THE BILO 4.2.1 DISTRIBUTION FILE

Michael P. Barnett
Department of Computer and Information Science
Brooklyn College of the City University of New York
Brooklyn, NY 11210
Internet:barnett@its.brooklyn.cuny.edu

Kevin R. Perry
Interactive Computer Graphics Laboratory
Computer and Information Technology
Princeton University, Princeton, NJ 08540
Internet:perry@princeton.edu

November 7, 1994

1. Introduction

The `bilo` (bracketed identifier/localized operator) library of MATHEMATICA[1] procedures:

1. supports several derivation and proof idioms of conventional applied mathematics, and
2. controls the structure and arrangement of mechanically generated formulas, precisely.

The file `bilo.4.2.1.tar` is the distribution file that contains version 4.2 of the `bilo` "package" (in the MATHEMATICA meaning of the word) and version 1 of the associated documentation. The only difference from version 4.1 is in the shortening of some file names to meet mathsource conventions. All non-tar files mentioned in this report are included in the distribution file. Further portions of the library are being streamlined for inclusion in later versions of the package and the distribution file. The current version supports:

1. implicit rule formation, described in the published paper [1], and
2. hierarchical addressing, described in the published paper [2] and the report [3].

Offprints of [1] and [2] are available, and `had1.tex` is the AMSTEX file of [3]. Some further features of `bilo` are described in [4], which deals primarily with the production of TEX output by procedures that will be incorporated in later versions of the package.

To see `bilo` usage on-line, load the `autorun` interpreter in the current distribution by:

```
<<autorun.m
```

and run the demonstration/validation files — these have the extension `test`. Thus, to see the examples in [1] executed, enter the command:

```
autorun[firsym.test]
```

[1] Mathematica is a trademark of Wolfram Research Inc.

This simulates an interactive session which executes the successive statements in the control file `firsym.test`. Section 2 describes `autorun`. As regards the major areas of `bilo` activity:

- Rule formation is demonstrated by the files:

  `rule.test`, `firsym.test`, `trig.test`

  The file `rule.test` forms rules of all the types supported by `bilo`. The file `firsym.test` derives all the displays in [1], and `trig.test` derives, by elementary methods, all the identities given in [5] for sin, cos and tan, which are free of Σ and Π expressions.

- Hierarchical addressing is demonstrated by the files:

  `had1.test`, `had2.test`, `tgt1.test`,
  `tgt2.test`, `pred.test`, `root.test`

  The files `had2.test` and `had1.test` cover the examples in [2] and [3], respectively. The other files just mentioned go through the addressing functions more extensively.

- Algebraic operations are demonstrated by the files:

  `fact.test`, `dist.test`, `coll.test`

- Sorting and skeletalizing, and operations on indexed sums, products and other aggregates are demonstrated, respectively, by:

  `ssk.test`, `agg.test`

Section 3 discusses the use of these files for validation.

Additional `bilo` material has been developed and will be incorporated in future distribution files. The three parts of the version number (4.2.1 in the present instance) refer to the overall system structure, to local revisions of coding, and to the associated documentation, respectively.

Version 4.2 of the package consists of:

1. the procedure `bilo.m` — this loads the subsidiary files, (items 2–9 in this list),
2. `bi.root` — some short procedures used in several other modules,
3. `bi.rule` — supports material described in [1],
4. `bi.target` — supports the hierarchical addressing, sorting and skeletalizing, see [2–4],
5. `bi.predicates` — supports the predicates described in [2] and [3],
6. `bi.algebra` — supports factoring, distribution and collection, and operations on sums and other aggregates,
7. `bi.alias` — defines large numbers of aliases of functions, described in [2] and [3], that increase readability of input and output,
8. `u.alias` — `usage` statements to export alias names,
9. `u.primary` — `usage` statements for all the other names to be exported.

The `bi.alias` file is constructed mechanically from a list of the forms of alias and a list of the fragments to be incorporated. The construction of the `usage` files has been mechanized, too.

