Decoded AIS data multiple formats

Kiki Beumer

April 2025

1 Notebook Purpose and Workflow

This Jupyter notebook decodes *encoded* Automatic Identification System (AIS) sentences into a human-readable, tabular data set. The resulting dataset is used for interactive geospatial visualisation, but can be used for additional analysis as well. The main script

1. reads a log file whose path you provide in `file_path`;
2. determines whether each line is a *proprietary* timestamp sentence (e.g. `$PGHP`) or a payload sentence (`!AIVDM`, `!BSVDM`, `!AGVDM`, ...);
3. extracts or inherits the correct timestamp;
4. verifies the NMEA checksum with `verify_checksum`;
5. converts the six-bit ASCII payload to a binary string
6. and, following ITU R M.1371 bit layouts, decodes the desired fields (MMSI, navigation status, ROT, SOG, longitude, latitude etc.).

Timestamp formats recognized

- **ISO 8601 inline** (`2023-10-28T07:17:51.000Z`) used in `!BSVDM` or `!AIVDM` sentences.
- **NMEA 0183 Talker time** in the form `dd-mm-yyyy hh:mm:ss` (`1459720797`).
- **Proprietary $PGHP** format: `$PGHP,1,yyyy,mm,dd,HH,MM,SS,sss,*hh`.
- **Tag-Block** prefix ("`\s:`" or "`\T:`") that carries a Unix epoch and/ or human-readable stamp.
- **Multi-fragment** sentences where the timestamp is carried only in the first fragment.

Efficiency note

During testing the decoder parsed an 860 MB logfile in 3 min 1.9 s on standard hardware, while handling a 4 GB file led to slower performance and increased memory usage. The current performance is determined by the complexity limits summarized in the table.

$$\textit{Time complexity } \Theta(n), \qquad \textit{Space complexity } \Theta(n),$$

where n is the number of AIS sentences and the length of the sentence ($L \leq 82$ chars) is constant.

To improve scalability the notebook now limits payload decoding to message types 1, 2 and 3—the only classes that provide longitude and latitude required for downstream mapping—thereby cutting unnecessary branching and memory allocation.

Code section	**What it does**	**Per-item**	**# n**	**Cost**
List-load	Strip/store each log line.	$O(1)$	n	$O(n)$
Parser loop	Decode	$O(1)$	n	$O(n)$
Progress loop	Dummy iterate	$O(1)$	n	$O(n)$
			Total time	$\mathbf{\Theta(n)}$
Total space: $\mathbf{\Theta(n)}$				

Table 1: Time and Space complexity

Algorithms' efficiency improvements

Basic Algorithm

Table 2 summarizes the runtime profile of the `AIS_Message_Decoder_Kiki_Beumer_22April.ipynb` algorithm, listing each function's performance metrics. Table 3 describes the meaning of each metric for clearity. Ranking the functions from most to least time-consuming gives the following:

- `decode_message` — 461 s (6.6×10^6 calls)
- `extract_and_convert_payload` — 188 s
- `extract_timestamp_and_message` — 123 s
- `binary_to_float` — 114 s (9.2×10^6 calls)
- `verify_checksum` — 99 s

ncalls	tottime	cumtime	filename:lineno(function)
6 599 735	68.474	461.731	...298679368.py:1 (decode_message)
6 599 735	112.868	187.659	...769171144.py:1 (extract_and_convert_payload)
6 599 735	27.518	123.080	...481137939.py:4 (extract_timestamp_and_message)
9 191 332	19.253	114.183	...058317766.py:3 (binary_to_float)
6 599 735	68.833	99.103	...116254458.py:1 (verify_checksum)
9 191 332	35.926	94.930	ast.py:54 (literal_eval)
160 383 122	72.432	72.432	{built-in} builtins.format
26 398 940	18.605	67.264	re.py:197 (search)
4 595 666	3.248	65.644	...887684275.py:1 (get_long)
4 595 666	4.343	56.130	...291772410.py:1 (get_lat)
9 191 332	8.621	49.735	ast.py:33 (parse)
4 598 880	22.459	39.785	...019894860.py:1 (check_mmsi)
9 191 332	39.293	39.293	{built-in} builtins.compile
32 998 675	20.178	28.481	re.py:288 (_compile)
26 398 940	25.317	25.317	{method 'search' of 're.Pattern' objects}
268 960 657	23.865	23.865	{built-in} builtins.ord
6 599 735	16.057	21.610	...606084170.py:3 (get_ais_header)
1	0.000	21.093	pandas.core.frame:__init__
6 599 735	5.087	20.596	re.py:202 (sub)
1	0.000	14.980	pandas.core.internals.construction:nested_data_to_arrays

