Classification: Open

The Numerical Objects Report Series

Report 1997:11

Diffpack Tools for Automatic Generation of Result Reports

Are Magnus Bruaset

October 6, 1997

NUMERICAL OBJECTS

This document is classified as Open. All information herein is the property of Numerical Objects AS and should be treated in accordance with the stated classification level. For documents that are not publicly available, explicit permission for redistribution must be collected in writing.

Numerical Objects Report 1997:11

Title

Diffpack Tools for Automatic Generation of Result Reports

Written by

Are Magnus Bruaset

Approved by

Are Magnus Bruaset
October 6, 1997

Numerical Objects, *Diffpack* and *Siscat* and other names of Numerical Objects products referenced herein are trademarks or registered trademarks of Numerical Objects AS, P.O. Box 124, Blindern, N-0314 Oslo, Norway.

Other product and company names mentioned herein may be the trademarks of their respective owners.

Contact information

Phone:	+47 22 06 73 00
Fax:	+47 22 06 73 50
Email:	`info@nobjects.com`
Web:	`http://www.nobjects.com`

Copyright © Numerical Object AS, Oslo, Norway
October 6, 1997

Contents

1 Background **2**

2 A toolbox for implementation of report generators **4**

2.1 The `Reporter` interface 5

2.2 Using `ASCIIReporter` for simple text-based output 14

2.3 Using `LaTeXReporter` for LaTeX typeset output 15

2.4 Using `HTMLReporter` for hypertext output 17

2.5 How to implement new report formats 19

3 A brief look at the Diffpack menu system **22**

3.1 User dependent menu code 22

3.2 Multiple loops 24

4 A simple case study **26**

A Source listing of the test program **31**

A.1 The interface: CDFEM.h 31

A.2 The implementation: CDFEM.C 32

B Input file used by the test program **40**

C Output generated by the test program **42**

C.1 ASCII format 42

C.2 LaTeX format 48

C.3 HTML format 49

D Multi-format report generation **50**

This report should be referenced as shown in the following BibTeX entry:

```
@techreport{NO1997:11,
 author = "Are Magnus Bruaset",
 title = "Diffpack Tools for Automatic Generation of Result Reports",
 institution = "Numerical Objects AS, Oslo, Norway",
 type = "The Numerical Objects Report Series",
 number = "\#{}1997:11",
 year = "October 6, 1997",
}
```

Diffpack Tools for Automatic Generation of Result Reports

Are Magnus Bruaset[1]

[1]Numerical Objects AS, P.O. Box 124 Blindern, N-0314 Oslo, Norway. Email: amb@nobjects.no.

Abstract

Large scale numerical simulations tend to use a wide range of input parameters and produce vast amounts of output data. In this report we present some Diffpack utilities that will assist the developer in designing and implementing automatically generated result reports. To demonstrate the use of such tools, and in particular of the `Reporter` class hierarchy, we end the presentation by a simple case study.

Chapter 1

Background

The accelerating development of high performance computers have caused dramatic changes to the way we approach mathematical models. Today, we are able to do large scale numerical simulations based on models of very high complexity. In particular, we often want to solve realistic problems that involve large sets of input parameters and produce vast amounts of output data. For this reason, we need systematic ways to record and present simulation parameters as well as simulation results. One attractive possibility is to let the simulator generate reports on each run by itself. This approach forces documentation of every single experiment, ensures a uniform style of presentation and eliminates the risk of errors made when tabulating numerical data by hand.

Most database systems provide more or less standardized implementations of report generators and query languages that can be used to extract specified subsets of information from large collections of data. However, there are no de facto utilities for similar report management within the framework of numerical simulations. For some numerical applications the use of a general-purpose database system may be advantageous, but in most cases we can settle with automatically generated documents that combine text, tables and figures. In this note we will discuss how Diffpack applications can produce such reports with a minimum of coding. This is possible by using the `Reporter` class hierarchy, which can be successfully combined with the Diffpack menu system and plotting facilities, see [7, 11, 9]. Moreover, we have the possibility of choosing the actual report format, say as ASCII text or LaTeX typesetting, at run-time without any changes to the application code. The generic `Reporter` interface and certain attributes of some derived classes implementing specific report formats are discussed in Section 2. In Section 3 we take a brief look at the menu system, before concentrating on a particular code example. The case study in Section 4 demonstrates a typ-

ical application that combines the menu system's possibility of automatic parameter variation with the report management offered by class `Reporter`. The source code for this test problem is given in Appendix A together with an input file and the resulting output in Appendices B and C, respectively. Moreover, in Appendix D we briefly introduce the class `MultiReporter` that is designed to handle a collection of different report formats and detail levels.

