The MATHSCOUT Development Package

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 28, 2007

Introduction

The development package `msctDevelopment.tar.gz` is unpacked by `tar -xvvzf msctDevelopment.tar.gz` to a superset of the files that comprise the distribution package `msctDistribution.tar.gz`. The content of this superset and its organization in a hierarchy of directories facilitate a rapid and rigorous test of the effect of changes to the package and to the supporting material.

When we began to develop the MATHSCOUT package, we had to check the effects of changes to the MATHSCOUT functions, to the test data and to the documentation by laborious cutting and pasting into interactive MATHEMATICA sessions that took about an hour. The material in the development package provide a completely automated process that takes about three minutes. This process produces a report. The final line shows that the results are unaffected by the changes to the system, when this is the case. When the results have been affected, the report pinpoints the effects concisely. We call the process "validation" in accordance with software engineering terminology.

In this report,

- §1 gives a full account of the methods that we use to produce the displays that are included in the Tutorial,
- §2 deals with the `embedExtracts` function that copies verbatim displays into a LaTeX file from other text files,
- §3 deals with the `runTests` function that evaluates files of statements that test special cases and diagnostics of individual commands systematically,

---

[*] michaelb@princeton.edu

[†] joseph.capitani@manhattan.edu

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

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

- §4 explains the scripts that consist of a mixture of UNIX and MATHEMATICA code, that we use to invoke the processes discussed in §§1–3,
- §5 explains the validation process,
- §6 explains some of the coding of the MATHSCOUT scripts.

The topics 1–3 are discussed briefly in §4.1–§4.3 of the User's Guide.

1 Constructing the displays for the Tutorial

The statement `<<displays.in` constructs most of the displays that appear in the tutorial. The file `displays.in` consists of

1. some trivial housekeeping statements,
2. `inputGaussianFile` statements that load the output of our earlier Gaussian calculations on the water molecule, the nitrogen molecule and the zinc hydrate ion,
3. `extractedLineContaining` and `extractedLinesContaining` statements that extract a single line and a block of consecutive lines from the log lists,
4. assignment statements that give names to short scan lists,
5. `extractDataFrom` statements that apply scan lists to extracted pieces of the log lists,
6. `writeListFolded[`*outputFileName*`][`*listOfData*`]` statements, that write formated lists in accordance with the conventions described below,
7. statements that (i) abbreviate successive items of common structure within a line, and (ii) abbreviate a block of consecutive lines that have a common structure,
8. formating statements,
9. `readFile` statements, that read back the files which have just been written, for visual assurance,
10. MATHEMATICA `ListPlot` and `Show` graphics commands that create the graphs, and the `Display` command that writes the POSTFIX representations of these.

The statements that are displayed in the Tutorial and in this User's Guide simply replicate the format in which they were 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 `foldNamedListFully` operator. The action of the three other fold operators is shown in the next statement. This formats the list of multipole values that comprise the second display in §3.6 of the Tutorial, *i.e.*

```
{dipoles ==
 {X == 0, Y == 0, Z == -1.9678, Tot == 1.9678},
quadrupoles ==
 {XX == -7.2125, YY == -4.2883, ZZ == -6.0338, XY == 0,
XZ == 0, YZ == 0},
tracelessQuadrupoles ==
 {XX == -1.3676, YY == 1.5566, ZZ == -0.1889, XY == 0,
XZ == 0, YZ == 0},
octapoles ==
 {XXX == 0, YYY == 0, ZZZ == -1.1326, XYY == 0,
XXY == 0, XXZ == -0.2851, XZZ == 0, YZZ == 0,
YYZ == -1.2209, XYZ == 0},
hexadecapoles ==
 {XXXX == -5.3379, YYYY == -6.048, ZZZZ == -6.2481, XXXY == 0,
XXXZ == 0, YYYX == 0, YYYZ == 0, ZZZX == 0,
ZZZY == 0, XXYY == -2.1694, XXZZ == -1.9818, YYZZ == -1.7308,
XXYZ == 0, YYXZ == 0, ZZXY == 0}}
```

