Multimedia Project

Fotiou Dimitrios
Aristotle University of Thessaloniki
Department of Electrical and Computer Engineering

***Abstract*—This document is a project for the Multimedia Systems course of department of Electrical and Computer Engineering in Aristotle University of Thessaloniki. The purpose of the project is to implement a simplified version of the MP3 protocol. Below, the different parts of the protocol are presented, along with implementation details. Finally, some results regarding the quality of the encoding are presented.**

***Index Terms*—multimedia, MP3, codecs, audio**

I. Introduction

The MP3 protocol [1] is a form of audio compression used for storing and transmitting audio files over the Internet and on audio playback devices. MP3 consists of a compression algorithm that reduces the resolution of the audio without significantly affecting its quality. MP3 is one of the most widely used audio compression protocols and is based on the ISO/IEC 11172-3 protocol.

The basic components of MP3 are a frequency analysis of the input signal and decorelation of the signal. Then, a controlled distortion of the signal is applied, influenced by a psychoacoustic analysis that reduces the quantization accuracy in areas of the spectrum where quantization error is not perceptible. After that, Run Length Encoding and entropy coding via Huffman are used.

Below is an analysis of the implementation of each of these steps, along with some diagrams required in the assignment. At the end of the report, we draw some conclusions regarding the quality of the decoded audio and the degree of compression.

II. Implementation

A. Subbands

The first part of the task focuses on the division of the input samples spectrum. The division is performed using $M = 32$ theoretically non-overlapping filters. The filters are calculated using the standard impulse response $h(n)$ 1 as

$$h_i(n) = h(h)\cos\left(\frac{(2i+1)\pi}{2M}n + \frac{(2i+1)}{4}\pi\right) \quad (1)$$

By transforming $h_i(n)$ to the frequency domain using the discrete Fourier transform, we can see the filters' responses in $dB(20\log(H(f)))$ 2. It is also useful to map them in barks instead of Hz because it represents more accurately the distances between different frequencies as perceived by the human ear.

From the above, we conclude that the filters are bandpass with a bandwidth of $\frac{f_s}{2M} \approx 689Hz$

Filtering of audio samples is done segmentally, and frames of size $N \times M$ are produced, where N is the samples of a subband and M is the number of subbands. Each audio segment needs to be convolved in the time domain with the filter responses. The subband analysis function provides us with a ready-made function, and we only need to implement an input buffer. Its size is $(N-1)M + L$, and the additional samples are needed to perform the convolution process correctly. In each iteration, the buffer shifts by NM samples. The output of the convolution is an $N \times 1$ matrix for each subband.

The inverse process involves generating input samples from frames. For the reconstruction from subbands, the filters G are used, which are designed to allow full reconstruction of the samples filtered with the H filters. Just like the analysis function, the synthesis function is also provided to us. This time, the input buffer has a size of $((N-1) + \frac{L}{2}) \times M$, where L is the size of the G synthesis filters. We iteratively read the frames and add them to the beginning of the buffer after

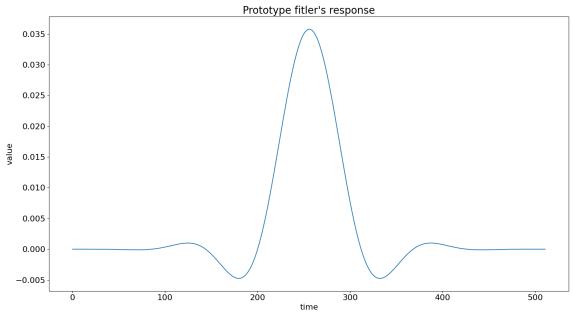

Figure 1. Prototype filter response

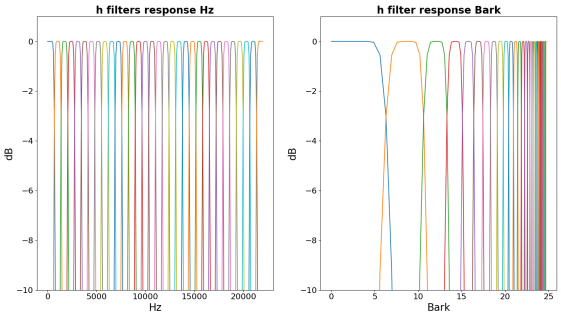

Figure 2. Filters' responses

shifting the previous elements.

