VMLS - Study Guide Chapter 1 **Name:** Ben Chen

Please always include this title page with your PDF. Include your name above.

- Submit your work in Gradescope as a PDF - you will identify where your “questions are.”
- Identify the question number as you submit. Since we grade "blind" if the questions are NOT identified, the work WILL NOT BE GRADED and a 0 will be recorded. Always leave enough time to identify the questions when submitting.
- One section per page (if a page or less) - We prefer to grade the main solution in a single page, extra work can be included on the following page.
- Long instructions may be removed to fit on a single page.
- **Do not start a new question in the middle of a page.**
- Solutions to book questions are provided for reference.
- You may NOT submit given solutions - this includes minor modifications - as your own.
- Solutions that do not show individual engagement with the solutions will be marked as no credit and can be considered a violation of honor code.
- If you use the given solutions you must reference or explain how you used them, in particular...

For full credit, EACH book exercise in the Study Guides must use one or more of the following methods and FOR EACH QUESTION. Identify the number the method by number to ensure full credit.

Method 1 - Provide original examples which demonstrate the ideas of the exercise in addition to your solution.

Method 2 - Include and discuss the specific topics needed from the chapter and how they relate to the question.

Method 3 - Include original Python code, of reasonable length (as screenshot r text) to show how the topic or concept was explored.

Method 4 - Expand the given solution in a significant way, with additional steps and comments. All steps are justified. This is a good method for a proof for which you are only given a basic outline.

Method 5 - Attempt the exercise without looking at the solution and then the solution is used to check work. Words are used to describe the results.

Method 6 - Provide an analysis of the strategies used to understand the exercise, describing in detail what was challenging, who helped you or what resources were used. The process of understanding is described.

1. (20 pts) Reading the book carefully is essential in this class and in all advanced mathematics. This is an exercise in annotation.

For this first exercise, pick a section or page from Chapter one of VMLS.

Read straight through the section once for an overview. Include the following 3 items for #1.

a) Write down your initial thoughts, questions, and first impressions ('whaaat???')
b) Now, return to the section and slowly work through each line using a pencil or pen.
   - Expand equations.
   - Identify key concepts and explain in your own words.
   - Make note - is that a vector or a scalar?
   - Fill in missing ideas or steps.
   - Include a screenshot of your annotation.

c) How has your understanding of the section changed?

Include a screenshot of your annotation.

1.4 Inner product

The (standard) *inner product* (also called *dot product*) of two n-vectors is defined as the scalar

$$a^T b = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n,$$

the sum of the products of corresponding entries. (The origin of the superscript 'T' in the inner product notation $a^T b$ will be explained in chapter 6.) Some other notations for the inner product (that we will not use in this book) are $\langle a, b\rangle$, $\langle a|b\rangle$, (a, b), and $a \cdot b$. (In the notation used in this book, (a, b) denotes a stacked vector of length $2n$.) As you might guess, there is also a vector *outer product*, which we will encounter later, in §10.1. As a specific example of the inner product, we have

$$\begin{bmatrix} -1 \\ 2 \\ 2 \end{bmatrix}^T \begin{bmatrix} 1 \\ 0 \\ -3 \end{bmatrix} = (-1)(1) + (2)(0) + (2)(-3) = -7.$$

When $n = 1$, the inner product reduces to the usual product of two numbers.

Properties. The inner product satisfies some simple properties that are easily verified from the definition. If a, b, and c are vectors of the same size, and γ is a scalar, we have the following.

- *Commutativity.* $a^T b = b^T a$. The order of the two vector arguments in the inner product does not matter.
- *Associativity with scalar multiplication.* $(\gamma a)^T b = \gamma(a^T b)$, so we can write both as $\gamma a^T b$.
- *Distributivity with vector addition.* $(a+b)^T c = a^T c + b^T c$. The inner product can be distributed across vector addition.

These can be combined to obtain other identities, such as $a^T(\gamma b) = \gamma(a^T b)$, or $a^T(b + \gamma c) = a^T b + \gamma a^T c$. As another useful example, we have, for any vectors a, b, c, d of the same size,

$$(a + b)^T (c + d) = a^T c + a^T d + b^T c + b^T d.$$

Something that was interesting that I noticed when reading the textbook about the fundamentals of vectors is that when you have 2 vectors (v1, and v2) transposed, the dot product is a singular number, which means that information is definitely lost over time.