The default representation cannot be displayed because it contains lines that would overset, but the formating action is apparent from the comparison of this display with the statement that folded the direct result of the extraction. This action was imposed by

```
formattedMultipoles =
multipoles //
 pipe[
  toElement[1][foldAfterEqual],
  toElement[2][
   toTheRhs[foldAfterElements[{4}]],
   foldAfterEqual],
  toElement[3][
   toTheRhs[foldAfterElements[{4}]],
   foldAfterEqual],
  toElement[4][
   toTheRhs[foldAfterElements[{4,8}]],
   foldAfterEqual],
  toElement[5][
   toTheRhs[foldAfterElements[{4,8,12}]],
   foldAfterEqual],
 foldListFully];
```

Folding is performed, in all, by

1. the targetable operator `foldAfterEqual`,
2. the targetable operator `foldListFully`, (this acts recursively to depth 2),
3. the targetable operator `foldNamedListFully`, (this combines the action of the two previous operators),
4. the switch `foldAllListsFully` (when this is set to `True`, each subsequent output line that is a list is treated as if `foldListFully` had been applied to it).

Abbreviation: We use a few simple tactics for abbreviation that provide more control than the builtin MATHEMATICA `Short` function. The following examples illustrate these tactics.

The first display in §3.4 of the tutorial, *i.e.*,

```
{Population analysis using the SCF density.,
...,
Condensed to atoms (all electrons):,
1 2 3 4 5 6,
1 Zn 10.413341 0.104376 -0.005140 -0.005895 0.103698 -0.005659,
<<17 lines>>,
19 H -0.005843 -0.000004 0.000000 0.000001 -0.000161 0.000004,
<<40 lines>>,
19,
1 Zn -0.005843,
<<17 lines>>,
19 H 0.322839,
Mulliken atomic charges:}
```

was produced from a 117–line piece of the zinc hydrate file by the statements

```
populationCursor =
 lineNumbersOf["Population analysis"][zincHydrateLog][[1]]

condensedCursor =
  lineNumbersOf["Condensed"][zincHydrateLog][[1]] -
   populationCursor + 1

shortFilletedInputForPopulationAnalysis =
extractedLinesContainingAtEnds[
 "Population", "Mulliken"][zincHydrateLog] //
 {#[[1]],
   "...",
   Take[#, {condensedCursor, condensedCursor+2}],
   "<<17 lines>>",
   Take[#, {condensedCursor+20, condensedCursor+20}],
   "<<40 lines>>",
   Take[#, {condensedCursor+61, condensedCursor+62}],
    "<<17 lines>>",
    Take[#, {condensedCursor+80, -1}]}& //
   Flatten;
```

The 4th display in §6.1 of the tutorial, *i.e.*

```
{{{0.6, -106.851}, <<24 pairs>>, {3.1, -108.914}}}
```

was produced by shortening a list of 26 pairs of coordinates called `energy[r]`. This was done by executing the following statement, that uses the MATHEMATICA `#...&` notation for a "pure function".

```
energy[r] // {#[[1]], "<<24 pairs>>", #[[-1]]}&
```

The 2nd display in §6.2 of the tutorial, *i.e.*

```
{distanceMatrix[1] ==
{{{0.}, {2.5445, 0.}, {3.0579, 1.01338, 0.}},
<<15 rows>>,
{3.05488, 4.47683, 4.2581, <<12 items>>, 4.97422, 4.97422, 1.63588, 0.}},
<<28 matrices>>,
distanceMatrix[30] ==
{{{0.}, {2.11744, 0.}, {2.7904, 0.977739, 0.}},
<<15 rows>>,
{2.78308, 3.50259, 3.3594, <<12 items>>, 4.46074, 4.46074, 1.6095, 0.}}}
```

was produced in a slightly more elaborate way. The entire list of 30 unfolded matrices, called `allDistanceMatrices`, extracted by the statement comprising the first display in §6.2 of the tutorial, was abbreviated by

```
filletedAllDistances =
 allDistanceMatrices //
  pipe[
  {#[[ 1]], "<<28 matrices>>", #[[-1]]}&,
  toElements[{1, -1}][
   toTheRhs[
  {#[[{1,2,3}]], "<<15 rows>>",
    ({#[[19, {1,2,3}]], "<<12 items>>",
       #[[19, {-4, -4, -2, -1}]]}  // Flatten) }&]]]
```

