Math3316 Project 3

Due: December 5, 2018

Pr.1. Nonlinear equation solvers: Complete the python script p1_nonlinear.py to solve $f(x) = 0$ for the given functions. You need to implement three solvers:

(1) Bisection

(2) Newton's method (Note that only $f(x)$ is given, so you need to find $f'(x)$ manually and pass it as an argument to your code that implements the Newton's method)

(3) Quasi-Newton's method (It is similar to Newton's method, but you do not need an explicit derivative, instead, you use finite difference to approximate the derivative. Your code should implement FFD, BFD, and CFD, using CFD as the default.)

For each method, you need to return a single tuple $(x, k, f(x))$, where x is the solution, k is the total number of iterations to reach the solution, and $f(x)$ is the value of the function evaluated at the solution. Print out the intermediate steps as well as the final step results using a format similar to the following (note your functions/values may be different)

```
--------- Newton's method ----------
k=0,  x=10,  f(x)=900.0
...
k=5,  x=4.641589428638087,  f(x)=3.8458739865632197e-05
k=6,  x=4.641588833606049,  f(x)=4.959588295605499e-12
k=7,  x=4.641588833605972,  f(x)=2.842170943040401e-14
Newton converged in 7 steps: x=4.641588833605972, f(x)=2.842170943040401e-14

--------- Quasi Newton's method,  FD scheme =CFD ----------
k=0,  x=10,  f(x)=900.0
...
k=5,  x=4.641589855595351,  f(x)=6.605429565809118e-05
...
k=7,  x=4.641588833606032,  f(x)=3.865352482534945e-12
k=8,  x=4.641588833605972,  f(x)=2.842170943040401e-14
Quasi-Newton(CFD) converged in 8 steps: x=4.641588833605972, f(x)=2.842170943040401e-14

--------- Quasi Newton's method,  FD scheme =BFD ----------
k=0,  x=10,  f(x)=900.0
...
k=5,  x=4.64169627091094,  f(x)=0.006944160436049174
k=6,  x=4.641582836229799,  f(x)=-0.0003876281574122231
k=7,  x=4.641589168555349,  f(x)=2.164879828114863e-05
...
k=13,  x=4.641588833605982,  f(x)=6.536993168992922e-13
k=14,  x=4.641588833605971,  f(x)=-2.842170943040401e-14
Quasi-Newton(BFD) converged in 14 steps: x=4.641588833605971, f(x)=-2.842170943040401e-14
```

The functions used to test your solvers are

$$f(x) = x^9 - 8x^6 + 5x + 10, \qquad x \in [-1, 1] \tag{1}$$

$$f(x) = (x+1)^4 - xe^{\sin(x)} + 5x - 8, \qquad x \in [-3, 3] \tag{2}$$

$$f(x) = e^{-x^2} + x^3 - 100, \qquad x \in [-10, 10] \tag{3}$$

For each function, the interval for bisection is given. For Newton and Quasi-Newton, use one of the end points of the given interval as the initial (as specified in the script).

Pr.2. Finite differences:

Complete the p2_FD_plot.py script to plot the forward, backward, and central finite difference (FD) of the first order, second order, and the third order derivatives of any input smooth function $f(x)$.

The FD schemes for computing the 3rd order derivative to be used in your code are listed below:

$$(FFD) \qquad f'''(x) = \frac{1}{h^3}(-f(x) + 3f(x+h) - 3f(x+2h) + f(x+3h))$$

$$(BFD) \qquad f'''(x) = \frac{1}{h^3}(-f(x-3h) + 3f(x-2h) - 3f(x-h) + f(x))$$

$$(CFD) \quad f'''(x) = \frac{1}{2h^3}(-f(x-2h) + 2f(x-h) - 2f(x+h) + f(x+2h))$$

Your python functions should have a keyword argument for the exact derivatives, i.e., when an analytic derivative of any order is available and passed via this keyword argument, you need to plot this exact derivative in addition to the FFD, BFD, and CFD.

The two functions used to test your code are [1]

$$f(x) = \cos(x^2 - x) \tag{4}$$

$$f(x) = x * s_{igmoid}(x) = x * \frac{1}{1 + e^{-x}}. \tag{5}$$

Produce plots to show the improving approximations of FD (to the exact derivative) when h is reduced. You need to plot 9 figures for each function (for each fixed h, there are 3 figures, each order of derivative needs one plot; and h takes 3 different values). There should be 18 figures in total.

Your plots should have appropriate legends, line styles, and titles to make them understandable, similar to the ones in Figure 1.

Note that when implemented correctly, the curves for the three FDs should become closer to each other as h becomes smaller. If the exact derivative is correctly computed (manually by you) and passed to the function, the three FD curves should converge to the exact derivative when h becomes very small, the four curves should become hard to distinguish by eyes (although there are still numerical difference as printed out in the test code).