Chapter 2

A toolbox for implementation of report generators

In this section we will address the use of the C++ class `Reporter` and its derived versions `ASCIIReporter`, `LaTeXReporter` and `HTMLReporter`. The presentation will assume basic knowledge of object-oriented programming in C++, see for instance [12, 3]. Moreover, some familiarity with the coding conventions used in Diffpack will be useful when examining the accompanying examples, see [10, 1, 7].

Technically speaking, `Reporter` is an abstract base class defining the interface used by all derived reporter classes, see Figure 2.1. Each derived class will produce output tailored to a specific format, e.g. as plain ASCII text or as a typeset document involving mathematical symbols and PostScript figures. The latter alternative is implemented in the `LaTeXReporter` class, which produces a report file that can be processed by the document preparation system LaTeX [5]. Such standards for embedding formatting instructions as part of a document are often referred to as a *markup language*. Throughout this note we will employ this term somewhat informally as a synonym to "report format".

As the observant user will notice, the interface offered by `Reporter` is indeed inspired by the structure of LaTeX documents. Nevertheless, it should be a relatively easy task to implement other formats, e.g. `troff`, as separate classes in the `Reporter` hierarchy. For instance, it took less than a day to implement the class `HTMLReporter`. This tool generates documents written in the hypertext markup language HTML, which can be presented by graphical browsers like `Mosaic` [13].

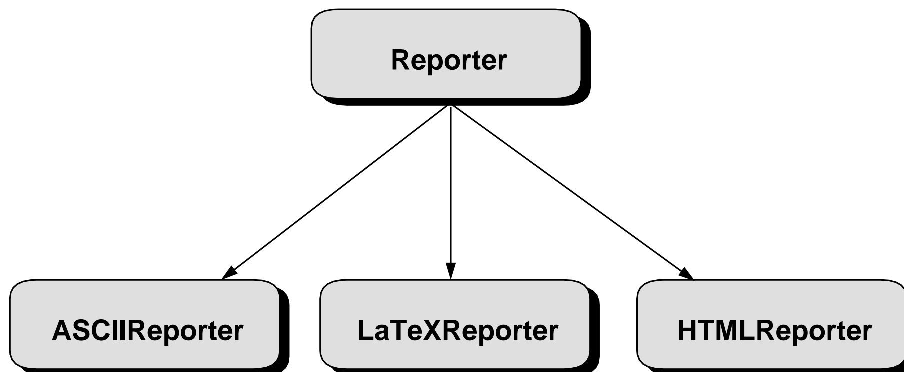

Figure 2.1: *The* `Reporter` *hierarchy.*

2.1 The `Reporter` interface

Most member functions specified as part of the `Reporter` interface will have different implementations depending on the actual document format. These functions are declared to be *virtual*, see for instance the command `section` introduced below. The mechanism of virtual functions let us create an object of the derived type `LaTeXReporter` and refer to it in terms of the abstract data type `Reporter`, still maintaining access to the functionality of the derived class. Since the C++ run-time system knows that this `Reporter` object is actually a `LaTeXReporter`, a call to `section` automatically invokes the LaTeX variant of the function.

Given a text string `myReportFormat` containing a valid class name from the `Reporter` hierarchy, we can make a certain instance `rep` by one of the alternative declarations:

- `Reporter& rep = *createReporter(myReportFormat,filename);`
- `Reporter* rep = createReporter(myReportFormat,filename);`
- `Handle(Reporter) rep = createReporter(myReportFormat,filename);`

As for most Diffpack class hierarchies, the accompanying function `createReporter` allocates a new instance of the class specified by `myReportFormat` and returns a pointer to that object. We will comment on the use of the additional argument `filename` at a later point of this discussion.

Throughout this note we will give examples where the identifier `rep` is assumed to be a handle to a reporter object. Readers that are not familiar with the notion of handles, may think of them as plain C or C++ style pointers[1].

