Modelling and Numerics (MA261) Assignment 2 Tom Dilby U1903831

1.1. $y_{n+1} = (1+P(\lambda h))\, y_n$. Let $\mu = \lambda h$. It follows that

$$y_n = (1+P(\mu))^n y_0 \quad \forall n \geq 0.$$

If $y_0 = 0$, trivially $y_n = 0 \ \forall n \in \mathbb{N}$. So we will assume $y_0 \neq 0$.
Now, for $y_n \to 0$ as $n \to \infty$, we need $|1+P(\mu)| < 1$.
($P(\mu) = \sum_{k=0}^{m} \alpha_k \mu^k$ for some $m > 0$ as P is a non-constant polynomial).

~~So $P(\mu)$, and indeed $1+P(\mu)$, are both $O(\mu^m)$ functions.~~
~~So, if m is even, as $\mu \to -\infty$,~~
As P is a non-constant polynomial as $\mu \to -\infty$,

$P(\mu) \to \infty$ if m is even

$P(\mu) \to -\infty$ if m is odd.

so clearly for ~~some values of μ~~, some $M > 0$, $|1+P(\mu)| > M$ for all $\mu < \bar{\mu}$ for some $\bar{\mu} < 0$. In other words, our sequence (y_n) will not converge to 0 for any arbitrary $\mu < 0$.
$\mu = \lambda h$ and $\lambda < 0$ so again, as $h > 0$, we have shown one can't pick any arbitrary $h > 0$ and ensure convergence of y_n to 0.

So the method is ~~never~~ unconditionally stable.

1.2. For reference, the "first" reaction will be $E + nS \xrightarrow{k_f} C_1$, the second $E + nS \xleftarrow{k_r} C$ and the third, $C \xrightarrow{k} P + E$.
The reactants/chemicals will be ordered as E, S, C then P.

a) $w = \begin{pmatrix} k_f E S^n \\ k_r C \\ kC \end{pmatrix}$ and $\Gamma = \begin{pmatrix} -1 & 1 & 1 \\ -n & n & 0 \\ 1 & -1 & -1 \\ 0 & 0 & 1 \end{pmatrix}$

b) $\Gamma^T \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \\ 1 \end{pmatrix} \neq \underline{0}$ so $\text{rank}(\Gamma^T) \geq 1$.

$\text{Rank}(\Gamma^T) + \text{Nullity}(\Gamma^T) = 3$. If we can find two linearly independent vectors in the kernel of Γ^T then $\text{nullity}(\Gamma^T) \geq 2$, and so $\text{Rank}(\Gamma^T) = 1$ and $\text{Nullity}(\Gamma^T) = 2$ so these two vectors will also span Γ^T's kernel (as k linearly independent vectors of a k-dimensional vector space must span the space).

Let $\underline{v} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix}$ $\Gamma^T \underline{v} = \begin{pmatrix} -1 & -n & 1 & 0 \\ 1 & n & -1 & 0 \\ 1 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} -x - ny + z \\ x + ny - z \\ x - z + w \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$ if

$\underline{v} \in \ker(\Gamma^T)$.

One solution is $x = 1$, $z = 1$ and $y = w = 0$. So let $c_1 = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix}$

Another solution is letting $z = 0$, $y = -1$, $x = n$ and $w = -n$.

So let $c_2 = \begin{pmatrix} n \\ -1 \\ 0 \\ -n \end{pmatrix}$ $\Gamma^T c_1 = \Gamma^T c_2 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$.

Suppose $\alpha c_1 + \beta c_2 = 0$. So $\begin{pmatrix} \alpha + \beta n \\ -\beta \\ \alpha \\ -\beta n \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$.

Clearly $\alpha = \beta = 0$ is the only solution so c_1 and c_2 are linearly independent thus, as explained above, span the kernel of Γ^T.

