Problem Set 8 - Solution

Jonasz Słomka

Unless otherwise specified, you may use MATLAB to assist with computations. Please provide a print-out of the code used and its output with your assignment.

1. More on relation between Fourier series and Discrete Fourier Transform (DFT).

   Let $f(x)$ be a periodic function of period 2π. We can express it as a Fourier series

   $$f(x) = \sum_{k=-\infty}^{\infty} c_k e^{ikx}. \tag{1}$$

   Suppose we want to approximate $f(x)$ by keeping only the first n lowest frequency amplitudes

   $$f(x) \approx \sum_{k=-n}^{n} c_k e^{ikx}. \tag{2}$$

   The number of terms on the rhs is $N = 2n + 1$. Let's sample $f(x)$ at points

   $$x_j = \frac{2\pi}{N} j \quad j = -n, \ldots, 0, \ldots n, \tag{3}$$

   that are uniformly distributed over the interval $(-\pi, \pi)$, and include the origin.

   If we denote the corresponding vector of samples by $f_j = f(x_j)$, then, at point x_j, the approximation (2) gives

   $$f_j = \sum_{k=-n}^{n} c_k \overline{\omega}^{kj}, \tag{4}$$

   where $\omega = e^{-i\frac{2\pi}{N}}$. Similarly, we can approximate the exact expression for the Fourier coefficient c_k by the finite sum

   $$c_k = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) e^{-ikx} dx \approx \frac{1}{2\pi} \sum_{j=-n}^{n} f_j \omega^{kj} \Delta x, \quad k = -n, \ldots, 0, \ldots n. \tag{5}$$

   Since $\Delta x = \frac{2\pi}{N}$, this is

   $$c_k = \frac{1}{N} \sum_{j=-n}^{n} f_j \omega^{kj}. \tag{6}$$

   We can see that Eq. (6) looks almost like the DFT and Eq. (4) is almost the inverse DFT. What is different is that the zero frequency mode is in the middle, rather than on the left. We need two steps to exactly recover the usual DFT.

First, note that f_j and c_k are labelled from $-n$ to n while in the definition of DFT we have labels from 0 to N. Let's relabel: $\tilde{f}_k = f_{k-n}$ and $\tilde{c}_k = c_{k-n}$. If we also change the summation to run from 0 to N, we get

$$\tilde{f}_j = \sum_{k=0}^{N-1} \tilde{c}_k \overline{\omega}^{(k-n)(j-n)}, \quad \tilde{c}_k = \frac{1}{N} \sum_{j=0}^{N-1} \tilde{f}_j \omega^{(k-n)(j-n)}. \tag{7}$$

We shift the above expressions by n

$$\tilde{f}_{j+n} = \sum_{k=0}^{N-1} \tilde{c}_k \overline{\omega}^{(k-n)j}, \quad \tilde{c}_{k+n} = \frac{1}{N} \sum_{j=0}^{N-1} \tilde{f}_j \omega^{k(j-n)}. \tag{8}$$

A small rearrangement gives

$$\overline{\omega}^{nj} \tilde{f}_{j+n} = \sum_{k=0}^{N-1} \tilde{c}_k \overline{\omega}^{kj}, \quad \omega^{kn} \tilde{c}_{k+n} = \frac{1}{N} \sum_{j=0}^{N-1} \tilde{f}_j \omega^{kj}. \tag{9}$$

If you remember that, for DFT and its inverse, factor like ω^{kn} in front of $\tilde{c}_{k+n}$ comes from the shift of $\tilde{f}_j$ by n, $\tilde{f}_j \to \tilde{f}_{j-n}$, then you realize that the above formulae are exactly inverse DFT and DFT, respectively, just with input and output shifted. MATLAB has built in commands to perform these shifts for you. All you need is `fftshift()` and `ifftshift()` on top of the usual `fft()` (DFT) and `ifft()` (inverse DFT). The transforms are

$$\begin{aligned} \texttt{f} &= \texttt{fftshift(ifft(ifftshift(N} * \texttt{c)))}, \\ \texttt{c} &= \texttt{fftshift(fft(ifftshift(f/N)))}. \end{aligned} \tag{10}$$

(a) In Problem set 6 you showed that the periodic function $f(x)$ defined by $f(x) = x$ on $(-\pi, \pi)$ has Fourier series given by

$$f(x) = -2 \sum_{k=1} \frac{(-1)^k}{k} \sin(kt). \tag{11}$$

Convert $f(x)$ into the complex form $f = \sum_{k=-\infty}^{\infty} c_k e^{ikx}$ (find c_k).

