Winnow

This document is adapted from the material in the *Winnow* online Help file. It was last modified April 2001. Its purpose is to provide a guide to *Winnow*'s (a) data processing functions, (b) utilities, (c) query language. The graphical display functions of *Winnow* are discussed in the online Help file.

1. INTRODUCTION

The *Winnow* utility is used to process the output files (.`SNK`) produced by the *Kalypso* (or *Snook*) program. Processing might typically involve extracting average values for state variables over a number of collision configurations (e.g. sputter coefficients, scattering cross-sections), or it might involve creating trajectory plots destined for graphical representation.

Dynamical variables information characterising your system was written to a .`SNK` file when you ran *Kalypso* (or *Snook*)[1]. The .SNK file captures the following dynamical variables information:

- position (`rx`,`ry`,`rz`)
- momentum (`px`,`py`,`pz`)
- elapsed time (`ti`)
- mass (`ms`)
- two integer indexes (`rw`,`rn`) labelling the atom involved (`rw` = row of .`TRG` file) and the run involved (`rn` = row of .`IMP` file, corresponding to a particular value of the collision impact parameter).

A typical .`SNK` file will contain information from more than one run (typically 10^2-10^4 runs). The structure of the file is represented in the following diagram (where each line represents one dynamical variables record; the meaning of the `rn` and `rw` parameters is also illustrated:

```
------   atom 0, run 1 (projectile: rw = 0)           ┐
------   atom 1, run 1 (1st target atom: rw = 1 )     │   rn = 1
------   atom 2, run 1 (2nd target atom: rw = 2)      │
......                                                │
------   atom N, run 1 (last target atom: rw = N)     ┘
------   atom 0, run 2                                ┐
------   atom 1, run 2                                │   rn = 2
......                                                │
------   atom N, run 2                                ┘
------   atom 0, run 3
------   atom 1, run 3

[.. and so on, down to the last atom of the last run ]
```

Information from this file can now be extracted by *Winnow* according to your requirements. There are two steps to this process:

- Firstly, you must filter out all the irrelevant records from your .`SNK` file. For example, you might only be interested in sputtered particles, or fast-moving particles, or you might only be interested in the projectile trajectory. This operation is conceptually similar to the filtering of records in a relational database.
- Then you can use *Winnow*'s math parser to convert your informational requirements into data-processing directives. For example, you may wish to query the mean energy of sputtered

---

[1] The .RUN input file of the *Kalypso* (or *Snook*) project (created by the *Spider* utility) specifies how often (and when) dynamical variables information will be written to disk. This will depend on your needs. For a trajectory plot, frequent writes are necessary as the system evolves. For sputter coefficients and ion scattering, you only need to capture the configuration at the termination of the simulation.

particles, or determine the average polar emission angle, or find the projectile range. If the quantity can be expressed algebraically, *Winnow* can probably get at it[2].

Before trying to carry out these operations, you need to familiarise yourself with the symbols in *Winnow*'s query language(`rx`, `px`, `ke`, `the`, `phi` etc.). *Winnow* also provides "Genius" functions that enable you make up queries and expressions, even if you do not know the language.

It should be mentioned that *Winnow* offers the option of dumping all the data in a `.SNK` file into a file of text format. Such text files can be loaded directly into spreadsheet programs (e.g. Excel), database programs (e.g. dBase) or statistical programs (like SPSS). If you find *Winnow*'s query language baffling, this would be an alternative way of processing your simulation data.

2. QUERY LANGUAGE

2.1. Introduction

This section describes the query language used by *Winnow*.

- Predefined indentifiers allow you to access the values of the variables (and certain functions of the variables) contained in the records of a SNK file. (They are also used to store constants.)
- *Winnow*'s query language allows you to customise output as required for your particular application. To do this, you need to specify what functions of dynamical variables (in terms of `rx,ry,rz,px...rn,rw`) you wish *Winnow* to produce.
- Conditional expressions required by the Filter operation are likewise specified by combining query expressions with relational operators (=, >, `>=,` etc) .

The query notation will be quite transparent to anyone with elementary programming or database experience . Most variables of interest can be accessed via predefined identifiers. For example: use either '`lz`' or '`rx*py - ry*px`' to represent the z-component of the angular momentum.

2.2. Syntax

Case is ignored. UPPER, lower or MiXeD case letters may be used for any input line symbol (`px` or `Px` or `PX`). *Blank* characters between valid symbols are ignored (`px*px` or `px * px`). The maximum length of the input line is 255 characters.

Comments may be nested between curly braces `{like this}`. Comments are useful as mnemonics in case you want to re-load the expression from a dialog box history list on a future occasion. Also, the comment will be printed in the output file if you use *Winnow*'s Averages or Collate options. Like blank characters, comments are ignored by the parser: `px*px {Here's a comment}`

---

[2] One interesting class of expression that *Winnow* cannot handle is simultaneous functions of the dynamical variables of two or more atoms: for example, the separation between two atoms: `√((x1-x2)²+(y1-y2)²+(z1-z2)²)`. You have to write custom code for such cases.

