MATHSCOUT User's Guide

Michael P. Barnett,[*]
Meadow Lakes, Hightstown, NJ 08520,
Joseph F. Capitani,[†]
Joined Departments of Chemistry and Biochemistry,
Manhattan College/The College of Mount Saint Vincent,
Riverdale, NY 10471,

May 31, 2007

The software described in this User's Guide supports the extraction of data from the output of many programs besides the Gaussian package. The methods that we used to develop, to test and to document this software have potential uses in other applications of MATHEMATICA. The accompanying file `msctDevelopment.tar.gz` contains the development software and full information on how to use it.

In this User's Guide,

- §1 describes the input commands,
- §2 describes the scanning commands,
- §3 describes some further commands that support auxiliary operations,
- §4 describes the major commands for mechanized documentation.
- §5 describes the extraction of all the data from the log files of the water and zinc hydrate calculations — these provide most of the examples in the published paper, in the Tutorial and in this Guide,
- §6 lists the files that comprise the distribution package.

1 The input commands

`loadFileAsListOfStrings[`*fileName*`]` saves each line of the specified file as a character string and returns the list of strings that it has saved.

`inputGaussianFile[`*fileName*`]` cosmeticizes this list by

1. ensuring that each `=` symbol is flanked by spaces,
2. removing `\r` return codes,

---

[*] michaelb@princeton.edu

[†] joseph.capitani@manhattan.edu

[‡] MATHEMATICA is a registered trademark of Wolfram Research Inc.

[§] GAUSSIAN is a registered trademark of Gaussian Inc.

3. removing redundant, leading and trailing spaces,
4. removing null records.

This list is called a log list in the description of MATHSCOUT and the individual elements are called lines.

2 The scanning commands

2.1 The extractionCycle

`extractDataFrom[`*logList*`,using[`*scanList*`]]` extracts data from the specified log list using the specified scan list. Successive commands in the scan list are interpreted by successive cycles in the execution of `extractDataFrom`. The following variables and lists are updated by the commands in the scan list.

1. `scanStatementNumber` is initialized to 0. It is increased by 1 in each cycle.
2. `currentLineNumber` points to the line in the log list that is the starting point for the action of the next command to be executed. Commands that extract data increase this to point to the line following the last to be extracted. Other commands skip to specified lines.
3. `currentExtract` consists of a single line or a list of lines extracted from the log list.
4. `associationList` returns the results of the extraction process. It is initialized to {}.
5. `nameForNextItem` is constructed by `makeNameFromCurrentLine`. It is associated with the result of the command that operates on an extract.
6. `currentTarget` is the internal name for the log list being scanned.

We refer to the value `currentExtract` interchangeably as current extract and *currentExtract*, and to the values of the other variables in roman and italic type, too.

2.2 Extraction commands

`extract[`*n* `lines]` assigns `currentExtract` to the list of *n* successive lines starting at the current line number. It increases the value of `currentLineNumber` by *n*.

`extractCurrentLine` assigns `currentExtract` to the current line. It increases `currentLineNumber` by 1.

`extractNextLine` assigns `currentExtract` to the line in `currentTarget` that is specified by `currentLineNumber` + 1. It increases `currentLineNumber` by 2.

`extractSuccessiveLinesBeforeNext[`*key*`]` assigns `currentExtract` to the list of lines that

1. starts at the current line number and
2. ends with the line preceding the next occurrence of *key*.

The cursor `currentLineNumber` is reset to point to the line containing *key*.

`extractSuccessiveLinesBeginning[`*key*`]` assigns `currentExtract` to the list consisting of lines *m* to *n* where

1. m is the current line number,
2. each of the m'th through n'th lines begins with *key*, and
3. the n'th line ends the target file or line $n+1$ does not contain *key*.

The cursor `currentLineNumber` is increased to $n+1$. If the m'th line does not begin with *key* then `currentExtract` is assigned to a null list.

`extractSuccessiveLinesIncludingNext[`*key*`]` assigns `currentExtract` to the list of lines that

1. starts at the current line number and
2. ends with the line containing the next occurrence of *key*.

The cursor `currentLineNumber` is reset to point to the line that follows.

`extractSuccessiveLinesIncluding[second[`*key*`]]` assigns `currentExtract` to the list of lines that

1. starts at the current line number and
2. ends with the line containing the second next occurrence of *key*.

The cursor `currentLineNumber` is reset to point to the line that follows.

`extractSuccessiveLinesIncludingNextNot[`*key*`]` assigns `currentExtract` to the list of lines that

1. starts at the current line number and
2. ends with the next line that does not contain *key*.

