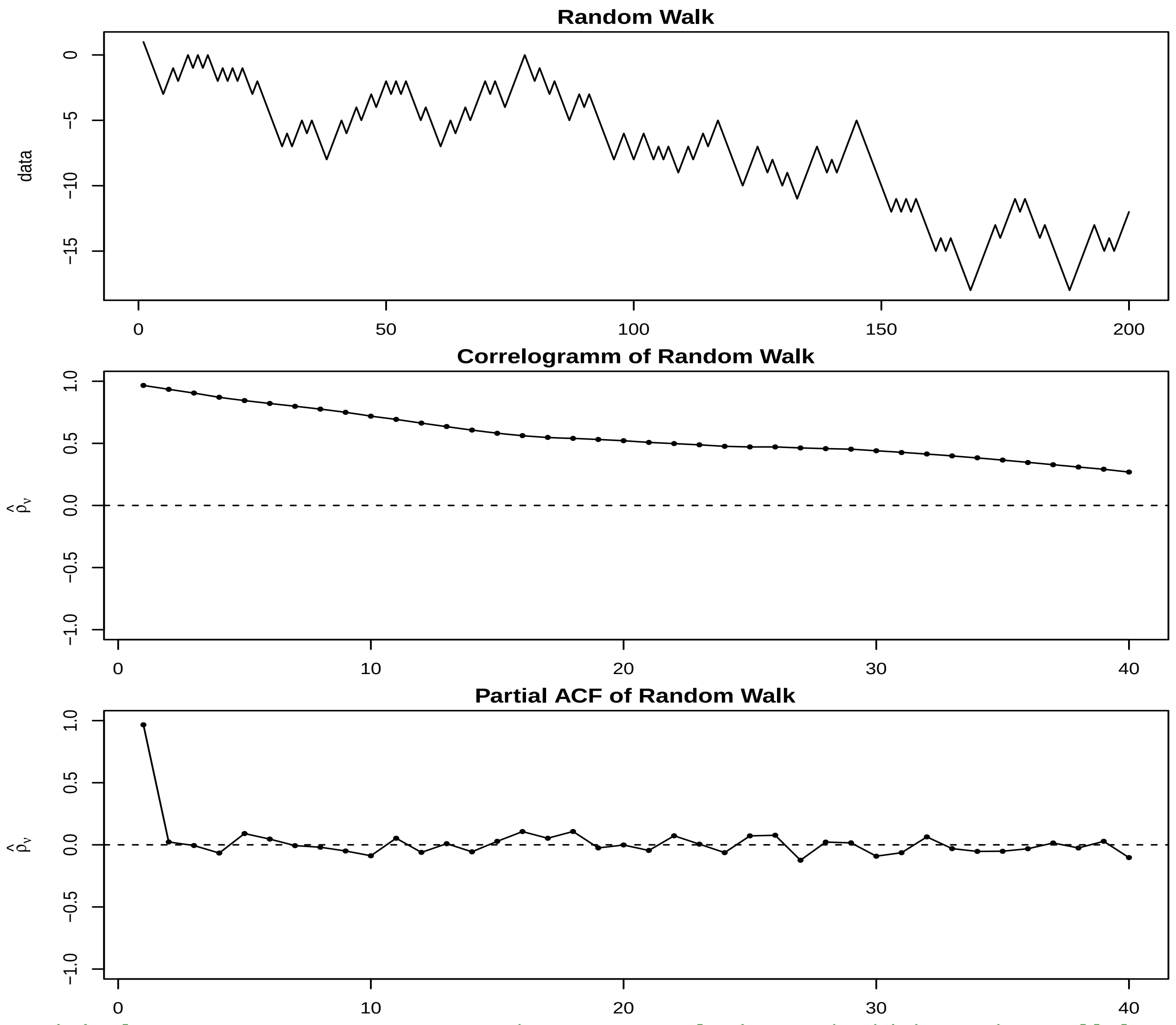
C2.3

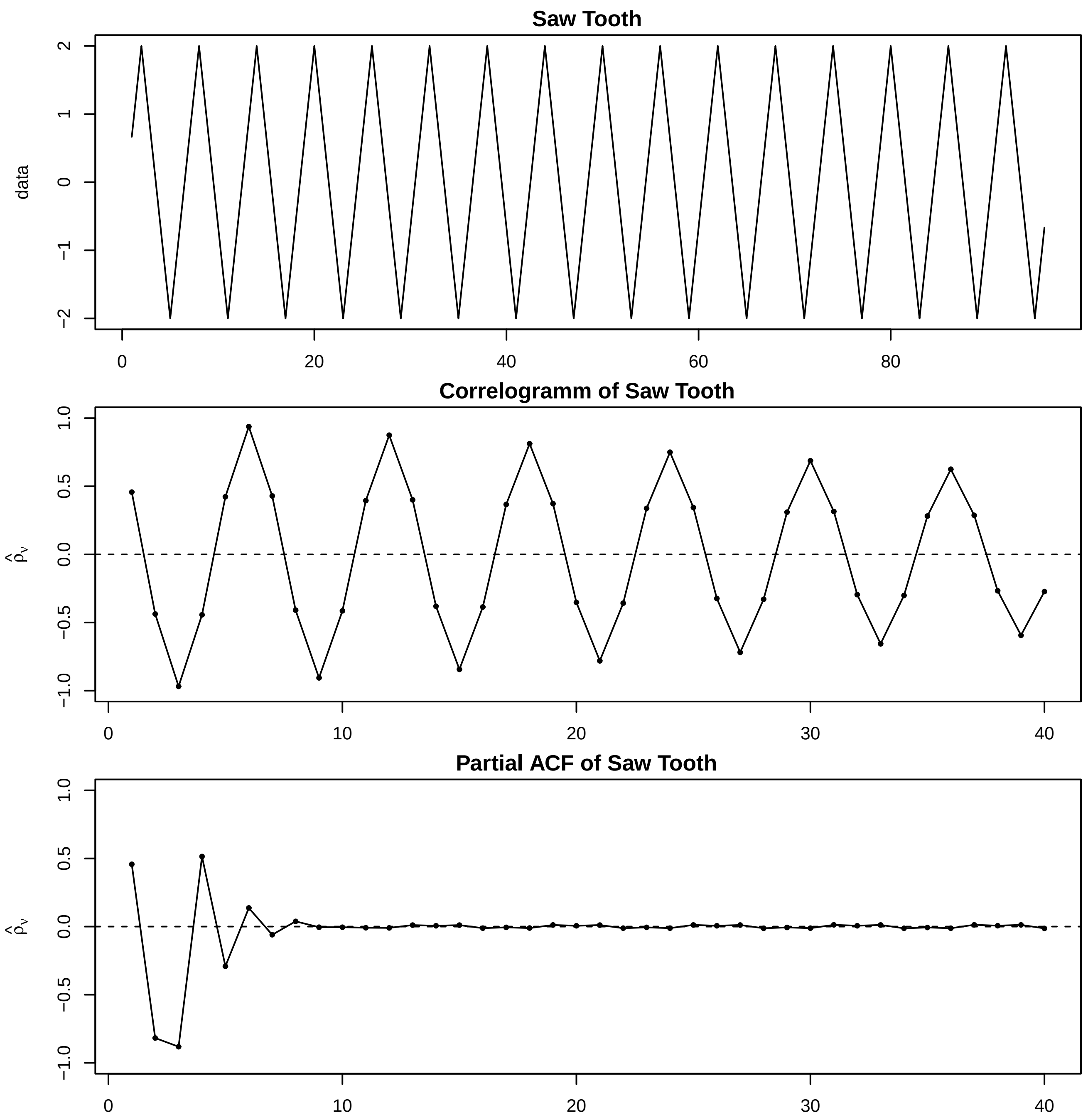
Home Work #2 by Alex Salo, Sept.2015



Since it's long term memory, we expect the auto-correlation to be high at the small lags and evade with the increase of the lag size. Correlogram confirms our expectations.

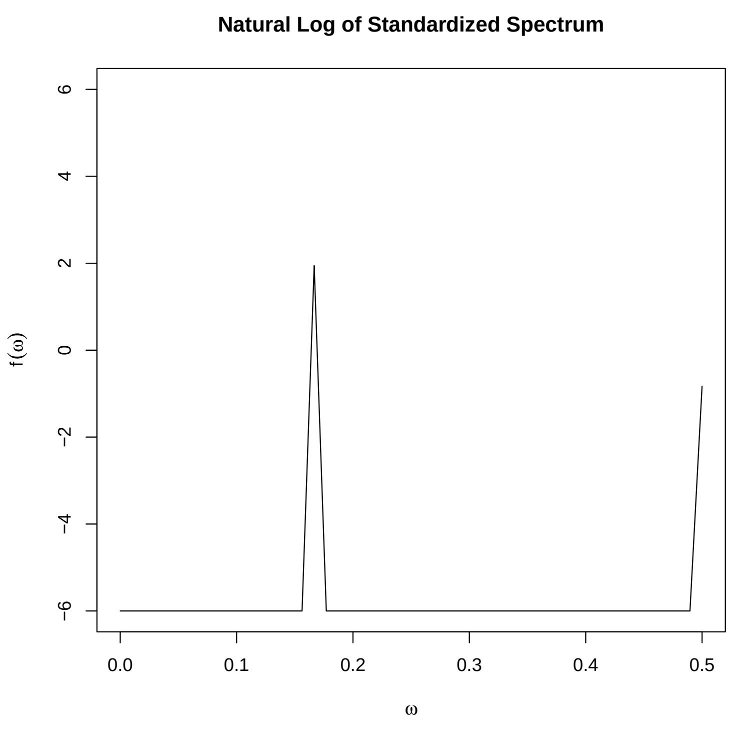
```
# Alex Salo ACF Plots
acfplots <- function(time, data, numlags, name){
      par(mfrow=c(3,1),mar = c(2,5,1.5,1) + 0.1)
      plot(time, data, type = "l", main = name)
      plotcorr(corr(data, numlags)$corr, paste("Correlogramm of", name, sep=" "))
      plotcorr(parcorr(data, numlags), paste("Partial ACF of", name, sep=" "))
}

### C2.3
random_walk <- function(n=200){
      a = wn(n,"unif")
      b = rep(-1, n)
      b[a > 0.5] = 1
      return(cumsum(b))
}
rw = random_walk(200)
acfplots(1:200, rw, 40, "Random Walk")
```

C2.6



Despite Saw Tooth is not a sine, the correlogram looks almost like a sinusoid.

```
sawtooth <- function(ncycles = 5, amp = 10){
      precision = 2 * amp
      t = seq(0, 2 * amp, length.out = precision)
      cycle = c(t[1:length(t)-1], rev(t)[1:length(t)-1]) - amp
      cycle = c(cycle[(precision / 2): length(cycle)], cycle[0: (precision / 2)])
      cycles = rep(cycle[2:length(cycle)], ncycles)
      return(cycles)
}
st = sawtooth(16, 2)
acfplots(1:length(st), st, 40, 'Saw Tooth')
```



There are two harmonics present.