[1] Function (5) has an interesting name — the Swish function. It is very recently proposed by researchers at Google Brain, they use (5) as an improved activation function for training deep neural networks. Interested students may look at this publication https://arxiv.org/pdf/1710.05941.pdf.

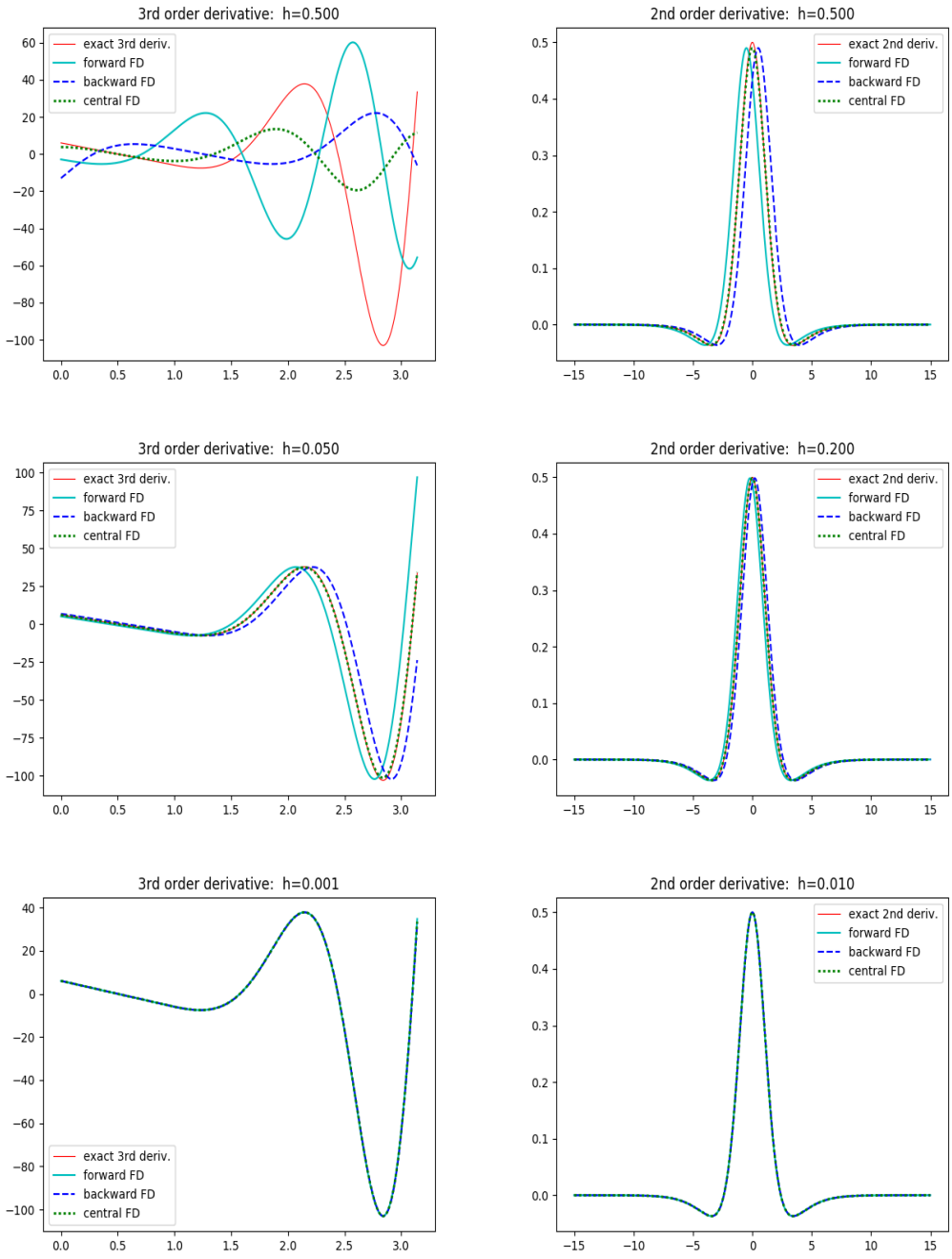


Figure 1: Left: 3rd order FDs for Function (4); Right: 2nd order FDs for Function (5)

Pr.3. The data contained in the file `EXP_LS_data_pert.txt` are obtained from a minor perturbation of the following exponential function

$$y = f(x) = e^{c_0 x \sin(x) + c_1 \cos(x^2) + c_2 x + c_3 x^2}.$$

The first column in the file are the x coordinates and the second column are the y coordinates of each data point (x, y). Apply least squares method to find out what the coefficients are.

