Homework 5: Computational Physics

Pablo Lopez Duque

October 14^{th}, 2020

1 Problem 1

There are three roots to the function $F(x) = e^x - x^4$. Determine all three solutions to $F(x) = 0$ using a bisection algorithm to within a tolerance of 10^{-6}. Hint: you will find the roots in the vicinity of -1, 1, and 9. Run your bisection algorithm on three different regions containing those points.

In order to determine the search intervals, we can plot $F(x)$ to avoid a random search. Figure 1 shows that there are 3 roots in the vicinity of points given in the hint.

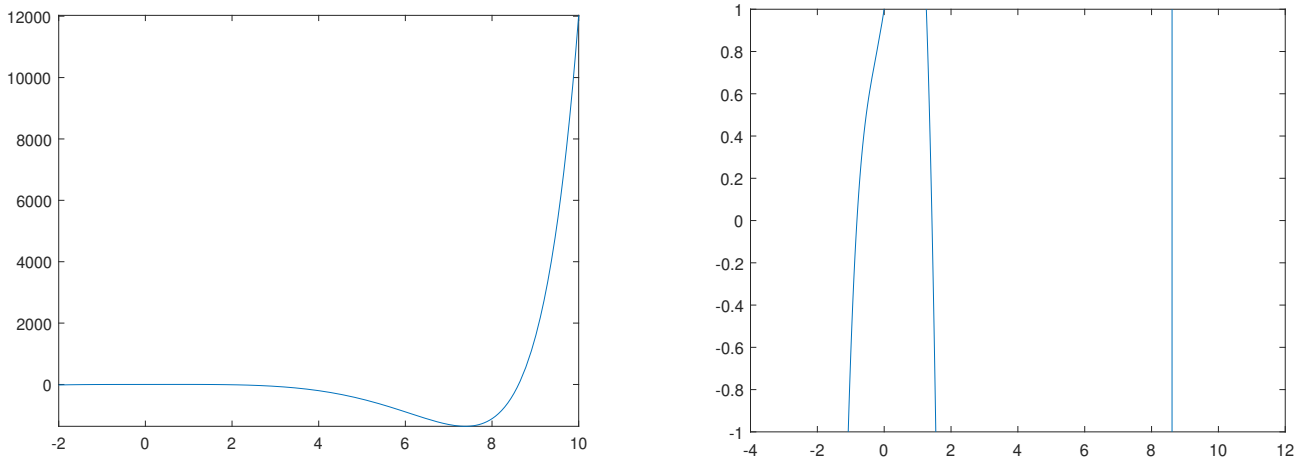

Figure 1: Left: $F(x)$ view in a large interval. Right: view of $F(x)$ to identify root search intervals. Clearly, $x \approx -1$, $x \approx 1$ and $x \approx 9$ are appropriate regions to analyze.

In the bisection algorithm, we first set an interval in which there is a change on the sign of the target function. Then, we compute the value at the mean of the two boundaries and narrow the interval in which the function still changes sign. Iteratively, we repeat the process until the value of the function reaches a specified tolerance. The code for this algorithm is included in the GitHub. First, we define a symbolic function that will let us evaluate the function at any point:

```
syms y real;
syms f(y);
f(y)=exp(y)-y^4;
%f(2)  %call for the function f(y) to be evaluated at y=2
```

Then, we define the parameters for our system and the variables where we will store the values of the function. Storage of the search intervals might be unnecessary, but in an overly cautiously approach, it does not hurt to have the values stored for cross checking.

```
%paramters and storage variables
ms=3;
a=[];
b=[];
x=[];
tic %for measuring the time
a0=7;  %change to search around different regions
b0=10; %change to search around different regions
x0=(a0+b0)/2;
x(1)=x0;
iter=1;
tol=1e-6;
```

Depending on the sign of the slope of $F(x)$ when the root is approached, we must select the boundaries in a specific order to make the implementation easier in later stages:

```
%vpa() evaluates a symbolic function numerically
if vpa(f(a0))< vpa(f(b0))
    a(1)=a0;
    b(1)=b0;
elseif vpa(f(a0))> vpa(f(b0))
    a(1)=b0;
    b(1)=a0;
```

