CHAMP Documentation

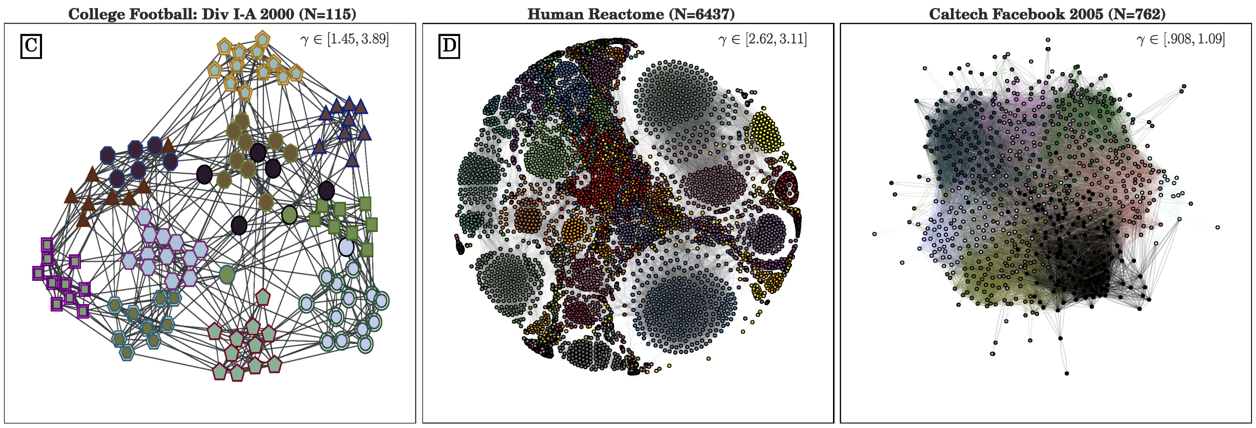
Release 1

William Weir

May 22, 2019

Contents

A modularity based tool for screening a set of partitions.


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)
$\gamma \in [.908, 1.09]$

The CHAMP python package provides two levels of functionality:

- Identifying the subset of partitions from a group of partitions (regardless of how they were discovered) with optimal modularity. See *Running CHAMP*.
- Parallelized implementation of modularity based community detection method, louvain with efficient filtering (*ala* CHAMP), management, and storage of the generated partitions. See *Louvain Parallel Extension* .

CHAPTER 1

Contents:

1.1 Background

1.1.1 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 manuscript

1.1.2 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:

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

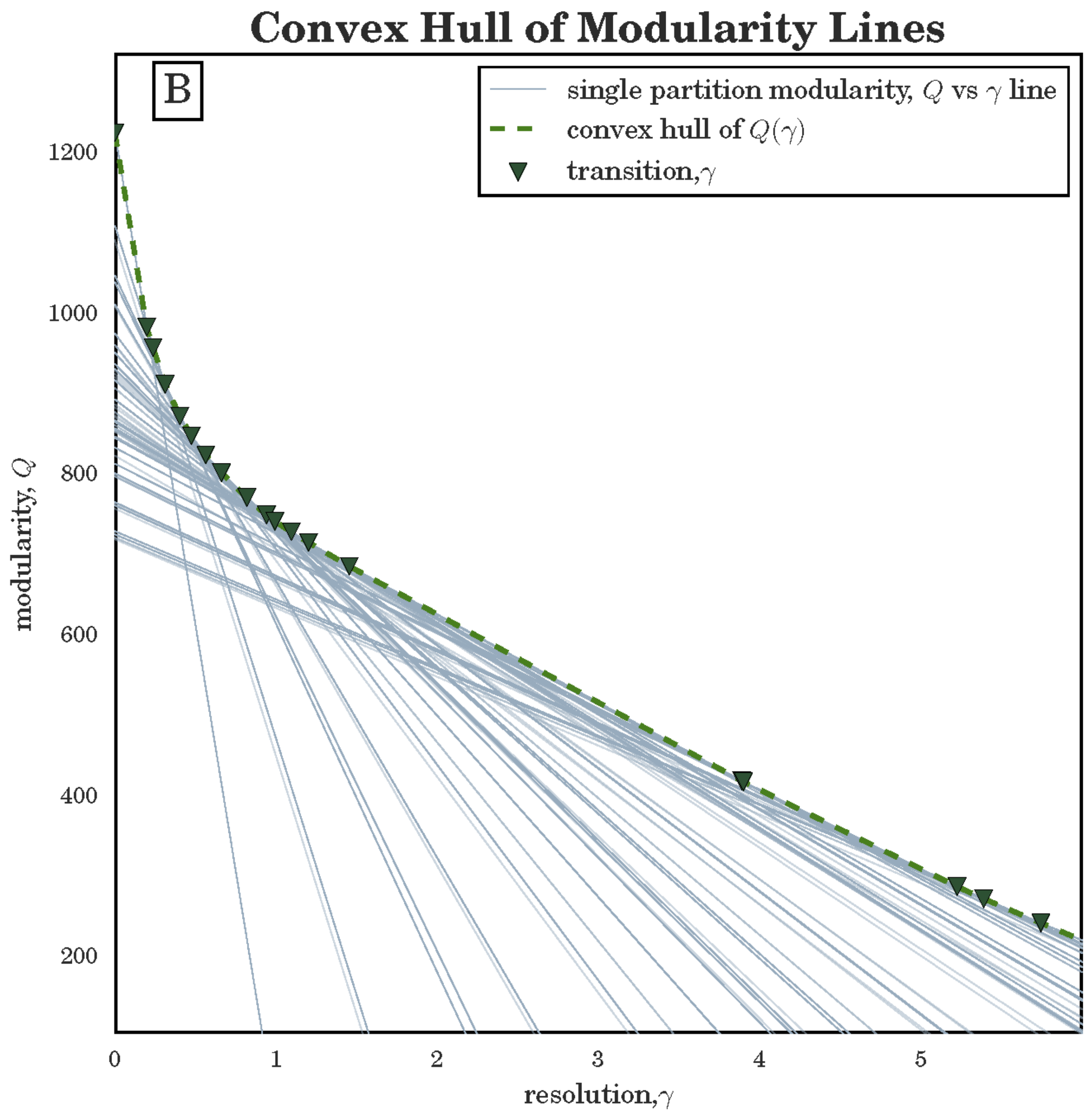
Each partition is represented by a line in the (γ, Q) space that is parameterized by two values:

$$A = \sum A_{ij}\delta(c_i, c_j)$$

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.

1.1.3 Multilayer CHAMP

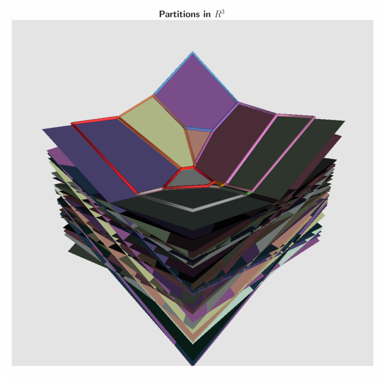
One of the strengths of modularity is that it has been extended in a principled way into a variety of network topologies in particular the multilayer context. The multilayer formulation [2] for modularity incorporates the interlayer connectivity of the network in the form of a second adjacency matrix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

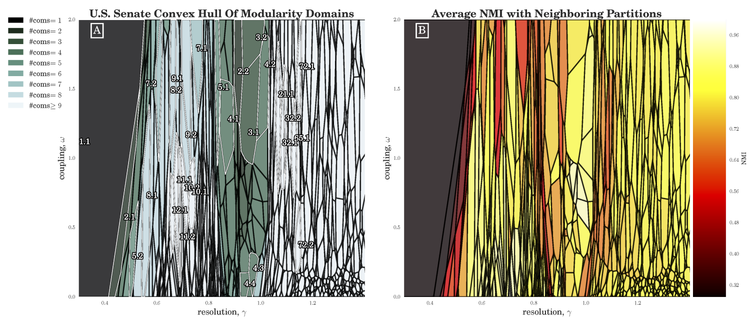
$$Q(\gamma) = \frac{1}{2m} \sum_{i,j} \left(A_{ij} - \gamma \frac{k_i k_j}{2m} + \omega C_{ij} \right) \delta(c_i, c_j) \tag{1.1}$$

Communities in this context group nodes within the layers and across the layers. The inclusion of the $C_i j$ 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

1.2 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 great community detection tools available (both in python and other langauges) that can be used to generate the starting collection of partitions which CHAMP can be applied to. We have incorporated a python version of the louvain algorithm into CHAMP to identify partitions on the basis of modularity maximization which CHAMP can then be applied to. See *Louvain Extension*.

Below we detail running of CHAMP for two scenarios:

1. Starting from an ensemble of partitions (without the corresponding partition coefficients for calculating the convex hull).
2. Starting with the partitions and the coefficients precalculated.

1.2.1 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 parallelization 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

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


rand_er_graph=ig.Graph.Erdos_Renyi(n=1000,p=.05)

for i in range(100):
    ncoms=np.random.choice(range(1,30),size=1)[0]
    if i==0:
        rand_partitions=np.random.choice(range(ncoms),replace=True,size=(1,1000))
    else:
        rand_partitions=np.concatenate([rand_partitions,np.random.choice(range(ncoms),
↪replace=True,size=(1,1000))])

print(rand_partitions.shape)

#get the adjacency of ER graph
A_mat=np.array(rand_er_graph.get_adjacency().data)
#create null model matrix
P_mat=np.outer(rand_er_graph.degree(),rand_er_graph.degree())

## Create the array of coefficients for the partitions
coeff_array=champ.champ_functions.create_coefarray_from_partitions(A_mat=A_mat,
                                                                   P_mat=P_mat,
                                                                   partition_
↪array=rand_partitions)
```

(continues on next page)

(continued from previous page)