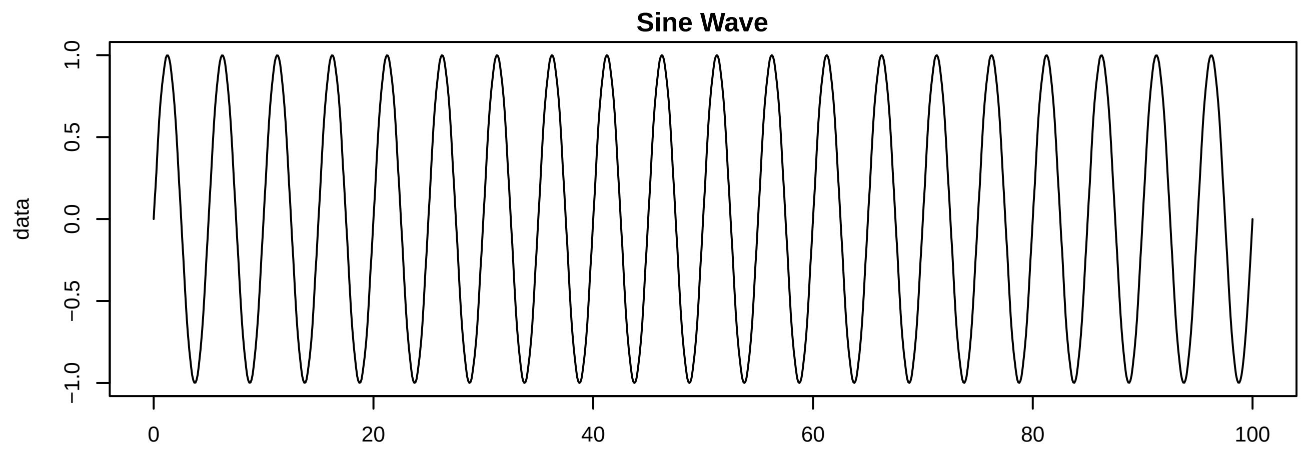
```
which(round(perdgm(st), 3) > 0)
```

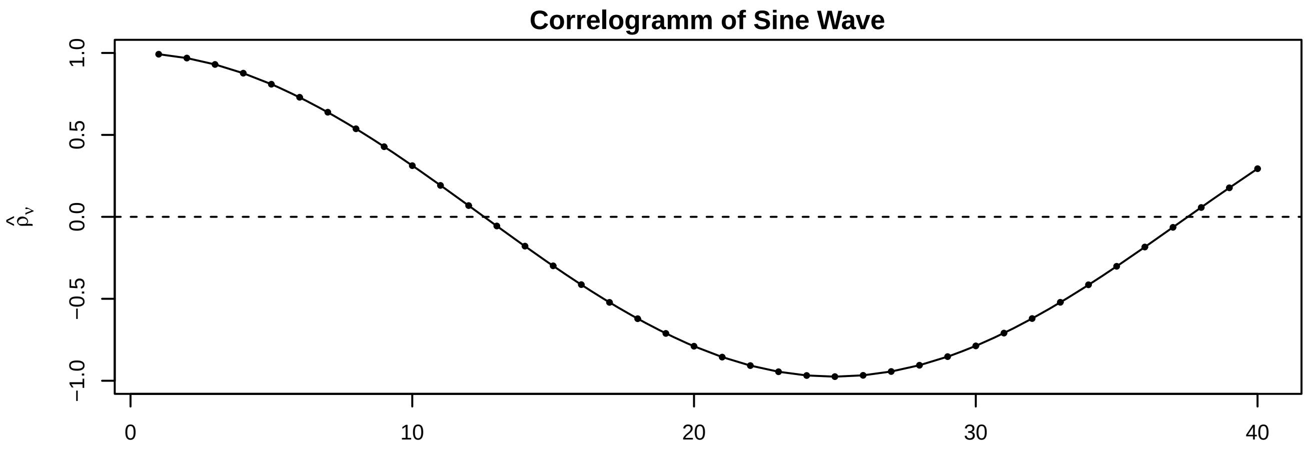
```
>> 17 49
```

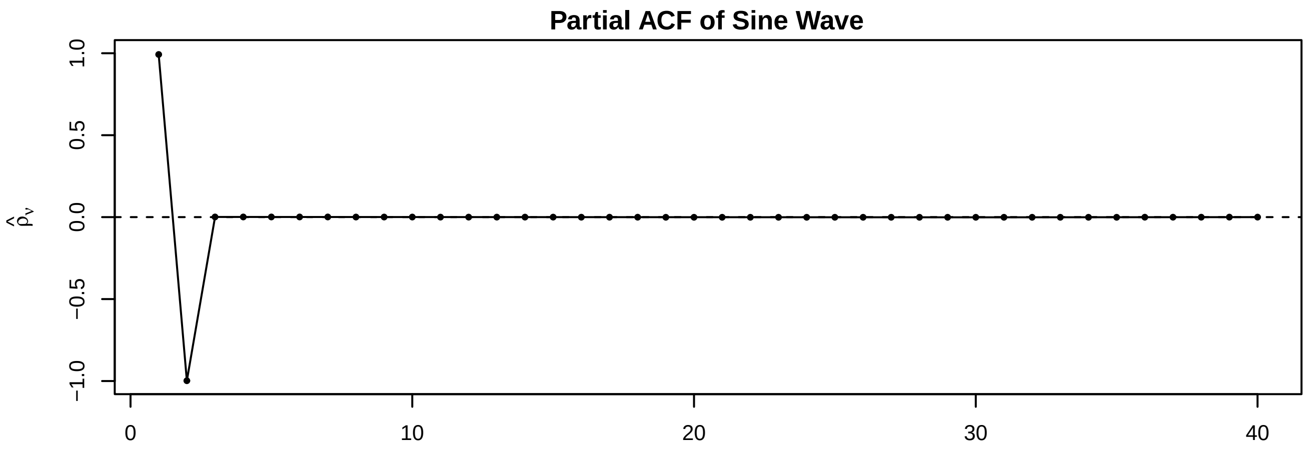
Then fundamental frequencies are (k – 1) / N: 16/96 = 1/6 and 48/96 = ½

16/96 is indeed a frequency that we wanted to have (since we have 16 cycles in 96 points of observation).