The cursor `currentLineNumber` is reset to point to the line that follows.

`extractTheResidue` assigns `currentExtract` to the list of lines that

1. starts at the current line number and
2. ends with the final line of `currentTarget`.

The cursor `currentLineNumber` is set to the length of `currentTarget` plus 1.

2.3 Skip commands

`skip[`n` lines]` adds n to `currentLineNumber`. n can be positive, negative or zero.

`skipToEnd` sets `currentLineNumber` to point to the final line in `currentTarget`.

`skipNextLine` increases `currentLineNumber` by 2.

`skipTo[`*key*`]` sets `currentLineNumber` to the line number of the first line in `currentTarget` that contains *key*.

`skipTo[`*key*`, offset[`n`]]` sets `currentLineNumber` to n plus the line number of the first line in `currentTarget` that contains *key*. The offset can be positive, zero or negative, consistent with the length of the target and the position of *key*.

`skipToFinal[`*key*`]` sets `currentLineNumber` to the line number of the line in `currentTarget` that contains the final occurrence of *key*. The `offset` argument can be included with the effect corresponding to its use in `skipTo[...]`. Here, n cannot be positive.

`skipToNext[`*key*`]` sets `currentLineNumber` to the line number of the next line in `currentTarget` that contains *key*, *i.e.* the first line containing *key* after the line specified by the current value of `currentLineNumber`. The `offset` argument can be included with the effect corresponding to its use in `skipTo[...]`.

`skipToNextLine` increases `currentLineNumber` by 1.

2.4 Naming selected lines

`callIt`[*name*] appends *name*==*valueOfCurrentExtract* to the association list.

`callThem`[*name*] is an alias for `callIt`[*name*] for mnemonic use when the current extract is a list.

`callCurrentLine`[*name*] coalesces `extractCurrentLine` and `callIt`[*name*].

`callSuccessive`[*lineCount* `lines`][*name*] coalesces the pair of commands `extractSuccessive`[*lineCount* `lines`] and `callThem`[*name*].

`callTheResidue`[*name*] coalesces `extractTheResidue` and `callIt`[*name*].

`callTheResult`[*name*] changes the top element v in the association list to the expression $name == v$. If the extract consists of more than one line, v is the list of these.

`callTheResults`[*name*] is an alias for `callTheResult`[*name*] for mnemonic use when the top item in the association list is a list.

`callSuccessiveLinesBeforeNext`[*key*][*name*] coalesces the pair of commands `extractSuccessiveLinesBeforeNext`[*key*] and `callThem`[*name*].

`callSuccessiveLinesBeforeNextNot`[*key*][*name*] coalesces of commands `extractSuccessiveLinesBeforeNextNot`[*key*] and `callThem`[*name*].

`callSuccessiveLinesIncluding`[*key*][*name*] coalesces the pair of commands `extractSuccessiveLinesIncluding`[*key*] and `callThem`[*name*].

`callSuccessiveLinesIncludingNext`[*key*][*name*] coalesces of commands `extractSuccessiveLinesIncludingNext`[*key*] and `callThem`[*name*].

`callSuccessiveLinesIncludingNextNot`[*key*][*name*] coalesces of commands `extractSuccessiveLinesIncludingNextNot`[*key*] and `callThem`[*name*].

`makeNameFromCurrentLine` converts the current line into a name that is associated with the result of operating on the next `currentExtract` with any of the commands in the next subsection. The name is constructed by the `makeName` function.

`makeName` constructs a name from a character string as follows:

1. the characters `.` `,` `:` `;` `(` `)` `[` `]` `<` `>` `?` `!` are deleted,
2. any string of 5 asterisks is deleted,
3. the symbols `+`, `-`, `/`, `**`, `=` are replaced by `plus`, `Dash`, `Slash`, `toPower` and `eq`, respectively,
4. each letter immediately after a space is capitalized,
5. spaces are then elided.

2.5 Operating on extracted lines

Each of the commands in the present subsection produces a list of results that is treated as follows.

1. If `makeNameFromCurrentLine` occurred immediately before the extraction, it is associated with the result by the `Equal` symbol `==`. If more than one item is extracted from a line in the extract, then the items from that line are held together as a list. If the extract consists of more than one line, then the results from the individual lines are held together as a list.
2. If `makeNameFromCurrentLine` did not occur immediately before the extraction, but `callTheResult[`*name*`]` occurs immediately after the command has acted on `currentExtract`, then the specified *name* is associated with the results in the corresponding manner.
3. If neither naming action occurred, the list status of the set of items from the operation is removed at the end of the scan, because the association list is flattened at that time.
4. In the syntactically acceptable case that both naming actions occur, the result is *nameFromCall* `==` `{`*nameFromEarlierLine* `==` `{`*results*`}}`, but no application has yet arisen.

