1. a)

$$u^T v = \begin{bmatrix} u_1(x_1,\dots,x_n) & \cdots & u_n(x_1,\dots,x_n) \end{bmatrix} \begin{bmatrix} v_1(x_1,\dots,x_n) \\ \vdots \\ v_n(x_1,\dots,x_n) \end{bmatrix}$$

$$= u_1 \cdot v_1 + u_2 \cdot v_2 + \dots + u_n \cdot v_n = f(x)$$

$$\nabla(u^T v) = \nabla(f(x)) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \frac{\partial}{\partial x_1}(u_1 \cdot v_1 + \dots + u_n \cdot v_n) \\ \vdots \\ \frac{\partial}{\partial x_n}(u_1 \cdot v_1 + \dots + u_n \cdot v_n) \end{bmatrix}$$

We can know use the scalar valued function product rule for each entry:

$$\frac{\partial}{\partial x_1}\left[f(x_1,\dots,x_n) \cdot g(x_1,\dots,x_n)\right] = g(x_1,\dots,x_n) \cdot \frac{\partial}{\partial x_1}\left[f(x_1,\dots,x_n)\right] + f(x_1,\dots,x_n) \cdot \frac{\partial}{\partial x_1}\left[g(x_1,\dots,x_n)\right]$$

Thus,

$$D(u^T v) = \begin{bmatrix} \left(v_1 \frac{\partial}{\partial x_1} u_1 + u_1 \frac{\partial}{\partial x_1} v_1\right) + \cdots + \left(v_n \frac{\partial}{\partial x_1} u_n + u_n \frac{\partial}{\partial x_1} v_n\right) \\ \vdots \\ \left(v_1 \frac{\partial}{\partial x_n} u_1 + u_1 \frac{\partial}{\partial x_n} v_1\right) + \cdots + \left(v_n \frac{\partial}{\partial x_n} u_n + u \frac{\partial}{\partial x_n} v_n\right) \end{bmatrix}_{n \times 1}$$

$$= \begin{bmatrix} v_1 \frac{\partial u_1}{\partial x_1} + \dots + v_n \frac{\partial u_n}{\partial x_1} \\ \vdots \\ v_1 \frac{\partial u_1}{\partial x_n} + \dots + v_n \frac{\partial u_n}{\partial x_n} \end{bmatrix} + \begin{bmatrix} u_1 \frac{\partial v_1}{\partial x_1} + \dots + u_n \frac{\partial v_n}{\partial x_1} \\ \vdots \\ u_1 \frac{\partial v_1}{\partial x_n} + \dots + u_n \frac{\partial v_n}{\partial x_n} \end{bmatrix}$$

Define D as:

$$Dv(x) = \begin{bmatrix} \frac{\partial v_1}{\partial x_1} & \frac{\partial v_2}{\partial x_1} & \cdots & \frac{\partial v_n}{\partial x_1} \\ \vdots & & & \\ \frac{\partial v_n}{\partial x_1} & \cdots & & \frac{\partial v_n}{\partial x_n} \end{bmatrix}_{n \times n}$$

We can see that our previous result is equivilent to:

$$= [v_1, \dots, v_n]_{1 \times n} \begin{bmatrix} \frac{\partial u_1}{\partial x_1} & \cdots & \frac{\partial u_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial u_n}{\partial x_1} & \cdots & \frac{\partial u_n}{\partial x_n} \end{bmatrix}_{n \times n} +$$

$$[u_1, \dots, u_n]_{1 \times n} \begin{bmatrix} \frac{\partial v_1}{\partial x_1} & \cdots & \frac{\partial v_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial v_n}{\partial x_1} & \cdots & \frac{\partial v_n}{\partial x_n} \end{bmatrix}_{n \times n}$$

By our definition of $D\,u(x)$, this is equivilent to:

$$v^T D u + u^T D v \quad \square$$

b) $$A v = \begin{bmatrix} a_{11}(x_1,\dots,x_n) & \cdots & a_{1n}(x_1,\dots,x_n) \\ \vdots & & \\ a_{n1}(x_1,\dots,x_n) & \cdots & a_{nn}(x_1,\dots,x_n) \end{bmatrix}_{n\times n} \begin{bmatrix} v_1(x_1,\dots,x_n) \\ \vdots \\ v_n(x_1,\dots,x_n) \end{bmatrix}_{n\times 1}$$

$$= \begin{bmatrix} a_{11}v_1 + a_{12}v_2 + \dots + a_{1n}v_n \\ \vdots \\ a_{n1}v_1 + a_{n2}v_2 + \dots + a_{nn}v_n \end{bmatrix}_{n\times 1}$$

