Tools for Evaluating the Robustness of a Data Format

John K. Johnstone[1]

[1]Computer Science and Ophthalmology, The University of Alabama at Birmingham, jkj@uab.edu

Corresponding author: John K. Johnstone, jkj@uab.edu

Abstract. This paper develops tests to evaluate the robustness of the read/write machinery of a data format: is the information content of the data preserved as it is transformed between external data files and internal data structures? It also develops tests to evaluate the robustness of the translation machinery between two data formats: is the information content of the data preserved as it is translated between different formats? A case study of a pair of data formats from optic morphometry is explored. It was the development of a new format for this complex data that prompted the exploration of this paper.

Keywords: data management, data format, reader, writer, CAD dataset, translation
DOI: https://doi.org/10.14733/cadaps.2025.882-892

1 INTRODUCTION

This paper considers mechanisms to evaluate the correctness of a new data format for data storage and its read-write machinery, as well as to guarantee the preservation of information content when translating between two data formats. This issue has increasing importance in today's world of large datasets, where the corruption of data has large consequences for computation. Formal mechanisms for data management and provenance are also increasingly being emphasized by the NIH and NSF [9, 10], illustrating the importance of this issue. The focus is on CAD datasets such as polygon meshes and point clouds, but the mechanisms generalize to other datasets. A major purpose of the paper is straightforward: to highlight the issue of robustness and verification of data. We also consider a case study of the use of these tests in the development of a new data format and its read/write machinery, and the incorporation of these tests into the software lifecycle.

These issues were highlighted for us recently in developing a new data format for optic nerve head point clouds in a biomedical context (ophthalmology), where there are high expectations for the sanctity and provenance of the underlying data and where the datasets are large and complex [6].

Consider the storage of data in a certain data format and its read/write machinery. Data is stored externally in a data file using a data format, and it is stored internally, in code, in a data structure. This symbiotic relationship between the internal and external representations is subtle, yet the preservation of information content in the data is obviously crucial as it is translated between internal and external representations, and between different data formats. There are many moving parts that must be coordinated here: the data format, the internal data structure, the reader, the writer, and any translators between different data formats. This issue becomes most tangible when developing and introducing a new data format: can the data structure be restored from the new data format? will the information content of the data stored in previous formats be preserved when translated to the new format?

The sanctity of data, and the importance of preserving its information content as it is massaged into different forms, demands that this issue be treated more deliberately and more formally. We introduce several tests that can be applied to guarantee that no information is lost in translating between internal and external representations of data, and between different external representations of data.

1.1 An Illustrative Example

As a concrete example, consider a triangle mesh as the dataset of interest. There are many internal representations of a triangle mesh (e.g., halfedge, winged edge, list of vertices/faces) and there are many external representations of a triangle mesh (e.g., OFF, PLY, OBJ formats [12, 16, 17]). Choose one of the internal representations of a triangle mesh (say, halfedge) and let I be the universe of valid internal half-edge representations of all triangle meshes. Choose one of the external representations of a triangle mesh (say, PLY) and let E be the universe of valid external PLY representations of all triangle meshes. A (halfedge, PLY) writer $w : I \rightarrow E$ is code whose input is a halfedge internal representation of a triangle mesh and whose output is a PLY external representation for a triangle mesh. A (PLY, halfedge) reader $r : E \rightarrow I$ is code whose input is a PLY external representation file for a triangle mesh and whose output is a halfedge internal representation of a triangle mesh. Note that there are three hyper-parameters to a reader/writer: the structure of interest (in this case, triangle mesh), the internal representation of this structure (half-edge), and the external representation of this structure (PLY). We are interested in the correctness of the translation from internal to external representation and back.

We are also interested in the correctness of the translation between external representations in two different data formats. This adds a fourth hyper-parameter: the second external representation of this structure (e.g., OBJ). Choose a second external representation of a triangle mesh (OBJ) and let E_2 be the universe of valid external OBJ representations of all triangle meshes. A (PLY, OBJ) translator $t : E \rightarrow E_2$ is code whose input is a PLY file for a triangle mesh and whose output is an OBJ file for the same triangle mesh. A (OBJ, PLY) translator $t_2 : E_2 \rightarrow E$ is code whose input is an OBJ file for a triangle mesh and whose output is a PLY file for the same triangle mesh. We also want to test the integrity of these translators.