```
#Calculate the intersection of all of the halfspaces.  These are the partitions that␣
↪form the CHAMP set.
ind2doms=champ.champ_functions.get_intersection(coef_array=coeff_array)
print(ind2doms)
```

1.2.2 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) or 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).

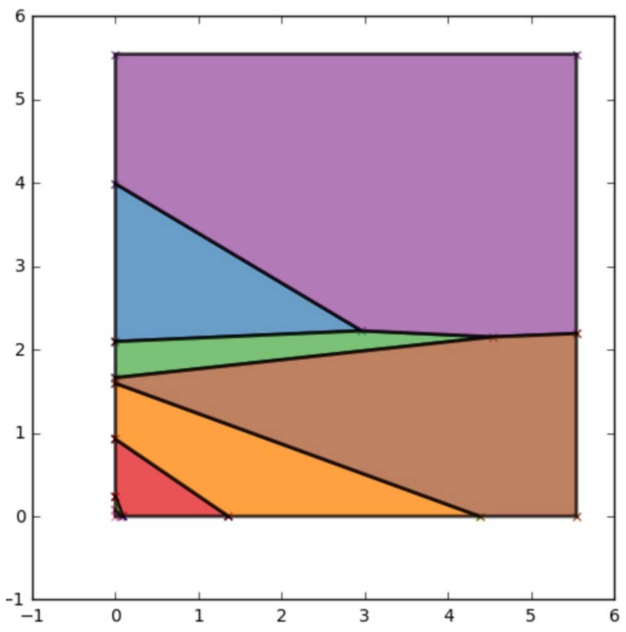
Returns dictionary mapping the index of the elements in the convex hull to the points defining the boundary of the domain

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] :

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



- genindex
- search

1.3 Visualizing Results

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

1.3.1 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

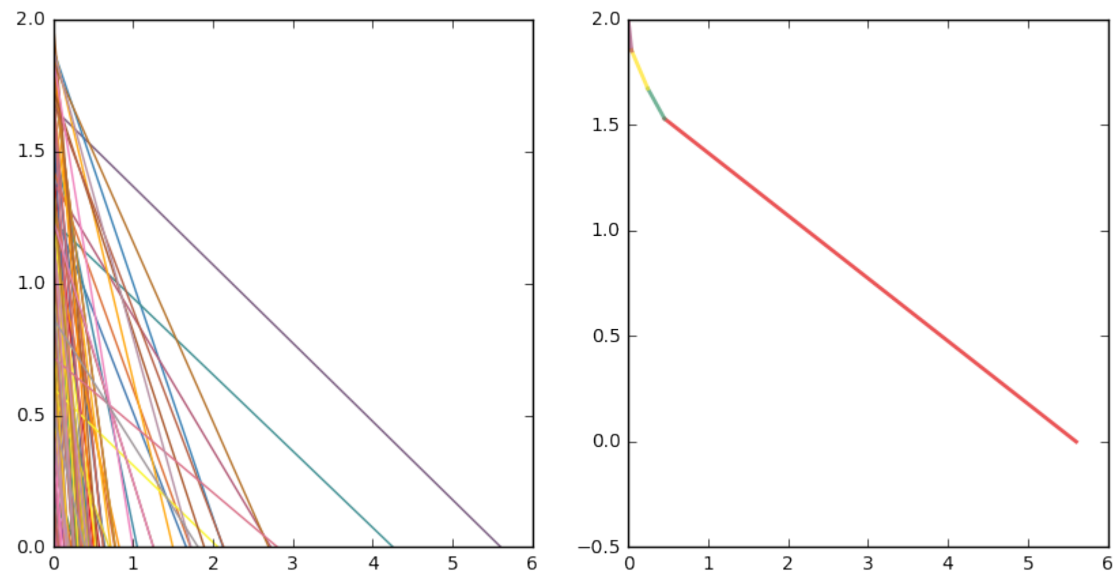
- **`ind_2_domains`** (*`{ ind: [ np.array(gam_0x, gam_0y), np.array(gam_1x, gam_1y) ] ,..}`*) – dictionary mapping partition index to domain of dominance
- **`ax`** – Matplotlib Axes object to draw the graph on
- **`colors`** – Either a single color or list of colors with same length as number of domains

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*, *partitions_other=None*, *index_2_dom_other=None*, *sim_mat=None*, *ax=None*, *cmap=None*, *title=None*)

Parameters

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

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

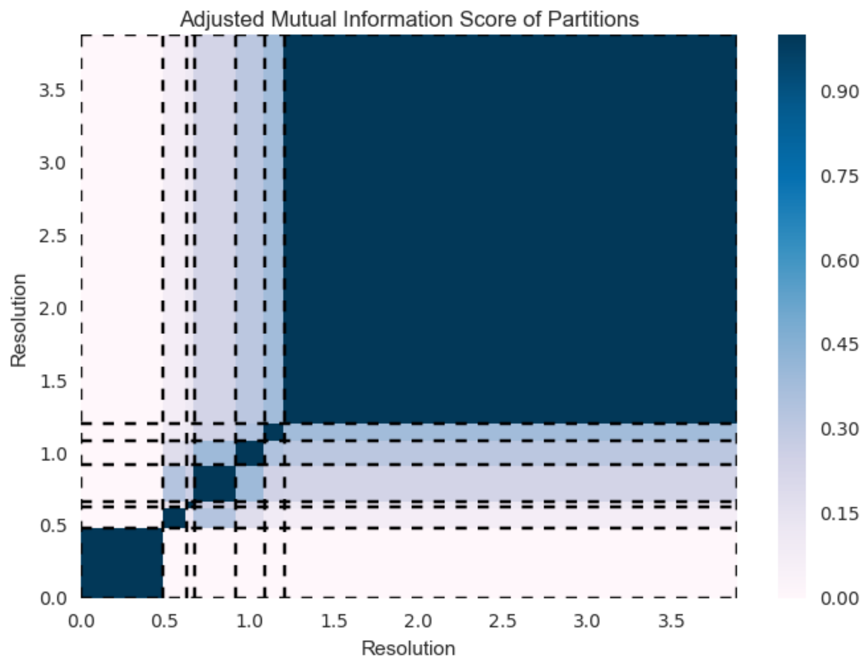
Returns axis drawn on , computed similarity matrix

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
```



