SpacePAC: Identifying mutational clusters in 3D protein space using simulation.

Gregory Ryslik
Yale University
gregory.ryslik@yale.edu

Yuwei Cheng
Yale University
yuwei.cheng@yale.edu

Hongyu Zhao
Yale University
hongyu.zhao@yale.edu

May 3, 2016

Abstract

The **SpacePAC** package is designed to identify mutated amino acid hotspots while taking into account tertiary protein structure. This package is meant to complement the **iPAC** (**?**) and **GraphPAC** (**?**) packages already available in Bioconductor. Specifically, this method identifies the 1,2, or 3 spheres that cover the most number of mutations and by using simulation, provides a p-value if the spheres hold enough mutations to be statistically significant. The package also allows one to use a Poisson distribution to find the most significant sphere. Both the simulation and Poisson methods allow the user to consider spheres of multiple radii when finding the mutational hotspots. By providing an approach that identifies mutational hotspots directly in 3D space, we provide an alternative to the **iPAC** and **GraphPAC** methods which ultimately rely on remapping the protein to one dimensional space.

1 Introduction

Recent pharmacological advances in treating oncogenic driver mutations (**?**) has led to the development of multiple methods to identify amino acid mutational hotspots. Two recent methods, **iPAC** and **GraphPAC** provided an extension to the *NMC* algorithm (**?**) by taking into account protein tertiary structure. Both **iPAC** and **GraphPAC** remap the protein to 1D space (**iPAC** via MDS and **GraphPAC** via a graph theoretic method) in order to apply the *NMC* methodology which relies upon order statistics. While the remapping increases the sensivity of the algorithm and leads to the identification of novel clusters,

it nevertheless requires remapping the protein to 1D space which resulting in information loss. Here we present two methods that consider the protein directly in 3D space to identify mutational hotspots. This allows us to forgo the 1D requirement that was ultimately emposed by both **iPAC** and **GraphPAC**.

As in **iPAC** and **GraphPAC**, in order to run the clustering methodology, three types of data are required:

- The amino acid sequence of the protein that we obtain from the Sanger Institute or the Uniprot database in FASTA format.
- The protein tertiary structure that we obtain from the Protein Data Bank.
- The somatic mutation data that we obtain from the Catalogue of Somatic Mutations in cancer.

An alignment (or some other alternative renconciliation) algorithm must be used to reconcile the structural and mutational data. The mutational data must be in the format of an $m \times n$ matrix for a protein that is n amino acids long. A "1" in the (i, j) element indicates that residue j for individual i has a mutation while a "0" indiciates no mutation. To be compatible with this software, please ensure that your mutation matrix has the R column headings of $V1, V2, \cdots, Vn$. Only missense mutations are currently supported, indels in the amino acid sequence are not. Sample mutational data for KRAS and PIK3cα are included in this package as data sets. For a full description on how to extract correct mutational and positional data, please see the **iPAC** documentation as the procedure is identical to what is documented there. For the remainder of this vignette, we assume the user is familiar with *get.AlignedPositions*, *get.Positions*, and the mutation data format.

Note, that there is no one source to obtain the mutational data and that this often requires prior work on the part of the user. One free source of data is the COSMIC database `http://cancer.sanger.ac.uk/cancergenome/projects/cosmic/`. Should you choose to use COSMIC, a local SQL server is required to load the mutational database and custom query must be made for the gene of interest. Note, that the mutational data should come from whole gene screens or whole genome studies. Mutational data can not be selectively chosen as this will violate the uniformatiy assumption that the algorithm requires to run.

Should you find a bug, or wish to contribute to the code base, please contact the author.

