Supplementary Data

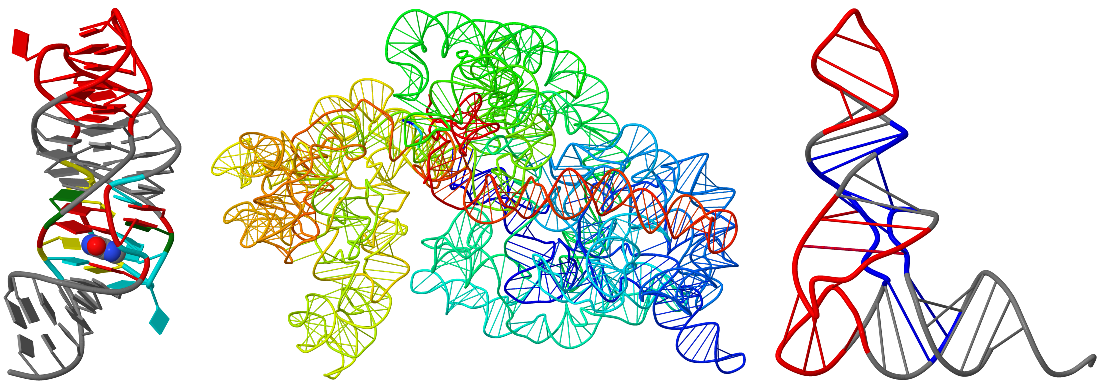
DSSR-Enhanced Visualization of Nucleic Acid Structures in Jmol

by

Robert M. Hanson and **Xiang-Jun Lu**

`DSSR v1.6.8, released on 2017-03-28`

`Jmol v14.15.1, released on 2017-04-27`



1 DSSR commands for integration to Jmol/JSmol

DSSR is stand-alone, command-line program written in ANSI C. The binary executables are only ∼1MB in size, and self-contained. With zero dependencies, no setup or configuration, it is trivial to get DSSR up and running. DSSR uncovers a wide range of RNA/DNA structural features in a consistent, easily accessible framework. It possesses a much richer set of functionalities for nucleic acid structural analysis (see the DSSR User Manual) than any other existing tools. Moreover, the program is efficient and robust, making it an ideal component to be integrated into other pipelines.

The DSSR-Jmol integration covers the most fundamental features of what DSSR has to offer, as outlined in the main text. This work fills a gap in RNA/DNA structural bioinformatics, since no such functionality is currently available in other popular molecular viewers (to the best of our knowledge). It brings the molecular graphics of 3D RNA structures to a similar level as that for proteins, and enables a much deeper analysis of structural characteristics.

Technically, the DSSR-Jmol integration benefits from the simple *structured* JSON format, and the clearly delineated unit identifier proposed by the Leontis-Zirbel group to unambiguously identify nucleotides/atoms. From the DSSR side, the interface is specified via the `--json=ebi` command-line option.

The DSSR-generated JSON output is in a compact, one-line form. For the following illustrations, we employ the command-line JSON processor jq (v1.5) to parse the DSSR output, using yeast tRNA^{Phe} (PDB id: `1ehz`) as an example. The commands and results presented here should give users a better understanding of the mechanics that connect DSSR and Jmol/JSmol.

1.1 Overview of DSSR with `--json=ebi`

Let the coordinate file of `1ehz` in mmCIF format be `1ehz.cif` (the PDB-formatted `1ehz.pdb` file is also fine), the DSSR commands for the Jmol/JSmol integration are as follows:

```
x3dna-dssr --json=ebi -i=1ehz.cif # default to stdout, which can be chained to jq
x3dna-dssr --json=ebi -i=1ehz.cif | jq keys # top-level keys, e.g., 'paths', 'pairs' etc.
x3dna-dssr --json=ebi -i=1ehz.cif -o=1ehz.json # save to a file
```

The contents of the right column in Table 1 of the main text were generated using the commands listed below. Note the `jq -c` option which produces a compact instead of pretty-printed output, to save space.

```
x3dna-dssr --json=ebi -i=1ehz.cif | jq -c '.paths | keys_unsorted'
x3dna-dssr --json=ebi -i=1ehz.cif | jq -c .counts
x3dna-dssr --json=ebi -i=1ehz.cif | jq -c .pairs[0]
x3dna-dssr --json=ebi -i=1ehz.cif | jq -c '.nts[9] | {nt_name, nt_id, is_modified, chi, puckering}'
```