2.3. Predefined identifiers

The following *predefined identifiers* denote (`x,y,z`) components of vector dynamical variables (always expressed in SI units):

- `rx ry rz` = position
- `vx vy vz` = velocity
- `px py pz` = momentum
- `lx ly lz` = angular momentum

Other predefined identifiers (all in SI units) are:

- `ti` = time elapsed (t = 0 at the start of a simulated trajectory)
- `ms` = mass
- `ke` = kinetic energy
- `phi`, `alt,phid,phi2, phi4, phi8, altd` = azimuthal and polar angles (see notes below)
- `rw,rn`: see notes below

The symbol "`phi`" represents the particle's azimuthal direction of motion (ϕ), defined by:

- `phi` = arctan(py/px)

The range of `phi` is from 0 - 2π rad (unlike the "`atan`" function (section 2.5), which maps to a value in the range -π/2 to +π/2, this routine assigns the angle to the correct quadrant, even if `px` = 0). To express "`phi`" in degrees (0-360) use the identifier "`phid`".

Several azimuthal angle identifiers are defined. All are expressed in degreees. The identifier `phi4` maps the phi values to the first quadrant, and is defined as:

- `phi4 = arctan(abs(py/px))`

For example, the angles 100, 260 and 280 are replaced by 80; the angles 170, 190 and 350 are all replaced by 10. This option of using `phi4` is useful if you want to count a scattering/emission yield which has 4 fold symmetry around the *z* axis.

The `phi8` symbol is similar to `phi4`. However, it also maps azimuthal angles > 45° to the range 0- 45° using the following algorithm:

```
angle = arctan(abs(py/px));
if angle > 45.0, then angle = 90.0 - angle;
phi8 = angle;
```

This is useful if the emission process has 2 sets of 4-fold axes (e.g. sputtering of a (100) cubic lattice by a normally incident projectile).

Finally, `phi2` maps azimuthal angles > 180° into the range 0-180°. This is useful if the simulation problem has 2-fold symmetry. For example, projectile bombardment of Cu(100) at an oblique angle of incidence.

The symbol "`alt`" (**alt**itude) represents the particle's altitudinal direction of motion (φ), defined by:

- `alt` = $\arctan(pz/\sqrt{(px^2+py^2)})$

The range of "`alt`" is from $-\pi/2$ - $\pi/2$. If $(px^2+py^2) = 0$, then "`alt`" = $-\pi/2$ or $+\pi/2$ (motion parallel to z-axis). If `pz` is negative, then “`alt`” is also negative. To express “`alt`” in degrees use the identifier “`altd`”.

The identifiers `rw` (row index) and `rn` (run number) are recognised. In contrast to the dynamical variables, `rw` and `rn` only take on *integral* values:

- `rw` identifies the atom’s position in the original `.TRG` file used for the simulation (`rw` = 0 for the projectile (which is *not* in the .TRG file), `rw` = 1 for the anchor atom [its initial collision partner], rw=2 for the next atom in the file and so on)
- `rn` references a row in the `.IMP` file (`rn` = 1 for the run that used the 1st line of this file to define the impact parameter, and so on)

Constants with predefined identifiers are:

- `pi` = π
- `ep` = the proton charge (1.602×10^{-19} C)
- `au` = the atomic mass unit (1.66056×10^{-27} kg)

All quantities in the `.SNK` file are expressed in **SI units**. To express a kinetic energy in eV you would need to form the expression ‘`ke/ep`.’ To express a mass in amu, use the expression 'ms/au'.

2.4. Arithmetic Operators and Arithmetic Expressions

The predefined identifiers (see above) may be combined in expressions with decimal numbers in integer or floating point notations (i.e. `35`, `35.0`, `3.5e10`), bracket symbols `()`, and the arithmetic operators (`* / + -`). The syntax rules for forming expressions conform with those used in many programming languages:

- The elementary arithmetic operators are supported: + - * / ()
- Exponentiation is carried out using the caret (^) symbol e.g. `px*px*px = px^3`
- Blank spaces are ignored

Here are some examples of valid arithmetic expressions:

```
ke/ep                      {kinetic energy in eV }
ke /    ep                 {kinetic energy in eV: blanks ignored }
px*px+py*py+pz*pz)/2/ms     { kinetic energy,equiv. to ‘ke’ }
-lz + lx                    { equivalent to ‘lx-lz’ }
2.3*(rx*rx+2*ry*ry)/(1.3+6.5){arbitrary numerical expression}
```

```
rw + 1   { an integer expression; see sect. 2.8 }
rw + 1.0 { a real number expression (meaningless) }
```

2.5. Functions

Functions are supported, via a Pascal-style syntax: '`function(arg)`', where the argument in brackets (`arg`) *must resolve to a real number expression* [3]. (Integer expressions can be typecast, if necessary, by multiplying by 1.0.) There are probably more functions than you will ever need. All trigonometric functions use radians for angular units.