Iteratively, we search for the root by computing the function at the average of the boundary values and selecting sub intervals where the function is still changing signs. We store the results and check the value by using the built-in function $fzero$:

```
while 1
    if abs(vpa(f(x(iter))))<tol
        break;
    elseif vpa(f(x(iter)))<0
        a(iter+1)=x(iter);
        b(iter+1)=b(iter);
    elseif vpa(f(x(iter)))>0
        a(iter+1)=a(iter);
        b(iter+1)=x(iter);
    end

   x(iter+1)=(a(iter+1)+b(iter+1))/2;
   iter=iter+1;
end
fr1=vpa(f(x(iter)))
r1=x(iter)
fprintf('check: ');
MLr1=fzero(@(y) exp(y)-y^4,8)
toc
```

As pointed out above, the if cases, selected the values in the boundaries in such a way that only one implementation is necessary inside the loop, which makes the code more efficient.
After looping through the 3 regions, that is selecting appropriate values for $a0$ and $b0$, we got the results shown below (elapsed time was calculated using tic-toc built-in function):

```
>> HW5_1alt

fr1 = 0.00000046539318136628083881483001422159
r1 = 8.6132
check:
MLr1 = 8.6132
Elapsed time is 2.350336 seconds.

fr2 = -0.00000072537446371143943280586959591687
r2 =  1.4296
check:
MLr2 = 1.4296
Elapsed time is 1.451865 seconds.

fr3 = -0.00000064351695369333617246825524422658
r3 =  -0.8156
check:
MLr3 = -0.8156
Elapsed time is 0.959342 seconds.
```

Which shows the approximate value of the roots up to a precision of 10^{-6} which was given by our tolerance.

2 Problem 2

Use Newton's method to find any $x*$ and $y*$ such that $F(x*, y*) = G(x*, y*) = 1$, where $F(x, y) = x^2 e^{-x^2} + y^2$ and $G(x, y) = \frac{x^4}{1+x^2y^2}$. Your method may use any existing functions in base matlab, any libraries in python, or should be contained in the STL of C++.

In order to have a good intuition of the result, it is often a good idea to plot both functions $F(x, y)$, $G(x, y)$ and the plane $z = 1$ in the same figure while showing different angles. Thus, one can approximately determine the answer and have a good cross checking value. Figure 2 shows different angles of the functions. In particular, the bottom right figure is helpful to identify the values at which all 3 intersect. Hence, the expect intersection points are close to $x \approx 1.2$ and $y \approx 0.8$ with all permutations of $+$ and $-$ signs for each component. That is, there are 4 solutions. This expectation is consistent with the values found by solving the system in Mathematica. The top images were generated in Matlab and bottom ones in Mathematica. Unfortunately, Matlab lacks an easy way to implement value dependent coloring on symbolic plotting.