Proceeding to the individual operations:

`useEmbeddedEqual["`w_1 `=` v_1 w_2 `=` $v_2 \dots w_n$ `=` v_n`"]` appends $\{\varphi_1$`==`$\bar{v}_1$, φ_2`==`$\bar{v}_2$, $\dots$, φ_{n-1}`==`$\bar{v}_{n-1}$, φ_n`==`$\hat{v}_n\}$ to the association list, where

1. each w_i consists of one or more strings separated by spaces,
2. the corresponding φ_i are formed from these by `makeName`, as described at the end of §2.4,
3. each $v_i, i = 1, \dots n-1$ contains no embedded spaces,
4. each $\bar{v}_i, i = 1, \dots n-1$ is
   - (a) the value of v_i if this is a number,
   - (b) the symbol or longer expression obtained by removing the quote marks from v_i, if (a) this conforms to MATHEMATICA syntax and (b) it is not evaluated automatically,
   - (c) the value of the expression, if removing the quote marks leads to automatic evaluation,
   - (d) the string v_i in all other cases.
5. the object $\hat{v}_n$ is
   - (a) $\bar{v}_n$ if v_n is free of embedded spaces and
   - (b) $\{\bar{v}_{n,1} \dots, \bar{v}_{n,m}\}$ when v_n consists of substrings $v_{n,1}, \dots, v_{n,m}$ separated by spaces.

`keepWords[{`$n_1, \dots, n_k$`}]` returns $\{w_{n_1}, \dots, w_{n_k}\}$ if `currentExtract` is a single string that splits into the list of strings $\{w_1 \dots, w_\ell\}$. If `currentExtract` is a list of strings, `keepWords` returns the list of lists formed by applying the function to these strings individually.

`deleteWords[{`$n_1, \dots n_k$`}]` peforms the complementary action.

`pairWords[`m, n`]` accepts a single integer or a list of integers in place of m and/or in place of n.

1. When m is an integer, a name is formed from the m'th word.
2. When m is a list of integers, a name is formed by coalescing the words to which these point, in accordance with the conventions described above.
3. When n is an integer, the n'th word is converted to a number, or a symbol, or a value, or it is kept as a string, in accordance with the conventions described above.
4. When n is a list of integers, the words to which these point are treated in the same way individually.
5. The function returns *name*`==`*result*, where *name* and *result* are the items formed by consideration of the arguments m and n, respectively.
6. When the current extract is a list of strings, the function returns the list of `==` statements formed from these individually, in the manner just described.

`pairWords[`$\{\{m_1, n_1\}, \ldots, \{m_\ell, n_\ell\}\}$`]` generalizes this action as follows.

1. When the current extract is a single string, the function returns the list of items formed by `pairWords[`m_1, n_1`]`, …, `pairWords[`m_ℓ, n_ℓ`]`.
2. When the current extract is a list of strings, the function returns the list consisting of the the lists just described, for these successive strings.

`putEqualAtSpace[`n`]` acts as follows.

1. When the current extract is a single string, the function returns w`==`$\bar{v}$, where w is the name formed from words 1 to n, and v is the value from the remaining word(s) in the line, in accordance with the conventions described for earlier functions in this section.
2. When the current extract is a list of strings, the function returns the list of items that it forms from these individually.

`rejoinRhs` acts on the top element of the association list when this is of the form *name*`==`$\{w_1, w_2, \ \ldots, w_n\}$, where the w_i are strings, and returns the `Equal` expression *name*`==`$w_1 w_2 \ \ldots w_n$, where the right hand side is the concatenation of the w_i.

`splitOnSpaces` splits the current extract into the list of words (substrings) that are separated by spaces when the current extract is a single string. When the current extract is a list of strings, the function constructs the list of word lists that it would form separately from these.

`toEachLine[`$f_1, f_2, \ldots, f_n$`]` changes `currentExtract`, when this is a single string s, to f_n`[`$\ldots f_2$`[`f_1`[`s`]]`$\ldots$`]`. When `currentExtract` is a list, each element s_i is replaced by f_n`[`$\ldots f_2$`[`f_1`[`s_i`]]`$\ldots$`]`.

`unfold` converts the folded representation of a rectangular matrix to the form of a simple list of lists.

`unfoldTriangular` converts the folded representation of the triangular abbreviation of a symmetric matrix to the form of a simple rectangular list of lists.