So this led me to think that (for example) if you keep transposing the v1 with the result of (v1 transposed v2), then you'll eventually get a super large number that isn't really reflective of what the original information entails

My gut reaction when seeing vectors for the first time was that they looked like sets and so when I see two vectors transposed next to each other, my brain jumps to "we should find the similarities of the two to form a new set" instead of seeing that the result of a transpose is 1 singular number.

I was convinced to believe this is the case when I read this "properties" section to realize that this transpose operation behaves similarly to how multiplication works in arithmetic. But what was even trickier was that the order of operation doesn't really matter in multiplication, but does matter for transpose because being mathematically the same does not mean the operation is computationally the same. Thomas does a good job of explaining this in the lecture when he talks about the # of operations changing due to the ordering of choosing when to transpose.

It took a little getting used to, but reframing the way I thought about this helped my understanding of section 1.4

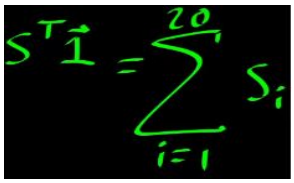
2. (10pts) Solve the Random exercise from the video and Piazza in your own words here.

1.9 *Symptoms vector.* A 20-vector s records whether each of 20 different symptoms is present in a medical patient, with $s_i = 1$ meaning the patient has the symptom and $s_i = 0$ meaning she does not. Express the following using vector notation.

(a) The total number of symptoms the patient has.

(b) The patient exhibits five out of the first ten symptoms.

a) The total number of symptoms is a simple for-loop python with a counter looking something that represents something like: for every element in the array/vector, increment +1 to the counter so the counter ends up being a sum of all the elements in the vector so mathematically, it can be represented as thesum of # of elements inside the 20-vector record "s". That's where Thomas (the lecturer in the video) gets the Sigma notation summing from i = 1 to 20 incrementing based on if s sub-i exists. 1 transpose s is just as correct as s transpose vector 1 because transpose has commutative property.




```
In [5]: record_of_symptoms = [1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1] # random example of positive/negative symptoms
        number_of_symptoms = 0

        for i in record_of_symptoms:
            number_of_symptoms += 1

        print(number_of_symptoms)

20
```

in summation notation:

$$\text{number\_of\_symptoms} = \sum_{i=1}^{\text{len(record\_of\_symptoms)}} \text{record\_of\_symptoms[i]}$$

a.k.a len(record_of_symptoms)

b) If we want to represent that the sum of the first 10 elements s = 5, then 20 in the first sigma notation turns into "10", and i = 1 (still incrementing by 1 still) and s sub-i = 5

in summation notation:

$$\text{number\_of\_positive\_symptoms\_in\_first\_ten} = \sum_{i=1}^{\text{len(record\_of\_symptoms)} = 10} \text{record\_of\_symptoms[i]} = 5$$

We change the length of the record of symptoms from 20 -> 10 because we no longer care about iterating through the entire list of vectors, and only want to evaluate the first 10. We change record_of_symptoms[i] = 5 because problem b states that the positive symptoms must be 5 (as a given). So now we're given the end result and now have to write the transpose notation for it, which would be some vector "m" transpose s = 5. m sub-i is representative of a 20-vector defined as 1 if and only if i is greater than 1, and less than 10 (we're referring "i" as index in the list) and 0 otherwise (which means m sub-i is "0" when we are at the 11th index.

So to describe this is:

$$m^T s = 5 \quad \text{or} \quad \sum_{i=1}^{10} s_i = 5$$

3. (10 pts) Explain the solution to 1.8 here in your own words. (Since you are given a solution, you will be graded on your ability to explain).

1.8 *Profit and sales vectors.* A company sells n different products or items. The n-vector p gives the profit, in dollars per unit, for each of the n items. (The entries of p are typically positive, but a few items might have negative entries. These items are called *loss leaders*, and are used to increase customer engagement in the hope that the customer will make other, profitable purchases.) The n-vector s gives the total sales of each of the items, over some period (such as a month), *i.e.*, s_i is the total number of units of item i sold. (These are also typically nonnegative, but negative entries can be used to reflect items that were purchased in a previous time period and returned in this one.) Express the total profit in terms of p and s using vector notation.

The problem is asking for total profit, which is the multiplication of # of items sold, and profit per item. p sub-i is the profit of the i-th item, and s sub-i is the total sale of the i-th item, which means (p sub-i)*(s sub-i) gives you the total profit for the i-th item (which is just 1 item in the sequence of items). So the total profit is to add the profit of each of these items together until you reach n-items (n = all the items).

