The HPCVL Personalized Test Creator: A User's Guide

Gang Liu ⟨*gang.liu@queensu.ca*⟩ and
Hartmut Schmider ⟨*hartmut.schmider@queensu.ca*⟩
High Performance Computing Virtual Laboratory,
Queen's University, Kingston, Ontario, Canada K7L 3N6
http://www.hpcvl.org

April 24, 2008

Abstract

This User's Guide explains the usage of the HPCVL **P**ersonalized **T**est **C**reator (PTC), a tool designed to produce randomized test sets in a University setting. The tool employs the LaTeX typesetting format to produce the testset, as well as solution sheets and marking sheets. Correct answers are based on User-supplied `FORTRAN` routines.

Contents

Chapter 1

Introduction

1.1 Personalized or Randomized Tests

When preparing a university exam or test, many professors are faced with the problem of minimizing the possibility of fraud. With the availability of electronic communication devices, this task has become even more difficult. One possibility to make exam fraud less likely is to provide each student with an individual set of questions. However, this is hard to do if many students take the test, and the number of equivalent test questions is limited. In many cases it is still possible to modify a few "parameters" from one test to the other:

1. Internal data may be chosen at random, leading to different results from the same basic calculations.
2. The order of questions may be changed from test to test to make communication about the results more difficult.
3. In multiple-choice questions, both the order of possible answers, and the identity of incorrect answers may be changed for each test.
4. A number of random questions may be chosen out of a pool of questions that can be considered of equal "value".

It is the task of the present program to facilitate the automatic generation of randomized test sets from a template, based on the above list of modifications.

Note that other applications of this program are conceivable. For instance, exercise questions may be programmed once and can then be re-used with different random values.

1.2 LaTeX and FORTRAN

The PTC uses as a format for both the input and output the LaTeX typesetting language which is available for virtually all computer platforms, and is completely in the public

domain. We have chosen this format for its universal cross-platform compatibility, and because of its widespread use among physical and mathematical scientists.

As it is necessary to perform some computations to relate the correct answers to randomized questions, a user-defined routine has to be supplied. This should be programmed in `FORTRAN`, a programming language that was chosen because of its simplicity, its ready availability on all platforms, and its focus on mathematical computations. In future versions of the `PTC`, we will consider other programming languages, such as C/C++, or Java.

Chapter 2

The Personalized Test Creator

The PTC is a simple `FORTRAN` program that converts an input LaTeX file containing test questions into multiple output files that contain test sheets, solution sheets, and marking sheets. The LaTeX input file with the original question sets contains additional PTC specific constructs that allow the randomization of test contents, such as the input quantities and the ordering of the answers in multiple-choice questions. The correct answers can be automatically computed using a user-defined `FORTRAN` subroutine with a fixed interface. This scheme allows the automatic generation of randomized tests with the proper answer and marking sheets from a single original file.

We suggest to check the example files that come with the `PTC` package while reading this manual.

2.1 Compilation/Installation

The PTC is a very simple (albeit somewhat longish) `FORTRAN` program that is supplied as source code. In a separate `FORTRAN` file, the subroutine `solve_the_question()` is supplied in a separate file. Compilation should be possible with any standard-compliant `FORTRAN90` compiler. A simple command lines such as

```
f90 -o PTC.exe PTC-1.f90 solve.f90
```

should suffice.

2.2 How To Use It

To use the PTC, the following procedure needs to be followed:

1. Edit the required input files. These are
   - `ini.dat` for general initialization,

- `exam_in.tex` which contains the main test content, including PTC constructs for randomization, and
- `solve_the_question()` a `FORTRAN` subroutine that is used to compute the correct answers from the randomized input data.

2. Re-compile (or re-link) the main executable to include the edited `solve_the_question()`.
3. Run the resulting PTC executable to generate multiple LaTeX output files (extension `.tex`).
4. Compile the required output files with LaTeX and print them out.

2.3 Input Files

The most important input file for `PTC` is `exam_in.tex`, which acts as a template for the randomized tests. This file is written by the user in "annotated" LaTeX format. The idea is that a test file is written pretty much in the same manner as it would be if only a single version of the test for all students was required. The resulting LaTeX file is then augmented by `PTC` constructs that all take the form `PTC-KEYWORD(ARG1;ARG2;...)`, where `KEYWORD` determines the nature of the construct, and `ARG1...` are arguments that contain necessary information that is interpreted by `PTC`. All arguments are separated by semicolons (;). The `exam_in.tex` input file is scanned by `PTC` for these constructs, and the contained information is used to "pre-process" the main body of the file. Details will be explained in later sections.

Another required input file is `ini.dat`, which contains basic information about the program run. Details are discussed below.

To enable the `PTC` program to obtain the correct results corresponding to a specific randomized test, it is normally necessary to specify the proper calculations that need to be performed. This is done in the form of a *user-defined subroutine* (in `FORTRAN`). This routine uses a specified interface that allows to supply variables and constants that appear in a question (both randomized or not), as input arguments, and return calculated results (intermediate of final) as output.

2.4 Output Files

The `PTC` is designed to produce files in LaTeX format as output. These are:

1. `exams.tex` contains the test sheets, i.e. all randomized versions of the exam, including only the questions. These are generated from `exam_in.tex`.
2. `answer.tex` contains answer sheets that may be used for marking if no detailed treatment is needed, i.e. only the correct results are included.

3. `solution.tex` contains more detailed solutions for each question, as well as discussion and answers, generated separately for each randomized test.
4. `shortest.answer.tex` contains a "bare minimum" answer with only the numbers, letters, etc. in the answer. This may be useful as a quick reference for marking.
5. `answer.and.solution.tex` is the combination of `answer.tex` and `solution.tex`, question-by-question and test-by-test.
6. `journal.tex` contains a complete combination of all output, including the test questions, one-by-one and grouped for each question in each test.

It must be pointed out that the bulk of any of this output has to be supplied by the user. The `PTC` program only supplies a means to systematically randomize "input data", i.e. data that appear in the questions, and compute the correct answers based on them. In other words, `PTC` does not generate tests, but rather modifies a user supplied test. The same holds for the solutions to the test questions.
All output pages - test sets, answer sheets, solutions - are numbered according to the formula

```
Page# = 1000 * Paper# + (Local Page# from 1)
```

and carry the current date. For easy regeneration, the seed value, initial and final test number are printed at the end of each test, answer, and solution.

2.5 Some Conventions and Warnings

All `PTC` constructs begin with `PTC-`, which is therefore a prefix that should not be used in another context.

The general form of a `PTC` construct is `PTC-KEYWORD(ARG1;ARG2;...)`, where `KEYWORD` determines the nature of the construct, and `ARG1...` are arguments that contain necessary information that is interpreted by `PTC`. All arguments are separated by semicolons (;).

Parenthesis *must be used pairwise* (matching) if they appear in a string argument. Semicolons (;) *must not be used* in an argument, as they serve as separators throughout. String arguments of `PTC` constructs do *not require quotes*, however they are still required in the `FORTRAN` code discussed in Section 4.

It is recommended to *avoid placing* `PTC` *constructs in a line after a % sign.* This is because `PTC` will process the construct, and place the output of this processing in the spot where the original construct appeared. Therefore, the % sign may "mask" part of this output and the latter may be interpreted by LaTeX as a mere comment. One consequence of this is that `PTC` counts up internal counters without the user's knowledge as the corresponding output does not appear. Similar caution should be exercised with % signs that may appear in an argument of a `PTC` construct.

It is better to place `PTC` constructs in a standalone line, although line breaks are allowed. If line breaks become necessary, it is best to break between arguments, i.e. after a ";" separator to make the code more readable. Note that `PTC` keywords such as `PTC-ANSWER` must **not** be split.