The rest of the paper is structured as follows. Section 2 discusses internal tests, which focus on the preservation of data in its internal form. Section 3 discusses external tests, which focus on the preservation of data in its external form. Section 4 discusses translation tests, which focus on the preservation of data while translating between two different data formats. Section 5 discusses a case study for these tests, an optic nerve head point cloud used in morphometry, including implementation details. Section 6 ends with conclusions.

2 INTERNAL TEST

An internal test is a fundamental test of a data format and its read/write machinery, since it reflects a fundamental purpose of a data file: as a temporary repository for the internal data structure. An external data file allows a computation to be restarted after a temporal break; an external data file also allows the underlying structure to be communicated to another processor so that it can continue the computation. So the file format records a snapshot of the meaningful elements of the data structure x, allowing x at time/place t/p to be restored as x at time/place t+delta/p or time/place t+delta/q. The basic question that must be answered in the affirmative is: if you store a data structure x for a while in a data file f, can you restore x from f?

Definition 1
Consider a dataset of interest (e.g., a triangle mesh or a point cloud).
Fix an internal data structure for this dataset (e.g., half-edge representation of a mesh).
Fix a data format for this dataset (e.g., OBJ representation of a mesh).
Let w be a writer, which translates an internal data structure to a data file in the data format.
Let r be a reader, which translates a data file in the data format to an internal data structure.
The data format's read/write machinery passes the ***internal test*** *on internal data x if*

$$r(w(x)) = x$$

There are at least three sources for the internal data x. Most purely, x may be built directly in the code, using some algorithm to build x. For example, using a graph-cut algorithm [14] to segment an image to generate a point cloud x, if the dataset of interest is a point cloud; or using Poisson reconstruction [7] of a point cloud to generate a triangle mesh x, if the dataset of interest is a triangle mesh; or using a cubic fitting algorithm [13] to generate a B-spline curve from a point cloud, if the dataset of interest is a B-spline curve.

x may also be built randomly with the correct structure. This is a good way to stress-test the quartet of reader, writer, data format, and data structure, since randomness helps to expose weakness.

Another simple way to build x is by reading a file f: $x = r(f)$.

Finally, note that there is an equality test at the heart of the internal test. When the components of the internal data are floating point, only approximate equality is expected, given finite precision. This issue is discussed in Section 5.3.

3 EXTERNAL TEST

An internal test is possible after creating an instance of the internal data structure. Therefore, it is natural to apply as one writes the structure out to a data file, to guarantee that the write is robust. But if you instead start with a data file, a different test may be more natural. The external test is symmetric to the internal test, starting from the external data file rather than from the internal data structure. It tests whether you can read a data file and then later restore the same data file by writing. Since whitespace differences in plain-text data files are irrelevant, the external test ignores these differences.

Definition 2
Consider a dataset of interest.
Fix an internal data structure for this dataset.
Fix a data format for this dataset.
Let w be a writer, which translates an internal data structure to a data file in the data format.
Let r be a reader, which translates a data file in the data format to an internal data structure.
Let squish be a function that strips extra whitespace from each line of a file.
The data format's read/write machinery passes the ***external test*** *on binary data file f if*

$$w(r(f)) = f$$

The data format's read/write machinery passes the ***external test*** *on plain-text data file f if*

$$\mathit{squish}(w(r(f))) = \mathit{squish}(f)$$

4 TRANSLATION TEST

Internal and external tests measure the robustness of the read/write machinery of a single format. Translation tests measure the robustness of a new format, by measuring its robustness in preserving the information in an existing format.

It might seem that the robustness of the translation between two formats is implied by the robustness of the read/write machinery of each format. After all, if you can read format 1 robustly into the internal data structure, and write format 2 robustly from the internal data structure, this seems to imply that you can translate from format 1 to format 2, using this reader/writer pair. Similarly the dual reader/writer pair (reading format 2 and writing format 1) would seem to establish the robustness of translation from format 2 to format 1.

The subtlety is that one format may contain more information than another, and this is often the case. For example, both PLY [15, 16] and OBJ formats [17] for polygon meshes offer the opportunity for much more information to be stored about a mesh than the OFF format [12]. The OFF format focuses on edges, faces, and optionally edges, while the OBJ and PLY formats also allow colour, texture, and other information to be encoded. This is also the case with the new format for optic nerve head point clouds that we have developed: it records only a subset of the data in an earlier format, because some of this original data has become vestigial and is no longer used in morphometric algorithms (Section 5).