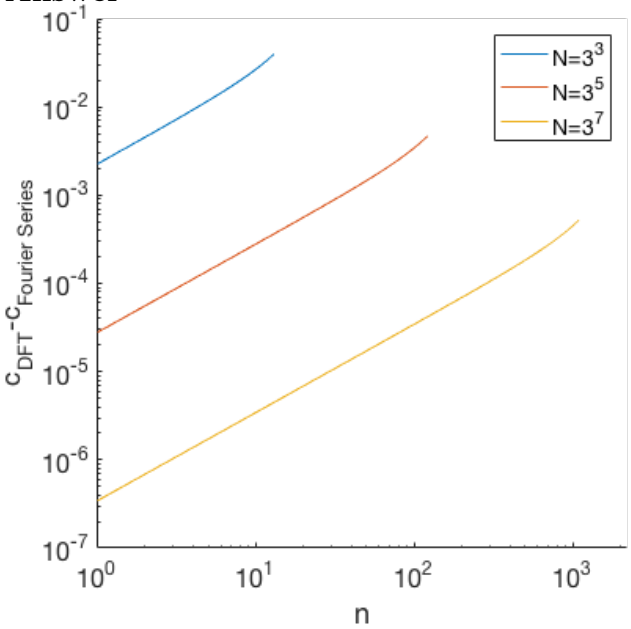
Answer: Since $\sin(kt) = \frac{e^{ikt} - e^{-ikt}}{2i}$, we get

$$\begin{aligned} c_0 &= 0, \\ c_k &= i\frac{(-1)^k}{k}, \quad k > 0, \\ c_k &= -c_{-k}, \quad k < 0. \end{aligned} \tag{12}$$

(b) Sample $f(x)$ at the grid points defined by Eq. (3). Do so for $N = 2n+1$ equal to 3^3, 3^5 and 3^7. Convert the samples to the coefficients using `c = fftshift(fft(ifftshift(f/N)))`. Plot the difference between these coefficients and the coefficients of the Fourier

series from (a). Use log-log scale. Comment on the behaviour of the difference for the three values of N.

Answer



Evaluating `c = fftshift(fft(ifftshift(f/N)))` is like approximating the integrals $c_k = \frac{1}{2\pi}\int_{-\pi}^{\pi} f(x)e^{-inx}dx$. The higher the discretization size, the better the approximation.

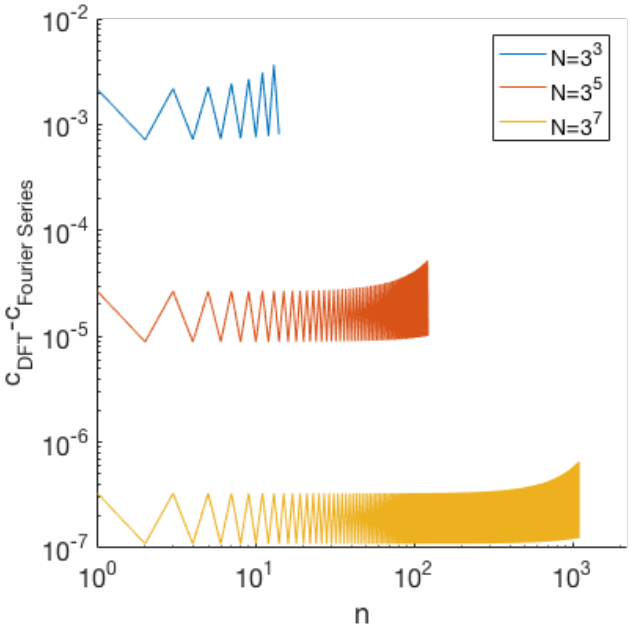
(c) In Problem set 7 you showed that the periodic function $g(x)$ defined by $g(x) = |x|$ on $(-\pi, \pi)$ has Fourier series given by

$$g(x) = \frac{\pi}{2} - \frac{4}{\pi}\sum_{n=1,3,5,\cdots}^{\infty} \frac{1}{n^2}\cos(nx). \tag{13}$$