Chapter 3

Preparing Input Files

3.1 `ini.dat`

The `ini.dat` file contains *four lines* with basic information about the test generation. These are

1. The number of the first test paper set to be generated, n_i.
2. The number of the last test paper set, n_f. A total of $n_f - n_i + 1$ papers will be generated, numbered from n_i to n_f.
3. The maximum marks for the test in `real` format. This is required to determine the "scale" of the marking.
4. A random integer acting as a seed. Different runs of the program with the same seed integer will yield the same result if n_i and n_f are kept constant. This enables the exact regeneration of a test with the same input files. If a *different* random set with the same questions is desired, a different seed number can be chosen.

3.2 `exam_in.tex`

This file contains the bulk of the test in LaTeX format. Since most of the content of this file will be copied without alteration into the output files - which are also in the LaTeX format - any LaTeX construct that is supported on the local machine can be used in this file. The only difference from a standard LaTeX file is that `PTC` specific constructs are used to denote the following components of the test:

- The title and tail sections of the exam, using the `PTC-EXAM-TITLE` and `PTC-EXAM-TAIL` constructs;
- Question sections as indicated by `PTC-QUESTION`;
- Constructs for specific subsets of output files: `PTC-ABSTRACT`, `PTC-ANSWER`, and `PTC-SOLUTION`;

- Input variables that can be randomized use constructs such as `PTC-III`, `PTC-REAL-MATRIX`, etc., depending on their type;
- Output constructs: `PTC-PRINT-IN` and `PTC-PRINT-OUT`;
- Multiple-choice questions can be constructed by `PTC-MULTICHOICE-AUTO` and `PTC-MULTICHOICE-USER`;
- Questions based on matching items use the `PTC-COLUMN-MATCH` structure;
- The `PTC-CHOOSE` construct is used to pick questions out of a pool, and/or let the student choose further.

Note that any LaTeX code outside of these constructs is ignored if not specified otherwise. Additionally, the `PTC-SOLUTION` and `PTC-ANSWER` constructs can be used to supply code that appears only in the solution and answer output files, respectively. Finally, with the `PTC-ABSTRACT` construct, code that appears in all output files (except `exams.tex` and `shortest.answer.tex`), can be included. In turn, we will discuss each of these constructs separately.

3.2.1 PTC-EXAM-TITLE and PTC-EXAM-TAIL

The title section of the exam may be placed between the `PTC-EXAM-TITLE` and `PTC-EXAM-TITLE-END` constructs. In principle, any LaTeX code enclosed by these is copied unaltered to the output *at the beginning of each exam sheet*, i.e. multiple times. Anything in the input file that appears **before** `PTC-EXAM-TITLE` is considered LaTeX preamble, and is copied *once* to the header of the output files. If there are multiple `PTC-EXAM-TITLE` constructs in the input file, only the first one is used, all others are Ignored.

`PTC-EXAM-TAIL` and `PTC-EXAM-TAIL-END` are treated analogously: the tail section of the exam should be placed inside these constructs, and is copied unaltered to the output *at the end of each exam sheet*, i.e. multiple times. Anything in the input file that appears **after** `PTC-EXAM-TAIL` is considered LaTeX postfix, and is copied *once* to the bottom of the output files. If there are multiple `PTC-EXAM-TAIL` constructs in the input file, only the last one is used, all others are ignored.

It is important to note that no formatting is applied to the entries after `PTC-EXAM-TITLE` and `PTC-EXAM-TAIL`. Instead it is left to the user to include the desired LaTeX formatting commands.

3.2.2 PTC-QUESTION

The `PTC-QUESTION` and `PTC-QUESTION-END` constructs are used to enclose an exam question. This structure can be used both stand-alone, or within `PTC-CHOOSE`. The `PTC-QUESTION` structure uses four arguments, which follow the keyword, enclosed in parenthesis, and separated by semicolons:

`PTC-QUESTION(QID,MARKS,PRI,SW)`

The meaning of the arguments is as follows:

- `QID` is of type *Integer* and serves as a question ID for internal reference to that question. `QID` must be a unique non-zero positive number. Questions with zero or negative numbers will be ignored. One way to retain the question ID and cause PTC to ignore a question is to preceed `QID` by a minus sign.
- `MARKS` is of type *Real* and denotes the total marks obtainable for the correct answering of the question. This number will be printed out on the test sheets after it was scaled to the total number of marks specified in `ini.dat`.
- `PRI` is of type *Integer* and gives the "priority" of the question. This number is used to determine the order in which questions appear in the test. Questions appear in order from lower to higher priority value. The order of questions with the same priority will be randomized. The actual value of the priority is irrelevant, only the relative size matters. We suggest to only use positive priorities.
- `SW` is of type *Integer* and is used as a "switch" to enable choosing between several versions of the same question. The versions may then be specified using the `PTC-SWITCH-ON` and `PTC-SWITCH-OFF` constructs (see below). If no `PTC-SWITCH-ON` and `PTC-SWITCH-OFF` structures are issued, the value of `SW` is ignored, and this feature is disabled.

The `PTC-SWITCH-ON` and `PTC-SWITCH-OFF` constructs are used within the `PTC-QUESTION` environment. Multiple sets of these may appear. Any text appearing before the first `PTC-SWITCH-ON` line, after the last `PTC-SWITCH-OFF` line, or between a `PTC-SWITCH-OFF` and a `PTC-SWITCH-ON` line, will be included in all versions of a question. Any text appearing after a `PTC-SWITCH-ON` line will be specific for one version of the question. `PTC-SWITCH-ON` takes one *Integer* type argument that denotes the version number of the question. If this version number is equal to the `SW` argument of the current `PTC-QUESTION` environment, the text is printed, otherwise it is ignored. For instance:

```
PTC-QUESTION(12;20.0;5;2)
         {... Block A ...}
      PTC-SWITCH-ON(1)
         {... Block B ...}
      PTC-SWITCH-ON(2)
         {... Block C ...}
      PTC-SWITCH-ON(3)
         {... Block D ...}
      PTC-SWITCH-OFF
         {... Block E ...}
      PTC-SWITCH-ON(1)
         {... Block F ...}
```

```
      PTC-SWITCH-ON(2)
         {... Block G ...}
      PTC-SWITCH-ON(3)
         {... Block H ...}
      PTC-SWITCH-OFF
         {... Block I ...}
PTC-QUESTION-END
```

causes Blocks A, C, E, G, and I to be used on the question sheet, whereas the same sequence with `SW=3` would use A, D, E, H, and I. If the switches are meant to be ignored in this question, `SW=0` will skip all switch sections, and only A, E, and I are used.

3.2.3 PTC-ABSTRACT, PTC-ANSWER, and PTC-SOLUTION

To augment each question with various items and comments that do not appear on the exam sheets, but are printed out in answer and solution files, we supply these three constructs which are to be used *exclusively within a question*. They are optional and it is up to the user how to use them.

The

```
PTC-ANSWER
   ...
PTC-ANSWER-END
```

combination encloses LaTeX code that is used in the answer sheets and therefore appears in `answer.tex`, `shortest.answer.tex`, `answer.and.solution.tex` and `journal.tex`. This is mostly used for printing out the correct answers without details. The text will not appear in `solution.tex`.

The

```
PTC-SOLUTION
   ...
PTC-SOLUTION-END
```

combination encloses LaTeX code that is used in the solution sheets and therefore appears in `solution.tex`, `answer.and.solution.tex` and `journal.tex`. Note that this is separate from the answer sheets although some content might well be similar or repeated. Typically, the solution sheets contain a detailed description on how to compute the answers to the exam questions. The text will not appear in `answer.tex`.

