Journal
doi.10.1093/journal/xxxxxx
Advance Access Publication Date: N/A
Manuscript Category

Data and text mining

Image-centric compression of protein structures improves space savings

Luke Staniscia [1,*], Yun William Yu [2,*]

[1]Department of Mathematics, University of Toronto, Toronto, M5S 1A1, Canada and
[2]Department Mathematics and Computer Science, University of Toronto, Toronto, M5S 1A1, Canada.

*To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on 08/27/2021; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: Because of the rapid generation of data, the study of compression algorithms to reduce storage and transmission costs is important to bioinformaticians. Much of the focus has been on sequence data, including both genomes and protein amino acid sequences stored in FASTA files. However, there are few specialized compressors for structural protein data contained in PDB and mmCIF files. Current standard practice is to use an ordinary lossless compressors such as gZip on a sequential list of atomic coordinates, but this approach expends bits on saving an arbitrary ordering of atoms, and it also prevents reordering the atoms for compressibility. In this article, we demonstrate for the first time that image-based compression can significantly improve compression ratios. To this end, we implement a prototype compressor 'PIC', specialized for point clouds of atom coordinates contained in PDB and mmCIF files. PIC maps the 3D data to a 2D 8-bit greyscale image and leverages the well developed PNG image compressor to minimize the size of the resulting image, forming the compressed file.
Results: PIC outperforms gZip in terms of compression ratio on proteins over 20,000 atoms in size, with a savings over gZip of up to 37.4% on the proteins compressed. In addition, PIC's compression ratio increases with protein size.
Availability: Prototype Python source code is available for download at https://github.com/lukestaniscia/PIC.
Contact: luke.staniscia@mail.utoronto.ca, ywyu@math.toronto.edu

1 Introduction

For over half a century, determining protein structure has been a primary means of understanding function and behavior (Ramachandran, 1963; Ilari and Savino, 2008). After proteins are characterized by researchers using various methods such as X-ray crystallography, NMR spectroscopy, and cryo-electron microscopy, various files are generated describing the protein and stored in online repositories such as the Protein Data Bank (Rose *et al.*, 2016; Berman *et al.*, 2012). One such file, the FASTA file, contains strings of characters representing the amino acids that make up the protein and its variants (Pearson, 1994). Other files, such as PDB (Protein Data Bank format) and mmCIF files, contain structural information about the protein (Westbrook and Fitzgerald, 2003). Although the Protein Data Bank is no longer growing exponentially, the number of new structures deposited is still quite formidable Berman *et al.* (2012).

FASTA files are used for storing both protein and genomic sequence information, and much work has been done to create customized sequence compression algorithms. It bears mentioning that the genomic sequence compression literature has recently seen significantly more activity with the advent of next-generation sequencing (Fritz *et al.*, 2011; Daniels *et al.*, 2013; Yu *et al.*, 2015; Hernaez *et al.*, 2019), and many protein sequence compressors take advantage of that work. For protein sequences, Hategan and Tabus (2004) introduce a single and double pass version of a amino acid sequence compressor for FASTA files that makes use of substitution matrices. MFCompress was introduced by Pinho and Pratas (2013), and expresses the amino acid sequences in their corresponding DNA bases, divides the data into three streams, and compresses the resulting streams. CoMSA is another compression algorithm for FASTA files introduced by Deorowicz *et al.* (2018) based on a generalized Burrows-Wheeler transform. Similarly to MFCompress, The Nucleotide Archival Format (NAF) introduced by Kryukov *et al.* (2019) is another compressor that works on amino acid sequences converted to their corresponding DNA bases by dictionary encoding this transformed string.

In addition to directly transforming and compressing the sequences in FASTA files, a significant amount of research has gone into read-reordering algorithms for genomic sequences in the BEETL (Cox *et al.*, 2012), SCALCE, (Hach *et al.*, 2012), MINCE (Patro and Kingsford, 2015), and

more. These methods are applicable when FASTA (and the related FASTQ) files are used to store multiple small fragments ('reads') of sequences; next-generation sequencing produces these reads in no particular order, so the reads can be safely reordered without losing important information. When properly performed, this reordering can significantly improve the compression ratio of standard compressors.

On the other hand, the primary data component of PDB and mmCIF protein structure files is a point cloud of coordinates belonging to the atoms that make up the protein. In the standard formats, each atom has its own separate ASCII-formatted line entry in the file that contains the type of atom, type of amino acid to which it belongs, atom and amino acid identifiers, followed by three floating point Cartesian coordinates, along with other information. The coordinates are measured in units of Angstroms $\mathring{A}$, where $1\mu m = 10,000\mathring{A}$ (Goodsell, nd). Unlike their FASTA counterparts, little work has been done to create compressors customized for the structural data contained in PDB and mmCIF files.