2.6 Flow of control

`if[`*criterion*`][{`*innerScanList*`}]` applies the inner scan list to the current residue. The criteria that are coded at present are

`currentLineContains[`*key*`]`,

`currentLineDoesNotContain[`*key*`]`,

`residueContains[`*key*`]`,

`residueDoesNotContain[`*key*`]`.

`exhaustively[{`*innerScanList*`}]` tries to apply the inner scan list to the current residue. If the attempt is successful, *e.g.* if the inner scan list begins with `skipToNext[`*key*`]` and the current residue contains *key*, then the program tries to apply the inner scan list to the residue that is left. This process is repeated until the attempt to apply the inner scan list to the diminishing residue has no effect.

`break` ends the action of `extractDataFrom` immediately and gracefully. It helps debug a long scan list.

2.7 Miscellaneous scan list commands

`math[Hold[`*mathematicaExpression*`]]` executes the MATHEMATICA expression that it contains. §5.3 describes a substantial example. The `pipe` and other unary functions described in §3.5 are very useful in the `math` command.

The following commands are applied to lists of items extracted from a data file by the methods described above or by the methods in the section that follows.

`tagConsecutively[{`a_1`==`$b_1, \ldots a_n$`==`v_n`}]` $\longrightarrow$ `{`a_1`[1]==`$b_1, \ldots a_n$`[`n`]==`v_n`}]`.

`coalesceUnderCommonName[{`x`==`v_1, x`==`$v_2, \ldots, x$`==`v_n`}]` is applied to a list of items selected from the output. It returns x`=={`$v_1, v_2, \ldots, v_n$`}`.

3 Auxiliary commands

The MATHSCOUT package contains several more commands that are useful in mechanized documentation and in the `math` command mentioned above.

3.1 Finding and extracting lines

Documentation sometimes requires the extraction of a line or several lines of a log file. The following commands address this need.

`lineNumbersOf[`*key*`][`*listOfStrings*`]` returns the list of pointers to the lines that contain the key as a substring.

`extractedLines[`n_1, n_2`][`*listOfStrings*`]` returns the list consisting of lines n_1 through n_2.

`extractedLinesContaining`[key] [$listOfStrings$] returns the list of lines that contain the key.

`extractedLineContaining`[key] [$listOfStrings$] returns the first member of this list.

`extractedLinesBeginning`[key, n] [$listOfStrings$] returns the list of n lines starting with the first that contains the key.

`extractedLinesContainingAtEnds`[key_1, key_2] [$listOfStrings$] returns the list of lines that begins with the first that contains key_1 and ends with the first subsequent line that contains key_2.

3.2 Operating on strings

Operations on strings are needed in documentation and, at times, in the operand of the scanning command `math`. The following commands address these needs.

The formal product $n * s$ or simply $n\, s$, where n is an explit integer and s is a string, returns the concatenation of n copies of s.

`despaced`[s] elides all spaces from a string.

`deleteIandRflags`[s] deletes isolated letters `I` and `R`. It is used delete these when they occur as flags in a checkpoint file.

`embedIn`[l, r] [s] returns the concatenation of l, s and r.

`spaceEvenly`[s] removes carriage return characters `\r`, ensures a space before and after each `=` symbol, then removes redundant spaces.

`spacedStringToStringList`[s] separates the substrings bounded by spaces into the successive elements of a list of strings.

`spacedStringToList`[s] further converts the elements of this list that have the syntax of MATHEMATICA numbers and symbols into these, and retains the other elements as strings.

`stringJoin`[$v_1, v_2, \ldots$] converts any arguments v_1 that are not strings into strings, and then applies `StringJoin`.

3.3 Operating on numbers

Operations on numbers are needed in documentation and, at times, in the operand of the scanning command `math`. The following commands address this need.

`average` and `percentageSpread` perform trivial arithmetical operations for the table that shows the convergence of the zinc oxide calculation in the Introduction of the Tutorial.

`integerList`[$n_1 - n_2$] returns the list of integers $n_1, n_1 + 1, \ldots, n_2$ when n_1 and n_2 are explicit integers.

The MATHSCOUT function `makeNumber` converts an input datum that has the syntax of an integer, real or floating point number to the corresponding MATHEMATICA representation, by default. For tabular display, as in Table 1 in §1 of the Tutorial,

1. the function `roundToDecimalPlace[`n`]` rounds the number to the specified precision and converts the result to a string,
2. the switch `preventConversion = True` forces `makeNumber` to return a number as a string.

`linearizeReal` converts a number from the MATHEMATICA `Real` format to a character string, containing `*10^`n if necessary.