This poses a different robustness issue than the original reader/writer pair robustness measured by internal and external tests. Now one is interested in whether the subset of common information is preserved. In the PLY/OFF case, this would be the preservation of vertex/face/edge information.

Definition 3
Let $\mathcal{F}$ and $\mathcal{G}$ be two data formats for encoding the same dataset $\mathcal{X}$.[1]
Let X be the universe of datasets that are representable in both formats.[2]
Let F be the universe of valid representations of X in the first format $\mathcal{F}$.
Let G be the universe of valid representations of X in the second format $\mathcal{G}$.
Let tog : $F \rightarrow G$ be a translator of a data file in format $\mathcal{F}$ to the equivalent data file in format $\mathcal{G}$.
Let tof : $G \rightarrow F$ be a translator of a data file in format $\mathcal{G}$ to the equivalent data file in format $\mathcal{F}$.
Let squish be a function that strips extra whitespace from each line of a file.
The data formats' translation machinery passes the ***translation test*** *on the pair of files f/g of the same dataset if:*

$$\textit{squish}(\textit{tof}(\textit{tog}(f))) = \textit{squish}(f)$$

and

$$\textit{squish}(\textit{tog}(\textit{tof}(g)) = \textit{squish}(g)$$

This translation test should be applied as each dataset is translated from one format to another. We note that consideration of the translation issue, as formalized in translation tests, also pushes the designer of a data format to consider the coverage of a data format more thoughtfully, since potential gaps in the new format become more visible in identifying the universe of datasets representable both in the new format and earlier formats.

Having defined our three classes of test – internal, external, and translation – we now consider the implementation of these tests through a case study.

5 CASE STUDY

As a case study for these tests, consider the read/write/translation machinery of an **ONH dataset**, a dataset from ophthalmology used in morphometry of the optic nerve head [4, 5]. An ONH dataset is a point cloud sampled from the optic nerve head, organized into categories and slices. Fig. 1 is an example. Categories are

[1]For example, $\mathcal{F}$ and $\mathcal{G}$ might be PLY and OFF formats for storing a mesh.

[2]For example, X might be polygon meshes that only contain pure geometric information about the vertex/edge/face of the mesh, without extra information such as texture coordinates or colours, so that all the information about the mesh in the PLY file is also representable in OFF format.

the anatomical structures of interest in the optic nerve head, such as the inner limiting membrane and anterior lamina (red and green, respectively). Slices are the planar slices through the optic nerve head, an artifact of the point cloud's acquisition by segmentation of images sliced out of a volume gathered using spectral-domain optical coherence tomography [8].

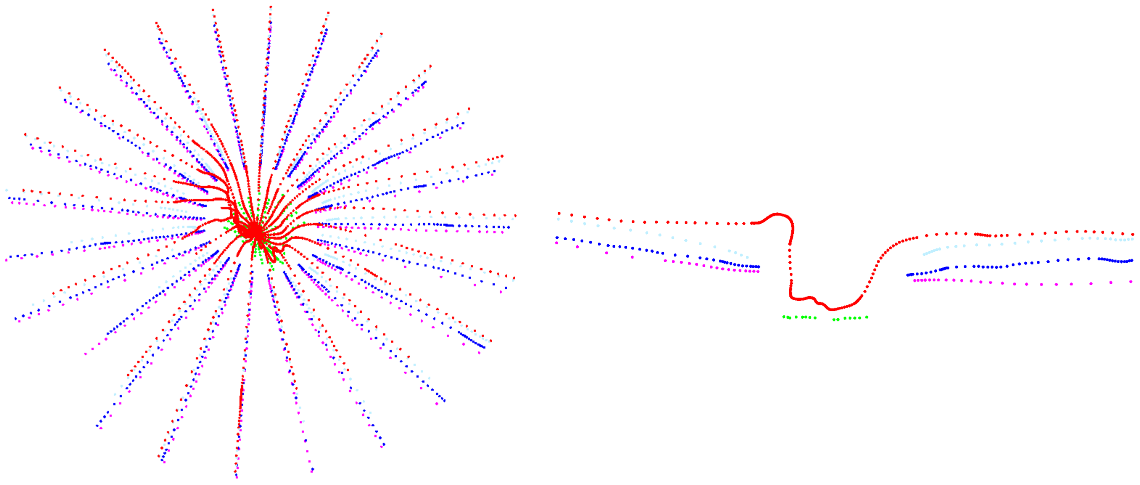
Figure 1: An ONH dataset. Left: The entire dataset, showing its organization into slices. Right: A single slice, showing various categories.