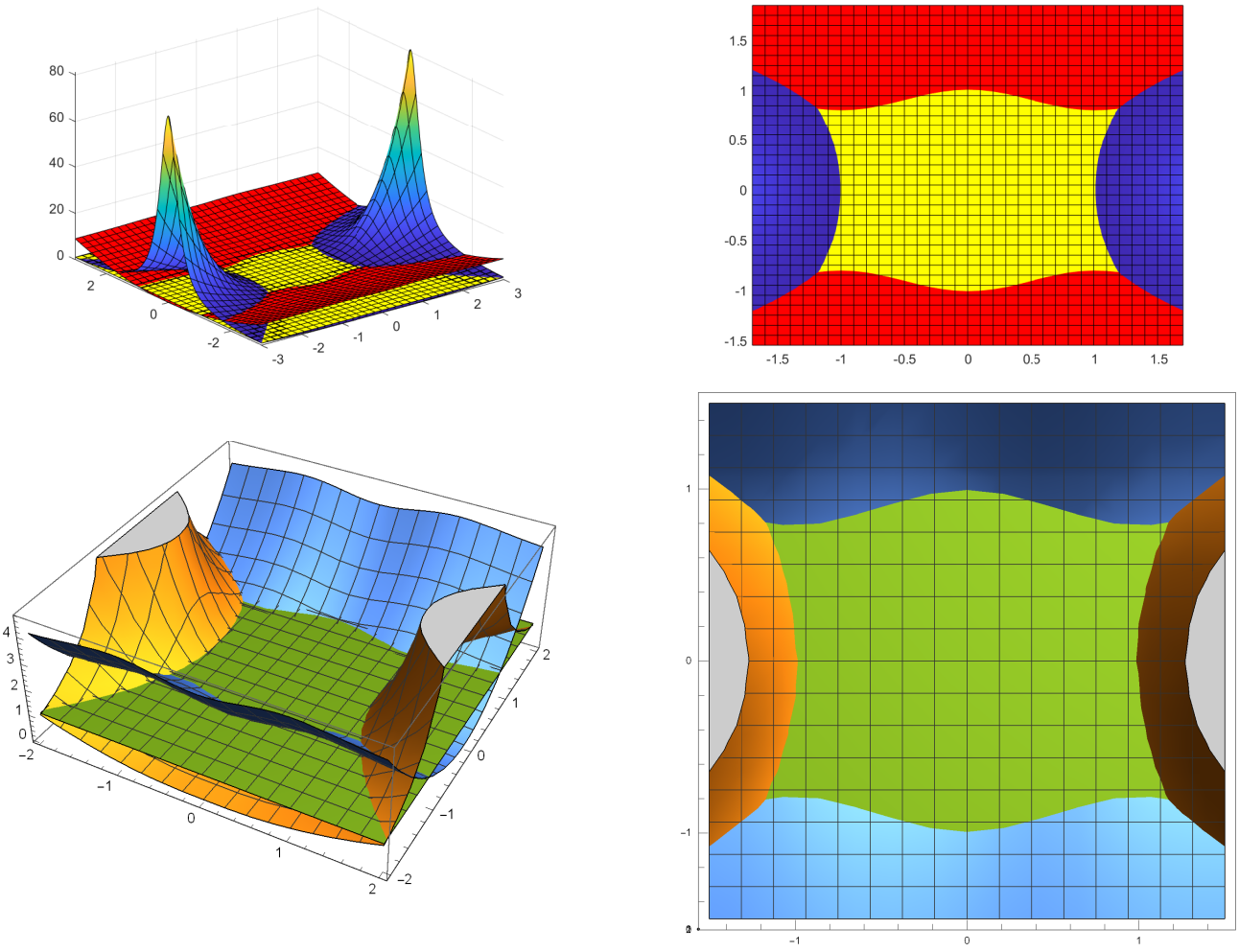

Figure 2: Top Left: Numerical and iterative solution for the largest eigenvector. Top right: Iterative solutions including error bars representing the error between the numerical and iterative solutions. Bottom center: Absolute error between the numerical and iterative solutions.

(a) **Self written code for Newton's Algorithm** One way of approaching this problem is by vectorizing

the one dimensional Newton's method. However, we must proceed with caution since the derivative term must be converted according to the particular application. In our case, we can define a $\mathbb{R}^2 \to \mathbb{R}^2$ function such that:

$$\vec{h}(\vec{r*}) = \vec{h}(\vec{r} - (\vec{r} - \vec{r*})) \approx \vec{h}(\vec{r}) - J(\vec{r})(\vec{r} - \vec{r*})$$

where $J(\vec{r})$ is the Jacobian matrix corresponding to the transformation that maps $\vec{h}(\vec{r*})$ into $\vec{h}(\vec{r})$.

So, whenever $\vec{h}(\vec{r*}) \approx 0$:

$$\Rightarrow \vec{r*} = \vec{r} - J^{-1}(\vec{r}) * \vec{h}(\vec{r})$$

If we define:

$$\vec{h}(\vec{r}) = \begin{bmatrix} F(\vec{r}) - 1 \\ G(\vec{r}) - 1 \end{bmatrix}$$

by using Newton's method we would be finding the solutions to the system of equations:

$$\begin{bmatrix} F(x,y) - 1 = 0 \\ G(x,y) - 1 = 0 \end{bmatrix}$$

which is equivalent to finding the intersection of both functions and the plane $z = 1$. The code for this implementation is included in the GitHub folder under $HW5\_2self.m$. First we use symbolic expressions to define the functions:

```
syms x y;
r = sym('x_%d',[2 1]);
F=@(r) r(1)^2*exp(-r(1)^2)+r(2)^2-1;
G=@(r) r(1)^4/(1+r(1)^2*r(2)^2)-1;
h=@(r) [F(r);G(r)];
J=@(r) jacobian(h(r),[r]);
Jinv=@(r) inv(J(r));
```