1.3.2 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

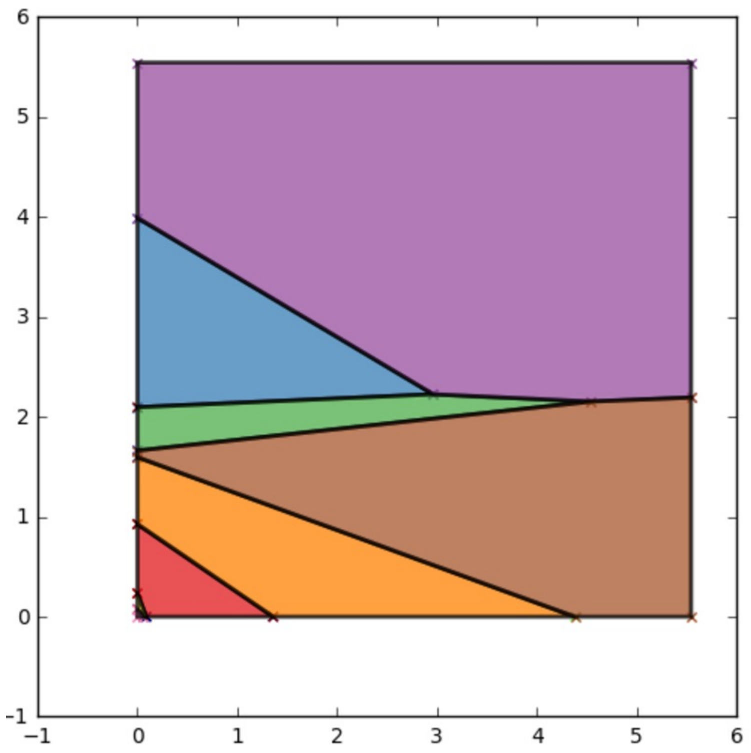
- **`ind_2_domains`** –
- **`ax`** –
- **`col`** –
- **`close`** –
- **`widths`** –
- **`label`** –

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

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

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 every *n* iterations.

Returns PartitionEnsemble of all partitions identified.

We also have created a convenient class for managing and merging groups of partitions called *`champ.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 constructor, the PartitionEnsemble applies CHAMP to all of its partitions, and stores the domains of dominance.

`class` `champ.`**`PartitionEnsemble`**(*graph=None, interlayer_graph=None, layer_vec=None, listofparts=None, name='unnamed_graph', maxpt=None, min_com_size=5*)

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. In the case of mutlilayer, graph should be a sinlge igraph containing all of the interlayer connections.
- **`interlayer_graph`** – For multilayer graph. igraph.Graph that contains all of the interlayer connections
- **`partitions`** – of membership vectors for each partition. If h5py is set this is a dummy variable that allows access to the file, but never actually hold the array of paritions.
- **`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
- **`ncoms`** – List with number of communities for each partition
- **`min_com_size`** – How many nodes must be in a community for it to count towards the number of communities. This eliminates very small or unstable communities. Default is 5
- **`unique_partition_indices`** – The indices of the paritions that represent unique coefficients. This will be a subset of all the partitions.
- ***`hdf5_file`*** – Current hdf5_file. If not None, this serves as the default location for loading and writing partitions to, as well as the default location for saving.

- ***twin_partitions*** – We define twin partitions as those that have the same coefficients, but are actually different partitions. This and the unique_partition_indices are only calculated on demand which can take some time.