C2.7



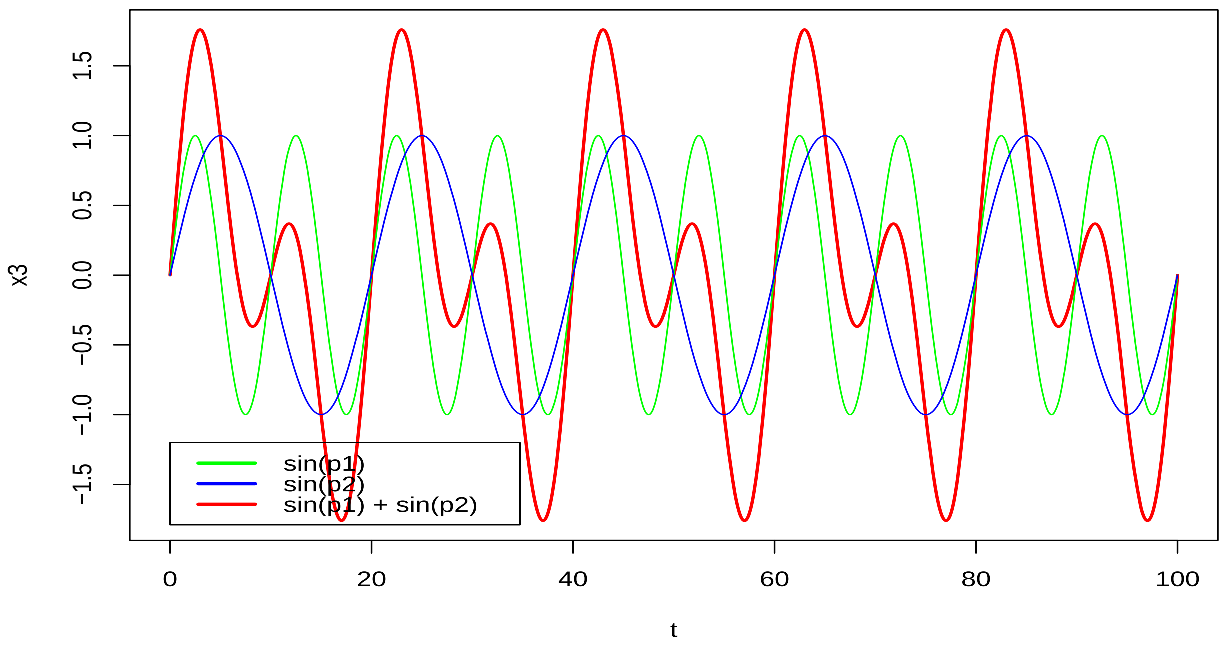




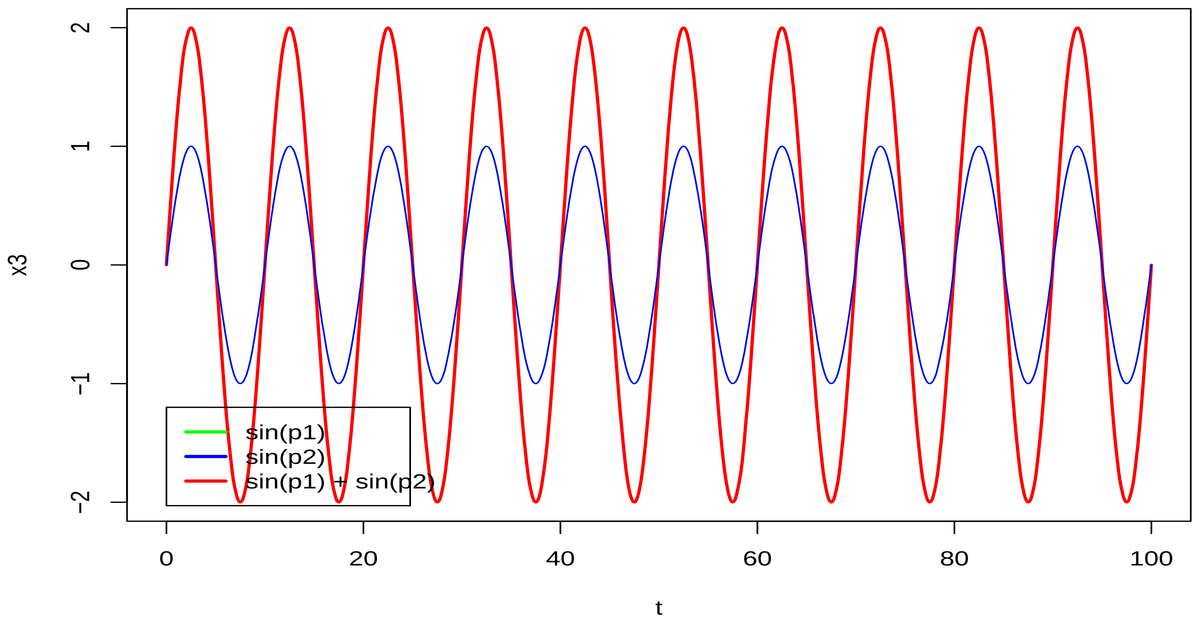
R(0) = 1 – no lag – correlation is 100%
Correlogram of sinusoid is also sinusoid
It also agrees with the results of Theorem 2.2

C2.8

a) p1 = 10, p2 = 20, **SS = 0**



b) P1 = 10, p2 = 10, **SS = 50**



By Thm 2.1 (a):

- Sum(Cos(2pi(t-1)*w_j)*Cos(2pi(t-1)*w_k)) = 0 if w_j != w_k -> confirms plot **a) SS = 0** (p1 = 10, p2 = 20, SS = 0)
- Sum(cos(w_j)*cos(w_k) = n / 2) if w_j = w_k != 0, ½ -> confirms plot **b) SS = 50** (p1=p2=10, SS = 100 / 2)

C2.12

```
### C2.12
primes = c(19991, 19993, 199999)
num_sample = c(100, 1000, 1500, 2000, 5000)
timecov <- function(n = 500, M = 40){
      x <- rnorm(n)
      t.corr <- system.time(corr(x, M))[3]
      t.corr1 <- system.time(corr1(x, M))[3]
      return(c(round(unname(t.corr), 8), round(unname(t.corr1), 2)))
}

for (prime in primes){
      for (M in num_sample){
            time = timecov(primes, M)
            print(paste("Prime: ", prime, " M: ", M, ", 2FFTs: ", time[1], ", Conv: ",
time[2], sep=""))
      }
}

[1] "Prime: 19991 M: 100, 2FFTs: 0, Conv: 0"
[1] "Prime: 19991 M: 1000, 2FFTs: 0, Conv: 0.01"