As mentioned in §3.5 of the User's Guide, the `pipe` function performs right-to-left composition, *i.e.*,

$$x \quad \texttt{// pipe}[f_1, f_2, \ldots, f_n] \quad \longrightarrow \quad f_n(\ldots f_2(f_1(x))\ldots).$$

The successive items in this expression act as follows.

1. The first item in the `pipe` expression uses the tactic described for the `energy[r]` example to reduce the entire list of 30 `Equal` statements with a matrix on the right hand side, to a list of 3 items, in which the middle item is the abbreviation `<<28 matrices>>`.
2. The targeting function `toElements[{1, -1}]` focuses action on the first and final elements of the abbreviated list, *i.e.*, the `Equal` statements containing the first and final matrices.
3. The targeting function `toTheRhs` focuses action on the right hand side of each.
4. An elementary combination of the MATHEMATICA `#...&` and `[[{...}]]` notations for a pure functions and a lists of parts then performs the further abbreviation in the same way that step 1 performed the outermost abbreviation.

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.

The rest of this section provides readers with prototypes and tests all the embed commands exhaustively.

Case 1 is illustrated by the line `\\!!"forThermal"!!\\` which follows the present line in `usersGuideRaw.tex`.

```
{Thermal correction to Energy = 0.024185,
Thermal correction to Enthalpy = 0.025129,
Thermal correction to Gibbs Free Energy = 0.003701,
Sum of electronic and zero-point Energies = -76.427488,
Sum of electronic and thermal Energies = -76.424653,
Sum of electronic and thermal Enthalpies = -76.423709,
Sum of electronic and thermal Free Energies = -76.445137}
```

Case 2 is illustrated by the line `\\!!"forThermal", 6!!\\` which follows the present line in `usersGuideRaw.tex`.

```
Sum of electronic and thermal Enthalpies = -76.423709,
Sum of electronic and thermal Free Energies = -76.445137}
```

Case 3 is illustrated by the line `\\!!"forThermal", "Sum"!!\\` which follows the present line in `usersGuideRaw.tex`.

```
Sum of electronic and zero-point Energies = -76.427488,
Sum of electronic and thermal Energies = -76.424653,
Sum of electronic and thermal Enthalpies = -76.423709,
Sum of electronic and thermal Free Energies = -76.445137}
```

Case 4 is illustrated by the line `\\!!"forThermal", 3, 2!!\\` which follows the present line in `usersGuideRaw.tex`.

```
Thermal correction to Gibbs Free Energy = 0.003701,
Sum of electronic and zero-point Energies = -76.427488,
```

Case 5 is illustrated by the line `\\!!"forThermal", "Sum", 1!!\\` which follows the present line in `usersGuideRaw.tex`.

```
Sum of electronic and zero-point Energies = -76.427488,
```

Case 6 is illustrated by the line `\\!!"forThermal", "Sum", 1, 2!!\\` which follows the present line in `usersGuideRaw.tex`.

```
Sum of electronic and thermal Energies = -76.424653,
Sum of electronic and thermal Enthalpies = -76.423709,
```

and, for negative offset, by the line `\\!!"forThermal", "Sum", -1, 2!!\\` which follows the present line in `usersGuideRaw.tex`.

```
Thermal correction to Gibbs Free Energy = 0.003701,
Sum of electronic and zero-point Energies = -76.427488,
```

The subsidiary case of a null item implying zero offset is illustrated by the line `\\!!"forThermal", "Sum", , 2!!\\` which follows the present line in `usersGuideRaw.tex`.

```
Sum of electronic and zero-point Energies = -76.427488,
Sum of electronic and thermal Energies = -76.424653,
```

