VeriNA3d: introduction and use cases

Diego Gallego [1,2]***, Leonardo Darre*** [1,3]***, Pablo D. Dans*** [1]***, and Modesto Orozco*** [1,2]

[1]Institute for Research in Biomedicine (IRB Barcelona), the Barcelona Institute of Science and Technology, Spain.
[2]Department of Biochemistry and Molecular Biomedicine, Faculty of Biology, University of Barcelona, Spain.
[3]Functional Genomics Laboratory and Biomolecular Simulations Laboratory, Institute Pasteur of Montevideo, Uruguay.

2019-06-19

Contents

1 Introduction: Structural Bioinformatics in R

The R language provides an excellent interface for statistical analysis, which is also interesting from the point of view of structural data. This gap was filled in 2006 by the R package bio3d (Grant et al. 2006). It was presented as a suite of tools to handle PDB formated structures, and trajectories. It integrates a variety of functions to analyse from sequence to 3D structure data (RMSD, NMA, PCA... see their documentation for details). As far as we know, bio3d represented the only structural package for R until now.

The R package presented in here, veriNA3d, does not replace bio3d at all. Rather, it was developed on top of it to cover additional necessities. The only common tool integrated in both packages is a parser for mmCIF files (see below). VeriNA3d is mainly intended (but not limited) to the analysis of Nucleic Acids. It integrates a higher level of abstraction than bio3d since it also allows the analysis of datasets, in addition to analysis of single structures. The functions in the package could be divided in the following blocks:

- Dataset level: Functions to get and analyse lists of pdb IDs. This includes access to the representative lists of RNA by (Leontis and Zirbel 2012) and other analytical functions.
- Structure level: Functions to get data, parse mmCIF files and analyse them. This includes a wrapper of DSSR (Lu, Bussemaker, and Olson 2015) and a function to calculate the ϵRMSD (Bottaro, Di Palma, and Bussi 2014).
- Plots: few functions to show the results of the previous analysis.

The complete list of functions can be found in the README.md file within the package, also accessible on the gitlab main page.

2 Parsing mmCIF files

2.1 Origin and standardization of mmCIF files

Atomic structural data of macromolecules has long been distributed in the PDB file format. However, one of its main limitations is the column size for the coordinates data, which didn't allowed to save molecules with more than 99999 atoms, more than 62 chains or more than 9999 residues (in a chain).

Given that the Protein Data Bank is continously growing and accepting bigger structures (e.g. a whole *E.coli* ribosome has over 140000 atoms - pdbID 4V4S), an alternative file format became the standard: the mmCIF file format.

The mmCIFs are an evolution of the Crystallographic Information File (CIF), originally used for small molecule structures. It stands for **m**acro**m**olecular **CIF** file, and it has actually coexisted with the PDB format since the 1997. However, since the PDB is easier to parse and such big structures didn't populate the database at the time, most software has been developed for the PDB format.

The PDB format was definetely frozen in 2014. However, it will still coexist with the standard mmCIF format as long as all softwares evolve to accept mmCIFs. Following this trend, the bio3d R package integrated a `read.cif` function in their version 2.3. At that time, we had already started the development of our own `cifParser` function. Given that the mmCIF format is constantly evolving and that both functions take slightly different approaches, we decided to offer our own version of it, which might provide an useful and fast alternative for users working with mmCIF files.

2.2 The CIF object

Parsing a particular file format often involves creating a new class of object. In R, the principal objects are called S3, S4 and RC. Our container for mmCIF data is an S4 object called **CIF**, in contrast with the S3 object (called **pdb**) in bio3d - it is worth noting that this difference does **not** affect the compatibility between the two packages (see below for details).

Since different mmCIF files usually have different sections of data (in addition to the coordinates), we carried out an analysis that checked which ones are always present in all mmCIF files (this included all mmCIF files in the Protein Data Bank in March 2018), and reached a list of 14 items:

- Atom_site
- Atom_sites
- Atom_type
- Audit_author
- Audit_conform
- Chem_comp
- Database_2
- Entity
- Entry
- Exptl
- Pdbx_database_status
- Struct

- Struct_asym
- Struct_keywords

The detailed description of each data sections can be found in the mmCIF main site.

The CIF object is created by the `cifParser` function and contains these 14 sections of data, which can be accessed with the CIF accessors. To see the accessor functions run:

```
library(veriNA3d)
?cif_accessors
```

To read a mmCIF file and access the coordinates data, use:

```
## To parse a local mmCIF file:
# cif <- cifParser("your-file.cif")
## To download from PDB directly:
cif <- cifParser("1bau")
cif
#>
#> -- mmCIF with ID: 1BAU ---------------------------------------------------------
#>
#> Author description: NMR STRUCTURE OF THE DIMER INITIATION COMPLEX OF HIV-1
#> GENOMIC RNA, MINIMIZED AVERAGE STRUCTURE
#>
#> mmCIF version:       5.279
#>
#> To extract coordinates and other data use accessor functions
#> (type ?cif_accessors for details)
## To see the coordinates:
coords <- cifAtom_site(cif)
head(coords)
#>   group_PDB id type_symbol label_atom_id label_alt_id label_comp_id
#> 1      ATOM  1           O           O5'            .             G
#> 2      ATOM  2           C           C5'            .             G
#> 3      ATOM  3           C           C4'            .             G
#> 4      ATOM  4           O           O4'            .             G
#> 5      ATOM  5           C           C3'            .             G
#> 6      ATOM  6           O           O3'            .             G
#>   label_asym_id label_entity_id label_seq_id pdbx_PDB_ins_code Cartn_x
#> 1             A               1            1                 ?  23.989
#> 2             A               1            1                 ?  24.965
#> 3             A               1            1                 ?  25.937
#> 4             A               1            1                 ?  26.741
#> 5             A               1            1                 ?  25.259
#> 6             A               1            1                 ?  24.868
#>   Cartn_y Cartn_z occupancy B_iso_or_equiv pdbx_formal_charge auth_seq_id
#> 1   8.289 -15.135         1              0                  ?           1
#> 2   9.100 -14.503         1              0                  ?           1
#> 3   9.725 -15.512         1              0                  ?           1
#> 4   8.744 -16.162         1              0                  ?           1
#> 5  10.527 -16.627         1              0                  ?           1
#> 6  11.838 -16.252         1              0                  ?           1
#>   auth_comp_id auth_asym_id auth_atom_id pdbx_PDB_model_num
```

```
#> 1           G           A          O5'                1
#> 2           G           A          C5'                1
#> 3           G           A          C4'                1
#> 4           G           A          O4'                1
#> 5           G           A          C3'                1
#> 6           G           A          O3'                1
```

2.3 Bidirectional compatibility with bio3d

Using the `cifParser` will often be the first step to analyse a structure. However, if the analysis requires any of the bio3d functions, then a conversion should be done with the `cifAsPDB` function.

```
pdb <- cifAsPDB(cif)
pdb
#>
#>  Call:  "1BAU"
#>
#>    Total Models#: 1
#>      Total Atoms#: 1486,  XYZs#: 4458  Chains#: 2  (values: A B)
#>
#>      Protein Atoms#: 0  (residues/Calpha atoms#: 0)
#>      Nucleic acid Atoms#: 1486  (residues/phosphate atoms#: 46)
#>
#>      Non-protein/nucleic Atoms#: 0  (residues: 0)
#>      Non-protein/nucleic resid values: [ none ]
#>
#>    Nucleic acid sequence:
#>       GGCAAUGAAGCGCGCACGUUGCCGGCAAUGAAGCGCGCACGUUGCC
#>
#> + attr: atom, xyz, calpha, model, flag, call
```

It takes a CIF object and generates an equivalent pdb object (as used by all bio3d functions). In addition, all veriNA3d functions are prepared to accept as input either the CIF or pdb objects. Therefore, the compatibility between the two packages is bidirectional.