Using D as defined above:

$$D(Av) = \begin{bmatrix} \frac{\partial}{\partial x_1}\underbrace{(a_{11}v_1 + \dots + a_{1n}v_n)}_{(Av)_1} & \cdots\cdots & \frac{\partial}{\partial x_1}\underbrace{(a_{n1}v_1 + \dots + a_{nn}v_n)}_{(Av)_n} \\ \vdots & & \vdots \\ \frac{\partial}{\partial x_n}(a_{11}v_1 + \dots + a_{1n}v_n) & \cdots\cdots & \frac{\partial}{\partial x_n}(a_{n1}v_1 + \dots + a_{nn}v_n) \end{bmatrix}_{n\times n}$$

$$
= \begin{array}{ccc}
\downarrow [D(AV)]_{11} & & \downarrow [D(AV)]_{1n} \\
\left[\begin{array}{ccc}
\left(v_1 \frac{\partial}{\partial x_1} a_{11} + a_{11} \frac{\partial}{\partial x_1} v_1\right) + \dots + \left(v_n \frac{\partial}{\partial x_1} a_{1n} + a_{1n} \frac{\partial}{\partial x_1} v_n\right) & \cdots & \left(v_1 \frac{\partial}{\partial x_1} a_{n1} + a_{n1} \frac{\partial}{\partial x_1} v_1\right) + \dots + \left(v_n \frac{\partial}{\partial x_1} a_{nn} + a_{nn} \frac{\partial}{\partial x_1} v_n\right) \\
\vdots & & \vdots \\
\left(v_1 \frac{\partial}{\partial x_n} a_{11} + a_{11} \frac{\partial}{\partial x_n} v_1\right) + \dots + \left(v_n \frac{\partial}{\partial x} a_{1n} + a_{1n} \frac{\partial}{\partial x} v_n\right) & \cdots & \left(v_1 \frac{\partial}{\partial x_1} a_{n1} + a_{n1} \frac{\partial}{\partial x_1} v_1\right) + \dots + \left(v_n \frac{\partial}{\partial x_1} a_{nn} + a_{nn} \frac{\partial}{\partial x_1} v_n\right)
\end{array}\right]_{n \times n} \\
\uparrow & & \uparrow \\
[D(AV)]_{n1} & & [D(AV)]_{nn}
\end{array}
$$

If we collect terms, we get $A \cdot DV$

$$
ADV = \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} \frac{\partial v_1}{\partial x_1} & \cdots & \frac{\partial v_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial v_n}{\partial x_1} & \cdots & \frac{\partial v_n}{\partial x_n} \end{bmatrix}
$$

By careful inspection of what remains, we can rewrite it as $\sum_{i=1}^{n} v_i \, D a_i$:

$$
\underset{\substack{\uparrow \\ \text{scalar} \\ \text{fxn}}}{v_i} D a_i = \begin{bmatrix} v_i \frac{\partial a_{i1}}{\partial x_1} & \cdots & v_i \frac{\partial a_{ni}}{\partial x_1} \\ \vdots & & \vdots \\ v_i \frac{\partial a_{i1}}{\partial x_n} & \cdots & v_i \frac{\partial a_{ni}}{\partial x_n} \end{bmatrix}_{n \times n}
$$

Note: the i-th term of the sum is the i-th term in each $[i,j]$ entry (excluding terms)

Finally, we are left with:

$$D(Av) = \underbrace{\underbrace{A}_{n \times n} \cdot \underbrace{Dv}_{n \times n}}_{n \times n} + \underbrace{\sum_{i=1}^{n} \overset{\uparrow}{\underset{\text{scalar}}{v_i}} \underbrace{Da_i}_{n \times n}}_{n \times n} = n \times n \qquad \square$$