Then, we input the initial parameters for the system:

```
r0=[1;1];
rold=r0;
rnew=[0;0];
iter=1;
tol=1e-6;
```

and finally iterate until convergence, i.e. reaching the desired tolerance.

```
while(1)
    rnew=rold-vpa(subs(Jinv([x;y]),{x,y},{rold(1),rold(2)}))...
    ...*vpa(subs(h([x;y]),{x,y},{rold(1),rold(2)}));
    sx(iter)=(rnew(1))
    sy(iter)=(rnew(2))
    rold=rnew;

    iter=iter+1;
    if(vpa(subs(h([x;y]),{x,y},{rold(1),rold(2)}))<tol)
        break;
    end
end
```

By selecting points close to the values we found using the plots, we can find the intersection points. The output is:

```
%for r0=[1,1]
root =
   1.173977710152049833686324299335 3
  0.807868817013698922542477234788 93

 %for r0=[1,-1]
root =
   1.173977710152049833686324299335 3
  -0.807868817013698922542477234788 93

%for r0=[-1,1]
root =
  -1.173977710152049833686324299335 3
  0.807868817013698922542477234788 93

 %for r0=[-1,-1]
root =
  -1.173977710152049833686324299335 3
  -0.807868817013698922542477234788 93
```

(b) **Matlab built-in function vpasolve(). *Cross checking** In order to cross check the previous implementation, one can use vpasolve() that solves different types of systems of equations given an initial guess. Since, we have a general idea on where the intersection points should be, the initial guess is modified to get all 4 solutions.

```
syms x y
[S1x,S1y]=vpasolve(x^2*exp(-x^2)+y^2==1,(x)^4/(1+x^2*y^2)==1,[-1,-1])
[S2x,S2y]=vpasolve(x^2*exp(-x^2)+y^2==1,(x)^4/(1+x^2*y^2)==1,[-1,1])
[S3x,S3y]=vpasolve(x^2*exp(-x^2)+y^2==1,(x)^4/(1+x^2*y^2)==1,[1,-1])
[S4x,S4y]=vpasolve(x^2*exp(-x^2)+y^2==1,(x)^4/(1+x^2*y^2)==1,[1,1])
```

The output is the following:

```
>> HW5_2
S1x = -1.173977709810860026677511234204 8
S1y =-0.807868817139055504812586366043 07

S2x = -1.173977709810860026677511234204 8
S2y = 0.807868817139055504812586366043 07

S3x = 1.173977709810860026677511234204 8
S3y =-0.807868817139055504812586366043 07

S4x = 1.173977709810860026677511234204 8
S4y = 0.807868817139055504812586366043 07
```

Which confirms our previous results in (a) up to the specified tolerances. It must be noticed that the expectation is for built in solvers to be more effective than self written code since they include a wide range of optimizations or combined methods to solve the non-linear equations.

3 Problem 3

Exponential decay is common in many physically relevant situations, ranging from chemical reactions to nuclear decay, with $x(t) = Ae^{-t/\tau}$. In cases where two processes are occurring (e.g. a fast and a slow chemical reaction), a bi-exponential decay is a better model, with $x(t) = Ae^{-t/\tau_1} + Be^{-t/\tau_2}$. In this problem, you will attempt to fit the data attached in the class Teams page using both of these models.

There are several ways to implement the fitting in both Matlab and Python. By far, the easiest implementation is by using the Matlab built-in function *fit* our Python built-in function *curve_fit*. The code is included in the GitHub folder, but is also described here. First, we define the functions that we want to fit to the data.

```
clear;
fitEqn = fittype('a*exp(-t/tau)','independent','t');
fitEqn2 = fittype('a*exp(-t/tau1)+c*exp(-t/tau2)','independent','t');
startPoints = [0.1 30];
startPoints2 = [2 10 10 5];
```

Then, we must load the data, which is straightforward in Matlab:

```
data1=load('data_1.csv');
data1xvals=data1(:,1);
data1yvals=data1(:,2);
```

Finally, we need to define the options with the method and initial values we set up before and call the *fit* function.

```
options = fitoptions('Method', 'NonLinearLeastSquares','Start', startPoints,'TolFun',1e-8)
options2 = fitoptions('Method', 'NonLinearLeastSquares','Start', ...
    startPoints2,'TolFun',1e-8)
[d1f1, d1good1, d1out1] = fit(data1xvals,data1yvals,fitEqn,options);
[d1f2, d1good2, d1out2] = fit(data1xvals,data1yvals,fitEqn2,options2);

%For plotting the results:
f1=figure;
plot(d1f1,data1xvals,data1yvals)
hold on
plot(d1f2,'k',data1xvals,data1yvals)
hold off
```

This fit function uses the NonLinearSquares method which is a optimized version for non-linear functions in which the fit parameters are determined so that they minimizes the square error between data points and the fit 'prediction'.

