Exercise P21: The major goal of this exercise is to show that the usual matrix-vector product algorithm is backward-stable when we regard the matrix as the input; see part (c). For simplicity I'll assume all entries are real numbers. Always assume axiom (13.5) and (13.7) hold.

a. Let us start in one dimension. Fix $x \in \mathbb{R}^1$. Show that if the problem is scalar multiplication, $f(a) = ax$ for $a \in \mathbb{R}^1$, then the obvious algorithm, $\tilde{f}(a) = fl(a) \otimes fl(x)$ is backwards stable.

Solution:
From the given algorithm we know that a and x must be rounded to floating point values, then we apply the $\otimes$ operation. By 13.5 we know that,

$$fl(a) = a(1 + \epsilon_1),$$

$$fl(x) = x(1 + \epsilon_2),$$

for some $|\epsilon_1|, |\epsilon_2| \leq \epsilon_{machine}$. Looking at $\otimes$ we know from 13.7 that,

$$fl(a) \otimes fl(x) = (fl(a) \times fl(x))(1 + \epsilon_3).$$

for some $\epsilon_3 \leq \epsilon_{machine}$. By substitution we get,

$$\begin{aligned}
\tilde{f}(a) &= fl(a) \otimes fl(x) \\
&= (fl(a) \times fl(x))(1 + \epsilon 3) \\
&= a(1 + \epsilon_1)(1 + \epsilon_1)(1 + \epsilon_3)x \\
&= \tilde{a}x, \\
&= f(\tilde{a}).
\end{aligned}$$

With the last equality let $\tilde{a} = a(1+\epsilon_1)(1+\epsilon_1)(1+\epsilon_3) = a(1+O(\epsilon_{machine}))$, and therefore scalar multiplication is backwards stable.

b. Fix $x \in \mathbb{R}^2$. Show that if $a \in \mathbb{R}^2$ then the obvious algorithm $\tilde{f}(a) = fl(a_1) \otimes f(x_1) \oplus fl(a_1) \otimes fl(x_2)$ for the inner product problem $f(a) = a^T x$ is backwards stable.

Solution:
First let's consider the obvious algorithm proposed in the problem statement,

$$\tilde{f}(a) = [fl(a_1) \otimes f(x_1)] \oplus [fl(a_1) \otimes fl(x_2)].$$

Substituting the solution from part a we get the following,

$$\begin{aligned}
\tilde{f}(a) &= \tilde{f}_{p.A}(a_1) \oplus \tilde{f}_{p.A}(a_2), \\
&= \hat{a}_1 x_1 \oplus \hat{a}_2 x_2.
\end{aligned}$$

From theorem 13.7 we know the following,

$$\hat{a}_1 x_1 \oplus \hat{a}_2 x_2 = (\hat{a}_1 x_1 + \hat{a}_2 x_2)(1 + \epsilon_1),$$

for some $\epsilon_1 \leq \epsilon_{machine}$. By substitution we get,

$$\begin{aligned}
\tilde{f}(a) &= (\hat{a}_1 x_1 + \hat{a}_2 x_2)(1 + \epsilon_1), \\
&= (\hat{a}_1 x_1 + \hat{a}_2 x_2)(1 + \epsilon_1), \\
&= \hat{a}_1(1 + \epsilon_1)x_1 + \hat{a}_2(1 + \epsilon_1)x_2, \\
&= \tilde{a}_1 x_1 + \tilde{a}_2 x_2, \\
&= f(\tilde{a}).
\end{aligned}$$