As mentioned, theoretically, the two above processes of analysis and synthesis should allow for full reconstruction. However, due to the implementation of convolution with buffers, some samples in start are lost. Nevertheless, with proper shifting, the original and final time series are quite close, as shown in Figures 3 and 4. Additionally, there is no audible difference between the original and reconstructed sound, and the average SNR of the final signal is $\sim 70dB$ if we don't consider obvious zero error values that cause the SNR to be infinite.

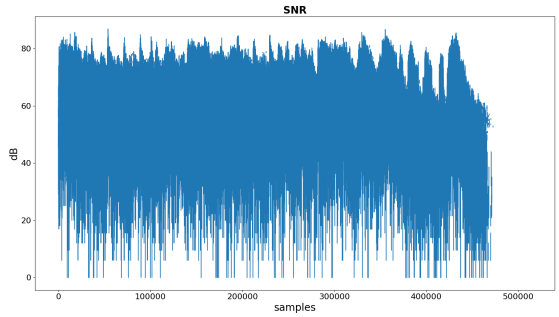

Figure 3. Reconstruction error SNR (dB) after inverting subband

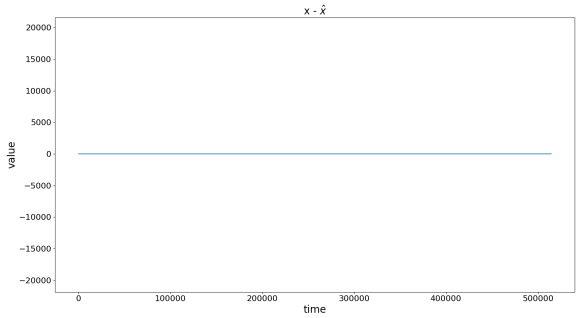

Figure 4. Reconstruction error $x - \hat{x}$ after inverting subband

In this subsection, we were asked to implement an initial version of the codec that encodes the data and then decodes it. The encoded data was successfully reconstructed, and the resulting sound is identical to the original one, with a minimal difference in amplitude. The codec implementation included splitting the signal into frames, applying analysis and synthesis functions to each frame, and encoding the resulting subband coefficients using variable length coding. The decoding process was the inverse of the encoding process. The codec is not yet optimized, but it provides a good starting point for future improvements.

B. Discrete Cosine Transform

After dividing the sound samples into frames and 32 subbands, we apply Discrete Cosine Transform (DCT) for the transfer from time domain to frequency domain and further partitioning of the spectrum. The dct function of scipy [2] was used to obtain the DCT coefficients. DCT is applied to each subband of a frame separately, and the coefficients are placed sequentially, one after the other. For the reverse

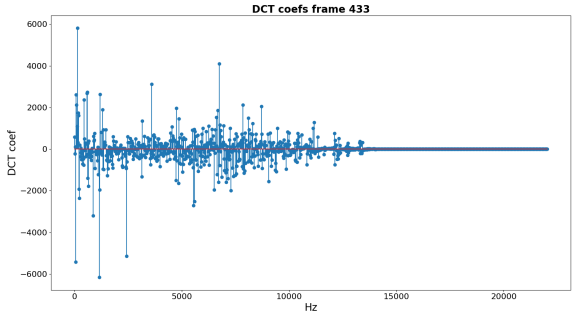

Figure 5. DCT coefficients of subbands example 1

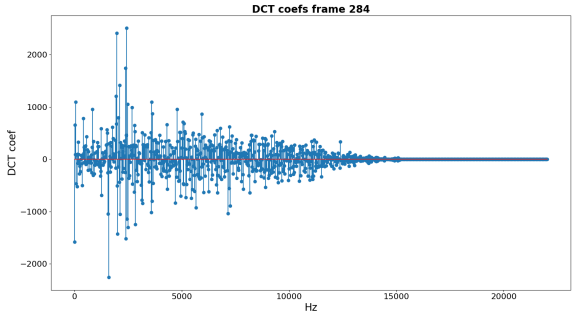

Figure 6. DCT coefficients of subbands example 2

process, the idct function of scipy was used to bring us back to the frequency domain. The inverse DCT is applied to the coefficients of each subband, and the original frame split into subbands in the time domain is finally created.