Case 7 is illustrated by the line `\\!!"forThermal", 2["Sum"]!!\\` which follows the present line in `usersGuideRaw.tex`.

```
Sum of electronic and thermal Energies = -76.424653,
Sum of electronic and thermal Enthalpies = -76.423709,
Sum of electronic and thermal Free Energies = -76.445137}
```

Case 8 is illustrated by the line `\\!!"forThermal", 2["Sum"], 2!!\\` which follows the present line in `usersGuideRaw.tex`.

```
Sum of electronic and thermal Energies = -76.424653,
Sum of electronic and thermal Enthalpies = -76.423709,
```

Case 9 is illustrated by the line `\\!!"forThermal", 2["Sum"], 1, 2!!\\` which follows the present line in `usersGuideRaw.tex`.

```
Sum of electronic and thermal Enthalpies = -76.423709,
Sum of electronic and thermal Free Energies = -76.445137}
```

The diagnostic for a request containing too many commas is triggered by

!!"water.log", "(5D, 7F)", 1, 3!!

Note that the \\ line break codes are not typeset. This is because they have taken effect, to typeset the unexecuted statement on a line by itself — the reason for their inclusion in the raw LaTeX file. The diagnostic for an embed request containing a MATHEMATICA syntax error is triggered by

!!"forThermal", 2["Sum], 2!!

The diagnostic for an embed request containing a file that is not found is triggered by

!!"ForThermal"!!

The diagnostic for an embed request containing a key that is not found is triggered by

!!"forThermal", "vibrational"!!

The diagnostic for an embed request specifying the n'th occurrence of a key with invalid n is triggered by

!!"forThermal", 7["Sum"], 2!!

and for negative n by

!!"forThermal", -3["Sum"], 2!!

The diagnostic for an embed request with second argument neither integer nor string is triggered by

!!"forThermal", Sum, 2!!

The diagnostic for a request with positive offset out of range is triggered by

!!"forThermal", "Sum", 7, 1!!

The diagnostic for an embed request excessively negative offset is triggered by

!!"forThermal", "Sum", -7, 1!!

The `embedExtracts` script collects the diagnostic messages, if any occur, and displays them after the output from the LaTeX and DVIPS steps. Thus, when the present section was run independently, it produced the following list

```
Error messages from invalid embed commands.
FullForm used in 1st line of each message.

record number: 1110, too many arguments (commas)
record: \\!!"water.log", "(5D, 7F)", 1, 3!!\\
record number 1119: Mathematica syntax error
record: \\!!"forThermal", 2["Sum], 2!!\\

record number 1119: file "forThermal" not found
record: \\!!"forThermal", 2["Sum], 2!!\\

record number 1125: file ForThermal not found
record: \\!!"ForThermal"!!\\

record number 1131: key "vibrational" not found in file "forThermal"
record: \\!!"forThermal", "vibrational"!!\\

record number: 1138, invalid number of lines starting "Sum"
requested in "forThermal"
record: \\!!"forThermal", 7["Sum"], 2!!\\

record number 1144, invalid 2nd argument Times[-1, 3["Sum"]]
record: \\!!"forThermal", -3["Sum"], 2!!\\

record number 1151, invalid 2nd argument Sum
record: \\!!"forThermal", Sum, 2!!\\

record number: 1158, offset 7 out of range
record: \\!!"forThermal", "Sum", 7, 1!!\\

record number: 1165, offset -7 out of range
record: \\!!"forThermal", "Sum", -7, 1!!\\
```

3 Production of usage examples

The file `usage.in` consists of statements that exercize different paths through the individual functions of MATHSCOUT. A typical item in this file is

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

When this test is interpreted by `runTests[`*mode*, *infile*, *outfile*`]`, the following is written into the output file if the mode is `"all"` or `"valid"`.

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

The procedure `runTests` treats a statement of the form `test[`*id*`] =` … as follows.

1. The test is written to the output file
   (a) if the mode is `"all"`,

   (b) if the mode is `"valid"` and the name of the test is not of the form `test[...,"d"]`,
   (c) if the mode is `"invalid''` and the name of the test is of the form `test[...,"d"]`.