The

```
PTC-ABSTRACT
   ...
PTC-ABSTRACT-END
```

combination encloses LaTeX code that is usually used as an abstract in both the answer and the solution sheets and therefore appears in `answer.tex`, `solution.tex`, `answer.and.solution.tex` and `journal.tex`.

3.2.4 Input Random Variables

An important component in the generation of personalized tests is the use of random variables. The `PTC` provides a simple interface for such variables, and internally keeps track of their use. The general form of a random variable is `PTC-KEY(ARGUMENTS)`, where `KEY` denotes a keyword specifying the type of random variable desired, and `ARGUMENTS` stands for a list of arguments separated by semicolons. The following table contains a list of keywords, together with an explanation of the required arguments. Arguments such as

KEY	**Type**	**Arguments**
`III`	Integer	triple integers
`RRR`	Real	triple reals
`CCC`	Complex	two triple reals, for real and imaginary parts
`SSS`	String	list of strings, to be chosen from
`LLL`	Logical	two strings, standing for "true" and "false", respectively
`VVV`	Vector	three triple reals, for each component of the vector
`INTEGER-MATRIX`	Matrix of Integers	character to denote brackets *(optional)*, two integers for matrix dimensions, triple integers for elements
`REAL-MATRIX`	Matrix of Reals	character to denote brackets *(optional)*, two integers for matrix dimensions, triple reals for elements
`COMPLEX-MATRIX`	Matrix of Complex	character to denote brackets *(optional)*, two integers for matrix dimensions, two triple reals for real and imaginary parts of elements, respectively
`STRING-MATRIX`	Matrix of Strings	character to denote brackets *(optional)*, two integers for matrix dimensions, list of string choices for elements

"triple integers" denote the upper and lower limit of an interval and the discrete step size to get from one to the other. For instance, the triple real `PTC-RRR`$(2.0, 7.0, 2.5)$ means a random choice from the set $\{2.0, 4.5, 7.0\}$, whereas the triple integer `PTC-III`$(4, 11, 2)$ is a random member of $\{4, 6, 8, 10\}$. Note that for real, complex, and vector types, the accuracy with which the step size is specified will carry over as the accuracy of the choices generated.

For instance, if the step size is specified as 0.10 an accuracy of 0.01 will be retained, while a step size of 0.8 will result in an accuracy of 0.1. The distinction between `FORTRAN real` and `double precision` is not made in the `PTC`, i.e. all real numbers are considered to be 8-byte or "double precision". Note that a step size of zero is acceptable. If such is specified, it is interpreted as meaning that only the *lower boundary* value is generated, with the accuracy with which that value was specified. For instance `PTC-RRR`$(5.93, 15.0, 0.000)$ will return 5.93, ignoring the number of figures in the specification of "zero".

String arguments do not require "quotes", and often consist of full-fledged LaTeX code, including white space and special symbols. There is only two restriction on these strings: They should not contain an semicolons (;) as the latter are used as separators, and parenthesis () can only appear in pairs. For strings, an additional construct `PTC-NEW-SSS` is supplied. This can be used to create a new string that has not been picked from a previously supplied list. The argument for this function is just the *Input Index* (see explanation below) of the random variable created before. For example, the call sequence `PTC-SSS(A;B;C) PTC-NEW-SSS(1) PTC-NEW-SSS(1)` creates a random permutation of the string `ABC`, assuming that the `PTC-SSS` call creates the first random variable in the current question.

All input constructs accept an additional, optional argument which has to come last in the list. If this argument is `PTC-HIDE`, then the random number or string is generated, but not printed out. All other aspects of the construct remain the same. This means it can be used as an input variable in the `solve_the_question()` subroutine (see Section 4), but it will not appear in the question where the construct was invoked.

The handling of logical arguments is also somewhat special. The `PTC-LLL` construct expects always two strings because for a logical variable there is only two choices, corresponding to "true" and "false". However, in `PTC` these can be chosen arbitrarily, for instance as *This statement is correct* and *This statement is blatant and deliberate lie.*

The *optional* character in the arguments for the matrix environment is used to determine the type of brackets that are used to enclose the matrix. Since there are an arbitrary number of possibilities, a few simple combinations were chosen, and omission of the character argument (and the separating semicolon) produces a simple kernel for the matrix environment. This kernel is of the form `a&b\\c&d` in the case of a 2×2 matrix, and is used in all of the bracket environments. We denote it as `Mkernel` in the following table which shows the bracket environments for each of the character arguments. Note that any matrix constructs need to be used in *math mode*, as they use environments that are only defined there. *This is left to the user.* To use these matrix environments, `usepackage{amsmath}` is required.

Any randomized variables generated by using one of the above constructs will be internally kept in a module that is later used by the subroutine `solve_the_question()` which will be

Character	**Environment**	Example
'a' or 'A'	\left(\begin{array}{ccc...} Mkernel \end{array}\right)	$\left(\begin{array}{cc} a & b \\ c & d \end{array}\right)$
'm' or 'M'	\begin{matrix} Mkernel \end{matrix}	$\begin{matrix} a & b \\ c & d \end{matrix}$
'p' or 'P'	\begin{pmatrix} Mkernel \end{pmatrix}	$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$
'b'	\begin{bmatrix} Mkernel \end{bmatrix}	$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$
'B'	\begin{Bmatrix} Mkernel \end{Bmatrix}	$\begin{Bmatrix} a & b \\ c & d \end{Bmatrix}$
'v'	\begin{vmatrix} Mkernel \end{vmatrix}	$\begin{vmatrix} a & b \\ c & d \end{vmatrix}$
'V'	\begin{Vmatrix} Mkernel \end{Vmatrix}	$\begin{Vmatrix} a & b \\ c & d \end{Vmatrix}$
none	Mkernel	n/a

discussed below.

It is important to note that any usage of a randomized-variable construct will generate a new entry into this module, and an internal counter called *Input Index* in the following, will be counted up from 1. The Input Index can later be used to refer to this specific variable, for instance using the PTC-PRINT-IN construct to print it (next section). Associated with this index is the type, value and accuracy of that variable. Vectors and matrices are considered to be single variables. All of these are kept in a module. The user must keep track which variable is referred to by which index, especially when coding the user-supplied subroutine solve_the_question().

Any reference to variables generated within a PTC-QUESTION environment must happen within the same environment, since with a new question the Input Index is reset to 1 and all the variables generated are discarded. This means that answers and solutions have to be coded within this environment as well, as they make reference to randomized variables via calls to solve_the_question(). For details, refer to later sections of this manual.

3.2.5 PTC-PRINT-IN and PTC-PRINT-OUT

After a randomized variable has been generated, it can be referred to on future occasions by use of the PTC-PRINT-IN construct. This construct takes only one argument, namely the *Input Index*. PTC-PRINT-IN will cause the LaTeX code for the construct to be replaced by the random variable associated with the Input Index. If appropriate, PTC will only insert parenthesis and commas (for vectors and complex numbers), but no other extra characters.

If the data that need to be printed out are of type `logical`, the situation is slightly different. As explained in Section 3.2.4, the generation of a logical random variable requires the specification of two strings standing for "true" of "false". If such a random variable is retrieved and printed out via `PTC-PRINT-IN` then a second (optional) argument can be used to create different outcomes. If this second argument is omitted, the chosen value is printed. If the second argument is `.TRUE.` then the value corresponding to "true" is printed. Likewise, if it is `.FALSE.`, then the value corresponding to "false" is printed. Finally, if the second argument is `.NOT.`, then the logical negation of the random variable is printed.