To verify the reconstruction from the DCT process, I calculate the sum of the squared difference between the original and final frame. I observe that I have a perfect reconstruction.

C. Psychoacoustic Model

Before moving on to selecting quantizers for each frame, we need to implement the psychoacoustic model. This model exploits the human auditory system's frequency threshold in silence, which sets a limit on the intensity below which frequencies are not perceived by the human ear. This threshold was derived from experiments and is provided ready-made in both the MP3 standard and this work.

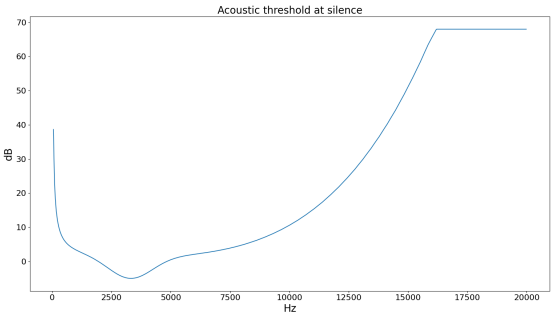

Figure 7. Acoustic threshold at silence

Practically, in the model, the frequencies that are not perceived by the human ear are decoded with a lower quality quantizer, as the error introduced will not be strongly perceived by humans. The model also exploits the property that frequencies close to strong tones are overshadowed by the strong tone and are not perceived. Therefore, the presence of strong tones would increase the threshold of audibility in a neighboring frequency region.

Examining the questions in this section, we first have the trivial calculation of the power of the DCT coefficients in dB. This is done as

$$P(k) = 10\log(|c(k)|^2) = 20\log(|c(k)|) \tag{2}$$

where $c(k)$ is the DCT coefficient of a frame. Then we have the implementation of Dksparse, which calculates a neighborhood matrix in some way. Specifically, if we symbolize this matrix with D, then $D(k, j) = 1$ if frequency j belongs to the neighborhood of frequency k. Finally, we need to implement the psychoacoustic model itself, which will determine the quantizer for each frequency in each frame.

$$\Delta_k \in \begin{cases} 2 & 2 < k < 282 \\ 2-13 & 282 \leq k < 570 \\ 2-13 & 570 \leq k < 1152 \end{cases} \tag{3}$$

Then the function that initializes the strong tones was implemented. A strong tone is considered any DCT coefficient that has power greater than its immediate neighbors and at least 7dB greater than the neighbors calculated above Δ_k. For each tone identified here, I take its power along with the sum of its immediate neighbors, in order to attribute to it the total power that, due to reduced clarity in the discrete spectrum, may have been shared among its neighborhood.

For the next basic function of the psychoacoustic model, we need to first implement the function that converts a frequency series to Barks. The formula is:

$$z(f) = 13\arctan(0.00076f) + 3.5\arctan((\frac{f}{7500})^2)(Bark) \tag{4}$$

Next, the implemented function is the ST_reduction which aims to filter the masking tones that have been detected. Specifically, it keeps only those above the hearing threshold in silence 7. Additionally, among those that are less than 0.5 barks apart, it keeps the one with the highest power.

Each tone has a different influence on the final hearing threshold, depending on its position. Before we calculate the influence of each tone, we need to find out how it spreads across different frequencies. The dispersion formula is given by

$$SF(i,k) = \begin{cases} 17\Delta_z - 0.4P_M(k) + 11 & -3 \leq \Delta_z < -1 \\ (0.4P_M(k) + 6)\Delta_z & -1 \leq \Delta_z < 0 \\ -17\Delta_z & 0 \leq \Delta_z < 1 \\ (0.15P_M(k) - 17)\Delta_z - 0.15P_M(k) & 1 \leq \Delta_z < 8 \end{cases} \tag{5}$$

where k is the frequency of the masking tone, i is the frequency at which we examine the influence of the k tone, and Δ_z is the difference between the i and k tone in barks.

