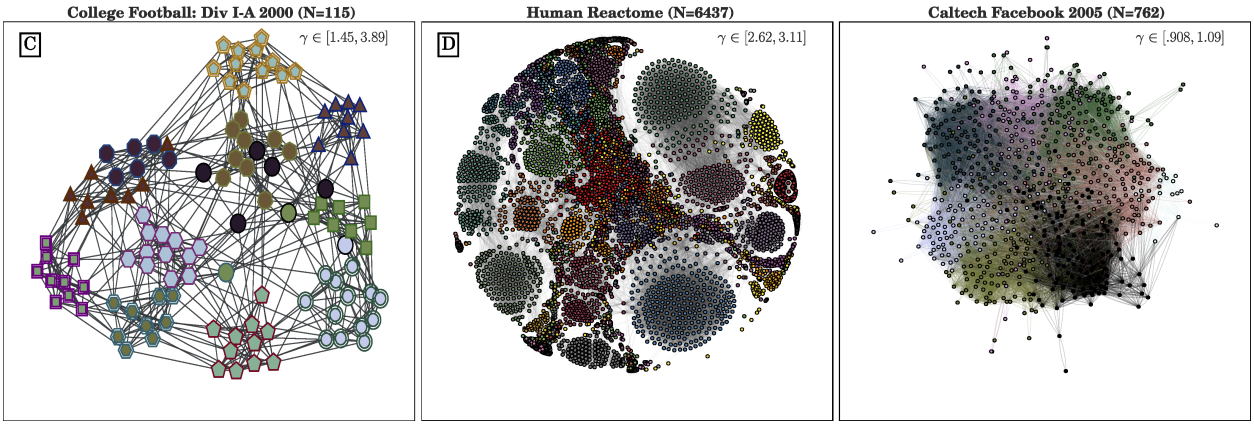
CHAMP Documentation

Release 1

William Weir

Jul 19, 2017

Contents


College Football: Div I-A 2000 (N=115)
C
$\gamma \in [1.45, 3.89]$
Human Reactome (N=6437)
D
$\gamma \in [2.62, 3.11]$
Caltech Facebook 2005 (N=762)
$\gamma \in [.908, 1.09]$

CHAPTER 1

Contents:

Background

Introduction

CHAMP (Convex Hull of Admissible Modularity Partitions) is an algorithm to find the subset of an ensembles of network partitions that are optimal in terms of modularity. Thus CHAMP is not a community detection algorithm *per say* but a method to assist in interpretation of a collection of partitions produced by ones favorite third party detection method (*e.g* Louvain, SBM, Infomap *etc.*). Instead CHAMP identifies the partitions that have a non-empty range of the resolution parameter, γ over which their modularity is larger than any other partition in the input ensemble. This is done by reformulating the problem in terms of finding the convex hull of a set of linear subspaces and solved using the pyhull implementation of the quickhull *[1]* algorithm.

CHAMP can greatly reduce the number of partitions considerable for future analyses by eliminating all partitions that are suboptimal across a given range of the resolution space. The CHAMP package also allows for visualization of the domains using the matplotlib library. Finally, the CHAMP package also includes a wrapper function for a python implementation of Louvain louvain_igraph in parallel over a range of resolutions.

