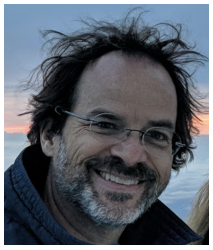
IN31B-30: NCO-JSON: A Flexible, Complete JavaScript Object Notation for netCDF

Charles S. Zender

Departments of Earth System Science and Computer Science, University of California, Irvine



PRESENTED AT:

SIMPLY COMPLETE

JavaScript Object Notation (JSON) is a widely used text format for data exchange. Previous netCDF-to-JSON translators were incomplete or overly complex. Here we describe NCO-JSON, a flexible JSON format that describes any classic or extended format netCDF dataset. NCO-JSON expresses the richness of the Common Data Model and increases interoperability between web services and netCDF data.

NCO-JSON is designed to be complete, reproducible, and legible. It looks...like JSON:

```
> ncks --json -v one in.nc

{

 "variables": {

   "one": {

     "type": "float",

     "attributes": {

       "long_name": "one"

     },

     "data": 1.0

   }

 }

}
```

NCO-JSON uses eight object types (`groups`, `dimensions`, `variables`, `shape`, `attributes`, `type`, `types`, `data`) to represent netCDF. These types give access to the complete netCDF namespace, so identifiers are not limited.

NETCDF CLASSIC

NCO-JSON represents a netCDF classic dataset as a `dimensions` list followed by a `variables` list. Each variable object must contain a `type` object and may contain an `attributes` list and a single `data` object. By default NCO-JSON formats metadata in the most legible and simple JSON syntax. This only differentiates between integers, floating point numbers, and strings:

```
> ncks --json -v one in.nc

{

 "variables": {

   "one": {

     "type": "float",

     "attributes": {

       "long_name": "one"

     },

     "data": 1.0

   }

 }

}
```

Multidimensional arrays must include a `shape` object that orders the relevant dimensions (from the `dimensions` object) before the `data` object. By default NCO-JSON prints multidimensional arrays with compound brackets that indicate the beginnings and ends of hyperslabs in each dimension:

```
> ncks -C -H --jsn_fmt=0 -v two_dmn_rec_var in.nc

{

 "dimensions": {

   "lev": 3,

   "time": 10

 },

 "variables": {

   "two_dmn_rec_var": {

     "shape": ["time", "lev"],

     "type": "float",
```

```
     "data": [[1.0, 2.0, 3.0], [1.0, 2.10, 3.0], [1.0, 2.20, 3.0], [1.0, 2.30, 3.0],
[1.0, 2.40, 3.0], [1.0, 2.50, 3.0], [1.0, 2.60, 3.0], [1.0, 2.70, 3.0], [1.0, 2.80,
3.0], [1.0, 2.90, 3.0]]

   }

 }

}
```

Adding 4 to any format level unrolls multi-dimensional arrays by removing compound brackets:

```
> ncks -C -H --jsn_fmt=4 -v two_dmn_rec_var in.nc

{

 "dimensions": {

   "lev": 3,

   "time": 10

 },

 "variables": {

   "two_dmn_rec_var": {

     "shape": ["time", "lev"],

     "type": "float",

     "data": [1.0, 2.10, 3.0, 1.0, 2.20, 3.0, 1.0, 2.30, 3.0, 1.0, 2.40, 3.0, 1.0, 2.50,
3.0, 1.0, 2.60, 3.0, 1.0, 2.70, 3.0, 1.0, 2.80, 3.0, 1.0, 2.90, 3.0, 1.0, 2.90, 3.0]

   }

 }

}
```

Compound brackets are probably more legible, and unrolled arrays are more compact. Both formats are equally valid JSON.