1.4.1 Partition Ensemble Example

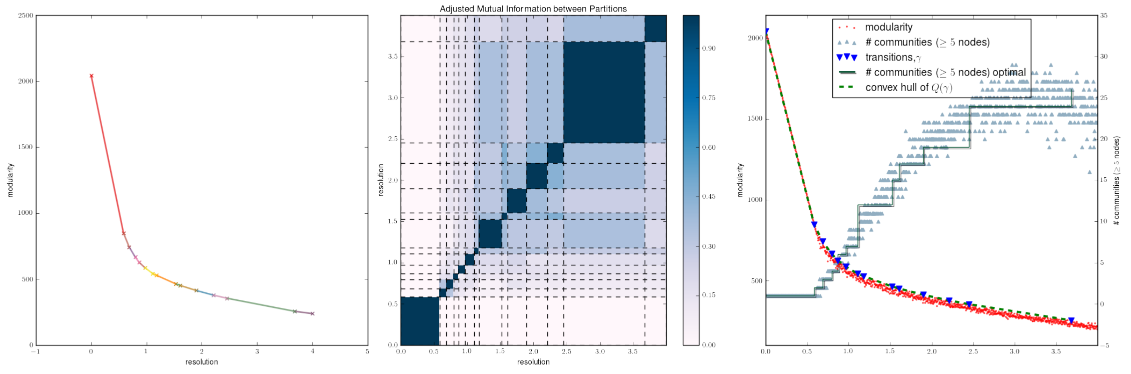
We use igraph to generate a random ER graph, call louvain in parallel, and apply CHAMP to the ensemble. This example took approximately 1 minute to run on 2 cores. We can use the PartitionEnsemble object to check whether all of the coefficients are unique, and to check whether all of the partitions themselve are unique (it is possible for two unique partitions to give rise to the same coefficients).

```
import champ
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=champ.run_louvain(tfile.name,nruns=5,gamma=1)

# parallelized wrapper
test_graph2=ig.Graph.Random_Bipartite(n1=100,n2=100,p=.1)
ens2=champ.parallel_louvain(test_graph2,
                                    numruns=1000,start=0,fin=4,maxpt=4,
                                    numprocesses=2,
                                    progress=True)

print ("%d of %d partitions are unique"%(len(ens2.unique_partition_indices),ens2.
↪numparts))
print ("%d of %d partitions after application of CHAMP"%(len(ens2.ind2doms),ens2.
↪numparts))
print ("Number of twin sets:")
print (ens2.twin_partitions)
#plot both of these
plt.close()
f,a=plt.subplots(1,3,figsize=(21,7))
a1,a2,a3=a
champ.plot_single_layer_modularity_domains(ens2.ind2doms,ax=a1,labels=True)
champ.plot_similarity_heatmap_single_layer(ens2.partitions,ens2.ind2doms,ax=a2,
↪title=True)

#PartitionEnsemble has method to plot downsampled summary of all partitions
#with optmal transitions and number of communities overlayed.

ens2.plot_modularity_mapping(ax=a3,no_tex=False)
plt.tight_layout()
plt.show()
```

Output:

```
Run 0 at gamma = 0.000. Return time: 0.0264
Run 200 at gamma = 0.800. Return time: 0.0621
Run 100 at gamma = 0.400. Return time: 0.0589
Run 300 at gamma = 1.200. Return time: 0.0660
Run 400 at gamma = 1.600. Return time: 0.0963
Run 500 at gamma = 2.000. Return time: 0.0970
Run 600 at gamma = 2.400. Return time: 0.0828
Run 700 at gamma = 2.800. Return time: 0.0769
Run 800 at gamma = 3.200. Return time: 0.0763
Run 900 at gamma = 3.600. Return time: 0.0818
847 of 1000 partitions are unique
11 of 1000 partitions after application of CHAMP
Number of twin sets:
    []
```

We see that there are a number of identical partitions here, there are no twin partitions . That is different partitions with the same coefficents.



PartitionEnsemble.**apply_CHAMP**(*subset=None*, *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
- **subset** – subset of partitions to apply CHAMP to. This is useful in merging two sets because we only need to apply champ to the combination of the two

PartitionEnsemble.**plot_modularity_mapping**(*ax=None*, *downsample=2000*, *champ_only=False*, *legend=True*, *no_tex=True*)

Plot a scatter of the original modularity vs gamma with the modularity envelope super imposed. Along with communities vs γ on a twin axis. If no orig_mod values are stored in the ensemble, just the modularity envelope is plotted. Depending on the backend used to render plot the latex in the labels can cause error. If you are getting RunTime errors when showing or saving the plot, try setting no_tex=True

Parameters

- **ax** (*matplotlib.Axes*) – axes to draw the figure on.

- **champ_only** (*bool*) – Only plot the modularity envelop represented by the CHAMP identified subset.
- **downsample** (*int*) – for large number of runs, we down sample the scatter for the number of communities and the original partition set. Default is 2000 randomly selected partitions.
- **legend** (*bool*) – Add legend to the figure. Default is true
- **no_tex** (*bool*) – Use latex in the legends. Default is true. If error is thrown on plotting try setting this to false.