2 Identifying Clusters Via Simulation

The general principle here is that we find the 1, 2 or 3 non-overlapping spheres that cover as many of the mutations as possible. We then simulate the mutations uniformly over the protein and use this distribution to calculate p values. Specifically, we proceed as follows:

- Let s be the number of spheres. $s \in \{1, 2, 3\}$.
- Let r be the radius currently being considered. Typically, $r \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and, if the data is obtained from the PDB, is measured in angstroms. For instance, when $r = 5$ all residues within 5 angstroms of the center of the sphere are included within the sphere.
- Simulate $N \geq 1000$ distributions of the mutations over the protein structure.

Next, let $X_{0,s,r}$ respresent the number of mutations captured within the spheres centered at residues p_1, p_2, p_3 for the observed data. The centers p_1, p_2, p_3 are chosen in a way such that the spheres capture as many mutations as possible (See Section **??**). Further, s represents the maximum number of spheres considered ($s \in \{1, 2, 3\}$). Let $X_{i,s,r}$ represent the same but for simulation i. For a given $\{s, r\}$, calculate $\mu_{s,r} = \underset{1 \leq i \leq N}{\text{mean}}\{X_{i,s,r}\}$ and $\sigma_{s,r} = \underset{1 \leq i \leq N}{\text{std. dev.}}\{X_{i,s,r}\}$. For each simulation, then calculate $Z_i = \max_i\{(X_{i,s,r} - \mu_{s,r})/\sigma_{s,r}\}$. The p-value is is then found as: $(\sum \mathbf{1}_{Z_0 > Z_i})/N$.

This process is best seen through Figure **??** below:

Radius:	r=3			r=9			$Z_{i=} \max(\frac{X_{i,s,r} - \mu_{s,r}}{\sigma_{s,r}})$
Num Spheres:	1	2	3	1	2	3	
obs	$X_{0,1,3}$	$X_{0,2,3}$	$X_{0,3,3}$	$X_{0,1,9}$	$X_{0,2,9}$	$X_{0,3,9}$	Z_0
i=1	$X_{1,1,3}$	$X_{1,2,3}$	$X_{1,3,3}$	$X_{1,1,9}$	$X_{1,2,9}$	$X_{1,3,9}$	Z_1
i=2	$X_{2,1,3}$	$X_{2,2,3}$	$X_{2,3,3}$	$X_{2,1,9}$	$X_{2,2,9}$	$X_{2,3,9}$	Z_2
i=3	$X_{3,1,3}$	$X_{3,2,3}$	$X_{3,3,3}$	$X_{3,1,9}$	$X_{3,2,9}$	$X_{3,3,9}$	Z_3
...	...	...	...	...	...	...	....
i=1000	$X_{1000,1,3}$	$X_{1000,2,3}$	$X_{1000,3,3}$	$X_{1000,1,9}$	$X_{1000,2,9}$	$X_{1000,3,9}$	Z_{1000}

Figure 1: Here we consider radii of 3 and 9 angstroms and want consider up to 3 spheres when identifying mutational hotspots (hence the number of spheres goes from 1 to 3). First, μ and σ are calculated over each column. Next, we normalize each entry in the column by calculating $Z_{i,s,r} = \frac{X_{i,s,r} - \mu_{s,r}}{\sigma_{s,r}}$. We then take the maximum over each row to get $Z_0, ..., Z_{1000}$. The percentage of times $Z_0 \geq Z_i$ where $i \in \{1, ..., 1000\}$, is the p-value of our observed statistic Z_0 (Z_0 is referred to as the Z.Score in the man pages and output.

An example of the code and ouput is shown in *Example 1* below. The sample code below allows up to 3 spheres (3 hotspots) but it can be set to only consider up to 1 or 2 spheres. We also consider 4 radii in the code below but you can consider as many sizes as you want. Note that the larger the radii, the more difficult it is to find non-overlapping spheres which increases the running time. Also, if the sphere sizes are *too* large, it might be impossible to find non-overlapping spheres. See Section **??** for the algorithm we use to identify the sphere positions.

