The GAVO STC Library

A library to process VO STC specifications

Author: Markus Demleitner
Email: gavo@ari.uni-heidelberg.de
Date: 2024-02-14

Contents

This library aims to ease processing specifications of space-time coordindates (STC) according to the IVOA STC data model with the XML and string serializations. Note that it is at this point an early beta at best. To change this, I welcome feedback, even if it's just "I'd need X and Y". Honestly.
More specifically, the library is intended to help in:

- supporting ADQL region specifications and conforming them
- generating registry coverage specifications from simple STC-S
- generating utypes for VOTable embedding of STC information and parsing from them

The implementation should conform to STC-S 1.33; what STC-X is supported conforms to STC-X 1.00 (but see Limitations).

Installation

If you are running a Debian-derived distribution, see Adding the GAVO repository. When you follow that recipe,

```
aptitude install python-gavostc
```

is enough.
Otherwise, you will have to install the source distribution. Unpack the .tar.gz and run:

```
python setup.py install
```

You will normally need to do this as root for a system-wide installation. There are, however, alternatives, first and foremost a virtual python that will keep your managed directories clean.
This library's setup is based on setuptools. Thus, it will generally obtain all necessary dependencies from the net. For this to be successful, you will have to have net access. If all this bothers you, contact the authors.

Usage

Command Line

For experiments, we provide a simple command line tool. Try:

```
gavostc help
```

to see what operations it exposes. Here are some examples:

```
$ gavostc help
Usage: gavostc [options] <command> {<command-args}
  Use command 'help' to see commands available.

Options:
  -h, --help            show this help message and exit
  -e, --dump-exception  Dump exceptions.

Commands include:
conform <srcSTCS>. <dstSTCS>  -- prints srcSTCS in the system of dstSTCS.
help -- outputs help to stdout.
parseUtypes --- reads the output of utypes and prints quoted STC for it.
parseX <srcFile> -- read STC-X from srcFile and output it as STC-S, - for stdin
resprof <srcSTCS> -- make a resource profile for srcSTCS.
utypes <QSTCS> -- prints the utypes for the quoted STC string <QSTCS>.
$ gavostc resprof "Polygon ICRS 20 20 21 19 18 17" | xmlstarlet fo
<?xml version="1.0"?>
<STCResourceProfile
  xmlns="http://www.ivoa.net/xml/STC/stc-v1.30.xsd"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.ivoa.net/xml/STC/stc-v1.30.xsd http://vo.ari.uni-heidelberg.de/docs
  <AstroCoordSystem id="thgloml">
    <SpaceFrame id="thdbgwl">
      <ICRS/>
      <UNKNOWNRefPos/>
      <SPHERICAL coord_naxes="2"/>
    </SpaceFrame>
  </AstroCoordSystem>
  <AstroCoordArea coord_system_id="thgloml">
    <Polygon frame_id="thdbgwl" unit="deg">
      <Vertex>
        <Position>
          <C1>20.0</C1>
          <C2>20.0</C2>
        </Position>
      </Vertex>
      <Vertex>
        <Position>
          <C1>21.0</C1>
          <C2>19.0</C2>
        </Position>
      </Vertex>
      <Vertex>
        <Position>
          <C1>18.0</C1>
          <C2>17.0</C2>
        </Position>
      </Vertex>
    </Polygon>
  </AstroCoordArea>
</STCResourceProfile>
$ gavostc resprof "Circle FK5 -10 340 3" | gavostc parseX -
Circle FK5 -10.0 340.0 3.0
$ gavostc conform "Position GALACTIC 3 4 VelocityInterval Velocity 0.01 -0.002 unit deg/cy" "Positic
Position FK5 264.371974024 -24.2795040403 VelocityInterval Velocity 0.00768930497899 0.0073745962452
$ gavostc utypes 'Redshift TOPOCENTER VELOCITY "z" Error "e_z" PixSize "p_z"'
AstroCoordSystem.RedshiftFrame.value_type                   = VELOCITY
AstroCoordSystem.RedshiftFrame.DopplerDefinition            = OPTICAL
AstroCoordSystem.RedshiftFrame.ReferencePosition             = TOPOCENTER
AstroCoords.Redshift.Error                                   -> e_z
AstroCoords.Redshift.Value                                   -> z
AstroCoords.Redshift.PixSize                                 -> p_z
$ gavostc utypes 'Redshift TOPOCENTER VELOCITY "z" Error "e_z" PixSize "p_z"'\
| gavostc parseUtypes
Redshift TOPOCENTER VELOCITY "z" Error "e_z" PixSize "p_z"
```

Programmatic

The library is written in python, and thus currently can only be operated from python programs. It should not be too hard to embed it into C or even Java programs. If you have such needs, contact the author.
See API for details.