We will consider two data formats for the ONH dataset, so that we can consider translation tests. A major difference between the two formats is the order of the point cloud. Since an ONH dataset (Fig. 1) consists of categories and slices, the storage of its point cloud may be organized either by category or by slice. In **slice-major order**, the point cloud is stored slice by slice, then within each slice, category by category. In **category-major order**, the point cloud is stored category by category, then within each category, slice by slice. In either slice-major or category-major order, the points of a slice's category (or a category's slice) are sorted along the slice. The present data format stores the point cloud in slice-major order. However, category-major order is more natural for optic morphometry computations, which are centered around individual categories. This is one of the motivations for introducing a new data format, which stores the point cloud in category-major order. The two formats are outlined in detail in [6]. We only give a short definition of each format here so that we can discuss issues that arise in implementing the read/write/translation machinery.

In the old format, each line of the file represents a point and has 6 entries: the Cartesian coordinates of the point, the category code of the point, the 0-based slice index of the point, and a Boolean mark. The file also begins with a two-line comment that defines a special point. The Boolean marks and the special point are vestigial elements that are ignored by all morphometric computations. Therefore, they are both left out of the new format.

The new format adopts the Stanford PLY format [16], a general purpose format for geometric datasets.[3] Table 1 outlines the PLY elements of the format, and their properties. The **category element** represents the categories in the point cloud. It has one property (code), which is a list of category codes. This element specifies the order of the categories in the category-major point cloud, which is also the order of the categories in the section element. The **vertex element** represents the entire point cloud in category-major order. It has three properties x, y, and z, representing the Cartesian coordinates. A **section** is the point cloud of a certain

[3]PLY format was originally built for polygon meshes, but is readily adaptable to point clouds.

category in a certain slice. The **section element** represents the sections of a single category. The section element has one property (start), which is a list of the start indices of the sections of the category in question.[4] The **precision element** represents the number of digits recorded after the decimal point in point cloud floats. (The need to record precision will be discussed further below.) Finally, comments in the PLY header give the category names associated with the category codes in the dataset.[5]

Element	Property(s)	Type
category	code	integer list
vertex	x, y, z	double, double, double
section	start	integer list
precision	p	integer

Table 1: The elements and properties of an ONH dataset in PLY format

5.1 Handling Deviations Between Two Formats

The ONH dataset illustrates the issue of two formats containing different information, which complicates translation tests. The new format for ONH datasets does not include some unnecessary data (slice indices) and some vestigial data (Boolean marks and the special point). As mentioned in Section 4, a format may also contain extra useful data: OBJ files for a triangle mesh may contain texture coordinates and vertex normals, data that cannot be recorded in an OFF file, which only stores vertices, faces, and perhaps edges. Both situations complicate translation tests.

Consider two formats for the same structure, one of which has extra information. We will call them the thin format and the fat format (the one with extra data). There are two options to implement a translation test: strip the extra information out of the fat format, or add the extra information to the thin format.

First consider two formats that one does not control, such as OBJ and OFF format for polygon meshes. Since it is impossible to add information to the thin format (OFF), the only option is to strip the extra information from the fat format (OBJ), either by finding files without the extra information (e.g., no texture coordinates) or actively stripping the extra information out of the datasets before applying the translation test.

There is more flexibility if you control the thin format, perhaps because you are designing it, as with the new ONH format. Here, an option can be offered to add the extra information to the new format. This extra information is only used in the translation tests. It is suggested to make the extra information trivially removable from the format, so that its removal is not a source of error.

Consider the fat format for ONH datasets. It is impossible to strip out the slice indices, since they are required to correctly interpret the data. Fortunately this data is easily reconstructed during translation, since it is implied by the new format, so it does not need to be added to the new format. However, the vestigial data cannot be reconstructed from the new format's data, so it is necessary to add the option to add these vestigial elements to the new format, to allow a translation test to pass. Here is how we implemented this for the new format.

First consider our internal data structure for the new format. Note how the data structure mirrors closely the data format. A close relationship between the internal and external forms of the data is a goal to be aimed for, adding clarity and transparency. The class has the following member variables:

[4]Since the sections of a category are contiguous, this suffices to define the section. The use of a single integer to represent each section is another advantage of the new format.