2. Since y is separable, we use

$$(Dr)_{ij} = \frac{\partial r_i}{\partial c_j} = \frac{\partial r(t_i)}{\partial c_j}$$

$$r = \begin{bmatrix} c_3 + c_1 t_1^{c_2} - y_1 \\ c_3 + c_1 t_2^{c_2} - y_2 \\ c_3 + c_1 t_3^{c_2} - y_3 \end{bmatrix}$$

$$Dr = \begin{bmatrix} \frac{\partial r_1}{\partial c_1} & \frac{\partial r_1}{\partial c_2} & \frac{\partial r_1}{\partial c_3} \\ \vdots & & \\ \frac{\partial r_3}{\partial c_1} & \cdots & \frac{\partial r_3}{\partial c_3} \end{bmatrix} = \begin{bmatrix} t_1^{c_2} & c_1 c_2 t_1^{c_2} & 1 \\ t_2^{c_2} & c_1 c_2 t_2^{c_2} & 1 \\ t_3^{c_2} & c_1 c_2 t_3^{c_2} & 1 \end{bmatrix}$$

3. $y = c_1 \cdot t \cdot e^{c_2 t}$

$$r = \begin{bmatrix} c_1 \cdot 1 \cdot e^{c_2} - 6.2 \\ c_1 \cdot 2 \cdot e^{2c_2} - 9.5 \\ \vdots \\ c_1 \cdot 10 \cdot e^{10c_2} - 11.9 \end{bmatrix}$$

$$Dr = \begin{bmatrix} \frac{\partial r_1}{\partial c_1} & \frac{\partial r_2}{\partial c_2} \\ \vdots & \\ \frac{\partial r_{10}}{\partial c_1} & \frac{\partial r_{10}}{\partial c_2} \end{bmatrix} = \begin{bmatrix} 1 \cdot e^{c_2} & 1^2 \cdot c_1 e^{c_2} \\ 2 \cdot e^{2c_2} & 2^2 \cdot c_1 e^{2 \cdot c_2} \\ \vdots & \vdots \\ 10 e^{10c_2} & 10^2 \cdot c_1 e^{10c_2} \end{bmatrix}$$

See code attached

4)

a) False, since the Gauss-Newton method seeks a solution to $\nabla r(c) = c$, it is possible that it finds a local instead of global minimum as its solution, or even a maximum.

b) False, the same answer for a): it may converge to a local solution or even a maximum depending on the initial guess and/or specific problem

c) True, while it shows better conditioning, this method should still converge to the same solution in this case, but likely in a more stable manner

d) Probably true, while it is likely the parameters will be different, it is not impossible that the same parameters are returned. However, they definitely yield the same <u>form</u> of model

5) a)

i) Linearized Least Squares

We modify the data based on the model to obtain a "linearized model". Regular least squares is then used, and the results are translated back to the original model

ii) Gauss-Newton

We "solve" the system of equations $r(x_i) = \vec{0}$ using Newton's method to find where the closest solution is (using gradient $= 0$ condition)

iii) Levenberg-Marquardt

The same idea as GN except that we add a user-tuned preconditioning/regularization term to increase stability/accuracy for an ill conditioned problem

b) We have $F(x) = DE(x) = r(x)^T Dr(x) = 0$ derivative condition

To setup Multi-variate Newton's Method:

$$x^{k+1} = x^k - (DF(x)^T)^{-1} F(x)^T$$

$$= D\left((Dr(x))^T r(x)\right)$$

$$= (Dr(x))^T Dr(x) + \sum_{i=1}^{m} r_i(x) D\left[(Dr(x))^T\right]_i$$

$$\approx (Dr(x))^T Dr(x)$$

$$= (r(x)^T Dr(x))^T = (Dr(x))^T r(x)$$

$$\Rightarrow x^{k+1} = x^k - \left((Dr(x))^T Dr(x)\right)^{-1} \left((Dr(x))^T r(x)\right)$$

$$-\left((Dr(x))^T Dr(x)\right)^{-1} \left((Dr(x))^T r(x)\right) = v$$

$$\Downarrow$$

$$v\left((Dr(x))^T Dr(x)\right) = -\left((Dr(x))^T r(x)\right)$$

Algorithm:

$A = Dr(x)$

$A^T A v = -A^T r \Rightarrow v = \dots$

$$x^{k+1} = x^k + v^k$$

c) we have the same problem setup as b) except we add a user-tuned regularization term:

