 nodes. This is also the reason why it is easier to apply the method of indeterminate coefficients to obtain the weights of the quadrature rule, rather than integrate an "osculating" polynomial for which $f(z_0)$ and $f(z_4)$ are unknown.

we have $$Q_C = h\left(f_1 + f_3 + \ldots + f_{2N-1}\right) + \frac{h^2}{24}\left(-f_0' + f_2' - f_2' + \ldots + f_{2N-2}' - f_{2N-2}' + f_{2N}'\right) =$$

$$= \boxed{h\left(f_1 + f_3 + \ldots + f_{2N-1}\right) + \frac{h^2}{24}\left(-f'(a) + f'(b)\right)}$$

which can also be written in other similar ways (using the Σ symbol, taking common factor *h*, etc.)

The error term of the compound rule is not asked, but let's obtain it anyway. It is obviously the sum of the errors (with their signs) of all the simple rules used:

$$E_C = \sum_{i=1}^{N} E_i = \sum_{i=1}^{N} \frac{-7 f^{4)}(\xi_i)}{5760} h^5 = \frac{-7h^5}{5760} \sum_{i=1}^{N} f^{4)}(\xi_i) = \frac{-7h^5}{5760} N \frac{\sum_{i=1}^{N} f^{4)}(\xi_i)}{N} = \frac{-7h^5}{5760} N \overline{f^{4)}}$$

where E_i is the error made in the i-th subinterval, ξ_i is some point in it, and $\overline{f^{4)}}$ is the arithmetic mean of the N values of $f^{4)}$ in those points. As such, it must be some intermediate value between at least two of those values $f^{4)}(\xi_i)$, so, if $f^{4)}$ is continuous between the nodes, by virtue of the Intermediate Value Theorem, there must be at least one ξ between the nodes for which $\overline{f^{4)}} = f^{4)}(\xi)$. Substituting:

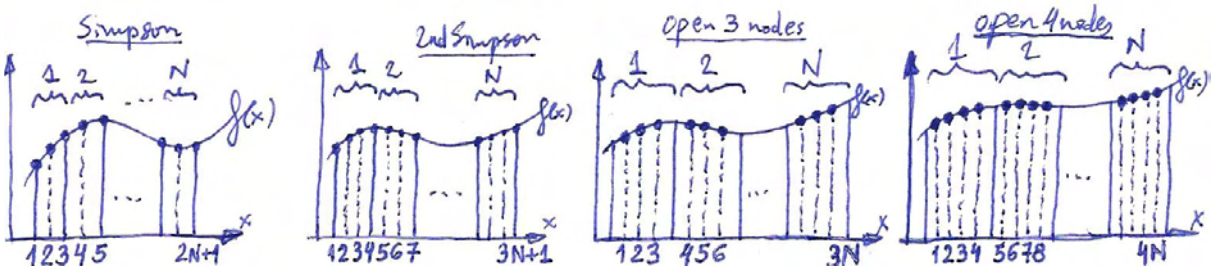
$$E_C = \frac{-7h^5}{5760} N f^{4)}(\xi) = \frac{-7h^5}{5760} \frac{b-a}{2h} f^{4)}(\xi) = \boxed{\frac{-7(b-a) f^{4)}(\xi)}{11520} h^4}$$

where N was substituted by $(b–a)/(2h)$ because each subinterval has width $2h$.

c) The first point of comparison is that the rule in this exercise needs the derivatives of $f(x)$, even if only on a and b. This alone may be a deal-breaker in many situations, since Newton-Cotes rules don't need any derivatives.

Having said this, we need to know what rules we have to compare ours with, i.e. what Newton-Cotes rules have the same polynomial degree of exactitude 3 as the protagonist in this exercise. The polynomial degree of interpolatory quadrature rules is $N \geq n$ by construction (where n+1 is the number of interpolation data), but with an odd number of simple nodes that are symmetrically placed on both sides of the integration interval midpoint, at least one unit is gained over n (and in the case of the Newton-Cotes rules, not more than 1; Gauss rules do much better, roughly doubling the polynomial degree of the Newton-Cotes rule of the same number of nodes).

According to this, the Newton-Cotes rules of the same polynomial degree $N = 3$ are the Simpson and the 2nd Simpson ones (closed of 3 and 4 nodes) and the open rules of 3 and of 4 nodes. Compounding them with the same number of subintervals N, one gets something like this:



We will assume that the comparison must be done in terms of computational cost and precision.