Table 2: Profiler output – top 20 functions ordered by cumulative time

Column	Meaning
`ncalls`	how many times the function was executed (recursive calls counted)
`tottime`	time spent *inside* the function body (excludes sub-calls)
`cumtime`	time spent in the function *and* all sub-functions it invoked
`filename:lineno(function)`	source location and function name

Table 3: Profile output columns and their meaning.

Enhanced Algorithm (24 April)

The notebook `AIS_Message_Decoder_24April.ipynb` implements a more advanced version of the decoder. Profiling results (Table 4) highlighted *decode_message* and *extract_and_convert_payload* as the main areas for optimization. Both functions have been optimized.

Improvements to `decode_message`

1. **Single payload conversion:** the entire 168-bit payload is now cast to an `int` once, eliminating ∼15 slice–to–int conversions per message.
2. **Bit-mask helper** (`_get_bits`): Instead of using string operations to get the values, this function uses built-in bit shifting and masking techniques for more efficient extraction.
3. **Centralised field map:** each field's length is defined only once.
4. **Signed-value handling in place:** The correction for ROT, latitude, and longitude values (which are in two's complement format) is done directly with numbers, without using binary strings in between.
5. **Early guard clause:** one length check and message-type filter prevent unnecessary work on invalid messages.
6. **Deferred dict construction:** all calculations are performed on locals first; the result dictionary is assembled once at the end.
7. **Removed redundant length tests:** the prior `if len(binary_str) < ...` blocks (10+ instances) have been collapsed into the single early guard.

Improvements to `extract_and_convert_payload`

1. **Module-level lookup tables:**
   - `_SIXBIT_VAL` – a 256-byte array mapping `ord(char)` → 6-bit value.
   - `_SIXBIT_BIN` – 64 pre-formatted 6-character binary strings.
2. **Direct array indexing:** `_SIXBIT_VAL[ord(ch)]` replaces dictionary lookups inside the loop.
3. **Efficient string assembly:** Combine parts in a list and use ".join(...) just once, avoiding slow repeated concatenations.
4. **No per-call dict build:** the original runtime construction of `six_bit_ascii` has been removed.
5. **Net result:** The function remains pure Python yet runs roughly six times faster, and it can still be further accelerated with Numba or NumPy batching if desired.

Table 4: Profiler output (24 April) — top 20 functions ordered by cumulative time

ncalls	tottime	cumtime	filename:lineno(function)
6 599 735	82.425	557.293	...22644730.py:12 (decode_message)
9 361 720	26.725	185.759	...058317766.py:3 (binary_to_float)
9 361 720	43.778	159.034	ast.py:54 (literal_eval)
6 599 735	33.879	150.558	...481137939.py:4 (extract_timestamp_and_message)
6 599 735	96.412	143.023	...595209264.py:10 (extract_and_convert_payload)
6 599 735	87.494	125.227	...116254458.py:1 (verify_checksum)
4 680 860	4.353	111.307	...291772410.py:1 (get_lat)
4 680 860	4.011	82.816	...887684275.py:1 (get_long)
26 398 940	23.085	81.456	re.py:197 (search)
9 361 720	11.381	73.970	ast.py:33 (parse)
9 361 817	60.229	60.229	{built-in} builtins.compile
4 680 860	29.593	51.208	...019894860.py:1 (check_mmsi)
429 409 235	47.644	47.644	{built-in} builtins.ord
9 361 720	13.788	35.571	ast.py:82 (_convert)
56 989 161	33.960	33.960	...22644730.py:5 (_get_bits)
33 000 279	22.542	32.872	re.py:288 (_compile)
26 398 973	31.336	31.336	{method 'search' of 're.Pattern' objects}
1	0.001	29.408	pandas.core.frame:__init__
6 599 735	20.634	27.591	...606084170.py:3 (get_ais_header)
6 601 031	6.508	25.501	re.py:202 (sub)

After optimisation, the following functions now account for the greatest share of execution time, with their time complexity ranked as follows.