The `PTC-PRINT-OUT` construct works on a similar basis. This is used to replace the construct by calculated values that depend on randomized variables and that have been produced by calls to the user-defined subroutine `solve_the_question()`. To indicate which value is desired, it uses an *Output Index* that is assigned inside `PTC-PRINT-OUT`. However, since it is re-computed for each question, `PTC-PRINT-OUT` must still be used in the same `PTC-QUESTION` environment that belongs to a specific question. `PTC-PRINT-OUT` uses additional arguments to specify the accuracy of entries in a specific variable. For instance, if the entry with Output Index = 5 is a vector, the call to `PTC-PRINT-OUT` requires 4 integer arguments, the first one being 5, the other three corresponding to the accuracy of each of the vector components: `PTC-PRINT-OUT(5,2,2,2)` will produce something that looks like "`(1.0,3.2,-1.6)`". If Output Index = 7 is associated with a 2×2 real matrix, only one accuracy argument is required, and `PTC-PRINT-OUT(7,3)` will yield a result like "`1.00&3.21\\-1.62&2.57`" which has the form of `Mkernel` above and can be used in LaTeX expressions.

The LaTeX segment "`\[\begin{vmatrix}PTC-PRINT-OUT(7,3)\end{vmatrix}\]`" yields

$$\begin{vmatrix} 1.00 & 3.21 \\ -1.62 & 2.57 \end{vmatrix}$$

As a rule, one accuracy argument is required for each "triple reals" argument in the above table explaining the usage of the generation of randomized variables of the same type.

3.2.6 `PTC-MULTICHOICE-AUTO` and `PTC-MULTICHOICE-USER`

Multiple-choice questions can be generated with the `PTC-MULTICHOICE-AUTO` and `PTC-MULTICHOICE-USER` environments. These are used inside a `PTC-QUESTION` construct. Any text preceding or following these constructs is just printed out as is, as part of the question. The constructs themselves are then used to generate multiple choices. The difference between the two is that `PTC-MULTICHOICE-AUTO` is used to generate multiple choices automatically, usually differing by a variable, while `PTC-MULTICHOICE-USER` leaves the generation of multiple choices to the user.

PTC-MULTICHOICE-AUTO

PTC-MULTICHOICE-AUTO employs the PTC-CHOICES construct to generate multiple answers to a question that differ by the values of variables. The following is a prototype example for the use of PTC-MULTICHOICE-AUTO in the case where there is only a single real value to be handled:

```
PTC-QUESTION()
         {... Question ...}
      PTC-MULTICHOICE-AUTO(NC;NONE)
         {... Prefix ...}
      PTC-CHOICES(IANS;IACC;PROB;XMIN;XMAX;YMIN;YMAX)
         {... Postfix ...}
      PTC-MULTICHOICE-AUTO-END
        {... Question tail ...}
PTC-QUESTION-END
```

This will print out `{... Question ...}` as the question, followed by a number of possible answers that all have the form `{...Prefix...} Value {...Postfix...}`, where the `Value` is determined by the arguments of `PTC-CHOICES`. The `{... Question tail...}` part follows this list of possible answers.

The meaning of the arguments in `PTC-MULTICHOICE-AUTO` is as follows:

- `NC` is an integer that indicates the number of possible answers or choices.
- `NONE` is an *optional* string that will be inserted *as is* in place of the last of the possible answers. It can be used to insert answers such as *"None of the above"*, and can include LaTeXconstructs. The inclusion of this will eliminate one of the other possible answers, possibly the correct one.

The meaning of the arguments in `PTC-CHOICES` is as follows:

- `IANS` is an integer "Output Index", i.e. it indicates an element in the output arrays in the `BASE_DATA` module of `PTC-1.f90` which contains the correct answer to this question. This number is determined by the user when coding `solve_the_question()`. See Section 4 below for details.
- `IACC` is the number of significant figures to appear in the output of the numerical answer component associated with this call. This needs only to be specified if the base type of the generated variable is `real`. It is omitted if it is of type `integer`, `logical`, or `string`.
- `PROB` is the probability that the resulting *wrong* value in an answer falls into the lower of two intervals. See details below. This needs only to be specified if the base type of the generated variable is `real` or `integer`. It is omitted if it is of type `logical` or `string`.

- `XMIN;XMAX;YMIN;YMAX` define the boundaries of two nested intervals that are placed around the correct value. See details below. This needs only to be specified if the base type of the generated variable is `real` or `integer`. It is omitted if it is of type `logical` or `string`.

The `PTC-CHOICES` call also serves to automatically generate *wrong answers* from the correct one. To this end, the user has to specify two intervals of values, an inner and and an outer one. This is done by `XMIN;XMAX;YMIN;YMAX`. `XMIN` and `XMAX` define the outer interval, and `YMIN` and `YMAX` define the inner interval. By default these are scale factors, i.e. the correct value `C` is multiplied by them, and we have

$$[XMIN \times C, YMIN \times C] < C < [YMAX \times C, XMAX \times C],$$

i.e. the outer and inner interval produce an interval below and an interval above `C`. The argument `PROB` is the probability that a wrong values is chosen from the one below. Naturally, the probability for the other interval is `1-PROB`. In total, `NC-1` wrong values are generated. If a wrong value has the same character presentation ("look") as the correct one or a previously generated wrong one, it is discarded and another attempt is made. If, for a given interval, the values are specified in the wrong order, it will be reversed automatically.

It is also possible to specify the intervals around the true value `C` in terms of absolute rather than relative values. If this is desired, a keyword may be placed in front of the argument. This keyword may either be `ADD` if the value needs to be added as is to the true value, or `RAW` if the value is to be used directly without any addition or scaling. For instance, let us assume the true value is `C=1.5`. Then the interval arguments `-1.0;3.0;0.0;2.0` will produce a low interval of $[-1.5, 0.0]$ and a high one of $[3.0, 4.5]$. The same intervals could be created by specifying `ADD -3.0;ADD 3.0;ADD -1.5;ADD 1.5`, or `RAW -1.5;RAW 4.5;RAW 0.0;RAW 3.0`, or any combination thereof. Note that these are real values, so the decimal point is mandatory, i.e. it must be `3.0` or `3.`, but not `3` without decimal point.

`PTC-CHOICES` may be called multiple times. The maximum number of answers being generated depends on the number of calls made to `PTC-CHOICES`. Here are the possible cases:

- `PTC-CHOICES` is invoked only once or twice, as in the above example. If possible, `NC` answers are generated in total, including the correct one. `NC` should not be chosen larger than 10.
- It is invoked more than twice, say n times. The maximum number is 2^n, as the multiple calls are considered to be components of the answer, and 2 choices are made for each (the correct one and a random incorrect one). All permutations of the chosen components are then generated and used as a possible answer (including the correct one). `NC` must not exceed 2^n in this case.
- If a call to `PTC-CHOICES` involves multi-component values such as vectors or complex numbers, then each of the components counts like a separate call. For instance,

generating a complex number is handled the same way as generating 2 real numbers, with a maximum supported NC of 10.

If NC exceeds the maximum values stated above, the execution will stop.

If the second, optional argument of PTC-MULTICHOICE-AUTO is specified, it is taken as an answer with the meaning "*None of the above*" and placed last in the list of multiple-choice answers. The program then eliminates a random answer, including the correct one from the list.

In the above example PTC-CHOICES is called only once to generate simple scalar values. If the requested values are for vectors, three components need to be produced for each wrong answer. Therefore,IACC;PROB;XMIN;XMAX;YMIN;YMAX all need to be specified three times, once for each component, i.e. for a vector the total number of arguments is $1+3\times 6 = 19$. For instance, let us assume that the correct answer for output index 5 is a vector (1.5,1.5,1.5). Then the argument list

```
5;
2;0.5;-1.0;3.0;0.0;2.0;
2;0.5;-1.0;3.0;0.0;2.0;
2;0.5;-1.0;3.0;0.0;2.0
```

produces vectors with components that each lie within the intervals $[-1.5, 0.0]$ or $[3.0, 4.5]$, with even probability, and are printed out to 2 significant figures, such as (-1.1,-0.53,3.9).