2. If the test is written to the output file (because its *id* is consistent with the mode), and it is evaluated and the value is displayed (because the test is not followed by a semicolon) , an arrow `=>` is written, followed by the output of the test. This is displayed in `FullForm` when it is `Null` and in an abbreviated form when it is long and repetitive.
3. If the test expression is written to the output file, and it generates diagnostic messages that are coded within MATHSCOUT, these messages are written to the output file.
4. If the test expression is written to the output file, and it generates MATHEMATICA diagnostic messages, these are saved and written to the output file at the end of the run.

Also,

1. The date and the time at which the run started are written at the start of the output file, followed by the model and identification of the hardware platform and the version of MATHEMATICA that are used.
2. A comment of the form `(*!...!*)` is written in the `all` and `valid` modes.
3. A comment of the form `(**...**)` is written in the `all` and `invalid` modes.
4. All other comments `(*...*)` are written in all three modes.
5. An executable MATHEMATICA statement or sequence of statements with a single `!` symbol at left and right, *e.g.*

   ```
   !waterLog = inputGaussianFile["GaussOut/water/water.log"];!
   ```

   is executed in the `"all"` and `"valid"` modes.
6. An executable MATHEMATICA statement or sequence of statements with a single `*` symbol at left and right, *e.g.*

   ```
   *fourthCouplet =
     {"Here's but three, come one more;",
       "two of both kinds make up four"}*
   ```

   is executed in the `"all"` and `"invalid"` modes.
7. The command `stop` ends the run immediately.

4 The hybrid Mathematica–Unix scripts

These are run from the working directory. `runDisplays` is typical of the UNIX scripts. It cleans out the subdirectory `displaysOut` if this contains files from a previous run, and then runs the MATHEMATICA kernel on the file of MATHEMATICA statements `forRunDisplays`. The `runDisplays` script is

```
cd msctSupport
rm -fR displaysOut
math < hybrid/forRunDisplays
cd ..
```

To invoke this, execute `msctSupport/hybrid/runDisplays` from the working directory. Control returns to the working directory. The MATHEMATICA script that is input by the 3rd line

1. loads the MATHSCOUT package,
2. changes directory to `displaysOut`, and
3. interprets the control file `displays.in` in the manner that is explained in §3. The displays are concatenated into the single file `displaysAll` for easy inspection of a printout.

```
<<"../msct.m";
Run["mkdir displaysOut"];
<<"displays.in";
concatenateFiles["displaysAll"]["*"];
Exit[]
```

In principle, the MATHEMATICA statements could be included directly in `runDisplays` by

```
cd msctSupport
rm -fR displaysOut
math -run '<<"../msct.m";  Run["mkdir displaysOut"]; \
<<"displays.in"; concatenateFiles["displaysAll"]["*"]; \
Exit[];'
```

Although this works when cut and paste, it triggers diagnostic messages and does not work when it is used as an executable.

The scripts in the `hybrid` directory write files to several other directories that are described in §5. In particular

1. `runDisplays` reconstructs the displays in `displaysOut`,
2. `runUsage` reconstructs the `.out` files in `usageOut` and `usageMore`,
3. `runScan` reconstructs the `piece` and `split` files in `water` and `zincHydrate`,
4. `runDocumentation` reconstructs the complete `tex` and `ps` files from the skeletal versions in `documentation`, for further conversion to the `pdf` files for the Tutorial, the User's Guide, this Development Guide and the short prototype document in the distribution file,
5. `regenerate` runs the four scripts just listed,
6. `compare` compares the files in `preserved` with the files that have been regenerated,
7. `cleanout` deletes the files produced by `regenerate`,
8. `preserve` writes the regenerated files into `preserved`, replacing its prior contents,

9. `update` reindexes the `msct.m` file after it has been modified,
10. `forCompare`, ..., `forUpdate` are very short files of MATHEMATICA statements that are used by some of the executable scripts which have just been listed.