Repeat (a) and (b) for $g(x)$.

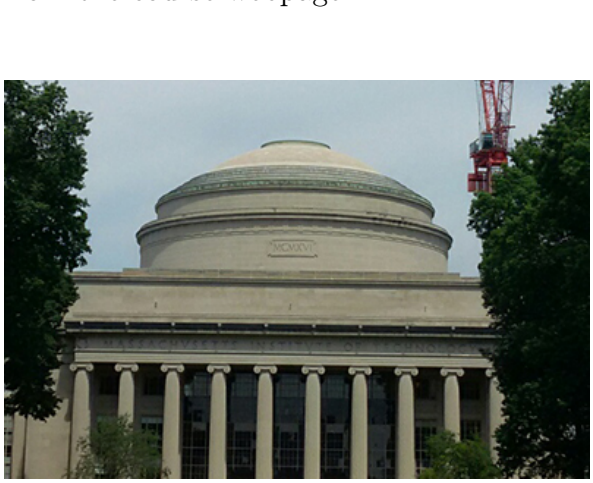
Answer: Since $\cos(kt) = \frac{e^{ikt}+e^{-ikt}}{2}$, we get

$$\begin{aligned} c_0 &= \frac{\pi}{2}, \\ c_k &= -\frac{2}{\pi}\frac{1}{k^2}, \quad k > 0,\ k \text{ odd}, \\ c_k &= c_{-k}, \quad k < 0,\ k \text{ odd}. \end{aligned} \tag{14}$$


N=3^3
N=3^5
N=3^7
c_DFT - c_Fourier Series
n
10^-2
10^-3
10^-4
10^-5
10^-6
10^-7
10^0
10^1
10^2
10^3

2. (Simplified) JPEG compression.

   Download the following two pictures of the MIT dome (`MITdome.jpg` and `MITdome2.jpg`) from the course webpage.




   You will perform image compression using the two-dimensional variants of the Discrete Cosine Transform (DCT) and the Discrete Sine Transform (DST).

   (a) In MATLAB, load the first file `MITdome.jpg`, convert it to the grayscale and display it, using

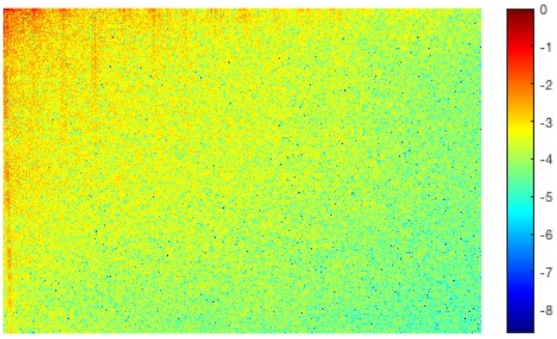
$$\begin{aligned} &\texttt{RGB} = \texttt{imread}(\texttt{'MITdome.jpg'}); \\ &\texttt{I} = \texttt{double}(\texttt{rgb2gray}(\texttt{RGB})); \\ &\texttt{figure}, \texttt{imshow}(\texttt{I}, [\texttt{0255}]) \end{aligned} \tag{15}$$

   Answer:

   (b) Apply the DCT to compute the frequency spectrum of the grayscale image. Display, on the log scale, the normalized magnitude of the coefficients. Use

$$\begin{aligned} &\texttt{J} = \texttt{dct2}(\texttt{I}); \\ &\texttt{figure}, \texttt{imshow}(\texttt{log10}(\texttt{abs}(\texttt{J})/\texttt{max}(\texttt{abs}(\texttt{J}(:)))), []), \texttt{colormap}(\texttt{jet}), \texttt{colorbar} \end{aligned} \tag{16}$$