Similarly, the arguments have to be specified twice if the generated value is complex, once for the real and once for the imaginary part of the complex number, leading to a total of 13 arguments. In other words, complex numbers are treated like 2-component vectors.

PTC-MULTICHOICE-USER

PTC-MULTICHOICE-USER leaves the generation of answers to a multiple-choice question to the user. Each possible answer is specified using the PTC-CHOICE construct. A typical form for this usage is:

```
PTC-QUESTION()
      {... Question ...}
      PTC-MULTICHOICE-USER(NONE)
      PTC-CHOICE(0)
       {... Wrong Answer ...}
      PTC-CHOICE(1)
       {... Correct Answer ...}
         ...
      PTC-CHOICE(0)
       {... Wrong Answer ...}
```

```
      PTC-MULTICHOICE-USER-END
      {... Question tail ...}
PTC-QUESTION-END
```

PTC-MULTICHOICE-USER has only one *optional* argument, namely the string NONE. This is used in the same manner as specified above for the second argument of PTC-MULTICHOICE-AUTO. Each use of PTC-CHOICE is followed by the text {...Answer...} of one possible answer. The only argument PTC-CHOICE is an integer that is 1 if the answer is the correct one, and 0 otherwise. Note that multiple correct choices are acceptable. PTC will generate the choices in a random order.

3.2.7 PTC-COLUMN-MATCH

Another form of question is "column matching". In this form, the student is given two columns of labeled items, and needs to match the labels of one column with the entries of the other. This is equivalent to re-ordering one column to match with the other. This is done by entering the labels of one column in "matching boxes" by the side of the other. For this, the PTC-COLUMN-MATCH construct allows the definition of two pools of entries, one for the left and the other for the right column. The PTC then chooses at random a given number of entries out of these pools to create a column matching test question. Here is the prototypical form of this construct:

```
PTC-COLUMN-MATCH(NE;SIDE)
   PTC-COLUMN-LEFT(IL1) left-entry-1
   PTC-COLUMN-LEFT(IL2) left-entry-2
      {...a total of m left-column entries...}
   PTC-COLUMN-LEFT(ILM) left-entry-m
   PTC-COLUMN-RIGHT(IR1) right-entry-1
   PTC-COLUMN-RIGHT(IR2) right-entry-2
      {...a total of n right-column entries...}
   PTC-COLUMN-RIGHT(IRN) right-entry-n
PTC-COLUMN-MATCH-END
```

Items that are to appear in the left-hand column are entered using the PTC-COLUMN-LEFT construct, and items for the right-hand column are entered through PTC-COLUMN-RIGHT. An item is a piece of LaTeXcode that follows directly after these constructs, and includes everything before the next construct that begins with PTC-COLUMN. The only argument of PTC-COLUMN-LEFT and PTC-COLUMN-RIGHT are integers that are used to match left- and right-hand column entries: entries with the same integer argument are considered a match, all others are not. The whole list is enclosed in a PTC-COLUMN-MATCH ... PTC-COLUMN-MATCH-END pair. Note that the numbers of left- and right-hand column entries in the pool (m and n above) do not have to be the same. However, it is important to make sure that there is enough entries in the pool to create the requested number of column items, and that matches receive the same integer argument. Entries that do not

have corresponding entries in the other column are considered to be matched correctly only if the corresponding box in the test is left blank, i.e. no match. Multiple matches are also supported.

PTC-COLUMN-MATCH itself uses two arguments:

- NE is the number of entries appearing in the columns of the test, i.e. the number of rows in both columns.
- SIDE is a keyword that determines where the "matching boxes" are to appear in the test. This can be one of LEFT, RIGHT, or BOTH. The value of this keyword has no impact on other aspects of the test.

The NE entries will be chosen at random from the pool in a fashion that maximizes the number of matches. Specifically, the system chooses at random entries for one column, and then - also at random - picks entries from the available matches in the other, giving preference to matching ones.

3.2.8 PTC-PRINT-AUTO-ANSWER

Multiple-choice questions and column-matching questions can be constructed using PTC-MULTICHOICE-AUTO, PTC-MULTICHOICE-USER, and PTC-COLUMN-MATCH constructs. These forms of questions have in common that they require the user to tell the system the correct answer. It is therefore not necessary to produce an entry into the PTC-ANSWER section manually. This is in fact done automatically by the system for these type of questions. However, manual entry is still required for standard questions, for instance "fill in the blank" or "compute the value". If both types of questions appear within a larger framework (for instance as part of a multi-part question), it is necessary to mark the answer section where to place the manually entered answer with respect to the automatically answered one.

For this purpose, the PTC-PRINT-AUTO-ANSWER construct is supplied. It is intended to be used in the following way:

```
PTC-ANSWER
   PTC-PRINT-AUTO-ANSWER(1)
   PTC-PRINT-AUTO-ANSWER(2)
   {...manual entry of answer to ‘‘compute-the-value’’ type question...}
   PTC-PRINT-AUTO-ANSWER(3)
PTC-ANSWER-END
```

In this example, we are handling the answers to a "four-part" question where the first, second, and fourth parts are of the multiple-choice or column-matching type, and therefore are answered automatically by the PTC system. The third question however requires the computation of a value and is not multiple-choice. It requires manual entry. The calls

to PTC-PRINT-AUTO-ANSWER are used to establish where the manual entry (or entries) are placed with respect to the automatic ones by labeling each automatic entry by one call. PTC assigns to each of the questions it can answer automatically an ordering number in order of appearance, in the above example from 1 to 3. This ordering number from 1 is the only argument required for PTC-PRINT-AUTO-ANSWER.

Note that PTC-PRINT-AUTO-ANSWER is only needed if there is several "parts" to a question, some of which are answered automatically and some not. If all are answered automatically, the system outputs the answers in sequence.

3.2.9 PTC-CHOOSE and PTC-QUESTION-TITLE

In some cases, several questions may be grouped together, and a random selection is made from the pool. For this purpose, the PTC-CHOOSE construct is supplied. This construct is meant to include the initial list of questions, and is often used in combination with PTC-QUESTION-TITLE. Here is a prototype of its use:

```
PTC-CHOOSE (NQ;NS;MARKS;PRI)

    PTC-QUESTION-TITLE
       {...Header common to all questions...}
    PTC-QUESTION-TITLE-END

    PTC-QUESTION(QID1;MARKS1;PRI1;SW1)
        {...First question...}
    PTC-QUESTION-ENDS
         {...A total of N question entries...}
    PTC-QUESTION(QIDN;MARKSN;PRIN;SWN)
        {...Last question...}
    PTC-QUESTION-END

PTC-CHOOSE-END
```

In this example, N questions appear between PTC-CHOOSE and PTC-CHOOSE-END. These make up the total pool of questions availailable. The arguments in the individual questions are of course the same as outlined in section 3.2.2. PTC-QUESTION-TITLE encloses general information that appears on the exam sheet before any of the questions, for instance comments that are relevant for all questions, or an explanation of how many questions to choose. PTC-CHOOSE requires four (4) arguments. These are:

- NQ denotes the number of questions that appear in the exam, chosen at random from the original pool of N. NQ must be positive, otherwise the whole structure is ignored.
- NS is the number of questions that the student needs to pick for answering, chosen from the list of NQ appearing in the exam.

- `MARKS` is the (real) maximum number of marks given for each of the correctly answered question. This has to be common for all questions in the pool and will *overwrite* the individual marks specified in the questions (`MARKS1...MARKSN`).
- `PRI` is the common priority for the questions in the pool. This has to be common for all questions in the pool and will *overwrite* the individual priority values specified in the questions (`PRI1...PRIN`).

