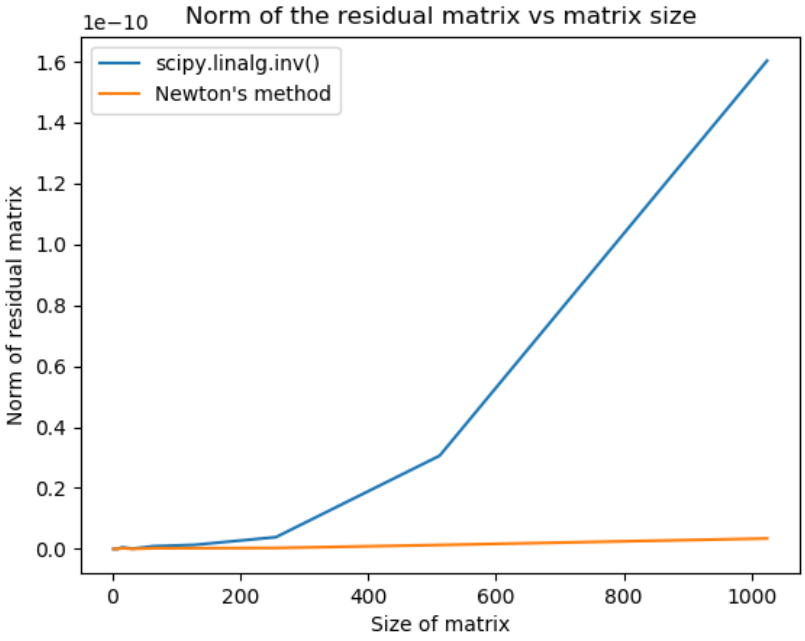
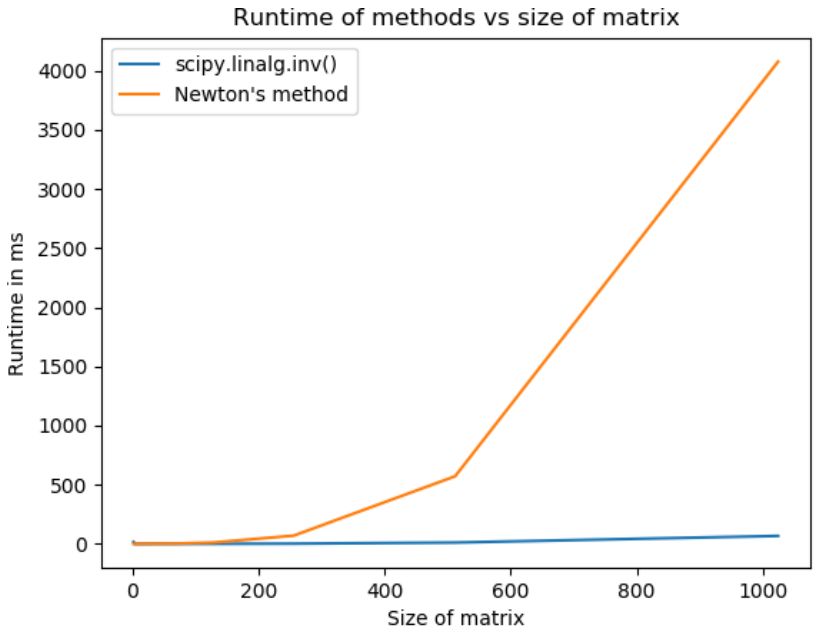
Q1):

a)

$$
\begin{aligned}
R_{k+1} &= I - AX_{k+1} \\
&= I - A(X_k + X_k(I - AX_k)) \\
&= I - A(X_k + X_kI - X_kAX_k)) \\
&= I - AX_k - AX_kI + (AX_k)^2 \\
&= I^2 - 2AX_k + (AX_k)^2 \\
&= (I - AX_k)^2 \\
&= {R_k}^2 \\
E_{k+1} &= A^{-1} - X_{k+1} \\
&= A^{-1} - X_k - X_k(I - AX_k)) \\
&= A^{-1} - X_k - X_kI + X_kAX_k \\
&= A^{-1} - 2X_k + X_kAX_k \\
&= (I - X_kA)(A^{-1} - X_k) \\
&= (A^{-1} - X_k)A(A^{-1} - X_k) \\
&= E_kAE_k
\end{aligned}
$$

b) The comparison of Newton's method and np.linalg.inv() are shown below:





From the plot above, we can see that the norm of the estimated matrix increases drastically along with the runtime of the algorithm. This might be because it takes more iteration to solve the algebra system and the error accumulates over time (especially for scipy.linalg.inv()).