3.4 LaTeX tabular output

The function `writeLaTeXtable[`$fileName, data$`]` writes the

`\begin{tabular} ... \end{tabular}`

representation of the table to disc with the specified file name. The data includes

1. `columnSpecifications[`*colSpecs*`]`, where *colSpecs* gives the argument denoted *cols* in the LaTeX literature, using the MATHSCOUT abbreviation typified by `3*"r|"` for `"r|r|r|"`,
2. `tabbedLine[`*items*`]`, that constructs the appropriate representation with interspersed `&` tab symbols and a final `\\` return code,
3. `hline` and `vline`, that introduce `\hline` and `\vline`, respectively.

The introduction of further features of the `tabular` environment is easy and circumscribed.

3.5 Composition

The `pipe` function performs right-to-left composition, *i.e.*,

x `// pipe[`$f_1, f_2, \ldots, f_n$`]` $\longrightarrow$ $f_n(\ldots f_2(f_1(x))\ldots)$.

`toTheLhs[`s`]` and `toTheRhs[`s`]` perform the corresponding composition on the left and right hand sides of the `Equal` expression s *in situ*. For example,

`x == y // toTheRhs[f, g]` $\longrightarrow$ `x == g[f[y]]`.

These functions can be used in a `pipe` expression. For example,

`x == y // pipe[toTheLhs[f], toTheRhs[g]]` $\longrightarrow$ `f[x] == g[y]`.

`toEachElement[`*action*`][`s`]` performs the composition on each element of the list s and, in general, to each argument of s if its full form is h`[`$args$`]`.

`toElement[`n`][`*action*`][`s`]` performs it on the n'th element (argument).

`toElements[{`$n_1, n_2, \ldots$`}][`*action*`][`s`]` performs it on the elements (arguments) at positions $n_1, n_2, \ldots$.

The functions `toTheLhs`, ..., `toElements` are examples of "targeting functions" that target specified subexpressions of the overall argument. The function `pipe` and the principle of targeting are important components of MATHSCAPE, which includes many more targeting functions (M. P. Barnett, Mathscape and Molecular Integrals, Journal of Symbolic Computation, 42 (2007) 265–289 and `http://www.princeton.edu/~allengrp/ms/mmi`).

The MATHSCOUT package replicates several MATHSCAPE functions for self-sufficiency. A simple example of elementary MATHEMATICA usage introduces a key feature of these. The expression `StringReplace["Hello","ll"->""]` returns `"Heo"`. We define the unary function `stringDelete` by

```
stringDelete[t_String][s_String] := StringReplace[s, t-> ""]
```

Then the "ll" is deleted from *any* string by the parameterized unary function (operator) `stringDelete["ll"]`. Thus, `stringDelete["ll"]["Hello"]` returns `"Heo"`, `stringDelete["ll"]["all"]` returns `"a"`, and so on. Besides those already mentioned in this subsection, there are unary wrappers for about 20 builtin MATHEMATICA functions of two arguments, to facilitate composition. Typically, `apply[`f`][`s`]` wraps `Apply[`f, s`]` and `delete[`n`][`s`]` wraps `Delete[`s, n`]`. Similarly,

`deleteCases`, `drop`, `extractPart` and `extractParts`, `fixedPoint`, `insert`, `map`, `partition`, `plus`, `stringAppend`, `stringDelete`, `stringDrop`, `stringPosition`, `stringReplace`, `stringTake`, `take`, `times`

wrap the MATHEMATICA functions that have the corresponding names that begin with a capital letter. Their actions are obvious from the defining statements in the actual package.

`extractLhs` and `extractRhs` extract the sides of an `Equal` expression.

4 Mechanized documentation

Three further commands perform major documentation functions. This section describes them briefly. Full details are given in the Development Guide in the development package.

4.1 Constructing the displays for the Tutorial

The line breaks and indention in the examples of MATHSCOUT commands in the Tutorial and in this User's Guide simply replicate what we typed. The `embedExtracts` function incorporates these statements by reading them, line by line, using a MATHEMATICA combination that is, typically,

```
currentLine = Read["displays.in", String];
WriteString["tutorialAll.tex", currentLine];
```

Lines that are copied this way will not overset, because we typed them to fit within the typeset line length, with the line breaks and the indentions that emphasize the structure of the statements that they comprise. An evaluated result that exceeds the allowed line length is folded under program control. Also, the structure of some lines that would fit is emphasized by folding. For example, the symbolic Z matrix is typeset at the start of §3.1 of the Tutorial as

```
{SymbolicZmatrix ==
{{O},
{H, 1, r},
{H, 1, r, 2, a}}}
```

It would be set, by default, as

```
{SymbolicZmatrix == {{O}, {H, 1, r}, {H, 1, r, 2, a}}}
```

It is broken into lines by

```
formattedSymbolicZmatrix =
 symbolicZmatrix // toElement[1][ foldNamedListFully];
```

This uses the targetable operator `foldNamedListFully`, that folds after the `==` symbol and at the end of each list element. We also use the targetable operators `foldListFully`, which acts recursively to depth 2, and `foldAfterEqual`, and the switch `foldAllListsFully`. The development package contains several examples of the usage of these operators, and examples of a few simple tactics to abbreviate displays that are repetitive.