The following functions are supported:

- `sin` = sine of angle e.g. '`sin (2*pi*rx)`'
- `cos` = cosine
- `tan` = tangent
- `atan` = arctangent (inverse tangent)
- `exp` = exponential e.g. '`exp(-rx/300)`'
- `ln` = logarithm to base *e*
- `log` = logarithm to base 10
- `abs` = absolute value (modulus) e.g. '`abs(lx)`'
- `sqrt` = square root e.g. '`sqrt(rx*rx+ry*ry+rz*rz)`'

The following examples are physically meaningless, but illustrate how the real number syntax works for `rw` (or `rn`):

- `sin(3*rw)`: invalid - integer argument
- `sin(1.0*3*rw)`: valid - argument is 'typecast' to real expression
- `sqrt(rx + 1)` : valid - mixed expression ('`1`' is integer, `rx` is real) is typecast to real

2.6. Conditional expressions

Conditional expressions arise only in connection with the Filter operation. As with arithmetic expressions, blank spaces are ignored in conditional expressions and may be used to improve readability. All conditional expressions must be enclosed in square brackets `[]` (which may be nested to any depth, as needed).

Conditional expressions involve comparisons ("relations") between two arguments A and B, which may be any valid arithmetic expression involving any of the predefined identifiers listed in section 2.3 (`rx, ry, rz,`…).

Depending on the values of the compared items A and B, and the nature of the comparison, any given conditional expression will resolve to one of two values, either TRUE or FALSE.

A simple example would be the following expression: `[rz > 0.0]`. Here, `rz` is compared with zero: only when the expression is evaluated to a TRUE value (i.e. when `rz` is greater than

---

[3] This condition enforces programming discipline on the user: not a bad thing if you consider that the results may get published somewhere. See section 2.8 for more information on numeric types.

zero) does the Filter option initiate action (write data to disk). In this example, data are only stored for those particles which are above the target surface (defined by $z = 0.0$). To express this idea in database terminology: the expression enclosed by square brackets `[...]` constitutes a relation or condition which must be satisfied by any record which is written to the output file. What the preceding example has achieved is to filter out (isolate) data for ejected particles from the .`SNK` file and store it in a new .`SNK` file. This filtered data can then be processed further e.g. to generate an energy spectrum of ejected particles.

2.7. Logical and relational operators

The relational operators =, <, >, >=, <=, <> are recognised, with the following meanings:

- `[A = B]` TRUE if A is equal to B
- `[A > B]` TRUE if A is greater than B
- `[A < B]` TRUE if A is less than B
- `[A >= B]` TRUE if A is greater than or equal to B
- `[A <= B]` TRUE if A is less than or equal to B
- `[A <> B]` TRUE if A is not equal to B

The arithmetic expressions involved in the comparison must either be both real or both integer expressions (see section 2.8). Conditional expressions may be combined using the '&' (logical AND) and '`|`' (logical OR) operators. The following examples illustrate the syntax, but it would not be sensible to use these expressions for filtering because they always resolve to the same value:

```
[1=1] & [2=2]      {TRUE}
[1=1] & [px > px] {FALSE}
[1=1]|[2=3]        {TRUE}
[1=2]| [2=3]       {FALSE}
[1=2] | [px = px] {TRUE}
```

A realistic example of filtering is the following:

```
[ke/ep > 10.0] { particle KE > 10 eV }.
```

An example of a meaningless comparison with invalid syntax is:

```
[ke/ep > rw] { ERROR: rw is an integer! }.
```

Any conditional expression may be negated using the '!' (NOT) operator:

```
[1=1]     {TRUE}
![1 = 2]  {also TRUE}
```

Again, it is stressed that blank spaces are ignored by the parser. The '=' sign or some other relational operator must be present inside the brackets, as the conditional expression always involves a comparison.

Here are more examples of valid conditional expression syntax:

```
[pz*pz/(2*ms) > 10*ep] & [ [rz > 0.0] | [pz > 0.0] ]
```

The above expression is TRUE if the kinetic energy associated with motion normal to surface is > 10 eV and the particle is either outside the surface (z > 0) or is travelling away from the surface (in a positive z direction). Note the way brackets are used to nest sub-expressions.

```
[atan(pz/sqrt(py*py+px*px)) > 50*pi/180] & [rz > 0.0]
```

The above expression is TRUE if the particle has left the surface and is travelling at an of-surface angle of > 50°. This example shows a typical use of the arctangent function. A better way to express this is:

```
[altd > 50.0] & [rz > 0.0]
```

The Appendix contains more examples of Filter expressions.

2.8. Numeric types

Two numeric types are recognised: ‘integer’ and ‘real’ (floating point) types. Enforcing type distinctions in syntax helps to avoid certain kinds of semantic and physical errors which might otherwise go unnoticed. The conventions outlined below are similar to those found in most programming languages.