Valasatava *et al.* (2017) did a deep investigation on compressing 3D coordinates of atoms in proteins by investigating a full gamut of compression techniques. Their final recommendation was to apply "intramolecular compression", which aims to reduce the size of each protein via three steps: encoding, packing, and entropy compression. The encoding step transforms floating point coordinates into alternate representations, such as Integer, Delta, Predictive, Wavelet, and Unit Vector encodings. Integer encoding multiplies the floating point location coordinates by a power of 10 and rounds the result to the nearest integer. This encoding strategy is often lossy as not all decimal places of precision are kept in the integer encoded value. However, some amount of loss of precision is acceptable because of both measurement error, and due to the natural uncertainty of exact atom locations in a protein. We will use a variant of integer encoding in our algorithm PIC; for details on the other encodings, see (Valasatava *et al.*, 2017). After packing the encoded coordinate vectors using either recursive indexing or variable packing, the resulting packed coordinates are entropy encoded using standard methods like gZip (Deutsch *et al.*, 1996) or brotli (Alakuijala *et al.*, 2018), which are both combinations of LZ77 dictionary based encoding and Huffman encoding.

However, Valasatava *et al.* (2017)'s investigation focused primarily on compression of atomic coordinates as sequential objects stored within a text file, treating the data as sequential, much like in FASTA files without reordering. Unlike protein/genomic sequences, of course, 3D atomic point clouds are not naturally sequential, so carefully preserving the order of the atoms as listed out in a PDB or mmCIF file is not as important (or some would argue, important at all). Given the background above, one logical next step would be to perform a principled reordering of the atoms to improve compressibility, but of course, how to perform that reordering is not clear a priori as point clouds are very different from sequenced genomic reads in underlying structure. To resolve this question, we turn to an alternate paradigm for compressing point cloud data sets proposed in the field of LIDAR imaging. Houshiar and Nüchter (2015) proposed a new compression algorithm for the 5D point cloud data generated by LIDAR scans of real-world scenes. The LIDAR scans produced tuples of data points containing coordinates of a point in space in the scene, along with reflectance and colour data of the surface at that location. Their compression algorithm converts the Cartesian coordinates to spherical coordinates, maps the angular coordinates to the axes of an image, and the radial component, colour, and reflectance data to pixel's fields at the mapped location. The radial component, colour, and reflectance data are written to the R, G, and B components respectively of a single coloured image as well as the greyscale intensity field of three separate consecutive images. The resulting images were compressed using PNG, JPEG 100 (lossless, perfect quality JPEG), JPEG2000, no compression TIFF, LZW TIFF, and Pack Bits TIFF lossless image compressors. The authors found that compressing three greyscale images using the PNG compressor performed the best in terms of compression ratio.

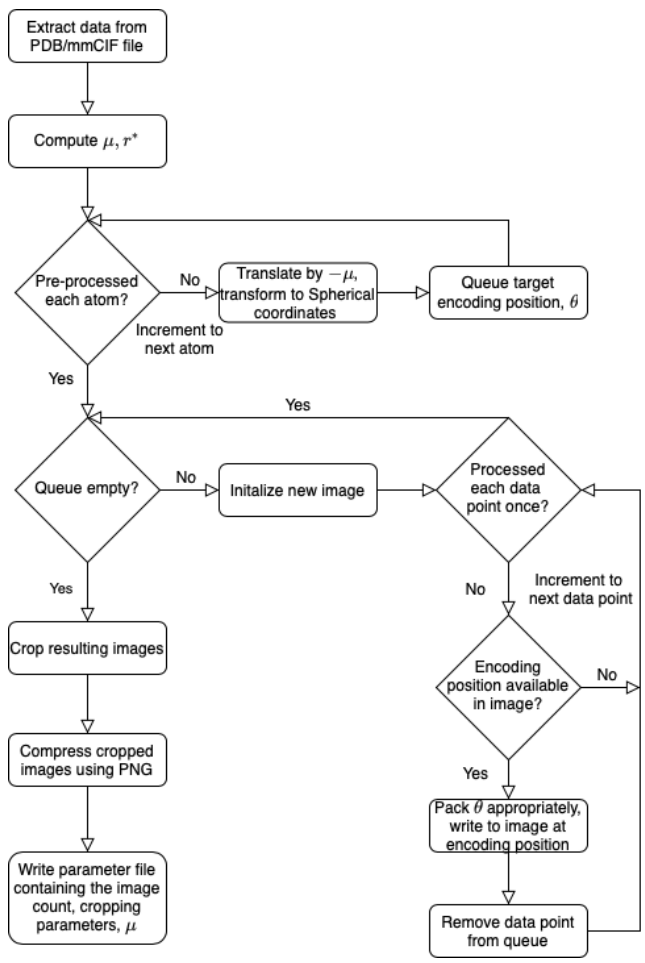


Fig. 1: Flow chart diagram of the PIC compression algorithm.