4.2 Embedding the displays in the Tutorial

The MATHSCOUT statement `embedExtracts[inRoot, outRoot]` reads the LaTeX file `inRoot.tex` that contains lines of the forms

1. `\\!!`$externalFileName$`!!\\`,
2. `\\!!`$externalFileName, startLineNumber$`!!\\`,
3. `\\!!`$externalFileName, startKey$`!!\\`,
4. `\\!!`$externalFileName, startLineNumber, lineCount$`!!\\`,
5. `\\!!`$externalFileName, startKey, lineCount$`!!\\`,
6. `\\!!`$externalFileName, startKey, offset, lineCount$`!!\\`,
7. `\\!!`$externalFileName, n$`[`$startKey$`]!!\\`,
8. `\\!!`$externalFileName, n$`[`$startKey$`]`$, lineCount$`!!\\`,
9. `\\!!`$externalFileName, n$`[`$startKey$`]`$, offset, lineCount$`!!\\`.

Each of these lines is replaced by

```
\small\begin{verbatim}
...
\end{verbatim}\normalsize
```

where the ... stand for lines extracted from the specified external file. for the five cases as follows:

1. the entire file,
2. from the starting line number to the end of the file,
3. from the first line that starts with the specified key to the end of the file,
4. the specified number of lines beginning at the specified line number,
5. the specified number of lines beginning with the first line that starts with the specified key,
6. the specified number of lines beginning at the specified offset from the first line which starts with the specified key,

7. from the n'th line that starts with the specified key to the end of the file,
8. the specified number of lines beginning at the n'th line which starts with the specified key,
9. the specified number of lines beginning at the specified offset from the n'th line which starts with the specified key.

4.3 The usage files

Testing the many special cases that can occur in the execution of a function is a major chore in the development of a software package. The MATHSCOUT function `runTests[`$mode, inFile, outFile$`]` addresses this need. A typical item in a file that it processes is

```
test[keepWords, oneLine, makeName] =
 extractDataFrom[ {"abra cadabra", "double double toil and trouble"},
  using[{makeNameFromCurrentLine, extractCurrentLine, keepWords[{1, 3}]}]]
```

The corresponding item in the output file, in the default mode, is

```
test[keepWords, oneLine, makeName] =
 extractDataFrom[ {"abra cadabra", "double double toil and trouble"},
  using[{makeNameFromCurrentLine, extractCurrentLine, keepWords[{1, 3}]}]]
=>
{abraCadabra == {"double", "toil"}}
--------------------------------------------------------
```

The input file consists of (1) statements of the form `test[`id`] =` ... and (2) comments. Full details are given in the development package.

4.4 Miscellaneous housekeeping

`indexTheScripts[`$oldName, newName$`]` is used to put a date and time stamp and an index to the scripts at the beginning of the MATHSCOUT package when it is updated.

`concatenateFiles[`$listSpecification$`]` constructs a single file that consists of the specified files, each under an identifying header. We wrote it to help print collections of short files compactly.

`compareFiles[`f_1, f_2`]` compares the files with the specified names, ignoring lines that specify the date and time of the run and the hardware and software platform.

`compareDirectories[`$d_1, d_2, listSpecification$`]` compares the specified files in directory d_1 with the files with the same names in directory d_2. Any further files in either directory are ignored. The output reports the number of files that match and the number that do not match. Mismatches of POSTSCRIPT and other files are reported separately.

`compareDirectories[{{`a_1, b_1`},{`a_2, b_2`},`$\ldots$`}}]` compares all the files in directories $a_1, a_2, \ldots$ with the corresponding files in directories $a_2, b_2, \ldots$, respectively, and reports the individual and total numbers of matches and mismatches.