Code Example 1: Running Spaceclust with 3 spheres with radii 1,2,3,4.

```
> library(SpacePAC)
> ##Extract the data from a CIF file and match it up with the canonical protein sequence.
> #Here we use the 2ENQ structure from the PDB, which corresponds to the PIK3CA protein.
> CIF <- "http://www.pdb.org/pdb/files/2ENQ.cif"
> Fasta <- "http://www.uniprot.org/uniprot/P42336.fasta"
> PIK3CA.Positions <- get.AlignedPositions(CIF, Fasta, "A")
> ##Load the mutational data for PIK3CA. Here the mutational data was obtained from the
> ##COSMIC database (version 58).
> data(PIK3CA.Mutations)
> ##Identify and report the clusters.
> my.clusters <- SpaceClust(PIK3CA.Mutations, PIK3CA.Positions$Positions, numsims =1000,
+   simMaxSpheres = 3, radii.vector = c(1,2,3,4), method = "SimMax")

Processing radius # 1 : radius length = 1.00 : Percentage complete 0.00
Processing radius # 2 : radius length = 2.00 : Percentage complete 0.25
Processing radius # 3 : radius length = 3.00 : Percentage complete 0.50
Processing radius # 4 : radius length = 4.00 : Percentage complete 0.75

> my.clusters

$p.value
[1] 0.036

$optimal.num.spheres
[1] 1

$optimal.radius
[1] 2

$optimal.sphere
   Line.Length Center Start End Positions MutsCount  Z.Score Within.Range
21           1    345   345 345       345         2 5.172137          345

$best.1.sphere
   Line.Length Center Start End Positions MutsCount  Z.Score Within.Range
21           1    345   345 345       345         2 5.172137          345

$best.2.sphere
  Line.Length1 Line.Length2 Center1 Center2 Start1 End1 Start2 End2 Positions1
1            1            1     420     345    420  420    345  345        420
2            1            1     453     345    453  453    345  345        453
  Positions2 MutsCount1 MutsCount2 MutsCountTotal  Z.Score Within.Range1
1        345          1          2              3 5.172137           420
2        345          1          2              3 5.172137           453
  Within.Range2 Intersection
1           345
2           345
```

```
$best.3.sphere
  Line.Length1 Line.Length2 Line.Length3 Center1 Center2 Center3 Start1 End1
1            1            1            1     453     420     345    453  453
  Start2 End2 Start3 End3 Positions1 Positions2 Positions3 MutsCount1
1    420  420    345  345        453        420        345          1
  MutsCount2 MutsCount3 MutsCountTotal  Z.Score Within.Range1 Within.Range2
1          1          2              4 5.172137           453           420
  Within.Range3 Intersection
1           345

$best.1.sphere.radius
[1] 2

$best.2.sphere.radius
[1] 2

$best.3.sphere.radius
[1] 2

$bad.2.sphere.message
NULL

$bad.3.sphere.message
NULL

$bad.2.sphere.radii
NULL

$bad.3.sphere.radii
NULL
```

Using PyMOL (**?**) (see Section **??**), we can now visualize these spheres in Figure **??** below. If you would like to render one sphere at a time see *make.3D.Sphere* for a built in R function.

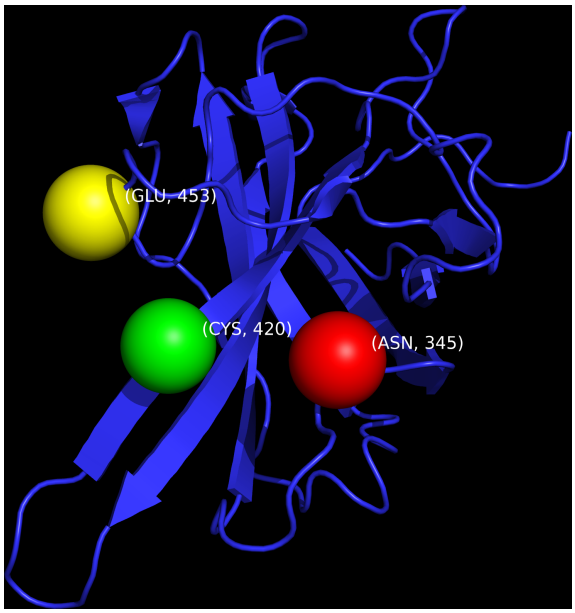

Figure 2: Plotting the 2ENQ structure with the 3 most significant spheres as shown in Code Example 1.

