Chapter 7

Creative Problems in Signal Processing, Linear Algebra and Analysis

In this chapter, I discuss beautiful off-the-beaten-path math challenges, with solution and generalizations including problems yet to be solved. When appropriate, I also show how AI can automate many of the mechanical computations, not only to save a considerable amount of time, but also to guarantee that they are error-free. Some of the material is presented as tutorial, rather than problems to solve. The links in blue point to Wikipedia references about basic topics, or mathematical forums where the topic is discussed. You won't find any of these problems in standard textbooks or in college exams. While AI significantly helps, it cannot solve them at this time, requiring outside the box thinking that current tools are not yet capable of. The mathematical level is equivalent to first year in college.

The topics covered include discrete convolution with Gaussian kernel. special integrals and series, recurrence relations, sequences, Fibonacci numbers, college-level algebra, Vandermonde and circulant matrices, elementary complex number theory, some elements of probability theory, limits, convergence, asymptotics, and special functions such as Gamma and hyperbolic functions. This list is not exhaustive. In-depth knowledge of the underlying concepts is not needed to solve these problems or understand the material.

7.1 Non-causal discrete convolution with Gaussian kernel

The discrete Gaussian convolution of the Riemann zeta function, one of the most famous in number theory, leads to some curious and unexpected results. Here I provide a solid introduction to this topic, of interest to anyone dealing with signal processing problems. Starting with the initial conditions $h_1(k) = \frac{1}{3}$ if $-1 \leq k \leq 1$ and $h_1(k) = 0$ otherwise, I iteratively define the kernel h_{n+1} as the convolution $h_{n+1} = h_1 * h_n$, that is:

$$h_{n+1}(k) = \sum_{j=-n}^{n} h_1(j) h_n(k-j). \tag{7.1}$$

Note that $h_n(k) = 0$ if $|k| > n$. For $-n \leq k \leq n$, the integers $3^n h_n(k)$ are known as trinomial coefficients [Wiki], a generalization of binomial coefficients. As $n \to \infty$, their distribution approximates a Gaussian one with zero mean and variance $\sigma_n^2 = 1/(2\pi h_n^2(0))$. This is illustrated in Figure 7.2, where you can not distinguish h_n from its Gaussian approximation. Thus h_n is called a Gaussian kernel. Also, since negative values of k are allowed, it is non-causal. Given a discrete signal X_t, I define its convolution by h_n as

$$Y_t = \sum_{k=-n}^{n} h_n(k) X_{t-k}. \tag{7.2}$$

Here, the input signal is a mixture of cosines with incommensurate periods, thus non periodic, and defined as

$$X_t = \sum_{k=r}^{m} (-1)^{k+1} \cdot \frac{1}{k^\sigma} \cos\Big(\theta(t-\tau)\log k\Big). \tag{7.3}$$

When $r = 1, m = \infty, \theta = 1$ and $\tau = 0$, X_t is the real part of the Dirichlet eta function, a close sister of the Riemann zeta function, also used as a universal function in deep neural networks in chapter 4. Here $\sigma = 0.75$.

Since the the convolution of a cosine signal with a Gaussian kernel is also a cosine of the same period, the resulting Y_t has the following form:

$$Y_t = \sum_{k=r}^{m} (-1)^{k+1} \cdot \frac{\lambda_k}{k^\sigma} \cos\Big(\varphi_k + \theta(t-\tau)\log k\Big) \tag{7.4}$$

where λ_k, φ_k can be computed exactly and indicate a change respectively in amplitude and phase (but not in period) in each cosine term after the convolution. Figure 7.1 shows scaled X_t in green, here with $r = 1$, $\theta = 0.5$, $\tau = 4.16$ and $\sigma = 0.75$. The kernel h_n has $n = 200$. The resulting output Y_t looks like a perfect cosine of period $L = 2\pi/(\theta \log 2)$. That is, all the terms in (7.4) vanish except for $k = 2$. In other words, $\lambda_k = 0$ unless $k = 2$. The red curve features Y_t when $r = 5$, with all other parameters unchanged. This corresponds to a truncated X_t that no longer contains the term $k = 2$, and this time the output Y_t is no longer a pure cosine.