5 The extraction of data for water and zinc hydrate

5.1 Splitting the file into major pieces

The file `full.auto` extracts all the data from the Gaussian log files from calculations on water and zinc hydrate. Each file is split into five main pieces by the statement

```
majorSplit = extractDataFrom[waterLog, using[scanList[0]]];
```

where

```
scanList[0] =
{skipTo["Gaussian, Inc., Pittsburgh PA, 2003"],
 skip[2 lines],
 callSuccessiveLinesIncluding[second["GradGrad"]][piece[1]],
 callSuccessiveLinesIncludingNext["GradGrad"][piece[2]],
 callSuccessiveLinesIncludingNext["Optimization completed"][piece[3][all]],
 callSuccessiveLinesIncludingNext["Normal termination "][piece[4]],
 callSuccessiveLinesIncludingNext["Normal termination "][piece[5]]};
```

The action on the water file is typical. The name `majorSplit` is assigned to the following list.

```
{piece[1] ==
{Gaussian 03: x86-Win32-G03RevB.04 2-Jun-2003,
05-Jun-2005,
<<48 lines>>,
Number of steps in this run = 20 maximum allowed number of steps = 100.,
GradGradGradGradGradGradGradGradGradGradGradGrad},
piece[2] ==
{Input orientation:,
---------------------------------------------------,
<<127 lines>>,
Cartesian Forces: Max 0.013276618 RMS 0.008181775,
GradGradGradGradGradGradGradGradGradGradGradGrad},
piece[3][all] ==
{Berny optimization.,
Internal Forces: Max 0.012670009 RMS 0.010380286,
<<222 lines>>,
Predicted change in Energy = -2.522250D-08,
Optimization completed.},
piece[4] ==
{-- Stationary point found.,
----------------------------,
<<111 lines>>,
File lengths (MBytes): RWF = 12 Int = 0 D2E = 0 Chk = 7 Scr = 1,
Normal termination of Gaussian 03 at Sun Jun 05 17:43:01 2005.},
piece[5] ==
{Link1: Proceeding to internal job step number 2.,
---------------------------------------------------,
<<14 lines>>,
File lengths (MBytes): RWF = 12 Int = 0 D2E = 0 Chk = 7 Scr = 1,
Normal termination of Gaussian 03 at Sun Jun 05 17:43:42 2005.}}
```

The names `piece[1]`, ..., `piece[5]` are assigned to the five subsidiary lists by

```
majorSplit /. Equal -> Set;
```

The 2nd, 3rd and 5th are split into still smaller pieces. Piece 2 is split by

```
split[2] = extractDataFrom[piece[2], using[scanList[2]] ];
```

where

```
scanList[2] =
{callSuccessiveLinesIncludingNext["NBsUse"][piece[2.1]],
  callSuccessiveLinesIncludingNext["******"][piece[2.2]]   ,
  callTheResidue[piece[2.3]]  } ;
```

Piece 3 is split into separate pieces that contain data for the successive Berny iterations by

```
iterationCount =
 piece[3] // lineNumbersOf["Step number"] // Length

iterationSplitter =
 {Table[
   callSuccessiveLinesIncludingNext[
    "GradGradGradGradGradGrad"][piece[3[ii]]],
  {ii, 2 * iterationCount - 4}],
  callTheResidue[piece[3[2* iterationCount - 3]]]} //
   Flatten;

split[3] =
 extractDataFrom[piece[3], using[iterationSplitter]];

split[3] /. Equal -> Set;
```

Piece 5 is split by

```
split[5] = extractDataFrom[piece[5], using[scanList[5]] ];
```

where

```
scanList[5] =
{callSuccessiveLinesBeforeNext["Exact polarizability"][ piece[5.1]],
  callSuccessiveLinesBeforeNext["- Thermochemistry -"][piece[5.2]],
  callSuccessiveLinesIncludingNext["GradGrad"][piece[5.3]],
  callTheResidue[piece[5.4]]};
```

5.2 Extracting data from the major pieces

The individual data are extracted from `piece1`, `piece2.1`, ..., `piece5.4`, by `scanList[`*id*`]`, where *id* is `1`, `2.1`, `2.2`, `2.3`, `3[1]`, `3[even]` and `3[odd]`(which act in alternation on all the subsidiary pieces of `piece3` except the first and last), `3[-1]`, which acts on the last, and `4`, ..., `5.4`. These scan lists consist largely of modules `scanList[A]`, ..., `scanList[M]`, that act on sequences of lines in the successive pieces as follows. The lines are numbered to facilitate inspection in the files `piece1`, ..., `piece5.4`.

`scanList[A]`:

piece1: water 40–50, zinc hydrate 108–318,
piece4: water 1–10, zinc hydrate 1–211
piece5.1: water 29–39, zinc hydrate 45–255.

`scanList[B]`:
piece2.1: water 1–30, zinc hydrate 1–108,
piece3[even]: water 1–30, zinc hydrate 1–108,
piece3[-1]: water 31–60, 225–333,
piece4: water 13–42, 213–320,
piece5.1: water 43–72, zinc hydrate 259–366.