- `decode_message` — 557 s (6.6×10^6 calls)
- `binary_to_float` — 186 s (9.4×10^6 calls)
- `extract_timestamp_and_message` — 151 s
- `extract_and_convert_payload` — 143 s
- `verify_checksum` — 125 s

Comparison of running time

The second profiling run, `AIS_Message_Decoder_Kiki_Beumer_24April` (try 2), reduced the time spent in `extract_and_convert_payload` by about 24% and cut some regular-expression costs, but it moved more work into `decode_message`, raising that function's cumulative time by roughly 20%. Because `decode_message` runs for every NMEA line, this extra cost pushes the total runtime up from 722 s in `AIS_Message_Decoder_Kiki_Beumer_22April` (try 1) to 882 s. Extra helper functions now called inside `decode_message`, such as `binary_to_float` and `_get_bits`, also rank higher, showing that added Python work cancels much of try 2's earlier gains. Overall, try 1 remains the faster and more straightforward implementation, compared to try 2. The best way forward is to combine the improvements (`AIS_Message_Decoder_Kiki_Beumer_24April.2` contains a

combination improving the code as shown in Table 5.

Table 5: Profiler output (combined algorithm) — top 20 functions ordered by cumulative time

ncalls	tottime	cumtime	filename:lineno(function)
6 599 735	42.476	249.640	...298679368.py:1 (decode_message)
6 599 735	16.792	77.829	...481137939.py:4 (extract_timestamp_and_message)
9 191 332	13.019	74.757	...058317766.py:3 (binary_to_float)
6 599 735	47.725	72.087	...595209264.py:10 (extract_and_convert_payload)
6 599 735	44.791	63.856	...116254458.py:1 (verify_checksum)
9 191 332	23.132	61.738	ast.py:54 (literal_eval)
26 398 940	11.522	43.127	re.py:197 (search)
4 595 666	2.150	42.706	...887684275.py:1 (get_long)
4 595 666	2.764	36.966	...291772410.py:1 (get_lat)
9 191 332	5.376	32.704	ast.py:33 (parse)
9 191 332	26.190	26.190	{built-in} builtins.compile
4 598 880	14.001	24.716	...019894860.py:1 (check_mmsi)
429 408 701	24.361	24.361	{built-in} builtins.ord
32 998 675	12.795	17.994	re.py:288 (_compile)
26 398 940	16.815	16.815	{method 'search' of 're.Pattern' objects}
1	0.001	14.609	pandas.core.frame:__init__
6 599 735	10.146	13.668	...606084170.py:3 (get_ais_header)
6 599 735	3.203	13.075	re.py:202 (sub)
165 047 061	12.157	12.157	{method 'append' of 'list' objects}
1	0.000	10.131	pandas.core.internals.construction:nested_data_to_arrays

Time Comparison for Different NMEA Formats

In this section, we compare the time efficiency of processing different datasets containing various NMEA formats. This allows us to verify that all supported formats are correctly handled by the decoding algorithm. Additionally, it provides insight into the scalability and performance of the code on these different datasets.

The table below summarizes the time taken to decode each dataset. Note that decoding times may vary depending on several factors, such as hardware performance, dataset size, background processes, and the complexity of the data itself.

Dataset	NMEA Format	File Size (MB)	Algorithm	Running Time (m)
Dataset001.ais	ISO 8601 inline	451 MB	Basic	29
			Enhanced	51
			Combined	12
iala-log-20131106	Proprietary	861 MB	Basic	5
			Enhanced	23
			Combined	15
IALAGLADSTONE0506_ITU123_20230607_00	Tag Block	15.8 MB	Basic	0.2
			Enhanced	0.5
			Combined	0.4
Hatter_Barn_April_2016	Multi-fragment	148 MB	Basic	2
			Enhanced	5
			Combined	4
iala-log-20131212	Proprietary	1.99 GB	Basic	30
			Enhanced	25
			Combined	45

Table 6: Time comparison of decoding different NMEA format datasets