In this manuscript, we take inspiration from the NGS read-reordering literature and combine the intramolecular compression techniques of Valasatava *et al.* (2017) with the image-centric methods of Houshiar and Nüchter (2015). In section 2, we outline our new compression algorithm, PIC, for the structural protein data contained in PDB and mmCIF files. Design choices and methodology are examined in detail followed by a pseudo-code outline of the compression algorithm. In section 3, we give compression results for 20 proteins compressed using both PIC and gZip and show PIC outperforms gZip in terms of compression ratio for all of these proteins over 20,000 atoms in size. We also give the images that constitute the compressed files for a few of the compressed proteins. In section 4, we highlight some trends in the compression results and make note of the advantages of the PIC compressor over gZip for structural protein data compression.

2 Algorithm

The PIC compression algorithm has has three main components, namely mapping each atom to a position in an image, encoding information at that position, and compressing the resulting image. A high-level overview is given in algorithm 1 and figure 1, and details are furnished in the following text.

Algorithm 1 PIC compression algorithm

1: Read PDB/mmCIF file
2: Compute μ and r^*
3: **for** each atom in the protein **do**
4: Translate atom's Cartesian coordinates by $-\mu$
5: Convert the translated coordinates to spherical coordinates
6: Compute target encoding position (x, y)
7: Queue $[(x, y), \theta]$
8: **end for**
9: **while** the queue is not empty **do**
10: Initialize a new image
11: **for** each data point in the queue **do**
12: **if** (x, y) is not available **then**
13: Compute (x^*, y^*)
14: **end if**
15: **if** an encoding position is available in the current image **then**
16: Pack θ based on the encoding position
17: At the encoding position, write packed θ
18: Remove the data point from the queue
19: **end if**
20: **end for**
21: **end while**
22: Crop the resulting image(s)
23: Compress the cropped image(s) using the PNG compressor on the highest compression ratio setting
24: Construct the parameter file with the image count, compression parameters, and μ to enable decompression

2.1 Mapping

Cartesian coordinates of atoms stored in the protein's PDB or mmCIF file are extracted and the global centroid μ of all the coordinates is computed. The coordinates are translated by $-\mu$ so that the global centroid becomes the new reference point or origin for the coordinates. This transformation minimizes the instances of collisions when mapping the coordinates to the image. To decompress the images, μ is stored along with the images.

The translated coordinates are then transformed to spherical coordinates. Each spherical coordinate component is rounded to a precision of one decimal place. Valasatava *et al.* (2017) noted that experimental measurements that produce the Cartesian coordinates determine an atom's position with a degree of uncertainty, greater than $0.2\mathring{A}$. This allows for the exploitation of lossy compression to store the coordinates only up to a tenth of an $\mathring{A}$, which is generally sufficient to preserve the essential structural information provided by lossless representation.

The radial r and azimuth ϕ spherical coordinate components of each atom are positionally encoded to the horizontal and vertical axis of an eight bit pixel greyscale image as follows

$$(x, y) = (10r, \varepsilon\phi)$$

where ε is a user-defined parameter that sets the number of pixels per azimuth angle degree. Letting r^* be the maximal radial component across all spherical coordinates, the dimensions of the resulting image are $10r^* \times 360\varepsilon$. Further note that while $x \in \mathbb{Z}_{\geq 0}$, y is not necessarily an integer. However, ε is chosen such that $360\varepsilon, 8y \in \mathbb{Z}_{\geq 0}$ for all y and $\varepsilon \geq 1.25$. This ensures there is at least one bit available per tenth of an azimuth angle degree and each y coordinate has an integer bit-level position on the vertical axis. In this way, we view each column in the image as a bit sting that is being written to.

Care must be taken when choosing ε. Setting ε too large will produce a large image, degrading the compression ratio. On the contrary, choosing a small ε will induce more collisions when data is mapped to the image. This results in increased compression time, as alternate data storage locations need to be considered. A decrease in the compression ratio may also be experienced in this case as more data points will need pointers to their intended locations and additional images may need to be populated to store all the required data.

The remaining elevation angle θ is stored in the image's pixel intensity values beginning at the data point's (x, y) encoding position in the image. Further details on how the elevation angle is formatted or *packed* and stored in the image is described in section 2.2. This encoding scheme was selected as it positionally encodes the spherical coordinates r and ϕ with the largest range of values and encodes the smallest ranging coordinate θ in the image's pixel's intensity values. Thus each coordinate takes up the fewest amount of pixels when encoded into the image, allowing for more data to be stored in the image before another image needs to be generated.

In the event that a data point is mapped to a position that does not have availability to hold all the required information, an alternate encoding position (x^*, y^*) is determined systematically. A position (x, y) has availability if all bits at positions between and including (x, y) and $(x, y + l - 1/8)$, where l is the length of the encoded elevation angle in bytes, have not had data previously written to them. Beginning at the data point's target encoding position (x, y), the positions $(x, y + i/8 \mod 360\varepsilon), 0 < i < 8 \cdot 360\varepsilon = 2880\varepsilon$ are scanned subsequently to find the first position with availability. This position is the alternate encoding position. All encoding positions (x, y) also satisfy $y < y + l - 1/8 < 360\varepsilon$, ensuring no data points begin at the bottom of the image and finish at the top to enable proper decompression of the image.