3 Downloaded files

The `cifParser` and other functions download the mmCIF files when they are not provided by the user. The core code that performs this operation is `cifDownload`.

`cifDownload` uses a temporal directory to save files, which can be found running `tempdir`. These files are saved accross the same R session, so no file will be downloaded twice, but removed after the session is terminated.

In some cases, such as intensive use of the package or limited Internet connection, the user might want to keep the files accross different R sessions. This functionality can be achieved with a simple process.

By default, `cifDownload` tries to find a directory called "~/.veriNA3d_mmCIF_files/". Only when it does not find it, it will use a temporary directory. If you want to keep files across R sessions, just create this new folder in your home directory and veriNA3d will start using it to save all the downloaded mmCIF files. Note that the only way to recover the space in your drive is to remove those files *manually*.

Creating this directory can be easily done running the following command in R.

```
dir.create("~/.veriNA3d_mmCIF_files/")
```

On the other hand, if you prefer to save your files in a specific directory of your choice, you can create a symbolic link called .veriNA3d_mmCIF_files pointing to your desired location, and it will also work. Instructions for this particular task are not included, since they are system specific.

4 Getting data from Application Programming Interfaces (API)

Getting data is the first step of any pipeline, and parsing files is just one of the many ways data can be accessed. Application Programming Interfaces (API) are an intermediate point of access to a remote database. APIs offer the users a series of *endpoints* or *calls*, which are just **links**. Thus, instead of dealing directly with the database with SQL or other language, the user can just use the appropriate *call* to send a query to the API, and it will return the desired data.

In addition to parsing mmCIF files, veriNA3d also offers a series of functions to send queries to different APIs. Since sending queries to remote APIs requires Internet access, the full functionality of veriNA3d might depend on a good connection.

To see all the query functions, run:

```
?queryFunctions
```

IMPORTANT NOTE: The APIs accessed by veriNA3d are free of use with no limit of queries per user. However, this could change if the users of the APIs use them irresponsibly. Servers could eventually fall down if they receive more *calls* than they can actually process. To avoid that, veriNA3d actually saves in memory the result of any query, and any time that you use the same query again, it will take the cached result. To see this effect, run this test:

```
## Run a query for the first time, which will access the API
tech <- queryTechnique("4KQX", verbose=TRUE)
#> [1] "Querying: http://www.ebi.ac.uk/pdbe/api/pdb/entry/summary/4KQX"
#> [1] "Getting expType from API"
#> [1] "Saving expType in RAM"

## Run the same query for the second time, which will get it from memory
tech <- queryTechnique("4KQX", verbose=TRUE)
#> [1] "Querying: http://www.ebi.ac.uk/pdbe/api/pdb/entry/summary/4KQX"
#> [1] "Getting expType from RAM"

## See result
tech
#> [1] "X-ray diffraction"
```

However, this is only saved across the current session of R, and any script that uses the query functions will send them to the API every time it is run. VeriNA3d does not guarantee the correct service of the APIs and it does not monitor any of your processes. However, the API providers can known which IP address is querying their services at any time. To avoid overloading the servers, please make a responsible use (e.g. save locally the data that you use frequently).

4.1 Querying the EMBL-EBI REST API

An invaluable resource for structural & computational biologists is the PDBe REST API (Velankar et al. 2015). Around this technology, the package includes the following set of functions:

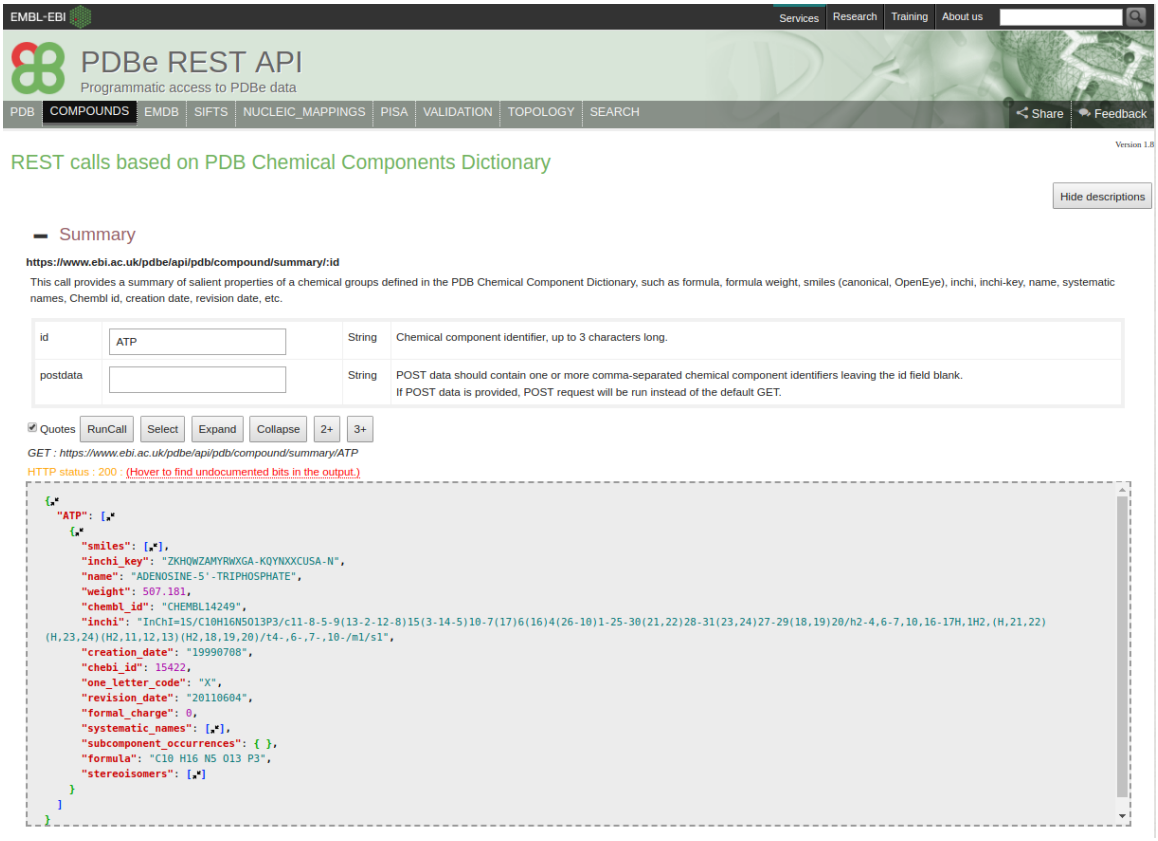
- queryAuthors: List of authors
- queryReldate: Release date
- queryDepdate: Deposition date
- queryRevdate: Revision date
- queryDescription: Author description
- queryEntities: Entitity information
- queryFormats: File formats for the given ID
- queryModres: Modified residues
- queryLigands: Ligands in structure
- queryOrgLigands: Ligands in structure (substracting ions)
- queryResol: Resolution (if applicable)
- queryTechnique: Experimental Technique
- queryStatus: Released/Obsolete and related status information

The list of functions is **intentionally limited** in comparison with the dozens of *endpoints* of the REST API. Integrating them all would needlessly increase the total amount of functions of the package. Moreover, the API might offer more and more *endpoints* with the time, and trying to keep them all would make the manteinance of this package more difficult. To allow the users to access their desired endpoints, an alternative method is provided.