For more details and results see our preprint

Modularity

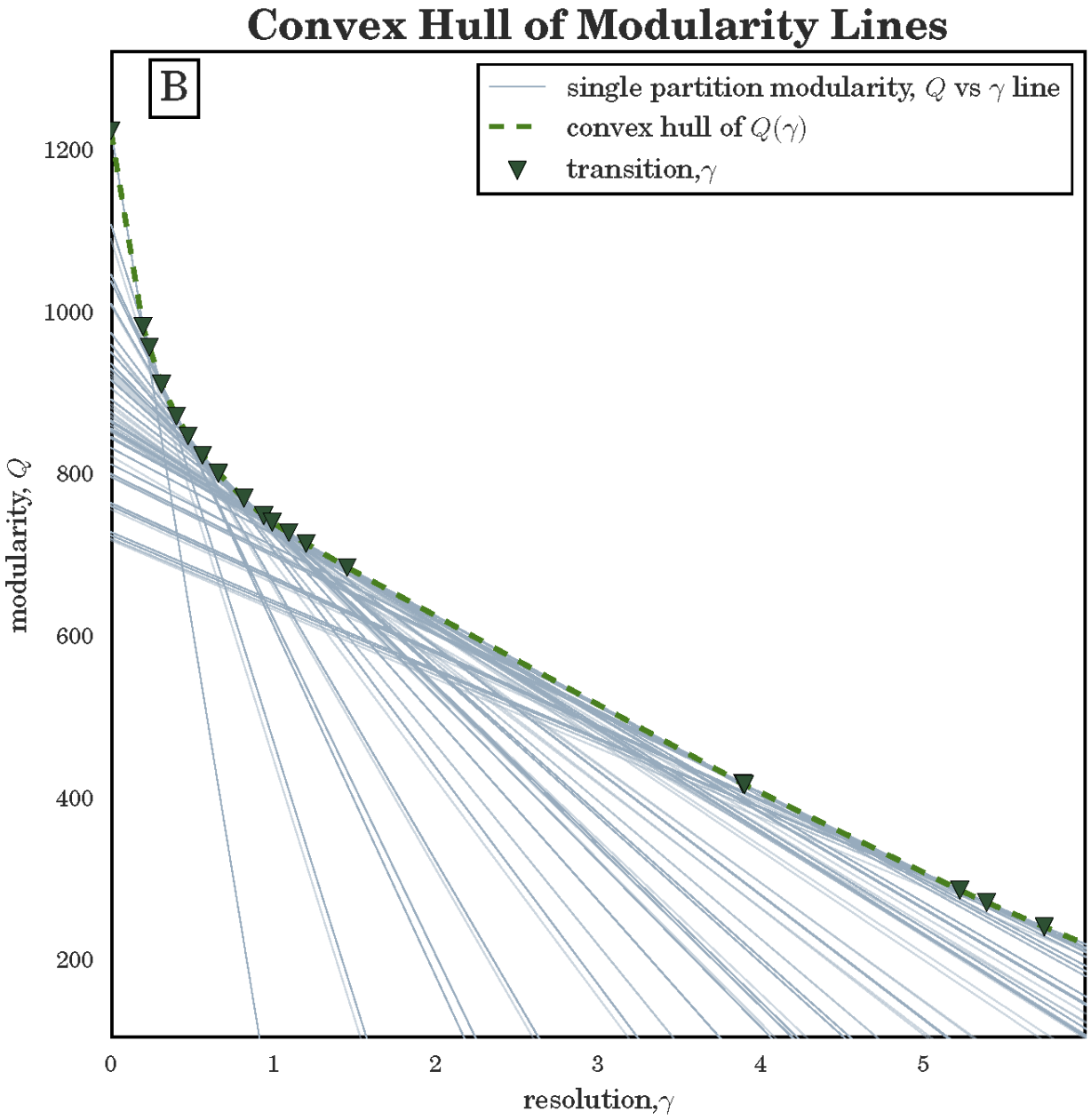
> Each partition is represented by a line in (γ, Q) domain. CHAMP find the lines that form the outer most surface.

In CHAMP, partitions are compared on the basis of modularity:

$$Q(\gamma) = \frac{1}{2m} \sum_{i,j} \left(A_{ij} - \gamma \frac{k_i k_j}{2m} \right) \delta(c_i, c_j) \,,$$

Each partition is represented by a line in the (γ, Q) space that is parameterized by two values:

$$A = \sum A_{ij}\delta(c_i, c_j)$$
Sum of edges internal to communities
$$\hat{P} = \sum P_{ij}\delta(c_i, c_j)$$
Expected number of edges internal to communities under random null model



SingleLayer_CHAMP depicts graphically the concept behind CHAMP. Most of the lines lie close to but below the outer curve. CHAMP identifies which partitions are part of the outer envelope of $Q(\gamma)$ and over which ranges of the resolution parameter, γ they are dominant.