`scanList[C]`:
piece2.1: water 31–45, zinc hydrate 109–124,
piece3[even]: water 31–45,
piece3[-1]: water 41–75,
piece5.1: water 73–83,

`scanList[D]`:
piece2.2: water 5–9, zinc hydrate 5–14,
piece3[even]: 47–51,
piece3[-1]: 77–81,
piece5.1: water 90–94,

`scanList[E]`:
piece2.2: water 11–20, zinc hydrate 16–25,
piece3[even]: water 52–59,
piece3[-1]: water 83–89,
piece5.1: water 95–102,

`scanList[F]`:
piece2.3: water 1–32, zinc hydrate 1–158,
piece4: water 44–75,
piece5.1 water 128–159,

`scanList[G]`:
piece2.3: water 33–56, zinc hydrate 159–182,
piece4: water 76–99,
piece5.1: water 172–195,

`scanList[H]`:
piece2.3: water 57–66, zinc hydrate 183–209,
piece3[even]: water 60–69,
piece3[-1]: water 90–99,
piece5.3: water 45–54,

`scanList[I]`:
piece3[1]: water 1–4, zinc hydrate 1–4,
piece3[-1]: water 1–4, 101–104, zinc hydrate 1-4, 400–403,

`scanList[J]`:
piece3[1]: water 7–11,
piece3[-1]: water 8–12, 108–112,

`scanList[K]`:
piece3[1]: water 12, zinc hydrate 7–47,

piece3[-1]: water 13, 115,

`scanList[L]`:
piece3[1]: water 15–28, zinc hydrate 52–265,
piece3[-1]: water 16–29, 118–128, 409–629,

`scanList[M]`: (this subsumes several of the preceding lists):
piece3[-1]: water 1–29, zinc hydrate 1–629.

5.3 Using explicit Mathematica in a scan

The extraction of the eigenvalues at the start of `piece3[1]` and `piece3[-1]` uses the `math` command. This provides a way of using explicit MATHEMATICA code in a scan. For zinc hydrate, the relevant lines of `piece3[1]` are

```
Berny optimization.
...
Second derivative matrix not updated -- first step.
Eigenvalues --- 0.00436 0.00798 0.01010 0.01090 0.01860
...
Eigenvalues --- 0.38249 0.38780 0.38896 0.39265 0.41129
Eigenvalues --- 0.415291000.000001000.000001000.000001000.00000
Eigenvalues --- 1000.000001000.000001000.000001000.000001000.00000
Eigenvalues --- 1000.000001000.000001000.000001000.000001000.00000
...
Eigenvalues --- 1000.000001000.000001000.000001000.000001000.00000
Eigenvalues --- 1000.000001000.000001000.00000
RFO step: Lambda = -4.61406065D-02.
```

The extraction of the eigenvalues from the water file is handled adequately by

```
scanList[K] =
{extractSuccessiveLinesBeginning["Eigenvalues"],
  deleteWords[{1, 2}], callTheResult[eigenvalues]};
```

For zinc hydrate, this works for the first few lines, but then leaves the coalesced lines that run the values 1000.00000 together, without intervening spaces, as character strings. To accommodate this, we wrote a short MATHEMATICA script called `putSpacesInCoalescedNumbers`, and we modified `scanList[K]` to

```
scanList[K, modified] =
{extractSuccessiveLinesBeginning["Eigenvalues"],
  math[Hold[putSpacesInCoalescedNumbers]],
  splitOnSpaces,
  callTheResult[eigenvalues]};
```

The procedure `putSpacesInCoalescedNumbers` is included in `full.auto`, without any need to alter the actual MATHSCOUT package. The way that this, and any further procedures, can be incorporated in the package, is explained in the distribution package. This would allow the omissionof the `math[Hold[...]]` wrapping.

6 The distribution package

The distribution package `msctDistribution.tar.gz` is 1.36 MB long. It unpacks to 2.38 MB. The unpacked files include

1. The MASCOUT package. This has the file name `msct.m` (.96 MB).
2. The pdf files of the Tutorial (.58 MB) and the User's Guide (.53 MB).
3. The directory `scanDem` (1.13 MB). This contains
    - (a) the Gaussian `log` files for the water molecule and the zinc hydrate ion, and
    - (b) the control file `full.auto` that contains the scanning commands to extract data from these and from the corresponding files for other chemical species.
4. The directory `docDem` (.56 MB). This contains a prototype for using the automated documentation component of MATHSCOUT

The README file contains scripts that can be cut and paste to run the data extraction and the documentation prototype.