The core of all the query functions is `queryAPI`, which integrates all the error-handling and cache functionalities. With the `queryAPI` function, any user can design their own queries, with a simple process. Herein a couple of examples.

4.1.1 Example 1

This snapshot shows the REST API website and a *call* that is not implemented in veriNA3d:



The link of this *endpoint* is: https://www.ebi.ac.uk/pdbe/api/pdb/compound/summary/ATP

The `queryAPI` function can understand and send this query using the arguments 'ID', 'API', 'string1', and 'string2' properly:

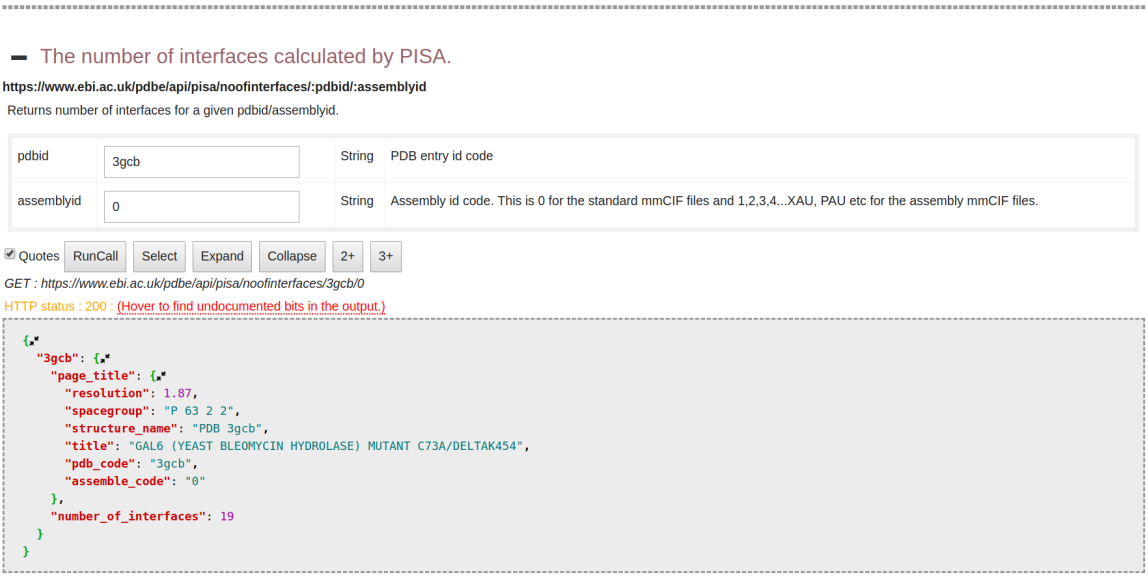
```
atpsummary <- queryAPI(ID="ATP", API="ebi",
                       string1="pdb/compound/summary/", string2="")
str(atpsummary$ATP)
#> 'data.frame':    1 obs. of  15 variables:
#>  $ smiles                 :List of 1
#>   ..$ :'data.frame': 2 obs. of  3 variables:
#>   .. ..$ program: chr  "CACTVS" "OpenEye OEToolkits"
#>   .. ..$ version: chr  "3.341" "1.5.0"
#>   .. ..$ name   : chr  "Nc1ncnc2n(cnc12)[C@@H]3O[C@H](CO[P@](O)(=O)O[P@@](O)(=O)O[P](O)(O)=O)[C@@H](O)[C@H
#>  $ inchi_key              : chr "ZKHQWZAMYRWXGA-KQYNXXCUSA-N"
#>  $ name                   : chr "ADENOSINE-5'-TRIPHOSPHATE"
#>  $ weight                 : num 507
#>  $ chembl_id              : chr "CHEMBL14249"
#>  $ inchi                  : chr "InChI=1S/C10H16N5O13P3/c11-8-5-9(13-2-12-8)15(3-14-5)10-7(17)6(16)4(26-1
#>  $ creation_date          : chr "19990708"
#>  $ chebi_id               : int 15422
#>  $ one_letter_code        : chr "X"
#>  $ revision_date          : chr "20110604"
#>  $ formal_charge          : int 0
#>  $ systematic_names       :List of 1
```

```
#>   ..$ :'data.frame': 2 obs. of  3 variables:
#>   .. ..$ program: chr  "ACDLabs" "OpenEye OEToolkits"
#>   .. ..$ version: chr  "10.04" "1.5.0"
#>   .. ..$ name   : chr  "adenosine 5'-(tetrahydrogen triphosphate)" "[[(2R,3S,4R,5R)-5-(6-aminopurin-9-yl)
#>  $ subcomponent_occurrences:'data.frame':    1 obs. of  0 variables
#>  $ formula                 : chr "C10 H16 N5 O13 P3"
#>  $ stereoisomers           :List of 1
#>   ..$ :'data.frame': 1 obs. of  2 variables:
#>   .. ..$ name        : chr "9-{5-O-[(S)-hydroxy{[(R)-hydroxy(phosphonooxy)phosphoryl]oxy}phosphoryl]-beta
#>   .. ..$ chem_comp_id: chr "HEJ"
```

- The common root in all the REST API *endpoints* is "https://www.ebi.ac.uk/pdbe/api/", which is internally managed by the function by using *API="ebi"*.
- The *string1="pdb/compound/summary/"* indicates everything that comes after the root and before the ID.
- The *ID="ATP"* obviously represents the desired structure, either a 4 character string for a pdbID or <= 3 character string for compounds.
- The *string2=""* is also a necessary argument that reflects that nothing comes after the ID.

4.1.2 Example 2

This snapshot shows a second *call* not implemented in veriNA3d:



The link of this *endpoint* is: https://www.ebi.ac.uk/pdbe/api/pisa/noofinterfaces/3gcb/0

The proper call with `queryAPI` would be:

```
PISAsummary <- queryAPI(ID="3gcb", API="ebi",
                        string1="pisa/noofinterfaces/", string2="0")
str(PISAsummary$"3gcb")
#> List of 2
```

```
#>  $ page_title          :List of 6
#>   ..$ resolution    : num 1.87
#>   ..$ spacegroup    : chr "P 63 2 2"
#>   ..$ structure_name: chr "PDB 3gcb"
#>   ..$ title         : chr "GAL6 (YEAST BLEOMYCIN HYDROLASE) MUTANT C73A/DELTAK454"
#>   ..$ pdb_code      : chr "3gcb"
#>   ..$ assemble_code : chr "0"
#>  $ number_of_interfaces: int 19
```

This second example shows a case in which the "string2" argument is necessary. If you are unsure about the real link that is actually being constructed, you can always use *verbose=TRUE* to see it printed.