With the last equalities, we let $\tilde{a}_i = \hat{a}_i(1 + \epsilon_1) = a_i(1 + O(\epsilon_{machine}))$. Now consider the following,

$$\begin{aligned}
\frac{||\tilde{a} - a||}{||a||} &= \frac{||(a_1(1 + O(\epsilon_{machine})) - a_1) + (a_2(1 + O(\epsilon_{machine})) - a_2)||}{||a||}. \\
&\leq \frac{||(a_1(1 + O(\epsilon_{machine})) - a_1)|| + ||(a_2(1 + O(\epsilon_{machine})) - a_2)||}{||a||}. \\
&= \frac{||(a_1(O(\epsilon_{machine}))|| + ||a_2(O(\epsilon_{machine}))||}{||a||}. \\
&= \frac{||a_1|| + ||a_2||}{||a||} ||(O(\epsilon_{machine}))||. \\
&= O(\epsilon_{machine}).
\end{aligned}$$

c. Fix $A \in \mathbb{R}^n$. Show that if $A \in \mathbb{R}^{mxn}$ then the obvious algorithm $\tilde{f}(A)$ for the matrix-vector product problem $f(A) = Ax$ is backward stable.

Solution:
First let $\hat{f}(a)$ be the algorithm for the inner product problem from section b, and recall that Ax can be written as a set of inner products with the rows a_i of A and x.

$$\tilde{f}(A) = Ax = \begin{bmatrix} \dots & a_1 & \dots \\ \hline \dots & \dots & \dots \\ \hline \dots & a_m & \dots \end{bmatrix} x = \begin{bmatrix} a_1^* x \\ \vdots \\ a_m^* x \end{bmatrix} = \begin{bmatrix} \hat{f}(a_1) \\ \vdots \\ \hat{f}(a_m) \end{bmatrix}$$

From section b we showed that the inner product algorithm $\hat{f}(a)$ is backwards stable, therefore $\hat{f}(a) = f(\hat{a})$ for some $\hat{a}$ with,

$$\frac{||\hat{a} - a||}{||a||} = O(\epsilon_{machine}).$$

By substitution we get the following,

$$\tilde{f}(A) = Ax = \begin{bmatrix} \hat{f}(a_1) \\ \vdots \\ \hat{f}(a_m) \end{bmatrix} = \begin{bmatrix} f(\hat{a_1}) \\ \vdots \\ f(\hat{a_m}) \end{bmatrix} = f(\tilde{A}).$$

Where the rows of $\tilde{A}$ are composed of $\hat{a}_i$. By construction we know that the relative error on the matrix $\tilde{A}$ when considering the induced infinity norm must a multiple of relative error of the rows $\hat{a}_i$ under the inner product algorithm. Note that there are n inner product computations on an A^{nxm} matrix therefore we get the following,

$$\frac{||\tilde{A} - A||}{||A||} = n\frac{||\hat{a} - a||}{||a||} \leq O(n\epsilon_{machine}) = O(\epsilon_{machine}).$$

Also note that hidden inside the bound for the backwards errors of the inner product is a factor of m so the the constant on $O(\epsilon_{machine})$ looks to be dependent on m and n(as expected from lecture 14).

d. Fix $A \in \mathbb{R}^{mxn}$. explain in at most 8 sentences why the obvious algorithm, $\tilde{f}(x)$ for the matrix-vector product problem $f(x) = Ax$ is not generally backwards stable. However, this result depends on dimension. In fact for what m, n do we already know that this $\tilde{f}(x)$ is stable.

Solution:
As described in the previous problem, and at the end of lecture 14 the bounds of the backwards error are determined by,

$$\frac{||\tilde{x} - x||}{||x||} \leq C\kappa(A)\epsilon_{machine}$$

where $\kappa(A)$ is the condition number on A and C, as we've shown, is a product of m and n. With a large enough matrix, one could consider a problem on the order of $mn = 1/\epsilon_{machine}$ effectively exploding the backwards error. Clearly the algorithm is backwards stable for small n and m, we showed that the inner product algorithm (which is itself a form of matrix vector product) were $m = 1$ and $n = 2$ is backwards stable in part b.