5 Installation and validation

The command `tar -xvvzf msctDevelopment.tar.gz` unpacks the development package `msctDevelopment.tar.gz` into a hierarchy of directories and files in the working directory where it was downloaded. These support the following validation process that was mentioned in the Introduction.

A complete set of the files produced by the most recent succesful automated validation comprises the directory `preserved`. The validation reruns the steps that created these files and writes these new files in the directory `regenerated`. Then the files in these two directories are compared, and the matches and mismatches are recorded. In the following tabulation of the unpacked files, directories are typeset in boldface and underlined. The working directory is where where the development file was downloaded and unpacked.

- **working directory**
  - `msct.m`, README, `validate`,
  - `referenceReport` [for comparison with output of `validate`],
  - **msctSupport**
    - `displays.in` [reconstructs displays embedded in tutorial, see §1],
    - **displaysOut** [receives output of `displays.in`],
    - `usage.in` [reconstructs major demonstrations of usage, see §3],
    - **documentation** [contains files to reconstruct LaTeX files for the documentation]
      - `cbFig01.ps`, `table01.tex` [diagram and table for C_4H_4 example in Tutorial],
      - `tutorialRaw.tex`, `userGuideRaw.tex`,
      - `developmentRaw.tex`, `docDemoRaw.tex` [unembedded LaTeX files],
      - **tex** [LaTeXfiles with displays embedded]
        - `tutorial.tex`...`docDemo.tex`,
    - **GaussOut** [files from Gaussian runs used in examples]
      - `full.auto` [scan lists to extract all data from water and zinc hydrate],
      - **nitrogen**
        - `n2scan.log`,
      - **water**
        - `water.chkpt`, `water.log`,
      - **zincHydrate**
        - `zincHydrate.log`,
    - **usageOut** [to hold output produced from `usage.in`]
    - **usageMore**
      - `usageShort.in`, `usageStop.in`, `usageTrap.in` [show conventions of `runUsage`],
    - **preserved** [results for comparison with regenerated files]
      - **displaysOut**
        - `dipoles2`, ..., `zeroPoint` [41 files written by `displays.in`],
        - `displaysAll` [concatenation of these],
      - **documentation**
        - `tutorial.tex`, `userGuide.tex`, `development.tex`, `docDemo.tex`,
      - **usageMore**
        - `usageShort.in`, `usageShortAll.out`, ..., `usageTrap.out`,
        - `usageSmallFiles.out` [concatenation of the 8 other files],

- **usageOut**
  - `usageAll.out`, `usageCorrect.out`, `usageIncorrect.out` [produced from `usage.in`],
- **water** [from full extraction of water log]
  - `piece1`, ..., `piece2.3`, `piece3_01`, ..., `piece3_03`, `piece4`, ..., `piece5.4`,
  - `split1`, ..., `split2.3`, `split3_01`, ..., `split3_03`, `split4`, ..., `split5.4`,
- **zincHydrate** [from full extration of zinc hydrate log]
  - `piece1`, ..., `piece2.3`, `piece3_01`, ..., `piece3_57`, `piece4`, ..., `piece5.4`,
  - `split1`, ..., `split2.3`, `split3_01`, ..., `split3_57`, `split4`, ..., `split5.4`,
  - `split3_01mod`,
- **hybrid** [about 30 scripts to test and modify the package].

The executable `validate` in the working directory begins with a statement that is redundant when `validate` is used for the first time after downloading. The statement permits subsequent reruns by deleting all the files that `regenerate` constructs.

```
msctSupport/hybrid/cleanout;
```

Next, the `validationReport` is begun. The header is written by

```
echo ' ' > validationReport;
echo 'VALIDATION REPORT' >> validationReport;
date >> validationReport;
echo '==========' >> validationReport;
echo ' ' >> validationReport;
```

Next, an explanatory remark is written into `validationReport` followed by the output of the following UNIX pipe.

```
msctSupport/hybrid/compare | tail | grep matches >> validationReport;
```