For convenience in terms of comparison, the figures will include the fits for both types of fit functions.

(a) **Fit all three datasets to a single exponential model, $x(t) = Ae^{-t/\tau}$ with A and τ fitting parameters, and describe the method you used to find the fit values A and τ (including your initial conditions). Be sure to try a few initial conditions, to see if you find similar best fit parameters. Then, fit all three datasets to a bi-exponential model, $x(t) = Ae^{-t/\tau_1} + Be^{-t/\tau_2}$ with A, B, τ_1 and τ_2 fitting parameters. Again, describe the method and initial conditions you used, and be sure to use more than one set of initial conditions.**

Note: I merged parts (a) and (b) as it becomes easier to compare by putting both fits in the same plot.

Different initial conditions for each one of the datasets and each one of the fits are summarized in tables 1-3 below, which includes the coefficient of determination, R^2 in the labels.

Table 1: initial values the fit parameters of Eqn1$x(t) = Ae^{-t/\tau}$ and Eqn2= $x(t) = Ae^{-t/\tau_1} + Be^{-t/\tau_2}$

fit	parameter	Data 1 initial	Data 2 initial	Data 3 initial	Data 1 optimal	Data 2 optimal	Data 3 optimal
fitEqn1	A	1	1	1	0.7464	0.7167	0.4738
	τ	30	30	30	13.51	14.43	16.32
fitEqn2	A	1	10	2	0.5072	0.5548	0.08026
	B	30	3	1	0.4956	0.4664	0.4388
	$\tau 1$	10	1	10	1.978	1.852	1.831
	$\tau 2$	50	30	30	20.27	22.28	17.53

Figure 3 shows the fits for both equations corresponding to parameters in table 1. As it can be seen there fits are seemingly different for data 1 and data 2, but the distinction is weaker for data 3.