Exercise 15.2: Consider an algorithm for the problem of computing the (full) SVD of a matrix. The data for this problem is a matrix A and the solution is three matrices U (unitary), Σ (diagonal), and V (unitary) such that $A = U\Sigma V^*$.

a. Explain what it would mean for this algorithm, to be backwards stable.

Solution:
Recalling the definition of backwards stability, for some algorithm $\tilde{f}$ which describes a proble, f for each input x,

$$\tilde{f}(x) = f(\tilde{x}),$$

For some $\tilde{x}$ where,

$$\frac{||\tilde{x} - x||}{||x||} = O(\epsilon_{machine}).$$

Applying this definition we know that our algorithm for computing the full SVD will produce an SVD factorization for $\tilde{A}$ where the relative difference between A and $\tilde{A}$ is on the order of $\epsilon_{machine}$.

b. In fact, for a simple reason, this algorithm, cannot be backward stable. Explain,

Solution:
If the range of the problem is defined by all possible SVD and the input space is defined by all matrices mxn than the condition $\tilde{f}(A) = f(\tilde{A})$ for some $\tilde{A}$ where the relative difference $\tilde{A}$ is on the order of $\epsilon_{machine}$ can never be satisfied as it assumes that two matrices A and $\tilde{A}$ have same SVD. A contradiction as $A \neq \tilde{A}$ and the SVD is unique.

c. Fortunately, the standard algorithms for computing the SVD are stable. Explain what stability means for such an algorithm.

Solution:
The approach described at the beginning of lecture 31 describes transforming the SVD problem into an eigenvalue problem which is more sensitive to perturbations. In doing so we condition the problem of producing an SVD on only the singular values and not the entire factorization $U\Sigma V^*$.

Exercise 16.2: The idea of this exercise is to carry out an experiment analogous to the one described in the lecture, but for SVD instead of QR factorization.

a. Write a Matlab program that constructs a $50x50$ matrix $A = USV'$ where U and V are random orthoganal matrices and S is a diagonal matrix whose diagonal entries are random uniformly distributed numbers in $[0, 1]$ sorted into non-increasing order. Have your program compute $[U2, S2, V2] = svd(A)$ and the norms of $U-U2$, $V-V2$, $S - S2$,and $A - U2S2V2$. Do this for five matrices and comment on the results.

Solution:
Consider the following Matlab script,
Code:

```
uNorms = []
vNorms = []
sNorms = []
aNorms = []

for i = 1:5
    [U, S, V] = svd(randn(50));
    S = diag(sort(rand(50,1), 'descend'));
    A = U*S*V';
    [U2, S2, V2] = svd(A);

    uNorms(i) = norm(U - U2);
    vNorms(i) = norm(V - V2);
    sNorms(i) = norm(S - S2);
    aNorms(i) = norm(A - U2*S2*V2');
end
```

From the norm data we can see that the svd() implementation is producing error on the unitary matrices of size 2. The full reproduced matrix A gives an accuracy on the same level as the sigma matrix (as expected, that's how the induced two-norm is defined), and the sigma matrix is accurate on the order of 10^{-14}. As the hint suggests looking at the matrices $U' * U2$ and $V' * V2$ suggests that we actually produced very similar unitary matrices, except the signs of the columns are swapped around.
Console:

```
sNorms'
   1.0e-14 *

   0.549113722895636
   0.422311634295492
   0.424924035119725
   0.293167585255438
   0.507413198961319

uNorms'
   2.000000000000001
```

```
    2.000000000000001
    2.000000000000001
    2.000000000000001
    2.000000000000001

vNorms'
    2.000000000000001
    2.000000000000001
    2.000000000000001
    2.000000000000001
    2.000000000000001

sNorms'
    1.0e-14 *

    0.111155715336585
    0.134297590715998
    0.161259615988629
    0.113121818827519
    0.258295908813477
```

b. Fix teh signs in your computed SVD so that the difficulties of A go away. Run the program again for five random matrices and comment on the various norms. Do they have a connection with $cond(A)$?