1.2 The `paths` object (accessible features)

```
x3dna-dssr --json=ebi -i=1ehz.cif | jq .paths
```

Running the above command gives the following output, which lists the 16 DSSR-derived features accessible to Jmol/JSmol (e.g., `pairs`) and their corresponding keys for unit ids (e.g., `nt1,nt2` for `pairs`, see below). This list is fixed (per the current version), and does not change with each analyzed structure.

```
{
  "pairs": "nt1,nt2",
  "multiplets": "nts_long",
  "helices": "pairs.nt1,nt2",
  "stems": "pairs.nt1,nt2",
  "isoCanonPairs": "nt1,nt2",
  "coaxStacks": "stem_indices.pairs.nt1,nt2",
  "hairpins": "nts_long",
  "bulges": "nts_long",
  "iloops": "nts_long",
  "junctions": "nts_long",
  "kissingLoops": "hairpin_indices.nts_long",
  "ssSegments": "nts_long",
  "stacks": "nts_long",
  "nonStack": "nts_long",
  "hbonds": "res_long;atom1_id,atom2_id",
  "nts": "nt_id"
}
```

Within Jmol/JSmol, these 16 keys can serve as selection keywords. For example, the script `select hairpins` will pick up all nucleotides involved in hairpin loops (if any, see `counts` below).

1.3 The `counts` object (actually available features)

```
x3dna-dssr --json=ebi -i=1ehz.cif | jq .counts
```

Running the above command gives the following output, which contains the actual DSSR-derived features in the analyzed structure (`1ehz.cif`). The keys of the `counts` list are a subset of the 16 keywords in `paths`, and change with each analyzed structure. For example, `1ehz` has 14 features among the 16 shown above for `paths`.

```
{
  "pairs": 34,
  "multiplets": 4,
  "helices": 2,
  "stems": 4,
  "isoCanonPairs": 1,
  "coaxStacks": 2,
  "hairpins": 3,
  "junctions": 1,
  "kissingLoops": 1,
  "ssSegments": 1,
  "stacks": 11,
  "nonStack": 4,
  "hbonds": 118,
  "nts": 76
}
```

1.4 Queryable properties

1.4.1 In `pairs`

```
x3dna-dssr --json=ebi -i=1ehz.cif | jq .pairs[0,3,7,8,11]
```

Running the above command extracts five base pairs with indices `1,4,8,9,12` as shown in the following output. Note that arrays in JavaScript (`jq`) are 0-indexed. The keys `nt1` and `nt2` designate the unit ids of the two nucleotides forming a pair, and they match the value of the `pairs` key in the `paths` object (`"pairs":  "nt1,nt2"`, see above). The other keys (`bp`, `name`, `Saenger`, `LW`, and `DSSR`) correspond to the abbreviated pair type (M±N), common name (if any), classifications by Saenger, Leontis-Westhof (LW), or DSSR, respectively. See the DSSR User Manual for details.

```
{
  "index": 1,
  "nt1": "|1|A|G|1||||",
  "nt2": "|1|A|C|72||||",
  "bp": "G-C",
  "name": "WC",
  "Saenger": "19-XIX",
  "LW": "cWW",
  "DSSR": "cW-W"
}
```

```
{
  "index": 4,
  "nt1": "|1|A|G|4||||",
  "nt2": "|1|A|U|69||||",
  "bp": "G-U",
  "name": "Wobble",
  "Saenger": "28-XXVIII",
  "LW": "cWW",
  "DSSR": "cW-W"
}
{
  "index": 8,
  "nt1": "|1|A|U|8||||",
  "nt2": "|1|A|A|14||||",
  "bp": "U-A",
  "name": "rHoogsteen",
  "Saenger": "24-XXIV",
  "LW": "tWH",
  "DSSR": "tW-M"
}
{
  "index": 9,
  "nt1": "|1|A|U|8||||",
  "nt2": "|1|A|A|21||||",
  "bp": "U+A",
  "name": "--",
  "Saenger": "n/a",
  "LW": "tSW",
  "DSSR": "tm+W"
}
{
  "index": 12,
  "nt1": "|1|A|2MG|10||||",
  "nt2": "|1|A|G|45||||",
  "bp": "g+G",
  "name": "--",
  "Saenger": "n/a",
  "LW": "cHS",
  "DSSR": "cM+m"
}
```

