Homework 8: Vectors

Due:

All the directions from homework 2 apply to this homework. In particular, remember to hand in a printout for procedures, and either a calculation or a transcript showing your testing.

We highly recommend that you put each problem in a separate file, and when making a transcript use the Scheme procedure `load` to read that file in to the Scheme interpreter.

Code for the vector problems is in the file `ch9.ss` in the directory `/home/cs227/lib`.

1 Section 9.2, Vectors

1. (0 points; suggested practice) Play with examples like those on pages 271 and 274 – 276 of the text.

2. (8 points; basic) A basic point about vectors is that the value of a vector can be thought of as a function. Such functions can be represented in Scheme by `lambda` expressions.

   For example, the vector viewed at prompt `[2]` on page 269 of the text can be thought of as the function represented by `(lambda (i) (add1 i))` which is the same as `add1`. Of course, since vectors are finite functions, a better representation might be

   ```
   (lambda (i)
     (if (and (<= 0 i) (<= i 5))
         (add1 i)
         (error i " is not a valid index")))
   ```

   Similarly the vector at prompt `[6]` on page 269 can be thought of as the function

   ```
   (lambda (i)
     (if (and (<= 0 i) (<= i 3))
         (* i i)
         (error i " is not a valid index")))
   ```

   To test this, one can use `view` and `vector-generator`. For example,

   ```
   > (load "ch9.ss")
   > (view ((vector-generator
             (lambda (i)
               (if (and (<= 0 i) (<= i 3))
                   (* i i)
                   (error i " is not a valid index"))))
            4))
   #(0 1 4 9)
   >
   ```

   Write down and test similar Scheme representations for the vectors viewed at (a) prompt `[7]` on page 269, (b) prompt `[2]` on page 271.

3. (0 points; suggested practice) Do the same as the previous exercise for prompts [3] and [4] on page 271.

4. (5 points; extra credit only) In the previous 2 problems (even in the examples) there is much repeated code. How could you eliminate the repetition? (Give code to do it.)

5. (3 points) Write a procedure `vector-stretch-fill` that is like `vector-stretch` (program 9.5), but takes an additional argument, `fill-value`. The `fill-value` is used as the initial value for elements of the result for which there is no corresponding element in the argument `vec`. For example, if `v` is the result of `(vector 'a 'b 'c)`, then `(view (vector-stretch-fill v 6 'new))` prints `#(a b c new new new)`.

6. (3 points; basic) Define a procedure `vec-` that takes two vectors as arguments and returns a vector. You may assume the arguments have the same length. If the argument vectors have values $\langle a_i \rangle_{i=0,\ldots,n-1}$ and $\langle b_i \rangle_{i=0,\ldots,n-1}$, then the result has value $\langle a_i - b_i \rangle_{i=0,\ldots,n-1}$. For example,

```
> (view (vec- (vector 1 2 3 4) (vector 0 1 2 3)))
#(1 1 1 1)
```

7. (4 points; basic) The *weighted score* of a vector of data, $\langle d_i \rangle_{i=0,\ldots,n-1}$, with weights $\langle w_i \rangle_{i=0,\ldots,n-1}$ and possible scores $\langle p_i \rangle_{i=0,\ldots,n-1}$, is given by the formula

$$\frac{(w_0 \times d_0) + \ldots + (w_{n-1} \times d_{n-1})}{(w_0 \times p_0) + \ldots + (w_{n-1} \times p_{n-1})}.$$

Write a procedure `weighted-score` that takes a vector of numbers, `data`, and a vector of numbers between 0.0 and 1.0, `weights`, and a vector of numbers, `possible`, and returns the weighted score of the data. You can assume that all vectors have the same length, that not all of the weights are 0.0, and that the sum of the possible scores is greater than 0.0. For example, `(weighted-score (vector 90 100) (vector 0.3 0.7) (vector 100 100))` returns 0.97.