[1] "Prime: 19991 M: 1500, 2FFTs: 0, Conv: 0.08"
[1] "Prime: 19991 M: 2000, 2FFTs: 0, Conv: 0.07"
[1] "Prime: 19991 M: 5000, 2FFTs: 0, Conv: 0.41"

[1] "Prime: 19993 M: 100, 2FFTs: 0, Conv: 0"
[1] "Prime: 19993 M: 1000, 2FFTs: 0, Conv: 0.03"
[1] "Prime: 19993 M: 1500, 2FFTs: 0, Conv: 0.03"
[1] "Prime: 19993 M: 2000, 2FFTs: 0, Conv: 0.07"
[1] "Prime: 19993 M: 5000, 2FFTs: 0, Conv: 0.41"

[1] "Prime: 199999 M: 100, 2FFTs: 0, Conv: 0"
[1] "Prime: 199999 M: 1000, 2FFTs: 0, Conv: 0.04"
[1] "Prime: 199999 M: 1500, 2FFTs: 0, Conv: 0.05"
[1] "Prime: 199999 M: 2000, 2FFTs: 0, Conv: 0.06"
[1] "Prime: 199999 M: 5000, 2FFTs: 0, Conv: 0.4"
```

Number of computations for n=19991 and M = 1000:

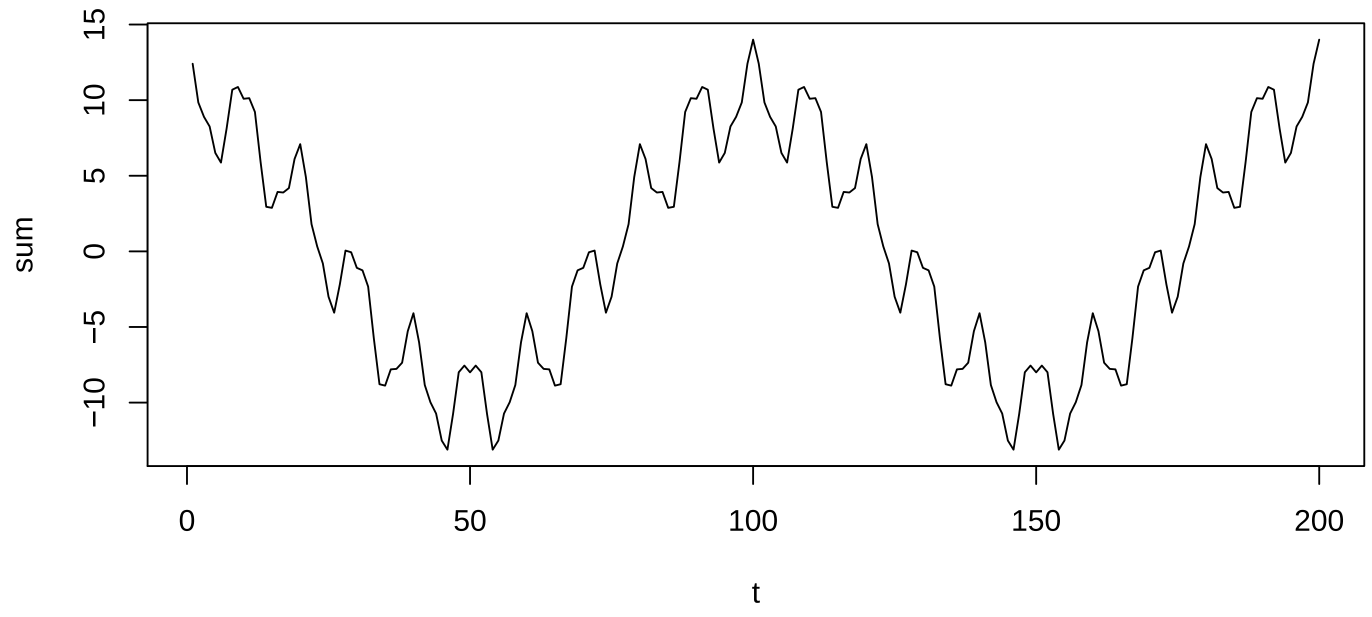
- 2FFT: 2^15 = 32768, 2^14 = 16384 -> we pad the prime 19991 (plus M=1000) with zeros to form 2^25 = 32768 (that's almost half of the data points are meaningless).
  Then the number of computations is given by: 2 (two ffts) * 32768 * log_2(32768) = **983040** computations.
- Convolution: We don't have to pad with zeros here, but we need to compute all the pairs thus doing ~n^2 computations which gives us 19991*(1000+1) - (1000*1001)/2 = **19510491** computations.