So then, from the notes and recalling order of appearance of the chemicals,

$H_1 = E + C$ $\quad H_2 = nE - S - nP$ are the two resulting conserved quantities.

c) Let E_0, S_0, C_0 and P_0 denote the starting values of the chemicals.
We know $C_0 = P_0 = 0$.
It follows $E + C = E_0 + C_0 = E_0$. So $E = E_0 - C$, or $C = E_0 - E$.

Also, $nE - S - nP = nE_0 - S_0 - nP_0 = nE_0 - S_0$

So $S = nE - nE_0 + S_0 - nP$

Thus $E = \frac{1}{n}(S + nE_0 - S_0 + nP) = \frac{1}{n}(S - S_0) + P + E_0$.

And $C = E_0 - E = \frac{1}{n}(S_0 - S) - P$.

d) Recalling $\frac{d}{dt} y = \Gamma w(y)$, we get

$$S' = -n k_f S^n E + n k_r C$$

(using part c) $= -n k_f S^n \left(\frac{1}{n}(S - S_0) + P + E_0\right) + n k_r \left(\frac{1}{n}(S_0 - S) - P\right)$

$= -k_f S^n (S - S_0) - n k_f S^n P - n k_f E_0 S^n + k_r (S_0 - S) - n k_r P$

$= -k_f S^{n+1} + k_f S_0 S^n - n k_f S^n P - n k_f E_0 S^n + k_r S_0 - k_r S - n k_r P$

~~$-n k_f S^n S$~~

and $P' = kC = \frac{k}{n}(S_0 - S) - kP$

With initial conditions $P_0 = 0$ and $S_0 = S_0$.

1.3. Assume for sim x_0, $x_k \to x$ as $k \to \infty$. Pick any $k \geq 0$.

$x_{k+1} = \phi(x_k)$.

As $\phi \in C^p$, we can use Taylor's Theorem to conclude

$$\phi(x_k + h) = \phi(x) + h\phi'(x) + \cdots + \frac{h^{p-1}}{(p-1)!}\phi^{(p-1)}(x) + O(h^p).$$

Letting $h = x_k - x$, we get

$$x_{k+1} = \phi(x_k) = \phi(x) + (x_k - x)\phi'(x) + \cdots + \frac{(x_k - x)^{p-1}}{(p-1)!}\phi^{(p-1)}(x) +$$

$$+ O((x_k - x)^p).$$

But $\phi'(x) = \cdots = \phi^{(p-1)}(x) = 0$, so

$x_{k+1} = O((x_k - x)^p) + \phi(x) = O((x_k - x)^p) + x$ as $\phi(x) = x$.

i.e.

$$|x_{k+1} - x| = |O((x_k - x)^p)|$$

So $|x_{k+1} - x| \leq C|x_k - x|^p$ for all $k \in \mathbb{N} \cup \{0\}$ so the order of convergence of (x_k) is at least of order p.

Adi Jayakumar (1905687), Tom Dalby (1903831)

MA261 Modelling and Differential Equations Assignment Sheet 2

The implementations of all Python functions mentioned in questions 2.0 – 2.2 can be found in the file “Q1.py”. The implementations of all Python functions mentioned in 2.3 are the same as in the previous assignment and can be found in the file “Q2.py”

2.0)
The required functions were implemented and called “newton(F, DF, x0, eps, k)” and “backwardEuler(f, Df, t0, y0, h)” and can be found in the code. Note that “backwardEuler” calls “get_y1_BE” which checks convergence and sets the tolerance and maximum iterations for Newton’s method.

2.1)
The Python function that was passed into ‘*evolve*’ and the exact solution in the case *c = 1.5* were called “y_prime(t, y)” and its derivative called “dy_prime(t, y)” these can be found in the code.

We then had a function ‘get_*eoc_matrix_and_errors*’ which computed the matrix to be passed into ‘*computeEocs*’ and the errors at time *T* for a given solution method.

For this question, the above function was called with the argument as specified earlier and the returned list containing the data for the EOC was passed into ‘*computeEocs*’ and the ‘*errors*’ list was printed out to console.