[5]Since PLY does not have a string type, the best compromise is to add this information in structured comments.

1. the point cloud of the entire optic nerve head, which is stored in a column-major data matrix[6] of shape (3, nPt), with the columns/points sorted by category, then by slice within each category
2. an integer vector of the category codes, in order of their appearance in the point cloud[7]
3. an (int, string) associative map between category code and category name, documenting the integer category codes
4. the number of slices per category, an integer
5. a data matrix of shape (2, nCat * nSlice), whose $j * \text{nSlice} + i$th column stores the start and end indices of the ith section of the jth category (indexing into the global point cloud)[8]
6. the precision of the point cloud's floats, to be used in writing

To handle the optional vestigial elements of the fat format, a child class has the following additional member variables:

7. a vector of Boolean marks, one per point in the point cloud
8. a special point (to be inserted into the leading comment)
9. increment between consecutive slice indices (since this increment is consistent, but not necessarily 1)

Along with the internal data structure, the external format is also adapted: an optional vestigial PLY element, with two properties: a slice increment, and a special point; and an optional additional property for the vertex element (a Boolean mark). These elements and properties are only used in translation tests. During translation of the old format to the new format, vestigial elements can optionally be added in the new format.

5.2 Order of Testing

Now consider the order in which to perform the three batches of tests. To understand this issue, again consider ONH datasets as an example. There is presently no algorithm that directly builds the internal data structure of ONH datasets: the underlying image segmentation task to build the point clouds from planar slices of the OCT volume is too challenging, so it is replaced by manual segmentation by a domain expert. For other datasets, even if an algorithm that builds the internal data structure is available, it may be tedious to first implement this algorithm. For example, if the data is a triangle mesh, building a triangle mesh from a point cloud using Poisson reconstruction [7] may be a challenge one wants to avoid, especially as one concentrates on read/write machinery.

Therefore, often the only two ways to build an internal data structure are by reading a data file or by randomly generating the data structure. Reading an existing data file is the more natural option, since it works with real data, which is unlikely to be random. Randomized data can be reserved for later stress tests.

This makes external tests, starting with a file, the natural first test. In particular, starting with a battery of external tests on data files, the read/write machinery can be refined. The stable reader can then be used to populate the internal data structure, and this can be used to conduct a battery of internal tests.

Now consider the addition of translation tests. There are two cases to consider here: when both data formats are established (e.g., OBJ and PLY for a triangle mesh), and when one of the data formats is new (e.g., our new format for ONH datasets). When both formats are established, a natural order for the tests is

[6]The numerical library Eigen [1] is used to store matrices and vectors.
[7]Different datasets may include different categories; categories are identified by integer codes.
[8]The end index is added, although redundant, for simplicity of future computation.

as follows: external tests of format 1, internal tests of format 1, external tests of format 2, internal tests of format 2, translation tests between format 1 and format 2. Translation tests are delayed until confidence has been established in the two formats individually.

However, if the second format is being developed anew, the order may be altered, since data files for the new format are not readily available at the beginning: external/internal tests of the established format 1, translation tests between format 1 and the new format 2, a pause to construct datasets in format 2, then external/internal tests of format 2. This is a bit subtle, since confidence in the read/write machinery of the new format is low during the translation tests. Therefore, we have actually found it to be a good idea to make a small battery of translation tests on a few data sets, in order to populate some files in the new format, then debug the read/write machinery of the new format on these data sets using external and internal tests, finally reverting to a full battery of translation tests, and a full battery of external/internal tests of the new format.

5.3 Testing Equality

At the heart of each of the internal/external/translation tests is a test of equality, which requires care for floating point values. You should never ask if two floats are exactly equal, and you need to write floats to a consistent precision, otherwise one can never expect equivalent files. We now consider these two issues.

5.3.1 Equality in an External Test

Testing equality of floats in an external data file, during an external or translation test, requires care in the precision with which a float is written. The **precision** of a float is the number of digits recorded after its decimal point. For example, the precision of the float 4745.190 is 3. Writers need to be consistent in, and aware of, the precision with which they output floats, otherwise the equality test of an external or translation test cannot be expected to succeed. Datasets should be consistent in the precision with which floats are recorded. If a dataset does not have consistent precision of its floats, it can easily be converted into one that does, by computing the maximum precision of a float in the dataset, then padding floats below this precision with trailing zeroes.