The idea here is to include a total number of `N` questions in this construct, then specify a number `NQ`$\leq$`N` of questions that will be randomly selected by the `PTC` to appear on each of the exam sheets. Then, there is the further opportunity to let the student pick at will a number `NS`$\leq$`NQ` from those for answering. Of course, if the students are required to answer all questions, one will choose `NS`=`NQ`, and if the whole question pool needs to appear in each test, one will choose `NQ`=`N`. Due to the large degree of flexibility in this construct, all questions have to be equivalent as far as marking and priority is concerned. Therefore these values appear as arguments in `PTC-CHOOSE` and overwrite the corresponding values in the individual questions.

Chapter 4

The solve_the_question() Subroutine

4.1 General Structure and Output Index

The FORTRAN routine solve_the_question() is used to compute numerical values that depend on the randomized variables appearing in the test questions. These may either be intermediate values or parts of the solution or answer to the question. A template for the routine (ptc-user.f90), as well as an example routine (ptc-user-example.f90) are supplied with the original source code of PTC to illustrate how this user-defined subroutine should be structured. The interfacing of the routine with the main body of PTC is done via a standard argument list, and with a module BASE_DATA that contains variables that have been generated by calls to the randomized-variable constructs discussed above, as well as results from the computations that are done in solve_the_question(). The routine solve_the_question()is not called directly by the user, but is used by PTC to generate data that are needed for questions and answers. The most common access is from PTC-PRINT-OUT.
The header part of the user-supplied routine is fixed and should not be altered:

```
SUBROUTINE SOLVE_THE_QUESTION(PRINT_OUT_NUMBER, AN_ACCESSORY_INTEGER)
  USE BASE_DATA
  USE RAND_INTERFACE
  USE CONSTANT_INTERFACE
  IMPLICIT NONE
  INTEGER  ::  PRINT_OUT_NUMBER, AN_ACCESSORY_INTEGER
  INTEGER  ::  I, J, K, L, M, N
  QUESTION_IDENTIFIER=QUESTION_POSITION(0,QUESTION_NUMBER)
  SELECT CASE (QUESTION_IDENTIFIER)
  CASE(:0)
    PRINT*,' Illegal Question ID: ', QUESTION_IDENTIFIER, QUESTION_NUMBER
    STOP
```

This serves to include the proper modules (such as BASE_DATA which contains most of the needed inputs and outputs of the routine). The SELECT CASE (QUESTION_IDENTIFIER)

statement directs the routine to the proper question when it is called. The content of the CASE(ID) statements that follow this header are the parts that need to be programmed by the user. Each question is handled separately and uses the same basic structures. Whenever solve_the_question() is called, one and only one of these CASE blocks is executed. Obviously, the user needs to be aware which question corresponds to which question identifier, as specified in an argument of the PTC-QUESTION construct.

An important quantity in the computations done in solve_the_question() is the *Output Index*. This is an integer chosen by the user for each of the quantities computed within a question. This index is used to label each of the computed quantities, to access information associated with them, and to print it out using PTC-PRINT-OUT. It therefore has to be unique. Often, it is best to chose consecutive numbers starting with 1, but any positive choice is allowed.

4.2 Simple Variables

To explain the use of some of the structures in BASE_DATA, it is best to look at a simple example. Let's assume the question with QID=5 supplies the students with a Force vector $\vec{F}$ and a mass m and expects them to compute the acceleration $\vec{a} = \vec{F}/m$. The components of $\vec{F}$ were generated in the question by means of a PTC-VVV construct which returns a random 3-component vector and keeps its elements in an array GENERATED_V(3,*) from where they are available for usage in solve_the_question(). Similarly, the mass is a randomized real variable that was generated using PTC-RRR and ends up in an array GENERATED_R(*). Since the Input Index (discussed above) is counted up by one every time one of these constructs is called, the elements of $\vec{F}$ reside in GENERATED_V(1:3,1) and m is in GENERATED_R(2) for this question. We are computing only one output quantity, namely the acceleration, so let us choose the Output Index to be 1. With this information, we are ready to code CASE(5):

```
CASE(5)
   F=GENERATED_V(:,1)
   M=GENERATED_R(2)
   A=F/M
   CALCULATED_V(:,1)=A
   CALCULATED_TYPE(1)=V_TYPE
   CALCULATED_NUMBER=1
```

Of course we need to declare REAL*8 M,F(3),A(3) in the header if we want to use those quantities. This is done in the example for clarity. Direct computation is shorter and requires less additional local variables, but is less readable:

```
CASE(5)
```

```
CALCULATED_V(:,1)=GENERATED_V(:,1)/GENERATED_R(2)
CALCULATED_TYPE(1)=V_TYPE
CALCULATED_NUMBER=1
```

4.3 Data Types

As in the above example, the array `CALCULATED_TYPE(:)` is used to declare the proper type for each output variable, whereas `CALCULATED_NUMBER` needs to be set to the maximum value of the Output Index for a given question, proper number of calculated entries for this case or question. The (pre-declared) possible types for `CALCULATED_TYPE(:)` are listed in the table below.

Meaning	**Name of Constant**	**Value**
Integer	`I_TYPE`	1
Real	`R_TYPE`	2
Complex	`C_TYPE`	3
String	`S_TYPE`	4
Logical	`L_TYPE`	5
Vector	`V_TYPE`	6
Integer Matrix	`I_MATRIX_TYPE`	101
Real Matrix	`R_MATRIX_TYPE`	102
Complex Matrix	`C_MATRIX_TYPE`	103
String Matrix	`S_MATRIX_TYPE`	104

For the simple data types (`I_TYPE, R_TYPE, and C_TYPE`), arrays are provided in the `BASE_DATA` module that are used to contain calculated values of the corresponding type. These are called `CALCULATED_I`, `CALCULATED_R`, and `CALCULATED_C`, respectively. Note that for each Output Index, only one element of these is used, depending on `CALCULATED_TYPE`. The others are allocated but unused. For quantities of type `V_TYPE`, a two-dimensional array `CALCULATED_V(3,*)` is supplied, in which the first index is used to denote the three vector components. Otherwise, it works the same as for the simpler data types.

4.4 Strings

For the case of strings (i.e. `CALCULATED_TYPE(?)=S_TYPE`) a more elaborate scheme has to be employed. Since strings can be very long, it is too inefficient to allocate space for them for each Output Index even if they are not used. Therefore, instead of providing a "`CALCULATED_S`" array directly to contain the string, we use the "`CALCULATED_I`" array element to contain an integer "address" for the desired string, which is stored in a common character array `OUTPUT_SSS`. Here is an example:

```
CASE(6)
   CALCULATED_TYPE(1)=S_TYPE
   OUTPUT_SSS(2)='Ottawa'
   CALCULATED_I(1)=2
   CALCULATED_NUMBER=1
```

In this case the string "Ottawa" will be printed when `PTC-PRINT-OUT(1)` is invoked. We have used the "address" `2` to store this string, but could have used any other, as long as it is not used otherwise.