8. (9 points; extra credit only) Write a procedure `interactive-weighted-score` that takes a vector of numbers between 0.0 and 1.0, `weights`, and a vector of numbers, `possible`, and which: prompts for data scores, prints the weighted score of the data, then prompts, and prints in this way until "quit" is entered instead of a score, when it quits and prints "Bye." For example,

```
> (interactive-weighted-score (vector 0.3 0.7) (vector 100 100))
Data 1? 90
Data 2? 100
The weighted score of #(90 100) is: 0.97
Data 1? quit
Bye.
```

9. (6 points) In physics, one often uses vectors of 3 numbers to represent locations in space, or forces etc. that are directed in space. The number 3 is special because our universe has 3-dimensional space, where the 3 directions are called x, y, and z. In physics, such vectors are so common they are usually just called "vectors". For example, the laws of mechanics are summarized by the vector equation $\vec{F} = m\vec{a}$, where $\vec{F} = \langle F_x, F_y, F_z \rangle$ is a vector of 3 numbers, representing the force vector, m is the mass, and $\vec{a} = \langle a_x, a_y, a_z \rangle$ is the acceleration vector.

Write a procedure, `force`, that takes a number, `m`, a vector of 3 numbers, `a`, and returns the vector of 3 numbers $\langle m \times a_x,\ m \times a_y,\ m \times a_z \rangle$ that represents the force vector. For example,

```
(force 3.0 (vector 5.2 3.1 8.7)) ==> #(15.6 9.3 26.1)
(force 72 (vector -8.2 -3.14 -32.0)) ==> #(-590.4 -226.08 -2304.)
```

Note that your answers will not match these exactly, due to the approximations of Scheme's inexact (floating point) numbers, but that is okay.

By the way, Geordi La Forge (of Star Trek the next generation) wants you to be sure that he can use your program for universes with different numbers of dimensions.

10. (10 points) The "fields" discussed in physics can be thought of as curried vector functions. For example, to find the gravitational field generated by some mass, you give it the mass, and get a function that takes a radius vector, and returns a function that takes a mass and gives a force vector. That is, the type of a gravitational field procedure would be

```
mass -> (radius-vector -> (mass -> force-vector))
```

Here `mass` is a type of numbers, and both `radius-vector` and `force-vector` are 3-element vectors of numbers (presumably in units of length and force, respectively). Write a procedure `grav-field` that has this type, and such that `(grav-field` m_1`)` is the gravitational field produced by m_1. (This can be thought of as associating to each location in space and to each mass placed at that location, a force vector.) That is, `(((grav-field` m_1`)` r`)` m_2`)` $= \langle -Gm_1m_2/r_x^2, -Gm_1m_2/r_y^2, -Gm_1m_2/r_z^2 \rangle$, where $r = \langle r_x, r_y, r_z \rangle$ is the radius vector, and G is the universal gravitational constant. You can assume that the MKS (or SI) system is used for all dimensions, and thus that $G = 6.670 \times 10^{-11}$ (with units $\mathrm{Nm}^2/\mathrm{kg}^2$). (Don't worry if you don't understand that last sentence; just use 6.670×10^{-11} as the value of G.) If any component of the radius vector is zero, the result is defined to be zero in that component. For example,

```
(((grav-field 1) (vector 1 0 1)) 1) ==> #(-6.67e-11 0.0 -6.67e-11)
(((grav-field 1) (vector 1 2 3)) 10)
    ==> #(-6.67e-10 -1.6675e-10 -7.41111111111111e-11)
(((grav-field 0.001) (vector 0.05 0 0)) 0.5)
    ==> #(-1.3339999999999998e-11 0.0 0.0)
(let ((mass-of-earth 5.96e24) ; kg
      (radius-of-earth 6370e3)) ; km
  (((grav-field mass-of-earth) (vector 0 0 radius-of-earth)) 1))
     ==> #(0.0 0.0 -9.79700272815321)
```

Note that these answers are inexact. (By the way, the units in the results are Newtons; the last result says that a 1kg mass placed at the earth's surface feels a force of about 2.2 pounds.)

11. (8 points; extra credit only) Do the same as the previous problem for the electric field generated by a point charge.

2 Section 9.3, pages 278–283, Representing Vectors

12. (5 points; basic) Do exercise 9.4. You can test just the built-in representation of vectors. You can handwrite your answer to the last question.

13. (0 points; suggested practice only) Do exercise 9.2.