Returns axes drawn upon

Return type matplotlib.Axes

`PartitionEnsemble.`**`get_unique_partition_indices`**(*reindex=True*)

This returns the indices for the partitions who are unique. This could be larger than the indices for the unique coeficient since multiple partitions can give rise to the same coefficient. In practice this has been very rare. This function can take sometime for larger network with many partitions since it reindex the partitions labels to ensure they aren't permutations of each other.

Parameters **reindex** – if True, will reindex partitions that it is comparing to ensure they are unique under permutation.

Returns list of twin partition (can be empty), list of indicies of unique partitions.

Return type list,np.array

`PartitionEnsemble.`**`twin_partitions`**

We define twin partitions as those that have the same coefficients but are different partitions. To find these we look for the diffence in the partitions with the same coefficients.

Returns List of groups of the indices of partitions that have the same coefficient but are non-identical.

Return type list of list (possibly empty if no twins)

1.4.2 Creating and Managing Large Partition Sets

For large sets of partitions on larger networks, loading the entire set of partitions each time you want to access the data can take a fair amount of time. Having to load all of the partitions defeats the purpose of using CHAMP to find the optimal subset. As such we have equiped `champ.louvain_ext.PartitionEnsemble` objects with the ability to write to and access hdf5 files. Instead of loading the entire set of partitions each time, only the coefficients, the domains of dominance, the underlying graph information is loaded, vastly reducing the loading time and the memory required for large numbers of runs. Each individual partitions (or all partitions at once) can be access from file only if needed, as the example below illustrates:

1.4.3 Saving and Loading PartitionEnsembles

```
import champ
import igraph as ig
import numpy as np

np.random.seed(0)
test_graph=ig.Graph.Erdos_Renyi(n=200,p=.1)
times={}
run_nums=[100]

```

(continues on next page)

(continued from previous page)

```
#saving ensemble
ensemble=champ.parallel_louvain(test_graph,numprocesses=2,numruns=200,start=0,fin=4,
↪maxpt=4,progress=False)
print "Ensemble 1, Optimal subset is %d of %d partitions"%(len(ensemble.ind2doms),
↪ensemble.numparts)
ensemble.save("ensemble1.hdf5",hdf5=True)

#openning created file
ensemble2=champ.PartitionEnsemble().open("ensemble1.hdf5")
#partitions is an internal class the handles access
print ensemble2.partitions
#When sliced a numpy.array is returned
print "ensemble2.partitions[38,:10]:\n\t",ensemble2.partitions[38:40,:10]

#You can still add partitions as normal.  These are automatically
#added to the hdf5 file
ensemble2add=champ.parallel_louvain(test_graph,numprocesses=2,numruns=100,start=0,
↪fin=4,maxpt=4,progress=False)
ensemble2.add_partitions(ensemble2add.get_partition_dictionary())
print "ensemble 2 has %d of %d partitions dominant"  %(len(ensemble2.ind2doms),
↪ensemble2.numparts)

#When open is called, the created EnsemblePartition assumes all
#the state variables of saved EnsemblePartition, including default save file.
#If save() is called, it will overwrite the old ensemble file
print "ensemble2 default hdf5 file: ",ensemble2.hdf5_file
```

Output:

```
Ensemble 1, Optimal subset is 11 of 200 partitions
200 partitions saved on ensemble1.hdf5
ensemble2.partitions[38,:10]:

        [[1 0 1 0 0 0 0 0 1 1]
        [1 1 2 0 1 0 2 1 2 1]]

ensemble 2 has 12 of 300 partitions dominant
```

You can see with the above example that CHAMP is reapplied everytime new partitions are added to the PartitionEnsemble instance. In addition, whole PartitionEnsemble objects can be merged together in a similar fashion. You can either have a new PartitionEnsemble generated, containing the contents of both the inputs. Or one partition can be merged into the other one. It automatically uses the larger of the two to merge into. In addition, if either PartitionEnsemble has an associated hdf5 file, the merger will just expand the contents of the file and keep the same PartitionEnsemble object.

`PartitionEnsemble.`**`add_partitions`**(*partitions*, *maxpt=None*)

Add additional partitions to the PartitionEnsemble object. Also adds the number of communities for each. In the case where PartitionEnsemble was openned from a file, we just appended these and the other values onto each of the files. Partitions are not kept in object, however the other partitions values are.

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

`PartitionEnsemble.`**`hdf5_file`**

Default location for saving/loading PartitionEnsemble if hdf5 format is used. When this is set it will automatically resave the PartitionEnsemble into the file specified.

`PartitionEnsemble.`**`save`**(*filename=None*, *dir='.'*, *hdf5=True*, *compress=9*)

Use pickle or h5py to store representation of PartitionEnsemble in compressed file. When called if object has an assocated hdf5_file, this is the default file written to. Otherwise objected is stored using pickle.