4.5 Logical Variables

If the data type to be computed is `LOGICAL` we use a combination of the simple scheme that applies to reals and integers, and of the string-handling method explained above. This is because logical variables have a value (`.TRUE.` or `.FALSE.`), but are also associated with two strings as, for example, specified in `PTC-LLL` (see Section 3.2.4). In the following example, we make it dependent on a variable `SMOKES` (assumed to be between 0 and 1) whether a person is a smoker (`.TRUE.`) or a non-smoker (`.FALSE.`):

```
CASE(7)
   CALCULATED_TYPE(1)=L_TYPE
   OUTPUT_SSS(2)='Smoker'
   OUPTUT_SSS(3)='Non-Smoker'
   CALCULATED_POSITION(T_POST,1)=2
   CALCULATED_POSITION(F_POST,1)=3
   IF (SMOKES.GT.0.5) THEN
    CALCULATED_L(1)=.TRUE.
   ELSE
    CALCULATED_L(1)=.FALSE.
   ENDIF
   CALCULATED_NUMBER=1
```

Here, we use the array `CALCULATED_L` to store the truth value of the variable with Output Index `1`. But we also store the strings that will be printed out when `PTC-PRINT-OUT(1)` is invoked, namely in the string array `OUTPUT_SSS`. To keep track of the array elements for these strings, we need an auxiliary two-dimensional integer array `CALCULATED_POSITION`. The first index of this array is a "position" that distinguishes between the two strings in question, in our case the ones corresponding to `.TRUE.` and `.FALSE.`, respectively. The second index is just the Output Index, which is `1` here. Note that for the handling of logical variables, the pre-defined constants `T_POST` and `F_POST` are supplied. The array `CALCULATED_POSITION` can also be used in other cases where multiple strings have to be handled and kept track of, from "addresses" `CALCULATED_POSITION(1,?)` to `CALCULATED_POSITION(2,?)`.

4.6 Matrices

For the case of the matrix data types I_MATRIX_TYPE, R_MATRIX_TYPE, C_MATRIX_TYPE, and S_MATRIX_TYPE the situation is a little more complicated. For these, the array CALCULATED_M is used, which is of type MATRIX_TYPE. This type is declared in the BASE_DATA module, and it takes the following form:

```
TYPE MATRIX_TYPES
     INTEGER :: SIZES(2)
     INTEGER :: BRACE_INDEX
     INTEGER, DIMENSION(:,:), POINTER :: INTEGER_M
     DOUBLE PRECISION, DIMENSION(:,:), POINTER :: REAL_M
     DOUBLE COMPLEX  , DIMENSION(:,:), POINTER :: COMPLEX_M
     CHARACTER(LEN=MAX_CHARS), DIMENSION(:,:), POINTER :: STRING_M
END TYPE MATRIX_TYPES
```

The array SIZES is meant to contain the shape of the matrix, i.e. the number of rows and columns, respectively. This can be done directly, for instance for a 2×3 matrix that happens to be the 5th data entry that is calculated:

```
CALCULATED_M(5)%SIZES(1)=2
CALCULATED_M(5)%SIZES(2)=3
```

It is then also necessary to allocate one of the corresponding sub-types (INTEGER_M, REAL_M, COMPLEX_M, or STRING_M) before using this data type. Deallocation is later done automatically by the code. Let's assume the above 2×3 matrix is complex:

```
CALCULATED_TYPE(5)=C_MATRIX_TYPE
ALLOCATE(CALCULATED_M(5)%COMPLEX_M( &
   CALCULATED_M(5)%SIZES(1),        &
   CALCULATED_M(5)%SIZES(2))
```

Here, we have also set CALCULATED_TYPE to the proper value. After this was done, we can directly assign the elements of the complex matrix, for instance inside a double loop:

```
DO J=1,3; DO I=1,2
   CALCULATED_M(5)%COMPLEX_M(I,J)= ...
END DO; END DO
```

where ... indicates some expression, a function call or similar.
Finally, we need to specify the type of brackets that are to be used to enclose the matrix when it is printed out. This is done by specifying the sub-value BRACE_INDEX:

```
CALCULATED_M(5)%BRACE_INDEX=5
```

where we have chosen the round brackets from the amsmath package. For this assignment, the user can either use the values in the following table, or the corresponding names of predefined constants. The value of 0 (or MATRIX_DATA_ONLY) is the default, i.e. if BRACE_INDEX is not set, no brackets will be used.

BRACE_INDEX	**Name of constant**	Example
0	MATRIX_DATA_ONLY	$\begin{matrix} a & b \\ c & d \end{matrix}$
1	MATRIX_BRACE_A	$\left(\begin{matrix} a & b \\ c & d \end{matrix}\right)$
3	MATRIX_BRACE_M	$\begin{matrix} a & b \\ c & d \end{matrix}$
5	MATRIX_BRACE_P	$\left(\begin{matrix} a & b \\ c & d \end{matrix}\right)$
7	MATRIX_BRACE_LOWER_B	$\left[\begin{matrix} a & b \\ c & d \end{matrix}\right.$
8	MATRIX_BRACE_UPPER_B	$\left\{\begin{matrix} a & b \\ c & d \end{matrix}\right.$
9	MATRIX_BRACE_LOWER_V	$\left\vert\begin{matrix} a & b \\ c & d \end{matrix}\right.$
10	MATRIX_BRACE_UPPER_V	$\left\Vert\begin{matrix} a & b \\ c & d \end{matrix}\right.$

4.7 Some Additional Routines

For the convenience of the user, a few additional routines are supplied and may be called from inside `solve_the_question()`. These are:

- `DOUBLE PRECISION FUNCTION GET_CONSTANT_VALUE_OF (CSTRING)`
  allows access to internal physical constants that are defined in `PTC`. For a list of these, see the table in Section 5.1. The string argument `CSTRING` can be either of the descriptions in column 2 of the table in Section 5.1.3 of this manual, or of the `KEYWORD`s in column 4. Note that the string has to be *exact*, and must be enclosed in double quotes "", unlike in the case of `PTC` constructs.
- `DOUBLE PRECISION FUNCTION MYRAND()`
  returns a random real value on the interval $[0, 1]$.
- `INTEGER FUNCTION AN_INTEGER_RANDOM(L,U)`
  returns an random integer value between from `[L,U]`.
- `DOUBLE PRECISION FUNCTION A_DOUBLE_RANDOM(L,U)`
  returns an random real value between `[L,U]`.

Chapter 5

Auxilliary Constructs

The PTC offers an array of constructs that are not necessarily employed when preparing an input .tex file or coding solve_the_question(). In some situations these constructs allow the user to access internal information or design a test in a more flexible manner. Here is a list of these constructs:

- Constructs dealing with internally defined constants are of the form PTC-CONSTANT-*.
- Parameters that are used internally can be accessed by PTC-PRINT-PARAMETER.

We will discuss these in the following sections.

5.1 Internal Constants and PTC-CONSTANT Constructs

The PTC uses several constants that are set inside of the program. These constants may be accessed via constructs that begin with PTC-CONSTANT. Because some of these constants may be of use in the construction of a test, there is many ways to print them out inside a question or in the header of a test.
A complete table of the constants currently in use can be found on the next page. The first column contains the "Constant Number" NCONST which is used to recover a specific constant through a PTC construct. The list of available constants may easily be extended. All constants are set in the routine set_constants(), and new ones can be added using the current entries as a guideline. The entries of the table below have been generated from the values in the routine. Entries that are required for each constant include name, symbol, value, accuracy, unit, and a unique key for identification. The accuracy with which they are printed here is limiting, i.e. if the user requests greater accuracy the request is ignored.