Since this is not directly a linear least square problem, you need to first transform it into a linear least squares problem. This can be done by noticing that

$$\log(y) = c_0 x \sin(x) + c_1 \cos(x^2) + c_2 x + c_3 x^2.$$

So your least squares fitting should be performed on the data $(x, \log(y))$, and the fitting basis functions should be $\{\sin(x), \cos(x), x, x^3\}$. That is, you get a least squares equation $Ax = b$ from the $n + 1$ "interpolation" equations

$$c_0 x_i \sin(x_i) + c_1 \cos(x_i^2) + c_2 x_i + c_3 x_i^2 = \log(y_i) \tag{6}$$

for all the data points (x_i, y_i), $i = 0, 1, 2, 3, \cdots, n$. Writing these out as an over-determined matrix equation you get the least square problem $Ax = b$. You should first figure out what the A matrix is. It should be of dimension $(n + 1) \times 4$. The x and b vectors clearly should be

$$x = \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix}, \qquad b = \begin{bmatrix} \log(y_0) \\ \log(y_1) \\ \vdots \\ \log(y_n) \end{bmatrix}.$$

After you construct the A and b, you can call the `lstsq()` function from either `numpy.linalg` or `scipy.linalg` to solve for the coefficients.[2]

Complete the `p3_LS_exp.py` file to solve this problem.

Your code should produce a run log that reports the coefficients c_0, c_1, c_2, c_3, and produce a plot showing the original data and the least square fit, similar to Figure 2. (For the plot, be careful that the least square solution you get may be $\log(y)$, so you need to transform it back to y.)

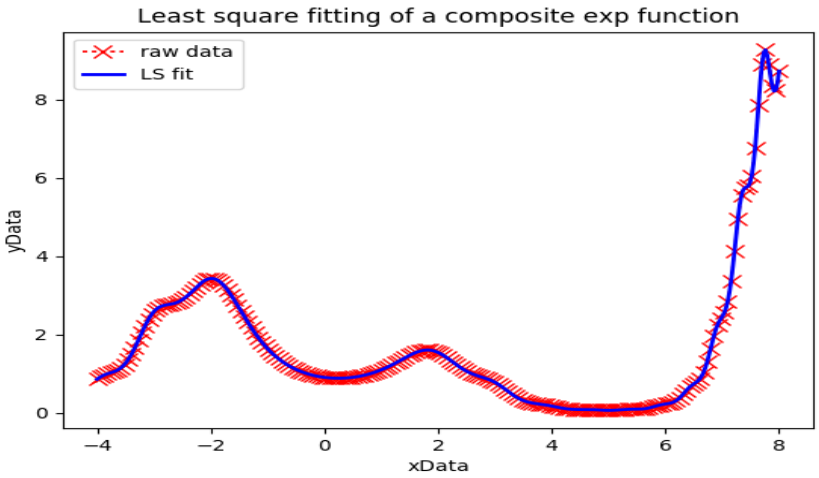


Figure 2: Least square fitting of an exponential function. (The fitting should be tight, because the data points are extracted from minor perturbations of an exponential function of form (6).)

[2]Hint: The i-th row of A should be $[x_i \sin(x_i),\ \cos(x_i^2),\ x_i,\ x_i^2]$, you can construct A column by column using numpy.

Pr.4. Several composite Simpson's rules for evaluating $\int_a^b f(x)dx$ are listed below,

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \frac{h}{3}\sum_{j=0}^{K-1}\left(f(x_{2j})+4f(x_{2j+1})+f(x_{2(j+1)})\right)\\ &= \tfrac{h}{3}\left(f(x_0)+4\textstyle\sum_{j=0}^{K-1}f(x_{2j+1})+2\sum_{j=1}^{K-1}f(x_{2j})+f(x_{2K})\right), \quad (n=2K)\end{aligned}$$

$$\begin{aligned}\int_a^b f(x)dx &\approx \frac{3h}{8}\sum_{j=0}^{K-1}\left(f(x_{3j})+3f(x_{3j+1})+3f(x_{3j+2})+f(x_{3(j+1)})\right)\\ &= \tfrac{3h}{8}\left(f(x_0)+3\textstyle\sum_{j=0}^{K-1}\left(f(x_{3j+1})+f(x_{3j+2})\right)+2\sum_{j=1}^{K-1}f(x_{3j})+f(x_{3K})\right), \quad (n=3K)\end{aligned}$$

$$\int_a^b f(x)dx\approx \tfrac{h}{48}\left(17f(x_0)+59f(x_1)+43f(x_2)+49f(x_3)+48\textstyle\sum_{i=4}^{n-4}f(x_i)+49f(x_{n-3})+43f(x_{n-2})+59f(x_{n-1})+17f(x_n)\right),$$

where $h=\frac{b-a}{n}$; $x_j=a+jh,\ j=0,...,n$.