The spreading of each tone to different frequencies is utilized in calculating the overall influence of each masking tone on the hearing threshold in silence. The formula that calculates the influence is:

$$T_M(i,k) = P_M(k) - 0.275z(f_k) + SF(i,k) - 6.025 \tag{6}$$

where k is the tone we are examining the influence on the i frequency. Having calculated the effect of each masking tone, we can easily calculate the overall acoustic threshold in Global_Masking_Thresholds as the sum of the powers of the threshold in silence and the overall threshold of the tones at each frequency. Here, it is important to note that the sum is not done in dB, but the final result is converted to dB.

To summarize regarding the psychoacoustic model, we implement algorithm 1, which uses the above functions to calculate the final threshold of a frame.

Algorithm 1 Psychoacoustic Model

Input: c, D, Tq
Output: Tg

1: $ST = STinit(c, D)$
2: $STred, STpower = STreduction(ST, c, Tq)$
3: $tm = MaskingThresholds(STred, STpower, Kmax)$
4: $Tg = GlobalMaskingThresholds(tm, Tq)$

Two examples of the auditory threshold of a frame are shown in figures 8 and 9. In the first example, there is a significant influence of masking tones, while in the second one, the threshold is mainly determined by the auditory threshold in silence.

D. Quantization

After the psychoacoustic model, the next step is to implement the quantizer. Initially, we need to determine to which critical band each function belongs in critical_bands. The different frequency groups are shown in 10.

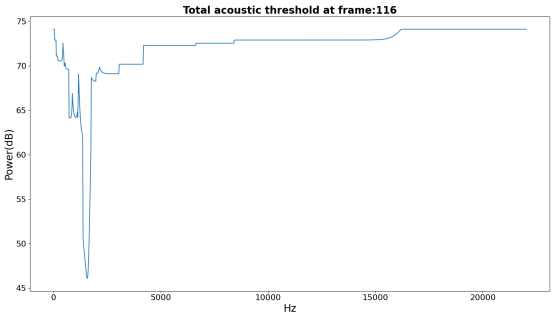

Figure 8. Total acoustic threshold example 1

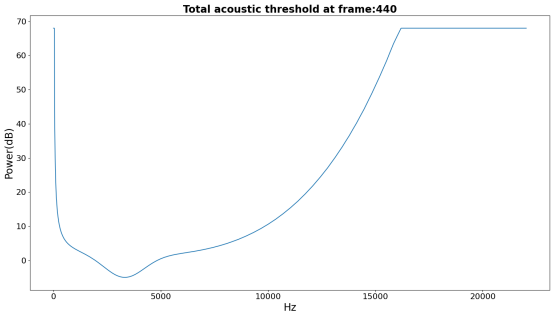

Figure 9. Total acoustic threshold example 2

Figure 10. Critical Bands

Band No.	Center Frequency (Hz)	Bandwidth (Hz)
1	50	-100
2	150	100-200
3	250	200-300
4	350	300-400
5	450	400-510
6	570	510-630
7	700	630-770
8	840	770-920
9	1000	920-1080
10	1175	1080-1270
11	1370	1270-1480
12	1600	1480-1720
13	1850	1720-2000
14	2150	2000-2320
15	2500	2320-2700
16	2900	2700-3150
17	3400	3150-3700
18	4000	3700-4400
19	4800	4400-5300
20	5800	5300-6400
21	7000	6400-7700
22	8500	7700-9500
23	10500	9500-12000
24	13500	12000-15500
25	19500	15500-

The set of frequencies that fall within a critical band are quantized with the same quantizer. However, before quantization, they are normalized. Specifically, for the coefficients of each band, the scale factors of each critical zone are calculated as follows:

$$Sc(band) = max(|c(i)|^{\frac{3}{4}}) \tag{7}$$

and the normalized coefficients are calculated as

$$\tilde{c}(i) = sign(c(i)) \frac{|c(i)|^{\frac{3}{4}}}{Sc(band)} \tag{8}$$

the normalized coefficients are quantized, while the scale factors of each critical band are stored and used during dequantization.

Then, the quantizer is implemented, which takes as input the number of bits to be used and the normalized DCT coefficients. The levels of the quantizer are given by

$$\begin{aligned} d = [&-1, -(2^{b-1}-1)w_b, -(2^{b-1}-2), ..., \\ &-wb, wb, ..., (2^{b-1}-2)w_b, (2^{b-1}-1)w_b, 1] \end{aligned} \tag{9}$$

where $w_b = \frac{1}{2^b - 1}$. The quantizer is non-uniform, with the two levels on each side of zero merged into one, resulting in a total of $2^b - 1$ zones. The dequantizer reverses the above process and maps each symbol to the center of the corresponding level.