Limitations

- Internally, all dates and times are represented as datetimes, and all information whether they were JDs or MJDs before is discarded. Thus, you cannot generate STC with M?JDTime.
- All stc:DataModel utypes are ignored. On output and request, only stc:DataModel.URI is generated, fixed to uri stc.STCNamespace.
- "Library" coordinate systems for ECLIPTIC coordinates are not supported since it is unclear to me how the equinox of those is expressed.
- On system transformations, ellipses are not rotated, just moved. No "wiggles" (errors, etc) are touched at all.
- There currently is not real API for "bulk" transforms, i.e., computing a transformation once and then apply it to many coordinates. The code is organized to make it easy to add such a thing, though.
- Serialization of floats and friends is with a fixed format that may lose precision for very accurate values. The solution will probably be a floatFormat attribute on the frame/metadata object, but I'm open to other suggestions.
- Reference positions are not supported in any meaningful way. In particular, when transforming STCs, transformations between all reference positions are identities. This won't hurt much for galactic or extragalactic objects but of course makes the whole thing useless for solar-system work. If someone points me to a concise collection of pertinent formulae, adding real reference positions transformations should not be hard.
- The behaviour of some transforms (in particular FK5<->FK4) close to the poles need some attention.
- Empty coordinate values (e.g., 2D data with just one coordinate) are not really supported. Processing them will, in general, work, but will, in general, not yield the expected result. This is fixable, but may require changes in the data model.

- No generic coordinates. Those can probably be added relatively easily, but it would definitely help if someone had a clear use case for them
- Spectral errors and their "wiggles" (error, size, etc) must be in the same "flavor", i.e., either frequency, wavelength, or energy. If they are not, the library will silently fail. This is easily fixable, but there's too much special casing in the code as is, and I consider this a crazy corner case no one will encounter.
- No reference on posAngles, always assumed to be 'X'.
- Spatial intervals are system-conformed analogous to geometries, so any distance information is disregaded. This will be fixed on request.
- No support for Area.
- Frame handling currently is a big mess; in particular, the system changing functions *assume* that the frames on positions, velocities and geometries are identical. I'll probably more towards requiring astroCoords being in astroSystem.

Extensions to STC-S

- After ECLIPTIC, FK4, or FK5, an equinox specification is allowed. This is either J<number> or B<number>.
- For velocities, arbitrary combinations of spaceUnit/timeUnit are allowed.
- To allow the notation of STC Library coordinate systems, you can give a System keyword with an STC Library tag at the end of a phrase (e.g., `System TT-ICRS-TOPO`). This overwrites all previous system information (e.g., `Time ET Position FK4 System TT-ICRS-TOPO` will result in TT time scale and ICRS spatial frame). We admit it's not nice, and are open to suggestions for better solutions.

Other Deviations from the Standard

- Units on geometries default to deg, deg when parsing from STC-X.
- The equinox to be used for ECLIPTIC isn't quite clear from the specs. The library will use a specified equinox if given, else the time value if given, else the equinox will be None (which probably is not terribly useful).

Bugs

- Conversions between TT and TCB are performed using the rough approximation of the explanatory supplement rather than the more exact expression.
- TT should be extended to ET prior to 1973, but this is not done yet.
- STC-S parse errors are frequently not very helpful.
- Invalid STC-X documents may be accepted and yield nonsensical ASTs (this will probably not be fixed since it would require running a validating parser, which with XSD is not funny, but I'm open to suggestions).

API

The public API to the STC library is obtained by:

```
from gavo import stc
```

This is assumed for all examples below.

The Data Model

The STC library turns all input into a tree called AST ("Abstract Syntax Tree", since it abstracts away the details for parsing from whatever serialisation you employ).

The ASTs are following the STC data model quite closely. However, it turned out that -- even with the changes already in place -- this is quite inconvenient to work with, so we will probably change it after we've gathered some experience. It is quite likely that we will enforce a much stricter separation between data and metadata, i.e., unit, error and such will go from the positions to what is now the frame object.