2.1 Quickly Identifying Mutational Positions

In the approach described in Section **??**, we find the 1, 2 or 3 spheres that cover the most mutations. If you consider 1 sphere, the number of possible spheres is linear in the length of the protein (namely, a sphere centered at each residue.) If we consider 2 spheres, there are $\binom{n}{2}$ possible sphere combinations if the protein is n residues long (and ignoring sphere overlap). If we consider 3 spheres, there are $\binom{n}{3}$ such combinations. For a medium-sized protein like PIK3Cα which is 1,068 residues long, considering 3 spheres provides 202,461,116 possible positions. To quickly find the best sphere orientation, we execute Algorithm **??** shown below. Algorithm **??** is shown for 2 spheres but is trivially extendable to 3 or more as well.

To illustrate how this algorithm works, suppose you have a protein with 5 mutated amino acids. Further, suppose that residue 1 has 50 mutations, residue 2 has 40 mutations, residue 3 - 30 mutations, residue 4 - 20 mutations and residue 5 - 10 mutations. For clarity, we proceed with unique mutation counts on each residue, but the algorithm is unaffected if there are identical counts on some positions. We first construct the table shown in Figure **??**, with the inside calculated to be the sum of the number of mutations in amino acid i and amino acid j. Observing, that the table is symmetric, we only need to evaluate the residues below the diagonal as the entries on the diagonal come from residues that overlap one another perfectly.

Algorithm 1 Here we are interested in finding the two non-overlapping spheres that contain the most mutations.

Require: Sorted vector of counts v with length >= 2
  starti = 2;
  startj = 1;
  k = length(v);
  cand = [(starti, startj,v[starti] + v[startj])]
  while (!is.empty(cand)) **do**
    index = max(cand) {Comment: max upon the last element in the 3-tuple.}
    i,j,s = cand[index]
    cand = cand[-index] {Comment: Removes the current max}
    if (No overlap between sphere i and j) **then**
      Return (i, j, v[i]+v[j]) {Comment: Successful combination found.}
    end if
    if (j ==1) and ($i < k$) **then**
      cand.append((i+1, j,v[i+1] + v[j]))
    end if
    if ($j < i$) and (i!= j+1) **then**
      cand.append((i, j+1,v[i] + v[j+1]))
    end if
  end while
  Return NULL {Comment: No succesful combination found.}

		1	2	3	4	5
	Counts	50	40	30	20	10
1	50	100	90	80	70	60
2	40	90	80	70	60	50
3	30	80	70	60	50	40
4	20	70	60	50	40	30
5	10	60	50	40	30	20

Figure 3: In our example, the protein has 5 residues with mutations. The residues are sorted from largest to smallest (so mutation 1 has the largest number of mutations and so forth), and the inside of the table is calculated as the sum of the mutations on both residues. In the actual code, only the lower half of the table is considered and then only sequentially to decrease running time, but we present the whole table here for clarity. Further, in the code, there is no need to set up a matrix, only the vector of mutation counts and the current location in that vector need be considered. We present the algorithm here in matrix form for ease of exposition.

The algorithm then proceeds by appending to the potential "candidate" stack the element below and the element to the right starting from the (2,1) position.

After every two potential appends (can be 0, 1 or 2) to the stack (assuming you're not on an edge and an append is possible), the maximum value over all the 3-tuple's third positions is found (thus the max mutation count in both spheres). The 3-tuple with this maximum value is then evaluated to see if the spheres overlap. If the spheres do *not* overlap, then the succesful case has been found and the algorithm terminates and returns the result. If the spheres *do* overlap, the next set of elements are appended to the stack and the process continues. By proceeding in this way, (pseudocode shown in Algorithm **??**), at each iteration, the pair of spheres being considered contain the maximum number of mutations possible from the remaining set. Further, we do not need to consider all the positions as once the first pair of non-overlapping spheres is found, we know that this is the combination of spheres that is both non-overlapping and captures the most mutations. To see this process, see Figure **??**.