We observed that the experimental order of convergence was converging to 1 and therefore concluded that the Backward Euler method converges linearly in the case of this function. Furthermore, we were able to see that the errors seemed to roughly halve as the step size halved as well.

The observed EOCs and errors were

j such that *h_j = T/N_02^j*	EOCs	Errors
0	N/A	0.006090
1	0.985792	0.003075
2	0.992084	0.001546
3	0.995780	0.000775
4	0.997815	0.000388
5	0.998887	0.000194
6	0.999438	0.000097
7	0.999718	0.000049
8	0.999859	0.000024
9	0.999929	0.000012
10	0.999965	0.000006

2.2)
We implemented the Crank-Nicholson method in the Python function “CrankNicholson(f, Df, t0, y0, h)”, which called “get_y1_CN(F, DF, t0, y0, h)” which checks that Newton’s method converges.

In order to generate the convergence data for this method, ‘*get_eoc_matrix_and_errors*’ was called with “CrankNicholson” as an argument. The data it generated was as follows

j such that *h_j = T/N_02^j*	EOCs	Errors
0	N/A	0.000365
1	2.009956	0.000091
2	2.002442	0.000023
3	2.000608	0.000006
4	2.000152	0.000001
5	2.000038	0.000000
6	2.000009	0.000000
7	2.000002	0.000000
8	1.999989	0.000000
9	1.999970	0.000000
10	1.999987	0.000000

We concluded that as the EOC seemed to be converging to *2* that this method converges quadratically. This also seems to follow in the errors, which seem to decease by a factor of 4 each iteration.

2.3)
The required function was passed into the “compute_eoc_matrix_and_errors” function which calls the “evolve” function which returns the approximations for a given step size and then calculates the error for that approximation using the exact solution. Finally, this method returns 2 matrices, one of which is the error matrix and the other is the m x 2 matrix [[h1, e1], … , [hm, em]] which was used to calculate the EOCs. Then, that matrix is passed into “computeEocs” and the list of EOCs and errors were printed to the console.

The errors and EOCs for the forward Euler method for the ODE in question were

j such that *h_j = T/N_02^j*	EOCs	Errors
0	N/A	0.000355
1	1.732236	0.000107
2	0.706665	0.000065
3	4.252644	0.000003
4	-3.326378	0.000034
5	4.309978	0.000002
6	-1.792726	0.000006
7	0.348819	0.000000
8	4.550057	0.000001
9	-1.379553	0.000000
10	3.708349	0.000000

The errors and EOCs for Heun's method for the ODE in question were:

j such that *h_j = T/N_02^j*	EOCs	Errors
0	N/A	0.000141
1	-0.000546	0.000141
2	1.269910	0.000059
3	-0.000082	0.000059
4	4.093470	0.000003
5	-2.328722	0.000017
6	3.311674	0.000002
7	1.034534	0.000001
8	-1.034547	0.000002
9	1.966257	0.000000
10	-0.000001	0.000000

The EOCs do not seem to have an obvious limit for either method, however the errors clearly do converge to 0 for both methods. This indicates that the implementation of the methods were correct and that the issues with the EOCs is caused by the ODE itself. Both methods require the function "f" in the question to be differentiable, which this function is not at 1/sqrt(2). Therefore, the standard convergence results are not necessarily applicable in this case.

Another explanation could be that for step sizes which do not land near 1/sqrt(2), on the iteration which crosses over 1/sqrt(2), that iterate will pick up a large error since it will be using the wrong expression for the derivative after 1/sqrt(2). Therefore, for such step sizes, the error may in fact be larger than for a larger step size where the issue previously discussed does not occur.

Furthermore, this issue is exacerbated by the fact that the absolute errors were very small, so even a small variation in the error from one step size to the next will be magnified when taking logarithms when calculating the EOCs. Therefore, one ends up with an unpredictable sequence of EOCs.

However, both methods do converge on the solution and so in this case, one may as well use the forward Euler method since it is cheaper to perform.