The computational cost of the rule in this exercise is equal to N evaluations of f, namely one at each subinterval midpoint (apart from the two evaluations of f'). And, as can be seen in the figure, the compound Simpson rule requires $2N$+1 evaluations of f, the compound second Simpson rule requires

$3N+1$, the compound open rule of 3 nodes requires $3N$, and the one of 4 nodes requires $4N$, all with the same polynomial degree.

However, this comparison is quite unfair and ultimately useless, because the number of subintervals does not have to be the same for all five rules. A more useful comparison would be to use, for every rule, the number of subintervals needed for its precision to be similar to the precision of the other rules, and then compare the corresponding computational costs. So let us have a look at precision.

All rules being of the same polynomial degree, they are also of the same order of convergence (since both numbers are directly liked in interpolatory quadrature rules of any kind, even Gauss or osculating). In our case the order of convergence (sometimes also called order of precision) of the simple rule is 5 –the power of h in the error term we obtained in section a)–, so the order of precision of the compound rule is 4 (always one unit less, quite naturally). The same will be true of the other 4 rules considered here.

But the fact that the *order* of precision of all the rules is the same does not mean that the *precisions* themselves are the same! In all 5 cases, for sufficiently small values of h, the error is approximately proportional to h^4; but the constant of proportionality may change a lot. If one method has $E \approx k_1 h^4$ and another method has $E \approx k_2 h^4$ with $k_2 \approx k_1/1000$, the second method is about a thousand times more precise than the first one (for equal values of h!) even if the order of precision is 4 in both cases.

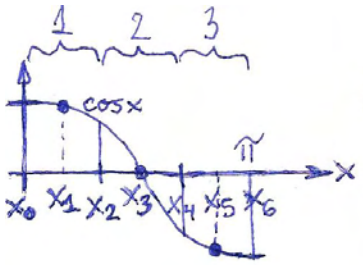
Doing this comparison properly would need the full error terms of all 5 compound rules involved, because that is where you see the values of the constants of proportionality between h^4 and the errors. I am sure this analysis is well out of the scope that the examiners had in mind with this question, but just in case here are the terms corresponding to the Newton-Cotes rules (actually calculated, not looked up in a table, by the function `anm_nc`):

```
>> [Q,h,xi,wi,Es,Ks,Ec,Kc] = anm_nc(3,1); Ec  % Compound Simpson rule error
Ec =  -1/180 * h^4 * (b-a) * f^{4)}(\xi)
>> [Q,h,xi,wi,Es,Ks,Ec,Kc] = anm_nc(4,1); Ec  % Compound 2nd Simpson rule E
Ec =   -1/80 * h^4 * (b-a) * f^{4)}(\xi)
>> [Q,h,xi,wi,Es,Ks,Ec,Kc] = anm_nc(3,0); Ec  % Compound 0pen N-C 3 nodes E
Ec =    7/90 * h^4 * (b-a) * f^{4)}(\xi)
>> [Q,h,xi,wi,Es,Ks,Ec,Kc] = anm_nc(4,0); Ec  % Compound 0pen N-C 4 nodes E
Ec =  19/144 * h^4 * (b-a) * f^{4)}(\xi)
```

Be aware that this is not as simple as just comparing the constants −1/180, −1/80, 7/90 and 19/144 with the one in the error term of the compound rule of this exercise (which we have not calculated), because h is not always the same fraction of $(b–a)$. And finally note that not only is the power of h the same in all error terms, but also the order of differentiation of f, which allows for meaningful comparisons. This is not going to happen (because all these numbers are linked); but if one error term depended on $f^{4)}$ and another one on $f^{5)}$, we could not really compare computational costs for roughly the same precision (or precisions for the same computational cost).

d) The rule gives: $Q_C = \boxed{0}$

as is obvious from the figure.



$f_0' = f_6' = 0$

$f_1 + f_3 + f_5 = 0$ (by symmetry)

The exact value of the integral is also 0, so 0 is both an upper and a lower bound of the error made.

Yep, as simple as this. You are always allowed to go the simplest way to answer the questions!