Once the quantizer and dequantizer have been implemented, the process of quantizing each critical band must also be implemented. Depending on the overall threshold of audibility for each band, it will be encoded with a different quantizer, meaning a different number of bits. For this purpose, for each critical band, the quantization process starts with a few bits. Then, the dequantization process is performed, and the error is estimated. If it is below the threshold of audibility for that specific band, that quantizer is chosen. Otherwise, the number of bits is increased. Practically, we do not want the final quantization/dequantization error to be significantly noticeable to the human ear.

The reverse process is relatively simple since the bits of each quantizer for each critical band and the scale factors of the DCT coefficients have been stored.

E. Run Length Encoding

After calculating a symbol for each DCT coefficient, we apply run-length encoding. This algorithm looks for consecutive zeros and encodes them with the initial symbol along with a number that indicates how many times the symbol 0 is repeated. Therefore, the function returns an $R \times 2$ array, where R is the number of lengths that were detected. The first column stores the first symbol, while the second stores how many times the symbol 0 was repeated.

The reverse process is done by reading each row of the array extracted by RLE and repeating the element 0 after the element in the first column as many times as indicated by the second column.

This step is useful because in several frames, we have many repeated zeros. Although we store both the symbol and an additional number, it turns out that the total number of elements needed to be stored is smaller. However, this assumes, as I said, that there are enough and long enough paths, otherwise, this step leads to an increase in memory requirements.

F. Huffman

Here, we focus on entropy coding. The Huffman code requires the creation of a binary tree. For this reason, some subroutines were implemented that define a node of a tree and construct it sequentially, starting from the top.

Initially, the distinct symbols are identified along with their frequencies. Then, the tree is built by creating a node for each symbol and combining the two nodes with the lowest frequency until a single node is left, which is the root of the tree.

Next, we assign a binary code to each symbol by traversing the tree. At each node, we assign a 0 to the left child and a 1 to the right child. The binary code of each symbol is then the sequence of 0s and 1s from the root to the leaf corresponding to that symbol.

Finally, we apply the Huffman code to the output of the run-length encoding step. We first write the codebook to the output file, which contains the binary code for each symbol. Then, we encode the output by replacing each symbol with its binary code.

The Huffman coding step reduces the number of bits needed to represent the data, which is particularly useful in cases where the input data has a high degree of redundancy.

After constructing the tree, we search for all the unique symbols, and each time we visit a left node, we add a 0 to the code sequence of the symbol. Conversely, if we visit a right node, we add a 1. This process leads to the creation of a dictionary with which we map each symbol to a binary string. The symbols that appear more frequently have shorter bit sequences because they are higher up in the binary search tree.

The inverse process is done by utilizing the symbols and their corresponding frequencies that we stored in the Huffman process. With this, we reconstruct the tree and the mapping of symbols to 0 and 1 bitstreams. By reading the bits of the encoded frame one by one, we recognize symbols. This can be done because there is no encoded symbol that is a prefix of another encoded symbol.

G. MP3 codec

As a final stage, we combine the above routines to implement a coder function for encoding signals in MP3, a decoder function for reversing the previous process, and finally a codec for combining the coder and decoder.

In 2, 3, 4 we observe a simplified version (not all inputs and outputs are included) in pseudocode that presents the main flow of the encoder, decoder, and codec.

It is worth noting that during the process of extracting the Huffman bits, as indicated by the problem statement, the sequence of 01 was large and stored in a binary file. Additionally, the additional information required to reconstruct the sound from the bitstream was stored in a separate file. These were the scale factors of the DCT coefficients, the number of bits used for the quantizer of each critical band, in each frame, the frequency table for reversing the Huffman coding, and finally the number of bits in each frame. The latter is not generally part of the standard, but I used it to easily know how many bits to read from the binary file for each frame.