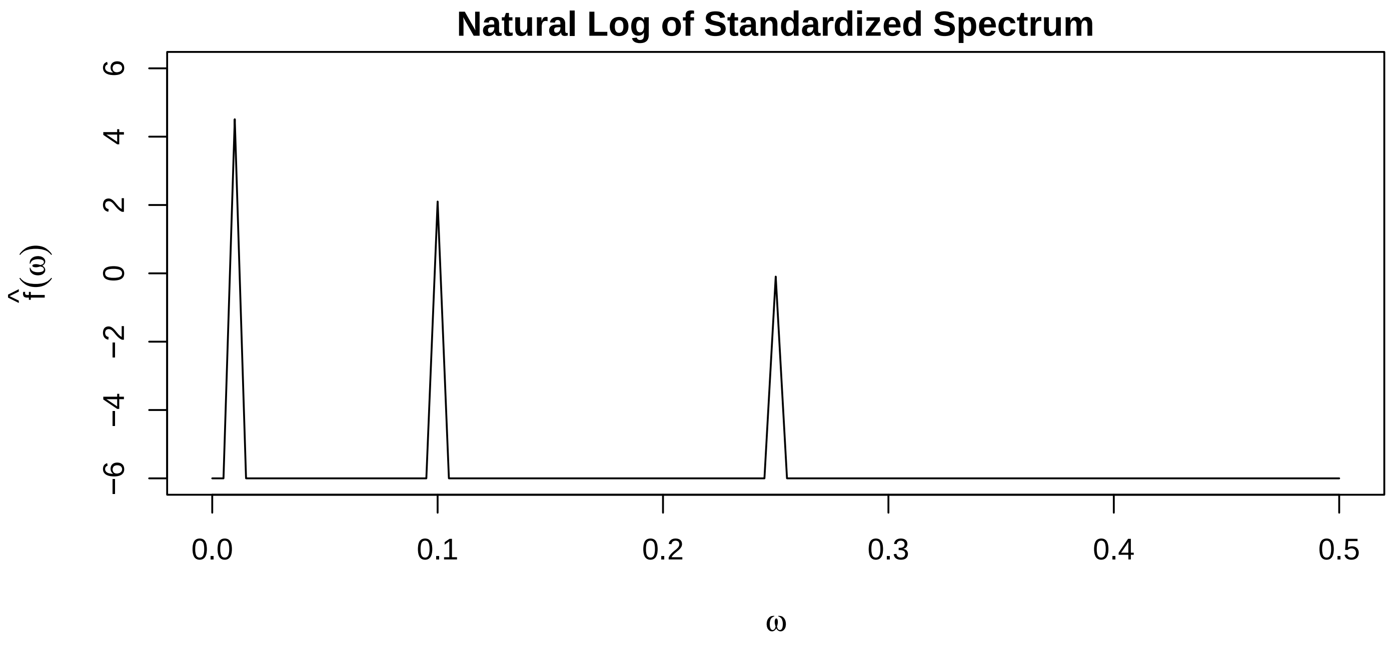
Thus despite the fact, that we added so many zeros for ffts, it computes the transformation **20 times faster** than convolution.
Since the convolution formula has time complexity O(Mn – (M^2)/2) then if M is close to n formula turns into O((n^2)/2) which is just quadratic time bound, which is very poor performance. However, when M is relatively small then Mn would be just O(n) which is linear time complexity. We can further speculate, that if we bound M to be a log_2(n) (i.e. for the n = 19991, M = 14) then convolution would actually work faster than 2FFTs. However if we are interested in big lags – 2FFTs would do the work faster. These derivations supported by the observation in the experiment: 2FFTs finishes faster than a 1/100 of a second, while convolution takes up to a half a second (the more M – the longer it takes).

Interestingly, even for the n = 199999 it takes only half a second – the same result for n=19991 (keeping M=5000 constant). This might be due to the internal R optimization technique and\or due to the multithreaded CPU.

C2.14

```
plot_three_sine <- function(n=200, noise_variance=0,
            amplitudes=c(10, 3, 1), frequencies=c(100, 10, 4)){
      t = seq(1, n)
      result = vector()
      for (i in 1:3){
            result = rbind(result, amplitudes[i] * cos(2*pi*t / frequencies[i]))
      }
      sum = colSums(result)
      if (noise_variance != 0)
            sum = sum + rnorm(n, 0, noise_variance)
      plot(t, sum, type="l")
      return (sum)
}

plot_std_periodogram <- function(ts){
      N = length(ts)
      variance = sum((ts - mean(ts))^2) / N
      zk_sq = Mod(fft(ts)^2)[1:(N/2+1)]
      Ck_sq = zk_sq / N
      # Standartize and truncate
      Ck_sq = Ck_sq/variance
      Ck_sq[Ck_sq < exp(-6) ] = exp(-6)
      Ck_sq[Ck_sq > exp(6) ] = exp(6)
      k = 1:(N/2+1)
      omega_k = (k - 1) / N
      plot(omega_k, log(Ck_sq), type="l", ylim = c(-6, 6),
            xlab = expression(omega), ylab = expression(hat(f)*(omega)),
            main = "Natural Log of Standardized Spectrum")
      return(list(frequencies=round(Mod(fft(ts))[1:(N/2+1)] * 2 / N, 6),
                        periodogram=log(Ck_sq)))
}

C2.14 <- function(amplitudes=c(10, 3, 1), frequencies=c(100, 10, 4)){
      par(mfrow=c(2,1),mar = c(4,5,1.5,1) + 0.1)
      sine_sum = plot_three_sine(amplitudes=amplitudes, frequencies=frequencies)
      periodogram = plot_std_periodogram(sine_sum)
      return(list(sine_sum=sine_sum, frequencies=periodogram$frequencies,
                      periodogram=round(periodogram$periodogram, 2)))
}
c214 = C2.14(c(10, 3, 1))

$frequencies
  [1]  0  0 10  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  3  0  0  0  0
 [26]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [51]  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
 [76]  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
[101]  0

# reconstruct sine waves
N = length(c214$sine_sum) # length of the original serie (n=200)
amps = c214$frequencies[c214$frequencies > 0] # find non zero amps
> 10 3 1
freqs = N / (-1 + which(c214$frequencies > 0)) # which natural frequencies?
> 100  10   4
plot_three_sine(amplitudes=amps, frequencies=freqs) # sine reconstructed
```