The verbiage in this problem is tricky because you think of being done when you multiply p*s because multiplying profit and how many items sold gives you the total profit, right? In this case, no because that's just 1 item in the potential n-catalog of items. So let's say you sell utensils. p*s might only represent your profit selling 1 utensil (forks), but does not include the number of knives and spoons you've sold. The summation notation adapts it for all profit for all different kinds of items so we're adding the p*s of forks with the p*s of spoons and the p*s of knives (and etc.) until it captures all the items you've profited.

$$\text{Total profit} = \sum_{i=1}^{n} p_i s_i = p^T s$$

p transpose s where this is the dot product to represent the profit vector of p relative to sales vector of s

4. (10 pts) Explain the solution to 1.16 here in your own words. (Since you are given a solution, you will be graded on your ability to explain).

1.16 *Inner product of nonnegative vectors.* A vector is called *nonnegative* if all its entries are nonnegative.

(a) Explain why the inner product of two nonnegative vectors is nonnegative.

Let a and b be two n-dimensional non-negative vectors, meaning that a sub-i >= 0 and b sub-i >= 0 for i = 1, 2, ... until n. The inner product of a and b is a transpose b. Each term at a sub-i and b sub-i in the summation notation is going to be the product of two non-negative numbers, which results in a non-negative number. Then when you add the non-negative terms together, since all the inputs are non-negative, this will also result in a non-negative number so that the inner product is going to be non-negative.

(b) Suppose the inner product of two nonnegative vectors is zero. What can you say about them? Your answer should be in terms of their respective sparsity patterns, *i.e.*, which entries are zero and nonzero.

If a tranpose b = 0, then for each i (a sub-i)*(b sub-i) is >= 0. The only way their sum can be zero is if (a sub-i) * (b sub-i) = 0 for every i. This means that for each i, at least one of a sub-i or b sub-i must be zero. In terms of sparsity, the nonzero entries of a and b must occur disjointly, in other words, if a sub-i > 0, then b sub-i has to be = 0 and vice versa.

If the inner product is zero, the nonzero entries of the 2 vectors cannot overlap with each other which means that their nonzero elements must be disjointed.

To give an example with tug of war where there's team A and team B. Team A = [1, 0, 0, 1, 0] where 1st and 4th person is pulling with all their strength and team B = [0, 1, 1, 0, 1] where 2nd, 3rd, 5th person is pulling with all their strength. When team A pulls, but team B does nothing, their contribution is 1*0 = 0. When Team B pulls but team A doesn't do anything, their contribution is 0*1 = 0. So the inner product ends up being a transpose b = 0. The two teams never get to oppose each other at the same spot and the pulling efforts never overlap, so when one team pulls, the other is doing nothing.

5. (50 points) Create a Jupyter Notebook of the following:

The Python **Companion** (linked at the top of the course) includes examples of code that can help you throughout the course and provides the basis for the fundamentals of vector operations we will use throughout the course.

Every week you can use these code elements to illustrate concepts and ideas from the weeks.

This week we guide you through an example.

1. Find the Python Companion and sections 1.1 - 1.4,

2. Select at least 10 code blocks (you may want to do all of them) that look useful.

3. Rewrite (best) or copy and paste each code cell **into your own Jupyter Notebook**.

4. Add notes in your own words in a text block above about the code using the notes given AND incorporate ideas from the book (cite the book page).

5. For each section of code include at least one additional specific example of using the code.

6. **Attach a PDF of your Jupyter Notebook here** to include in the single PDF study guide.

study guide 1

January 22, 2025

Chapter 1.1

[6]:
```
# page 2

x = [-1.1, 0.0, 3.6, -7.2] # vectors can be numbers, positive, negative, decimal
len(x) # built in counter, basically a summation notation
```

[6]:
```
4
```

[7]:
```
y = [1, 0.5, 0.1, 2] # example vector 2

x*y # you can't multiply 2 vectors, and that's reflected in python as well!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16144\3597416476.py in <module>
      1 y = [1, 0.5, 0.1, 2] # example vector 2
      2
----> 3 x*y # you can't multiply 2 vectors, and that's reflected in python as␣
 ↪well!

TypeError: can't multiply sequence by non-int of type 'list'
```

