2nd ASTERICS-OBELICS International School

4-8 June 2018, Annecy, France.

H2020-Astronomy ESFRI and Research Infrastructure Cluster
(Grant Agreement number: 653477).

A short course in PyVO

Markus Demleitner *(msdemlei@ari.uni-heidelberg.de)*
Hendrik Heinl *(heinl@ari.uni-heidelberg.de)*

Please open a browser and point it to

http://docs.g-vo.org/pyvo

Coming up (possibly): VO redux, operating simple data access services, multi-service queries, sending results around on the desktop, parameter discovery, TAP queries, async services, UCDs, ObsTAP, Registry, VO for the solar system, datalink, remote manipulation. . .

Prerequisites

- python and astropy, of course
- ⟨TOPCAT⟩ for viewing and visualising tables
- ⟨Aladin⟩ to work with images
- PyVO. Get it from
  - https://pypi.python.org/pypi/pyvo
  - or try `apt-get install python-pyvo`
  - or try `pip install pyvo`
  - or try `conda install pyvo`

What's the VO

The VO is a set of standards that let clients discover and interrogate astronomical data services in a uniform manner. Standards include:

- Registry – describing and finding services
- VOTable, UCD – writing tables with rich metadata
- SAMP – connecting software components
- SCS, SIAP, SSAP – querying catalog, image, and spectral services
- TAP – running remote database queries
- Datalink – bundling up complex data and services
- MOC, HiPS – sky coverage and hierarchical imaging

What's PyVO?

PyVO provides APIs for lots of VO protocols.

It's glue between astropy and python in general and the astronomical data services in the VO.

PyVO works for both python2 and python3. We hope the examples here do, too.

It's a community project. You're welcome to contribute at ⟨PyVO on github⟩

Running Simple Services

When querying “simple” remote services (image, spectral, cone search; *not* directly TAP), PyVO has a consistent pattern:

```
# <prot> is SIA, SSA, SCS, SLA...
import pyvo

# construct a service object with a service's endpoint URL
service = pyvo.dal.<prot>Service(access_url)

#call the search method with the protocol's parameters
for result in service.search(<parameters>):
  ...work on dict-like object result...
```

You’ll soon learn who to find out the access URLs.

Query a Single Image Service

Example: SIAP, the VO's protocol to access image servers.

Query a VO service for a list of images covering a small field on the sky, and download one of these images:

```
svc = pyvo.sia.SIAService(ACCESS_URL)
images = svc.search((11,35), (0.1, 0.1), verbosity=2)
image=iamges[0]
image.cachedataset()
```

[See trivial.py]

For SIAP, `pos` (as a tuple of ra and dec) and `size` (in degrees, either one radius or extent in ra and dec) are mandatory. More parameters: ⟨in the pyvo docs⟩.

This is Python

The advantage of doing this in Python is that it's easy to add your own logic:

```
svc = pyvo.sia.SIAService(ACCESS_URL)
for pos in [
    (10, 20),
    (45, 85)]:
  images = svc.search(pos, (0.5, 0.5), verbosity=2)
  for row in images:
    if not DATE_MIN<row.dateobs<DATE_MAX:
      continue
    row.cachedataset()
```

[See multisiap.py]

Also: `row.cachdataset` saves the image to your local disk under a name sensible for the metadata.

And now all-VO

The nice thing about standard services: Handle one, and you get them all. So, let's add a query to the Registry and run our query all over the VO –

```
for svc in registry.search(servicetype="image"):
  try:
    search_one_service(svc.accessurl, image_sender)
  except Exception:
    import traceback; traceback.print_exc()
```

[See globalsiap.py]

Rule: In multi-service queries, expect at least one service to be broken. Write your scripts to cope.

Add SAMP Magic

SAMP lets you exchange data between VO clients. Your script is a VO client, too. Let's make it broadcast some of the found images:

```
with vohelper.SAMP_conn() as conn:
  ... (search) ...
  vohelper.send_image_to(conn, None, image.acref)
```

[See globalsiapsamp.py]

(also, vohelper.py abstracts SAMP here).