5 The ϵRMSD to compare structures

A new interesting metric to compare Nucleic Acid structures is the ϵRMSD (Bottaro, Di Palma, and Bussi 2014), currently available in the BaRNAba python package (Bottaro et al. 2018). The metric was implented in the `eRMSD` function and completely reproduces BaRNAba results for the structures tested.

The following example shows how to get the ϵRMSD between two models of the same structure:

```
## Parse cif file
cif <- cifParser("2d18")
## Select a couple of models
model1 <- selectModel(cif=cif, model=1)
model3 <- selectModel(cif=cif, model=3)

## Calculate the eRMSD
eRMSD(cif1=model1, cif2=model3)
#> [1] 0.3000208

## The RMSD can also be calculated easily
RMSD(cif1=model1, cif2=model3)
#> [1] 1.555
```

6 Generate substructures

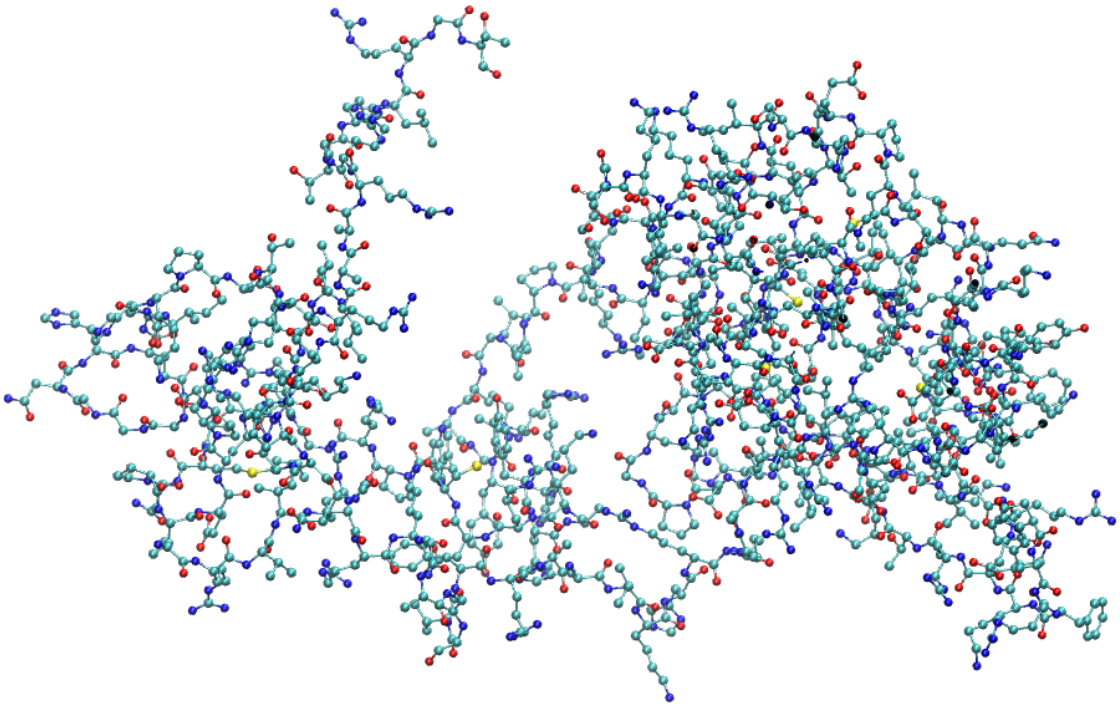
In many cases you might be interested on a particular region of a structure (e.g. a peptide from a complex, or a ligand and its binding site). For a given structure, `trimSphere` can generate a smaller pdb object and save it to a PDB file if desired. The region of interest can be selected by using the chain identifier and the residue index, or with the `atom.select` function from bio3d (which in turn allows you to select regions of the structure in a variety of ways). In addition to the region of interest, the function can also include the surrounding region by seting a cutoff distance.

6.0.1 Example 1

```
## Parse human ribosome - takes around 12 seconds in R-3.5
cif <- cifParser("6ek0")

## Query entities and check them
ent <- queryEntities("6ek0")
head(ent[, c("entity_id", "molecule_name", "in_chains")])
#>   entity_id            molecule_name in_chains
#> 1         1        28S ribosomal RNA        L5
#> 2         2         5S ribosomal RNA        L7
#> 3         3       5.8S ribosomal RNA        L8
#> 4         4 60S ribosomal protein L8        LA
#> 5         5 60S ribosomal protein L3        LB
#> 6         6 60S ribosomal protein L4        LC

## Generate a smaller pdb with the 60S ribosomal protein L8
chain <- "LA"
protL8 <- trimSphere(cif, chain=chain, cutoff=0)
protL8
#>
#>  Call:  trim.pdb(pdb = pdb, inds = sel)
#>
#>    Total Models#: 1
#>      Total Atoms#: 1898,  XYZs#: 5694  Chains#: 1  (values: A)
#>
#>      Protein Atoms#: 1898  (residues/Calpha atoms#: 248)
#>      Nucleic acid Atoms#: 0  (residues/phosphate atoms#: 0)
#>
#>      Non-protein/nucleic Atoms#: 0  (residues: 0)
#>      Non-protein/nucleic resid values: [ none ]
#>
#>    Protein sequence:
#>       GRVIRGQRKGAGSVFRAHVKHRKGAARLRAVDFAERHGYIKGIVKDIIHDPGRGAPLAKV
#>       VFRDPYRFKKRTELFIAAEGIHTGQFVYCGKKAQLNIGNVLPVGTMPEGTIVCCLEEKPG
#>       DRGKLARASGNYATVISHNPETKKTRVKLPSGSKKVISSANRAVVGVVAGGGRIDKPILK
#>       AGRAYHKYKAKRNCWPRVRGVAMNPVEHPFGGGNHQHIGKPSTIR...<cut>...LRGT
#>
#> + attr: atom, helix, sheet, seqres, xyz,
```

```
#>         calpha, call

## The same command with the argument file would save it directly:
trimSphere(cif, chain=chain, cutoff=0, file="output.pdb")
```

The output.pdb file can then be visualized using your favourite external molecular viewer. The following picture was taken using VMD software (Humphrey, Dalke, and Schulten 1996).