- The predefined identifiers `rw`, `rn` and all numbers written in integer format (0, 1, -1..) are treated as integer-type numbers.
- All other numbers, variables and constants are of real-type.
- Mixed integer-real expressions (`2*pz`, `1+1.0` etc.) are automatically cast to real-type.
- Arguments passed to functions must be in real format (this can always be enforced by typecasting: multiplying by 1.0):
  ```
  exp(10) {invalid}
  exp(1.0*10)valid}
  ```
- Both sides of a conditional expression must be of identical type:
  ```
  [3 > 2]       {TRUE}
  [3.0 > 2]   {syntax error: can’t compare real with integer}
  [3.0 > 2.0]   {TRUE}
  [3*1.0 > 2.0] {TRUE - left hand side is typecast to real }
  ```
- Otherwise the type identity of expressions is the same as that of the constituents, except for integer division (which always results in a real number).
  ```
  rw*2 + 1   {integer}
  2*pz + 3   {real}
  rw*2 + 1.0 {real}
  rw/2 + 1   {real - integer division}
  ```

2.9. Parser errors

Most common parser errors are syntax errors and arithmetic errors (divide by zero etc.). The parser error message will indicate the point (^) in the input expression at which it detected an error. E.g., for the Filter option input:

Example 1:

```
[px > 0]
       ^
```

The pointer indicates character #8 reading from the left. The parser expected a floating point number (such as 0.0) rather than an integer, but only detected an error when it encountered the ‘]’ character. Solution: change ‘0’ to ‘0.0’.

Example 2:

```
2.0*px > 3e-20]
  ^
```

The parser evaluates both the missing ‘[’ and the real number ‘2.0’ as atomic tokens. It expects to find the pattern BRACKET REAL, rather than REAL, and indicates an error at the end of the illegal REAL token. Solution: add the missing ‘[’ before ‘2.0’.

To get a feel for the error-response of the parser, you could try feeding it some deliberate errors (e.g. ‘`px*px + px**px`’).

Common syntax errors include:

- using `x`,`y`,`z` instead of `rx`,`ry`,`rz`;
- opening parentheses ‘(‘ ‘[’ without later closing them;
- using integers in places where real numbers are expected.

If the cause of the error is not obvious, try simplifying the expression until it parses correctly. Remember that no program can protect you against logical errors. **A typical non-syntactical error in this category is forgetting to couch the expression in SI units.**

3. FILTERING

The key to getting at the information contained in `.SNK` files is the Filter option, which discards unwanted data from your raw `.SNK` file. Put simply, you select the data which you want to examine further, and store it in a new file.

For example, suppose you are interested in the fraction of sputtered atoms with energy above 1 eV; a sputtered atom is defined as one that is moving away from the surface `[pz > 0.0]` and is above the surface beyond interaction range (e.g. 6 Å) `[rz > 6.0e-10]`. These conditions, plus the 1 eV energy condition `[ke/ep > 1.0]`, can be expressed in *Winnow*’s query language as the following conditional expression:

```
[rz > 4.0e-10] & [pz > 0.0] & [ke/ep > 1.0]
```

This expression is translated by *Winnow*'s parser into computational directives. The output is sent to a new `.SNK` file (the default name contains the tilde ‘~’ character). If you feed the above expression to *Winnow*'s Filter option, it will filter out all data that doesn’t satisfy the condition. You can then proceed to Average or Collate the data written in the new file, or Filter it again according to some other criteria. Filtering a large file is quite a slow operation, even on a fast Pentium PC.

Here are some further examples of the use of conditional expressions (note the nesting of brackets in the composite expressions C and D):

(A). Filter out (discard) projectile data:
Use filter condition: ‘`[rw > 0]`’.

(B). Filter out atoms (discard) with less than 10 eV kinetic energy:
Use filter condition: ‘`[ke/ep >= 10.0]`’.

C. Keep data for ejected atoms ($z > 0$) with ke > 0.5 eV only.
Use filter condition: ‘`[pz > 0.0] & [rz > 0.0] & [ke/ep > 0.5]`’.

D. Combine Filter with Collate to determine origins of ejected atoms ($z > 0$) :
(1) Filter with condition: ‘`[pz > 0.0] & [rz > 0.0]`’.
(2) Subsequently, a ‘Collate’ operation[4] on the atom-by-atom basis gives the total number of ‘hits’ satisfying this condition. In other words, we obtain a breakdown of the ejected atom population by ‘row number’ (the parameter `rw`, which identifies the atom with reference to its position in the .`TRG` file: see section 1). The collation function used in this case is irrelevant, as we are simply counting atoms.

4. AVERAGING

The Averages operation calculates statistical information based on the *entire* .`SNK` input file. The user defines some function (let’s call it `q`) of the system dynamical variables, expressed in terms of the predefined identifiers listed in section 2.3. The Averages operation writes a line of output based on this function, which consists of the mean value (`<q>`), root mean square (rms) value (`√<q²>`), and the standard deviation (`σq`) of `q` respectively:

`<q>        √<q²>        σq.`

Up to 7 such functions can be specified in one averaging operation. The output is written to a file with .`TXT` extension, e.g. `DYNVARS.SNK` → `DYNVARS.TXT`.

For example, you might wish to calculate the mean x-velocity component of in a `SNK` file filled with records of sputtered atoms: simply enter ‘`vx`’ on the input line. Then for this case `<q> = <vx>`, while `√<q²>` corresponds to the rms x velocity component.