NCONST	Constant Name	Symbol	Value and Unit
1	Acceleration due to earth's gravity	g	9.80 m/s^2
2	Avogadro's number	N_A	6.0221367×10^{23} mol^{-1}
3	Boltzmann's constant	k	1.380658×10^{-23} J/K
4	Coulomb's constant	k	8.99×10^9N·m^2/C^2
5	Electron charge magnitiude	e	$1.60217733 \times 10^{-19}$C
6	Permeability of free space	μ_0	$1.25663706 \times 10^{-6}$T·m/A
7	Permittivity of free space	ϵ_0	$8.854187817 \times 10^{-12}$C^2/(N·m^2)
8	Pi	π	3.14159265
9	Planck's constant	h	$6.6260755 \times 10^{-34}$ J·s
10	Mass of electron	m_e	$9.1093897 \times 10^{-31}$ kg
11	Mass of neutron	m_n	$1.6749286 \times 10^{-27}$ kg
12	Mass of proton	m_p	$1.6726231 \times 10^{-27}$ kg
13	Speed of light in vacuum	c	299792458. m/s
14	Universal gravitational constant	G	6.67259×10^{-11} N·m^2/kg^2
15	Universal gas constant	R	8.314510 J/(mol·K)

5.1.1 PTC-CONSTANT-WHOLE-TABLE

The PTC-CONSTANT-WHOLE-TABLEconstruct allows the one-command generation of a complete table of all constants used by PTC, similar to the one above. This construct does not require any arguments.

5.1.2 The PTC-CONSTANT-TABLE Constructs

To produce selective tables of some of the constants, the PTC provides the three constructs PTC-CONSTANT-TABLE-HEAD, PTC-CONSTANT-TABLE-LINE, and PTC-CONSTANT-TABLE-TAIL. A simple table can be constructed in the following prototype form:

```
PTC-CONSTANT-TABLE-HEAD
   PTC-CONSTANT-TABLE-LINE (NCONST1;IACC1)
   {...total of N lines with constants...}
   PTC-CONSTANT-TABLE-LINE (NCONSTN;IACCN)
PTC-CONSTANT-TABLE-TAIL
```

PTC-CONSTANT-TABLE-HEAD and PTC-CONSTANT-TABLE-TAIL are producing the header and tail part of a LaTeXtable, respectively, and do not need any arguments. For each constant used, one PTC-CONSTANT-TABLE-LINE call is issued. These have two arguments: the constant number NCONST to identify the constant requested; this may be found in the first column of the table above or in the source code of set_constants(), and the accuracy IACC with which the constant is to be printed. If $1 \leq$IACC$\leq$MACC, where MACC is the maximum accuracy with which the constant is available (as defined in set_constants() and printed in the above table), IACC significant digits are printed. If IACC$\geq$MACC or IACC≤ 0, the system defaults to the maximum value MACC.

5.1.3 Other PTC-CONSTANT Constructs

The form of a table is not the only one in which internal constants may be accessed. To print out constants in various forms, we supply the following constructs:

- PTC-CONSTANT-VALUE supplies *only* the value of a constant (no units!).
- PTC-CONSTANT-VALUE-UNIT prints the value *with* the unit.
- PTC-CONSTANT-EQUATION returns a small equation including the symbol for the constant, for instance "$c = 299792458.m/s$".
- PTC-CONSTANT-NAME-EQUATION produces an equation including both name and symbol for the constant, for example "Speed of light in vacuum $c = 299792458.m/s$".

All of these constructs need two arguments: the integer Constant Number NCONST identifying the constant in question (see table), and the accuracy IACC with which it is to be displayed. These constructs are also supplied in an alternative form, where in the construct name PTC-CONSTANT-* a keyword identifying the constant in question is switched in after PTC-CONSTANT, resulting in a name PTC-CONSTANT-KEYWORD-*. In this form the construct needs only one argument, namely the accuracy IACC, as the identity of the constant is determined in the construct name. Here is a list of the presently used keywords for internal constants:

NCONST	Constant	Symbol	KEYWORD
1	Acceleration due to earth's gravity	g	EARTH-ACCELERATION
2	Avogadro's number	N_A	AVOGADRO-NUMBER
3	Boltzmann's constant	k	BOLTZMANN-CONSTANT
4	Coulomb's constant	k	COULOMB-CONSTANT
5	Electron charge magnitiude	e	ELECTRON-CHARGE
6	Permeability of free space	μ_0	FREE-PERMEABILITY
7	Permittivity of free space	ϵ_0	FREE-PERMITIVITY
8	Pi	π	PI
9	Planck's constant	h	PLANCK-CONSTANT
10	Mass of electron	m_e	ELECTRON-MASS
11	Mass of neutron	m_n	NEUTRON-MASS
12	Mass of proton	m_p	PROTON-MASS
13	Speed of light in vacuum	c	SPEED-OF-LIGHT
14	Universal gravitational constant	G	GRAVITATIONAL-CONSTANT
15	Universal gas constant	R	UNIVERSAL-GAS-CONSTANT

For example, to produce the output "$\pi = 3.14$", we could use PTC-CONSTANT-EQUATION(8,3), or the alternative form PTC-CONSTANT-PI-EQUATION(3); or for "9.1094×10^{-31} kg", both PTC-CONSTANT-VALUE-UNIT(10,5) and PTC-CONSTANT-ELECTRON-MASS-VALUE-UNIT(5) will be suitable.

5.2 Internal Parameters and PTC-PRINT-PARAMETER

The PTC-PRINT-PARAMETER construct is used to print out internal parameters. It has only one argument, namely a keyword to identify the parameter that is to be printed. The following table contains a list of all possible keyword and an explanationd of the associated parameters.

KEYWORD	Parameter
PAPER	Number of the current paper
TOTAL-PAPERS	Total number of papers
QUESTION	Number of the current question
TOTAL-QUESTIONS	Total number of questions to be generated in each paper
SUBQUESTION	Number of the current subquestion within the current question
TOTAL-SUBQUESTIONS	Total number of subquestions to be generated within the current question
TOTAL-REQUIRED-SUBQUESTIONS	Total number of subquestions to be answered by the student within the current question
GLOBAL-SUBQUESTIONS	Total number of (sub)questions to be generated within each paper
GLOBAL-REQUIRED-SUBQUESTIONS	Total number of (sub)questions to be answered by the student within each paper

These parameters may for example be used to label (sub)questions in the testset. The LaTeX code

```
\begin{center}Please look at \large\textbf{Question
   PTC-PRINT-PARAMETER(PAPER).%
   PTC-PRINT-PARAMETER(QUESTION).%
   PTC-PRINT-PARAMETER(SUBQUESTION)
}\normalsize of this paper.\end{center}
```

produces for the 5th subquestion of the 3rd question in paper 17 something that looks like

Please look at **Question 17.3.5** of this paper.

Note that the usage of % signs together with PTC constructs involves some risk (just as their usage does in LaTeX code in general). If a % is followed by a PTC construct, the output of that construct may be "masked" and not appear in the final output text. Nevertheless, as PTC processes all constructs, it may have changed the state of the document, for instance by increasing counters such as Input Indices, a fact that may be hidden from the user. It is left to the user to exercise caution in the usage of % symbols.

5.3 PTC-NOTE

The construct `PTC-NOTE` gives the user the opportunity to insert comments or "notes" between questions in the test. It is meant as a standalone construct that encloses such text:

```
PTC-NOTE(NPRI)
    {...Text of the note...}
PTC-NOTE-END
```

The only argument is a priority number `NPRI` to indicate where the note is to be inserted. It will appear on the test sheets after questions with lower priority values and before questions with equal or higher priority values.

Chapter 6

Help and Support

The PTC is provided and distributed by the High-Performance Virtual Laboratory (HPCVL). It was written and is maintained by Gang Liu. The software may be downloaded at no charge from www.hpcvl.org, and is provided *as is*. This means that neither the author nor HPCVL accept any responsibility for consequences arising out of the usage of this software, especially for damage, be it mental or physical, to your health, hard-drive, possessions, relatives, or pets.
That being said, we are of course happy to obtain feedback from the users of the program, and are always open for suggestions as to how to improve it. We also offer support and help with installing and running the software, as long as it is kept in mind that this is not our day job. Please direct any requests to Gang.Liu@QueensU.CA.
If you have questions and suggestions about the manual, you can also contact Hartmut.Schmider@QueensU.CA who has written most of it.