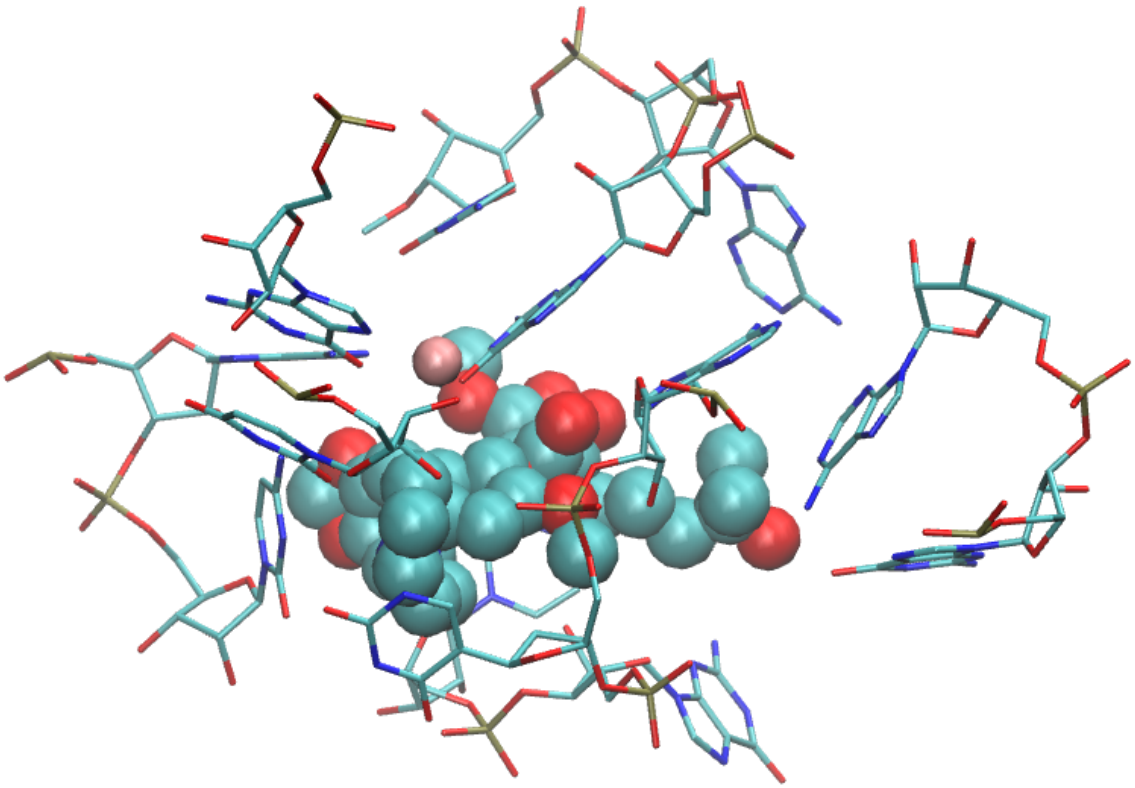


6.0.2 Example 2

To get the desired region of interest and its sorroundings, let's see a second example using the same structure:

```
## Load bio3d library
library(bio3d)

## Get pdb object from CIF
pdb <- cifAsPDB(cif)

## Get list of ligands in the human ribosome 6EK0
queryLigands("6ek0", onlyligands=T)
#> [1] "MG"  "HMT" "ZN"  "HYG"

## Get the atomic index for a desired ligand
HMTligand_inds <- pdb$atom$eleno[which(pdb$atom$resid == "HMT")]
```

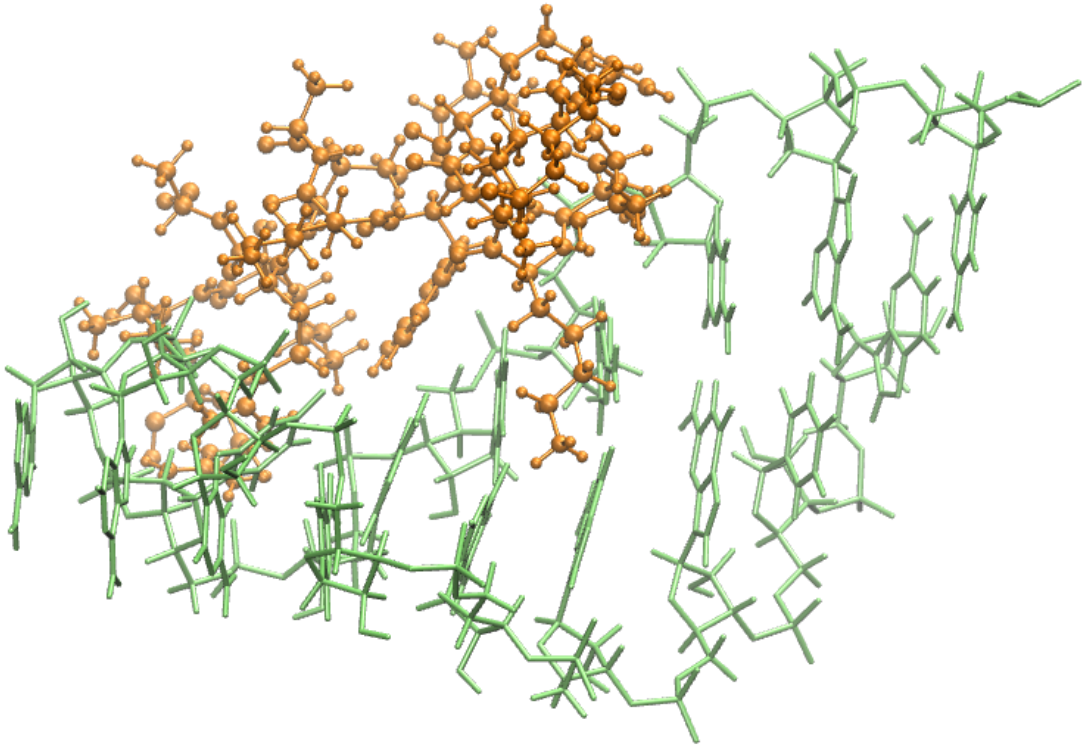
```
## Use bio3d function to select the ligand using its atom indices
sel <- atom.select(pdb, eleno=HMTligand_inds)

## Get substructure and sorroundings at 10 Angstroms
HTMligand <- trimSphere(pdb, sel=sel, cutoff=5)
#> [1] "Computing distances ..."
#> [1] " ... done"
#> [1] "Finding the atom details ..."
#> [1] " ... done, the output is coming"

## And generate file to visualize it
trimSphere(pdb, sel=sel, file="output2.pdb", cutoff=5)
#> [1] "Computing distances ..."
#> [1] " ... done"
#> [1] "Finding the atom details ..."
#> [1] " ... done, the output is coming"
```



6.0.3 Example 3

A third useful example would be the generation of a pdb with the interacting region between two molecules in the structure. To achieve the goal, veriNA3d also counts with the `findBind ingSite` function, as shown below:

```
## Parse another pdb for this example
pdb <- cifAsPDB("1nyb")

## Find region of interaction between RNA and protein
data <- findBindingSite(pdb, select="RNA", byres=TRUE)

## Get atom indices from interacting region molecules
eleno <- append(data$eleno_A, data$eleno_B)

## Select using bio3d
sel <- atom.select(pdb, eleno=eleno)

## Get substructure
trimSphere(pdb, sel=sel, file="interacting_site.pdb", verbose=FALSE)
```



7 Calculate phosphate pair-wise distances

Atomic distances are calculated internally in the `trimSphere` function, a resource that is also made available to the user through the functions `measureEntityDist` (returns all the distances between the atoms of two entities) and `measureElenoDist` (returns all the distances between two sets of atoms).