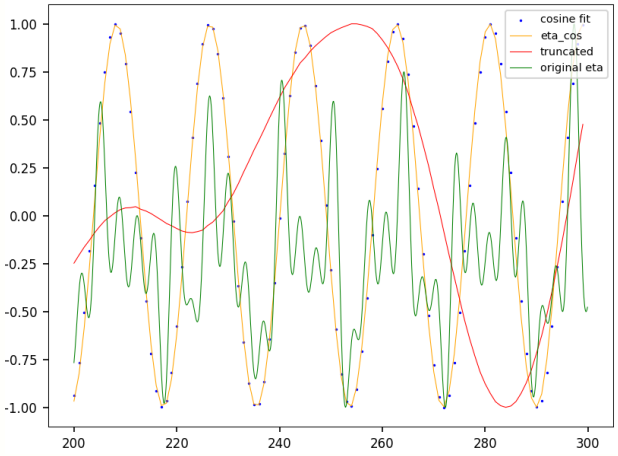

Figure 7.1: X_t in green, Y_t in orange with blue dots for cosine fit, with t on X-axis. See text for red curve

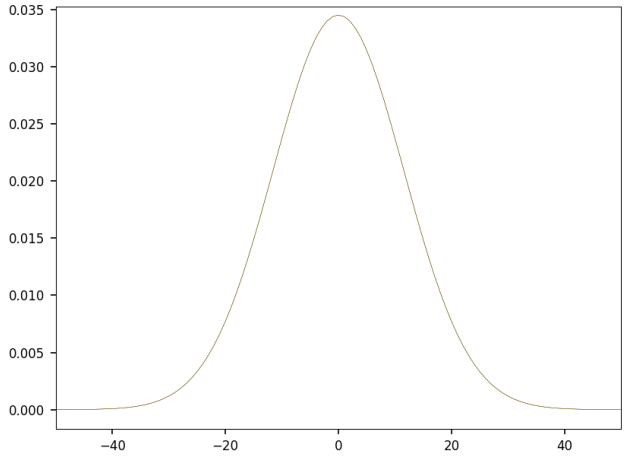

Figure 7.2: Coefficients $h_n(k)$ with k on the X-axis, and their Gaussian approximation ($n = 200$)

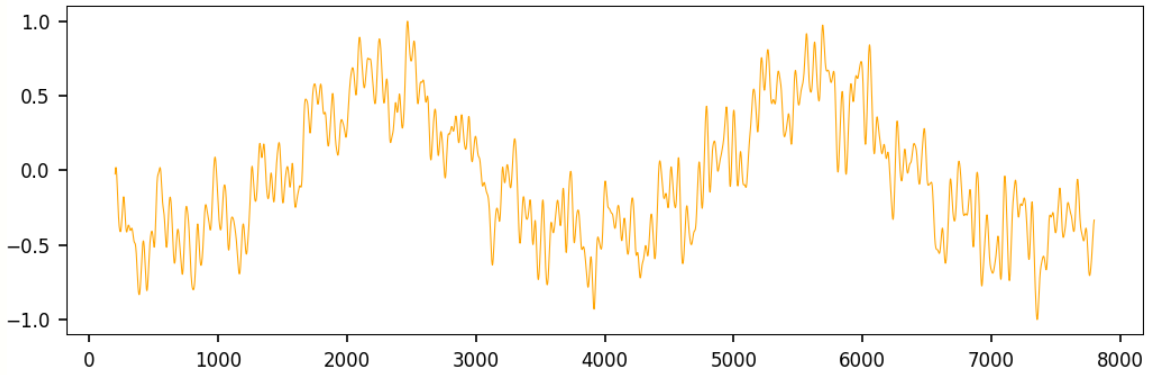

Figure 7.3: Y_t based on different X_t with $\theta = 4.9$ instead of $\theta = 0.5$ in Figure 7.1 (t on X-axis)

7.1.1 Problem

Since the convolution transforms X_t into a simple cosine in Figure 7.1, at least almost perfectly if not perfectly, it is not invertible for all practical purposes. And if it was, inverting would be a numerically unstable operation. Yet in Figure 7.3, Y_t seems more complex at first glance. Now the problem consists of addressing the following questions:

- How sensitive is the output Y_t to the parameters $m, r, \sigma, \tau, \theta$ and the kernel h_n?
- Estimate the phase and period of the cosine output in the case shown in Figure 7.1, without using curve fitting techniques. These two values correspond to φ_2 and λ_2 in (7.4), as the other λ_k for $k \neq 2$ are almost or exactly zero.
- Is Y_t periodic Figure 7.3?
- Discuss the inversion operation (deconvolution) and its feasibility.