Algorithm:

$$A = Dr(x)$$

$$\left(A^T A + \lambda \operatorname{diag}(A^T A)\right) v = -A^T r \implies v = \dots$$

$$x^{k+1} = x^k + v^k$$

d) $v : \mathbb{R}^n \to \mathbb{R}^n \qquad u : \mathbb{R}^n \to \mathbb{R}^n$

$$D(v^T u) = v\,Du + u\,Dv$$

e) $A : \mathbb{R}^n \to \mathbb{R}^{n \times n} \qquad v : \mathbb{R}^n \to \mathbb{R}^n$

$$D(Av) = A\,Dv + \sum_{i=1}^{n} v_i\,Da_i$$

Problem 3

Gauss-Newton

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from numpy import log, exp

def gauss_newton(t, y, Dr, y_pred_fxn, c0, tol=1e-6, max_iter=100000):
    c = c0.copy()

    for j in range(max_iter):
        r = y_pred_fxn(t,c) - y

        if np.linalg.norm(r) < tol:
            break

        A = Dr(t,c)

        v = np.linalg.solve(A.T @ A, -A.T @ r)

        c += v

    return c, j
```

In [ ]:
```
# Estabish data vectors
t = np.arange(1,11, dtype=np.float64).T
y = np.array([6.2, 9.5, 12.3, 13.9, 14.6, 13.5, 13.3, 12.7, 12.4, 11.9]).T

Dr = lambda t, c: np.column_stack((t*exp(t*c[1]) , t*t*c[0]*exp(t*c[1])))
y_pred_fxn = lambda t, c: c[0] * t * exp(c[1]*t)

c_gn, iter_gn = gauss_newton(t,y,Dr,y_pred_fxn, np.array([6.,.1]))

y_pred_gn = c_gn[0] * t * exp(c_gn[1] * t)

print(f"c0: {c_gn[0]}, c1: {c_gn[1]}")
print(f"Iterations: {iter_gn}")
```

```
c0: 7.054228542648305, c1: -0.18289874347158333
Iterations: 99999
```

Levenberg-Marquardt

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from numpy import log, exp

def lev_mar(t, y, Dr, y_pred_fxn, c0, lmbd, tol=1e-6, max_iter=100000):
    c = c0.copy()

    for j in range(max_iter):
        r = y_pred_fxn(t,c) - y

        if np.linalg.norm(r) < tol:
            break

        A = Dr(t,c)
```

```
        D = np.diag(np.diag(A.T @ A))
        v = np.linalg.solve(A.T @ A + lmbd * D, -A.T @ r)

        c += v

    return c, j
```

In [ ]:
```
# Estabish data vectors
t = np.arange(1,11, dtype=np.float64).T
y = np.array([6.2, 9.5, 12.3, 13.9, 14.6, 13.5, 13.3, 12.7, 12.4, 11.9]).T

Dr = lambda t, c: np.column_stack((t*exp(t*c[1]) , t*t*c[0]*exp(t*c[1])))
y_pred_fxn = lambda t, c: c[0] * t * exp(c[1]*t)

c_lm, iter_lm = lev_mar(t,y,Dr,y_pred_fxn, np.array([6.,.1]), 1)

y_pred_lm = c_lm[0] * t * exp(c_lm[1] * t)

print(f"c0: {c_lm[0]}, c1: {c_lm[1]}")
print(f"Iterations: {iter_lm}")
```

```
c0: 7.054228542648297, c1: -0.18289874347158316
Iterations: 99999
```

Data Linearization

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from numpy import log, exp

# Estabish data vectors
t = np.arange(1,11, dtype=np.float64).T
y = np.array([6.2, 9.5, 12.3, 13.9, 14.6, 13.5, 13.3, 12.7, 12.4, 11.9]).T

# Define (linearized) least squares vectors
A = np.column_stack((np.ones(10, dtype=np.float64), t))
b_ = log(y) - log(t)

# Solve linearized least squares
c_ = np.linalg.solve(A.T @ A, A.T @ b_.T)
c = c_.copy()
c[0] = exp(c[0]) # Convert linearized solution to orginal form

print(f"c1: {round(c[0],3)}, c2: {round(c[1],3)}")