In order to find bounds of the error made it is also possible to apply the error term of the simple rule, obtained in section a), to each one of the three subintervals; or the error term of the compound rule, obtained in section b) to the whole interval. But there's no reason to use any other method; in fact, no method can provide better (i.e. tighter) bounds than the ones we obtained!

Part 2, Exercise 1

a) We first transform the ODE into a system of order 1 by calling $y_1 = y$, $y_2 = y'$, $y_3 = y''$, so:

$$\begin{cases} y_1' = y_2 \\ y_2' = y_3 \\ y_3' = -2y_1 + y_3 + \log t \end{cases} \quad \text{or} \quad \mathbf{y}' = \underbrace{\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ -2 & 0 & 1 \end{pmatrix}}_{J} \mathbf{y} + \underbrace{\begin{pmatrix} 0 \\ 0 \\ \log t \end{pmatrix}}_{\mathbf{g}(t)} = \mathbf{f}(t, \mathbf{y})$$

The initial conditions are: $t_0 = 1, \quad \mathbf{y_0} = \begin{pmatrix} -1 \\ 0 \\ 1 \end{pmatrix}$

The Enhanced Euler method (aka Heun method) can be obtained by modifying the Trapezoidal one a little. In terms of the typical Runge-Kutta constants (since it is an RK2 method), its advance formula can be written as:

$$\begin{cases} \mathbf{k_1} = \mathbf{f}(t_k, \mathbf{y_k}) h_k \\ \mathbf{k_2} = \mathbf{f}(t_k + h_k, \mathbf{y_k} + \mathbf{k_1}) h_k \end{cases} \rightarrow \quad \mathbf{y_{k+1}} = \mathbf{y_k} + \frac{\mathbf{k_1} + \mathbf{k_2}}{2}$$

Since the step size h is not specified I will follow the principle of least effort and use $h = 0.1$. I will also let Matlab do the operations for me with double precision, rounding to 6 significant digits only at the end:

```
>> format long g            % format short g would show less than 6 digits
>> f = @(t,y) [y(2); y(3); -2*y(1)+y(3)+log(t)];  % or matrix notation too
>> t0=1; y0=[-1; 0; 1]; h=0.1; t1=1.1; t2=1.2;    % IC and values of h, ti
>> k1 = f(t0,y0) * h
k1 =                     0
                       0.1
                       0.3
>> k2 = f(t0+h,y0+k1) * h
k2 =                    0.01
                        0.13
          0.339531017980433
>> y1 = y0 + (k1+k2)/2
```

```
y1 =                -0.995
                     0.115
          1.31976550899022
>> k1 = f(t1,y1) * h       % overwrite previous k1
k1 =                0.0115
         0.131976550899022
         0.340507568879454
>> k2 = f(t1+h,y1+k1) * h  % overwrite previous k2
k2 =    0.0246976550899022
         0.166027307786967
         0.380959463466363
>> y2 = y1 + (k1+k2)/2
y2 =    -0.976901172455049
         0.264001929342994
          1.68049902516312
>> [tout,yout] = anm_ode(f,[1 1.2],y0,2,'Enhanced Euler')   % to be sure
tout =             1                       1.1                       1.2
yout =            -1                    -0.995       -0.976901172455049
                   0                     0.115        0.264001929342994
                   1           1.31976550899022         1.68049902516312
>> signif(yout,6)
ans =             -1                    -0.995                  -0.976901
                   0                     0.115                   0.264002
                   1                   1.31977                     1.6805
```

So the approximate values of $y(1.1)$, $y'(1.1)$ and $y''(1.1)$ are -0.995, 0.115 and 1.31977 respectively; and the approximate values of $y(1.2)$, $y'(1.2)$ and $y''(1.2)$ are -0.976901, 0.264002 and 1.6805.

Part 2, Exercise 2

The general form of the advance formula of a linear multistep method is:

$$\mathbf{y_{n+k}} = -\sum_{j=0}^{k-1} \alpha_j \mathbf{y_{n+j}} + h\sum_{j=0}^{k} \beta_j \mathbf{f_{n+j}} \qquad (n = 0,..., N-k)$$

where k is the number of steps of the multistep method, $\mathbf{f_i} = \mathbf{f}(t_i, \mathbf{y_i})$ is the right-hand side of the system of ODEs to be solved, and N is the total number of steps that will take us from the initial point at t_0 to the final point at $t_f = t_N$.