Q2):

a) One way that we can efficiently calculate the value of d_n is to take the last row of the inverse of A and matrix multiply by $b + f$ to obtain the value of d_n, since $b + f$ is constant except the last row, then we can treat this problem as a scaler root finding problem, hence speed up the calculations.

d) By Newton's method:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Since the handout have mention that d_n and f_n have a linear relationship, we can just numerically calculate f'(f_n) with formula $f'(x) = \frac{f(x+h) - f(x)}{h}$ and then plug it in newton's method

f) The result for different method are shown below:

dn/fn/runtime

size	Newton's method	fsolve	Brent's method
32	3.6158e-17 / 3.8781e-01 / 2.99e-01ms	0.e+00 / 3.8781e-01 / 6.5050e-01ms	3.6158e-17 / 3.8781e-01 / 6.0635e+00ms
64	1.8289e-17 / 3.8138e-01 / 2.9190e-01ms	0.e+00 / 3.8138e-01 / 6.3070e-01ms	0.e+00 / 3.8138e-01 / 3.1130e-01ms
128	1.8396e-17 / 3.7818e-01 / 3.3220e-01ms	0.e+00 / 3.7818e-01 / 6.843e-01ms	0.e+00 / 3.7818e-01 / 3.506e-01ms
256	9.2248e-18 / 3.7659e-01 / 3.8160e-01ms	-9.2248e-18 / 3.7659e-01 / 8.205e-01ms	-9.2248e-18 / 3.7659e-01 / 1.6616e+00ms
512	3.6953e-17 / 3.7579e-01 / 5.285e-01ms	0.e+00 / 3.7579e-01 / 9.584e-01ms	1.8477e-17 / 3.7579e-01 / 6.395e-01ms
1024	-1.8490e-17 / 3.754e-01 / 6.368e-01ms	0.e+00 / 3.754e-01 / 1.2755e+00ms	0.e+00 / 3.754e-01 / 6.8490e-01ms
2048	2.7746e-17 / 3.752e-01 / 9.8670e-01ms	-9.2488e-18 / 3.752e-01 / 1.9256e+00ms	-9.2488e-18 / 3.752e-01 / 1.4099e+00ms
4096	-1.8501e-17 / 3.7510e-01 / 1.5835e+00ms	0.e+00 / 3.7510e-01 / 3.1719e+00ms	0.e+00 / 3.7510e-01 / 1.7341e+00ms
8192	-9.2244e-18 / 3.7503e-01 / 3.8791e+00ms	-9.2244e-18 / 3.7503e-01 / 7.7339e+00ms	-9.2244e-18 / 3.7503e-01 / 5.2408e+00ms
16384	3.7324e-17 / 3.7510e-01 / 7.7829e+00ms	0.e+00 / 3.7510e-01 / 1.5738e+01ms	0.e+00 / 3.7510e-01 / 8.3199e+00ms
32768	0.e+00 / 3.7434e-01 / 1.4171e+01ms	0.e+00 / 3.7434e-01 / 3.5892e+01ms	0.e+00 / 3.7434e-01 / 1.6434e+01ms
65536	3.6151e-17 / 4.3385e-01 / 3.9593e+01ms	1.2050e-17 / 4.3385e-01 / 7.9291e+01ms	1.2050e-17 / 4.3385e-01 / 4.0713e+01ms

We observe that as n gets larger, the error roughly stays the same as n grew larger. However, compare to Newton's method, fsolve and Brent's method gives us a more accurate result. For me, I would use fsolve since it provides an optimal point between runtime and accuracy compare to the other two methods.

g) The exact value for fn appears to be 0.39.

