

ECSE 343: Numerical Methods in Engineering

Assignment 4: Discrete Fourier Transform

Due: Thursday, ~~March 31st~~ April 7th, 2022 at 11:59pm EST on [myCourses](#)
Final weight: **20%**


We suggest you take time to read the entire handout *before* starting your work.

Contents

- 1 Assignment Policies
 - 1.1 Late policy and plagiarism
 - 1.2 Library imports and writing tests
- 2 Discrete Fourier Transform
- 3 2D Discrete Fourier Transform
 - 3.1 Leveraging Factorization
- 4 Fourier for Image Convolution & Deconvolution
 - 4.1 2D Fourier Convolution
 - 4.2 2D Fourier Deconvolution
 - 4.2.1 Laplacian Regularization
- 5 You're Done!

1 Assignment Policies

Download and modify the standalone Python script, renaming the file according to your student ID as **YourStudentID.py** → e.g., if your id is **234567890**, your submitted filename should be **234567890.py**

 Every time you submit a new file, your previous submission will be overwritten. We will only grade the **final submitted file**, so feel free to submit often as you progress through the assignment.

1.1 Late policy and plagiarism

All the assignments are to be completed individually, unless stated otherwise. You are expected to respect the [late policy](#) and [collaboration/plagiarism](#) policies. If you have questions or doubts regarding these policies,

please **contact the TA** or speak to the Professor *in class*.

1.2 Library imports and writing tests

The code we provide includes a superset of all the imports you are allowed to use when implementing your solution.



You must not use any additional imports for your solution, other than the ones provided by us.

Do not edit the imports at the top of the .py file. Doing so will result in an assignment grade of **0%**.

One exception to this policy is that, in the `__main__` block, you are allowed to import any additional routines you would like in order to test your solution code. Do **not** use `import` in any of your solution routines **nor** at the top of the Python file; only optionally in `__main__`.



It is *very difficult* to successfully complete this assignment without writing your own tests in `__main__`. We strongly recommend against relying exclusively on the example tests we provide. Think about what kind of input/output behaviours you can write and test against.

2 Discrete Fourier Transform

Given an input continuous function f sampled at N equally-spaced discrete points on an integer lattice $x \in \{0, \dots, N-1\}$ in the primal domain (e.g., time, space, etc.), we denote this discretized signal as $f(x) \equiv f[x]$. Note that, in general, f can be a *complex-valued* function (and, so too the sampled values $f[\cdot]$); however, our examples will only consider real-valued input functions $f : \mathbb{R} \rightarrow \mathbb{R}$.

The **discrete Fourier transform** (DFT) $\hat{f}[\omega]$ of the sampled signal is then given by:

$$\hat{f}[\omega] = \sum_{k=0}^{N-1} f[k] \exp\left(\frac{-2\pi i k \omega}{N}\right), \quad (1)$$

which can be equivalently expressed as a matrix-vector operation (see lecture slides) as

$$\hat{\mathbf{f}} = \mathbf{M}_N \mathbf{f},$$

where $\mathbf{f} = (f[0], f[1], \dots, f[N-1])$ is a vector of all the sampled function values, \mathbf{M}_N encodes the linear operator form of the DFT for a length- N input, and $\hat{\mathbf{f}} = (\hat{f}[0], \hat{f}[1], \dots, \hat{f}[N-1])$ is a vector of the sampled DFT of f . The (discrete) signal $\hat{f}[x]$ can be interpreted as a discrete sampling of the continuous Fourier transform $\mathcal{F}\{f_N(x)\}$ of an N -periodic summation of $f(x)$ — i.e., $f(x)$ repeated periodically.

Note that, regardless of whether the input signal is real- or complex-valued, the DFT $\hat{f}[x]$ **always** has complex-valued elements: $\text{Re}(\hat{f})$ encodes the frequency amplitudes and $\text{Im}(\hat{f})$ their phases.