If (x^*, y^*) is the alternate encoding position for a data point with target position (x, y), and $y \leq y^* < y + 0.1\varepsilon$, the encoded elevation angle is stored begining at (x^*, y^*) as is. Otherwise, a pointer p is encoded and stored along with the encoded elevation angle at (x^*, y^*). p points to the largest $y' \in \{i/8 | 0 \leq i < 2880\varepsilon\}$ that satisfies $y \leq y' < y + 0.1\varepsilon$, namely $y' = y + 0.1\varepsilon - 1/8$. The stored pointer is the integer $p = 8(y^* - y')$. Note that $p > 0$ as $y^* > y'$. The decompressor then knows that the intended azimuth angle for the data point is that belonging to the position $(y^* - p/8) = y'$.

In the case that an alternate encoding position cannot be found in the current image, another image is generated, if not already done by a previous data point. The above mapping procedure is repeated in that image to locate an encoding position for the data point. This process repeats until an encoding position is determined for each atom's coordinate.

2.2 Packing

Elevation angles are stored in pixels' greyscale values begining at their corresponding data point's (x, y) encoding position. Each pixel has an 8-bit intensity field. Due to the variable lengths of the binary elevation angles and use of pointers, the following packing scheme is used to store the elevation angles so they can be properly decompressed.

If no pointer is needed, the elevation angle is integer encoded as 10θ and converted into its binary representation. If the binary representation has length less than $\lceil \log_2(1801) \rceil = 11$ bits, 0 bits are added to the front until the representation is 11 bits long. Two additional bits 1 and 0 are added to the front of the resulting binary string in that order to signify the start of a new data point and to notify the decompressor the data point has no pointer, respectively.

If a pointer is required, a similar but expanded packing scheme is used. The second bit is set to 1 instead of 0 to signify to the decompressor that the data point has a pointer. The pointer p is converted to its binary representation and prefixed with 0 bits until it has length $\lceil \log_2(2880\varepsilon) \rceil$. The adjusted binary representations of the pointer and elevation angles follows the two bit prefix in that order.

Protein ID	Atom Count	Original File Size (KB)	Rounded Coordinates Text Size (KB)	Rounded Coordinates Binary Size (KB)	gZip Size (KB)	gZip CR	PIC Size (KB)	PIC CR	Compression Time (min:sec)	Images Number Used	Images Space Used (%)	Decompression Time (min:sec)
2ja9	1458	163.3	24.1	6.6	**6.3**	**3.834**	10.0	2.412	0:0.5	1	[0.9]	0:1.3
2jan	12591	1101.2	206.1	56.7	**54.1**	**3.813**	61.2	3.368	0:3.2	1	[2.7]	0:5.8
2jbp	27367	2397.4	447.8	133.4	130.2	3.439	**108.8**	**4.117**	0:10.2	1	[11.1]	0:10.5
2ja8	32000	2831.2	507.6	144.0	139.6	3.637	**138.0**	**3.678**	0:13.2	1	[6.5]	0:14.3
2ign	41758	3579.1	666.7	187.9	180.8	3.688	**147.3**	**4.526**	0:25.4	1	[9.5]	0:24.0
2jd8	50351	4457.6	828.1	226.6	219.7	3.769	**196.8**	**4.207**	0:41.9	1	[7.7]	0:35.9
2ja7	63924	5605.5	1077.0	287.7	278.6	3.866	**258.8**	**4.161**	1:6.9	1	[10.2]	1:2.1
2fug	73916	6386.9	1180.7	360.3	347.5	3.398	**283.3**	**4.168**	1:34.0	1	[10.7]	1:26.4
2b9v	80710	6818.4	1279.8	393.5	379.4	3.373	**289.0**	**4.428**	1:48.0	1	[10.3]	1:48.4
2j28	95358	8152.3	1526.2	429.1	412.2	3.702	**346.6**	**4.403**	2:35.6	1	[13.7]	2:43.3
6hif	118753	12726.2	2105.2	534.4	516.2	4.078	**372.2**	**5.656**	4:21.7	2	[34.0, 0.1]	5:3.4
3j7q	140540	16027.2	2529.7	737.8	707.6	3.575	**475.6**	**5.318**	6:18.4	1	[20.3]	7:36.2
3j9m	158384	17995.2	2845.4	772.1	765.8	3.716	**525.7**	**5.413**	8:18.4	1	[21.7]	9:54.6
6gaw	178372	20825.4	3179.9	869.6	862.1	3.688	**587.6**	**5.411**	10:49.1	1	[23.5]	12:55.6
5t2a	200172	22787.6	3253.9	900.8	872.4	3.73	**651.7**	**4.993**	15:17.6	2	[31.1, 1.7]	16:36.6
4ug0	218776	24906.9	3841.4	1066.5	1056.7	3.635	**707.3**	**5.431**	17:50.7	2	[33.8, 1.7]	20:9.5
4v60	241956	24377.8	4207.8	1179.5	1167.2	3.605	**730.2**	**5.762**	18:34.3	2	[45.6, 2.1]	23:19.8
4wro	260090	35661.1	4363.1	1267.9	1246.2	3.501	**848.8**	**5.14**	25:33.1	1	[29.6]	30:3.3
6fxc	281510	31329.0	5067.1	1477.9	1424.2	3.558	**917.7**	**5.522**	29:24.4	2	[34.6, 1.0]	32:55.4
4wq1	299951	40130.9	5042.1	1462.3	1438.0	3.506	**968.8**	**5.204**	33:10.3	2	[34.7, 0.2]	39:54.2