Q3):

a) The standard form of method 2 is shown below:

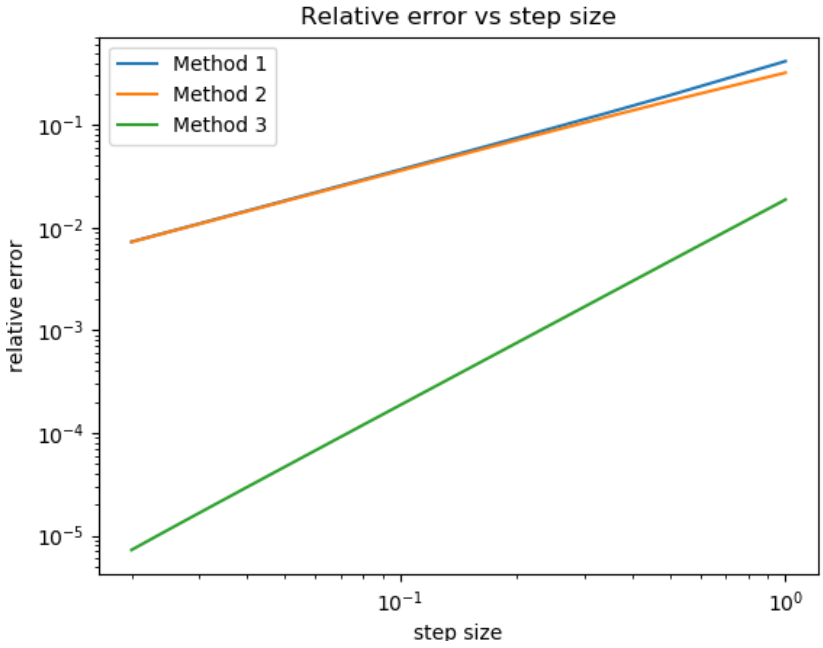
$$\begin{bmatrix} S_{(ti+1)} - S_{(ti+1)} + \frac{h\beta}{N} S_{(ti+1)} I_{(ti+1)} \\ E_{(ti+1)} - E_{(ti)} - \frac{h\beta}{N} S_{(ti+1)} I_{(ti+1)} + \omega h E_{(ti+1)} \\ I_{(ti+1)} - I_{(t)} - \omega h E_{(ti+1)} + h\gamma I_{(ti+1)} \\ R_{(ti+1)} - R_{(ti)} - h\gamma I_{(ti+1)} \end{bmatrix}$$

And its Jacobian is:

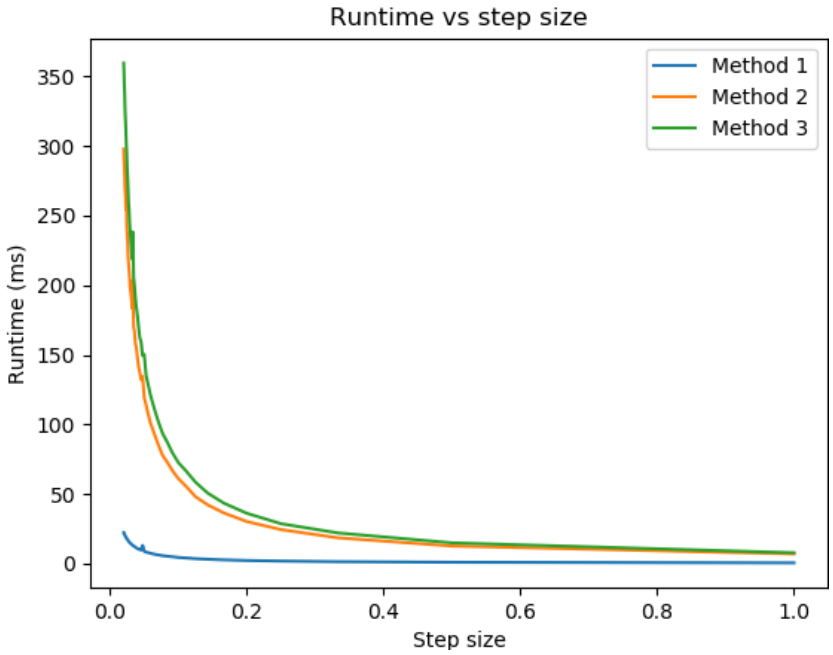
$$\begin{bmatrix} 1 + \frac{h\beta}{N} I_{(ti+1)} & 0 & \frac{h\beta}{N} S_{(ti+1)} & 0 \\ -\frac{h\beta}{N} I_{(ti+1)} & 1 + \omega h & \frac{h\beta}{N} S_{(ti+1)} & 0 \\ 0 & -\omega h & 1 + h\gamma & 0 \\ 0 & 0 & -h\gamma & 1 \end{bmatrix}$$