If a .`TXT` file of the same name already exists, the output is *appended* to the existing file. This is in contrast to the over-write behaviour of the Collate operation. Normally the averaging would be carried out after the .`SNK` file has been filtered (see section 3) to remove unwanted data.

5. COLLATING

The Collate operation calculates statistical averages on an atom-by-atom, or run-by-run basis, according to the user’s specifications. To use it, you need to understand the information content of the .`SNK` file.

---

[4] See section 5 for information about the Collate option in `Winnow`.

- In the atom-by-atom case, the output data is collated according to the ‘row number’ (rw) index (see section 1). This represents an *average over particles*.
- In the run-by-run case, the ‘run number’ index (rn) from the `.IMP` files is used as the collation key. This represents an *ensemble (configuration) average*.

The collated averages <..> are output to a `.DAT` file with a column lay-out which has a layout similar to that of an ‘Average’file (see previous section):

```
   <q>      √<q²>      σq      key      hits
```

where:

- `q` represents the expression input by the user
- `σq` = standard. deviation of q,
- key = rw or rn (row number or run number, as selected by user)
- hits = number of data used for averages

Collated output overwrites the contents of any appropriately named `.DAT` file that may exist. E.g. collating `DYNVARS.SNK` sends output by default to `DYNVARS.DAT`; if the latter already exists, it is overwritten.

The atom-by-atom average could be used to determine (for example) which atoms were sputtered most frequently in a real-life experiment (which is averaged over all impact parameters).

The run-by-run average may be useful in establishing how individual configurations contribute to the overall behaviour. If the `.SNK` file only contains data from a single run then the run-by-run average is equivalent to the Averages option. (In fact, the Averages option is equivalent to collating by both `rw` and `rn` simultaneously.)

You specify what function of dynamical variables gets written to the collated file using *Winnow*'s query language (section 2.3). For example, ‘`px*px`’ represents the square of the x-momentum component.

6. FORMAT COLUMNS OPERATION

This command performs a simple conversion of a binary `.SNK` file to a text (human-readable) `.DAT` format, laid out in up to 10 columns listing variables or functions specified by the user. In the simplest case, you can use this option to dump your `.SNK` data to a `.TXT` file for transfer to a spreadsheet program (for example).

You enter the functions you want to dump using *Winnow*'s query language (see section 2).

A formatted file can be read by spreadsheet and other data analysis programs, but cannot be processed further by *Winnow*. The `.SNK` to `.DAT` conversion expands the file size by a factor of three (this is the reason that the output of *Kalypso* (or *Snook*) is written in binary format).

For example, if you wish to plot particle kinetic energies versus their distance from the (0,0,0) point, use the following inputs:

- Column 1: `sqrt(rx*rx + ry*ry + rz*rz)` `{distance}`
- Column 2: `ke/ep` `{kinetic energy in eV}`

This will yield a .`DAT` output file suitable for use by any graphing program. Again, it is stressed that up to 10 columns of parameters can be written to your .`DAT` file.

7. CONVERT

The Convert option performs a simple conversion ('dump') of the records in a binary .`SNK` file to a text (human-readable) .`DAT` format; this file has the values of the dynamical variables in each record laid out in columns as follows:

```
rx ry rz px py pz ti ms rw rn
```

The meanings of the symbols are defined in section 2.3. Unlike other processing commands in Winnow, the output produced by the Convert feature does *not* use base SI units (kg m s) for its dynamical variables fields.

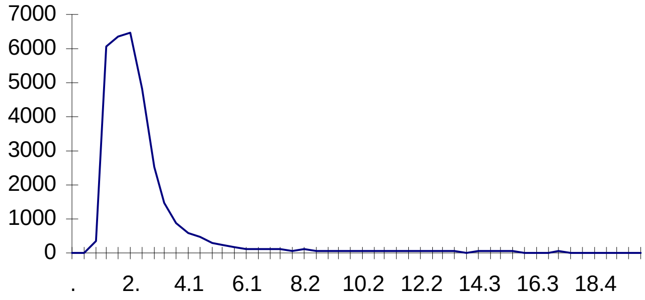
- `rx ry rz` are expressed in Å
- `px py pz` are expressed in (kg m s^{-1})×10^{22}
- ti is expressed in fs
- ms is expressed in kg
- `rw` and `rn` are integers.

A converted file can be read by commercial spreadsheet programs, but cannot be processed further by *Winnow*.

8. SPECTRUM

This option uses information in a .`SNK` file to create the ‘spectrum’ (frequency histogram) of some function of dynamical variables. The spectrum is output in the form of a .`DAT` file eg.: `TEST.SNK` --> `TEST.DAT`, and can optionally be displayed on screen. This is an important function, because it enables you to look at the distribution of the values of some arbitrary function of dynamical variables.

The figure below is based on data obtained from a trial run of 625 trajectory simulations in the 600 eV Xe → Mo(100) system; it shows the distribution of atoms as a function of z-position (in Å), above the sample surface (where $z = 0$) at the conclusion of the simulations (when all particle energies in the system were < 1.5 eV). Data for $z < 1.0$ Å are not shown.