Answer:



(c) Perform the compression `J` $\rightarrow$ `Jcompressed` by keeping only the largest amplitudes that contain 99.9% of the image energy. This should be very similar to the Speech Signal Compression Example we did in class, the difference is that now the signal is a matrix, rather than a vector. To convert a matrix into a vector by, stacking its columns on top of each other, simply put `vecJ = J(:)`. To reshape the vector `vecJ` back into a matrix, use `reshape(vecJ,m,n)`, where `m,n` are the dimensions of the matrix `[m,n] = size(J)`. The image energy is defined as a sum over the squares of the Fourier amplitudes. In MATLAB, this is `EnergyJ = norm(J(:))^2`. Calculate the percentage of nonzero amplitudes left after the compression.

Answer:

After the compression, there is 34.0469% nonzero amplitudes left.

(d) Once you compressed the image in the frequency domain, go back to the physical domain by inverting the DCT

```
Icompressed = idct2(Jcompressed);
```
(17)

and display the image

```
figure, imshow(Icompressed, [0255])
```
(18)

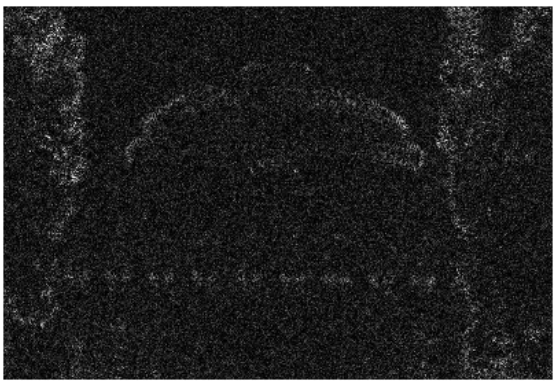
Answer:

(e) Show the (10 times) magnified difference between the original and the compressed images

$$\texttt{figure}, \texttt{imshow}(10 * \texttt{abs}(\texttt{I} - \texttt{Icompressed}), [0255]) \tag{19}$$

Where is the discrepancy the largest? In other words, which features of the original image are badly captured after the compression?

Answer:

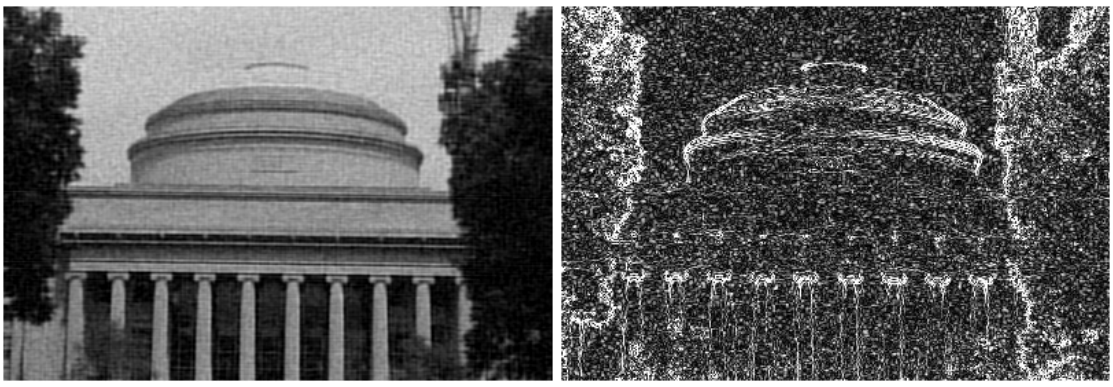


Sharp edges suffer most from the compression. This is a direct manifestation of the Gibbs phenomenon in practice.

(f) Repeat the above analysis for two other compression thresholds: keep 99% and 98% of the image energy. In each case, record the percentage of nonzero amplitudes left after the compression.