Before running this, start Aladin so the images are displayed.

Custom Parameters

SIAP and SSAP services can define custom parameters. Discover them using a FORMAT=METADATA URL parameter.

Pass custom parameters as keyword arguments to search:

```
svc.search((107, -10), (0.1, 0.1),
  dateObs="57050/58050",
  bandpassId="SDSS i'")
```

[See siapextra.py]

Syntax trouble: Old-style VO services (parameters usually declared as `char[*]` or `double`) write intervals with slashes.

New-style (SIAv2, datalink...) have *interval* xtypes and type `double[2]`. Their intervals are written with a blank.

Enter TAP

What we've seen so far doesn't scale when you're interested in more regions.

Also, only fairly basic constraints are supported.

TAP is far more powerful.

Sample use case: Integrate photometry from different source catalogs, do some local work on results, try to obtain spectra for interesting candidates.

Step 1a: Synchronous Queries

Run queries via TAP:

```
access_url = "http://dc.g-vo.org/tap"

service = pyvo.dal.TAPService(access_url)
result = service.run_sync(
  """SELECT raj2000, dej2000, jmag, hmag, kmag
        FROM twomass.data
        WHERE jmag<3""")
for row in result:
  print(row["raj2000"], row["jmag"])
```

[See fetch3.py]

`result.to_table()` is an astropy.table instance – here, we take a column from it. To save it, say:

```
with open("result.vot") as f:
  result.to_table().write(output=f, format="votable")
```

Step 1b: Three Queries, TOPCAT

Separate “science” from “code” as much as possible:

```
 QUERIES = [
  ("twomass", "http://dc.zah.uni-heidelberg.de/tap",
    """SELECT TOP 1000000 raj2000, dej2000, jmag, hmag, kmag
    ...
  ("allwise", "http://tapvizier.u-strasbg.fr/TAPVizieR/tap",
    """SELECT raj2000, dej2000, w1mag, w2mag, w3mag, w4mag
    ...

with vohelper.SAMP_conn() as conn:
  topcat_id = vohelper.find_client(conn, "topcat")
  for short_name, access_url, query in QUERIES:
    service = pyvo.dal.TAPService(access_url)
    result = service.run_sync(query.format(**locals()), maxrec=90000)
    vohelper.send_table_to(conn, topcat_id, result.table, short_name)
```

[See fetch2.py]

Also new: send retrieved tables directly to TOPCAT.

Step 2: Go Async

When doing a lot of queries or long-running queries, run them asynchronously and in parallel.

```
jobs = set()
for short_name, access_url, query in QUERIES:
  job = pyvo.dal.TAPService(access_url).submit_job(
    query.format(**locals()), maxrec=9000000)
  job.run()
  jobs.add((short_name, job))

 while jobs:
  time.sleep(5)
  for short_name, job in list(jobs):
    if job.phase not in ('QUEUED', 'EXECUTING'):
      jobs.remove((short_name, job))
      vohelper.send_table_to(conn, topcat_id,
          job.fetch_result().table, short_name)
      job.delete()
```

[See fetch2_async.py]

Step 3a: UCDs build SEDs

Can we build SEDs from the results of the three services?

Not simply; photometry metadata in the VO isn't quite sufficient for that yet. However, UCDs let us do a workaround:

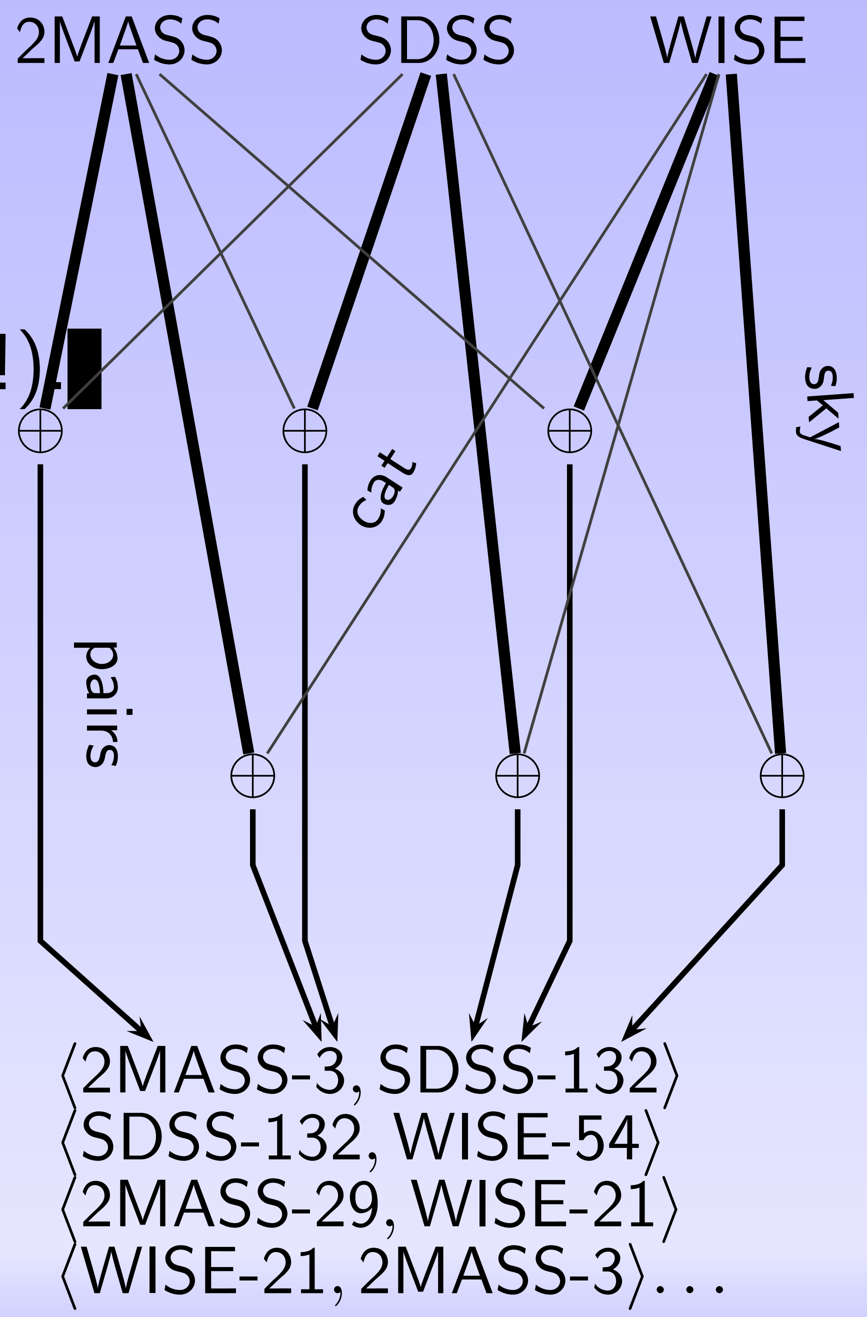
```
UCD_TO_WL = {
  "phot.mag;em.opt.u": 3.5e-7,
  "phot.mag;em.opt.b": 4.5e-7,
  "phot.mag;em.opt.v": 5.5e-7,
  "phot.mag;em.opt.r": 6.75e-7, ...

  for row in rows:
    for index, col in enumerate(row):
      ucd = row.columns[index].meta.get("ucd", "").lower())
      if ucd.startswith("phot.mag"):
        if ucd in UCD_TO_WL:
          phots.append((UCD_TO_WL[ucd], col))
```

Step 3b: Aggregate Photometry

Construction of “clusters” is in vohelper.py and uses astropy’s `SkyCoords` and `match_catalog_to_sky` (asymmetric!).