Iteration	Testing	Appending	Candidate Stack
1	(2,1,90)	(3,1,80)	(3,1,80)
2	(3,1,80)	(4,1,70), (3,2,70)	(4,1,70), (3,2,70)
3	(4,1,70)	(4,2,60), (5,1,60)	(3,2,70), (4,2,60), (5,1,60)
4	(3,2,70)	-	(4,2,60), (5,1,60)
5	(5,1,60)	(5,2,50)	(4,2,60), (5,2,50)
6	(4,2,60)	(4,3,50)	(5,2,50), (4,3,50)
7	(5,2,50)	(5,3,40)	(4,3,50), (5,3,40)
8	(4,3,50)	-	(5,3,40)
9	(5,3,40)	(5,4,30)	(5,4,30)
10	(5,4,30)	-	-

Figure 4: Beginning in position (2,1,90), we remove (2,1,90) from the list and append [(3,1,80),(2,2,80)]. We then test and remove [(2,2,80)], leaving just (3,1,80) in the list. After each append to the list, we find the max element in the list by mutation count and pick the element with the maximum number of mutations between both spheres.

3 Identifying Clusters Via The Poisson Distribution

This function uses a Poisson distribution to find signficant spheres. Assuming there are m total mutations on n amino acids, we first calculate an average mutation rate per amino acid, $\bar{\lambda} = m/n$. We then consider n spheres, where sphere $i \leq n$ is centered at amino acid i. For sphere i, we calculate $\lambda_{i,r} = \bar{\lambda} a_{i,r}$ where $a_{i,r}$ is the number of amino acids within sphere i (with radius r. Finally, for each sphere we calculate $Pr(X \geq x)$ where x is the observed number of mutations in the sphere and X follows a Poisson distribution with paramater $\lambda_{i,r}$. Given that this calculation occurs n times (once for each sphere), a multiple

comparison adjustment (specified in the "multcomp" parameter) is applied to account for the multiple spheres. The p-values that are below the significance level α are returned to the user. Finally, if more than one radii is considerd, each p value is multiplied by the length of the radii vector to account for the multiple comparisons introduced by considering multiple radii.

Due to the numerous multiple comparison adjustments required, we recommend using the Simulation approach described in Section **??**. We have left the Poisson method in this package in the case that you are only considering one radius for shorter proteins.

Code Example 2: Running Spaceclust using the Poisson distribution.

```
> ##Extract the data from a CIF file and match it up with the canonical protein sequence.
> #Here we use the 2ENQ structure from the PDB, which corresponds to the PIK3CA protein.
> CIF <- "http://www.pdb.org/pdb/files/3GFT.cif"
> Fasta <- "http://www.uniprot.org/uniprot/P01116-2.fasta"
> KRAS.Positions <- get.Positions(CIF, Fasta, "A")
> data(KRAS.Mutations)
> ##Identify and report the clusters.
> my.clusters <- SpaceClust(KRAS.Mutations, KRAS.Positions$Positions, radii.vector = c(1,2,3,4),
+   alpha = .05, method = "Poisson")
Processing radius # 1 : radius length = 1.00 : Percentage complete 0.00
Processing radius # 2 : radius length = 2.00 : Percentage complete 0.25
Processing radius # 3 : radius length = 3.00 : Percentage complete 0.50
Processing radius # 4 : radius length = 4.00 : Percentage complete 0.75
> my.clusters
$result.poisson
   Line.Length Center Start End Positions MutsCount       P.Value
13           3     13    12  14    12, 13       131 5.723774e-165
12          50     12    11  60    12, 13       131 5.497664e-149
11           3     11    10  12        12       100 2.904602e-114
60          50     60    12  61    12, 61       105 1.867889e-109
14           3     14    13  15        13        31  1.097621e-19
     Within.Range
13     12, 13, 14
12 11, 12, 13, 60
11     10, 11, 12
60 12, 59, 60, 61
14     13, 14, 15

$best.radii
[1] 4
```