Multilayer CHAMP

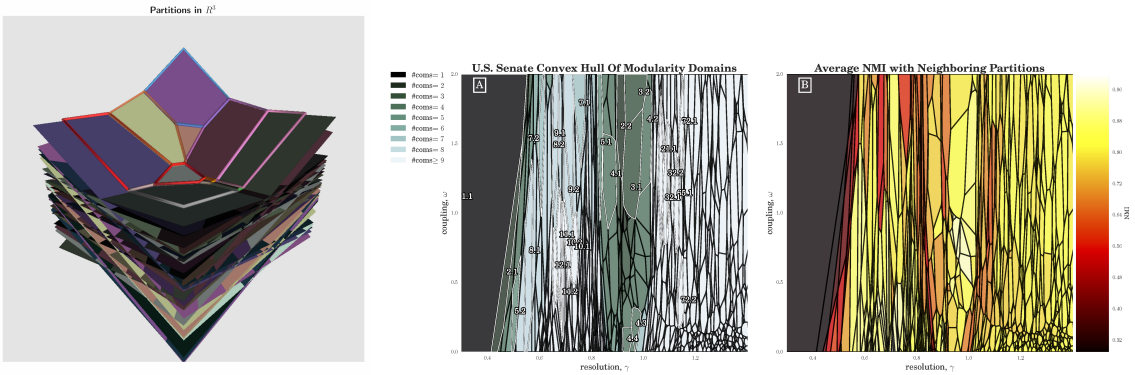
One of the strengths of modularity is that it has been extended in a principled way into a variety of network topologies in particular the multilayer context. The multilayer formulation *[2]* for modularity incorporates the interlayer connectivity of the network in the form of a second adjacency matrix C_{ij}

$$Q(\gamma) = \frac{1}{2m} \sum_{i,j} \left(A_{ij} - \gamma \frac{k_i k_j}{2m} \ + \omega C_{ij} \right) \delta(c_i, c_j) \tag{1.1}$$

Communities in this context group nodes within the layers and across the layers. The inclusion of the C_ij boost the modularity for communites that include alot interlayer links. There is an additional parameter, ω that tunes how much weight these interlink ties contribute to the modularity. With the additional parameter, each partitions can be represented in the (γ, ω, Q) space by three coefficients. The two in equation *single layer coefficients* and :

$$C = \sum C_{ij} \delta(c_i, c_j)$$

Sum of interlayer edges internal to communities



In the multilayer case, we look for the planes that define the intersection of the area above all of the planes as depicted in *3D Planes*. These domains are now 2D polygons in the (γ, ω) space as shown in *Domains*.

References

- genindex
- search

Running CHAMP

CHAMP uses the quick hull algorithm to find the intersection of the space above all of the planes representing the input set of partitions as shown in *Single Layer* and *Multilayer*. There are many tools

Starting from Partitions

If the partitions were generated using a modularity based community detection method, it's better to calculate the coefficients while optimizing the communities and feed these into CHAMP directly. This is especially true, if the community detection is being performed in parallel. However, if the partitions were generated using some other form of community detection algorithm, we provide a method to compute these coefficients directly and allow for parallization of this process on supported machines.

champ.champ_functions.**create_coefarray_from_partitions**(*partition_array*, *A_mat*, *P_mat*, *C_mat=None*, *nprocesses=0*)

Parameters

- **partition_array** – Each row is one of M partitions of the network with N nodes. Community labels must be hashable.
- **A_mat** – Interlayer (single layer) adjacency matrix
- **P_mat** – Matrix representing null model of connectivity (i.e configuration model - $\frac{k_i k_j}{2m}$
- **C_mat** – Optional matrix representing interlayer connectivity
- **nprocesses** (*int*) – Optional number of processes to use (0 or 1 for single core)

Returns size $M \times$ Dim array of coefficients for each partition. Dim can be 2 (single layer) or 3 (multilayer)