Thus, we don't document the data model fully yet. The gory details are in dm.py. Meanwhile, we will try to maintain the following properties:

- All objects in ASTs are considered immutable, i.e., nobody is supposed to change them once they are constructed.
- An AST object has attributes time, place, freq, redshift, velocity refererring to an objects describing quantities or None if not given. These are called "positions" in the following.
- An AST object has attributes timeAs, areas, freqAs, redshiftAs, velocityAs containing sequences of intervals or geometries of the respective quantities. These sequences are empty if nothing is specified. They are called areas in the following.
- Both positions and areas have a frame attribute giving the frame (for spatial coordinates, these have flavor, nDim, refFrame, equinox, and refPos attributes, quite like in STC).
- Positions have a values attribute containing either a python float or a tuple of floats (for spatial and velocity coordinates). For time coordinates, a datetime.datetime object is used instead of a float
- Positions have a unit attribute. We will keep this even if all other metadata move to the frame object. The unit attribute follows the coordinate values, i.e., they are tuple-valued when the values are tuples. For velocities and redshifts, there is a velTimeUnit as well.
- ASTs have a cooSystem attribute with, in turn, spaceFrame, timeFrame, spectralFrame, and redshiftFrame attributes.
- NULL is consistently represented as None, except when the values would be sequences, in which case NULL is an empty tuple.

Parsing STC-X

To parse an STC-X document, use `stc.parseSTCX(literal) -> AST`. Thus, you pass in a string containing STC-X and receive a AST structure.
Since STC documents should in general be rather small, there should be no necessity for a streaming API. If you want to read directly from a file, you could use something like:

```
def parseFromFile(fName):
  f = open(fName)
  stcxLiteral = f.read()
  f.close()
  return stc.parseSTCX(stcxLiteral)
```

The return value is a sequence of pairs of `(tagName, ast)`, where tagName is the namespace qualified name of the root element of the STC element. The tagName is present since multiple STC trees may be present in one STC-X document. The qualification is in standard W3C form, i.e., `{<namespace URI>}<element name>`. If you do not care about versioning (and you should not need to with this library), you could find a specific element using a construct like:

```
def getSTCElement(literal, elementName):
  for rootName, ast in stc.parseSTCX(literal):
    if rootName.endswith('}'+elementName):
      return ast
getSTCElement(open("M81.xml").read(), "ObservationLocation")
```

Note that the STC library does not contain a validating parser. Invalid STC-X documents will at best give you rather incomprehensible error messages, at worst an AST that has little to do with what was in the document. If you are not sure whether the STC-X you receive is valid, run a schema validator before parsing.
We currently understand a subset of STC-X that matches the expressiveness of STC-S. Most STC-X features that cannot be mapped in STC-X are silently ignored.

Generating STC-X

To generate STC-X, use the `stc.getSTCX(ast, rootElmement) -> str` function. Since there are quite a few root elements possible, you have to explicitely pass one. You can find root elements in `stc.STC`. It is probably a good idea to only use `ObservatoryLocation`, `ObservationLocation`, and `STCResourceProfile` right now. Ask the authors if you need something else.
There is the shortcut `stc.getSTCXProfile(ast) -> str` that is equivalent to `stc.getSTCX(ast, stc.STC.STCResourceProfile)`.

Parsing STC-S

To parse an STC-S string into an AST, use `stc.parseSTCS(str) -> ast`. The most common exception this may raise is stc.STCSParseError, though others are conceivable.

Generating STC-S

To turn an AST into STC-S, use `stc.getSTCS(ast) -> str`. If you pass in ASTs that use features not supported by STC-S, you should get an STCNotImplementedError or an STCValueError.

Generating Utypes

For embedding STC into VOTables, utypes are used. To turn an AST object into utypes, use `stc.getUtypes(ast) -> dict, dict`. The function returns a pair of dictionaries:

- the first dictionary, the "system dict", maps utypes to values. All utypes belong to AstroCoordSystem and into this group.
- the second dictionary, the "columns dict", maps values to utypes.