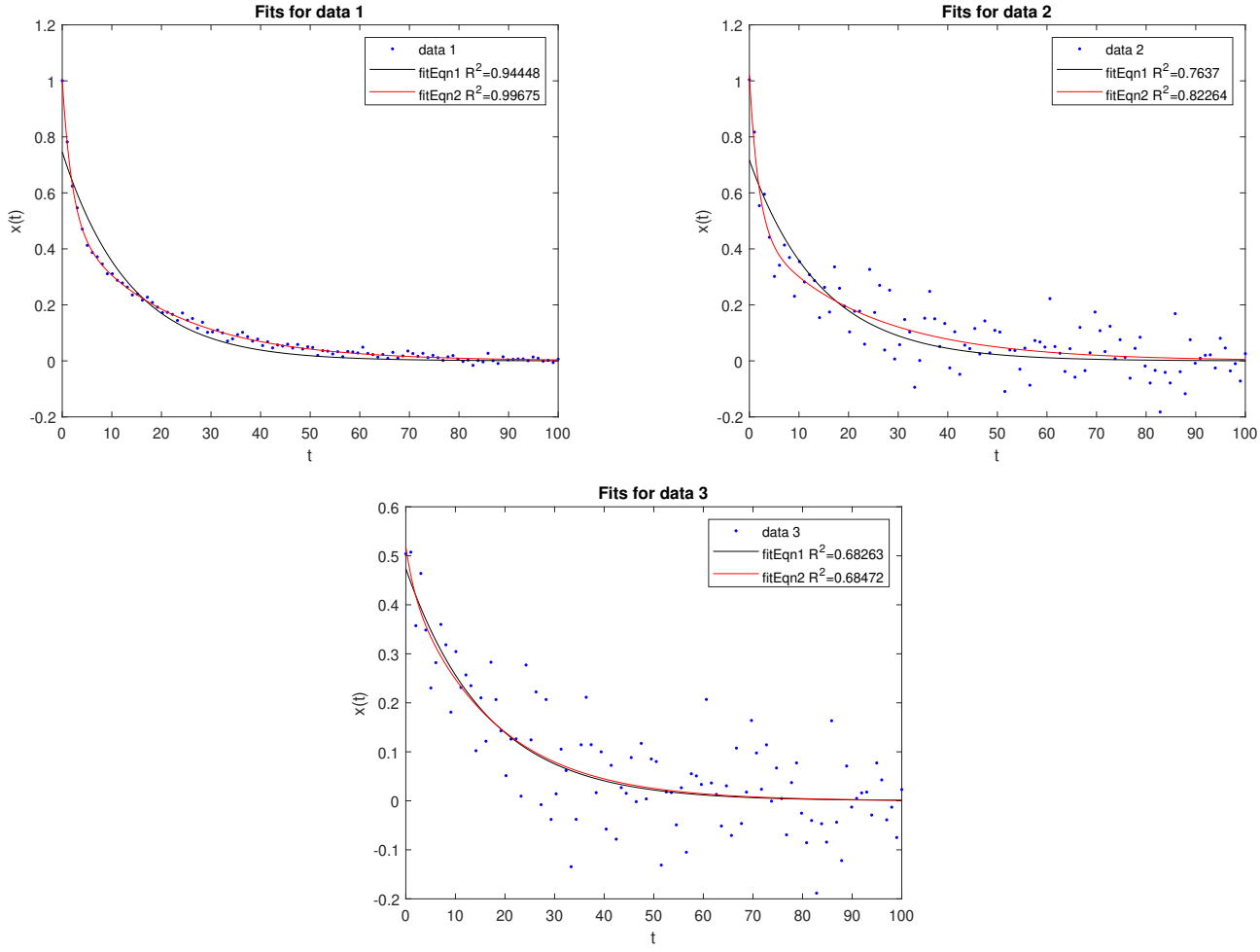

Figure 3: fits with functions Eqn1 and Eqn2 and parameters in table 1 for: Top Left: data 1. Top right:data 2. Bottom center: data 3.

Table 2: initial values the fit parameters of Eqn1$x(t) = Ae^{-t/\tau}$ and Eqn2= $x(t) = Ae^{-t/\tau_1} + Be^{-t/\tau_2}$

fit	parameter	Data 1 initial	Data 2 initial	Data 3 initial	Data 1 optimal	Data 2 optimal	Data 3 optimal
fitEqn1	A	1	1	1	0.7465	0.7167	0.4738
	τ	1	1	1	13.51	14.43	16.32
fitEqn2	A	1	1	1	0.5072	0.4664	0.2369
	B	1	1	1	0.4956	0.5548	0.2369
	$\tau 1$	1	1	1	1.978	22.28	16.32
	$\tau 2$	1	1	1	20.27	1.852	16.32

Figure 4 shows the fits for both equations corresponding to parameters in table 2. It can be seen that for both data 1 and data 2, there is no significant change in the fitting parameters. However, for data 3, picking the same initial values, leads to only a exponential function, that is both equations become equal, as the amplitudes for fitEqn2 become half of the fitEqn1 parameters.

Table 3: initial values the fit parameters of Eqn1$x(t) = Ae^{-t/\tau}$ and Eqn2= $x(t) = Ae^{-t/\tau_1} + Be^{-t/\tau_2}$

fit	parameter	Data 1 initial	Data 2 initial	Data 3 initial	Data 1 optimal	Data 2 optimal	Data 3 optimal
fitEqn1	A	30	1	1	0.7464	0.7167	0.4738
	τ	1	30	30	13.51	14.43	16.32
fitEqn2	A	30	10	2	0.4956	0.4664	0.4388
	B	1	3	1	0.5072	0.5548	0.08026
	$\tau 1$	50	1	10	20.27	22.28	17.53
	$\tau 2$	10	30	30	1.978	1.852	1.831

Figure 5 shows the fits for both equations corresponding to parameters in table 3. There is absolutely no difference from the values found for table 1, except that the coefficients are inversed.