Further modules have been debugged and used, outside the "package" environment, to deal with topics that include:

1. elementary functions and special functions of mathematical physics,
2. indefinite and definite integrals,
3. total and partial derivatives and derivative matrices,
4. logic and non-standard algebras,
5. matrices and determinants,
6. conversion to TEX.

These will be incorporated in future versions of the package. Applications that used these modules and the earlier versions of the library include work on Legendre functions [6], robot kinematics [7], molecular integrals, based on methods introduced in [8], and the electronic structure of helium and two-electron atomic ions, based on [9]. The control files and output files for the work on Legendre functions, robotics and helium are available. These files demonstrate a variety of practical considerations that led to numerous features of `bilo`.

2. The `autorun` interpreter

A preliminary account of `autorun` was given in [10]. Version 4 is provided in the present distribution. `autorun` reads and interprets a file of MATHEMATICA statements. Each is displayed alongside a tag, which has the default form:

`In[`$m.n$`]:=`

where m is the number of the line in which `autorun` was invoked in the interactive session, and n is the sequence number of the statement in the control file. The result of evaluating the statement is displayed alongside a tag, which has the default form:

`Out[`$m.n$`]=`

The control file can contain two other kinds of item. MATHEMATICA statements that follow a # on column 1 of the input record are executed but, by default, they are not displayed. We call such statements "directives". Also, a body of text is transmitted to the output display by either:

1. sandwiching it between `#startText` and `#endText` directives, or
2. beginning each record of the text segment with an * symbol.

Four pre- and postprocessing functions, that default to `Identity`, provide considerable control.

1. `ipc` — this converts input statements into a preferred form that is recorded,
2. `ivc` — this converts input statements from a variant notation into valid MATHEMATICA,
3. `opc` — this acts on the immediate result of MATHEMATICA evaluation to give output of preferred appearance,
4. `tpc` — this operates on each block of text before it is displayed.

Depending on the options that have been selected (see below), `autorun` displays one or more of the following expressions, for each input statement:

1. the "input as read",
2. the "restyled input", produced by `ipc`,
3. the "input to MATHEMATICA", produced by `ivc`,
4. the "output from MATHEMATICA", produced from 3,
5. the "restyled output", produced by `opc`.

These are displayed alongside tags with the default forms:

`In[`$m.n$`]:=` `in[`$m.n$`]:=` `IN[`$m.n$`]:=` `Out[`$m.n$`]=` `out[`$m.n$`]=`

These tags can be altered, too, or suppressed.

Usually, a file name appears as the only, or first, argument in an `autorun` invocation. The name may be in quotes, or it may be an expression which gives the file name when it has been converted to a string and any internal spaces elided. For example, `autorun[oink . oink]` gives the same output as `autorun["oink.oink"]`. Within a control file, each MATHEMATICA statement is ended by a nul or blank record or by the end of the file. So is each directive. Also, if the nth element of a list g evaluates to the string representation of the nth record of a file f, then `autorun[`g`, ...]` has the same effect as `autorun[`f`, ...]`. An `autorun` file can contain statements that invoke `autorun`. The statement `autobreak[]` in a control file sends control back to the interactive session, from any level of invocation.

Numerous options can be specified by statements that follow the file name or list in the invoking expression. Each can be an option name followed by `=` or `:=` and the option value. Also, the statements may invoke procedures and read files that make one or more such assignments. MATHEMATICA allows such action in general — default values are assigned within `autorun` to each option name *opt* for which `Not[ValueQ @@ Hold[`*opt*`]]` is `True`. Options can be changed dynamically by assignment statements in the control file and by statements, in the file, that invoke procedures or read files containing such assignments.