The output of `compare` ends with a two-line summary of the comparisons, and the pipe extracts this. None of the new files have been generated yet, so the comparison shows that none of the reference files are matched. Then `regenerate` is run, followed by `compare`. This time there should be a match between many or all of the reference files and the regenerated files. The comparison pinpoints the mismatches, that can be checked to make sure that they reflect valid changes. The new results are preserved. The directory is copied into the report before and after this action, to show that the preservation did occur. Within `regenerate`, the script `runDocumentation`

1. copies the "raw" LaTeX files and the other files which provide some of the embedded displays from several directories into `displaysOut`, which contains most of the displays,
2. runs `embedExtracts` in this directory,
3. copies the `.tex` and `.ps` files into `documentation`,
4. deletes the other redundant files from `displaysOut`.

The validation can be cut and paste up to the second `compare` action, to avoid wiping out the old preserved files prematurely.

6 Internal operation

6.1 Input and output

Gaussian output files are loaded by `ReadList[..., String]` statements. Elementary character string operations remove the trailing carriage return character `\r`, remove leadinga and trailing spaces, and regularize the internal spacing. The `display.in` control file, the `usage.in` and related control files, and the "raw" `tex` files are read by `Read[..., String]` statements, in cycles that use string operations to identify

1. special records, such as the `\\!!...!!\\` lines that specify external sources of displays,
2. the breaks between successive items, and
3. the characteristics of individual items, such as whether a statement in a usage input file is to be considered in the mode that was specified in the `runTests` statement that is being executed.

These operations are also applied to the elements of the lists of strings loaded by `ReadList`.

Output is written by

1. `Write` statements, when it is assumed that the `PageWidth` option provides sufficent format control for the complete output expressions,
2. `WriteString`, when
   (a) successive lines of an input file are simply copied without change into an output file, *e.g.* in the embed process that converts a "raw" `tex` file into the complete version,
   (b) tighter control is needed, *e.g.*, to force line breaks between successive items in a list, in the files constructed by interpreting `displays.in`.

The two styles are alternated in the interpretation of usage files. `WriteString` records the input, the diagnostic messages, the `=>` connectives, and the lines of hyphens. `Write` records the evaluated expressions.

6.2 Scan control cycle

The design of processors to interpret lists of specialized commands is a commonplace elementary programming exercize. Usually, it is handled by a case statement that switches to a separate coding sequence or subprogram for each command. This requires expansion of the case statement when the command set is expanded. We avoid the use of case statements that branch on different commands. The `extractDataFrom` function operates on the entirety of a list that has been loaded from a Gaussian output file, or on a piece that has been extracted from it, as follows.

In each cycle, the script

1. increments `scanStatementNumber`, that points to the command under current attention in the scan list,

2. sets `currentAction` to this command,
3. tranfers material from the list being scanned to `currentExtract` when the command performs an extraction,
4. increases the variable `currentLineNumber`, that points to the line under current attention in the file that is being scanned, when a skip or extraction is performed,
5. appends the result of the command to the `associationList` when commands such as `useEmbeddedSpaces` are interpreted.

The interpretation of `skipNextLine` is typical. The script that interprets the scan list

1. uses `ToString` to convert the command `skipNextLine`, which has the syntactic status of a symbol, to `"skipNextLine"`, which has the status of a string,
2. converts this to the string `"$skipNextLine"`,
3. uses `ToExpression` to convert this string to the symbol `$skipNextLine`.

The MATHSCOUT package includes the `SetDelayed` statement

```
$skipNextLine := currentLineNumber = currentLineNumber + 2
```

The system evaluates the right hand side automatically, thereby increasing the line cursor by 2.

In a similar manner, `useEmbeddedSpaces`, `unfold` and other commands that have no explicit arguments are converted to function names by prepending a `$`. The command `callTheResidue[`*name*`]` is converted to `$callTheResidue`. This is the name of the function

```
$callTheResidue :=
  (AppendTo[
    associationList,
      If[Length[currentAction] >= 1,
        currentAction[[1]], nullName] ==
     Take[currentTarget, {currentLineNumber, -1}]];
    currentLineNumber = Length[currentTarget] + 1;)
```

Although this does not have an explicit argument, the main control script has assigned the actual command to `currentAction`. The name to be assigned to the residue is available, accordingly, as `currentAction[[1]]`.