For the first rule (rule 1/3), the formula says that you multiply $f(x)$ evaluated at even numbered nodes (except x_0 and x_n) and at odd numbered nodes by 2 and 4, respectively. Note $n=2K$, so at the beginning of your function, you should make n even –by adding 1 if necessary.

For the second rule (rule 3/8), the formula says that you multiply $f(x)$ evaluated at $3j$ nodes (except x_0 and x_n) and on $3j+1$, $3j+2$ numbered nodes, by 2 and 3, respectively. Note $n=3K$, so at the beginning of your function, you should make n to be a multiple of 3 (by adding 1 or 2 to the input n if necessary).

For the third rule (rule 1/48), the n can be any integer no smaller than 7.

Implement the above rules by completing the file `p3_simpson.py`. For this problem, the n input to the function is fixed, i.e., the quadrature rules are non adaptive. With increasing n the error for all quadrature rules should, with the three Simpson's rules performing much better than the mid-point rule and the trapezoidal rule. (Make sure your N should by multiples of 2 and 3 for the 1/3 and 3/8 rules, respectively, otherwise your errors likely won't decrease much.)

Figure 3 shows two sample plots.

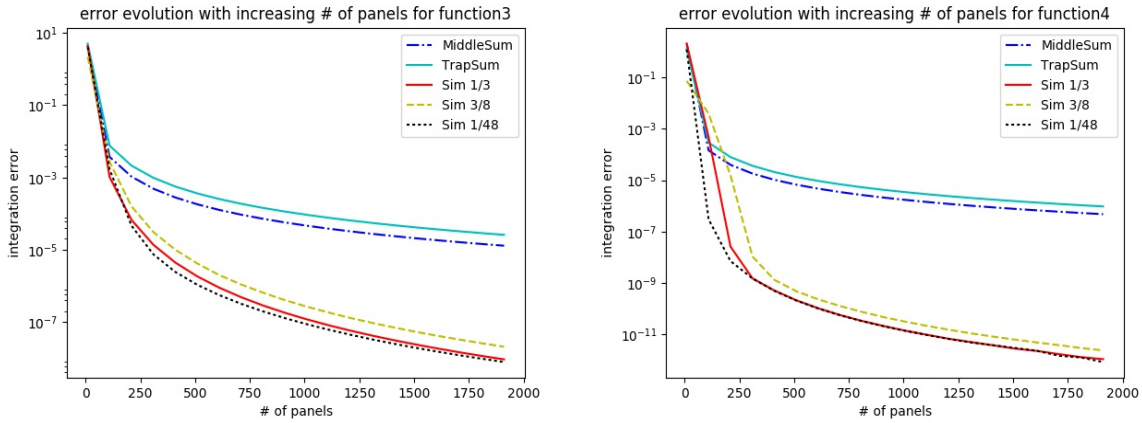


Figure 3: Error plots for the 3rd and the 4th functions in `p3_simpson.py`.

Pr.5. Implement the adaptive composite trapezoidal rule, for which the number of panels n should be determined by a specified tolerance input to the function. The procedure is to use $n = 2^k$ equi-length panels at the k-step iteration, starting with $k = 0$, and stopping when the difference between the approximate integral values of two adjacent iterations is below the given tolerance.

Denoting the integral value computed at the k-th step (i.e., using $n = 2^k$ panels, which means $h = \frac{b-a}{2^k}$ at the k-th step) as I_k, then the updating formula for I_k to approximate $\int_a^b f(x)\,dx$ is

$$\begin{aligned} I_{k+1} &= \frac{1}{2}I_k + \frac{b-a}{n}\sum_{j=0}^{\frac{n}{2}-1} f(x_{2j+1}), \qquad \text{(I.e., only need to sum over } \tfrac{n}{2} \text{ new nodes out of } n+1 \text{ total nodes)} \\ &= \frac{1}{2}I_k + h\sum_{j=0}^{2^{k-1}-1} f(a+(2j+1)h). \end{aligned} \tag{7}$$

If you start with $k = 0$, then you start with $h = \frac{b-a}{2^0} = b - a$ and $I_0 = \frac{h}{2}(f(a) + f(b))$; then you iterate (7) for $k = 0, 1, 2, 3, ...$

If you start with $k = 1$, then you start with $h = \frac{b-a}{2^1} = \frac{b-a}{2}$ and $I_1 = \frac{h}{2}(f(a) + 2f(a+h) + f(b))$; then you iterate (7) for $k = 1, 2, 3, ...$

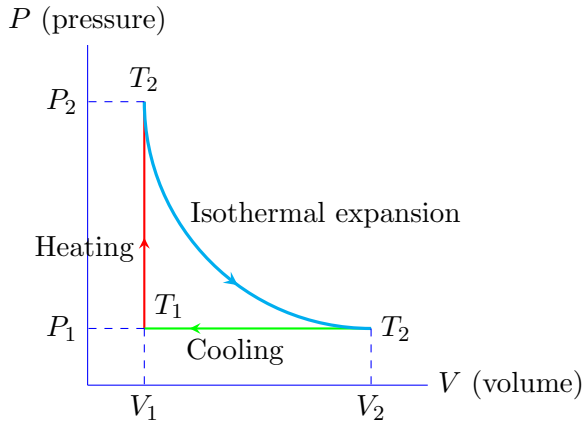
No matter which k you start from, your h at the k-th iteration (7) is $h = (b-a)/2^k$, which is half of the h used at the previous iteration.