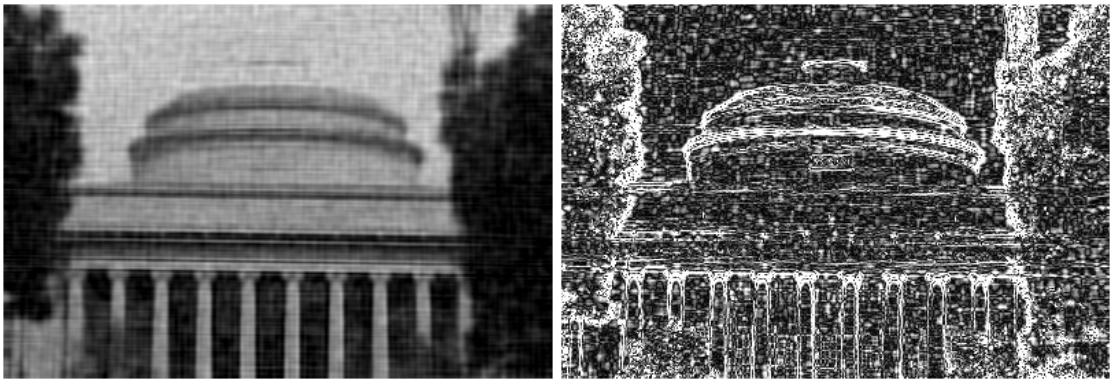
Answer:

Let's do the 99% threshold first.



I needed only 5.5781% of the amplitudes.

Now comes the 98% threshold.



I needed as little as 1.5432% of the amplitudes and the resolution is still not that bad.

(g) Repeat the analysis by using DST rather than DCT, still for the same image, `MITdome.jpg`. There is no `dst2()` command in MATLAB. You will have to use the one-dimensional version `dst()` twice, first on columns, then on rows. This is done by

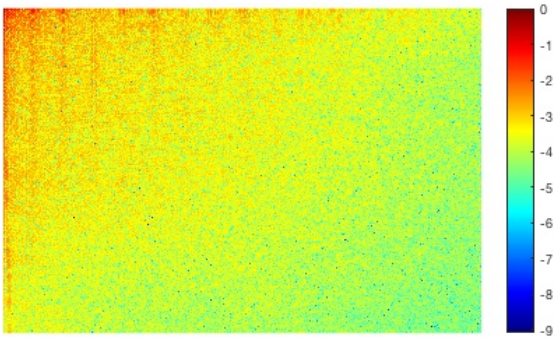
$$\texttt{J2} = \texttt{dst}(\texttt{dst}(\texttt{I}).').'; \tag{20}$$

Similarly, to apply the (2D) inverse of DST use

$$\texttt{J2} = \texttt{idst}(\texttt{idst}(\texttt{I}).').'; \tag{21}$$

Again, for the three thresholds, record the percentage of nonzero amplitudes left after the compression. How does it compare with the DCT case?

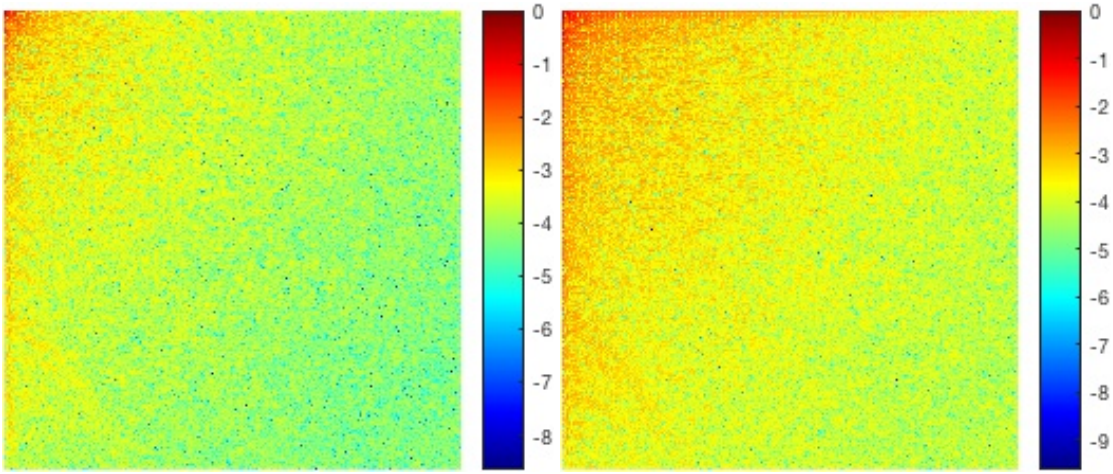
Answer:



Using DST I got for the three thresholds: 34.2086%, 5.7904% and 1.7463% nonzero amplitudes. This is only marginally worse than using DCT.

(h) Now repeat the analysis (for both DCT and DST) with the other image, `MITdome2.jpg`. Which of the two (DCT or DST) gives a more efficient compression? Can you explain why?

Answer: For `MITdome2.jpg`, the amplitudes look like (DCT left, DST right):

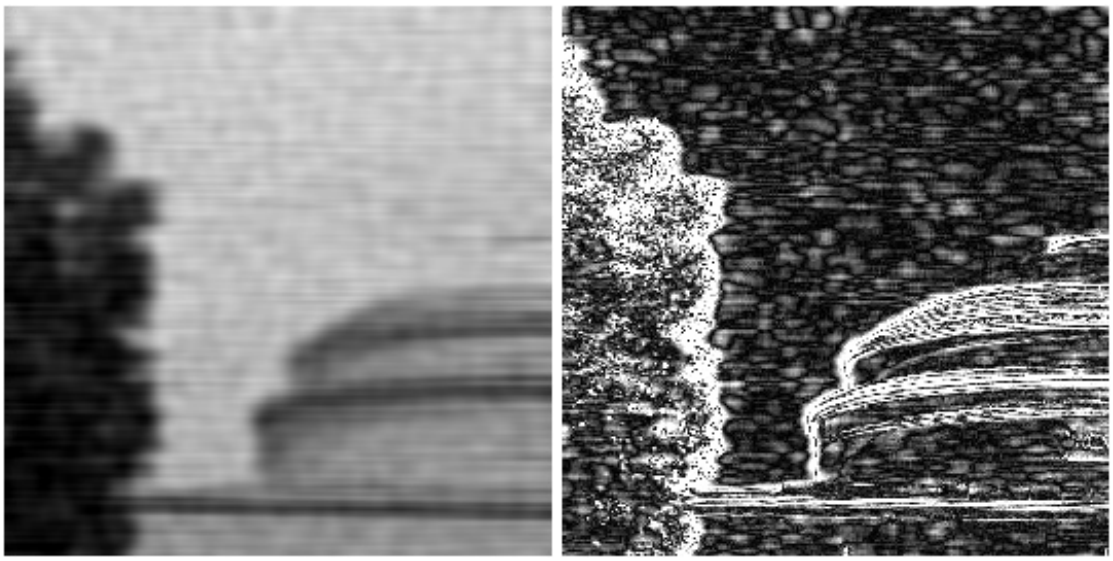


	DCT	DST
99.9%	22.3860%	23.2773%
99%	0.7434%	1.6629%
98%	0.0929%	0.4534%

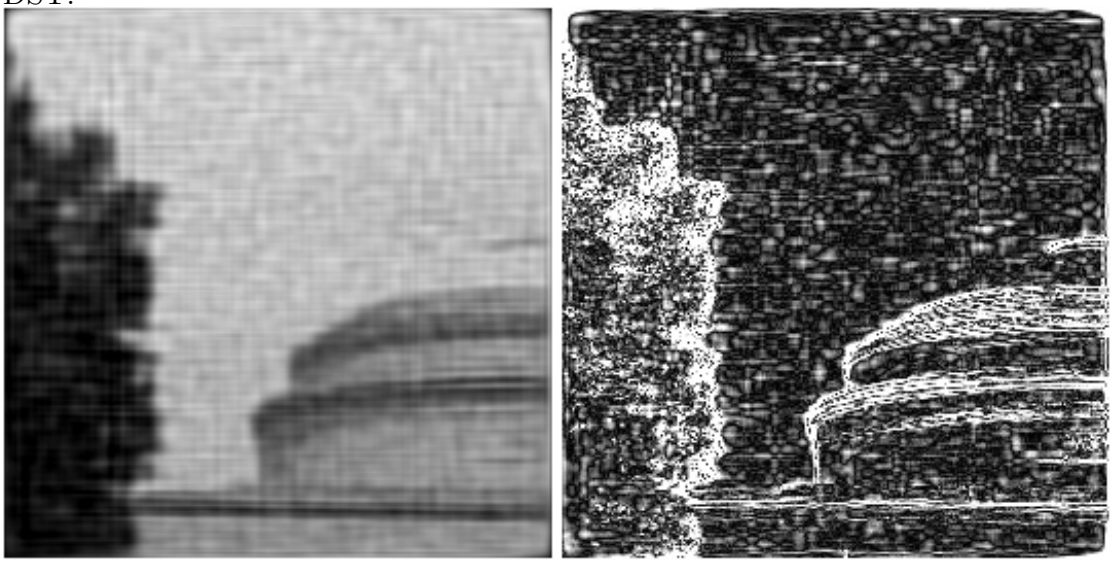
For the high resolution (99.9%) there is still little difference. As we decrease the desired resolution, we see that DCT beats DST by a factor of two in the 99% case, and by a factor of five in the 98% case. This is because DST is wasting energy on resolving the boundaries. Odd flipping inherent in the definition of DST creates sharp jumps if the four edges of the picture are significantly different (from zero). This is the case for `MITdome2.jpg` and not so much for `MITdome.jpg` where the two trees around the dome make the picture almost periodic in the horizontal direction.

For the 99% case let's compare the DCT and DST.

DCT:



DST:



It is now clear that DST does badly not only at the contours of the tree and the dome (same as DCT), but also around the four edges of the picture itself.