Coeffients from Partitions Example

Starting from Partition Coefficients

In practice, it is often easier to calculate the coefficients while running performing the community detection to generate the input ensemble of partitions, especially if these partitions are being generated in parallel. If these have been generated already, one can apply CHAMP directly via the following call. The same command is used in both the Single Layer and Multilayer context, with the output determined automatically by the number of coefficients supplied in the input array.

champ.champ_functions.**get_intersection**(*coef_array*, *max_pt=None*)

Calculate the intersection of the halfspaces (planes) that form the convex hull

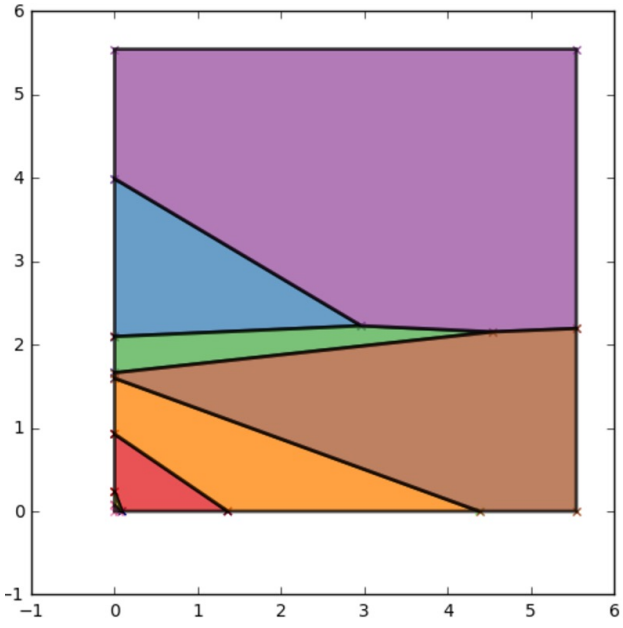
Parameters

- **coef_array** (*array*) – NxM array of M coefficients across each row representing N partitions
- **max_pt** (*(float,float)*) – Upper bound for the domains (in the xy plane). This will restrict the convex hull to be within the specified range of gamma/omega (such as the range of parameters originally searched using Louvain).

Returns dictionary mapping the index of the elements in the convex hull to the points defining the boundary of the domain

Applying CHAMP to Coefficients Array Example

```
import champ
import matplotlib.pyplot as plt

#generate random coefficent matrices
coeffs=champ.get_random_halfspaces(100,dim=3)
ind_2_dom=champ.get_intersection(coeffs)


ax=champ.plot_2d_domains(ind_2_dom)
plt.show()
```



Output[1] :

- genindex
- search

Visualizing Results

The CHAMP package offers a number of ways to visualize the results for both single layer and multilayer networks.

Single Layer Plots

`champ.`**`plot_line_coefficients`**(*coef_array*, *ax=None*, *colors=None*)

Plot an array of coefficients (lines) in 2D plane. Each line is drawn from y-intercept to x-intercept.

Parameters

- **`coef_array`** (*`np.array`*) – $N \times 2$ array of coefficients representing lines.
- **`ax`** (*`matplotlib.Axes`*) – optional matplotlib ax to draw the figure on.
- **`colors`** (*`[list,string]`*) – optional list of colors (or single color) to draw lines

Returns matplotlib ax on which the plot is draw

`champ.`**`plot_single_layer_modularity_domains`**(*ind_2_domains*, *ax=None*, *colors=None*, *labels=None*)

Plot the piece-wise linear curve for CHAMP of single layer partitions

Parameters

- **`ind_2_domains`** (*`{ ind: [ np.array(gam_0x, gam_0y), np.array(gam_1x, gam_1y) ] ,..}`*) – dictionary mapping partition index to domain of dominance
- **`ax`** – Matplotlib Axes object to draw the graph on

[1] Note that actual output might differ due to random seeding.