Herein it is shown how to calculate the pair-wise distances between all the phosphate atoms of an RNA structure, which could be easily adapted for alpha carbons or other elements.

```
## Parse pdb
pdb <- cifAsPDB("1nyb")

## Select P atoms by element number (eleno)
ind <- which(pdb$atom$elety == "P")
eleno <- pdb$atom$eleno[ind]

## Count number of phosphates
total <- length(eleno)

## Execute function to measure the distances
P_distances <- measureElenoDist(pdb=pdb, refeleno=eleno, eleno=eleno,
                                n=total, cutoff=100)
```

8 Manage Nucleic Acid datasets

VeriNA3d also provides a set of tools to work at a higher level of abstraction, which allow the user to manage datasets of structures at once. Such approaches are often found in the literature in a variety of studies, either with descriptive (Murray et al. 2003, Darré et al. (2016)), validation (Jain, Richardson, and Richardson 2015) or modeling (Z. Wang and Xu 2011, J. Wang et al. (2015)) purposes. However, as far as we know, no software with dataset analysis purposes has ever been freely distributed before.

8.1 Non-redundant lists of RNA

VeriNA3d integrates the function `getRNAList`, which returns the desired non-redundant list of RNA structures, as found in (Leontis and Zirbel 2012) website. Here it is shown an example of how to make a non-redundant dataset of protein-RNA complexes with resolutions better than 2 Angstroms.

```
## Get non-redundant list from Leontis website
rnalist <- getRNAList(release="3.47", threshold="2.0A")
head(rnalist)
#>   Equivalence_class Representative     Class_members
#> 1    NR_2.0_29848.1       3DIL|1|A          3DIL|1|A
#> 2    NR_2.0_47162.1       3PDR|1|X 3PDR|1|X;3PDR|1|A
#> 3    NR_2.0_39627.3       2R8S|1|R          2R8S|1|R
#> 4    NR_2.0_35133.2       2Z75|1|B          2Z75|1|B
#> 5    NR_2.0_66115.1       6CB3|1|B 6CB3|1|B;6CB3|1|A
#> 6    NR_2.0_31222.1       5FJC|1|A          5FJC|1|A
```

The previous object is a data-frame with three columns:

- Equivalence class
- Representative structure
- Members of the Equivalence class

Note that the output is formated according with Leontis&Zirbel nomenclature (AAAA|M|C), where "AAAA" is the PDB accession code, "M" is the model and "C" is the Chain.

The representative structures might not necessarily have the desired properties (e.g. being a protein-RNA complex). To get only representatives which are complexed with a protein, all the members of the class should be checked, which can be done with `getAltRepres`.

```
## Set progressbar=TRUE to see the progress
protrna <- getAltRepres(rnalist=rnalist, type="protRNA", progressbar=FALSE)
head(protrna)
#>    Representative Equivalence_class
#> 3        2R8S|1|R    NR_2.0_39627.3
#> 7        5B2P|1|B    NR_2.0_90160.4
#> 11       6B14|1|R    NR_2.0_27557.1
#> 13       5B2R|1|A    NR_2.0_33394.3
#> 14       3U4M|1|B    NR_2.0_73731.2
#> 15       5CCB|1|N    NR_2.0_72172.1
```

Many equivalence classes will lack a proper representative, so a NA value is returned. To get the definite list in a more friendly format and removing missing data, use `represAsDataFrame`.

```
nrlist <- represAsDataFrame(protrna)
head(nrlist)
#>   Equivalence_class  pdb model chain
#> 1    NR_2.0_55702.1 1C9S     1     W
#> 2   NR_2.0_14136.11 1DFU     1     M
#> 3   NR_2.0_14136.12 1DFU     1     N
#> 4    NR_2.0_18996.1 1DI2     1     E
#> 5    NR_2.0_50004.1 1FXL     1     B
#> 6    NR_2.0_61174.1 1H2C     1     R
```

8.2 Queries

When analysing a dataset, a particular piece of data might be useful, such as the file formats available or if a particular ligand is present or not in all the pdb files. A handy function to approach this need is `applyToPDB`. Here it's shown how to get the structures in the previous list that contain Mg ions:

```
## Set progressbar=TRUE to see the progress
nrlist_mg <- applyToPDB(FUN=hasHetAtm, listpdb=nrlist$pdb,
                        hetAtms="MG", progressbar=FALSE)
nrlist <- cbind(nrlist, Mg=nrlist_mg[, 2])
head(nrlist)
#>   Equivalence_class  pdb model chain    Mg
#> 1    NR_2.0_55702.1 1C9S     1     W FALSE
#> 2   NR_2.0_14136.11 1DFU     1     M  TRUE
#> 3   NR_2.0_14136.12 1DFU     1     N  TRUE
#> 4    NR_2.0_18996.1 1DI2     1     E FALSE
#> 5    NR_2.0_50004.1 1FXL     1     B FALSE
#> 6    NR_2.0_61174.1 1H2C     1     R FALSE

## To see only the structures containing Mg use
head(nrlist[nrlist$Mg == TRUE, ])
#>    Equivalence_class  pdb model chain   Mg
#> 2    NR_2.0_14136.11 1DFU     1     M TRUE
#> 3    NR_2.0_14136.12 1DFU     1     N TRUE
#> 7     NR_2.0_70420.1 1J1U     1     B TRUE
#> 9     NR_2.0_26264.1 1JID     1     B TRUE
#> 19    NR_2.0_18961.1 1WMQ     1     D TRUE
#> 20    NR_2.0_19039.1 1WPU     1     C TRUE
```

In the example, the argument hetAtms is passed to the hasHetAtm function.