Parameters

- **filename** – name of file to write to. Default is created from name of ParititonEnsemble: "%s_PartEnsemble_%d" %(self.name,self.numparts)
- **hdf5** (*bool*) – save the PartitionEnsemble object as a hdf5 file. This is very useful for larger partition sets, especially when you only need to work with the optimal subset. If object has hdf5_file attribute saved this becomes the default
- **compress** (*int [0, 9]*) – Level of compression for partitions in hdf5 file. With less compression, files take longer to write but take up more space. 9 is default.
- **dir** (*str*) – directory to save graph in. relative or absolute path. default is working dir.

`PartitionEnsemble.`**`save_graph`**(*filename=None*, *dir='.'*, *intra=True*)

Save a copy of the graph with each of the optimal partitions stored as vertex attributes in graphml compressed format. Each partition is attribute names part_gamma where gamma is the beginning of the partitions domain of dominance. Note that this is seperate from the information about the graph that is saved within the hdf5 file along side the partions.

Parameters

- **filename** – name of file to write out to. Default is self.name.graphml.gz or :type filename: str
- **dir** (*str*) – directory to save graph in. relative or absolute path. default is working dir.

`PartitionEnsemble.`**`open`**(*filename*)

Loads pickled PartitionEnsemble from file.

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

Returns writes over current instance and returns the reference

1.4.4 Merging PartitionEnsembles

```
import champ
import igraph as ig
import numpy as np

np.random.seed(0)
test_graph=ig.Graph.Erdos_Renyi(n=200,p=.1)
times={}
run_nums=[100]

ensemble=champ.parallel_louvain(test_graph,numprocesses=2,numruns=200,start=0,fin=4,
↪maxpt=4,progress=False)
print "Ensemble 1, Optimal subset is %d of %d partitions"%(len(ensemble.ind2doms),
↪ensemble.numparts)
ensemble.save("ensemble1.hdf5",hdf5=True)
```

(continues on next page)

(continued from previous page)

```
ensemble2=champ.parallel_louvain(test_graph,numprocesses=2,numruns=200,start=0,fin=4,
↪maxpt=4,progress=False)
print "Ensemble 2, Optimal subset is %d of %d partitions"%(len(ensemble2.ind2doms),
↪ensemble2.numparts)
ensemble2.save("ensemble2.hdf5",hdf5=True)

#Esembles can be merged as follows:

#Create new PartitionEnsemble from scratch
ensemble3=ensemble.merge_ensemble(ensemble2,new=True)
print "Ensemble 3, Optimal subset is %d of %d partitions"%(len(ensemble3.ind2doms),
↪ensemble3.numparts)

#Use largest of the 2 PartitionEnsembles being merged  and modify
ensemble4=ensemble.merge_ensemble(ensemble3,new=False)
print "Ensemble 4, Optimal subset is %d of %d partitions"%(len(ensemble4.ind2doms),
↪ensemble4.numparts)
print "ensemble4 is ensemble3: ",ensemble4 is ensemble3
```

Output:

```
Ensemble 1, Optimal subset is 13 of 200 partitions
Ensemble 2, Optimal subset is 15 of 200 partitions
Ensemble 3, Optimal subset is 15 of 400 partitions
Ensemble 4, Optimal subset is 15 of 600 partitions
ensemble4 is ensemble3: True
```

`PartitionEnsemble.`**`merge_ensemble`**(*otherEnsemble*, *new=True*)

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
- **`new`** (*bool*) – create a new PartitionEnsemble object? Otherwise partitions will be loaded into the one of the original partition ensemble objects (the one with more partitions in the first place).

Returns PartitionEnsemble reference with merged set of partitions

1.4.5 Improvement with HDF5 saving and loading

The following example gives a sense of when it is beneficial to save as HDF5 and general runtimes for parallelized Louvain. We also found that the overhead is dependent on the size of the graph as well, so that for larger graphs with a lower number of partitions, the read/write time on HDF5 can be greater. Total runtime was about 2.5 hours on 10 CPUs.

```
import champ
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
import seaborn as sbn
import numpy as np
import pandas as pd
import igraph as ig
from time import time

np.random.seed(0)
test_graph=ig.Graph.Erdos_Renyi(n=1000,p=.1)

times={}
run_nums=[100,1000,2000,3000,10000]

for nrun in run_nums :
    rtimes=[]
    t=time()
    ens=champ.parallel_louvain(test_graph,numprocesses=10,numruns=nrun,start=0,fin=4,
↪maxpt=4,progress=False)
    ens.name='name'
    rtimes.append(time()-t)

    #normal save (gzip)
    print "CHAMP running time %d runs: %.3f" %(nrun,rtimes[-1])
    t=time()
    ens.save()
    rtimes.append(time()-t)
    t=time()

    #normal open
    t=time()
    newens=champ.PartitionEnsemble().open("name_PartEnsemble_%d.gz"%nrun)
    rtimes.append(time()-t)
    t=time()

    #save with h5py
    t=time()
    ens.save(hdf5=True)
    rtimes.append(time()-t)
    t=time()

    #open with hdf5 format
    t=time()
    newes=champ.PartitionEnsemble().open("name_PartEnsemble_%d.hdf5"%nrun)
    rtimes.append(time()-t)
    times[nrun]=rtimes

tab=pd.DataFrame(times,columns=run_nums,index=['runtime','save','load','hdf5save',
↪'hdf5load'])
colors=sbn.color_palette('Set1',n_colors=tab.shape[0])
plt.close()
f,(a1,a2)=plt.subplots(1,2,figsize=(14,7))
for i,ind in enumerate(tab.index.values):
    if ind=='runtime':
        a1.plot(tab.columns,tab.loc[ind,:],label=ind,color=colors[i])
    else:
        a2.plot(tab.columns,tab.loc[ind,:],label=ind,color=colors[i])

a1.set_xlabel("#partitions")
```