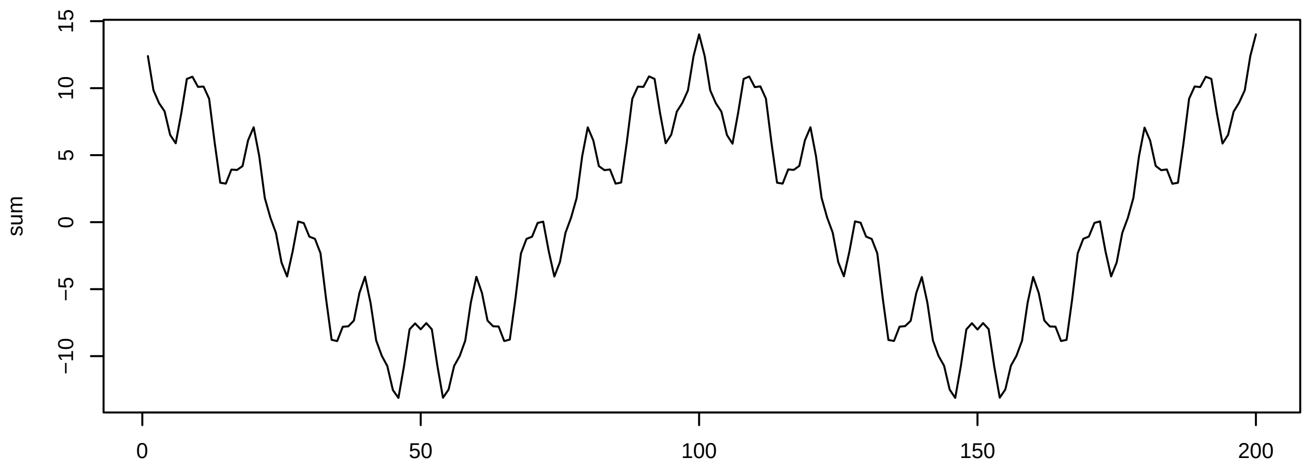
No Noise added.

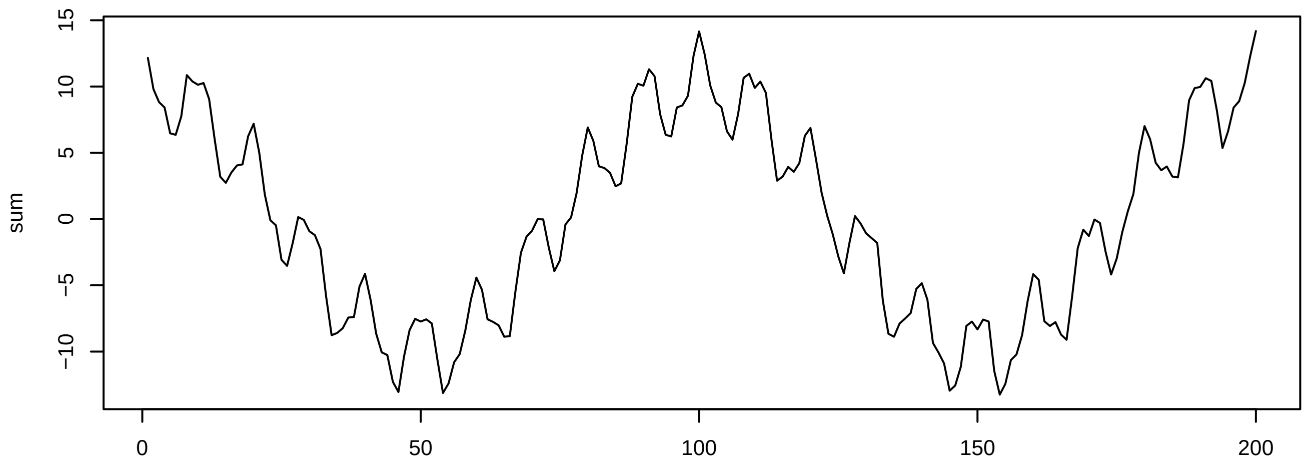
C2.17

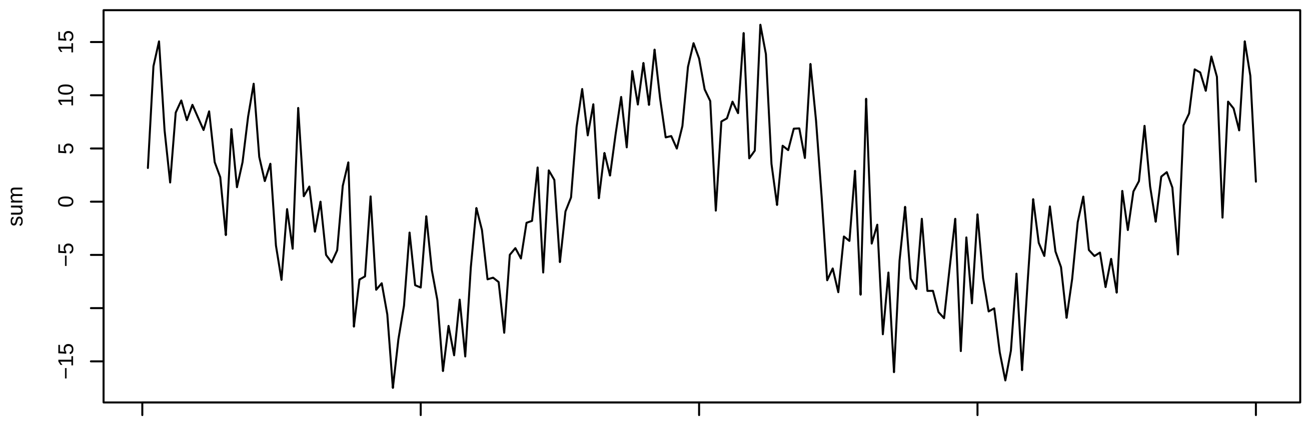
First let us plot the same sum of sines with `amplitudes=c(10, 3, 1), frequencies=c(100, 10, 4)`

However will change the amount of variance in white noise added:

```
variance = c(0.01, 0.25, 4)
for (i in 1:3)
      plot_three_sine(noise_variance=variance[i])
```