NETCDF EXTENDED

NCO-JSON supports the Extended Common Data Model including user-defined types and hierarchical groups as shown in the following examples. We begin with **Enumerated Types:**

```
> ncks --json enum.nc

{

 "types": {

   "enum_ubyte_t": [ "Clear":0, "Cumulonimbus":1, "Stratus":2, "Missing":128 ],

 }

 "dimensions": {

   "lon": 4

 },

 "variables": {

   "cld_flg": {

     "shape": ["lon"],

     "type": "enum_ubyte_t",

     "attributes": {

       "_FillValue": Missing

     },

     "data": ["Stratus", "Missing", "Cumulonimbus", "Clear"]

   }

 }

}
```

Next we show **Variable Length Arrays** (vlens):

```
> ncks --json vlen.nc

{

 "types": {

   "int(*)" : "vlen_int_t",

 },
```

```
 "dimensions": {
   "lat": 2,
 },
 "variables": {
   "vlen_int_1D": {
     "shape": ["lat"],
     "type": "vlen_int_t",
     "attributes": {
       "_FillValue": [-999]
     },
     "data": [[17, 18, 19], [1, 2, 3, 4, 5, 6, 7, -2147483647, 9, -2147483647]]
   }
 }
}
```

Finally we show **Groups:**

```
> ncks --json grp.nc
{
 "groups": {
   "g1": {
     "variables": {
       "g1v1": {
         "type": "int"
         "data": 1
       }
     }
   },
   "g2": {
     "variables": {
       "g2v1": {
```

```
            "type": "int"

            "data": 2

          }

        }

      }

    }

  }
```

REPRODUCIBILITY

These NCO commands produce NCO-JSON output in order of increasing reproducibility:

`ncks --json` # Default (i.e., most legible)

`ncks --jsn_fmt=0` # Same as above

`ncks --jsn_fmt=1` # Legible+Pedantic

`ncks --jsn_fmt=2` # Always pedantic

Note the absence of explicit attribute types in the default (non-pedantic) formats of all netCDF atomic types:

```
> ncks --jsn_fmt=0 -v att_var in.nc

...

"attributes": {

  "byte_att": [0, 1, 2, 127, -128, -127, -2, -1],

  "char_att": "Sentence one.\nSentence two.\n",

  "short_att": 37,

  "int_att": 73,

  "float_att": [70.010, 69.0010, 68.010, 67.010],

  "double_att": [70.010, 69.0010, 68.010, 67.0100010],

  "ubyte_att": [0, 1, 2, 127, 128, 254, 255, 0],

  "ushort_att": 37,

  "uint_att": 73,

  "int64_att": 9223372036854775807,

  "uint64_att": 18446744073709551615,

  "string_att": "Hello, World"

},
```

The default formatting is the most legible, yet is ambiguous about which specific netCDF atomic type underlies the data. This ambiguity must be resolved to preserve exact reproducibility of the original data type under round-trip translations. NCO-JSON therefore offers formats that are more pedantic because they turn each attribute into an object that expliciltyly includes its netCDF atomic type:

```
> ncks --jsn_fmt=2 -v one in.nc

{
```

```
 "variables": {

   "one": {

     "type": "float",

     "attributes": {

       "long_name": { "type": "char", "data": "one"}

     },

     "data": 1.0

   }

 }

}
```

The "Legible+Pedantic" mode outputs attributes of three netCDF atomic types (`int, float, char`) without any explicit explicit `type` object because these three types map 1-to-1 to native JSON types. In this mode all other netCDF atomic types (`short, double, string, unsigned byte, ...`) are output with explicit type information. The idea here is that JSON is often used to convey metadata for which the subtle differences between the atomic types makes no difference, so only use extra fomatting for non-default types:

```
> ncks --jsn_fmt=1 -v att_var in.nc
...
"att_var": {
  "dims": ["time"],
  "type": "float",
  "attributes": {
    "byte_att": { "type": "byte", "data": [0, 1, 2, 127, -128, -127, -2, -1]},
    "char_att": "Sentence one.\nSentence two.\n",
    "short_att": { "type": "short", "data": 37},
    "int_att": 73,
    "float_att": [73.0, 72.0, 71.0, 70.010, 69.0010, 68.010, 67.010],
    "double_att": { "type": "double", "data": [73.0, 72.0, 71.0, 70.010, 69.0010,
68.010, 67.0100010]}
  },
  "data": [10.0, 10.10, 10.20, 10.30, 10.40101, 10.50, 10.60, 10.70, 10.80, 10.990]
...
```