3. Fast multiplication through convolution.

   (a) Given two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ with components a_i and b_i, respectively, the discrete convolution of $\boldsymbol{a}$ and $\boldsymbol{b}$ is a third vector $\boldsymbol{a} * \boldsymbol{b}$ with components

$$(\boldsymbol{a} * \boldsymbol{b})_n = \sum_{i+j=n} a_i b_j. \tag{22}$$

   The sum on the rhs goes over all pairs of indices i and j such that $i + j = n$. If $\boldsymbol{a}$ has n_a components, and $\boldsymbol{b}$ has n_b components, show that $\boldsymbol{a} * \boldsymbol{b}$ has $n_a + n_b - 1$ components.

   Answer: Think about sliding the vector $\boldsymbol{b}$ against $\boldsymbol{a}$ and the resulting overlap. Sliding the right end of (flipped) $\boldsymbol{b}$ against $\boldsymbol{a}$ gives n_a overlaps. Sliding the left end

against $\boldsymbol{a}$ takes the length of $\boldsymbol{b}$ minus one step (after n_b steps $\boldsymbol{a}$ and $\boldsymbol{b}$ are already stacked back-to-back so no overlap), so in total we have $n_a + n_b - 1$ overlaps.

(b) Consider two polynomials

$$p(x) = a_0 + a_1 x + a_2 x^2, \quad q(x) = b_0 + b_1 x + b_2 x^2 + b_3 x^3. \tag{23}$$

Find the polynomial $r(x) = p(x)q(x)$. Show that the coefficients c_n of $r(x)$ are the discrete convolution of the coefficients a_i and b_i.