The present version of `autorun` supports the following options:

- `displayInputAsRead` — defaults to `all`, alternative is `none`.
- `displayRestyledInput` — defaults to `ifDifferent` (i.e. if restyled input is not the same as input as read), alternatives are `all` and `none`.
- `displayInputToMath` — same pattern.
- `displayOutputFromMath` — defaults to `nonNull`, alternative is `none`.
- `displayRestyledOutput` — defaults to `ifDifferent`, (i.e. if restyled output is not the same as output from MATHEMATICA), alternatives are `all` and `none`.
- `ipc, ivc, opc, mpc` — these are the pre- and postprocessors described above. They default to `Identity`. Values assigned to these are function heads.
- `prefixIAR` — precedes the line number in the tag for the input as read. Default is "`In[`".
- `suffixIAR` — follows the line number in the tag for the input as read. Default is "`]:= `".

- `prefixRSI`, `prefixITM`, `prefixOFM`, `prefixRSO` — strings corresponding to `prefixIAR`, with default containing `in,` `IN,` `Out,` `out`, as described above.
- `suffixRSI`, `suffixITM`, `suffixOFM`, `suffixRSO` — strings corresponding to `suffixIAR`.
- `internalLineNumber` — defaults to initial 0, incremented by 1 whenever a statement in the control file is evaluated.
- `externalLineNumber` — defaults to `$Line` (line number of MATHEMATICA session).
- `range` — defaults to `{1, EndOfFile}` — restricts interpretation to the designated part of the file or list.

3. The .test control files

These help to revalidate `bilo` after changes to the coding, as well as to explain usage. To reduce the time needed to inspect the output, each MATHEMATICA statement in several files either:

1. gives `Null` output,
2. is a `SameQ` expression of the form

$$expressionToBeEvaluated \quad \texttt{===} \quad previouslyDeterminedCheckValue$$

The user just has to check that each non-null output expression is `True`. This is automated by putting the statements:

```
checkList = {}; opc = AppendTo[checkList, #]&
```

at the start of the file, and:

```
{Union[checkList] === {Null, True}, Plus /@ checkList}
```

at the end. These evaluate, respectively, to `True` and an expression of the form:

$$i \texttt{ Null + } j \texttt{ True}$$

thereby providing a very rapid visual check. If errors have occurred, the relevant statement numbers are given by:

```
Complement[
 Range[Length[checkList]],
 Position[checkList, #, 1]& /@ {True, Null} // Flatten]
```

The technique is extended in other files. In `trig.test`, the first part of the file constructs and displays entities which are assigned to names of the form `eqn[`*id*`]`. The second part of the file consists entirely of statements of the form:

$$\texttt{eqn[}id\texttt{] ===} \; previouslyDeterminedCheckValue$$

The `checkList` tactic is applied just to this second part.

In `had1.test`, each statement either:

1. gives `Null` output,
2. is a `SameQ` expression, as shown above,
3. is of the form `show[`*expressionToBeDisplayed*`]`.

The statements that display expressions within the head `show` are included to demonstrate formatting and to permit step by step inspection of derivations by users who wish to do this. The end result of each such derivation is checked by a statement that should return the value `True`.

In `had1.test`, the user can inspect the formatted expressions as just described. Also, the `FullForm` version of each `show` expression was cut and pasted electronically from a session in which the appearance was checked carefully, and is reloaded in a directive of the form:

`# pfm = ` *validatedFullFormExpression*

Then the formatted version is checked against this in a `===` statement. The `checkList` tactic is expanded by the initial assignment:

```
opc =
 If[Head[#] === show,
     AppendTo[checkList, shown]; #[[1]],
     AppendTo[checkList, #]; #]&;
```

and the test at the end contains `{Null, True, shown}`, instead of just `{Null, True}`.

The overall process is streamlined by wrapping `autorun` in:

`autolog[`*out*, in_1, …, in_k, {*opts*}`]`

This writes the file *out* with the output of `autorun[`*cat*, *opts*`]`, where *cat* is the concatenation of in_1, …, in_k. It is wrapped in:

`autocheck[`*v*`]` $\Rightarrow$ `autolog[`*v*`.out, startTest, `*v*`.test, endTest]`

where `startTest` contains statements to initialize `checkList` and `opc`, and `endTest` contains statements to display the information that `checkList` builds up.