That is more legible than fully pedantic formatting that includes type objects for every attribute and is therefore fully reproducible:

```
> ncks --jsn_fmt=2 -v att_var in.nc

...

"attributes": {

 "byte_att": { "type": "byte", "data": [0, 1, 2, 127, -128, -127, -2, -1]},

 "char_att": { "type": "char", "data": "Sentence one.\nSentence two.\n"},

 "short_att": { "type": "short", "data": 37},
```

```
 "int_att": { "type": "int", "data": 73},

 "float_att": { "type": "float", "data": [70.010, 69.0010, 68.010, 67.010]},

 "double_att": { "type": "double", "data": [70.010, 69.0010, 68.010, 67.0100010]},

 "ubyte_att": { "type": "ubyte", "data": [0, 1, 2, 127, 128, 254, 255, 0]},

 "ushort_att": { "type": "ushort", "data": 37},

 "uint_att": { "type": "uint64", "data": 73},

 "int64_att": { "type": "int64", "data": 9223372036854775807},

 "uint64_att": { "type": "uint64", "data": 18446744073709551615},

 "string_att": { "type": "string", "data": "Hello, World"}

},
```

Since fully pedantic mode takes more space and is less legible, use it when reproducibility is a paramount concern, i.e., when it may be important to reconstruct the original dataset during a round-trip of netCDF->JSON->netCDF.

COMPARE TO CDL, XML, HDF5-JSON

netCDF has long supported two ASCII data formats, the Common Data Language (CDL), and the netCDF Markup Language (NcML), an XML dialect. In addition, HDF5 has a complete JSON dialect that also works for netCDF4 data. Below are dumps of the same file in CDL, XML, and HDF5-JSON.

First, CDL provides a complete netCDF representation that is also legible:

```
> ncks -v one in.nc

netcdf in {

 variables:

   float one ;

     one:long_name = "one" ;

 data:

   one = 1 ;

} // group /
```

The same file expressed in NcML is much more opaque to humans:

```
> ncks --xml -v one in.nc

<?xml version="1.0" encoding="UTF-8"?>

<ncml:netcdfxmlns:ncml="http://www.unidata.ucar.edu/   namespaces/netcdf/ncml-2.2"
location="file:in.nc">

 <ncml:variable name="one" type="float" shape="">

   <ncml:attribute name="long_name" separator="*" value="one" />

   <ncml:values>1.</ncml:values>

 </ncml:variable>

</ncml:netcdf>
```

As a dialect of XML, NcML is supported by existing cyberinfrastructure, e.g., THREDDS and OPeNDAP.

Third, HDF5-JSON represents the full HDF5 data model (a superset of netCDF) that includes object references as UUIDs. HDF5-JSON is necessarily more complex and verbose than NCO-JSON.

```
> jelenak@thg:~$ h5tojson one.nc

{

  "apiVersion": "1.1.1",

  "datasets": {
```

```
"f1d21bba-86e3-11e8-83df-760060ca3401": {
  "alias": [
    "/one"
  ],
  "attributes": [
    {
      "name": "long_name",
      "shape": {
        "class": "H5S_SCALAR"
      },
      "type": {
        "charSet": "H5T_CSET_ASCII",
        "class": "H5T_STRING",
        "length": 3,
        "strPad": "H5T_STR_NULLPAD"
      },
      "value": "one"
    }
  ],
  "creationProperties": {
    "allocTime": "H5D_ALLOC_TIME_LATE",
    "fillTime": "H5D_FILL_TIME_IFSET",
    "fillValue": 9.969209968386869e+36,
    "layout": {
      "class": "H5D_CONTIGUOUS"
    }
  },
  "shape": {
    "class": "H5S_SCALAR"
  },
```

```
        "type": {
          "base": "H5T_IEEE_F32LE",
          "class": "H5T_FLOAT"
        },
        "value": 1.0
      }
    },
  "groups": {
      "f1d0bac6-86e3-11e8-b54d-760060ca3401": {
        "alias": [
          "/"
        ],
        "attributes": [
          {
            "name": "_NCProperties",
            "shape": {
              "class": "H5S_SCALAR"
            },
            "type": {
              "charSet": "H5T_CSET_ASCII",
              "class": "H5T_STRING",
              "length": 57,
              "strPad": "H5T_STR_NULLPAD"
            },
            "value": "version=1|netcdflibversion=4.4.1.1|hdf5libversion=1.10.2"
          }
        ],
        "links": [
          {
```

```
                "class": "H5L_TYPE_HARD",
                "collection": "datasets",
                "id": "f1d21bba-86e3-11e8-83df-760060ca3401",
                "title": "one"
            }
        ]
    }
  },
  "root": "f1d0bac6-86e3-11e8-b54d-760060ca3401"
}
```