The method is explicit when $\mathbf{y_{n+k}}$ is directly expressed in terms of values that are already known by the time one wants to calculate it. In other words, the unknown in the advance formula, $\mathbf{y_{n+k}}$, appears on its left-hand side alone, so there is no need to iterate. This happens when $\beta_k = 0$.

Otherwise the method is implicit, which means that the unknown $\mathbf{y_{n+k}}$ appears on both sides of the advance formula, which must be then solved iteratively (unless $\mathbf{f}(t,\mathbf{y})$ is simple enough that one can manipulate and isolate $\mathbf{y_{n+k}}$). This happens when $\beta_k \neq 0$.

The associated first and second characteristic polynomials are respectively (with $\alpha_k = 1$):

$$\rho(z) = \sum_{j=0}^{k} \alpha_j z^j ; \qquad \sigma(z) = \sum_{j=0}^{k} \beta_j z^j$$

The method is *consistent*[4] iff $\rho(1)=0$ and $\rho'(1)=\sigma(1)$

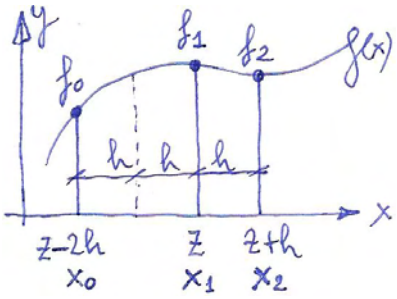
The method is *stable*[5] iff the roots z_i of $\rho(z)$ verify $\|z_i\|\le 1$ on the complex plane, and the ones with modulus 1 are simple roots.

The method is *convergent* iff it is *consistent* and *stable*[6], with order of convergence p iff $\rho(1)=0$ (already checked for consistency) and:

$$\frac{1}{m}\sum_{j=0}^{k}\alpha_j j^m=\sum_{j=0}^{k}\beta_j j^{m-1} \qquad \text{(for } m=1,2,\ldots,p\text{, but not } p+1\text{)}^7$$

Part 2, Exercise 3

a) The idea behind all interpolatory differentiation formulas is to differentiate an interpolation polynomial instead of the original function $f(x)$. In our case it's for the second derivative at z using the three nodes provided: $f''(z)=\underbrace{p_2''(x)}_{D}+E$



Using the Lagrange representation of the interpolation polynomial:

$$p_2''(x)=\left(\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}f_0+\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}f_1+\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}f_2\right)''=$$

$$=\left(\frac{(x-z)(x-z-h)}{(-2h)(-3h)}\right)''f_0+\left(\frac{(x-z+2h)(x-z-h)}{(2h)(-h)}\right)''f_1+\left(\frac{(x-z+2h)(x-z)}{(3h)(h)}\right)''f_2=$$

$$=\left(\frac{x^2+\ldots}{6h^2}\right)''f_0+\left(\frac{x^2+\ldots}{-2h^2}\right)''f_1+\left(\frac{x^2+\ldots}{3h^2}\right)''f_2=\frac{2}{6h^2}f_0+\frac{2}{-2h^2}f_1+\frac{2}{3h^2}f_2$$

$$D=p_2''(z)=\frac{1}{3h^2}f_0+\frac{-1}{h^2}f_1+\frac{2}{3h^2}f_2=A_0f_0+A_1f_1+A_2f_2 \quad\Rightarrow\quad A_0=\frac{1}{3h^2},\quad A_1=\frac{-1}{h^2},\quad A_2=\frac{2}{3h^2}$$

Substituting:

$$\boxed{f''(z)=\frac{f(z-2h)-3f(z)+2f(z+h)}{3h^2}+E}$$

---

[4] Consistency means that the local errors tend to 0 *faster* than h.

[5] Stability of the method itself, regardless of the problem it is solving and the step size. A stable method can work unstably (i.e. it can propagate local errors in an amplified manner, typically rendering totally useless results) if the step size h is larger than some threshold h_t. The methods that never become unstable when applied on stable ODEs or systems thereof are called *unconditionally stable* or *A-stable*.

[6] If the local errors tend to 0 faster than h (consistency), and if they are composed in a damped rather than an amplified manner, i.e. if each global error is less than the sum of all the previous local errors (stability), then the numerical solution in a fixed interval $[t_0,t_f]$ will tend to the exact one as h tends to 0 uniformly (convergence).