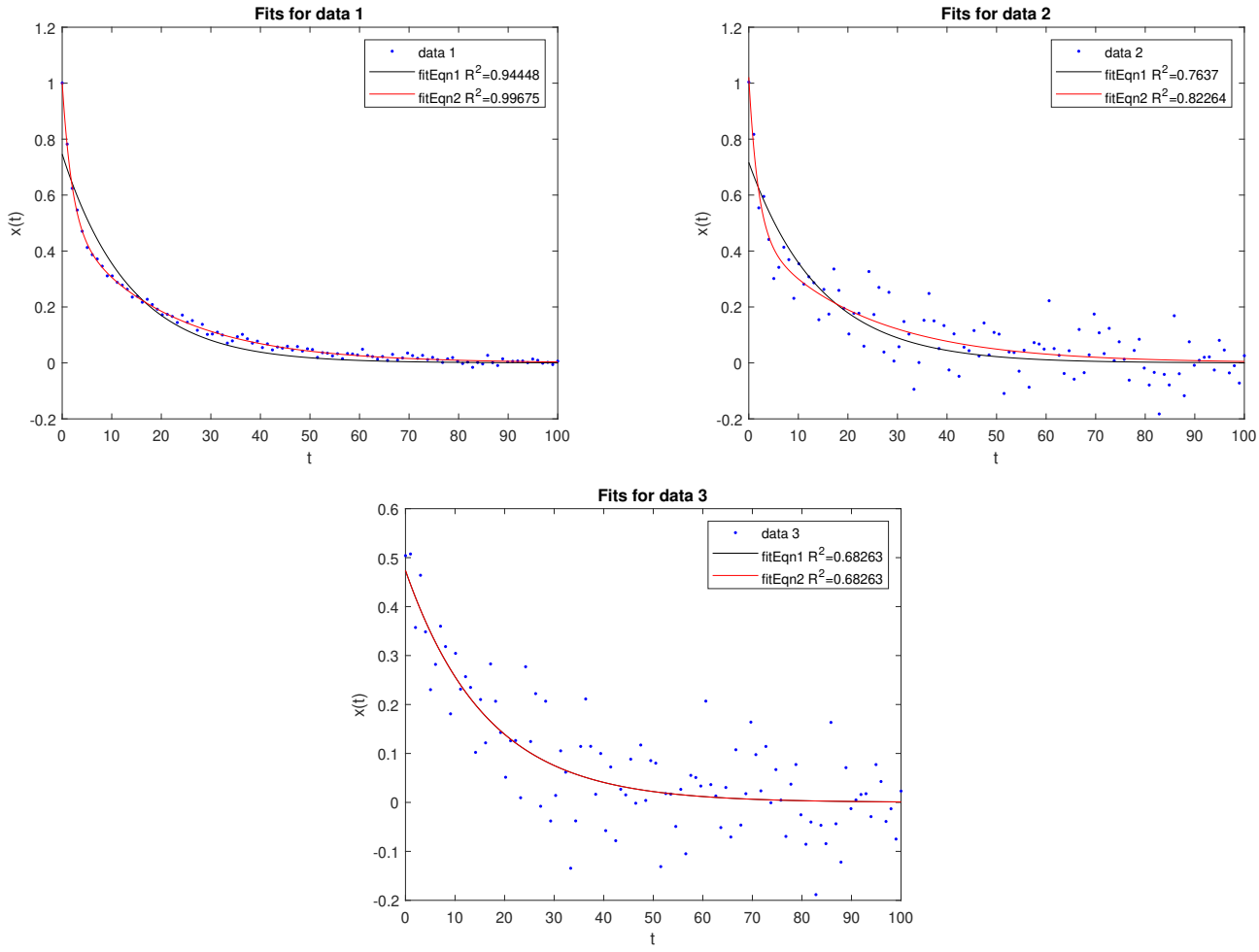

Figure 4: fits with functions Eqn1 and Eqn2 and parameters in table 2 for: Top Left: data 1. Top right:data 2. Bottom center: data 3.

(b) **Each dataset was drawn from either a single exponential or bi-exponential function (with added noise). Given the results from (a) and (b), which datasets do you think are single-exponential and which do you think are bi-exponential? Justify your conclusions.** Based on the values for the parameters, we can confidently say that both data 1 and data 2 correspond to a bi-exponential fit. For data 1, the fit coefficient of determination becomes almost 1 under this type of fit compared to 0.94 for the simple exponential. Also,from the trend of the points, it becomes clear that the bi-exponential fit corresponds better to the data. The same argument can be made for data 2. However, in this case, the R^2 becomes lower due to the noise. Still, the increment in R^2 together with the data trend, make it clear that it is also a bi-exponential fit. Another point to consider for both data 1 and data 2 is that the amplitudes for the exponential curves are similar in value, meaning that there are significant contributions from both curves. However, for data 3, the amplitude of one of the exponentials is much smaller, which hints that this is probably a simple exponential curve. Nevertheless, it is important to notice that the R^2 is so low in this case that we clearly have an underfit of the data, without adding more points, any arguments made based on these fits will be weak at best.