Complete the provided test script `p5_trap_adaptive.py`.

Note, your code should be adaptive, that is, at the $(k+1)$-th iteration, the function values evaluated at the k-th should be reused. You cannot call a fixed n composite trapezoidal rule, first by $n = 2^k$, then by $n = 2^{k+1}$. Doing this would get no credit for this problem, because this problem is designed to train you on implementing a smarter algorithm that can reuse previously computed data for faster computations.

Note to pass the checks in the test code, your functions (both for this problem and for the previous problem) should return the integral value and the number of panels used.

Pr.6. The following figure shows the thermodynamic cycle of an engine. The efficiency of this engine for monatomic gas depends on the quotient of two absolute temperatures $\frac{T2}{T1}$ as in formula (8)



$$\boxed{\text{efficiency} = \frac{\ln(\frac{T2}{T1}) - (1 - \frac{T1}{T2})}{\ln(\frac{T2}{T1}) + (1 - \frac{T1}{T2})/(\gamma - 1)}}, \quad (8)$$

where $\gamma > 1$ is a constant associated with certain engine design.

You task is to write a code that applies a nonlinear solver developed for Problem 1 to solve for the $\frac{T2}{T1}$ associated with efficiency that ranges from 10% upto 80% (when γ is fixed).

Clearly, the $\frac{T2}{T1}$ value depends on not only the efficiency but also the γ. For the numerical simulations, you need to first fix the γ as a constant then loop the efficiency in `range(0.1, 0.81, 0.025)` and report their associated $\frac{T2}{T1}$ values. You only need to test two values of γ: $\gamma = 1.7$ and $\gamma = 5$.

For the nonlinear solvers, you can choose from bisection, Newton, or Quasi-Newton.[3] You can call bisection, use $[1.001, 10^6]$ as the enclosing interval to search for the root. (A too small interval may miss the root associated with higher efficiency values.)

There is a barely completed script `p6_engin_eqn.py` provided for this problem, complete this script.

You code should produce a run log similar to the following, which reports the $\frac{T2}{T1}$ values (this is the *root* that you solve for using a nonlinear solver) associated with all of the efficiency values in `range(0.1, 0.81, 0.025)`, for the two γ values specified.

```
efficiency=0.100,  T2/T1(1.7)=1.6461596286910254,  T2/T1(5)=1.3047338533006936
efficiency=0.125,  T2/T1(1.7)=1.8747521055451544,  T2/T1(5)=1.4023149174728506
efficiency=0.150,  T2/T1(1.7)=2.141344028396624,  T2/T1(5)=1.5111174409909183
efficiency=0.175,  T2/T1(1.7)=2.4540706442590383,  T2/T1(5)=1.6329422233122717
efficiency=0.200,  T2/T1(1.7)=2.8232591278213977,  T2/T1(5)=1.769965962872021
efficiency=0.225,  T2/T1(1.7)=3.2621500850476046,  T2/T1(5)=1.9248390167860003
efficiency=0.250,  T2/T1(1.7)=3.7879030690741664,  T2/T1(5)=2.100813942667268
.
.
efficiency=0.650,  T2/T1(1.7)=241.62805811733682,  T2/T1(5)=24.13910732521706
efficiency=0.675,  T2/T1(1.7)=415.4693086371629,  T2/T1(5)=32.656107873417064
efficiency=0.700,  T2/T1(1.7)=779.0767222337848,  T2/T1(5)=46.14503400139296
efficiency=0.725,  T2/T1(1.7)=1632.819859423925,  T2/T1(5)=68.93344515590948
efficiency=0.750,  T2/T1(1.7)=3958.5027615003405,  T2/T1(5)=110.73093215381824
efficiency=0.775,  T2/T1(1.7)=11664.163500986495,  T2/T1(5)=196.07490071224095
efficiency=0.800,  T2/T1(1.7)=44983.34237347994,  T2/T1(5)=397.3832683356137
```

You code should also plot the $\frac{T2}{T1}$ values found, similar to Figure 4. Note that the first subplot uses `plt.plot()`, which does not produce a clear plot, mainly because part of the array values far out-weight the others. The second subplot uses `plt.semilogy()`, which produces a far clearer plot using the same data. (If you have not used `plt.semilogy()` before, you need to look up its document. In summary, its syntax is the same as `plt.plot()`, but it plots the y-axis not with equal spacing but with a base-10 logarithmic spacing.)