Table 1. PIC compression algorithm, $\varepsilon = 2.5$, results. Rounded Coordinates Text Size and Binary Size are the sizes of the text and binary files, respectively, that contain only the Cartesian coordinates found in the original file, rounded to one decimal place. The binary file is then gZipped. The gZip and PIC compression ratios (CR) are the ratios of the Rounded Coordinates Text Size to the size the gZip file and PNG image output(s) from the PIC compressor, respectively. Bolded values are the best of gZip and PIC. Compression and decompression times are for the PIC algorithm, though because our PIC implementation is not optimized for speed, these numbers should only be used to compare against Table 2. Image Space Used is a tuple that gives the proportion of the image space that was used to encode the protein coordinate data, or part thereof, in each image constructed by the PIC compressor.

For $0 \leq i < 8l$, bit i of the packed string is mapped to the bit at position $(x, y + i/8)$ in the image. This packing scheme ensures that each data point has one of two possible lengths, the exact length of which can be determined by the second bit located at $(x, y + 1/8)$. This is a key feature that allows for the proper decompression of the image.

2.3 Cropping and Compression

The resulting image(s) are cropped and compressed using the PNG lossless image compressor on the highest compression ratio setting. These image(s) make up the compressed version of the protein's point cloud of atom coordinates in the PDB or mmCIF file.

Images are cropped to remove any all-black rows and columns on the edge of the image. To decompress the images, two cropping parameters are stored along with each image generated to reverse the cropping.

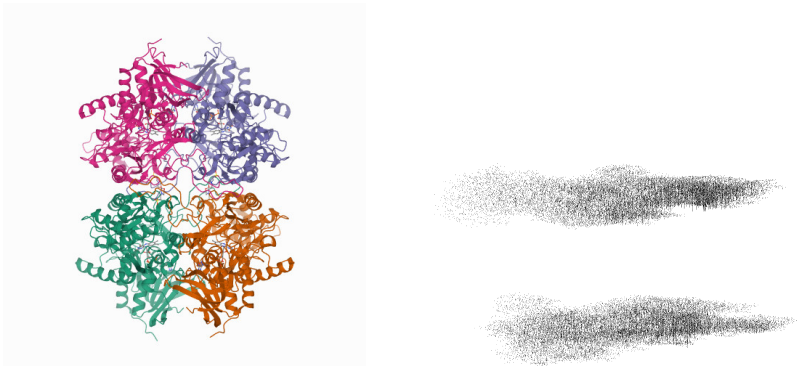
Other lossless image compressors investigated in Houshiar and Nüchter (2015) were also examined. Similarly to the results found by Houshiar at al., PNG was selected for use in the algorithm as it offers the highest compression ratios of the aforementioned compressors at comparable compression times.

2.4 Decompressed File

The original and decompressed files are identical up to the coordinates of the atoms. As noted in section 2.1, since there is a tolerance of up to $0.2\mathring{A}$ in each coordinate component, each decompressed coordinate is within a euclidean ball of radius $0.2\sqrt{3}\mathring{A}$ about the original coordinate.

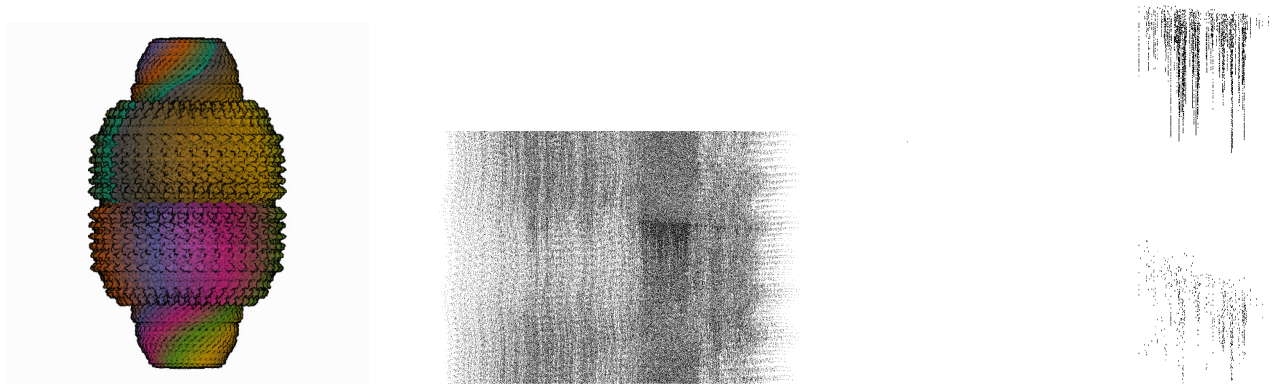
Protein ID	Compression Time (min:sec)	Decompression Time (min:sec)
2ja9	0:0.3	0:1.1
2jan	0:1.6	0:3.8
2jbp	0:3.3	0:7.5
2ja8	0:4.6	0:8.1
2ign	0:7.6	0:14.2
2jd8	0:10.5	0:20.4
2ja7	0:15.2	0:33.0
2fug	0:20.8	0:45.4
2b9v	0:27.4	0:58.6
2j28	0:37.9	1:21.1
6hif	1:21.3	2:11.8
3j7q	1:45.1	3:1.5
3j9m	2:13.7	3:56.1
6gaw	3:14.8	5:9.1
5t2a	5:7.6	6:24.0
4ug0	5:42.6	7:44.6
4v60	6:8.4	9:18.6
4wro	6:43.3	11:6.3
6fxc	9:33.1	12:31.5
4wq1	10:48.9	14:59.6