[7] If one of the coefficients is 0^0, the indetermination is resolved in favor of $0^0=1$. There are often good reasons to make the rule $x^0=1$ prevail over the rule $0^x=0$. Even Matlab will agree –just go and give it `0^0` to calculate.

In general, and for the same reasons, one can always calculate the coefficients as $A_i = L_i^{k)}(z)$, where A_i is the coefficient of the rule that multiplies f_i, $L_i(x)$ is the corresponding Lagrange base polynomial, k is the order of differentiation, and z the point where we want to approximate $f^{k)}$.

b) I will write the Taylor series that I need and manipulate them as I see appropriate ("ad-hoc"). If $f \in C^3([z-2h, z+h])$:

$$f(z+h) = f(z) + f'(z)h + \frac{f''(z)}{2!}h^2 + \frac{f'''(\xi_1)}{3!}h^3 \quad \text{for some } \xi_1 \text{ between } z \text{ and } z+h$$

$$f(z-2h) = f(z) - f'(z)2h + \frac{f''(z)}{2!}4h^2 - \frac{f'''(\xi_2)}{3!}8h^3 \quad \text{for some } \xi_2 \text{ between } z-2h \text{ and } z$$

I am trying to get rid of $f'(z)$, so I will multiply the first expression by 2 and add term by term:

$$2f(z+h) + f(z-2h) = 3f(z) + 0 + \frac{f''(z)}{2!}6h^2 + \frac{2f'''(\xi_1) - 8f'''(\xi_2)}{3!}h^3$$

As $h \to 0^+$, $\xi_1 \to z^+$ and $\xi_2 \to z^-$, so if $f \in C^3$ in a neighborhood of z, the last numerator will tend to $-6f'''(z)$, and actually be equal to $-6f'''(\xi)$ for some ξ between ξ_1 and ξ_2, and hence in $(z-2h, z+h)$. This is easy to prove using Weierstrass's Intermediate Value Theorem, but the given explanation will suffice here. Substituting that numerator by $-6f'''(\xi)$ and isolating $f''(z)$, which is what we're interested in:

$$\boxed{f''(z) = \frac{f(z-2h) - 3f(z) + 2f(z+h)}{3h^2} + \frac{f'''(\xi)}{3}h} \quad \text{for some } \xi \in (z-2h, z+h)$$

The differentiation formula D is the same as before, but now we also have the error term E, which tends to 0 at least as fast as h (or faster, if $f'''(z) = 0$), so $E = O(h)$ and the order of convergence is 1.

The systematic application of the general theory with $k = n = m = 2$ should give the same result. On the one hand,

$$f''(z) = \underbrace{A_0 f_0 + A_1 f_1 + A_2 f_2}_{D} + E = \sum_{i=0}^{2} A_i f(x_i) + E$$

Now calling $x_i = z + h_i$ (so in our case $h_0 = -2h$, $h_1 = 0$, $h_2 = h$), and if $f \in C^3([z-2h, z+h])$:

$$f(x_i) = f(z) + f'(z)h_i + \frac{f''(z)}{2!}h_i^2 + \frac{f^{3)}(\xi_i)}{3!}h_i^3 \quad \text{for some } \xi_i \text{ between } z \text{ and } x_i$$

Substituting:

$$f''(z) = \sum_{i=0}^{2} A_i \left(f(z) + f'(z)h_i + \frac{f''(z)}{2!}h_i^2 + \frac{f^{3)}(\xi_i)}{3!}h_i^3 \right) + E =$$

$$= \underbrace{\left(\sum_{i=0}^{2} A_i \right)}_{=0} f(z) + \underbrace{\left(\sum_{i=0}^{2} h_i A_i \right)}_{=0} f'(z) + \underbrace{\left(\frac{1}{2!} \sum_{i=0}^{2} h_i^2 A_i \right)}_{=1} f''(z) + \underbrace{\frac{1}{3!} \sum_{i=0}^{2} A_i f^{3)}(\xi_i) h_i^3 + E}_{=0}$$