# Use fitted model to obtain predictions
y_pred = c[0]*t*exp(c[1]*t)
```

```
c1: 7.122, c2: -0.184
```

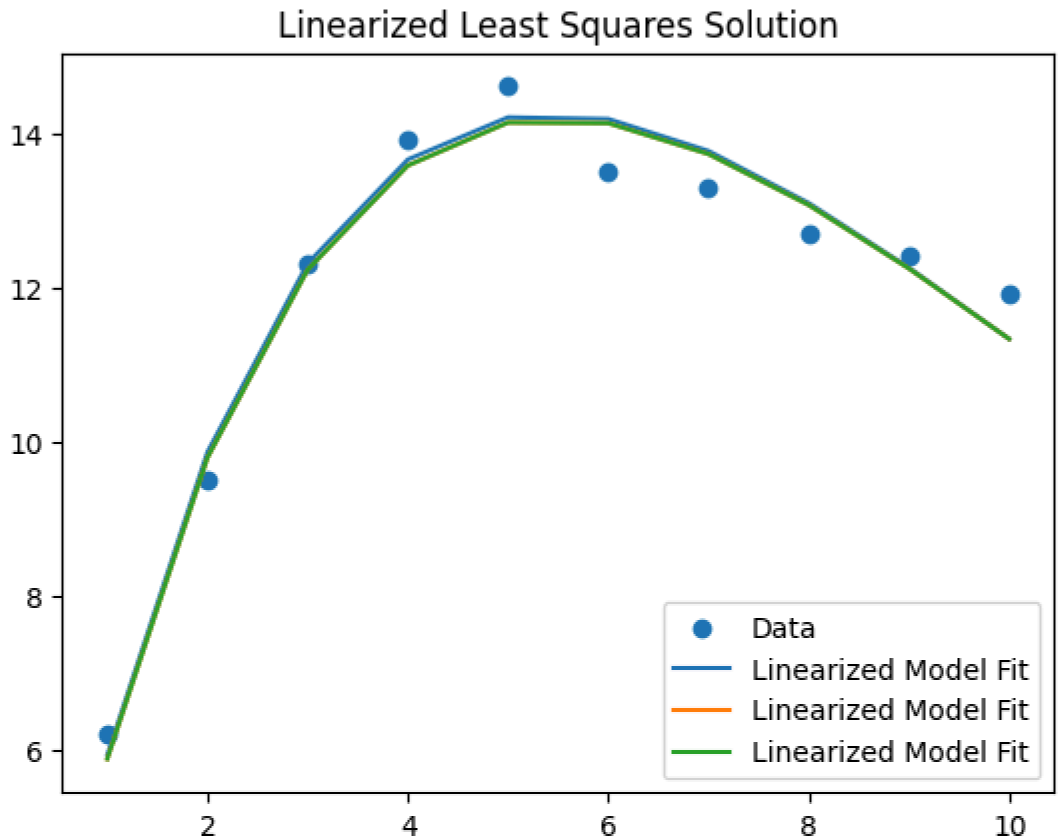
Results and Discussion

In [ ]:
```
with np.printoptions(precision=5, suppress=True):
    print(f"Sum of Residuals (Gauss Newton): {np.linalg.norm(y-y_pred_gn)}")
    print(f"Sum of Residuals (Levenberg-Marquardt): {np.linalg.norm(y-y_pred_lm)}")
    print(f"Sum of Residuals (Linearized): {np.linalg.norm(y-y_pred)}")
```

```
Sum of Residuals (Gauss Newton): 1.2543574926529986
Sum of Residuals (Levenberg-Marquardt): 1.254357492653
Sum of Residuals (Linearized): 1.2667076986711072
```

In [ ]:

```
plt.scatter(t,y, label='Data')
plt.plot(t,y_pred, label='Linearized Model Fit')
plt.plot(t,y_pred_gn, label='Linearized Model Fit')
plt.plot(t,y_pred_lm, label='Linearized Model Fit')
plt.legend()
plt.title('Linearized Least Squares Solution')
plt.show()
```



There is minor improvement from Gauss-Newton vs the original data linearization method, but going from Gauss-Newton to Levenberg-Marquardt there is almost no change. This indicates that the problem is well conditioned and the additional robustness provided by Levenberg-Marquardt is no needed here.