


```

==> (print-stream (smooth-n integers 0))
([STREAM] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ....
==> (print-stream (smooth-n integers 1))
([STREAM] 1.5 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5 10.5 11.5 12.5 13.5 ....
==> (print-stream (smooth-n integers 2))
([STREAM] 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ....
==> (print-stream (smooth-n integers 3))
([STREAM] 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5 10.5 11.5 12.5 13.5 ....

```

Amusing hypothesis: (1) for even numbers of smoothing, say $2k$ we get the integers starting at $k + 1$. (2) for odd numbers of smoothing, say $2k + 1$, we get the sum of the stream of 0.5 and the integers starting at $k + 1$

```

==> (print-stream (smooth-n integers 42)) ; 2*21
([STREAM] 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 ....
==> (print-stream (smooth-n integers 37)) ; 2*18 + 1
([STREAM] 19.5 20.5 21.5 22.5 23.5 24.5 25.5 26.5 27.5 28.5 29.5 30.5 ....

```

(b) We chose to generate a signal with just a bit of noise.

```

(define      4b-sig (car (make-test-signal 3)))
(define smooth-4b-sig (smooth-n 4b-sig 4))
(define diff-4b-sig (diff      4b-sig))

==> (print-stream 4b-sig)
([STREAM] 0 2 0 1 2 0 1 0 1 2 1 0 2 2 2 2 2 0 1 1 1 0 2 2 0 2 0 2 0 0 2 0 1 2 2
          1 0 2 2 1 2 2 2 0 1 1 1 1 1 0 2 2 0 0 2 1 1 0 1 2 0 2 1 0 2 1 1 2 0 1
          0 0 0 1 2 1 2 0 0 1 2 1 4987.4673 4987.4673 4988.4673 4987.4673 ....

==> (print-stream smooth-4b-sig)
([STREAM] 0.875 1.0 1.0625 0.8125 0.5625 0.625 1.0 1.25 1.0625 1.0
          1.4375 1.875 2.0 1.875 1.4375 0.9375 0.8125 0.875 0.8125 0.9375
          1.3125 1.375 1.125 1.0 1.0 0.875 0.625 0.625 0.8125 0.875
          1.125 1.5625 1.5625 1.125 1.0 1.375 1.625 1.625 1.75 1.8125
          1.4375 0.9375 0.8125 0.9375 1.0 0.9375 0.8125 0.9375 1.3125 1.25
          0.75 0.6875 1.0625 1.125 0.8125 0.6875 0.9375 1.125 1.125 1.125
          1.0 0.9375 1.125 1.25 1.25 1.125 0.8125 0.5 0.25 0.125
          0.375 0.9375 1.375 1.4375 1.125 0.625 0.5 0.9375 312.9667
          1559.521 3429.3213 4676.0635 4987.8423 4987.78 4987.905 ....

==> (print-stream diff-4b-sig)
([STREAM] 2 -2 1 1 -2 1 -1 1 1 -1 -1 2 0 0 0 0 -2 1 0 0 -1 2 0
          -2 2 -2 2 -2 0 2 -2 1 1 0 -1 -1 2 0 -1 1 0 0 -2 1 0 0
          0 0 -1 2 0 -2 0 2 -1 0 -1 1 1 -2 2 -1 -1 2 -1 0 1 -2 1
          -1 0 0 1 1 -1 1 -2 0 1 1 -1 4986.4673 0 1 -1 ....

==> (print-stream (diff smooth-4b-sig))
([STREAM] 0.125 0.0625 -0.25 -0.25 0.0625 0.375 0.25 -0.1875 -0.0625
          0.4375 0.4375 0.125 -0.125 -0.4375 -0.5 -0.125 0.0625 -0.0625
          0.125 0.375 0.0625 -0.25 -0.125 0.0 -0.125 -0.25 0.0
          0.1875 0.0625 0.25 0.4375 0.0 -0.4375 -0.125 0.375 0.25
          0.0 0.125 0.0625 -0.375 -0.5 -0.125 0.125 0.0625 -0.0625
          -0.125 0.125 0.375 -0.0625 -0.5 -0.0625 0.375 0.0625 -0.3125
          -0.125 0.25 0.1875 0.0 0.0 -0.125 -0.0625 0.1875 0.125
          0.0 -0.125 -0.3125 -0.3125 -0.25 -0.125 0.25 0.5625 0.4375
          0.0625 -0.3125 -0.5 -0.125 0.4375 312.0292 1246.5543 1869.8003

```



```

                                (smooth-n integers 3))) ; 2.5...
==> (print-stream punt6)
([STREAM] 0 0 0 0 0 0 9 10 11 12 13 14 15 ....
==> (define punt9 (get-signal-strength (mark integers 0 3)      ; punt first 9
                                (smooth-n integers 3))) ; 2.5...
==> (print-stream punt9)
([STREAM] 0 0 0 0 0 0 0 0 0 12 13 14 15 16 ....