Since method three is similar to method two, its Jacobian matrix is just method two's Jacobian matrix with h replace to h/2.

c) The following plot illustrate the relative error and runtime with relation to step size:

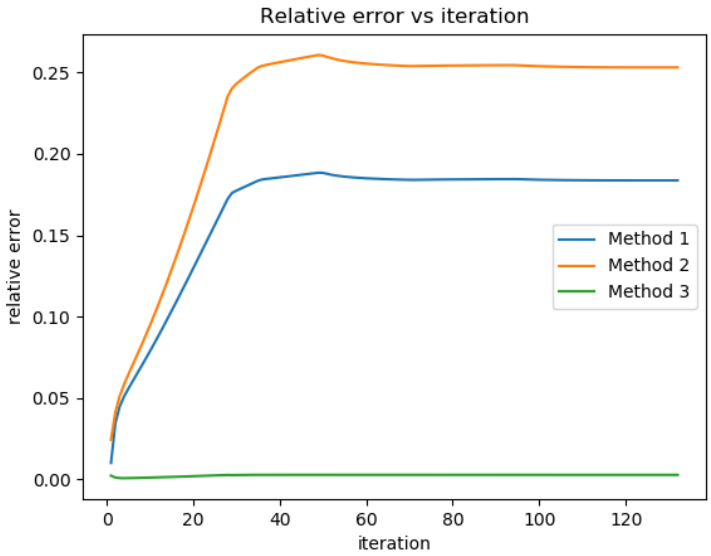


We can see as the step size increases, the relative error for the result also increases “linearly” in the loglog plot which agrees with our theoretical convergence rate since the slope in the plot is approximately the same is the theoretical value. The error increase along with the step size is because we are using our approximation over a larger time step, therefore the result might be in accurate. The next plot shows the runtime of each algorithm with different sizes of n:



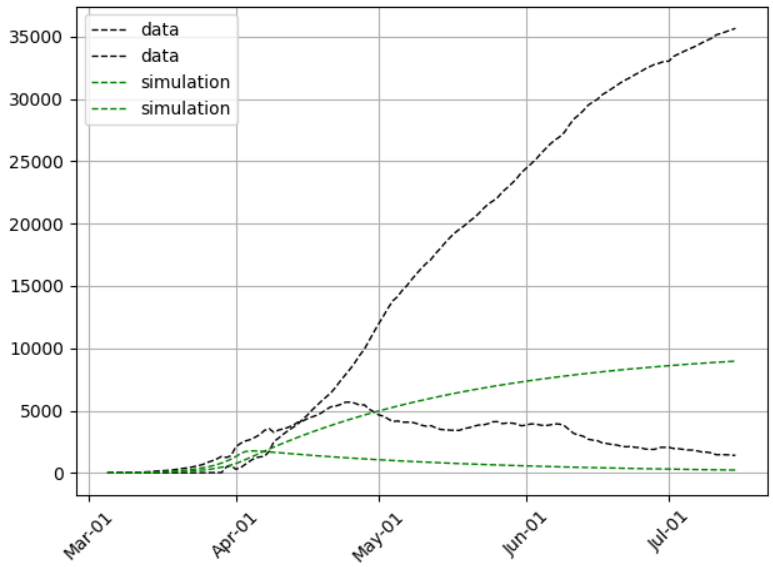
As we can see with the increase of step sizes, the runtime of all the methods drops significantly, which is what we expected, since smaller step sizes will take more iteration to finish. Notice that method 2 and 3 take significantly more time to finish when step sizes are small. This happens since we are try to solve a nonlinear system in each step.

d) The relative error for each iteration is shown in the plot below:



The plot above only uses the last component of simulated solution and true solution to calculate relative error. The relative error in the plot above increases as the number of iteration increases, this is due to numerical errors (truncate and rounding) in each step of the calculation accumulate over time. Notice that the relative error of method 3 is the smallest amount other methods, this is likely cause by the method 3 has the most accurate estimation for each iteration.

e) In this question I want to explore what happen if wears protection gears such as masks at the very beginning given the initial condition. To implement this condition, I set the beta factor to 0.23328388 for all the beta factor in beta list except for the first element in the list. The following plot shows the prediction:



We can see that with lower beta values, the number of infected cases reduced drastically, and the pandemic will end far sooner than what we currently have.