Among the five pairs, note the following characteristics:

`"index":1` – a Watson-Crick pair (G1–C72)

`"index":4` – a Wobble pair (G4–U69)

`"index":8` – a reverse Hoogsteen pair (U8–A14)

`"index":9` – an unnamed pair (U8+A21)

`"index":12` – another unnamed pair with a modified base (2MG10+G45)

1.4.2 In `nts`

```
x3dna-dssr --json=ebi -i=1ehz.cif | jq .nts[9]
```

Running the command above extracts numerous structural parameters for 2MG10 (with index `10`) as shown in the following output. As with `pairs`, the keys in `nts` can be employed by the Jmol SQL for DSSR, leading to numerous possibilities to query DSSR-derived parameters that characterize nucleotides.

```
{
  "index": 10,
  "index_chain": 10,
  "chain_name": "A",
  "nt_resnum": 10,
  "nt_name": "2MG",
  "nt_code": "g",
  "is_modified": true,
  "nt_id": "|1|A|2MG|10||||",
  "dbn": "(",
  "alpha": 177.814,
  "beta": 147.203,
  "gamma": 60.066,
  "delta": 89.323,
  "epsilon": -126.196,
  "zeta": -88.738,
  "epsilon_zeta": -37.459,
  "bb_type": "..",
  "chi": 169.599,
  "baseSugar_conf": "anti",
  "form": "A",
  "ssZp": 4.682,
  "Dp": 4.635,
  "splay_angle": 23.874,
  "splay_distance": 3.63,
  "splay_ratio": 0.208,
  "eta": 27.783,
  "theta": -130.257,
  "eta_prime": 97.236,
  "theta_prime": -130.105,
  "eta_base": 134.838,
  "theta_base": -110.259,
  "v0": 7.818,
  "v1": -28.008,
  "v2": 36.712,
  "v3": -32.963,
  "v4": 15.862,
  "amplitude": 36.954,
  "phase_angle": 6.563,
  "puckering": "C3'-endo",
  "sugar_class": "~C3'-endo",
  "bin": "23p",
  "cluster": "2g",
  "suiteness": 0.64,
  "filter_rmsd": 0.018,
  "frame": {
    "rsmd": 0.018,
    "origin": [
      65.696,
      45.135,
      18.125
    ],
    "x_axis": [
      0.69,
      0.714,
      -0.117
    ],
    "y_axis": [
      -0.707,
      0.7,
      0.101
    ],
    "z_axis": [
      0.154,
      0.013,
      0.988
    ]
  }
}
```