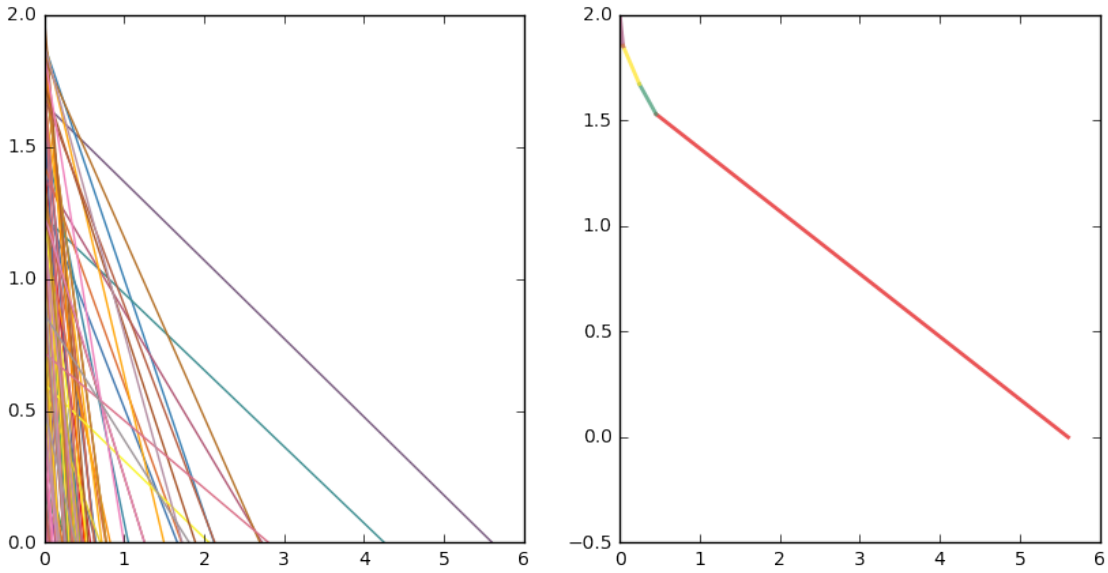
- **colors** – Either a single color or list of colors with same length as number of domains

Returns ax Reference to the ax on which plot is drawn.

Single Layer Example

```
import champ
import matplotlib.pyplot as plt


#generate random coefficent matrices
coeffs=champ.get_random_halfspaces(100,dim=2)
ind_2_dom=champ.get_intersection(coeffs)

plt.close()
f,axarray=plt.subplots(1,2,figsize=(10,5))
champ.plot_line_coefficients(coeffs,axarray[0])
champ.plot_single_layer_modularity(ind_2_dom,axarray[1])
plt.show()
```

Output[1] :



Heatmap Example

In most cases CHAMP reduces the number of considerable parttions drastically. So much so that it is feasible to calculate the similarity between all pairs of paritions and visualize them ordered by their domains. The easier way to do this is to wrap the input partitions into a `louvain_ext.PartitionEnsemble` object . Creation of the *PartitionEnsemble* object automatically applies CHAMP and allows access to the dominant partitions.

`champ.`**plot_similarity_heatmap_single_layer**(*partitions*, *index_2_domain*, *sim_mat=None*, *ax=None*, *cmap=None*, *title=None*)

Parameters

- **partitions** –
- **index_2_domain** –

[1] Note that actual output might differ due to random seeding.

- **sim_mat** –
- **ax** (*Matplotlib.Axes*) – Axes to draw the figure on. New figure created if not supplied.
- **cmap** (*Matplotlib.colors.Colormap*) – Color mapping. Default is plasma
- **title** (*boolean or string*) – True if add generic title, or string of title to add