Consider this issue for ONH datasets. The new ONH format records precision in the data format (in the precision element). Precision is also recorded in the internal data structure. Then this precision is used when writing the dataset. Since the old ONH format does not record precision, it must be detected: as a file in the old format is read, the precision of each float is detected and the maximum precision is maintained; this precision is then transferred to the new format as the old format is translated to the new format.

5.3.2 Equality in an Internal Test

Testing equality of floats in an internal data structure, during an internal test, also requires care. Since floating point storage and arithmetic are inexact, one should not ask if two floats are exactly equal. Instead, one should ask for equality within a tolerance. Therefore, the goal is a test for equality of two floats within a tolerance, ideally symmetric in the floats, and ideally vectorizable, so that we can efficiently apply the test to a matrix, such as the data matrix associated with a point cloud. This type of equality test is missing from C++ and C++ libraries like Eigen [1].[9] We now consider some options for this equality test.

Consider a real number x and its floating point representation $\mathrm{fl}(x)$. We know that

$$\mathrm{fl}(x) = x(1+\delta)$$

[9]Eigen's matrix comparison function *isApprox* is not element-wise - it is a global comparison based on Frobenius norm - and therefore not of interest for us.

where $|\delta| \le u$ and u is the unit roundoff (half the size of the gap between 1 and the nearest floating point number, or half of machine epsilon) [2, Section 2.7]. So the error in storing x is

$$|\mathsf{fl}(x) - x| = |\delta||x| \le u|x|$$

The key insight is the influence of the size of the scalar on the expected error. Since fl(x) lies in a ball of radius $u|x|$ about x, the floating point representations of two equal scalars could lie anywhere in this ball. Therefore, the idealistic equality test $|\mathsf{fl}(x) - \mathsf{fl}(y)| = 0$ is replaced by the equality test

$$|\mathsf{fl}(x) - \mathsf{fl}(y)| \le 2u|x|$$

To make the test symmetric in x and y, this is changed to

$$|\mathsf{fl}(x) - \mathsf{fl}(y)| \le u(|x| + |y|)$$

and, since x and y are unavailable, we use their best proxies, fl(x) and fl(y):

$$|\mathsf{fl}(x) - \mathsf{fl}(y)| \le u(|\mathsf{fl}(x)| + |\mathsf{fl}(y)|)$$

But accounting for very small x, y (where $u(|x| + |y|)$ could be smaller than u), we adjust this to

$$|\mathsf{fl}(x) - \mathsf{fl}(y)| \le \max(u, u(|\mathsf{fl}(x)| + |\mathsf{fl}(y)|))$$

which sets the most aggressive threshold to unit roundoff. Finally, the unit roundoff u may be softened to $\epsilon > u$ to allow a softer test of approximate equality (say after accumulated error). This leads to the following equality test for doubles,

$$|\mathsf{fl}(x) - \mathsf{fl}(y)| \le \max(\epsilon, \epsilon * (|\mathsf{fl}(x)| + |\mathsf{fl}(y)|)) \Rightarrow x = y$$

where $\epsilon > u$. C++'s std::numeric_limits<Type>::epsilon may be useful in choosing ϵ.

An alternative equality test, inspired by NumPy's allclose function [11], uses two tolerances: an absolute tolerance *atol* and a relative tolerance *rtol*. Two scalars x and y are considered equal if:

$$|\mathsf{fl}(x) - \mathsf{fl}(y)| \le \mathsf{atol} + \mathsf{rtol} * |\mathsf{fl}(y)|$$

The relative tolerance is similar to the earlier test's ϵ, while the absolute tolerance handles small scalars. To restore symmetry, this can be replaced by

$$|\mathsf{fl}(x) - \mathsf{fl}(y)| \le \mathsf{atol} + \mathsf{rtol} * (|\mathsf{fl}(x)| + |\mathsf{fl}(y)|) \Rightarrow x = y$$

5.4 C++ Implementation

All of the tests have been implemented in C++ for both of the formats of our case study. Because of bugs in software and flaws in design, failing the tests is common in early work. For example, failure to detect precision of floats and preserve consistent precision in writing floats and in the data formats led to early failure of external and translation tests, and redesign of the internal data structures and data formats, even to the point of changing the open-source PLY format reader *happly* [3].