The 15 test files are run in succession by invoking the shellscript `testBilo` from the UNIX command line. This consists of the single statement:

```
math -run "<<runThrough"
```

The file `runThrough` loads `autorun.m` and `bilo.m`, maps `autocheck` onto the list of rootnames of the `.test` files, and then summarizes the check data in a file `summary.out`. The 16 `.out` files produced in the validation that was run prior to construction of the `tar` file were renamed `.chk` to avoid overwriting when the process is repeated by users of the distribution file.

4. List of files

The `tar file` contains:

1. `bilo.m`, `bi.root`, `bi.rule`, `bi.target`, `bi.predicates`, `bi.algebra`, `bi.alias`, `u.primary`, `u.alias` — these constitute the `bilo` package, as stated in Section 1.

2. `trig.test`, `firsym.test`, `had1.test`, `had2.test`, `rule.test`, `tgt1.test`, `tgt2.test`, `pred.test`, `root.test`, `fact.test`, `dist.test`, `coll.test`, `agg.test`, `ssk.test` — these are the test files explained in Section 1.

3. the corresponding `.chk` files, formed by renaming the `.out` files produced in the validation run, and `summary.chk` produced as explained at the end of Section 3.

4. `had2.extra` and `had1.extra` — these contain a few procedures, needed by `had2.test` and `had1.test`, that are in `bilo` modules to be included in later versions of the package, but are incidental to the implicit rule formation and hierarchical addressing features.

5. `autorun.m` — the `autorun` package, version 4.

6. `testBilo`, `runThrough`, `startTest` and `endTest` — these streamline the demonstration and validation of the package as described above.

7. `ld421.tex` and `ld421.ps` — the LaTeX and PostScript[1] files of this account of the distribution file.

8. `had1.tex` and `had1.ps` — the AMSTeX and PostScript files of [3].

9. `README.4.2.1` — the ASCII "readme" file.

5. Changes relative to distribution 4.1.1

Some file names and the bibliography have been changed slightly. All names that began `bilo.`, except `bilo.m`, have been changed to begin `bi`. The usage files have been renamed to begin with `u`. The `.test` and `.extra` files have been renamed to make the first 5 letters unique, and the `.tex` and `.ps` changed for consistency. Two previously "in-press" citations have been specified.

Bibliography

1. M. P. Barnett, Implicit rule formation in symbolic computation, *Computers Math. Applic.* **26** 35–50, 1993.

2. M. P. Barnett and K. R. Perry, Hierarchical addressing in symbolic computation, *Computers Math. Applic.* **28** 17–35, 1994.

3. M. P. Barnett and K. R. Perry, Hierarchical addressing in `bilo`, this distribution: files `had1.tex` and `had1.ps`.

4. M. P. Barnett and K. R. Perry, Symbolic calculation for electronic publishing, *TUGboat* **15** (3) 285–292 Nov. 1994.

5. I. S. Gradshteyn and I. M. Rhyzhik, *Tables of Integrals, Series and Products*, (tr. and ed. by A. Jeffrey), Academic Press, New York, 1980.

6. M. P. Barnett. Summing $P_n(\cos\theta)/p(n)$ for certain polynomials $p(n)$ *Computers Math. Applic.* **21** 79–86, 1991.

7. Chen, Zhan-zhan, *Symbolic calculation of inverse kinematics of robot manipulators.* M.A. Thesis, Brooklyn College of the City University of New York, 1991.

8. M. P. Barnett and C. A. Coulson, Evaluation of integrals occurring in the theory of molecular structure, *Phil.Trans.Roy.Soc. A*, **243**, 221, 1951. 1951.

9. C. L. Pekeris, Ground state of two-electron atoms, *Phys. Rev.* **112** 1649–1658, 1958.

10. M. P. Barnett, Some simple ways to construct and to use formulas mechanically, *ACM SIGSAM Bulletin*, **28**(2), 21–29, 1991.

[1] PostScript is a registered trademark of Adobe Systems, Inc.