1.4.3 In `stems`

```
x3dna-dssr --json=ebi -i=1ehz.cif | jq .stems[0]
```

Running the above command extracts the acceptor stem (with index `1`) in `1ehz`. Note that the `pairs` key and its sub-keys `nt1,nt2` match the `"stems":"pairs.nt1,nt2"` entry in `paths` for selecting unit ids. As with `pairs`, the keys in `stems` (as well as `helices`) can be employed by the Jmol SQL for DSSR, opening numerous possibilities to query desired features.

```
{
  "index": 1,
  "helix_index": 1,
  "strand1": "GCGGAUU",
  "strand2": "CGCUUAA",
  "bp_type": "|||||||",
  "helix_form": "AA....",
  "num_pairs": 7,
  "pairs": [
    {
      "index": 1,
      "nt1": "|1|A|G|1||||",
      "nt2": "|1|A|C|72||||",
      "bp": "G-C",
      "name": "WC",
      "Saenger": "19-XIX",
      "LW": "cWW",
      "DSSR": "cW-W"
    },
    {
      "index": 2,
      "nt1": "|1|A|C|2||||",
      "nt2": "|1|A|G|71||||",
      "bp": "C-G",
      "name": "WC",
      "Saenger": "19-XIX",
      "LW": "cWW",
      "DSSR": "cW-W"
    },
    {
      "index": 3,
      "nt1": "|1|A|G|3||||",
      "nt2": "|1|A|C|70||||",
      "bp": "G-C",
      "name": "WC",
      "Saenger": "19-XIX",
      "LW": "cWW",
      "DSSR": "cW-W"
    },
    {
      "index": 4,
      "nt1": "|1|A|G|4||||",
      "nt2": "|1|A|U|69||||",
      "bp": "G-U",
      "name": "Wobble",
      "Saenger": "28-XXVIII",
      "LW": "cWW",
      "DSSR": "cW-W"
    },
    {
      "index": 5,
      "nt1": "|1|A|A|5||||",
      "nt2": "|1|A|U|68||||",
      "bp": "A-U",
      "name": "WC",
      "Saenger": "20-XX",
      "LW": "cWW",
      "DSSR": "cW-W"
    },
    {
      "index": 6,
      "nt1": "|1|A|U|6||||",
      "nt2": "|1|A|A|67||||",
      "bp": "U-A",
      "name": "WC",
      "Saenger": "20-XX",
      "LW": "cWW",
      "DSSR": "cW-W"
    },
    {
      "index": 7,
      "nt1": "|1|A|U|7||||",
      "nt2": "|1|A|A|66||||",
      "bp": "U-A",
      "name": "WC",
      "Saenger": "20-XX",
      "LW": "cWW",
      "DSSR": "cW-W"
    }
  ]
}
```

1.4.4 In `coaxStacks`

```
x3dna-dssr --json=ebi -i=1ehz.cif | jq .coaxStacks[0]
```

Running the above command extracts the coaxial-stacking interactions within the first helix that incorporates the acceptor stem (with index `1`) and the T stem (with index `4`) in `1ehz`. The `stem_indices` key provides an array of pointers to the corresponding stems, linking to pairs detailed above. The `"coaxStacks":"stem_indices.pairs.nt1,nt2"` entry in `paths` shows the hierarchy of the chain "coaxStacks → stems → pairs → nt1,nt2" to the unit ids.

```
{
  "index": 1,
  "helix_index": 1,
  "num_stems": 2,
  "stem_indices": [
    1,
    4
  ]
}
```

1.4.5 In `junctions`

```
x3dna-dssr --json=ebi -i=1ehz.cif | jq .junctions[0]
```

Running the above command extracts the four-way junction in `1ehz`, corresponding to the central roundabout in the classic tRNA cloverleaf secondary structure diagram. Note the `nts_long` key whose value lists the unit ids of all the nucleotides in the 'closed' junction loop.

As with other DSSR-derived features outlined above, the keys in `junctions` can be employed by the Jmol SQL for DSSR, leading to many queries of practical significance. As a simple example, one can use the `num_stems` key to find all three-way (or four-way, etc.) junctions in a given structure.

```
{
  "index": 1,
  "type": "4-way junction",
  "bridging_nts": [
    2,
    1,
    5,
    0
  ],
  "stem_indices": [
    1,
    2,
    3,
    4
  ],
  "summary": "[4] 2 1 5 0 [A.7 A.66 A.10 A.25 A.27 A.43 A.49 A.65] 7 4 4 5",
  "num_nts": 16,
  "nts_short": "UUAgCgCGAGgUCcGA",
  "nts_long": "|1|A|U|7||||,|1|A|U|8||||,|1|A|A|9||||,|1|A|2MG|10||||,|1|A|C|25||||,|1|A|M2G|26||||,|1|A|C
      ↪ |27||||,|1|A|G|43||||,|1|A|A|44||||,|1|A|G|45||||,|1|A|7MG|46||||,|1|A|U|47||||,|1|A|C|48||||,|1|A
      ↪ |5MC|49||||,|1|A|G|65||||,|1|A|A|66||||",
  "num_stems": 4,
  "bridges": [
    {
      "index": 1,
      "num_nts": 2,
      "nts_short": "UA",
      "nts_long": "|1|A|U|8||||,|1|A|A|9||||"
    },
    {
      "index": 2,
      "num_nts": 1,
      "nts_short": "g",
      "nts_long": "|1|A|M2G|26||||"
    },
    {
      "index": 3,
      "num_nts": 5,
      "nts_short": "AGgUC",
      "nts_long": "|1|A|A|44||||,|1|A|G|45||||,|1|A|7MG|46||||,|1|A|U|47||||,|1|A|C|48||||"
    },
    {
      "index": 4,
      "num_nts": 0,
      "nts_short": "",
      "nts_long": ""
    }
  ]
}
```

1.4.6 In other features

Structural properties in `helices`, `kissingLoops` etc. can be similarly queried.