Solution:
The following Matlab script fixes the sign problem found in part a.
Code:

```
uNorms = []
vNorms = []
sNorms = []
aNorms = []
Acond = []

for i = 1:5
    [U, S, V] = svd(randn(50));
    S = diag(sort(rand(50,1), 'descend'));
    A = U*S*V';
    [U2, S2, V2] = svd(A);
    Acond(i) = cond(A);
    V2 = V2*diag(round(diag(V'*V2)))';
    U2 = U2*diag(round(diag(U'*U2)))';

    uNorms(i) = norm(U - U2);
```

```
    vNorms(i) = norm(V - V2);
    sNorms(i) = norm(S - S2)/norm(S);
    aNorms(i) = norm(A - U2*S2*V2')/norm(A);
end
```

Console

```
aNorms =
   1.0e-13 *

   0.027457453418757
   0.103590411306370
   0.039279425441070
   0.029778718590716
   0.044468560072418

uNorms =
   1.0e-10 *

   0.001605039100297
   0.225048039034936
   0.002210452931389
   0.003542706911795
   0.002785172958743

vNorms =
   1.0e-10 *

   0.001609522071231
   0.225044242257287
   0.002211971042967
   0.003546098831671
   0.002787234228752

sNorms =
   1.0e-14 *

   0.138278487657420
   0.203236345132158
   0.147080497177988
   0.159085266887556
   0.158695549021527

 Acond =
   1.0e+02 *

   0.483303128115473
```

```
    0.427979820171040
    0.484458804204254
    2.071752832435978
    0.757539213962305
```

From here we see a huge jump in the accuracy in the norms of our unitary matrices. From the looks of it it seems as though finding the unitary factors to the SVD has relatively the same conditioning as finding sigma values and the input matrix A. We can really explore the relationship between the unitary factors and the condition number of A by simulating very ill-conditioned matrices. Doing so we see that the norms in our unitary factors have increased by a factor of 10^{10} similarly to the condition number of out matrix.

Code:

```
uNorms = []
vNorms = []
sNorms = []
aNorms = []
Acond = []

for i = 1:5
    [U, S, V] = svd(randn(50));
    x1 = 5000000*rand(25,1)-25;
    x2 = .0005*rand(25,1);
    S = diag(sort([x1' x2'], 'descend'));
    A = U*S*V';
    [U2, S2, V2] = svd(A);
    Acond(i) = cond(A);
    V2 = V2*diag(round(diag(V'*V2)))';
    U2 = U2*diag(round(diag(U'*U2)))';

    uNorms(i) = norm(U - U2);
    vNorms(i) = norm(V - V2);
    sNorms(i) = norm(S - S2)/norm(S);
    aNorms(i) = norm(A - U2*S2*V2')/norm(A);
end
```

```
>> aNorms'
    1.0e-14 *

    0.281021515060486
    0.773610009677406
    0.731129340264174
    0.341214719366703
    0.202734945660890
```

```
>> uNorms'
   1.0e-03 *

   0.224750383854456
   0.199826275610365
   0.240281169009316
   0.324787260618243
   0.087691111790049


>> vNorms'
   1.0e-03 *

   0.224219579648742
   0.199607413114002
   0.243273246289136
   0.324361586942921
   0.088270115302406


>> sNorms'
   1.0e-14 *

   0.151382643383725
   0.183175896995704
   0.134350421198050
   0.126464322882021
   0.296318070925046


>> Acond'
   1.0e+12 *

   0.884885420339284
   4.442917268303241
   0.075883079322376
   0.429441761479880
   0.634439894325554
```

c. Replace the diagonal entries of S by their sixth powers and repeat b. Do you see significant differences between the results of this exercise and those of the experiment for QR factorization?