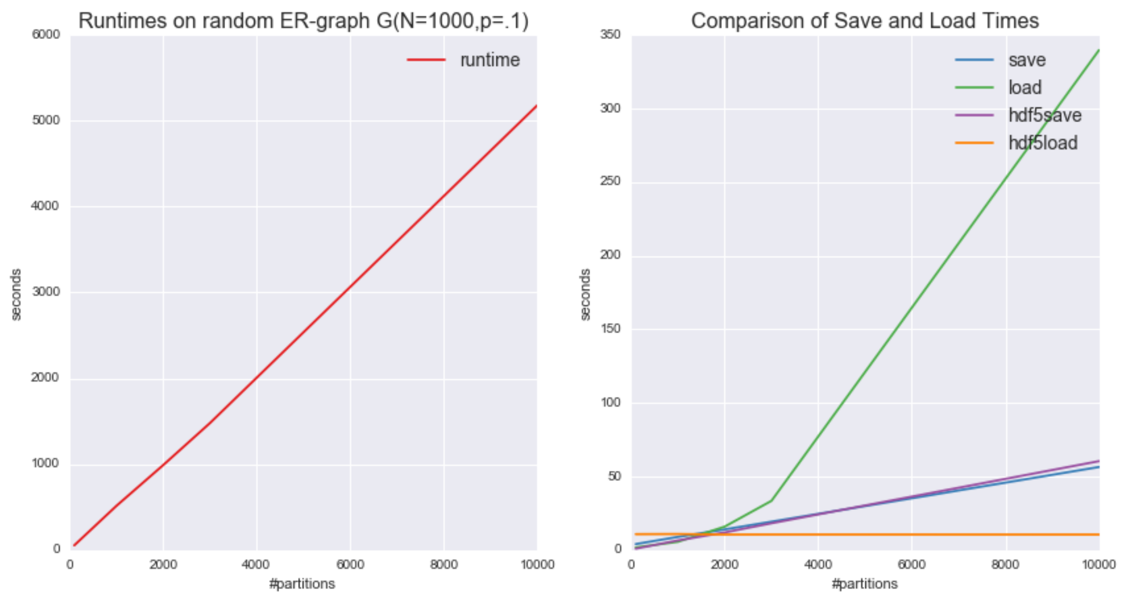
(continues on next page)

(continued from previous page)

```
a1.set_ylabel("seconds")
a2.set_xlabel("#partitions")
a2.set_ylabel("seconds")

a2.set_title("Comparison of Save and Load Times",fontsize=16)
a1.set_title("Runtimes on random ER-graph G(N=1000,p=.1)",fontsize=16)
a1.legend(fontsize=14)
a2.legend(fontsize=14)
plt.show()
```

Output:

```
CHAMP running time 100 runs: 53.771
CHAMP running time 1000 runs: 498.642
CHAMP running time 2000 runs: 988.658
CHAMP running time 3000 runs: 1479.512
CHAMP running time 10000 runs: 5091.727
```



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
```

We have also created a conda-forge recipe for creating a conda environment. If you wish to install through conda, first add conda-forge channel:

```
conda config --add channels conda-forge
```

Then you can create a environment using prepackaged versions from conda forge:

```
conda install champ
```

2.1 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 : Python numerical analysis library.
- sklearn :Machine learning tools for python.

- python-igraph :igraph python version for manipulation of networks.
- matplotlib :Python data visualization library.
- louvain :Vincent Traag’s implementation of louvain algorithm.
- h5py : HDF5 file format library for python.

These should all be handled automatically if using pip to install.

CHAPTER 3

Citation

Please cite:

bibtex

For more details and results see our manuscript

- genindex
- search

CHAPTER 4

Acknowledgements

This project was supported by the James S. McDonnell Foundation 21st Century ScienceInitiative-Complex Systems Scholar Award grant #220020315, by the National Institutes of Health through Award Numbers R01HD075712, R56DK111930 and T32GM067553, and by the CDC Prevention Epicenter Program. The content is solely the responsibility of the authors and does not necessarily represent the official views of the funding agencies.

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. *Algorithms*, 2017. URL: http://www.mdpi.com/1999-4893/10/3/93, doi:10.3390/a10030093.

[2] William H. Weir, Ryan Gibson, and Peter J Mucha. Champ package: convex hull of admissible modularity partitions in python and matlab. 2017. https://github.com/wweir827/CHAMP.

Index