- Even if deconvolution is not possible due to multiple X_t signals leading to the same Y_t, is the output Y_t still useful to discover interesting features of X_t? Discuss the case in Figure 7.1.

You are free to ask questions to AI to help you answer the above questions. This may help you understand the inner mechanics even if you know very little about signal processing. As a side note, the problem discussed in chapter 3 is also based on convolutions and strings auto-convolutions in particular, though my presentation of the topic has been simplified in this book to facilitate the reading. See also numerical approximation via quantization in section 8.3.2. Finally, zoom in on any figure to get a much higher resolution.

7.1.2 Solution

If the convolution is invertible, retrieving the original signal X_t from the output Y_t is done with the following formula, involving an infinite number of weights $h_n^*(k)$ even though the kernel h_n has a finite number of non-zero values $h_n(k)$:

$$X_t = \sum_{k=-\infty}^{\infty} h_n^*(k) Y_{t-k}. \tag{7.5}$$

Here h^* is the inverse of h. A search for "inverse transform to $y(t) = h(-1)x(t-1)+h(0)x(t)+h(1)x(t+1)$" on (say) Perplexity.ai will tell you the mechanics and how to proceed, see here. Additional questions such as "what are the required conditions to make discrete convolution invertible" leads to full details, see here. You may also ask for book references on the topic, asking for the PDF versions of these books, see here. One that stands out is chapter 2 in the book "Discrete–Time Signal Processing" [51]. Answers from AI are typically short compared to courses on the topic, yet detailed enough to help you dig further while starting with a high level summary. In short, to compute $h_n^*(k)$, you need to write $1/H(z)$ as a Laurent series [Wiki] (a Taylor series with positive and negative powers), and the coefficients in that series are the $h_n^*(k)$ in question:

$$H(z) = \sum_{k=-n}^{n} h_n(k) z^{-k}, \quad \frac{1}{H(z)} = \sum_{k=\infty}^{\infty} h_n^*(k) z^k \tag{7.6}$$

Formula (7.6) involves Z-transforms and the transfer function H. For the convolution to be invertible, one needs the polynomial $z^n H(z)$ to have no zero on the unit disc in the complex plane. For the inverse to be stable, all roots must be inside that disc.

I am now about to address the other questions. Before starting, note that we work with integer values of t. In some cases, Figures 7.1 and 7.3 can be misleading as X_t is not an exact value but an interpolation if t is not an integer. In particular, if Y_t is a simple cosine with a non-integer period, its interpolated values in the picture may make it appear as a more complicated function than it really is. Below is a summary of my answers:

- The convolution is not sensitive to σ, τ, m or the kernel, but it is to r and θ. To see this, run the Python code in section 7.1.3 with different parameter sets. The sharp contrast between the two orange curves Y_t in Figures 7.1 and 7.2 shows the sensitivity to θ, while $r > 2$ cannot produce the pure cosine seen with $r = 1, 2$ as it relies solely on the term $k = 2$ in (7.4). The length of the signal X_t also has some impact. In the code, it is denoted as `M`.
- In Figure 7.2, the pattern observed between $t = 4000$ and $t = 7500$ seems to repeat itself when you look at a much larger time window, suggesting periodicity. But it is not exactly periodic and may fade away when t becomes very large.
- You can fit a cosine function to Y_t in Figure 7.1 as follows. First scale Y_t so that the values are in $[-1, 1]$. Then detect the peaks. The average distance between two successive peaks is the period. Then look at the abscissa modulo the period, corresponding to the peaks. The average value is the phase. In short, you can fit Y_t to $\alpha\cos(\beta t+\gamma)$ without standard curve fitting technique to determine the optimum α, β, γ. See the functions `rescale` and `detect_period` in the code. I also fitted a Gaussian distribution to the kernel without curve fitting, see `fit_kernel_to_Gaussian` in the code.
- Despite the convoluted Y_t acting as a blurring filter, loosing some information about the original signal (not being invertible in particular), it still carries important indicators about X_t. For instance, see Figure 7.1, where minima in X_t almost match those in Y_t.