Answer: Multiply out the two polynomials and collect terms of the same power

$$\begin{aligned} p(x)q(x) &= (a_0 + a_1 x + a_2 x^2)(b_0 + b_1 x + b_2 x^2 + b_3 x^3) \\ &= a_0 b_0 + (a_0 b_1 + a_1 b_0)x + (a_0 b_2 + a_1 b_1 + a_2 b_0)x^2 \\ &\quad + (a_0 b_3 + a_1 b_2 + a_2 b_1)x^3 + (a_1 b_3 + a_2 b_2)x^4 + a_2 b_3 x^5. \end{aligned} \tag{24}$$

On the other hand, let's convolve $\boldsymbol{a} = [a_0, a_1, a_2]$ with $\boldsymbol{b} = [b_0, b_1, b_2, b_3]$. From (a), we know that $\boldsymbol{a} * \boldsymbol{b}$ has 6 entries. From the definition of convolution, we get

$$\begin{aligned} (\boldsymbol{a} * \boldsymbol{b})_0 &= \sum_{i+j=0} a_i b_j = a_0 b_0, \\ (\boldsymbol{a} * \boldsymbol{b})_1 &= \sum_{i+j=1} a_i b_j = a_0 b_1 + a_1 b_0, \\ (\boldsymbol{a} * \boldsymbol{b})_2 &= \sum_{i+j=2} a_i b_j = a_0 b_2 + a_1 b_1 + a_2 b_0, \\ (\boldsymbol{a} * \boldsymbol{b})_3 &= \sum_{i+j=3} a_i b_j = a_0 b_3 + a_1 b_2 + a_2 b_1, \\ (\boldsymbol{a} * \boldsymbol{b})_4 &= \sum_{i+j=4} a_i b_j = a_1 b_3 + a_2 b_2, \\ (\boldsymbol{a} * \boldsymbol{b})_5 &= \sum_{i+j=5} a_i b_j = a_2 b_3. \end{aligned} \tag{25}$$

(c) From part (b) it should be intuitively clear that, in the general case

$$p(x) = \sum_{i=0}^{P} a_i x^i, \quad q(x) = \sum_{j=0}^{Q} b_j x^i, \tag{26}$$

the coefficients of the polynomial $r(x) = p(x)q(x)$ are given by $\boldsymbol{a} * \boldsymbol{b}$. In other words, multiplying two polynomials means convolving their coefficients. We have seen in class that `fft()` can be used to cyclically convolve two vectors. Two avoid the cyclic property, we must add enough zeros at the end of $\boldsymbol{a}$ and $\boldsymbol{b}$ (zero-padding) so that their length is at least $(P+1) + (Q+1) - 1 = P + Q + 1$. Denote the extended vectors by $\tilde{\boldsymbol{a}}$ and $\tilde{\boldsymbol{b}}$. Then the the following single line in MATLAB produces $\boldsymbol{a} * \boldsymbol{b}$

$$\texttt{ifft(fft(ã).*fft(b̃))}. \tag{27}$$

Now suppose you want to multiply two really big numbers, one with 100 and the other with 400 digits. If you want to use Long Multiplication (as in elementary school), you would need about $100 \times 400 = 40000$ operations. Explain how you can use the above polynomial multiplication-convolution relation and `fft`() to reduce this number. With `fft`(), how many operations do you roughly need?

Answer: Any number with a finite number of digits in the decimal representation can be thought of as an evaluation of the polynomial $p(x) = a_0 + a_1 x + a_2 x^2 + \ldots$, where a_i are the digits of the representation at $x = 10$. For example, to the number 5031, we associate the polynomial

$$p(x) = 1 + 3x + 5x^3. \tag{28}$$

Then, of course, $5031 = p(10)$. Thus, to multiply two large numbers, with the number of digits, say n_a and n_b, we can form the two corresponding polynomials (of order $n_a - 1$ and $n_b - 1$, respectively). Instead of multiplying the polynomials directly, we can use the zero padding technique and the formula (27). In total, after zero padding, we have three FFTs (of size $N = n_a + n_b - 1$) and one multiplication (of size N). The number of operations is roughly (FFT has complexity $n \log n$)

$$3N \log N + N \approx 3N \log N. \tag{29}$$

This is much slower growth than $n_a n_b$ required when doing the Long Multiplication. For $n_a = 400$ and $n_b = 100$, the number of operations is around 9000.