[8]:
```
import numpy as np

# testing for vector equality but trying different examples

x = np.array([-1.0, 0.0, 3.6, -7.2])
y = np.array([-1, 0, 3.6, -7.20])

x == y

# so comparing vectors is floating point agnostic?
```

[8]:
```
array([ True,  True,  True,  True])
```

[9]:
```
# experimenting with nested vectors in python

a = np.array([1, 2, 3, [3, 4]])
b = np.array([1, 2, 3, 5])

a*b
```

```
C:\Users\raiu\AppData\Local\Temp\ipykernel_16144\137072018.py:3:
VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences
(which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths
or shapes) is deprecated. If you meant to do this, you must specify
'dtype=object' when creating the ndarray.
  a = np.array([1, 2, 3, [3, 4]])
```

[9]:
```
array([1, 4, 9, list([3, 4, 3, 4, 3, 4, 3, 4, 3, 4])], dtype=object)
```

[10]:
```
# cannot add different size vectors in python

c = np.array([1, 2, 3])
d = np.array([1, 2, 3, 5])

c + d
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16144\494062773.py in <module>
      4 d = np.array([1, 2, 3, 5])
      5
----> 6 c + d

ValueError: operands could not be broadcast together with shapes (3,) (4,)
```

[11]:
```
# but can just add them together if they're the same size, follows the linear␣
 ↪algebra rules
c = np.array([1, 2, 3])
d = np.array([9, 6, 9])

c + d
```

[11]:
```
array([10,  8, 12])
```

[12]:
```
# page 6

# this is different than concatenate
e = np.concatenate((c,d))
print(e)
```

```
[1 2 3 9 6 9]
```

[13]:
```
# page 6

# vector slicing works to access certain elements in vector
# how does this work a little differently than normal python array?
# numpy array is wrapped in parantheses before the brackets

w = np.array([1,8,3,2,1,9,7])
u = w[1:4]
print(u)
```

```
[8 3 2]
```

[14]:
```
w[3:6] = [100, 200, 300]
print(w)

# why does the output not have commas? is that something with numpy?
# we took the original array and replaced i = 3:6 with 3 new elements
# notice the extra space between the digits added and original digits?
```

```
[  1   8   3 100 200 300   7]
```

[15]:
```
w[3:7] = ['a', 'b', 'c', 'd']
print(w)

# cannot be letters since numpy array is numbers?
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_16144\3212277145.py in <module>
----> 1 w[3:7] = ['a', 'b', 'c', 'd']
      2 print(w)
      3
      4 # cannot be letters since numpy array is numbers?

ValueError: invalid literal for int() with base 10: 'a'
```

[16]:
```
w[3:7] = [99, 91, 98, 0.01]
print(w)

# must have enough elements to satisfy from 3 -> 7
# why did the last number print as "0"?
```

```
[ 1  8  3 99 91 98  0]
```

Chapter 1.2 Vector Addition

[17]:
```
# page 10

import numpy as np
x = np.array([1,2,3])
y = np.array([100,200,300])
print('Sum of arrays:', x+y)
print('Difference of arrays:', x-y)
```

```
Sum of arrays: [101 202 303]
Difference of arrays: [ -99 -198 -297]
```

[18]:
```
x = np.array([1.001,2.99,3.999])
y = np.array([1,2,3])
print('Sum of arrays:', x+y)
print('Difference of arrays:', x-y)

# earlier, it rounded my digits, but seems like it's accurate now?
```

```
Sum of arrays: [2.001 4.99  6.999]
Difference of arrays: [0.001 0.99  0.999]
```

Chapter 1.3 Scalar Vector Multiplication

[19]:
```
# page 11
x = np.array([1,2,3])
print(x/2.2)
```

```
[0.45454545 0.90909091 1.36363636]
```

[20]:
```
x = np.array([1,2,3])
print(x/0.012)

# odd formatting, i assume from np
```

```
[ 83.33333333 166.66666667 250.        ]
```

[21]:
```
x = np.array([1,2,3])
print(x/0.012222222222222222222222222222)

# testing for floating point accuracy
```

```
[ 81.81818182 163.63636364 245.45454545]
```

[22]:
```
x = np.array([1,2,3])
print(x/0.012222222222222222222222222224444444444444444444444444444)

# why does it seem like it's not that accurate?
```

```
[ 81.81818182 163.63636364 245.45454545]
```

[23]:
```
# page 12
a = np.array([1,2])
b = np.array([3,4])
alpha = -0.5
beta = 1.5
c = alpha*a + beta*b
print(c)
```

```
[4. 5.]
```