Table 2. PIC compression algorithm, $\varepsilon = 2.5$. Compression and decompression times are for the PIC algorithm reconstructing only the 3D Cartesian coordinates, metadata is not maintained. Note that because our PIC implementation is not optimized for speed, these numbers should only be used to compare against Table 1; when metatdata is not maintained, the runtime is significantly decreased, suggesting that most of the runtime is spent on reconstructing the initial arbitrary ordering of atoms in the text format.



(a) 3D structure Divne (2006) (b) PIC compressor output 1/1

Fig. 2: 3D structure and PIC compressor PNG image output for 2ign. Some attributes and symmetries in the 3D structure are observed in the corresponding PIC-compressed image. The upper and lower parts of the 3D structure of protein 2ign can be seen in PIC generated image as two separate masses of black pixels, one over the other.



(a) 3D structure Kato *et al.* (2014) (b) PIC compressor output 1/2 (c) PIC compressor output 2/2

Fig. 3: 3D structure and PIC compressor PNG image output for 4v60. Some attributes and symmetries in the 3D structure are observed in the corresponding PIC-compressed image. The spiked edge of the 4v60 protein can be seen on the right side of the first outputted image from the PIC compressor.

3 Results

Table 1 gives statistics and compression results on 20 proteins compressed using gZip and PIC where $\varepsilon = 2.5$ and the decompressed files are identical to the original with the lossy coordinate transform. Table 2 gives similar results where only the 3D coordinates are reconstructed in the decompressed file and metadata is not maintained from the original file. Figures 2 and 3 compare the 3D structures of proteins to the images created by the PIC compressor. Figures 4 and 5 visualize some of the results found in table 1.

As can be seen from table 1, the proposed PIC algorithm has superior compression ratio performance than the standard gZip text compressor for all proteins over 20,000 atoms in size—importantly, note that this is after the integer compression/precision reduction, so this is an apples-to-apples comparison. This is seen visually in figure 4, as all except two points belonging to the two proteins with the fewest number of atoms lie above the diagonal, the region where PIC has better compression ratio performance. In figure 5, the gZip compression ratio decays while PIC's compression ratio increases with atom count.

Furthermore, unlike most compression algorithms, we can visually inspect the transformed image because it is itself a projection mapping of the original 3D structure. In figures 2 and 3, we show the PIC outputted images. For easier viewing, these images are inverted, five-fold contrast enhanced versions of the actual images outputted by the PIC compressor.

These results were obtained by running the *PIC.py* script in the command terminal with the "-e" option. The experiments were ran on an Apple MacBook Pro with a 3.5 GHz dual-core processor and 16 GB of memory.

4 Discussion

As expected, as atom count increases, more images are populated by PIC and more of the image space of the constructed images is used. In addition to the increased data load in only a slightly larger image width wise, this is due to an increased number of collisions as atom count increases. This causes the use of pointers, increasing the average number of bits used per data point, and, when no alternate location can be found in the current image, the population of a new image, increasing the image count.

The lower compression and decompression times in table 2 compared to those in table 1 show that PIC is fast at the compression and decompression of the 3D Cartesian coordinate point clouds. Majority of the increases in the compression and decompression times as reported in 2 are due to manipulations and carry through of the metadata to the decompressed file. This includes matching up decompressed atom coordinates with the corresponding atom metadata and reordering atom data to the proper locations in the file amongst other data such as header