Of course, the columns dict doesn't make much sense with ASTs actually containing values. To sensibly use it it a way useful for VOTables, you can define your columns' STC using "quoted STC-S". In this format, you have identifiers in double quotes instead of normal STC-S values. Despite the double quotes, only python-compatible identifiers are allowed, i.e., these are *not* quoted identifiers in the SQL sense. The `stc.parseQSTCS(str) -> ast` function parses such strings.
Consider:

```
In [5]:from gavo import stc
In [6]:stc.getUtypes(stc.parseQSTCS(
   ...:'Position ICRS "ra" "dec" Error "e_p" "e_p"'))
Out[6]:
({'AstroCoordSystem.SpaceFrame.CoordFlavor': 'SPHERICAL',
  'AstroCoordSystem.SpaceFrame.CoordRefFrame': 'ICRS',
  'AstroCoordSystem.SpaceFrame.ReferencePosition': 'UNKNOWNRefPos'},
 {'dec': 'AstroCoords.Position2D.Value2.C2',
  'e_p': 'AstroCoords.Position2D.Error2Radius',
  'ra': 'AstroCoords.Position2D.Value2.C1'})
```

Note that there is no silly "namespace prefix" here. Nobody really knows what those prefixes really mean with utypes. When sticking these things into VOTables, you will currently need to stick an "stc:" in front of those.

Parsing Utypes

When parsing a VOTable, you can gather the utypes encountered to dictionaries as returned by `getUtypes`. You can then pass these to `parseFromUtypes(sysDict, colDict) -> ast`. The function does not expect any namespace prefixes on the utypes.

Conforming

You can force two ASTs to be expressed in the same frames, which we call "conforming". As mentioned above, currently only reference frames and equinoxes are conformed right now, i.e., the conversion from Galactic to FK5 1980.0 coordinates should work correctly. Reference positions are ignored, i.e. conforming ICRS TOPOCENTER to ICRS BARYCENTER will not change values.
To convert coordinates in ast1 to the frame defined by ast2, use the `stc.conformTo(ast1, ast2) -> ast` function. This could look like this:

```
>>> p = stc.parseSTCS("Circle ICRS 12 12 1")
>>> stc.conformTo(p, stc.parseSTCS("Position GALACTIC"))
>>> stc.conformTo(p, stc.parseSTCS("Position GALACTIC")).areas[0].center
(121.59990883115164, -50.862855782323962)
```

Conforming also works for units:

```
>>> stc.conformTo(p, stc.parseSTCS("Position GALACTIC unit rad")).areas[0].center
(2.1223187792285256, -0.8877243003685894)
```

Transformation

For simple transformations, you can ask DaCHS to give you a function just turning simple positions into positions. For instance,

```
from gavo import stc

toICRS = stc.getSimple2Converter(
        stc.parseSTCS("Position FK4 B1900.0"),
        stc.parseSTCS("Position ICRS"))
print(toICRS(30, 40))
```

shows how to build turn positions given in the B1900 equinox (don't sweat the reference system for data that old) to ICRS.

Equivalence

For some applications it is necessary to decide if two STC specifications are equivalent. Python's built-in equivalence operator requires all values in two ASTs to be identical except of the values of id attributes.
Frequently, you want to be more lenient:

- you might decide that unspecified values match anything
- you may ignore certain keys entirely (e.g., the reference position when you're doing extragalactic work or when a parallax error doesn't matter)
- you may want to view certain combinatinons as equivalent (e.g., ICRS and J2000 are quite close)

To support this, the STC library lets you define `EquivalencePolicy` objects. There is a default equivalence policy ignoring the reference position, defining ICRS and FK5 J2000 as equivalent, and matching Nones to anything. This default policy is available as `stc.defaultPolicy`. It has a single method, `match(sys1, sys2) -> boolean` with the obvious semantics. Note, however, that you pass in systems, i.e., `ast.cooSystem` rather than ASTs themselves.
You can define your own equivalence policies. Tell us if you want that and we'll document it. In the mean time, check `stc/eq.py`.

Hacking

For those considering to contribute code, here is a short map of the source code:

- cli -- the command line interface
- common -- exceptions, some constants, definition of the AST node base class
- conform -- high-level code for transformations between reference systems, units, etc.
- spherc.py, sphermath.py -- low-level transformations for spherical coordinate systems used by conform

- times -- helpers for converting time formats, plus transformations between time scales used by conform.
- dm -- the core data model, i.e. definitions of the classes of the objects making up the ASTs
- stcsast.py, stcxast.py -- tree transformers from STC-S and STC-X concrete syntax trees to ASTs.
- scsgen.py, stcxast.py -- serializers from ASTs to STC-S and STC-X
- utypegen.py, utypeast.py -- code generating and parsing utype dictionaries. These are thin wrappers around the STC-X code.
- stcs.py, stcsdefaults.py -- a grammar for STC-S and a definition of the defaults used during parsing and generation of STC-S.
- units.py -- units defined by STC, and transformations between them