Now we determine A_0, A_1, A_2 by imposing the first three equations indicated at the bottom of the last expression, and the error term by imposing the last one:

$$\begin{cases} A_0 + A_1 + A_2 = 0 \\ -2hA_0 + 0A_1 + hA_2 = 0 \\ (-2h)^2 A_0 + 0^2 A_1 + h^2 A_2 = 2! \end{cases}$$

Working with the extended matrix to apply Gauss's method:

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ -2 & 0 & 1 & 0 \\ 4 & 0 & 1 & 2/h^2 \end{pmatrix} \begin{matrix} \\ < F_2 + 2F_1 > \\ < F_3 - 4F_1 > \end{matrix} \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 2 & 3 & 0 \\ 0 & -4 & -3 & 2/h^2 \end{pmatrix} < F_3 + 2F_2 > \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 2 & 3 & 0 \\ 0 & 0 & 3 & 2/h^2 \end{pmatrix}$$

From the 3rd equation, $A_2 = \frac{2}{3h^2}$. Substituting into the 2nd, $A_1 = \frac{-1}{h^2}$. And into the 1st, $A_0 = \frac{1}{3h^2}$. Therefore *D* is the same as above. As for *E*:

$$E = -\frac{1}{3!}\sum_{i=0}^{2} A_i f^{3)}(\xi_i) h_i^3 = \frac{-1}{6}\left(A_0 f^{3)}(\xi_0) h_0^3 + A_1 f^{3)}(\xi_1) h_1^3 + A_2 f^{3)}(\xi_2) h_2^3\right) =$$

$$= \frac{-1}{6}\left(\frac{1}{3h^2} f^{3)}(\xi_0)(-2h)^3 + 0 + \frac{2}{3h^2} f^{3)}(\xi_2) h^3\right) = \frac{4 f^{3)}(\xi_0) - f^{3)}(\xi_2)}{9} h$$

As $h \to 0$, the last numerator tends to $3f^{3)}(z)$ and is equal to $3f^{3)}(\xi)$ for some ξ between ξ_1 and ξ_2 and hence between the nodes, so we get the error term $E = \frac{3f^{3)}(\xi)}{9} h = \frac{f^{3)}(\xi)}{3} h$ just like before.

c) The error in the derivative is the derivative of the error, because:

$$f''(z) = p_2''(z) + E \quad \Rightarrow \quad E = f''(z) - p_2''(z) = \left(f(z) - p_2(z)\right)'' = e''(z)$$

where $e(x)$ is the error of the interpolation polynomial, whose expression we know. Making convenient use of some theorems:

$$E = e''(z) = \left(f[x_0, x_1, x_2, z]\Pi(z)\right)'' = \left(f[x_0, x_1, x_2, z]\Pi(z)' + f[x_0, x_1, x_2, z]'\Pi(z)\right)' =$$

$$= f[x_0, x_1, x_2, z]\Pi(z)'' + 2 f[x_0, x_1, x_2, z]'\Pi(z)' + f[x_0, x_1, x_2, z]''\Pi(z) =$$

$$= \frac{f^{3)}(\xi_1)}{3!}\Pi''(z) + 2\frac{f^{4)}(\xi_2)}{4!}\Pi'(z) + 2\frac{f^{5)}(\xi_3)}{5!}\Pi(z)$$

where $\Pi(x) = (x - x_0)(x - x_1)(x - x_2) = (x - z + 2h)(x - z)(x - z - h)$

Here z is a node, so $\Pi(z) = 0$.

First derivative at z: $\Pi'(z) = 0 + (z - z + 2h)(z - z - h) = -2h^2$

Second derivative: $\Pi(x) = x^3 + x^2(-z + 2h - z - z - h) + x\ldots = x^3 + x^2(-3z + h) + x\ldots$

$\Pi''(x) = 6x + 2(-3z + h) + 0$; $\Pi''(z) = 6z + 2(-3z + h) = 2h$

Substituting: $E = \frac{f^{3)}(\xi_1)}{3!} 2h + 2\frac{f^{4)}(\xi_2)}{4!}(-2h^2) = \boxed{\frac{f^{3)}(\xi_1)}{3} h - \frac{f^{4)}(\xi_2)}{6} h^2}$

This is in no contradiction with the expression of *E* obtained before, because these ξ_1, ξ_2 are different from the other ones; and in fact the dominant term, $[f^{3)}(\xi_1)/3]\, h$, coincides.