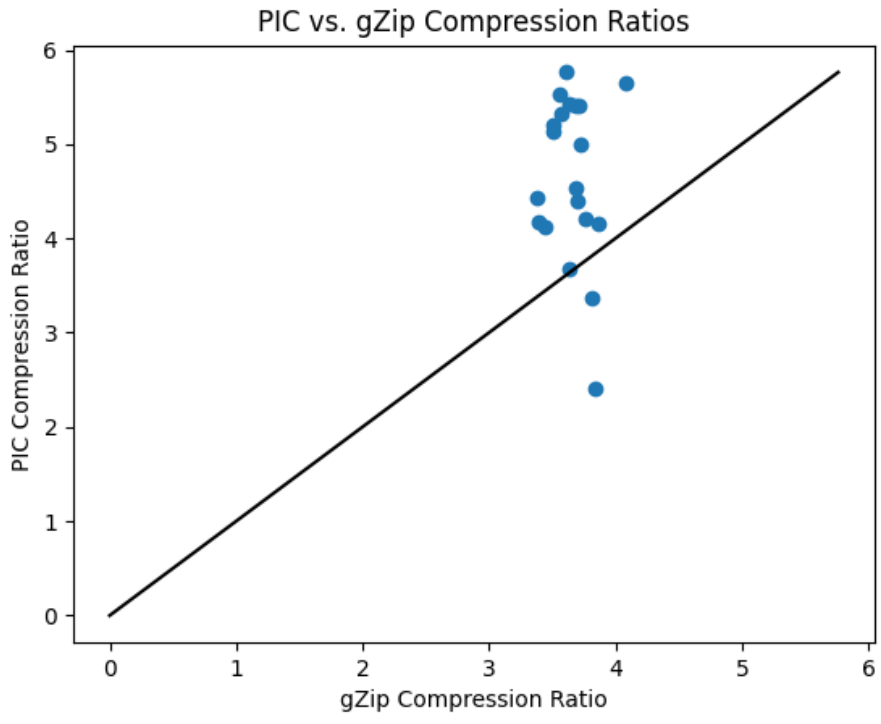


Fig. 4: PIC compression ratios plotted against gZip compression ratios for each protein compressed in table 1. Points in the region above the diagonal indicates a protein with better compression ratios using PIC than gZip. Vice versa below the diagonal.

Fig. 5: PIC and gZip compression ratios for the proteins compressed in table 1 plotted against the number of atoms that make up the compressed protein.

information. Thus PIC is an efficient compression algorithm for structural protein data without arbitrarily specified ordering.

Furthermore, as noted in the respective figures, our prototype PIC implementation is not optimized for speed. It is not intended as a drop-in replacement for gZip, but is instead meant to show that image-centric compression of protein atomic point clouds can provide significant space savings. The Python implementation takes on the order of minutes for a single compression/decompression, which is significantly slower than the order of seconds for gZip compression.

Other values of ε investigated include $\{1.25, 5, 10\}$. Only the results for $\varepsilon = 2.5$ are show as this value produced the best compression ratios. As stated in section 2.1, higher ε increased image sizes and consequently decreased the compression ratios. Setting $\varepsilon = 1.25$ increased compression times as collisions increased due to the decreased image size and alternate mapping locations needed to be considered. Compression ratios also decreased slightly as gains in the compression ratios from decreased image sizes were overcame by the additional use of pointers and higher number of images generated by the PIC algorithm. Importantly, in this prototype study, we have given results from only a single set of parameters for all sizes of proteins for principled benchmarking. In future work, it may be preferable to set parameters dynamically for each protein and to store them, as in standard practice in many file compression formats.

This is promising as the structures of more complex proteins are deconstructed, stored in databases, and transmitted amongst researchers. Further, as the PIC algorithm leverages the standard and widely used PNG image compressor, the algorithm can be easily implemented on a variety of platforms and systems.

5 Conclusion

In this paper, we have introduced PIC, an new compression algorithm that leverages positional encoding techniques and the well-developed, widely available PNG image compressor to encode and compress structural protein data in PDB and mmCIF files. The algorithm encodes two of the three dimensions of an atomic coordinate from the point cloud stored in the file to a position in the image space and stores the remaining dimension in pixels' intensity values around that location. The resulting image is then compressed with the lossless image compressor PNG. We showed PIC has a compression ratio superior of that of gZip for proteins with more than 20,000 atoms, and improves with the size of the protein being compressed, reaching up to 37.4% on the proteins we examined.

More important than just providing a prototype, we demonstrate in this paper that the paradigm of image-centric compression is superior in efficacy than simply applying a standard sequential compressor to the atomic point clouds. This result is consistent with both point-cloud compression in LIDAR imaging and read-reordering for NGS sequence compression. Importantly, this improvement in compression ratios persists even though we store the necessary metadata to undo any atom-reorderings; still, we would recommend that the ordering information be entirely discarded, as it is for LIDAR and read re-ordering—we only kept all of that information to ensure that we performed a fair comparison in our benchmarks.

Ultimately, we hope that this study points the way for future image-centric compression of protein structures. The PIC algorithm itself, if reimplemented in a faster language, is certainly competitive on compression ratios already, and furthermore is easy to implement because the PNG image format is already implemented on many platforms. As structural protein files with increasing complexity are deconstructed, added to databases, and transmitted amongst researchers, targeted compression techniques will become ever more necessary.