[3]Hint: You need to figure out what the nonlinear equation $f(x) = 0$ is to solve for a root. Note that the efficiency (8) is a function of two variables $\frac{T2}{T1}$ and γ. If γ is fixed, the right-hand-side in (8) is reduced to a function of a single variable $\frac{T2}{T1}$. Once you figure out what the $f(x)$ is, you can apply a nonlinear solver to find its root, and this root would equal to $\frac{T2}{T1}$ corresponding to a fixed efficiency. Note that the $f(x)$ in the equation $f(x) = 0$ to solve for the root is **not** the function in the right-hand-side of (8), you need to go one step further to get this $f(x)$.

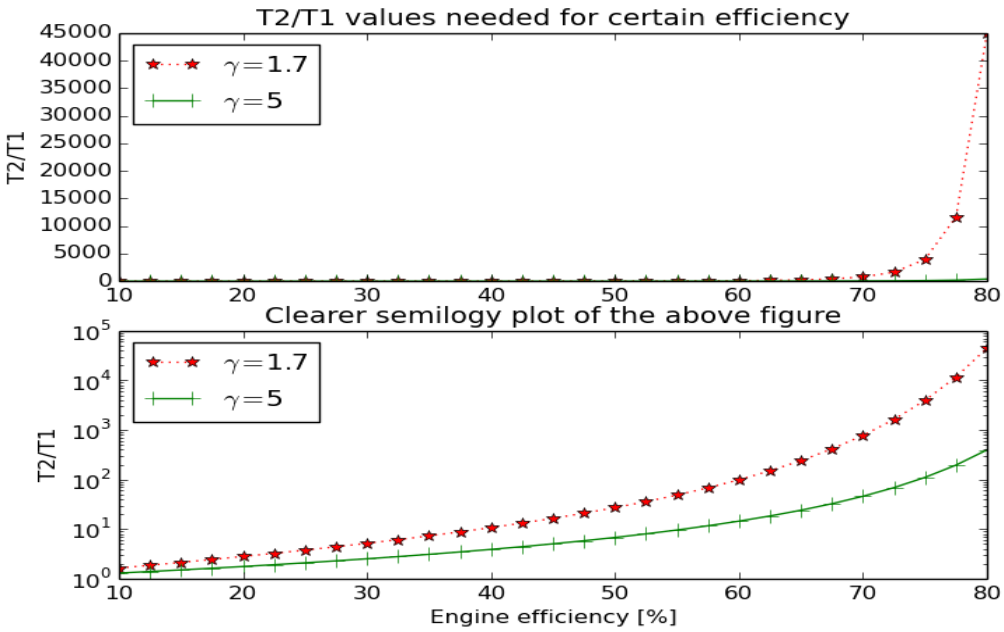


Figure 4: What intuition would you draw from the simulation? One main conclusion from the simulation as shown in this plot is that γ is important: If you can design an engine that has a higher γ value, it becomes easier for the engine to have higher efficiency, since it does not need $T2$ to be significantly higher than $T1$ to reach a certain efficiency threshold.

Pr.7. The forward Euler method and the 4th order Runge-Kutta method (RK4) are implemented in the `ODEs.py` script. The file `demo_ODEs.py` contains several examples on how to use the self-coded methods, it also shows how to call the ODE solver `odeint()` provided by the `scipy.integrate` module.

We apply the self-coded solvers to several realistic problems in `demo_ODEs.py`. They include

(i) Logistic ODE model (see https://en.wikipedia.org/wiki/Logistic_function), which has been used in many areas, such as in ecology to model population growth, and in economics to model diffusion of new products or new technology. The model is $\boxed{y'(t) = ry(1 - \frac{y}{K}),\ y(t_0) = y_0}$, where r and K are constants. This ODE has a theoretical solution $\boxed{y(t) = \frac{y_0 K}{y_0 + (K - y_0)e^{-rt}}}$, which can be used to verify the numerically computed solutions.
The results are shown in the first two plots of Figure 5. (They show that RK4 is efficient and accurate for this problem, while Euler becomes less and less accurate when h value is increased, as expected — due to accumulation of errors from earlier steps.)

(ii) The predator-prey model, also called the Lotka-Volterra model (see https://en.wikipedia.org/wiki/Lotka-Volterra_equations), which is frequently used to describe the dynamics of a biological systems of two species – predator and prey. The model is a first order 2×2 ODE system,

$$x'(t) = ax - bxy,$$
$$y'(t) = cxy - dy, \qquad \text{with initial conditions } x(0) = x_0, \quad y(0) = y_0,$$

where a, b, c, d are positive constants, the $x'(t)$ and $y'(t)$ represent the growth rates of the prey and predator populations, respectively.
As seen in the third plot in Figure 5, our self-coded RK4 solver performs as well as `odeint`. This is quite remarkable, considering that our RK4 code contains only a few lines, and we compare it with the established and well-regarded solver `odeint()` from the scipy package!