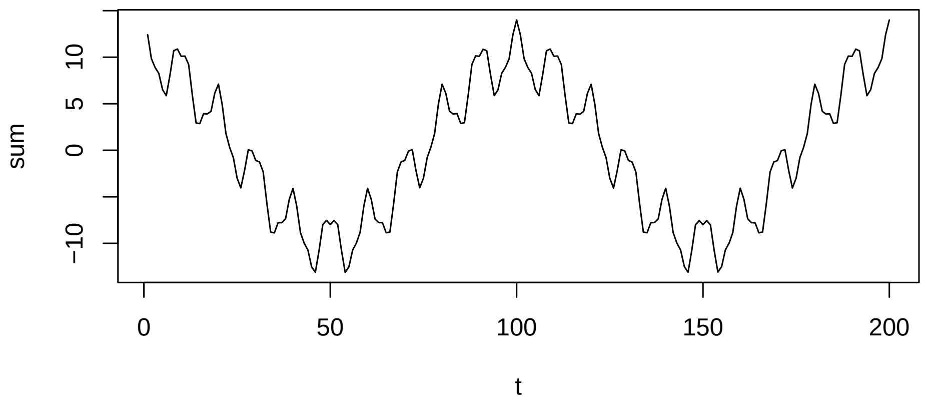
Obviously, noise affects more severe the sine with the lowest amplitude (=1).

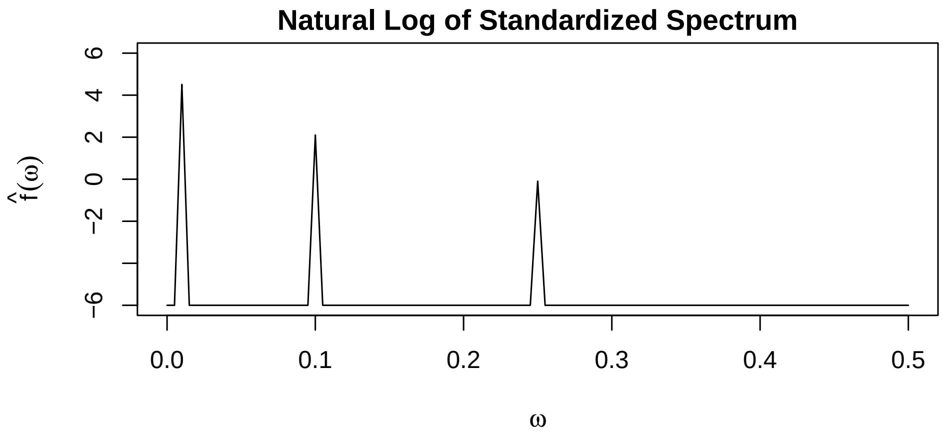
Now let us run the periodogram for these noise variances:

```
for (i in 1:3)
      C2.14(noise_variance=variance[i])
```

Noise variance = 0.01

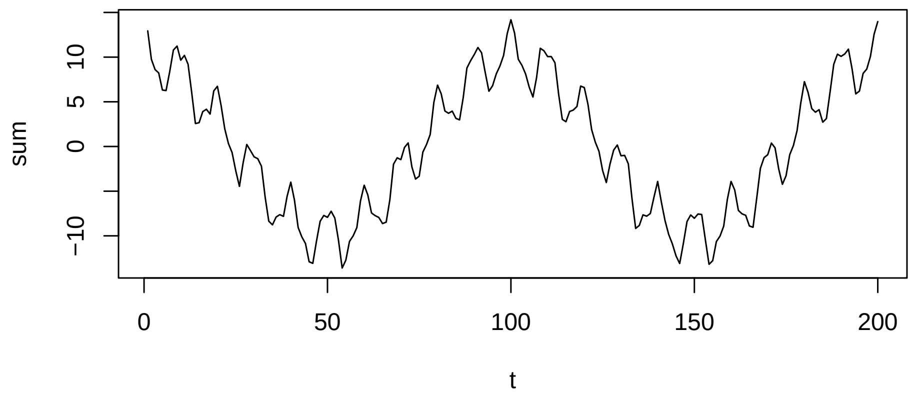
Hardly any differences are seen since noise amplitude is significantly smaller than the amplitude of the smallest sine (amp=1)

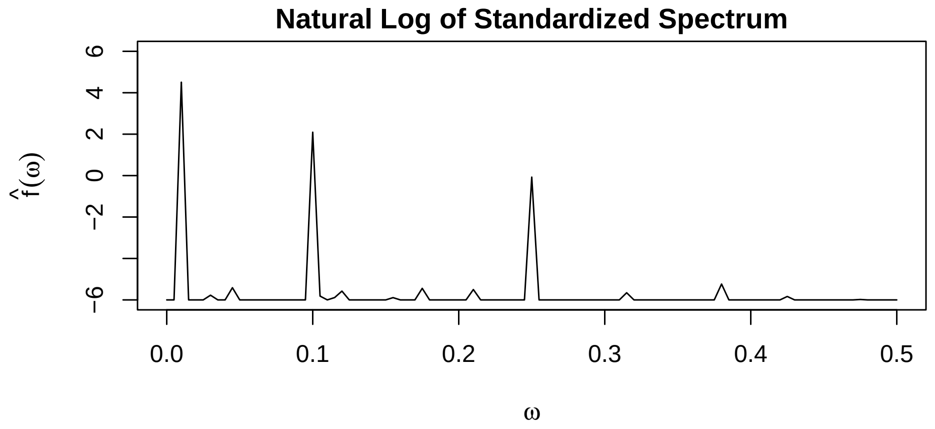




Noise variance = 0.25

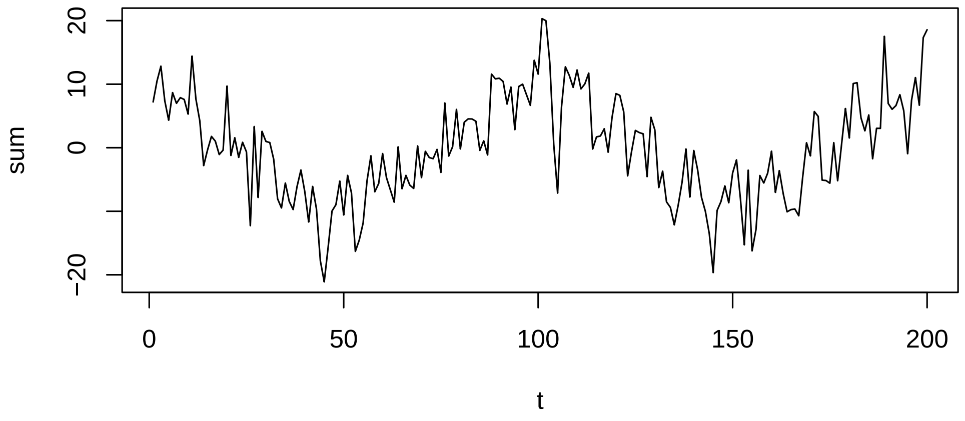
Periodogram now contains occasional spikes at the frequencies that we didn't model. That's because the noise amplitude is now comparable to the smallest sine (amp=1). However one can easily clean these pseudo spikes with filter and still get the correct resulting frequencies.