The **inverse discrete Fourier transform** (iDFT) allows us to transform back from this (discrete, complex-valued) sampled frequency spectrum to the original (possibly only real-valued) sampled signal in the primal domain, as

$$f[x] = \frac{1}{N} \sum_{k=0}^{N-1} \hat{f}[k] \exp\left(\frac{2\pi i k x}{N}\right). \quad (2)$$

Note the similarities in the mathematical definitions of the DFT and iDFT in Equations 1 and 2, comprising of a sign change in the complex exponential and different normalization constants. You will leverage these similarities to avoid code duplication: your DFT and iDFT implementations will rely on an implementation of a *generalized* (i.e., bidirectional) DFT.

Specifically, you will implement this generalized DFT formula for a sampled input $\Upsilon[\cdot]$,

$$\Psi[a; s] = \sum_{b=0}^{N-1} \Upsilon[b] \exp\left(s \frac{2\pi i b a}{N}\right),$$

where we obtain the DFT by setting $\Upsilon = f$ and $s = -1$ (and $a \equiv \omega$ and $b \equiv x$), and the iDFT with $\Upsilon = \frac{1}{N} \hat{f}$ and $s = 1$ (and $a \equiv x$ and $b \equiv \omega$). Your implementations will act on, and return, an **entire vector** of sampled primal- (i.e., \mathbf{f}) or frequency-domain (i.e., $\hat{\mathbf{f}}$) values.



Deliverable 1 [17.5%]

Implement the Generalized DFT Routine: Complete the implementation of the generalized DFT function Ψ as specified by Ψ and with the default parameter setting of $s = -1$.

As discussed above, the generalized DFT equation Υ can be used to realize both a forward and inverse DFT.



Deliverable 2 [2.5%]

Implement the inverse DFT Routine: Complete the implementation of the inverse DFT function iDFT that relies on calling your generalized DFT routine, above.

A careful numpy implementation should not rely on **any** for loops or related Python operations (e.g., loop comprehensions). You may also wish to leverage the fact that $\exp(\iota x) = \cos x + \iota \sin x$.

3 2D Discrete Fourier Transform

Let's now consider a generalization to 2D, where we sample a continuous function $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ on the (square) 2D integer lattice leading to a discretized 2D signal $f[x, y]$ at $(x, y) \in \{0, \dots, N-1\}^2$.

We can similarly define the sampled 2D DFT $\hat{f}[\omega_x, \omega_y]$ of $f[x, y]$ as

$$\hat{f}[\omega_x, \omega_y] = \sum_{k_x=0}^{N-1} \sum_{k_y=0}^{N-1} f[k_x, k_y] \exp\left(\frac{-2\pi\iota(k_x\omega_x + k_y\omega_y)}{N^2}\right), \quad (3)$$

which admits a similarly compact matrix-vector expression as

$$\widehat{\mathbf{F}} = \mathbf{M}_{N,N} \mathbf{F},$$

where we can arrange the sampled primal function values in a matrix \mathbf{F} with elements $(\mathbf{F})_{i,j} = f[i, j]$, with $\mathbf{M}_{N,N}$ encoding the linear operator form of the 2D DFT for a square length- $N \times N$ sampled input, and $\widehat{\mathbf{F}}$ is the matrix of the sampled 2D DFT coefficients with $(\widehat{\mathbf{F}})_{i,j} = \hat{f}[i, j]$.

Again, similarly to the 1D setting, DFT elements $\hat{f}[\cdot, \cdot]$ are always complex-valued with $\text{Re}(\hat{f})$ encoding frequency amplitudes and $\text{Im}(\hat{f})$ their phases.

The 2D inverse DFT is now

$$f[x, y] = \frac{1}{N^2} \sum_{k_x=0}^{N-1} \sum_{k_y=0}^{N-1} \hat{f}[k_x, k_y] \exp\left(\frac{2\pi\iota(k_x x + k_y y)}{N^2}\right) \quad (4)$$

and a similar extension of the *generalized DFT* routine applies in the 2D setting.

3.1 Leveraging Factorization

After little manipulation, one can observe that the 2D DFT and iDFT in Equations 3 and 4 can be *factorized* into several applications of their 1D counterparts.