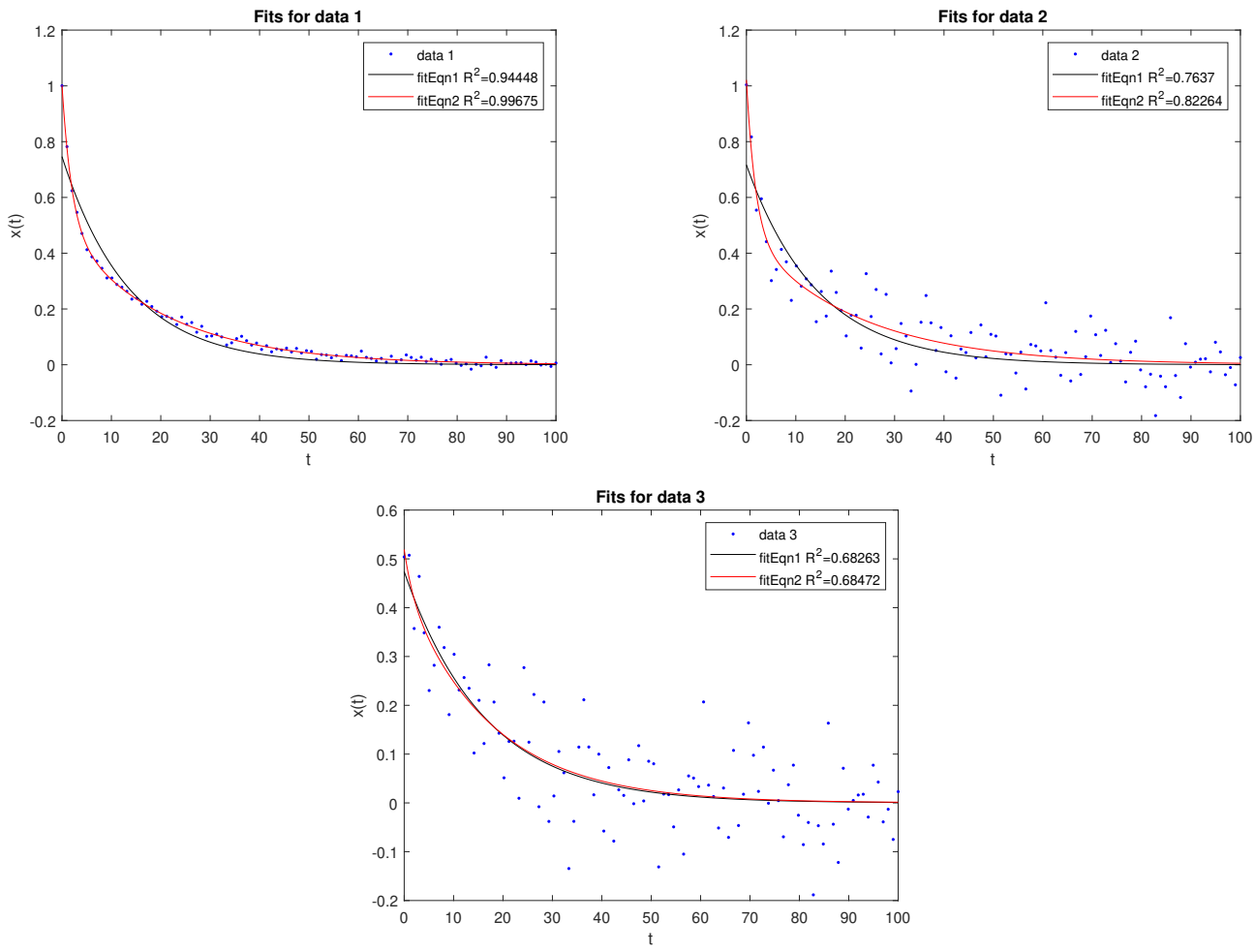

Figure 5: fits with functions Eqn1 and Eqn2 and parameters in table 3 for: Top Left: data 1. Top right:data 2. Bottom center: data 3.