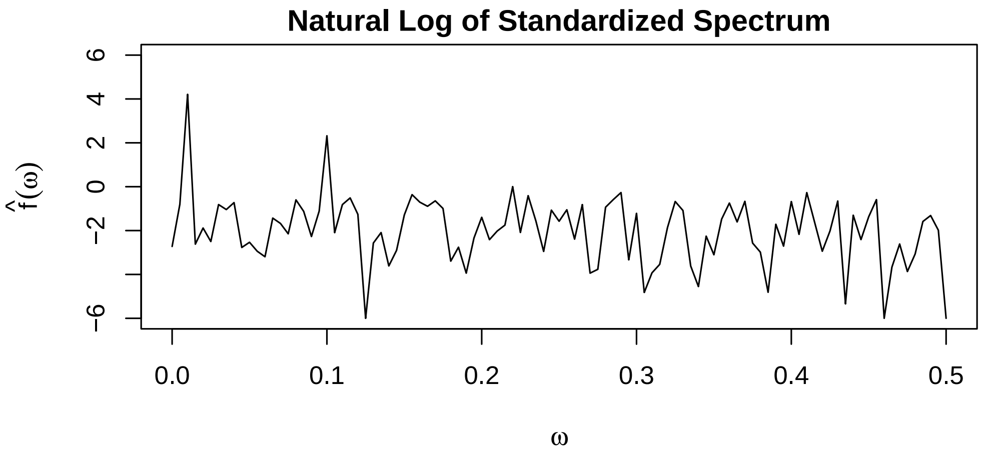




Noise variance = 4

Periodogram now contains all sorts of occasional spikes at the frequencies that we didn't model. Now it's hard to filter out what is the true process data's frequencies and what is the noise's. However, since with high amp=10 is better off – we still can see it clearly. Sine with amp=3 is distinct too. Yet sine with amp=1 is not noticeable any longer.





Thus we can conclude, that if the data has noise which has amplitude of the same degree as the data itself, then we can hardly separate the resulting frequencies of the process and of the noise.

T2.1

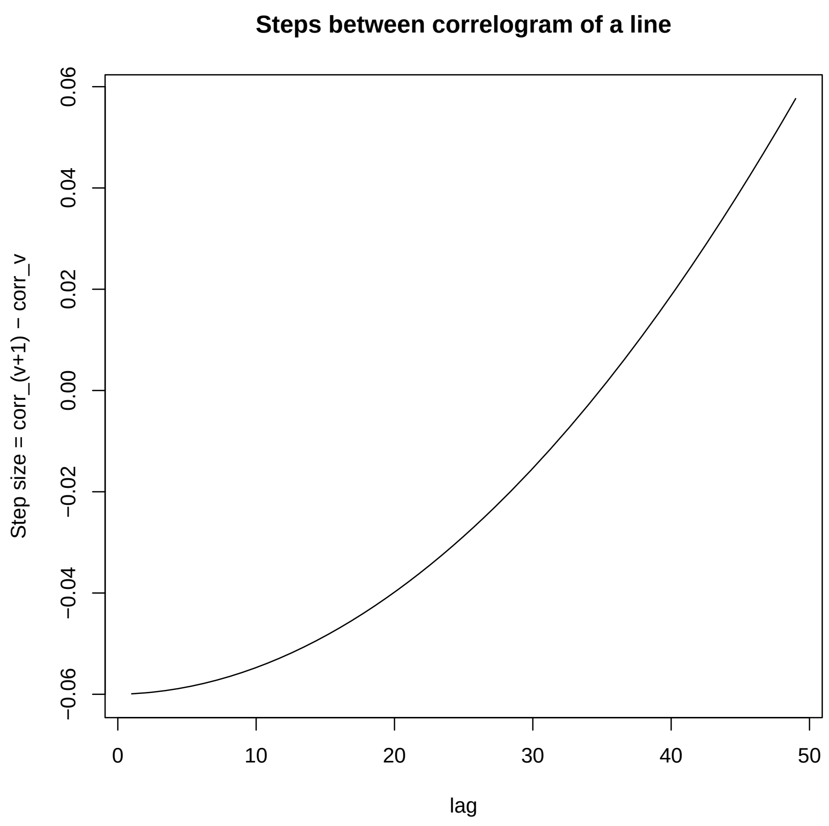
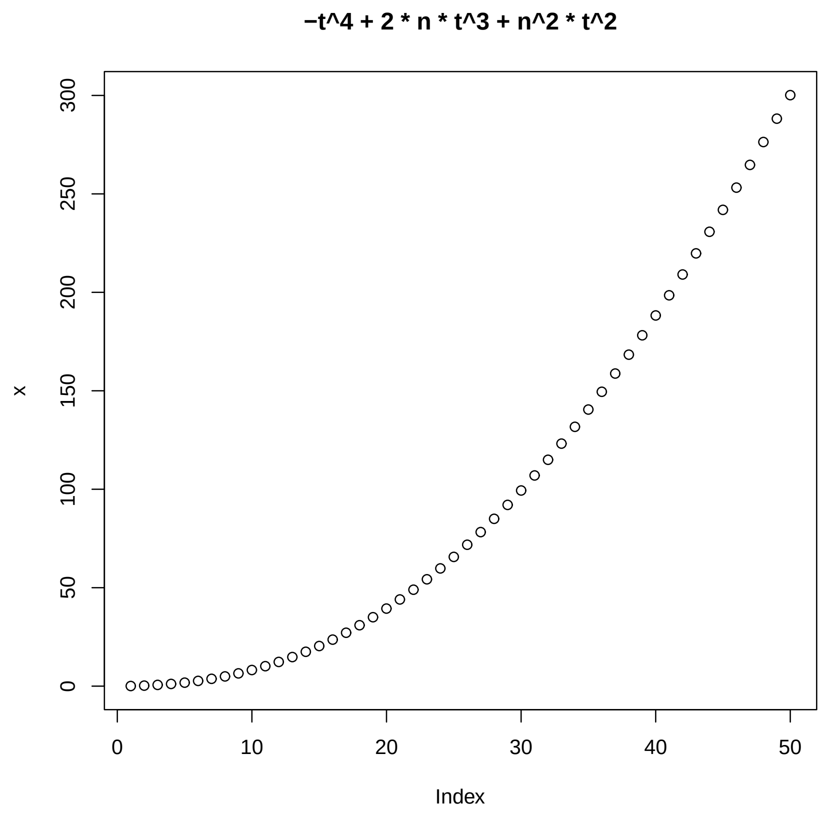


Illustration for the result of T2.1 – Correlogram of a line is **not** a line. It's in fact a polynomial of 4th degree. To actually see the non linearity we can plot the step size against the lag (above). That's exactly how the function `-t^4 + 2 * n * t^3 + n^2 * t^2` looks like (below) for the first 50 lags.


−t^4 + 2 * n * t^3 + n^2 * t^2
x
Index