[24]:
```
o = np.array([222,2222])
p = np.array([2.000000001,0.0000001])
triple = -0.005
penta = 1.0000505
e = triple*o + penta*p
print(c)
```

```
[4. 5.]
```

[25]:
```
# take the steps above and turn it into a function
def lincomb(coef, vectors): # input is the coefficient + vectors
    n = len(vectors[0]) # n = length of first vector
    comb = np.zeros(n) # combination is zeros? what is zeros? must be something␣
 ↪from numpy library
    for i in range(len(vectors)): # for loop to iterate the vector length
        comb = comb + coef[i] * vectors[i] # then update teh combination
    return comb

lincomb([alpha, beta], [a,b]) # the coefficient is passed as list as 1 argument
# the vectors is passed as a list for another argument
```

[25]:
```
array([4., 5.])
```

[26]:
```
# experimenting with different number to see how this changes the linear comb␣
 ↪result

r = np.array([1,2])
t = np.array([3,4])
alpha = -1.111111111
beta = 2.666666

lincomb([alpha, beta], [a,b])

# comb = comb + coef[i] * vectors[i] <-- this is the bulk of the logic
```

[26]:
```
array([6.88888689, 8.44444178])
```

Chapter 1.4 Inner Product

[27]:
```
# page 14

import numpy as np
x = np.array([-1,2,2])
y = np.array([1,0,-3])
print(np.inner(x,y))

# inner product is already a method in numpy! don't have to rebuild from␣
 ↪scratch every time
```

```
-7
```

[28]:
```
# page 14

x = np.array([-1,2,2])
y = np.array([1,0,-3])
x @ y

# "@" is used to perform inner product on numpy arrays, but they have to be in␣
 ↪np.array format
```

[28]:
```
-7
```

[29]:
```
(x/2)@ y

# can do any operation before inner product
# treat this like any other math arithmetic and follows PEMDAS order
```

[29]:
```
-3.5
```

Chapter 1.5 Complexity of Vector Computation

[30]:
```
# page 15

import numpy as np
a = np.random.random()
b = np.random.random()
lhs = (a+b) * (a-b)
rhs = a**2 - b**2
print(lhs - rhs)

# the left hand side and the right hand side are not exactly perfectly equal
# and this is due to the # of floating point operations (and how it rounds off
# errors)

# this is super relevant to what im doing as my full-time job because we're
```

```
# building hardware to process FP8 on ASIC dedicated to transformers
```

```
1.1102230246251565e-16
```

[31]:
```
# can compare complexity

import numpy as np
import time
a = np.random.random(10**5)

b = np.random.random(10**5)
start = time.time()
a @ b
end = time.time()
print(end - start)

# this is supposed to be "0.0006489753723144531"
```

```
0.0004971027374267578
```

[32]:
```
start = time.time()
a @ b
end = time.time()
print(end - start)

# this is supposed to be "0.0001862049102783203" according to the book, but it␣
 ↪rounded down?
```

```
0.0005202293395996094
```

The first inner product, of vectors of length 105 , takes around 0.0006 seconds. This can be predicted by the complexity of the inner product, which is 2n−1 flops. The computer on which the computations were done is capable of around 1 Gflop/s. These timings, and the estimate of computer speed, are very crude

[33]:
```
# re-writing the code above in more clearer way to experiment

# generate two random arrays
aa = np.random.random(10**7)
bb = np.random.random(10**7)

# using the numpy @ operator
start = time.time()
result_numpy = aa @ bb
end = time.time()
print(f"NumPy dot product time: {end - start:.6f} seconds")

# manual implementation of the dot product
```

```
start = time.time()
result_manual = sum(aa[i] * bb[i] for i in range(len(aa)))
end = time.time()
print(f"Manual dot product time: {end - start:.6f} seconds")

# check if the results are the same
print(f"Results match: {np.allclose(result_numpy, result_manual)}")
```

```
NumPy dot product time: 0.003478 seconds
Manual dot product time: 1.498940 seconds
Results match: True
```

Theoretically, the dot product has a time complexity of O(n), where n is the size of the vectors. However, real-world performance depends on implementation details, such as how the computation is optimized at the hardware and software level. By comparing methods: You see how libraries like NumPy leverage optimized C-based routines (e.g., BLAS) to achieve significant speedups. This shows how theoretical complexity does not always reflect real-world runtime due to optimizations.