In the Spectrum dialog box, the user needs to specify the function (F) whose distribution of values is of interest; this is done using *Winnow*'s query language (see section 2).

- For example, to examine the altitudinal angular distribution, specify F as:
  `altd {altitudinal angle, in degrees}`
- For a kinetic energy spectrum, specify F simply as:
  `ke {particle kinetic energy}`

Other inputs required from the user are the minimum and maximum limits of the spectrum, and the number of channels to be utilised by the spectrum. The query function is calculated for each record in the `.SNK` file. If the function falls within the spectrum range the appropriate channel count is incremented.

A given channel registers counts for data within a bandwidth ±0.5Δ, where the interval Δ is given by:

Δ = (Max limit - Min Limit)/(No. channels - 1)

The spectrum will thus in practice register counts from (Max Limit + Δ) to (Min Limit - Δ) i.e. a slightly greater range than that specified by the user, as the figure below shows:

```
<- Δ ->
      |     |     |     |     channel positions
   |     |     |     |     |  counting intervals
      <------------------>    range specified by user
<------------------------>  effective range
```

If you need to create spectra that are functions of angular variables, you need to take account of the fact that the result of an expression like:

`atan(py/px)   {azimuthal angle},`

is ambiguous (since there are two possibilities, separated by π). You might have to specify additional relations involving `py` and/or `px`.

In the physical system the momenta (`px`,`py`) and (-`px`,-`py`) represent diametrically opposed directions of motion, although their ratios give the same arctan result. In practice, this ambiguity only exists for the azimuthal angle (not the altitudinal/polar angle). To work around it, you first must filter the raw data into a pair of `.SNK` files containing respectively positive and negative px values. Of course, symmetry may obviate the need for this on some single crystal targets.

For expressions involving the azimuthal and altitudinal angles of motion, you can make use of the predefined angular variables "`phi`", "`alt`", "`phi4`", "`phi8`", "`phid`" and "`altd`" respectively, which always return angular values in the correct quadrant (see definition of the query language, section2.3, for more on these variables).

There is another minor difficulty with angular variables accessed via the a(rc)tan function: the effective spectral range (see above) is at best -π/2 to +π/2, regardless of the size of the spectrum counting interval (Δ). At the limits ±π/2 you will get anomalous count rates (usually 50% of what you expect). These limits are the same for the "`alt`" variables, but 0 and 2π for the "`phi`" variables. You may have to combine the counts at the limits by manual editing of the output file.

9. SCATTERING RELATIONS

This simple gadget calculates useful scattering parameters, based on standard binary collision formulae.

The user enters the projectile and target atomic masses (M_1, M_2) and a laboratory scattering angle for the projectile (θ_1 Lab).

The symbol E_0 represents the Lab projectile incident energy, while E_1 and E_2 represent the energies of the projectile and target respectively, after the collision.
On pressing the <Enter> key or Evaluate button, the following quantities are calculated:

1. θ2 Lab: This is the scattering (recoil) angle of the target.
2. θ COM (1), θ COM (2): These are the scattering angles in the centre-of-mass coordinate system (identical for both particles). If $M_1 > M_2$, the Lab projectile scattering angle must lie between 0 and a maximum value ($\theta_{max} = \arcsin(M_1/M_2)$). In this case there are two possibilities (1) and (2) for the COM angle, which still ranges from 0-180°. (See a textbook on classical dynamics for an explanation.) For $M_1 \leq M_2$, there is only one COM angle per Lab angle.
3. E_1/E_0, E_2/E_0: These energy ratios reflect the energy transferred during the collision. If $M_1 > M_2$, there are two possibilities for energy partition. These energy ratios are completely independent of the nature of the interaction potential or primary projectile energy. (The target is assumed to be stationary initially.)

10. FILE FORMATS

10.1. SNK file

The `.SNK` file is the native file format for the output written by *Kalypso* (or *Snook*). It consists of 'TSnookRec' binary records (40 bytes = 8x4 + 2x4 per record) whose Pascal definitions are:

```
TSnookRec = record
  rx,ry,rz,px,py,pz,ti,ms: single;{4-byte}
  rw,rn: integer; {4-byte}
end;
```

The record declaration in C would be:

```
typedef struct
  {
    float rx,ry,rz,px,py,pz,ti,ms;
    long rw,rn;
  } TSnookRec;
```

You can quickly count the number of records in a .`SNK` file by dividing the file size in bytes by 40.

The atomic positions (`rx,ry,rz`), momenta (`px,py,pz`), mass (`ms`) and the elapsed time (`ti`) are all expressed in SI units. Note that the dynamical variables are calculated to double precision by *Kalypso* (or *Snook*), and only rounded off to single precision for storage purposes. The `rw` (row number) variable = 0 for the projectile, otherwise it is > 0, the value corresponding to the row the atom occupied in the .`TRG` file used by *Kalypso* (or *Snook*). The `rn` (run number) variable likewise corresponds to the row of the .`IMP` file (= 1,2...).

10.2. DAT file

The .`DAT` file is a text file consisting of two or more numeric columns separated by arbitrary amounts of 'whitespace'.