From this plot we again see that the Euler method needs a small h to stay relatively accurate, but eventually (when t becomes large enough) will become less accurate.

The fourth plot in Figure 5 shows a phase-space plot of the prey-predator model with a varying x_0 and a fixed y_0, using solutions from the RK4 solver. This phase-space plot shows that each solution of an autonomous Lotka-Volterra model contains a cycle: without being disturbed, the bio-system would self-regulate and repeat the cycle perpetually. This serves as an example showing that numerical solutions, when used properly, can provide insight or new understanding about the problem being studied.

(iii) The angular motion of a pendulum under gravity with friction and external forcing:

$$z''(t) + bz'(t) + c\sin(z) = e^{-tz}.$$

This is a rather difficult nonlinear ODE, for which Euler method has to use a very small h to be relatively accurate, while RK4 again performs well with moderate h, as seen in the fifth plot in Figure 5.

(iv) A second order nonlinear ODE

$$u'' + u - \sqrt{u} = 0, \qquad u(0) = \frac{\pi}{50}, \quad u'(0) = 0, \qquad t \in [0, 10\pi].$$

After solving this ODE, we use the solution $u(t)$ to construct a parametric curve with coordinates $(\frac{1}{u(t)}\cos(t), \frac{1}{u(t)}\sin(t))$, which is shown in the last plot in Figure 5. From the plot we again see `RK4()` has similar performance to that of `odeint()`, while the Euler method needs to use a very small h but still lacks behind in accuracy when t becomes larger.

Your task for this problem: First, study and understand the code `demo_ODEs.py` and figure out how to call the three solvers `odeint(), euler(), RK4()` to solve ODEs.

Second, write a code to solve the Van der Pol oscillator, which is a classical nonlinear second order ODE model (see https://en.wikipedia.org/wiki/Van_der_Pol_oscillator)

$$u'' - \gamma(1 - u^2)u' + u = A\sin(\omega t), \qquad u(0) = u'(0) = 0,$$

where $u(t)$ is the position coordinate, and γ, A, ω are positive constants. Solve this equation for the case $\gamma = 0.8, A = 0.5, \omega = \pi/4$, using the three solvers. (Note that you need to transform the second order ODE into a first order ODE system in order to solve it via first order solvers `odeint(), euler(), RK4()`.) Then plot the solutions you obtain. Your plot should be similar to the plot on the left in Figure 6. For the RK4, use $h = 0.1$, for Euler use $h = 0.01$ (i.e., same h as shown in this plot).

Third, figure out how to call the three solvers `odeint(), euler(), RK4()` to solve the following third order nonlinear ODE,

$$y^{(3)} + 2yy'' - (y')^2 = \frac{x}{y+5} - 1, \qquad y(0) = y'(0) = 0, \quad y''(0) = 1, \quad x \in [0, 12].$$

Then plot the solutions you obtain. Your plot should be similar to the plot on the right in Figure 6. For the RK4 and Euler methods, use the h values specified in this plot.