2 DSSR web-API used by Jmol/JSmol

2.1 DSSR analysis and annotation using PDB IDs

The Jmol `LOAD` command can be combined with a request for DSSR annotation, as shown below:

```
LOAD =1ehz/dssr # https://files.rcsb.org/download/1ehz.pdb
```

The command carries out two web calls. First, it retrieves atomic coordinates in PDB format from RCSB. Jmol then retrieves a JSON output file from the DSSR server via a call to `http://dssr-jmol.x3dna.org/report.php?id=1ehz&opts=--json=ebi`. This creates for the specified model a Jmol variable `_M.dssr`, which contains all of the data discussed in Section 1 and is used for analysis and visualization of the structural features discussed in Section 3.

2.2 DSSR analysis from PDB-formatted data

Jmol can also send a full set or subset of structural data in PDB format to the DSSR server in order to retrieve a "custom" DSSR analysis. This is accomplished using the `SELECT` and `CALCULATE` commands. For example, DSSR analysis for a single model of the NMR ensemble `2krl` can be created as follows:

```
LOAD =2krl
SELECT model=6
CALCULATE structure dssr
```

The `SELECT` command specifies that the PDB data should be created only for the sixth model in the ensemble. The `CALCULATE` command then creates a PDB-formatted string consisting solely of `MODEL`, `ATOM`, and `HETATM` records for those selected atoms and then sends a request to the DSSR server using the API call `http://dssr-jmol.x3dna.org/report.php` with POST data `opts=--json=ebi` and `model=%MODEL` where `%MODEL` is the PDB-formatted string. In this way, Jmol can retrieve a DSSR analysis for any subset of a model, whether it be one particular model in an ensemble or one particular configuration of atoms involving alternative location indicators.

2.3 Reading DSSR JSON analysis data directly

DSSR data in JSON format can be read by Jmol directly by setting `_M.dssr` to a value found in a file or in a variable:

```
# x3dna-dssr -i=1d66.pdb --json=ebi -o=1d66.dssr

load 1d66.pdb
model 1 property dssr "1d66.dssr" # read directly from file '1d66.dssr'

zap
load 1d66.pdb
x = load("1d66.dssr") # or by first loading data to a variable
model 1 property dssr @x
```

Such data can be obtained directly from running `x3dna-dssr` on a local machine (line no.1) as described in Section 1, or that saved from some previous Jmol session using the `WRITE` command:

```
load =1ehz/dssr
x = _M.dssr
WRITE var x "1ehz.dssr"
```

3 Jmol/JSmol support for DSSR

The following section documents how Jmol/JSmol parses DSSR-derived structural features in JSON.

3.1 DSSR in Jmol/JSmol

After loading a file from the RCSB PDB with the `/dssr` attribute, `LOAD =1ehz/dssr`, Jmol/JSmol automatically accesses the DSSR information to provide the following searchable terms:

```
bulges
coaxStacks
hairpins
hbonds
helices
iloops
isoCanonPairs
junctions
kissingLoops
multiplets
nonStack
nts
pairs
ssSegments
stacks
stems
```

For example:

```
SELECT hairpins
SELECT ADD helices
DISPLAY REMOVE nonStack
COLOR {kissingLoops} red

COLOR PROPERTY DSSR stems # color each stem differently
COLOR NUCLEIC # color atoms based on single-letter DSSR nts.nt_code
SET cartoonBlocks # render each base as a DSSR-standard block
```

Other settings are also available:

```
SET cartoonSteps # render each base pair as a single rod
SET cartoonBlockHeight 1.0 # set the height (in Angstroms) of base blocks
SET cartoonLadders # render each base as a single rod
SET cartoonRibose # render each ribose ring
SET cartoonBaseEdges # render each base as a triangle, following Leontis-Westhof
```

Capitalization is irrelevant in Jmol command tokens or atom selections. So `select HAIRPINS` is the same as `SELECT hairPins`. Note the added braces `{}` in the `COLOR` command (line no.4). This is standard for most commands. Only the dedicated atom selection commands `SELECT`, `DELETE`, `DISPLAY`, `FIX`, `HIDE`, and `ZAP` work without these braces.