Acknowledgements

We would like to thank Jim Shaw, Ziye Tao, and Alex Leighton for their thought-provoking questions that inspired parts of this work.

Funding

This work was supported by the Ontario Graduate Scholarship Program and startup funds from the Computer and Mathematical Sciences department at the University of Toronto at Scarborough.

References

Alakuijala, J., Farruggia, A., Ferragina, P., Kliuchnikov, E., Obryk, R., Szabadka, Z., and Vandevenne, L. (2018). Brotli: A general-purpose data compressor. *ACM Transactions on Information Systems (TOIS)*, **37**(1), 1–30.

Berman, H. M., Kleywegt, G. J., Nakamura, H., and Markley, J. L. (2012). The protein data bank at 40: reflecting on the past to prepare for the future. *Structure*, **20**(3), 391–396.

Cox, A. J., Bauer, M. J., Jakobi, T., and Rosone, G. (2012). Large-scale compression of genomic sequence databases with the burrows–wheeler transform. *Bioinformatics*, **28**(11), 1415–1419.

Daniels, N. M., Gallant, A., Peng, J., Cowen, L. J., Baym, M., and Berger, B. (2013). Compressive genomics for protein databases. *Bioinformatics*, **29**(13), i283–i290.

Deorowicz, S., Walczyszyn, J., and Debudaj-Grabysz, A. (2018). CoMSA: compression of protein multiple sequence alignment files. *Bioinformatics*, **35**(2), 227–234.

Deutsch, P. *et al.* (1996). Gzip file format specification version 4.3.

Divne, C. (2006). 2ign: Crystal structure of recombinant pyranose 2-oxidase h167a mutant.

Fritz, M. H.-Y., Leinonen, R., Cochrane, G., and Birney, E. (2011). Efficient storage of high throughput dna sequencing data using reference-based compression. *Genome research*, **21**(5), 734–740.

Goodsell, D. S. (n.d.). Pdb101: Learn: Guide to understanding pdb data: Introduction to pdb data.

Hach, F., Numanagić, I., Alkan, C., and Sahinalp, S. C. (2012). Scalce: boosting sequence compression algorithms using locally consistent encoding. *Bioinformatics*, **28**(23), 3051–3057.

Hategan, A. and Tabus, I. (2004). Protein is compressible. In *Proceedings of the 6th Nordic Signal Processing Symposium, 2004. NORSIG 2004.*, pages 192–195. IEEE.

Hernaez, M., Pavlichin, D., Weissman, T., and Ochoa, I. (2019). Genomic data compression. *Annual Review of Biomedical Data Science*, **2**, 19–37.

Houshiar, H. and Nüchter, A. (2015). 3d point cloud compression using conventional image compression for efficient data transmission. In *2015 XXV International Conference on Information, Communication and Automation Technologies (ICAT)*, pages 1–8.

Ilari, A. and Savino, C. (2008). Protein structure determination by x-ray crystallography. *Bioinformatics*, pages 63–87.

Kato, K., Zhou, Y., Tanaka, H., Yao, M., Yamashita, M., and Tsukihara, T. (2014). 4v60: The structure of rat liver vault at 3.5 angstrom resolution.

Kryukov, K., Ueda, M. T., Nakagawa, S., and Imanishi, T. (2019). Nucleotide Archival Format (NAF) enables efficient lossless reference-free compression of DNA sequences. *Bioinformatics*, **35**(19), 3826–3828.

Patro, R. and Kingsford, C. (2015). Data-dependent bucketing improves reference-free compression of sequencing reads. *Bioinformatics*, **31**(17), 2770–2777.

Pearson, W. R. (1994). Using the fasta program to search protein and dna sequence databases. In *Computer Analysis of Sequence Data*, pages 307–331. Springer.

Pinho, A. J. and Pratas, D. (2013). MFCompress: a compression tool for FASTA and multi-FASTA data. *Bioinformatics*, **30**(1), 117–118.

Ramachandran, G. (1963). Protein structure and crystallography. *Science*, pages 288–291.

Rose, P. W., Prlić, A., Altunkaya, A., Bi, C., Bradley, A. R., Christie, C. H., Costanzo, L. D., Duarte, J. M., Dutta, S., Feng, Z., *et al.* (2016). The rcsb protein data bank: integrative view of protein, gene and 3d structural information. *Nucleic acids research*, page gkw1000.

Valasatava, Y., Bradley, A. R., Rose, A. S., Duarte, J. M., Prlić, A., and Rose, P. W. (2017). Towards an efficient compression of 3d coordinates of macromolecular structures. *PLOS ONE*, **12**(3), 1–15.

Westbrook, J. D. and Fitzgerald, P. M. (2003). The pdb format, mmcif formats, and other data formats. *Structural bioinformatics*, **44**, 159–179.

Yu, Y. W., Yorukoglu, D., Peng, J., and Berger, B. (2015). Quality score compression improves genotyping accuracy. *Nature biotechnology*, **33**(3), 240–243.