14. (5 points; extra credit only) Do exercise 9.3. You can ignore the last sentence.

15. (6 points; extra credit only) Barbara Liskov thinks that the elements of a vector should be printed abstractly. She points out that if you print a vector of vectors using `view`, you'll see the internal representation printed. Solve her problem by writing a procedure `view-abs` that takes a procedure, `element-print`, and a vector, `vec`, as arguments, and prints `vec`, using `element-print` to print the elements. For example,

```
> (view-abs view (vector (vector 1 2) (vector 3 4)))
#(#(1 2)
#(3 4))
> (view-abs write (vector "printed" "with" "quotes"))
#("printed" "with" "quotes")
```

16. (0 points; suggested practice only) Do exercise 9.5.

3 Section 9.3, pages 283–289, Imperative-Style Programming

For some of the problems below, we will tell you to write both a functional version and an imperative version. Functional versions are to be done without using mutation, often using `vector-generator`. Imperative versions are to be done using mutation (`vector-set!` or `vector-update!`), and without using `vector-generator`. Instead, use `make-vector` to create an empty vector, and mutate it in a loop. You can give each procedure different names: put `-imp` on the end of the name of the imperative one. An `-imp` procedure is different than a `!` procedure, as usually the `-imp` procedures do not modify their arguments; if necessary, use `vector-copy` to make a copy of the argument, but it is often more efficient to just never mutate the argument.

17. (10 points; basic) Do exercise 9.1, writing both a functional and an imperative version. As an example of the imperative version,

```
((successive-powers-imp 3) 5) ==> #(1 3 9 27 81)
```

18. (10 points; basic) Write an imperative version of `multiply-by-scalar`, call it `multiply-by-scalar-imp`. (Hint, you can do this by writing an imperative-style version of `vector-map`, call it `vector-map-imp`.) For example,

```
> (define my-vec (vector 1 2 3))
> my-vec
#(1 2 3)
> (multiply-by-scalar-imp 7 my-vec)
#(7 14 21)
> my-vec
#(1 2 3)
```

19. (10 points) Write an imperative version of `vector-apply-elementwise-to-both`, and use it to define `vec+-imp` and `vec*-imp`.

    For example,

```
> (define vec1 (vector 1 2 3))
> (define vec2 (vector 7 6 5))
> ((vector-apply-elementwise-to-both-imp *) vec2 vec1)
#(7 12 15)
> vec1
#(1 2 3)
> vec2
#(7 6 5)
> (vec+-imp vec2 (vector 1 2 0))
#(8 8 5)
> (vec*-imp vec2 vec1)
#(7 12 15)
> vec1
#(1 2 3)
> vec2
#(7 6 5)
```

20. (0 points; suggested practice only) Use `vector-generator` to define `vector-append` and `vector-reverse`. For example,

```
(vector-append (vector 1 2 3) (vector 4 5)) ==> #(1 2 3 4 5)
(vector-reverse (vector 1 2 3)) ==> #(3 2 1)
(vector-reverse (vector )) ==> #()
```

(The point of this is to contrast the imperative-style programs in the book with the functional style programs you will write.) Check your work on the computer.

21. (3 points; extra credit only) Use unrestricted lambda to define `vector-append` so that it appends any number of vectors.

22. (6 points; basic) Define a procedure `multiply-by-scalar!` that takes a number, `c`, and vector of numbers, `vec`, and mutates `vec` so in the final state, the ith element of `vec` is the result is `c` times the initial ith element of `vec`. Note that nothing is returned by `multiply-by-scalar!`, that is, it has type:

```
number, (vector of number) -> void
```

For example,

```
> (define my-vec (vector 3 4 5 6))
> my-vec
#(3 4 5 6)
> (multiply-by-scalar! 2 my-vec)
> my-vec
#(6 8 10 12)
> (multiply-by-scalar! 2 my-vec)
> my-vec
#(12 16 20 24)
> (multiply-by-scalar! 3 (vector 2 7))
> my-vec
#(12 16 20 24)
```