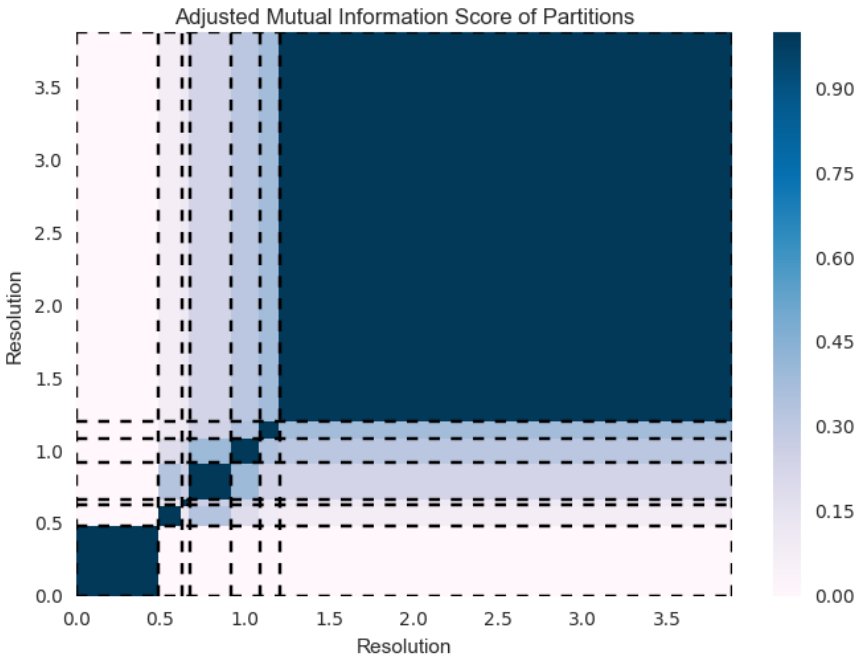
Returns axis drawn on , computed similarity matrix

Return type matplolib.Axes,np.array

```
import champ
from champ import louvain_ext
import igraph as ig
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
test_graph=ig.Graph.Random_Bipartite(n1=100,n2=100,p=.1)

#parallelized wrapper
ensemb=louvain_ext.parallel_louvain(test_graph,
                                    numruns=300,start=0,fin=4,
                                    numprocesses=2,
                                    progress=True)



plt.close()
a,nmi=champ.plot_similarity_heatmap_single_layer(ensemb.partitions,ensemb.ind2doms,
↪title=True)
plt.show()
```

Output[1] :

```
Run 0 at gamma = 0.000. Return time: 0.0275
Run 100 at gamma = 1.333. Return time: 0.0716
Run 200 at gamma = 2.667. Return time: 0.0717
```



Multiayer Plots

In the multilayer case, each domain is a convex ploygon in the (γ, ω) plane.

champ.**plot_2d_domains**(*ind_2_domains*, *ax=None*, *col=None*, *close=False*, *widths=None*, *label=False*)

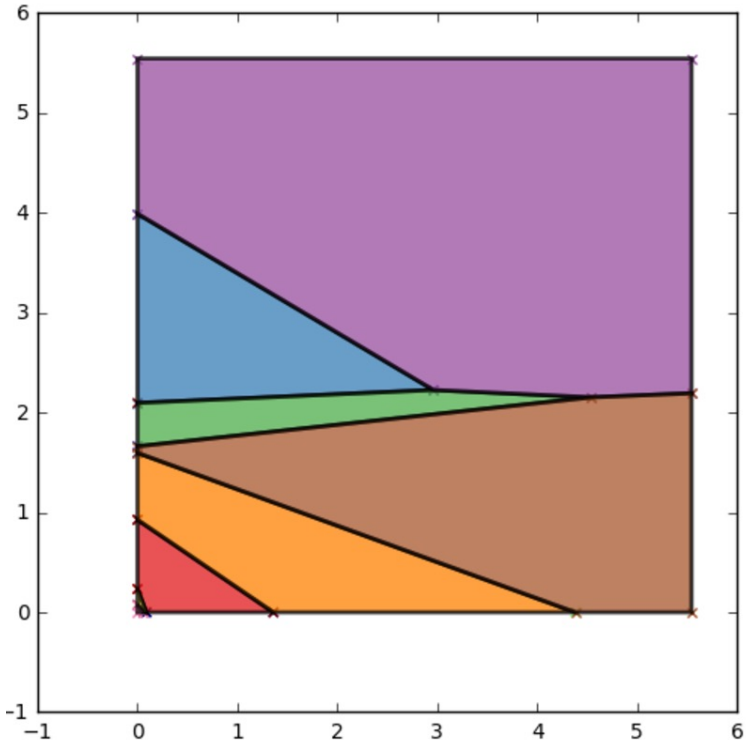
Parameters

- **ind_2_domains** –
- **ax** –
- **col** –
- **close** –
- **widths** –
- **label** –