Furthermore, one can add `--xxx[=yyy]` flags to the Jmol `LOAD` command to be passed onto DSSR, allowing for future expansion. For example,

```
LOAD =1ehz/dssr--non-pair=true # detect non-pairing interactions
LOAD =1ehz/dssr--non-pair # same as above
# http://dssr-jmol.x3dna.org/report.php?id=1ehz&opts=--json=ebi%20--non-pair

LOAD =1ehz/dssr--non-pair%20--u-turn # also detect U-turn motifs
# http://dssr-jmol.x3dna.org/report.php?id=1ehz&opts=--json=ebi%20--non-pair%20--u-turn
```

3.1.1 Atom selection using `within(dssr, ...)`

In addition, atom selection from DSSR can be done using the `within()` method. In fact, the command `SELECT hairpins` is really just a shorthand for `SELECT within(dssr, "hairpins")`. The `within(dssr, ...)` syntax allows more flexibility, though. For example, using `within(dssr, ...)` one can select just one of the array elements returned by the selection. This is accomplished by appending `..n`, where `n` is an integer starting with `1`. Note that `0` requests the last array element in Jmol, not the first.

```
SELECT within(dssr, "nts") # all nucleotides
SELECT within(dssr, "nts..2") # just the second nucleotide
```

3.1.2 Unit IDs and direct atom selection

Atom selection in Jmol is based on unit ids proposed by the Leontis-Zirbel group. Here is an example unit id:

```
SELECT "|1|A|A|44||||"
```

In the `SELECT` command, Jmol parses all strings for unit ids. This means that the strings can have unrelated content without issue. This is important for *indirect* atom selection and for more general selections using the Jmol SQL for DSSR (see below), and it is what is being utilized behind the scenes with `within(dssr, ...)`.

3.2 Jmol SQL for DSSR

Jmol supports a rich query language for exploring DSSR structural annotations (with shorthands), as well as for selecting nucleotides/atoms associated with specific DSSR characteristics. It is referred to here as the "Jmol SQL for DSSR" and is described more fully in the Jmol Interactive Documentation.

3.2.1 The `_M.dssr` associative array

After loading a file from RCSB PDB with the `/dssr` attribute, `LOAD =1ehz/dssr`, the associative array `_M.dssr` holds all DSSR information. This array has the same main keys as can be used for atom selection (see above), along with some sub-keys:

```
bulges
coaxStacks
coaxStacks.stems
hairpins
hbonds
helicies
helicies.pairs
iloops
isoCanonPairs
junctions
kissingLoops
kissingLoops.hairpins
multiplets
nonStack
nts
pairs
ssSegments
stacks
stems
stems.pairs
```

Each key or sub-key is itself an array of associative arrays. For example,

```
LOAD =1ehz/dssr
PRINT _M.dssr.stems.length # 4
PRINT _M.dssr.stems[1].pairs.length # 7
PRINT _M.dssr.stems[1].pairs[1]
      # {
      # "DSSR" : "cW-W"
      # "LW" : "cWW"
      # "Saenger" : "19-XIX"
      # "bp" : "G-C"
      # "index" : 1
      # "name" : "WC"
      # "nt1" : "|1|A|G|1||||"
      # "nt2" : "|1|A|C|72||||"
      # }
```