23. (5 points) This problem is related to the Towers of Hanoi. Recall that a state of the puzzle consists of 3 integers, telling how many disks are on the left, center, and right posts. In this problem we will represent that state using a vector containing 3 integers. The contents of a state vector at index 0 represents the number of disks on the left post, at index 1 the center, and at 2 the right. Write a procedure, `tower-move!` that takes two symbols (either `l`, `c`, or `r`), `src` and `dest`, and returns a procedure that takes a state, `posts`, and mutates the vector so that the final value of `posts` has one less disk on `src` and one more on `dest`. For example,

```
> (define posts #(2 0 0))
> posts
#(2 0 0)
> ((tower-move! 'l 'c) posts)
> posts
#(1 1 0)
> ((tower-move! 'l 'r) posts)
> posts
#(0 1 1)
> ((tower-move! 'c 'r) posts)
> posts
#(0 0 2)
```

```
> ((tower-move! 'l 'r) (vector 7 0 0))
> posts
#(0 0 2)
```

24. (15 points; extra credit only) Write an imperative procedure `vector-tower-of-hanoi`, that is like `tower-of-hanoi`, except that instead of returning a list of moves, it returns a list of 3-element vectors. Each 3 element vector represents the state of the problem: how many disks are on the left, center, and right needles. For example, `#(3 2 1)` represents the state where there are three disks on the left needle, 2 on the middle, and one on the right. For example,

    ```
    (vector-tower-of-hanoi 2) ==> (#(2 0 0) #(1 1 0) #(0 1 1) #(0 0 2))
    ```

    Hint: use `vector-copy` and variable `posts` that is outside the recursion, holding the state.

25. (10 points) Define a procedure `vector-for-each` that takes a procedure of type `T -> void`, and a vector of type `T`, applies the procedure to each element of the vector. For example,

    ```
    > (define avec (vector 1 10 100))
    > (begin
        (vector-for-each (lambda (elem) (write elem) (display ", ")) avec)
        (newline))
    1, 10, 100,
    > (let ((sum (vector 0)))
        (let ((set-it! (lambda (x) (vector-set! sum 0 x)))
              (get-it (lambda () (vector-ref sum 0))))
          (vector-for-each
            (lambda (elem) (set-it! (+ (get-it) elem)))
            avec)
          (get-it)))
    111
    > avec
    #(1 10 100)
    ```

26. (15 points; extra credit only) It's not obvious how to make a procedural abstraction of `for-each` and `vector-for-each`. Can you do it? If so, write the code and show how to use it to define both `for-each` and `vector-for-each`.

27. (10 points; extra credit only) The `vector-for-each` program does not give the caller access to the vector itself, because there is no way to get at the index for an element. On the other hand, if the programmer had access to the indexes, then getting the elements would be easy. To that end write a procedure `vector-for` that takes a procedure `proc` of type `natural -> void` and a vector, and which has no return value. For example,

    ```
    > (define avec (vector 1 10 100))
    > (vector-for (lambda (i) (writeln i)) avec)
    0
    1
    2
    > (vector-for (lambda (i) (vector-set! avec i 0)) avec)
    > avec
    #(0 0 0)
    ```

    Use `vector-for` to write `vector-for-each`.

28. (6 points; extra credit only) On the bottom of page 270, the text says that Program 9.3 is "very inefficient." Analyze the program and estimate its order (see p. 126). (Computer Scientists call this "time complexity.")

4 Matrices (9.4)

The code for matrices is gotten by loading the file for this chapter.

29. (7 points) Do exercise 9.11. Ignore the directions in the last sentence about generalizing the procedure view.

30. (3 points) Do exercise 9.8. Use `matrix-generator` to ensure that your solution is independent of the representation of matrices. For example,

```
> (matrix-view ((matrix 2 3) 1 2 3
                             4 5 6))
1 2 3
4 5 6
```

31. (10 points) Do exercise 9.9. Write both a functional version using `matrix-generator`, and an imperative version. In the imperative version, use the following procedure

```
(define make-matrix ; TYPE: all T . T -> (natural, natural -> (matrix of T))
  (lambda (fill)
    (lambda (nrows ncols)
      ((matrix-generator (lambda (i j) fill)) nrows ncols))))
```

instead of `matrix-generator`. Call the imperative version `mat+-imp`.