8.3 Analysing datasets

When you have your desired list of PDB IDs, the function `pipeNucData` and `pipeProtNucData` can extract multiple structural parameters for all the dataset.

```
## After the download is finished, this dataset is analysed in less than 2 min
## (single core, intel i5 2.3Ghz). Set progressbar=TRUE to see the progress.
ntinfo <- pipeNucData(pdbID=nrlist$pdb,
                      model=nrlist$model,
                      chain=nrlist$chain,
                      progressbar=FALSE, cores=2)
#> [1] "Download completed, saved in:  /tmp/RtmpYEwKZB"
str(ntinfo)
#> 'data.frame':    3130 obs. of  83 variables:
#>  $ ntID               : int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ pdbID              : chr  "1C9S" "1C9S" "1C9S" "1C9S" ...
#>  $ model              : chr  "1" "1" "1" "1" ...
#>  $ chain              : chr  "W" "W" "W" "W" ...
#>  $ resno              : chr  "101" "102" "103" "104" ...
#>  $ insert             : chr  "?" "?" "?" "?" ...
#>  $ base_type          : Factor w/ 2 levels "pu","py": 1 1 2 1 1 1 1 2 1 1 ...
#>  $ resid              : chr  "G" "A" "U" "G" ...
#>  $ ntindex            : int  1 2 3 4 5 6 7 8 9 10 ...
#>  $ localenv           : Factor w/ 207 levels "5'-G-A","G-A-U",..: 1 2 3 4 5 6 2 3 4 5 ...
#>  $ first              : logi  TRUE FALSE FALSE FALSE FALSE FALSE ...
#>  $ last               : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
#>  $ P                  : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ O5p                : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ C5p                : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ C4p                : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ C3p                : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ O3p                : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ C2p                : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ C1p                : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ O4p                : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ N1                 : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ N9                 : logi  TRUE TRUE FALSE TRUE TRUE TRUE ...
#>  $ C2                 : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ C4                 : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ C6                 : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ H2p                : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
#>  $ O2p                : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ HO2p               : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
#>  $ lastP              : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
#>  $ big_b              : logi  FALSE FALSE TRUE TRUE FALSE FALSE ...
#>  $ dist.pre_O3p.P     : num  NA 1.61 1.61 1.61 1.61 ...
#>  $ dist.P.O5p         : num  1.61 1.61 1.61 1.59 1.59 ...
#>  $ dist.O5p.C5p       : num  1.45 1.43 1.45 1.43 1.44 ...
#>  $ dist.C5p.C4p       : num  1.52 1.52 1.51 1.53 1.53 ...
#>  $ dist.C4p.C3p       : num  1.53 1.53 1.53 1.53 1.53 ...
#>  $ dist.C3p.O3p       : num  1.43 1.42 1.42 1.42 1.42 ...
#>  $ dist.C3p.C2p       : num  1.52 1.53 1.52 1.53 1.52 ...
#>  $ dist.C2p.C1p       : num  1.54 1.52 1.53 1.53 1.53 ...
#>  $ dist.C1p.O4p       : num  1.4 1.42 1.42 1.41 1.42 ...
#>  $ dist.O4p.C4p       : num  1.46 1.46 1.46 1.46 1.46 ...
#>  $ dist.C1p.Nbase     : num  1.47 1.49 1.49 1.49 1.49 ...
```

```
#>  $ Break               : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
#>  $ puc_valid           : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ chi_valid           : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ kappa_valid         : logi  FALSE FALSE FALSE FALSE FALSE FALSE ...
#>  $ base_exists         : logi  TRUE TRUE TRUE TRUE TRUE TRUE ...
#>  $ eta_valid           : logi  FALSE FALSE FALSE FALSE FALSE TRUE ...
#>  $ theta_valid         : logi  TRUE FALSE FALSE FALSE TRUE TRUE ...
#>  $ dist.O3p.post_P     : num  1.61 1.61 1.61 1.61 1.62 ...
#>  $ dist.C1p.N9         : num  1.47 1.49 NA 1.49 1.49 ...
#>  $ dist.C1p.N1         : num  5.08 5.18 1.49 5.08 5.19 ...
#>  $ angle.P.O5p.C5p     : num  114 121 111 122 123 ...
#>  $ angle.O5p.C5p.C4p   : num  108 112 112 106 105 ...
#>  $ angle.C5p.C4p.C3p   : num  115 119 114 115 113 ...
#>  $ angle.C5p.C4p.O4p   : num  107 106 108 106 108 ...
#>  $ angle.C4p.C3p.O3p   : num  110 111 109 112 111 ...
#>  $ angle.C4p.C3p.C2p   : num  102 104 103 103 104 ...
#>  $ angle.C4p.O4p.C1p   : num  109 110 109 109 111 ...
#>  $ angle.C3p.O3p.post_P: num  120 122 119 122 118 ...
#>  $ angle.C3p.C2p.C1p   : num  103.2 99.7 102.6 101.2 102.6 ...
#>  $ angle.C2p.C1p.O4p   : num  109 108 105 107 107 ...
#>  $ angle.O3p.C3p.C2p   : num  112 110 116 109 110 ...
#>  $ angle.C3p.C2p.O2p   : num  114 113 111 111 113 ...
#>  $ angle.C1p.C2p.O2p   : num  105 113 114 112 107 ...
#>  $ alpha               : num  NA 295 311 329 277 ...
#>  $ beta                : num  161 190 175 172 158 ...
#>  $ gamma               : num  52 38.2 51.5 22.4 24.4 ...
#>  $ delta               : num  92.5 93.3 133.7 146.3 94.6 ...
#>  $ epsilon             : num  211 226 209 245 215 ...
#>  $ zeta                : num  279.2 300.9 141 85.2 271.8 ...
#>  $ nu0                 : num  0.863 12.774 333.108 338.676 5.655 ...
#>  $ nu1                 : num  339.5 329.2 36.9 33.7 336 ...
#>  $ nu2                 : num  30.4 36.7 328.2 327.9 32.3 ...
#>  $ nu3                 : num  329 329.3 17 21.2 330.4 ...
#>  $ nu4                 : num  19.17 11.39 6.21 359.84 15.11 ...
#>  $ kappa               : num  NA NA NA NA NA NA NA NA NA NA ...
#>  $ eta                 : num  NA 174 179 216 151 ...
#>  $ theta               : num  217.1 240 113.3 91.8 218.9 ...
#>  $ chi                 : num  259 203 254 246 266 ...
#>  $ pu_amp              : num  32.7 37.6 37 35 33.2 ...
#>  $ pu_phase            : num  16.8 359.13 152.17 161.72 8.53 ...
#>  $ Dp                  : num  4.35 4.75 2.26 1.58 4.49 ...
```

In this case, `pipeNucData` is downloading each structure from internet, but you can also download them manually before execution, save them in a directory and provide it to the function. When reading files locally, the function is obviously faster. The file format can either be pdb or cif, being the cif files the recommended option (they can also be read when compressed in .gz). To make it work with local files, the extension should also be provided (e.g. ".cif.gz", ".cif" or ".pdb").

```
ntinfo <- pipeNucData(pdbID=nrlist$pdb,
                      model=nrlist$model,
```

```
                            chain=nrlist$chain,
                            progressbar=FALSE, cores=2,
                            path="/your/path/to/the/dataset/", extension=".cif.gz")
```

The second dataset analysis available in the dataset is designed to study the interactions between proteins and nucleic acids, and would be executed as follows:

```
## After the download is finished, this dataset is analysed in less than 1 min
## (single core, intel i5 2.3Ghz). Set progressbar=TRUE to see the progress.
aantinfo <- pipeProtNucData(pdbID=nrlist$pdb,
                            model=nrlist$model,
                            chain=nrlist$chain,
                            progressbar=FALSE, cores=2)
#> [1] "Download completed, saved in:  /tmp/RtmpYEwKZB"
str(aantinfo)
#> 'data.frame':    85187 obs. of  19 variables:
#>  $ pdbID   : Factor w/ 169 levels "1C9S","1DFU",..: 1 1 1 1 1 1 1 1 1 ...
#>  $ model   : Factor w/ 1 level "1": 1 1 1 1 1 1 1 1 1 ...
#>  $ eleno_A : num  1 2 3 4 5 6 7 8 9 10 ...
#>  $ eleno_B : num  7303 7319 7303 7317 7303 ...
#>  $ distance: num  8.57 8.41 9.13 7.25 6.36 ...
#>  $ elety_A : chr  "P" "OP1" "OP2" "O5'" ...
#>  $ resid_A : chr  "G" "G" "G" "G" ...
#>  $ resno_A : int  101 101 101 101 101 101 101 101 101 101 ...
#>  $ chain_A : chr  "W" "W" "W" "W" ...
#>  $ insert_A: chr  "?" "?" "?" "?" ...
#>  $ alt_A   : chr  "." "." "." "." ...
#>  $ b_A     : num  45.1 44.4 46.2 46.6 46.5 ...
#>  $ elety_B : chr  "CB" "CE2" "CB" "CD2" ...
#>  $ resid_B : chr  "ARG" "PHE" "ARG" "PHE" ...
#>  $ resno_B : int  31 32 31 32 31 31 31 32 32 32 ...
#>  $ chain_B : chr  "L" "L" "L" "L" ...
#>  $ insert_B: chr  "?" "?" "?" "?" ...
#>  $ alt_B   : chr  "." "." "." "." ...
#>  $ b_B     : num  33.7 32.5 33.7 36.6 33.7 ...
```

9 The `dssr` wrapper: getting the base pairs

The DSSR software (Disecting the Spatial Structure of RNA) (Lu, Bussemaker, and Olson 2015) represents an unvaluable resource to handle RNA structures. Some of the functions of veriNA3d overlap with the functionalities of DSSR, and both applications provide unique different features. We implement a wrapper to execute DSSR directly from R and get the best of both worlds in one place.