Returns

Multilayer Example

```
import champ
import matplotlib.pyplot as plt

#generate random coefficent matrices
coeffs=champ.get_random_halfspaces(100,dim=3)
ind_2_dom=champ.get_intersection(coeffs)


ax=champ.plot_2d_domains(ind_2_dom)
plt.show()
```



Output[1] :

- genindex
- search

Louvain Parallel Extension

CHAMP can be used with partitions generated by any community detection algorithm. One of the most popular and fast algorithms is known as Louvain *[1]* . We provide an extension to the python package developed by Vincent Traag, louvain_igraph *[2]* to run Louvain in parallel, while calculating the coefficients necessary for CHAMP. Currently, this extension only support single-layer network. The random seed is set within each parallel process to ensure that the results are stochastic over each run. In general this is desireable because Louvain uses a greedy optimization schema that finds *local* optima.

`champ.louvain_ext.`**`parallel_louvain`**(*graph*, *start=0*, *fin=1*, *numruns=200*, *maxpt=None*, *numprocesses=None*, *attribute=None*, *weight=None*, *node_subset=None*, *progress=False*)

Generates arguments for parallel function call of louvain on graph

Parameters

- **`graph`** – igraph object to run Louvain on
- **`start`** – beginning of range of resolution parameter γ . Default is 0.
- **`fin`** – end of range of resolution parameter γ. Default is 1.
- **`numruns`** – number of intervals to divide resolution parameter, γ range into
- **`maxpt`** (*int*) – Cutoff off resolution for domains when applying CHAMP. Default is None
- **`numprocesses`** – the number of processes to spawn. Default is number of CPUs.

- **weight** – If True will use 'weight' attribute of edges in runnning Louvain and calculating modularity.
- **node_subset** – Optionally list of indices or attributes of nodes to keep while partitioning
- **attribute** – Which attribute to filter on if node_subset is supplied. If None, node subset is assumed to be node indices.
- **progress** – Print progress in parallel execution

Returns PartitionEnsemble of all partitions identified.

We also have created a convenient class for managing and merging groups of partitions called *champ.louvain_ext.PartitionEnsemble*. This class stores the partitions in membership vector form (i.e. a list of N community assignments), as well as the coefficients for the partitions. As part of its class methods, the PartitionEnsemble is able to apply CHAMP to its own partitions and store their domains.

class champ.louvain_ext.**PartitionEnsemble**(*graph=None*, *listofparts=None*, *name='unnamed_graph'*, *maxpt=None*)

Group of partitions of a graph stored in membership vector format

The attribute for each partition is stored in an array and can be indexed

Variables

- **graph** – The graph associated with this PartitionEnsemble. Each ensemble can only have a single graph and the nodes on the graph must be orded the same as each of the membership vectors.
- **partitions** – List of membership vectors for each partition
- **int_edges** – Number of edges internal to the communities
- **exp_edges** – Number of expected edges (based on configuration model)
- **resoltions** – If partitions were idenitfied with Louvain, what resolution were they identified at (otherwise None)
- **orig_mods** – Modularity of partition at the resolution it was identified at if Louvain was used (otherwise None).
- **numparts** – number of partitions
- **ind2doms** – Maps index of dominant partitions to boundary points of their dominant domains

add_partitions(*partitions*, *maxpt=None*)

Add additional partitions to the PartitionEnsemble object

Parameters **partitions** (*dict, list*) – list of partitions to add to the PartitionEnsemble

apply_CHAMP(*maxpt=None*)

Apply CHAMP to the partition ensemble.