4 Plotting

SpacePAC provides simplified plotting functionality. The function takes in the position matrix, the amino acid number at which the sphere should be centered

as well as the radius of the sphere. The alpha level specifies how dark or light the shading of the sphere should be. Only 1 sphere is able to be plotted at this time. For more advanced rendering options, we recommend the user to consider using the software package PyMOL at http://www.pymol.org (**?**).

The code renders the protein and sphere using the rgl package. Please run ?make.3D.Sphere for the syntax options. A figure of the KRAS protein with a sphere around residue 12 with a radius equal to 3 is shown below.

Code Example 3: Making a Plot.

```
>   ##To avoid RGL errors, this code is not run. However it has been tested and verified.
>   #library(rgl)
>   #CIF <- "http://www.pdb.org/pdb/files/3GFT.cif"
>   #Fasta <- "http://www.uniprot.org/uniprot/P01116-2.fasta"
>   #KRAS.Positions <- get.Positions(CIF, Fasta, "A")
>   #make.3D.Sphere(KRAS.Positions$Positions, 12, 3)
```

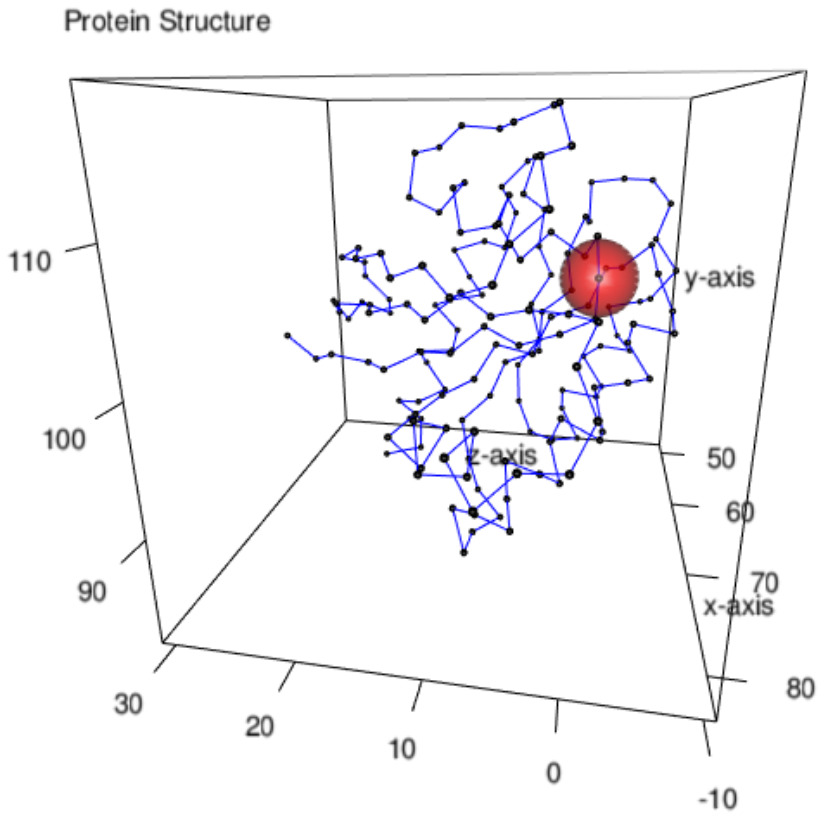

Figure 5: Screenshot of RGL graphics of KRAS structure with sphere of radius 3 at residue 12. The RGL window that will open is interactive and allows the protein to be rotated.