10.3. TXT file

The .`TXT` file is any text file which does not consist of purely numeric columns.

11. DISPLACEMENTS OPERATION

The Displacements function creates an output SNK file from 2 input SNK files. The positions and momenta written to the output file are the differences of the relevant components in the two input files.

Given two input files consisting of records such as:

```
 x1   y1   z1   px1   py1   pz1 ...   file #1,

  x2   y2   z2   px2   py2   pz2 ...   file #2,
```

the output file will have a corresponding set of records:

```
  x2-x1   y2-y1   z2-z1   px2-px1   py2-py1   pz2-pz1 ...
```

The remaining records in the output SNK file are reproduced from the second of the input files (file #2). Note: the two input SNK files used in the operation should be of the same size.

The purpose of the operation is to allow you to calculate atomic displacements (in real or momentum space) which have occurred between an initial and a final system state. The output file so created can be processed just like an ordinary SNK file. The most likely application is in calculating how far atoms have moved from their lattice sites: in this case, the first input file would conatain the initial conditions information (i.e. inivars.snk), and the appropriate expression to use with the output file would be: `sqrt(x*x+y*y+z*z)`. This kind of operation can also be achieved with the Merge option (see below).

12. CROSS-REFERENCE

The Cross-Reference operation carries out calculations based on coordinate data stored in the Target file used by your simulation. The user has to supply a file containing pairs of integers, which represent the 'row number' (`rw`) parameters extracted from a SNK file (probably by using the Format Columns command with parameters '`rw`' and '`rw`').

The Cross-Reference operation is used to answer the question: how far away from the origin (and from each other) were these atoms in the Target file? This is mainly useful for locating the origins of sputtered atoms.

The format of the file containing the integer pair is as follows:

```
21   3
2   34
5   6
...  ...
```

(i.e., pairs of free-format, whitespace-delimited integers). The integers in each pair can be identical or different. If they are the same, the output in Columns 3 and 4 (see below) will be zero. A list of pairs of different atoms would represent (for example) the constituents of a 2-atom cluster. However, you will have to generate this file yourself. (See the file `clusters.pas` which ships with this package for an example of how to extract the indexes of atoms sputtered as clusters.)

This operation reports the following information to the ouput file:

Column 1 (integer 1 in file)
Column 2 (integer 2 in file)
Column 3: r0
Column 4: r12
Column 5: rho0
Column 6: rho12

The meanings of the output items 3-6 are:

- r0 = distance between centre of mass of atoms 1 and 2 in target file, and origin of target file
- r12 = distance between atoms 1 and 2 in target file
- rho0 = lateral separation (in x,y plane) between centre of mass of atoms 1 and 2 in target file, and origin of target file respectively
- rho12 = lateral separation between atoms 1 and 2 in target file

Symbolically, these quantities are expressed as:

- `r12 = sqrt(sqr(x^[i]-x^[j) + sqr(y^[i]-y^[j]) + sqr(z^[i]-z^[j]) )`
- `rho12 = sqrt(sqr(x^[i]-x^[j]) + sqr(y^[i]-y^[j]) )`

- `r0 = 0.5*sqrt(sqr(x^[i]+x^[j]) +sqr(y^[i]+y^[j]) +sqr(z^[i]+z^[j]) )`
- `rho0 = 0.5*sqrt(sqr(x^[i]+x^[j]) + sqr(y^[i]+y^[j]) )`

13. FIND CLUSTERS

This is a specialised option used for survey purposes in sputtering simulations: for serious work, you will want to write your own routines. The operation examines a SNK file for the presence of stable dimer and trimer clusters. Stability is defined on the basis of the internal energy of the cluster, assuming a Morse potential interaction. The application of this task is, of course, in studies of sputtering. The algorithm assumes that the atoms listed in the SNK file have identical massese (which are used for computing the relative kinetic energy). The algorithm is discussed in: B.J. Garrison, N. Winograd and D.E. Harrison Jr., J. Chem. Phys. 69 (1978) 1440-1444. The source code used to implement this routine is provided in the `source\`*Winnow* directory (`FindClusters.pas`).

- Input file
  - SNK file to be analysed.

- Output files
  - A text file containing information about detected clusters (see below).
  - A SNK file which contains dynamical variables information about the dimer clusters. In brief, the (rx,ry,rz) values written to this file refer to the centre of mass of the cluster, while the (px,py,pz) and ms fields are written in such a way that when you compute the kinetic energy of the cluster using *Winnow,* you will get the correct answer in the Lab system. Don't use (px,py,pz) by themselves, because they do *not* refer to the momentum of the cluster.