```

Good. We expected to punt the first 6 elts of `punt6` since our mean was 0 and deviation was 2 (so we look for $2\sigma = 6$) then returned the once-more smoothed (i.e., averaged smoothed value and its successor) thrice-smoothed integers, which we know will be the integers starting at 3 (i.e., $4 = 2 \times 2$ so start at $2 + 1$). Thus, we punt 3–8 (first 6) and resume integers at 9. Similarly, `punt9` will punt the first 9 of the same stream.

Problem 7 We use `filter` to pass only those entries which are not 0 (notice that negative values *can* occur if our smoothed stream is decreasing in values. Neither `integers` nor `mystery-stream` do, but in general a stream might. We chose to group times and values by making a `cons` pair.

```

(define (dev-posns str)
  (filter (lambda (x) (not (= (car x) 0)))
    (combine-streams cons str integers)))

```

Since the stream produced by `dev-posns` is a stream of compound data— specifically, one in which each element is a pair— it would also be nice to define selectors to go along with the constructor.

```

(define (posn-value devposnstr) (car devposnstr))
(define (posn-time devposnstr) (cdr devposnstr))

```

```

==> (print-stream (dev-posns punt6))
([STREAM] (9 . 7) (10 . 8) (11 . 9) (12 . 10) (13 . 11) (14 . 12) ....

```

As expected.

Problem 8 As you can see, our above abstractions paid off...

```

(define (measure-time-spread-and-signals l-str r-str)
  (combine-streams (lambda (event1 event2)
    (list (- (posn-time event1)
            (posn-time event2))
          (posn-value event1)
          (posn-value event2)))
    l-str
    r-str))

```

Since the stream produced by this procedure is also a stream of compound data— spec., one in which each element is a triple— it would also be nice to define selectors to go along with this constructor too.

```

(define time-shift car)
(define left-value cadr)
(define right-value caddr)

```

We can test this by checking the time spread of the deviation positions of `punt6` and `punt9`. We should see time deviations of -3 since `punt6` anticipates `punt9` by three time stamps. Also, the corresponding values should always be 3 higher for `punt9` since it is essentially a three unit delayed version of `punt6`. Let's see...

```

==> (print-stream (measure-time-spread-and-signals (dev-posns punt6) (dev-posns punt9)))
([STREAM] (-3 9 12) (-3 10 13) (-3 11 14) (-3 12 15) (-3 13 16) ....

```

Roger dodger.

Problem 9 First, we define our mean and deviation to be the same as was provided in the problem set code listing, just to be on the safe side.

```
(define mean 0) ;the derivative of a bounded signal has zero average value
(define dev 20) ;who knows what evil lurks....
```

Next, let's define something to process the signals by smoothing them, marking the differentiations, and finally hacking out the signal strength and time-stamping the deviation positions.

```
(define (process-signal str smooth-factor)
  (let ((smoothed-str (smooth-n str smooth-factor)))
    (dev-posns (get-signal-strength (mark (diff smoothed-str) mean dev)
                                     smoothed-str))))
```

Now we can do the simulation by applying `compute-distance-and-angle` to each element (which is a triple!) of the stream produced by `measure-time-spread-and-signals`.

```
(define (simulate l-str r-str smooth-factor v b)
  (map (lambda (spread-n-sigs) (compute-distance-and-angle ( time-shift spread-n-sigs)
                                                           ( left-value spread-n-sigs)
                                                           (right-value spread-n-sigs)
                                                           v b))
    (measure-time-spread-and-signals (process-signal l-str smooth-factor)
                                     (process-signal r-str smooth-factor))))
```

Now lets try it out on `(make-test-signal 2)`

```
(define signals (make-test-signal 2))
(define left-signal (car signals))
(define right-signal (cdr signals))

==> (head (simulate left-signal right-signal 2 signal-velocity baseline))
(70.13121 0.77539754)
```

Looks like mickey has a distance of about 70 (in unknown units). He is oriented at an angle of 0.775 radians or about 44.4 degrees.