By applying a 1D DFT independently to each of the *rows* of \mathbf{F} , before applying the 1D DFT independently to the *columns* of the (partially-)transformed \mathbf{F} , we arrive at the 2D DFT transform of \mathbf{F} . In matrix form, this amounts to

$$\widehat{\mathbf{F}} = \mathbf{M}_{N,N} \mathbf{F} = \mathbf{M}_N (\mathbf{M}_N \mathbf{F}^T)^T, \quad (5)$$

You should leverage the factorization to accelerate your implementation.



Deliverable 3 [22.5%]

Implement a Generalized 2D DFT Routine: Complete the implementation of the generalized 2D DFT function `DFT2D`, which extends the 1D generalized routine to 2D.

Keep in mind that you **don't** have to explicitly form these matrices at any point in order to take advantage of the aforementioned factorization property.



Deliverable 4 [2.5%]

Implement the inverse 2D DFT Routine: Complete the implementation of the inverse 2D DFT function `iDFT2D` that relies on calling your generalized `DFT2D` routine.

4 Fourier for Image Convolution & Deconvolution

In Assignments 1 and 3, we explored convolution and deconvolution using linear operators that act on sampled signals *in the primal (i.e., pixel, spatial) domain*.

We constructed the convolution operators as a function of the underlying convolution kernel, and we implemented a *wrap*-based boundary condition in instances during convolution where kernels only partially overlapped the input domain.

Manually treating the indexing, which additionally required "*flattening*" 2D images into 1D vectors, required care.

Frequency-space representations are particularly well-suited to certain applications, among them convolution. We will use the DFT to perform image convolution and (regularized) deconvolution, leveraging these advantages:

1. convolution in the primal domain amounts to multiplication in the frequency domain, and
2. in the frequency domain, we can perform computation *directly* with native 2D signals (e.g., the image and kernel), instead of having to "flatten" them for the sake of formulating the forward (and backward) problems as systems of linear equations.

4.1 2D Fourier Convolution

The (forward) convolution of a 2D function $I(x, y)$ with a 2D kernel $k(x, y)$ yields a 2D output function $B(x, y) = I(x, y) \circledast k(x, y)$. In the image convolution settings we have explored, $I(x, y) \equiv I[x, y]$ is a discretely-sampled input image, $k(x, y) \equiv k[x, y]$ is a (typically square, odd edge-length) discrete kernel, and $B(x, y) \equiv B[x, y]$ is the blurred output image.

Given a matrix \mathbf{I} of the 2D sampled image pixel values, i.e., with $(\mathbf{I})_{i,j} = I(i, j)$, we obtain its DFT $\widehat{\mathbf{I}}$ as per Equation 5. For the kernel, we can also form a matrix \mathbf{K} of its sampled values before computing its DFT $\widehat{\mathbf{K}}$; here, however, we need take into account two additional points:

1. the kernel matrix \mathbf{K} needs to be 0-padded to match the size of the image matrix \mathbf{I} — this will be needed to allow for an element-wise product of their DFTs, later on; and,
2. the “location”/placement of the non-zero kernel elements in the matrix \mathbf{K} needs to be chosen carefully so to respect the same “wrap” boundary condition behaviour that we maintained in the primal domain¹.



Deliverable 5 [25%]

Fourier Kernel Matrix: Complete the implementation of the `FourierKernelMatrix` that takes as input an $n \times n$ input blur kernel (with odd n), a 2-tuple of the shape (i.e., dimensions) of the input/output image (`image_shape`), and (optionally) a forward 2D DFT function (`DFT2D_func`) that you will call in your `FourierKernelMatrix` solution. The function returns the $\widehat{\mathbf{K}}$ matrix corresponding to the DFT of \mathbf{K} . The non-zero kernel elements in \mathbf{K} should be placed in the matrix so to admit the same post-convolution wrap-based blurring behaviour as we saw in Assignments 1 and 3.



When implementing and debugging `FourierKernelMatrix`, you don't necessarily have to rely on your `DFT2D` routine (e.g., by passing something other than `DFT2D` as the `DFT2D_func` parameter).

¹ The shifting property of convolutions can give you a hint of exactly how to structure and place the non-zero kernel matrix elements in \mathbf{K} , before taking its DFT.