Every scanning command *f*`[`*v*`]` is converted to `$`*f*, and its argument is obtained as `currentAction[[1]]`. Every command *f*`[`*v*`][`*w*`]` is converted to `$`*f*, *v* is obtained as `currentAction[[0,1]]` and *w* as `currentAction[[1]]`. Users can introduce new commands with 0, 1 or 2 arguments **without** altering the control program. All that is needed is the definition of a function to perform the required operation, that is named by prepending `$` to the keyword comprising or beginning the new command.

The methodology that has just been described allows the accommodation of further commands by very localized insertions that do **not** alter any statements that are already present.

The control mechanism uses a list of executable scan functions, *i.e.* the left hand sides of the `:=` statements that begin with a single `$`. The procedure `listExecutables` constructs this, whenever the package is loaded, by

1. using `Save["executables", "$*"]` to save these statements in whatever directory is currently in use. (This was done to avoid making the function `listExecutables` search the package on disk when the command is executed in a subdirectory of the working directory where the package resides. It may be possible to simplify the process by addressing the supernode with the `"../"` notation.)
2. using `FindList["executables", "$", AnchoredSearch -> True]` to list the first lines of all `SetDelayed` statements currently available that have names which begin with a `$` symbol,
3. eliminating the MATHEMATICA commands that begin with a `$` symbol, and eliminating the MATHSCOUT commands that begin `$$`, by requiring the 2nd character to be a lower case letter,
4. running the procedure *after* the package has been loaded, so that the `usage` message assignments have taken effect. (This refers to the MATHEMATICA *functionName*`::usage=""` statements, and not to our `usage.in` and related files.)

The incorporation of the `putSpacesInCoalescedNumbers`, mentioned at the end of §5.3 of the Tutorial, into `msct.m` provides experience of modifying the package. It takes the following steps.

1. Make a backup copy of the current version of the package.
2. Cut and paste the procedure from `full.auto` into `msct.m`.
3. Change the name of the procedure to `$putSpacesInCoalescedNumbers`, just by typing a `$` symbol.
4. Type `usage::$putSpacesInCoalescedNumbers="";` in the `usage` section at the beginning of `msctBeingModified.m`.
5. Run `indexTheScripts["msctBeingModified.m", "msct.m"]`.
6. Revalidate.

6.3 Trapping diagnostic messages

To detect expressions that generate diagnostics when evaluated, even though they are syntactically correct, without disrupting the output by their immediate display, we use a tactic that is embodied in the code

```
$Messages = {messageFile = OpenWrite["mathDiagnostics" ]};
...
If[FileByteCount["mathDiagnostics" ] == 0,normal action, error action];
```

where ... stands for an expression that the user expects to be evaluable, but which may generate diagnostic messages in unforseen circumstances. This construction is used in `runTests`. Variations are used in the functions `$unfold` and `$unfoldTriangular`.

6.4 Indexing the package mechanically

The MATHSCOUT package consists of

1. the statement `Begin[msct]`, in accordance with the standard convention for beginning a package,
2. a date and time stamp,
3. a mechanically constructed index to the scripts, that lists the identifiers that begin all the `SetDelayed` (*i.e.* `:=`) in the package, with numbers that show their respective positions in the package,
4. the usage statements for the names that are used externally,
5. the body of the package and the closing `End` statements,
6. the statement `listExecutables` that constructs a list which is essential to the control mechanism described in §6.2.

A comment containing the sequence number appears above each `SetDelayed` statement. We first wrote a file that did not contain items 2 and 3 and did not contain the sequence numbers. These are incorporated, if absent, and updated, if present, by the procedure `indexTheScripts[`$oldName, newName$`]`.

7 Future developments

We will be glad to cooperate in the application and, if necessary, extension and adaptation of MATHSCOUT in further work on computational chemistry and other fields.