For three catalogs, we must perform six sky matches to get pairs, then walk the graph to gather the clusters.



Combine with “your” Code

This is python: Add your own logic!

Here: Let’s display the approximate SEDs and let the user interactively select “interesting” cases.

```
for pos, phots in seds:
    to_plot = np.array(phots)
    plt.semilogx(to_plot[:,0], to_plot[:,1], '-')
    plt.show(block=False)
    selection = raw_input(
      "s)elect SED, q)uit, enter for next? ")
    if selection=="q":
      break
    if selection=="s":
      selected.append(pos)
    plt.cla()
  return selected
```

Looking for Spectra

Suppose you have a couple of positions for "interesting" objects. Can we find spectra for them?

Let's use

ObsTAP = TAP with table `ivoa.obscore`

`ivoa.obscore` has lots of metadata on observational data products (spectra, cubes, timeseries).

Plan:

- Search for `obscore` services
- Use TAP upload to search to collect spectra
- Send spectra to SPLAT

Query the Registry

Iterate over all obscore services (here: see what data collections they house):

```
for svc_rec in pyvo.registry.search(datamodel="obscore"):
  svc = pyvo.dal.TAPService(svc_rec.access_url)
  result = svc.run_sync("SELECT DISTINCT obs_collection"
    " FROM ivoa.obscore")
  print("\n>>>>{}\n{}\n".format(
    svc_rec.short_name,
    "\n".join(
      r["obs_collection"] for r in result))
```

Do not run this script *just* for fun.

Query with Upload

For each ObsTAP service, we query against our object list:

```
if not svc.upload_methods:
  return

result = vohelper.run_sync_resilient(svc,
  """SELECT TOP 2000 oc.obs_publisher_did, oc.access_url
      FROM ivoa.obscore AS oc
      JOIN TAP_UPLOAD.pois AS mine
      ON 1=CONTAINS(
        POINT('ICRS', oc.s_ra, oc.s_dec),
        CIRCLE('ICRS', mine.ra, mine.dec, 0.01))
      WHERE oc.dataproduct_type='spectrum'
      """),
      uploads = {"pois": pois})
```

Collect Spectra finished

The rest is almost standard SAMP fare to get the spectra retrieved to SPLAT as they come in:

```
try:
  target_id = vohelper.find_client(conn, "splat")
except KeyError:
  sys.exit("Start Splat and try again")
...
for ds_name, access_url in specs:
  try:
    vohelper.send_spectrum_to(
      conn, target_id, access_url, ds_name)
  except vohelper.SAMPProxyError:
    print("  (Failed)")
```

[See get_spectra.py]

End of Part 1

We believe you now know enough to further explore PyVO and the VO on your own.

However, we've prepared a couple of extra slides on special topics. Here's some titles – let me know after the break what you'd like to do.

- Reacting to SAMP messages
- Solar System science with EPN-TAP
- Using datalink
- Multi-service TAP (Improvised)
- Walking a spectral grid (Improvised)

Higher SAMP Magic

Let's say you're debugging your pipeline and want to manually inspect "weird" objects by checking what a set of other catalogs have on them.

Plan: Write a program that other clients

- can send tables to (`table.load.votable`) and then
- when a table row is selected, computes a new table
- that's then broadcast.

Pattern for listening:

```
conn.bind_receive_notification(
  "table.highlight.row",
  self.handle_selection)
```

Doin' It With Class

Our program needs to manage quite a bit of state. At least:

- A table sent to us
- The SAMP connection

```
class VicinitySearcher(object):
  def __init__(self, client):
    self.client = client
    self.cur_table = self.cur_id = None
    self.client.bind_receive_call(
      "table.load.votable", self.load_VOTable)

  def load_VOTable(self, private_key, sender_id, msg_id, mtype,
      params, extra):
    self.cur_table = Table.read(params['url'])
    self.cur_id = params["table-id"]
    self.client.reply(msg_id,
      {"samp.status": "samp.ok", "samp.result": {}})
```

[See vicinitysearcher.py]

EPN-TAP 1: Discovery

EPN-TAP is a protocol for distributing solar system data; essentially, it's normal VO TAP plus a pre-defined table structure; the tables are always called epn_core.