TRADE-OFFS AMONG ASCII NETCDF FORMATS

Strengths & Weaknesses of ASCII netCDF Formats

	Strengths	Weaknesses
CDL	Legible, Complete	Niche
NcML	XML, THREDDS, OPeNDAP	netCDF4 gaps Verbose
JSON	JSON, Legible, (OPeNDAP)	???

STATUS AND FUTURE

NCO-JSON is a concise JSON dialect that can completely reproduce netCDF datasets. Multiple independent software projects have adopted the NCO-JSON dialect to represent netCDF-conforming datasets. These include NCO, ERDDAP, CF-JSON, and STAR-JSON. An OPeNDAP implementation is clearly feasible, given its recent support for COV-JSON.

To our knowledge, no software yet ingests NCO-JSON and produces netCDF. However, NCO-JSON is designed and ordered to make parsing it easy. A mechanism to define record dimensions is under consideration.

A manuscript that formally describes NCO-JSON is in preparation. We welcome your comments.

DISCLOSURES

We are indebted to Chris Barker and Pedro Vicent-Nunes for stimulating discussions of how to make this JSON format more economic, readable, and interoperable. Bob Simons contributed helpful corner case examples. Supported by DOE ACME DE-SC0012998, DOE ARPA-E DE-AR0000594, NASA ACCESS NNX14AH55A, and NSF ICER AGS-1541031. This research was supported as part of the Energy Exascale Earth System Model (E3SM) project, funded by the U.S. Department of Energy, Office of Science, Office of Biological and Environmental Research. This material is based upon work supported by the National Science Foundation under Grant AGS-1541031.

ABSTRACT

JavaScript Object Notation (JSON) is an increasingly popular text format for data exchange. netCDF encapsulates the Common Data Model (CDM) and a binary format for machine-independent and network-transparent storage of scientific data and metadata. Previous netCDF-to-JSON translators have been custom solutions with incomplete features, or based on more complex formats than the CDM. Here we describe a flexible JSON format that describes any classic or extended format netCDF dataset. This format, called NCO-JSON, expresses the richness of the CDM and increases interoperability between web services and netCDF data. NCO-JSON requires no reserved keywords and so is completely compatible with all netCDF datasets. It allows for selectable levels of fidelity to the original data and metadata. The most concise and human-legible form of NCO-JSON is also lossy. By design it distinguishes only the three atomic JSON datatypes (float, string, and int). This suffices for many purposes yet cannot guarantee bit-for-bit reproducibility of many netCDF datatypes, especially in round-trip translations. NCO-JSON uses a more complex object notation to encode the additional type information required to reproduce netCDF datasets with full fidelity. We present the rules and design of the NCO-JSON format, show results with real-world datasests, quantify the space advantages vs. alternate formats (both JSON and XML), and discuss corner cases and possible extensions.

REFERENCES

Bray, T. (2013), JavaScript Object Notation (JSON) documentation, http://www.json.org.

Caron, J. (2013), NetCDF Markup Language (NcML) documentation, http://www.unidata.ucar.edu/software/thredds/current/netcdf-java/ncml/#NcML22.

Caron, J. (2014), Unidata's Common Data Model version 4, http://www.unidata.ucar.edu/software/thredds/current/netcdf-java/CDM.

HDF Group (2015), HDF5: API Specification Reference Manual, The HDF Group, Champaign-Urbana, IL.

Simons, B. (2017), ERDDAP (Environmental Research Division Data Access Program), https://coastwatch.pfeg.noaa.gov/erddap

Zender, C. S. (2008), Analysis of Self-describing Gridded Geoscience Data with netCDF Operators (NCO), Environ. Modell. Softw., 23(10), 1338-1342, doi:10.1016/j.envsoft.2008.03.004.