7.1.3 Python code

I use `N` to represent n, and `offset` to represent τ. Also `M` is the time window, that is the number of integer values used for X_t starting at $t = 0$. Note that to get a decent precision on the Dirichlet eta function, you need to use

$m \geq 5000$ with even a much larger m when t is large. Also, you can use the function `fit_kernel_to_Gaussian` to get a good approximation of $\sqrt{2\pi}$. The program named `zeta2.py` is on my Google drive, here.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

mpl.rcParams['axes.linewidth'] = 0.5
plt.rcParams['xtick.labelsize'] = 8
plt.rcParams['ytick.labelsize'] = 8
plt.rcParams['legend.fontsize'] = 'x-small'

# The coefficients h_n(k) are stored in the array h[].
# X(t) and Y(t) are stored respectively in the arrays x[] and y[].


def update_hash(hash, key, count):
   if key in hash:
      hash[key] += count
   else:
      hash[key] = count
   return(hash)


def build_kernel(N):

   h = {}
   h[0] = 1
   h[1] = 1
   h[2] = 1
   sum = h[0] + h[1] + h[2]

   b = {}
   for n in range(1, N):
      for k in range(0, 2*n+3):
         if k in h:
            update_hash(b, k, h[k])
            update_hash(b, k+1, h[k])
            update_hash(b, k+2, h[k])
      sum = 0
      for k in range(0, 2*n+3):
         h[k] = b[k]
         b[k] = 0
         sum += h[k]

   for k in range(0, 2*N+1):
      h[k] = h[k]/sum
   return(h)


def eta(type, t, params):

   # different offsets lead to same period in cosine fit, but different phases
   theta = params[0]
   offset = params[1]
   sigma = params[2]
   start = params[3]
   nterms = params[4]
   if 'eta' in type:
      start = 1
   fval = 0
   mod = 1
   for k in range(start, nterms):
      if type in ('eta_cos','truncated'):
         wave = np.cos(theta * (t + offset) * np.log(k))
      elif type == 'eta_sin':
         wave = np.sin(theta * (t + offset) * np.log(k))
      fval += mod * wave / k**sigma
      mod = -mod
   return(fval)


def f(type, M, params):

   x = {}
   for t in range(M):
```

```
        if type in ('truncated','eta_cos','eta_sin'):
            x[t] = eta(type, t, params)
        elif type == 'basic':
            x[t] = np.cos(0.1 * t)
    return(x)


def detect_period(yval, N):

    # cosine fit to yval[t] for t in [N, M-N[: find period and phase
    max_arg = []
    npeaks = 0
    for t in range(2,len(yval)-1):
        if yval[t] > max(yval[t-1] , yval[t+1]):
            max_arg.append(N+t)
            npeaks += 1
    if npeaks > 1:
        period = (max_arg[npeaks-1]-max_arg[0])/(npeaks-1)
        moduli = np.mod(np.array(max_arg), period)
        phase = np.mean(moduli)
    else:
        period = -1
    return(period, phase)


def rescale(yval):

    yval = np.array(yval)
    ymin = np.min(yval)
    ymax = np.max(yval)
    yval = -1 + 2*(yval - ymin)/(ymax - ymin)
    return(yval)


def fit_kernel_to_Gaussian(a):

    from scipy.stats import norm
    # a is a hash (dictionary), turn it to array a_array
    a_array = list(a.values())

    xaxis = range(-N, N+1)
    plt.plot(xaxis,a_array, linewidth=0.2,color='red')
    mu = 0
    # a_array[N] is central value out of 2N+1
    sigma = 1 / (a_array[N]*np.sqrt(2*np.pi))
    pdf_values = norm.pdf(xaxis, loc=mu, scale=sigma)
    plt.plot(xaxis, pdf_values, linewidth=0.2, color='green')
    plt.show()
    return()


#--- Main

N = 200  # length of kernel a
M = 800  # length of signal x (starts at 0)
if M < 2*N:
    print("Error: M must be > 2N")
    exit()

h = build_kernel(N)
fit_kernel_to_Gaussian(h)

params = [0.5, 4.16, 0.75, 5, 500]
ctypes = {'eta_cos':'orange','truncated':'red'}

for type in ctypes:

    x = f(type, M, params)
    y = {}
    xval = []
    yval = []
    for t in range(N, M-N):
        y[t] = 0
        for k in range(2*N+1):
            # line below does this: y[t] += h[k]*x[t-N+k]
            update_hash(y, t, h[k]*x[t-N+k])
```