BioNumerics Exploration: decoding tracer-style data.

# Objectives

To extract sequencing data or visual chromatograms from a BioNumerics-exported tab-delimited text file (SEQTRACEFILES\_TAB\_delimited.txt). This file contained embedded fields that appeared to hold base64-encoded trace data—presumed to represent .ab1 chromatogram image information. These images provide a visual representation of the sequencing traces, with an initial focus on generating separate images for each individual trace, with the view to later overlay or align the four nucleotide fluorescence channels (A, C, G, T) into a single composite trace-style image."

# 2. Summary of investigation

## 2.1. Extraction

* The BioNumerics export file is structured as a tab-delimited table, beginning with a set of variables (data columns): OBJACTIONID, KEY, CONTIGFILE, TRACEID, DATA, INFO

|  |  |
| --- | --- |
| Column | Description |
| OBJACTIONID | Always -1 in trace rows, indicating a generic operation or batch import |
| KEY | Set to NULL, signifying no unique key assigned |
| CONTIGFILE | Contains a contig identifier, e.g. CTG29095454216519702500001 |
| TRACEID | A trace identifier, e.g. 000001CTG29095454216519702500001 |
| DATA | Contains an embedded XML tag: <Data FileType="ABI" DataType="B64D">B64D... |
| INFO | Includes a <Tracefile> tag with all supporting metadata for the file |

## 2.2. Block Identification and Parsing

* The script reads the entire input text file and splits it into discrete trace blocks using custom delimiters.
* A new block begins when a line starts with the marker -1 NULL CTG, which signifies the start of a sequence trace entry.
* The block ends when the line contains </Tracefile>, signaling the conclusion of that particular record.
* Any content between these markers is grouped together into a **block**, representing a single sequencing record and its associated metadata and data payload.

## 2.3. Base64 Decoding

Within each parsed trace block, the script attempts to extract ABI-encoded binary data that is embedded as base64 text. This data is found inside a specific XML-like tag:

<Data FileType="ABI" DataType="B64D">B64D...base64 content...</Data>

The presence of B64D is key—it acts as a custom prefix identifying that the payload is base64-encoded binary ABI data. The script uses a regular expression to match this content:

<Data FileType="ABI" DataType="B64D">B64D(.\*?)</Data>

Once captured, the base64 string is sanitized by removing non-base64 characters and padded appropriately to ensure it's a valid, decodable string. The Python base64.b64decode() function is then used to convert this clean string into raw binary data.

* Challenge: Base64 fields included noise and weren’t always a multiple of 4, indicating that cleaning was required.
* Fix: Cleaned characters, added padding where needed, and logged truncations.
* Result: Four .ab1 binary files were created, two of which contained no usable data (Blocks 1 and 4) and fallback empty payloads were produced instead. This occurred because:
  + The associated base64 strings extracted from those blocks were **invalid**—specifically, their lengths were one character more than a multiple of four, which is not allowed by the base64 spec.
  + Even after cleanup and padding correction, decoding failed
  + As a fallback, the script wrote placeholder binary files (empty) so output structure remained intact. These were still used in downstream attempts at image rendering and embedded extraction, but both steps failed due to the lack of real data.

## 2.4. ABI Header Detection

* *Expectation*: ABI files should begin with *ABIF* (ASCII) i.e. 41 42 49 46 (hex).
* *Reality*: All decoded files did not begin with 41 42 49 46.
* *Conclusion*: These were not true .ab1 files.

## 2.5. Zlib Decompression Attempt

* Hypothesis: Payloads might be compressed ABI files.
* Action: Applied zlib.decompress() to decoded data.
* Outcome: All decompression attempts failed; fallback used original data.

## 2.6. Image Format Detection

* Theory: Base64 might encode a chromatogram-style image, not an .ab1 file.
* Action: Scanned decoded binary for embedded image signatures (PNG, JPEG, GIF, BMP).
* Result:
  + One image candidate found from Block 2.
  + However, the image does not open, so it’s likely false positives or corrupted.

## 2.7. Grayscale Visualization of Binary

* Approach: Treated decoded binary as raw pixel intensity and rendered it as grayscale images using NumPy and matplotlib.
* Result: Only two images could be produced (Block 2 and Block 3), but these were static/noise. No patterns, no chromatogram signal.
* Conclusion: Binary does not encode raw image matrices.

## 2.8. Metadata Extraction

* Success: Extracted TRACEID, FileName, FileType, Trim1, Trim2, UsedRegions, Alignment, for all 4 blocks.
* Output: Saved clean .tsv file summarizing trace metadata.

# 3. Deliverables Collected

* Clean metadata table: trace\_metadata.tsv
* Attempted raw decoded binary .bin files
* Attempted grayscale renderings
* Attempted embedded image scans (even if unreadable)
* Full script suite (Python 3.12)

# 4. Conclusions

* The base64 payloads do not contain standard .ab1 ABI electropherogram files.
* They also do not encode images in formats like PNG or JPEG.
* Most likely, the payload is a custom visual rendering object, viewable only inside BioNumerics.

# 5. Recommended Next Steps

* Open the project in BioNumerics and export trace files in .ab1, .png, or FASTA format to see if this is possible.
* Contact BioNumerics support to inquire about decoding or interpreting B64D fields externally.