6 CONCLUSIONS

As data is handled by read/write machinery, and particularly as it is translated to a new format, the preservation of information content is crucial. The main contribution of this paper is a deliberate focus on the issue of robust testing of reader/writer pairs for a data format, and on robust testing of translators between data formats. Once this focus is applied deliberately and carefully, the path to these tests is natural, and can be applied universally in these environments.

We suggest adding the option to incorporate internal/external tests during the reading/writing of a dataset, as a conservative step to guarantee that no information is lost. It is also natural to add translation tests during translation of datasets between formats.

The tests of this paper contribute to the debugging phase of software development of readers, writers, and translators. We have also found that use of these tests during design of a new data format can help guide and refine the definition of the format, as it exposes differences with old formats during the development of translation tests, and emphasizes the symbiotic relationship between the internal and external forms of the data, such as influencing the format to mirror the desired internal representation, and vice versa. These tests can encourage the development of an optimal data format, if used thoughtfully.

Future work includes adoption and application of these IO tests in a wider spectrum of applications. In particular, we are interested in parametric curve datasets and formats, where there is little work.

ACKNOWLEDGEMENTS

We appreciate discussions with colleagues in UAB's Department of Ophthalmology about optic morphometry, and access to the ONH datasets used in this study.

John K. Johnstone, https://orcid.org/0000-0003-4033-0066

REFERENCES

[1] Eigen: https://eigen.tuxfamily.org

[2] Golub, G., Van Loan, C.: Matrix Computations. The Johns Hopkins University Press (Baltimore), 2013. https://doi.org/10.56021/9781421407944

[3] hapPLY: a C++ header-only parser for the PLY file format. https://github.com/nmwsharp/happly

[4] Johnstone, J.; Fazio, M.; Rojananuangnit, K.; Smith, B.; Clark, M.; Downs, J.C.; Owsley, C.; Girard, M.; Mari, J.-M.; Girkin, C.: Variation of the Axial Location of Bruch's Membrane Opening with Age, Choroidal Thickness and Race. Investigative Ophthalmology and Visual Science, 55(3), 2014, 2004–2009. https://doi.org/10.1167/iovs.13-12937

[5] Johnstone, J.; Rhodes, L.; Fazio, M.; Smith, B.; Wang, L.; Downs, J.C.; Owsley, C.; Girkin, C.: Measuring Mean Cup Depth in the Optic Nerve Head. Computer Aided Design and Applications, 13(5), 2016, 693–700. https://doi.org/10.1080/16864360.2016.1150716

[6] Johnstone, J.: A category-major interoperable data format for optic morphometry. CAD 2024 (Eger, Hungary), 2024, 121–125. https://doi.org/10.14733/cadconfP.2024.121-125

[7] Kazhdan, M., Bolitho, M., Hoppe, H.: Poisson Surface Reconstruction. Eurographics Symposium on Geometry Processing 2006.

[8] Lee, E.; Kim, T.; Weinreb, R.; Park, K.; Kim, S.; Kim, D.: Visualization of the lamina cribrosa using enhanced depth imaging spectral-domain optical coherence tomography. Am J Ophthalmol, 152(1), 2011, 87–95. https://doi.org/10.1016/j.ajo.2011.01.024

[9] 2023 NIH Data Management and Sharing Policy: https://grants.nih.gov/grants/guide/notice-files/NOT-OD-21-013.html

[10] NSF data management and sharing requirements:
https://new.nsf.gov/policies/pappg/23-1/ch-11-other-post-award-requirements#11D4

[11] numpy.allclose: https://numpy.org/doc/stable/reference/generated/numpy.allclose.html

[12] OFF format. https://en.wikipedia.org/wiki/OFF_(file_format)

[13] Piegl, L., Tiller, W.: The NURBS Book. Springer, 1997.
http://dx.doi.org/10.1007/978-3-642-59223-2

[14] Shi, J., Malik, J.: Normalized Cuts and Image Segmentation. IEEE Transactions on Pattern Analysis and Machine Intelligence, 22:8, 2000, 888-905. http://dx.doi.org/10.1109/34.868688

[15] The Stanford 3D Scanning Repository: https://graphics.stanford.edu/data/3Dscanrep

[16] Turk, G.: The PLY Polygon File Format: https://gamma.cs.unc.edu/POWERPLANT/papers/ply.pdf

[17] Wavefront OBJ format: https://en.wikipedia.org/wiki/Wavefront_.obj_file