- Input data
  - The user must specify the Morse potential parameters, including the potential cut-off (range). There is an option to ignore the potential, and just test for atoms which are within the stated range. The 'Exclude Multimers' option only registers the first cluster interaction for a given atom. This is best illustrated by an example: suppose atoms 1-2-3 are in fact part of a trimer. The cluster counting algorithm goes as follows:

    ```
    for i = 1 to N-1 do
         for j = i+1 to N do
    begin
    if DetectCluster(i,j) then
    begin
      nclusters = nclusters + 1;
      if ExcludeMultimers then break; // start next i
    end
    end;
    ```

  - If ExcludeMultimers is false, then dimers 1+2, 1+3 and 2+3 will be detected. If ExcludeMultimers is true, only 1+2 and 2+3 will be detected. Clearly, neither answer is correct (there are no dimers, only 1 trimer). You will thus need to examine the output by hand to deduce exactly how to classify difficult cases like this. If the system has no multimers, the results will be independent of the Exclude Multimers option.

14. MERGE AND REMERGE

14.1 Merge

This operation takes the atomic coordinates listed in the specified TRG file, and writes them to the output SNK file, overwriting the original values. For example, every record for atom #N in the input file will be substituted with the TRG file coordinates for the same atom, but will otherwise remain unchanged.

- Input files
    - SNK file
    - Corresponding TRG file

- Output
    - SNK file

The default operation writes the TRG file coordinates to the (rx, ry, rz) records in the SNK file. But the user can also choose to overwrite the (px, py, pz) records.

This operation is useful if you need to collate the post-simulation behaviour of the system on the basis of the original atomic locations in the TRG file.

14.2. Remerge

This operation restores information that was substituted by the Merge command. Suppose you used the Merge command to create a file `A.snk` from `dynvars.snk`. You may wish to do something to `A.snk` - for example, filter out edge atoms from the lattice. This will produce a smaller file, `B.snk`. To restore the original data to the records that remain in B.snk after filtering you can Remerge `B.snk` with `dynvars.snk`. The remerged output file, `C.snk`, will have similar records to `dynvars.snk`, except that it will lack those that were removed by the filter operation.

In effect, the Merge/Remerge combination allows you to filter a SNK file on the basis of attributes found in the TRG file.

APPENDIX: QUERY EXPRESSION EXAMPLES

In this appendix, some further examples of query expressions are given. Recall that these arise in 2 contexts:

- Function specifications
- Conditional expressions (Filter operations)

The conditional expressions combine function specifications with relational and logical operator symbols. Conditional expressions are only used by *Winnow*`'s` Filter option. In the examples below, comments are enclosed in curly braces `{thus}`: comments can be used freely in *Winnow*, since they are ignored by the parser.

The examples below show how to build up conditional expressions by combining several simple relations.

Function (expression) specifications

- `ke/ep {particle kinetic energy in eV}`
- `atan(pz/sqrt(sqr(px)+sqr(py)) {altitudinal flight angle, ψ}`
- `alt {altitudinal flight angle, ψ - same as preceding}`
- `rz*1e10 {z-position in Å: surface is at rz = 0.0 by default}`
- `ti*1e15 {elapsed time in fs}`
- `rw {index of the particle = 0,1,2..; 0 is the projectile}`

Conditional expressions

The following examples show what is possible in terms of conditional expression combination and nesting. However, in practice it may be better to use a sequence of simpler expressions, especially if the query language is unfamiliar.

`[atan(pz/sqrt(sqr(px)+sqr(py))> 50*2*pi/360]`
`{all particles flying at altitudinal angles greater than 50°}`

An equivalent and better way to express the foregoing filter is:

`[alt > 50*pi/180]` (or better still: `[altd > 50.0]`).

`[rz > 0.0] & [altd > 50.0]`
`{particles` *`emitted`* `at altitudinal angles greater than 50°}`

(In this example, emitted atoms have been defined as those with rz > 0.0.)

`[ke/ep > 10.0] & [rz > 0.0] & [altd > 50.0]`
`{particles with energy > 10 eV which are` *`emitted`* `at altitudinal angles greater than 50°}`

`[ti*1e15 < 300.0] & [ke/ep > 10.0] & [rz > 0.0] & [altd > 50.0]`

```
{particles with energy > 10 eV which were emitted at altitudinal
angles greater than 50° before 300 fs had elapsed}
```

```
[ti*1e15 < 300.0] & [[ke/ep >= 10.0] & [ke/ep =< 50.0]] & [rz >
0.0] & [altd > 50.0]
{particles with energy in the range 10-50 eV which were emitted
at altitudinal angles greater than 50° before 300 fs had elapsed}
```

The effect of the last example could also be achieved by the following sequence of 4 shorter filtering operations (using the output file from one filter operation as the input for the next):

```
1. [ti*1e15 < 300.0]
2. [[ke/ep >= 10.0] & [ke/ep =< 50.0]]
3. [rz > 0.0]
4. [altd > 50.0]
```

Exercise 1.
What would be the effect of substituting:

```
[ke/ep <= 10.0] & [ke/ep >= 50.0]
```

instead of the second item above?

Answer.
The answer is that you would get no output from your filtering operation, since this expression looks for particles with KE satisfying two mutually exclusive conditions (KE ≤ 10 and KE ≥ 50).

Exercise 2.
What would be the effect of substituting:

```
[ke/ep <= 10.0] | [ke/ep >= 50.0]
```

instead of the second item above?

Answer.
The answer is that your output would catch all particles except those in the energy range 10-50 eV.