3.2.2 Using `.select()`

The Jmol SQL for DSSR can target these values in order to extract specific array elements using the `.select()` function with a `where` clause:

```
PRINT _M.dssr.pairs.select("where name = 'Imino'")
      # {
      # "DSSR" : "cW-W"
      # "LW" : "cWW"
      # "Saenger" : "08-VIII"
      # "bp" : "g-A"
      # "index" : 21
      # "name" : "Imino"
      # "nt1" : "|1|A|M2G|26||||"
      # "nt2" : "|1|A|A|44||||"
      # }
PRINT _M.dssr.pairs.select("where name != 'WC'").count # 14
PRINT _M.dssr.coaxStacks[1].stems.select("where strand1='GACAC' or strand2='GACAC'")[1].pairs.count # 5
```

Note that this `.select()` function has nothing to do with atom selection – it is strictly an array function. All the keys (or any combination thereof) can be used in the Jmol SQL for DSSR to query base pairs (or junctions etc.) with specific characteristics. Here are some examples to illustrate the power and flexibility of the Jmol SQL for DSSR:

```
SELECT pairs # select all pairs, shorthand form
SELECT within(dssr, "pairs") # select all pairs
SELECT within(dssr, "pairs..1") # select the first pair; Jmol arrays are 1-indexed
SELECT within(dssr, "pairs WHERE index=9") # select pair with "index":9
SELECT within(dssr, "pairs WHERE name != 'WC'") # select non-WC pairs
SELECT within(dssr, "pairs WHERE name = 'WC'") # select WC pairs
SELECT within(dssr, "pairs WHERE name = 'Wobble'") # select G-U Wobble pairs

# select WC or Wobble pairs, i.e., canonical pairs
SELECT within(dssr, "pairs WHERE name = 'WC' OR name = 'Wobble'")

# select non-canonical pairs
SELECT within(dssr, "pairs WHERE name != 'WC' AND name != 'Wobble'")

# to select by "LW": "tSW", i.e. trans-Sugar-Watson pairs, per Leontis-Westhof
SELECT within(dssr, "pairs WHERE LW = 'tSW'") # e.g., U8+A21

# to select by "DSSR": "tm+W", i.e. trans-minor-Watson pairs, M+N type, per DSSR
SELECT within(dssr, "pairs WHERE DSSR = 'tm+W'") # e.g., U8+A21

SELECT within(dssr, "junctions WHERE num_stems = 3") # find all three-way junctions
```

3.2.3 Involving Jmol variables

The expressions in these selections are simply Jmol math expressions, where the variables are the keys in the associative arrays being targeted. As such, they can contain references to other Jmol variables:

```
x = "GACAC"; PRINT _M.dssr.coaxStacks[1].stems.select("where strand1=x or strand2=x")[1].pairs.count # 5
```

3.2.4 Indirect atom selection

Sub-elements of the `_M.dssr` array can be used for atom selection. This is done indirectly, using the `@...` notation with a math expression. In this case, we can use the `_M.dssr` array in the expression:

```
SELECT @{_M.dssr.pairs.select("where name != 'WC'")}
```

Note that unit ids are string values, while `_M.dssr.pairs` is not. This is not an issue; Jmol will convert the array to its string value and find all unit ids present in that string.

3.2.5 Using `within(dssr, ...)` and the Jmol SQL for DSSR

Atom selection can be done using the `within()` atom selection method in conjunction with the Jmol SQL for DSSR. In this case, the notation can be simplified:

```
SELECT within(dssr, "nts[WHERE is_modified]") # all modified nucleotides
```

or even just this, without the brackets:

```
SELECT within(dssr, "nts WHERE is_modified") # all modified nucleotides
```

Finally, if brackets are used, specific items in the returned subset can be selected using the `..n` notation:

```
SELECT within(dssr, "nts[WHERE is_modified]..3") # just the third modified nucleotide
```

3.3 Summary

There are several ways in Jmol to select residues/atoms based on DSSR-derived structural features:

- Use simple keywords with SELECT:

```
SELECT hairpins
```

- Use the `within(dssr, ...)` syntax:

```
SELECT within(dssr, "hairpins..2")
```

- Use the `_M.dssr` array either directly or with the `.select()` function:

```
SELECT @{_M.dssr.pairs[9]}
SELECT @{_M.dssr.pairs.select("where name != 'WC'")}
```

- Combine `within(dssr, ...)` with the Jmol SQL for DSSR:

```
SELECT within(dssr, "nts[WHERE is_modified]")
```