Algorithm 2 MP3 coder simplified

Input: x
Output: bitstream

$bitstream = []$
$yframes = subband\_analysis(x)$
for $frame \in yframes$ **do**
  $dct\_c = dct(frame)$
  $Tg = psychoacoustic\_model(dct\_coef)$
  $dct\_quant = quantize(dct\_c, Tg)$
  $rle\_sym = RLE(dct\_quant)$
  $huff\_syms = huffman(rle_sym)$
  $bitstream.append(huff\_syms)$

Algorithm 3 MP3 decoder simplified

Input: bitstream
Output: x_hat

$yframes = []$
for $frame\_bits \in bitstream$ **do**
  $rle\_syms = inv\_huffman(frame\_bits)$
  $dct\_quant = inv\_RLE(rle\_syms)$
  $dct\_quant = dequantize(dct\_quant)$
  $yframe = inv\_dct(frame)$
  $yframes.append(yframe)$
$x\_hat = subband\_synthesis(yframes)$

For the correctness of the above routines, a demonstration script was created with three options. The first is the codec option that encodes and decodes an audio signal and stores it, while also printing the compression ratio. For the compression ratio, in addition to the bitstream, I take into account the additional information stored, considering 16 bits per number.

The second option is the coder that only encodes the signal and stores the bitstream and the extra information in a folder. The bitstream is stored in binary, so the file size is realistic. For the extra information, it is stored in text format, with each frame's information separated by a newline.

Finally, the third option is the decoder, which reads the bitstream and the extra information from the folder and reconstructs the audio signal.

Last operation, which can only be performed after the coder or codec has been executed, is the decoder that decodes and stores the signal in PCM with 16bits/sample. To perform this, it uses the stored bitstream and the extra information.

Algorithm 4 MP3 codec simplified

Input: x
Output: x_hat

$bitstream = MP3coder(x)$
$x\_hat = MP3decoder(bitstream)$

III. Results

For the final results, it is worth mentioning the conventions that were taken. For the calculation of the size of the initial file, the 16bits/sample of the PCM were used. For the final file, in addition to the bitstream, the extra information is considered to be 16bits/number. Of course, the compression ratio is also shown in the table below without the extra information. As compression ratio, we consider the formula

$$CompressRatio = \frac{Uncompressed \quad Size}{Compressed \quad Size} \tag{10}$$

while the percentage of space freed up is calculated as

$$SpaceSaving = 1 - \frac{Compressed \quad Size}{Uncompressed \quad Size} \tag{11}$$

Acoustically are not significantly affected by the smoothing.

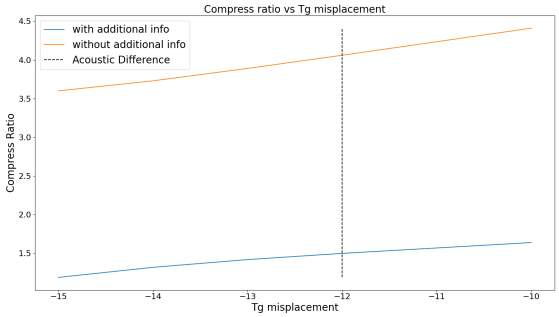

Figure 11. Compress ratio for different Tg displacement

Of course, in order for this to happen, it was necessary to shift the Tg downwards by 12-15dB. The relationship between compress ratio and Tg shift is shown in Figure III. For example, the final average SNR of the signal is approximately 10dB for a Tg shift of -13dB.

As stated in the assignment, this was expected because we implemented a simplified version of MP3. The lower we go on the curve, the stricter the quantizer is, and it encodes the signal with more bits. The advantage is that the reconstructed signal sounds better, but the compression ratio is reduced.

Finally, regarding the execution time, the average time for encoding is 1 minute, and for decoding it is 20 seconds.

In conclusion, implementing an MP3 protocol has a basic structure based on general ideas of data compression such as data de-interleaving, compression, and entropy coding. It differs from other compressions for different input forms, such as images, because it utilizes a psychoacoustic model to control quantization error. Although this implementation is a simplified version of MP3, it achieves decent compression with minimal acoustic cost to sound quality.

References

[1] F. I. for Integrated Circuits (IIS), "Mpeg audio layer-3 (mp3) coding technology," Fraunhofer Institute for Integrated Circuits (IIS), Tech. Rep., 1991.

[2] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.