Parameters **maxpt** (*int*) – maximum domain threshhold for included partition. I.e partitions with a domain greater than maxpt will not be included in pruned set

calc_expected_edges(*memvec*)

Uses igraph Vertex Clustering representation to calculate expected edges. see `louvain_ext.get_expected_edges()`

Parameters **memvec** (*list*) – membership vector for which to calculate the expected edges

Returns expected edges under null

Return type float

calc_internal_edges(*memvec*)
Uses igraph Vertex Clustering representation to calculate internal edges. see `louvain_ext.get_expected_edges()`

Parameters **memvec** (*list*) – membership vector for which to calculate the internal edges.

Returns

get_CHAMP_indices()
Get the indices of the partitions that form the pruned set after application of CHAMP

Returns list of indices of partitions that are included in the prune set sorted by their domains of dominance

Return type list

get_CHAMP_partitions()
Return the subset of partitions that form the outer envelop. :return: List of partitions in membership vector form of the paritions :rtype: list

get_adjacency()
Calc adjacency representation if it exists

Returns self.adjacency

get_coefficient_array()
Create array of coefficents for each partition

Returns np.array with coefficents for each of the partions

get_partition_dictionary(*ind=None*)
Get dictionary representation of partitions with the following keys:

'partition','resolution','orig_mod','int_edges','exp_edges'

Parameters **ind** (*int, list*) – optional indices of partitions to return. if not supplied all partitions will be returned.

Returns list of dictionaries

merge_ensemble(*otherEnsemble*)
Combine to PartitionEnsembles. Checks for concordance in the number of vertices. Assumes that internal ordering on the graph nodes for each is the same.

Parameters **otherEnsemble** – otherEnsemble to merge

Returns new PartitionEnsemble with merged set of partitions

static open(*filename*)
Loads pickled PartitionEnsemble from file.

Parameters **file** – filename of pickled PartitionEnsemble Object

Returns writes over current instance and returns the reference

save(*filename=None*)
Use pickle to dump representation to compressed file

Parameters **filename** –

Partition Ensemble Example

We use igraph to generate a random ER graph, call louvain in parallel, and apply CHAMP to the ensemble.

```
import champ
from champ import louvain_ext

import igraph as ig
import tempfile
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
test_graph=ig.Graph.Erdos_Renyi(500,p=.05)
#Create temporary file for calling louvain
tfile=tempfile.NamedTemporaryFile('wb')
test_graph.write_graphmlz(tfile.name)

#non-parallelized wrapper
ens1=louvain_ext.run_louvain(tfile.name,nruns=30,gamma=1)

#parallelized wrapper
ens2=louvain_ext.parallel_louvain(test_graph,
                                  numruns=10,
                                  numprocesses=2,
                                  progress=True)

#Output as gzipped file
ens2.save("test_esemble_file.gz")



#Apply Champ to Coefficients
coeffs2=ens2.get_coefficient_array()
ind2dom2=champ.get_intersection(coeffs2)
plt.close()
ax=champ.plot_single_layer_modularity(ind2dom2)
plt.show()
```

References

- genindex
- search

CHAPTER 2

Download and Installation:

The CHAMP module is hosted on PyPi. The easiest way to install is via the pip command:

```
pip install champ
```

For installation from source, the latest version of champ can be downloaded from GitHub:

https://github.com/wweir827/CHAMP

For basic installation:

```
python setup.py install
```

Dependencies

Most of the dependencies for CHAMP are fairly standard tools for data analysis in Python, with the exception of louvain_igraph. They include :

- NumPy
- sklearn
- igraph
- matplotlib
- louvain

CHAPTER 3

Citation

Please cite:

bibtex

- genindex
- search

Bibliography

[1] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Trans. Math. Softw.*, 22(4):469483, Dec 1996. doi:10.1145/235815.235821.

[2] P J Mucha, T Richardson, K Macon, and M A Porter. Community structure in time-dependent, multiscale, and multiplex networks. *Science*, May 2010.

[1] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, pages P10008, 2008.

[2] Vincent Traag. Louvain igraph. http://github.com/vtraag/louvain-igraph.

[1] William H. Weir, Scott Emmons, Ryan Gibson, Dane Taylor, and Peter J Mucha. Post-processing partitions to identify domains of modularity optimization. *arXiv.org*, 2017. arXiv:1706.03675.

Index