Note that installing veriNA3d does not automatically install DSSR, since we don't redistribute third-party software. Before any user can use our wrapper, the `dssr` function, DSSR should be installed separately. To address this installation we redirect you to the DSSR manual, where anyone can find the specific instructions for their system. Once DSSR is installed and working in your computer, you will also be able to use it with our wrapper. If the DSSR executable (named x3dna-dssr) is in your path, `dssr` will find it automatically. If the wrapper does not find it, you can still use it specifying the absolute path to the executable with the argument exefile. Find more information running `?dssr`.

One of the DSSR capabilities that users might be interested in is the detection and classification of base pairs. The following code shows a simple example. The output of the `dssr` wrapper is an object got from the json DSSR output. From R, json objects are parsed in the form of a tree of lists, with different types of information. Most of the interesting data is under the list models, sublist parameters, as shown herein.

```
## Execute dssr, the wrapper will donwload the (mmCIF) file if necessary
rna <- dssr("1bau")

## The contents of the `rna` object can be seen with
names(rna)
#> [1] "input_file" "num_models" "models"     "start_at"   "finish_at"
#> [6] "time_used"

## Then, the contents of the list model inside `rna`
names(rna$model)
#> [1] "index"      "model"      "parameters"

## And, inside the sublist parameters
names(rna$models$parameters)
#>  [1] "num_pairs"         "pairs"             "num_isoCanonPairs"
#>  [4] "isoCanonPairs"     "num_coaxStacks"    "coaxStacks"
#>  [7] "num_stacks"        "stacks"            "nonStack"
#> [10] "num_atom2bases"    "atom2bases"        "num_hairpins"
#> [13] "hairpins"          "num_iloops"        "iloops"
#> [16] "num_junctions"     "junctions"         "num_kissingLoops"
#> [19] "kissingLoops"      "dbn"               "chains"
#> [22] "num_nts"           "nts"               "num_hbonds"
#> [25] "hbonds"            "refCoords"         "metadata"

## The number 1 herein selects the model of the structure, and is needed even
## in XRAY structures.
class(rna$models$parameters$pairs[[1]])
#> [1] "data.frame"
```

```
## Finally, to get the data about base pairs, use:
rna$models$parameters$pairs[[1]][, c("index", "nt1", "nt2", "LW")]
#>    index     nt1     nt2  LW
#> 1      1  1:A.G1 1:A.C23 cWW
#> 2      2  1:A.G2 1:A.C22 cWW
#> 3      3  1:A.C3 1:A.G21 cWW
#> 4      4  1:A.A4 1:A.U20 cWW
#> 5      5  1:A.A5 1:A.U19 cWW
#> 6      6  1:A.U6 1:A.G18 cWW
#> 7      7  1:A.G7 1:A.C17 cSW
#> 8      8  1:A.A9 1:B.A16 cWS
#> 9      9 1:A.G10 1:B.C15 cWW
#> 10    10 1:A.G12 1:B.C13 cWW
#> 11    11 1:A.C13 1:B.G12 cWW
#> 12    12 1:A.G14 1:B.C11 cWW
#> 13    13 1:A.C15 1:B.G10 cWW
#> 14    14 1:A.A16  1:B.A9 cSW
#> 15    15  1:B.G1 1:B.C23 cWW
#> 16    16  1:B.G2 1:B.C22 cWW
#> 17    17  1:B.C3 1:B.G21 cWW
#> 18    18  1:B.A4 1:B.U20 cWW
#> 19    19  1:B.A5 1:B.U19 cWW
#> 20    20  1:B.U6 1:B.G18 cWW
#> 21    21  1:B.G7 1:B.C17 cSW
```

References

Bottaro, S., G. Bussi, G. Pinamonti, S. Reiber, W. Boomsma, and K. Lindorff-Larsen. 2018. "Barnaba: Software for Analysis of Nucleic Acids Structures and Trajectories." *RNA*, no. Epub (November).

Bottaro, S., F. Di Palma, and G Bussi. 2014. "The Role of Nucleobase Interactions in RNA Structure and Dynamics." *Nucleic Acids Research* 42 (21): 13306–14.

Darré, L., I. Ivani, P.D. Dans, H. Gómez, A. Hospital, and M. Orozco. 2016. "Small Details Matter: The 2'-Hydroxyl as a Conformational Switch in RNA." *Journal of the American Chemical Society* 138 (50): 16355–63.

Grant, B.J., A.P.C. Rodrigues, K.M. ElSawy, J.A. McCammon, and L.S.D. Caves. 2006. "Bio3d: An R Package for the Comparative Analysis of Protein Structures." *Bioinformatics* 22 (21): 2695–6.

Humphrey, W., A. Dalke, and K. Schulten. 1996. "VMD: Visual Molecular Dynamcis." *Journal of Molecular Graphics* 14 (1): 33–38.

Jain, S., D.C. Richardson, and J.S. Richardson. 2015. "Computational Methods for RNA Structure Validation and Improvement." In *Methods in Enzymology*, edited by S.A. Woodson and F.H.T. Allain, 558:181–212. Elsevier.

Leontis, N.B., and C.L. Zirbel. 2012. "Nonredundant 3D Structure Datasets for RNA Knowledge Extraction and Benchmarking." In *RNA 3D Structure Analysis and Prediction*, edited by N. Leontis and E. Westhof, 27:281–98. Springer Berlin Heidelberg.

Lu, X.J., H.J. Bussemaker, and W.K. Olson. 2015. "DSSR: An Integrated Software Tool for Dissection the Spatial Structure of RNA." *Nucleic Acids Research* 43 (21): e142.

Murray, L.J.W., W.B. Arendall III, D.C. Richardson, and J.S. Richardson. 2003. "RNA Backbone Is Rotameric." *Proceedings of the National Academy of Sciences of the United States of America* 100 (24): 13904–9.

Velankar, S., G. Van Ginkel, Y. Alhroub, G.M. Battle, J.M. Berrisford, M.J. Conroy, J.M. Dana, et al. 2015. "PDBe: Improved Accessibility of Macromolecular Structure Data from PDB and EMDB." *Nucleic Acids Research* 44 (D1): D385–395.

Wang, J., Y. Zhao, C. Zhu, and Y. Xiao. 2015. "3dRNAscore: A Distance and Torsion Angle Dependent Evaluation Function of 3D RNA Structures." *Nucleic Acids Research* 43 (10): e63.

Wang, Z., and J. Xu. 2011. "A Conditional Random Fields Method for RNA Sequence-Structure Relationship Modeling and Conformation Sampling." *Bioinformatics* 27 (13): i102–10.