Given the appropriately constructed \mathbf{K} , and its DFT $\widehat{\mathbf{K}}$, we can obtain the DFT $\widehat{\mathbf{B}}$ of the sampled output image \mathbf{B} (i.e., with $(\mathbf{B})_{i,j} = B(i, j)$) as

$$\widehat{\mathbf{B}} = \widehat{\mathbf{I}} \star \widehat{\mathbf{K}},$$

where \star is an *element-wise* product. Taking the real component of the inverse DFT of $\widehat{\mathbf{B}}$, we arrive at the final output blurred image matrix \mathbf{B} .

4.2 2D Fourier Deconvolution

In an unregularized setting, we can readily express the deconvolution problem using the DFT as

$$\widehat{\mathbf{I}} = \widehat{\mathbf{B}} \star^{-1} \widehat{\mathbf{K}}, \quad (6)$$

where \star^{-1} is an *element-wise* division. Taking the real component of the inverse DFT of $\widehat{\mathbf{I}}$ yields the unblurred image matrix \mathbf{I} . Since the conditioning of the DFT and iDFT are 1, the underlying conditioning of the convolution/deconvolution system remain as (un)stable as they were in the primal domain.

Unlike Assignments 1 and 3, we will not be providing you with the ground truth deblurred image for your reference, below. You can code up example tests for forward convolution and non-noisy/unregularized deblurring to sanity check your DFT/iDFT implementations; your code from A3 can come in handy, [here!](#)

4.2.1 Laplacian Regularization

We provide you with a blurred image polluted by mild noise, as well as an $n \times n$ blur kernel (which you will need to extend appropriately into \mathbf{K} , as above).

As mentioned briefly in Assignment 3, *Laplacian regularization* is well-suited to image deblurring problems: it adds a regularization term that penalizes large second derivatives in image-space, a statistical property of many natural images.

In the primal domain, the 3×3 *Laplacian regularization kernel* is

$$l(x, y) \equiv l[x, y] = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad (7)$$

which you can extend and arrange into a matrix \mathbf{L} with dimensions that match the image resolution and with structure that respects the *wrapping* boundary condition treatment (i.e., similar to the blur kernel's matrix \mathbf{K} structure).

We can solve the regularized least squares deconvolution problem, expressed in the primal domain,

$$\min_I \|B(x, y) - I(x, y) \circledast k(x, y)\|_2^2 + \lambda \|I(x, y) \circledast l(x, y)\|_2^2 \quad (8)$$

in the frequency domain as

$$\hat{\mathbf{I}} = \hat{\mathbf{B}} \star \underbrace{\left(\frac{\hat{\mathbf{K}}}{|\hat{\mathbf{K}}|^2 + \lambda |\hat{\mathbf{L}}|^2} \right)}_{\hat{\mathbf{K}}_{\text{reg}}^{-1}}, \quad (9)$$

where $\hat{\mathbf{L}}$ is the DFT of \mathbf{L} and the absolute value $|\cdot|$ of a complex value $a + bi$ is $\sqrt{a^2 + b^2}$. Taking the real component of the inverse DFT of $\hat{\mathbf{I}}$ yields our regularized and deblurred image matrix \mathbf{I} . Note that in Equation 9 the regularized kernel $\hat{\mathbf{K}}_{\text{reg}}^{-1}$ is already in an inverted form, which is why an element-wise multiplication (\star) is employed instead of an element-wise division (\star^{-1} , as in Equation 6); i.e., with $\lambda = 0$, Equations 6 and 9 are equivalent.



Deliverable 6 [30%]

Laplacian-regularized Fourier Deconvolution Kernel: Complete the implementation of `LaplacianInverseFourierKernel` to form and return $\hat{\mathbf{K}}_{\text{reg}}^{-1}$. The function parameters match those of `FourierKernelMatrix`. Choose and hardcode a λ regularization coefficient value that yields a qualitatively good deblurred image (i.e., when applied to our `blurred_noisy_image` test data using Equation 9; see our test code in `__main__`: *part of your grade for this deliverable will be based on whether we can discern important details from the deblurred image*).

5 You're Done!

Congratulations, you've completed the 4th assignment. Review the submission procedures and guidelines before submitting your Python script file assignment solution. Recall that any test code you include in your mainline (`__main__`) will not be graded, however it must still respect the [collaboration/plagiarism](#) policies.