For example,

```
> (define m1 ((matrix-generator (lambda (i j) i)) 2 3))
> (define m2 ((matrix-generator (lambda (i j) j)) 2 3))
> (matrix-view m1)
0 0 0
1 1 1
> (matrix-view m2)
0 1 2
0 1 2
> (matrix-view (mat+ m1 m2))
0 1 2
1 2 3
> (matrix-view (mat+-imp m1 m2))
0 1 2
1 2 3
> (matrix-view m1)
0 0 0
1 1 1
> (matrix-view m2)
0 1 2
0 1 2
```

32. (10 points) Do exercise 9.10. Write both a functional version using `matrix-generator`, and an imperative version.

33. (0 points; suggested practice only) Do exercise 9.12.

34. (10 points) Do exercise 9.13.

35. (12 points; extra credit only) Do exercise 9.14. Be sure to use the technique described, instead of the technique of exercise 9.13.

36. (20 points; extra credit only) In computer graphics, one can think of the terminal screen as a large matrix. (This is especially true of a so-called "bit-mapped display".) Let us call this matrix, `*screen*`; assume that it is a global variable. In this matrix, the elements are called *pixels* (i.e., picture elements). These pixels are the dots that make up the screen. Pixels in this problem will be symbols that represent colors (e.g., `red` or `blue`). (In reality, the pixels would eventually end up being numbers.)

    Write a procedure `draw-vert-line!` that takes a color (represented by a symbol), `shade`, and returns a procedure that takes a boolean, `upwards`, and returns a procedure that takes a non-negative integer, `length`, and returns a procedure that takes a row index `starti`, and a column index `startj` and mutates the matrix `*screen*` so that `length` pixels in column `startj` are changed to have color `shade`. If `upwards` is `#t`, then the changed pixels are those with indexes (i, j) such that $j = \texttt{startj}$, and $\max(0, 1 + \texttt{starti} - \texttt{length}) \leq i \leq \texttt{starti}$. If `upwards` is `#f`, then the changed pixels are those with indexes (i, j) such that $j = \texttt{startj}$, and $\texttt{starti} \leq i \leq \min(\texttt{starti} + \texttt{length} - 1, \texttt{max-screen-row})$, where `max-screen-row` is the value of `(sub1 (num-rows *screen*))`. Note that this prevents access to elements that are outside of `*screen*`. For example, after the call

    ```
    ((((draw-vert-line! 'red) #f) 10) 400 500)
    ```

    the 10 pixels in column 500 of `*screen*` at coordinates (400,500), (401,500), ..., (409,500) are red (assuming that these indexes are legal). After the call

    ```
    ((((draw-vert-line! 'blue) #t) 50) 25 101)
    ```

    only 26 pixels of `*screen*`, those at coordinates (0,101), (1,101), ..., (25,101) are blue, as the whole line will not fit. (But this isn't an error.) Your program should check that `starti` and `startj` are legal indexes into `*screen*`, and print the error message "`Illegal index to *screen*`" if they are not. The error message and the return value of a call to `draw-vert-line!` is not specified. You may assume that `length` is at least 0. You should assume that the global variable `*screen*` is already defined.

    You should *not* assume anything about the representation of matrices; low level solutions will lose points.

5 Chapter 10

We are not going to cover chapter 10 in class this semester. But there are many interesting exercises in that chapter. These exercises concern such basic computer algorithms such as sorting and searching, and an introduction to relational databases (in section 10.4).

37. (0 points; suggested practice only) Read chapter 10. This will help you understand vectors and imperative style programming.

38. (many points; extra credit only) Do one or more of the following exercises, for the extra credit listed. Your solutions should be clearly written with good style, clearly labeled, and clearly tested (to be easier on your TA). Note that you should, as you concentrate on algorithms and programming tradeoffs described in the chapter, not just getting the right answer. 10.1 (1 point), 10.2 (4 points), 10.3 (10 points), 10.4 (20 points), 10.5 (1 point), 10.6 (5 points), 10.7 (5 points), 10.8 (3 points), 10.9 (20 points), 10.10 (15 points).