Since the STC serializations and the sheer size of STC are not really amenable to a straightforward implementation, the stc*[gen|ast] code is not exactly easy to read. There's quite a bit of half-assed metaprogramming going on, and thus these probably are not modules you'd want to touch if you don't want to invest substantial amounts of time.
The conform, spherc, sphermath, units and time combo though shouldn't be too opaque. Start in conform.py contains "master" code for the transformations (which may need some reorganization when we transform spectral and redshift coordinates as well).
Then, things get fanned out; in the probably most interesting case of spherical coordinates, this this to spherc.py. That module defines lots of transformations and `getTrafoFunction`. All the spherical coordinate stuff uses an internal representation of STC, six vectors and frame triples; see conform.conformSystems on how to obtain these.
To introduce a new transformation, write a function or a matrix implementing it and enter it into the list in the construction of _findTransformsPath.
Either way: If you're planning to hack on the library, *please* let us know at gavo@ari.uni-heidelberg.de. We'll be delighted to help out with further hints.

Extending STC-S

Here's an example for an extension to STC-S: Let's handle the planetary ephemeris element.
Checking the schema, you'll see only two literals are allowed for the ephemeris: `JPL-DE200` and `JPL-DE405`. So, in `stcs._getSTCSGrammar`, near the definition of refpos, add:

```
plEphemeris = Keyword("JPL-DE200") | Keyword("JPL-DE405")
```

The plan is to allow the optional specification of the ephemeris used after refpos. Now grep for the occurrences of refpos and notice that there are quite a number of them. So, rather than fixing all those rules, we change the refpos rule from:

```
refpos = (Regex(_reFromKeys(stcRefPositions)))("refpos")
```

to:

```
    refpos = ((Regex(_reFromKeys(stcRefPositions)))("refpos")
  + Optional( plEphemeris("plEphemeris") ))
```

We can test this. In stcstest.STCSSpaceParsesTest, let's add the sample:

```
("position", "Position ICRS TOPOCENTER JPL-DE200"),
```

Now, the refpos nodes are handled in the _makeRefpos function, looking like this:

```
def _makeRefpos(node):
  refposName = node.get("refpos")
  if refposName=="UNKNOWNRefPos":
    refposName = None
  return dm.RefPos(standardOrigin=refposName)
```

The node passed in here is a pyparsing node. Since in our data model, None is always null/ignored, we can just take the planetary ephemeris if it's present, and the system will do the right thing if it's not there:

```
def _makeRefpos(node):
  refposName = node.get("refpos")
  if refposName=="UNKNOWNRefPos":
    refposName = None
  return dm.RefPos(standardOrigin=refposName,
    planetaryEphemeris=node.get("plEphemeris"))
```

Let's test this; testing STC-S to AST parsing takes place in stctest.py, so let's add a method to `CoordSysTest`:

```
def testPlanetaryEphemeris(self):
        ast = stcsast.parseSTCS("Time TT TOPOCENTER JPL-DE200")
        self.assertEqual(ast.astroSystem.timeFrame.refPos.planetaryEphemeris,
                "JPL-DE200")
```

Thus, we can parse the ephemeris spec from STC-S. To generate it, two things need to be done: The DM item must be transformed into the CST the STC-S is built from, and the part of the CST must be flattened out. Both things happen in stcsgen.py. The CST is just nested dictionaries. Refpos handline happens in refPosToCST, so replace:

```
def refPosToCST(node):
  return {"refpos": node.standardOrigin}
```

with:

```
def refPosToCST(node):
  return {
    "refpos": node.standardOrigin,
    "planetaryEphemeris": node.planetaryEphemeris,}
```

To flatten that out to the finished string, the flatteners need to be told that you want that key noticed. Grepping for repos shows that it's used in several places. So, let's define a "common flattener", which is a function taking a value and the CST node (i.e., a dictionary) the value was taken from and returns a string ready for inclusion into the STC-S. The flattener here would look like this:

```
def _flattenRefPos(val, node):
  return _joinWithNull([node["refpos"], node["planetaryEphemeris"]])
```

The `_joinWithNull` call makes sure that empty specifications do not show up the in result. This "global" flattener is now entered into `_commonFlatteners`, a dictionary mapping specific CST keys to flatten functions:

```
_commonFlatteners = {
...
        "refpos": _flattenRefPos,
}
```

The most convenient way to test this is to define a round-trip test. These again reside stcstest. Use `BaseGenerationTest` and add a sample pair like this:

```
("Redshift BARYCENTER JPL-DE405 3.5",
        "Redshift BARYCENTER JPL-DE405 3.5")
```

With this, you should be done.