Solution:
Similarly to our previous simulation, we have affected the condition number of our matrix A. Doing so we see that the accuracy in the unitary factors has decreased while our relative forward error in σ is still very good and our relative backwards error in A is also very good. Note that these results suggests that the svd() algorithm is backwards stable.
Code:

```
uNorms = []
vNorms = []
sNorms = []
aNorms = []
Acond = []

for i = 1:5
    [U, S, V] = svd(randn(50));
    x1 = 5000000*rand(25,1)-25;
    x2 = .0005*rand(25,1);
    S = diag(sort([x1' x2'], 'descend'));
    A = U*S*V';
    [U2, S2, V2] = svd(A);
    Acond(i) = cond(A);
    V2 = V2*diag(round(diag(V'*V2)))';
    U2 = U2*diag(round(diag(U'*U2)))';

    uNorms(i) = norm(U - U2);
    vNorms(i) = norm(V - V2);
    sNorms(i) = norm(S - S2)/norm(S);
    aNorms(i) = norm(A - U2*S2*V2')/norm(A);
end
```

```
>> aNorms'
   1.0e-14 *

   0.281021515060486
   0.773610009677406
   0.731129340264174
   0.341214719366703
   0.202734945660890
```

```
>> uNorms'
   1.0e-03 *

   0.224750383854456
```

```
    0.199826275610365
    0.240281169009316
    0.324787260618243
    0.087691111790049


>> vNorms'
    1.0e-03 *

    0.224219579648742
    0.199607413114002
    0.243273246289136
    0.324361586942921
    0.088270115302406


>> sNorms'
    1.0e-14 *

    0.151382643383725
    0.183175896995704
    0.134350421198050
    0.126464322882021
    0.296318070925046


>> Acond'
    1.0e+12 *

    0.884885420339284
    4.442917268303241
    0.075883079322376
    0.429441761479880
    0.634439894325554
```

Throughout this experiment we have had many attempts to perturb our input matrix A, and unlike in the QR factorization experiment, our method for computing the SVD continued to give us small backward error in the aNorms and small forward error in the sNorms. Had we framed our problem as described in exercise 15, we would have had to take into account the error in U and V in our forward error which would point to an ill-conditioned problem as we explored experimentally, and analogously in the QR factorization experiment.

Exercise 17.2: A triangular system (17.1) is solved by back substitution. Exactly what does Theorem 17.1 imply about the error $||\tilde{x} - x||$?

Solution:
Theorem 17.1 states that Algorithm 17.1 is backwards stable if the computed solution for some $\tilde{x} \in \mathbb{C}^m$ satisfies,

$$(R + \delta R)\tilde{x} = b,$$

for some upper-triangular $\delta R \in \mathbb{C}^{mxm}$ with,

$$\frac{||\delta R||}{||R||} = O(\epsilon_{machine}).$$

To reiterate our algorithm solves for x with R as input while fixing b. As we can see the Theorem makes a clear statement on how R is bounded, and in order to prove backwards stability we must also show that $|x|$ bounded. Note that the problem is defined by $Rx = b$ where b is fixed, by substitution into the first equation of the theorem we get,

$$\begin{aligned}
(R + \delta R)\tilde{x} &= Rx, \\
R\tilde{x} + \delta R\tilde{x} - Rx &= 0, \\
R\tilde{x} - Rx &= -\delta R\tilde{x}, \\
R(\tilde{x} - x) &= -\delta R\tilde{x}, \\
||R(\tilde{x} - x)|| &= ||\delta R\tilde{x}||, \\
||R||||(\tilde{x} - x)|| &= ||\tilde{x}||||\delta R||, \\
||(\tilde{x} - x)|| &= ||\tilde{x}||\frac{||\delta R||}{||R||}, \\
||(\tilde{x} - x)|| &= ||\tilde{x}||O(\epsilon_{machine}).
\end{aligned}$$

Thus Theorem 17.1 implies that $||(\tilde{x} - x)|| = ||\tilde{x}||O(\epsilon_{machine})$.