Let's try an all-VO query for data on Mars. For discovery, we use GloTS:

```
glots_svc = pyvo.dal.TAPService("http://dc.g-vo.org/tap")
epn_services = glots_svc.run_sync(
  "SELECT accessurl, ivo_string_agg(table_name, '#') as tables"
  "  FROM glots.services NATURAL JOIN glots.tables"
  "  WHERE table_name LIKE '%epn_core'"
  "  GROUP BY accessurl")
```

EPN-TAP 2: Querying EPN-TAP

EPN-TAP services are queried like any other TAP service. Use a table browser to see what columns are available or check ⟨the standard⟩.

```
for svcrow in epn_services.table():
  service = pyvo.dal.TAPService(svcrow["accessurl"])
  for table_name in svcrow["tables"].split("#"):
    print("\nQuerying {} on {}".format(
      table_name, svcrow["accessurl"]))
    for row in vohelper.run_sync_resilient(service,
        "SELECT TOP 2 * FROM {} WHERE target_name='Mars'".format(
        table_name).table():
      print(row)
```

[See epnquery.py]

Of course, you want to do smarter things than print a row.

Datalink: Related Infos

Datalink is a standard for “linking” files to datasets. Think previews, extracted objects, etc.

After a data discovery query *on a datalink-enabled service*, you can use the result’s `iter_datalinks` method:

```
for dl in result.iter_datalinks():
  for link in dl: # multiple links per dataset
    print link
```

Each link has a URL, a description, and machine-readable ⟨semantics⟩. E.g., to load previews:

```
for dl in matches.iter_datalinks():
    prev_url = dl.bysemantics("#preview").next()["access_url"]
    im = Image.open(io.BytesIO(requests.get(prev_url).content))
    ...
```

[See datalink-previews.py]

Datalink: Remote processing

Datalink also lets you declare processing services. SODA is a special set of parameters applicable to astronomical images (CIRCLE, POLYGON, TIME, BAND,. . .).

Save *a lot* of time by only downloading cutouts of the object you're interested in:

```
roi = SkyCoord.from_name('Mira')
for rec in svc.run_sync(
    "SELECT access_url, access_format FROM ivoa.obscore"
    " WHERE obs_collection='HDAP'"
    "AND 1=CONTAINS(CIRCLE('ICRS', {}, {}, 0.05),"
    "s_region)".format(roi.ra.deg, roi.dec.deg)):
    processed = rec.processed(
        circle=(roi.ra.deg, roi.dec.deg, 0.05))
```

[See datalink-soda.py]

Scaling TAP Queries

TBD (Take from https://blog.g-vo.org/adql-tricks-at-mpia/)

Operating Over Spectral Grids

TBD (let's have some spectral arithmetic here – anyone in for a nice python lib for rebinning spectra and computing RMSes?)

Splitting Up Queries

It usually pays to try and optimize ADQL queries (and we'll finally write a guide on this one of these days). But sometimes you just need to partition queries; for instance, your result set otherwise becomes too large, or your query really takes that long. In the latter case, you can play with execution_duration on async jobs:

```
job = svc.run_async("...")
job.execution_duration=10000
```

This will not help you when you hit the hard match limit. In such cases, the recommended way is to use the table's primary key to partition the data; usually, that should be the column with the UCD `meta.id;meta.main`. For a rough partition, where the partition sizes may be grossly different, just figure out the maximum value of the identifier. For our light version of Gaia

Acknowledgement

- H2020-Astronomy ESFRI and Research Infrastructure Cluster (Grant Agreement number: 653477).