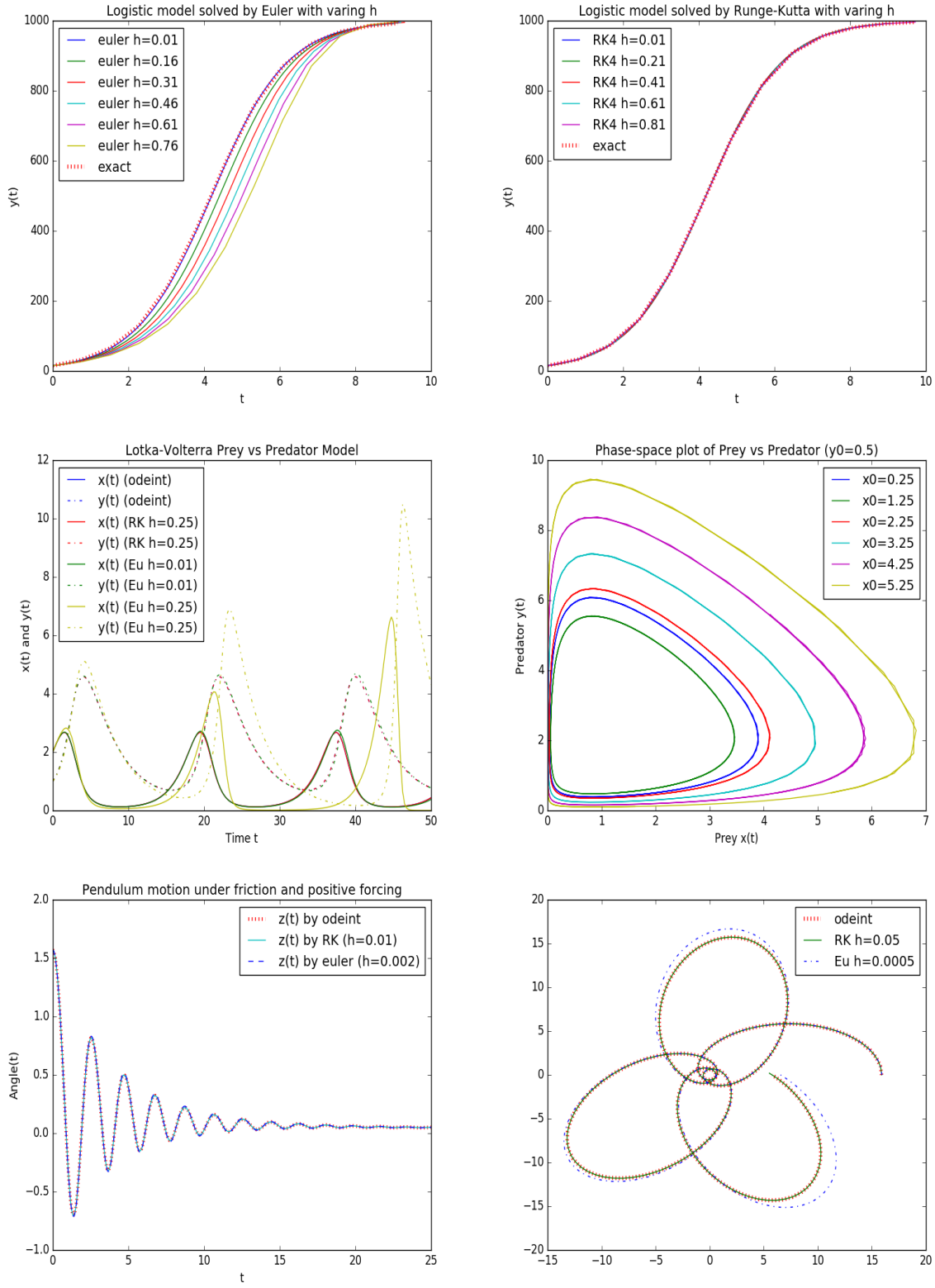
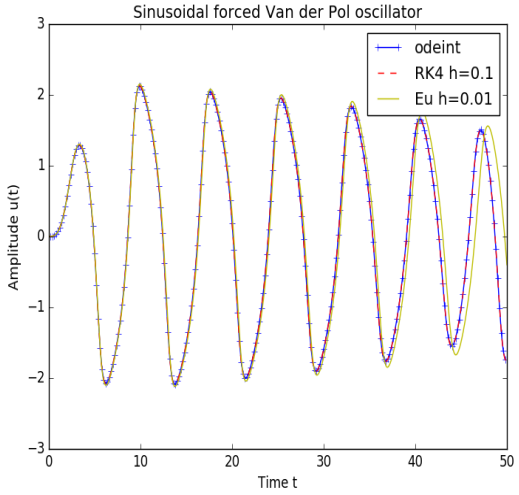


Figure 5: Results: using `odeint, euler, RK4` to solve some classical/realistic ODE problems.



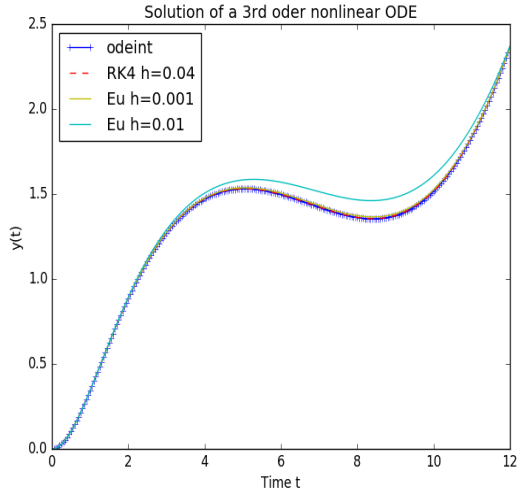


Figure 6: Left: Van der Pol oscillator. Right: A third order nonlinear ODE.

General comment:

Partially finished scripts are provided for each problem.
For the submission on or before the due date, the following items need to be submitted to Canvas:

1. Most importantly, your complete python script files for each problem (do not change the filenames as provided).
2. Log files and graph files produced by your complete python scripts. Endeavor to make them self-explanatory (so that the grader, instructor, or other people who have interest to check, can have easier time to see if your logs or plots make sense).