[1] In fact, a Diffpack handle is a plain pointer with reference counting, see [6, 8] for further details.

When `rep` is declared as a handle, the statements `rep->linebreak();` and `rep().linebreak();` are both valid calls of the member function `linebreak`. In contrast to a traditional pointer, the handle `rep` is dereferenced[2] by the function call `rep()`.

Opening and closing reports. Usually, a new report is started by allocating the desired reporter object as indicated above, using the function `createReporter`. The argument `filename` is then given to the new reporter object, which will open the specified file[3]. Once the `Reporter` object is available, the report document is composed by calling an appropriate sequence of member functions. To start with, we will take a look at the functionality that deals with the overall document structure, such as title pages and sectioning commands. The relevant part of class `Reporter`'s interface looks like this:

[2] Given the C++ pointer `Reporter* repPtr`, recall that the actual object referred to by `repPtr` is accessed by the explicit dereference `*repPtr`.

[3] A slightly more general output mode is available by explicitly allocating a reporter for the desired format, say `ASCIIReporter`, using an `Os` object as argument to the constructor. In this way, the generated report may be sent to any output stream that can be represented by class `Os`, not only to disk files.

```
   // Sectioning and other utilities

virtual Reporter& header      (const String& title, const String& author,
                               BooLean fullpage = dpTRUE,
                               BooLean singlefile = dpTRUE);

virtual Reporter& trailer     ();

virtual Reporter& insertDocument (const String& basename) =0;
virtual Reporter& insertVerbatim (const String& file)     =0;

virtual String getSectionRef  () =0;

virtual Reporter& section     (const String& txt) =0;
virtual Reporter& subsection  (const String& txt) =0;
virtual Reporter& pagebreak   ()                  =0;
virtual Reporter& linebreak   ()                  =0;
virtual Reporter& eject       (int n = 1)         =0;
virtual Reporter& drawline    ()                  =0;
        Reporter& flush       ();
```

By specifying the directive “`=0`”, this code listing indicates that some member functions are declared to be *pure virtual*. Since these functions are not implemented, it is impossible to allocate an object of type `Reporter`. Such *abstract base classes* serve as ancestors of derived classes that are forced to supply appropriate code segments for the missing functionality. Consequently, `Reporter` is indeed an abstract entity that provides an entry point to the hierachy of different report formats, thus defining a uniform interface that all derived classes will have to support. Seen in connection to the initial discussion on virtual functions it is then clear why the variable `rep` is represented by reference (`Reporter&`), as a pointer (`Reporter*`), or in terms of a handle (`Handle(Reporter)`).

When a new object is created, the report should first be initialized by calling the member function `header`. This function requires two `String` arguments holding the title and the author's name. Moreover, two optional boolean parameters may be used to (*i*) indicate whether the present report is a stand-alone document or to be included elsewhere, (*ii*) choose between a separate title page or a more compact format. At the end, the report should be explicitly closed by calling the function `trailer`.

Within a document, the functions `section` and `subsection` may be invoked in order to group the contents into sections and subsections, respectively. Both functions accept a `String` argument `txt` that is used to produce the corresponding headings, which are combined with a nested numbering

scheme. The remaining member functions defined in the code segment above are:

- `insertDocument` which inserts a document with appropriate embedded formatting instructions (e.g. LaTeX or PostScript[4] code).
- `insertVerbatim` which inserts a text file "as is". This function is useful when pasting segments of source code into a report.
- `getSectionRef` which returns a string containing the section number set by the last call to `section`.
- `pagebreak` which inserts a page break.
- `linebreak` which inserts a line break.
- `eject` which inserts `n` blank.
- `drawline` which draws a line across the full page width. The behaviour of his function is changed when building tables, see below.
- `flush` which empties the report file buffer.

In addition to these formatting utilities, the function `put` is used to write the actual report text. As we are about to see, the action of `put` depends on the document context.

From the partial interface description given above we observe that all functions that contribute directly to the report have a return value of type `Reporter&`. This property can lead to very compact coding by combining several function calls into a single statement, e.g.

```
rep->header("Sample Report", "Kåre
Dump").section("Introduction").flush();
```

Such contructions can also involve other member functions that will be introduced in the next few pages.

[4]PostScript is a page description language with graphical capabilities that is commonly accepted as an industry standard. Mot software vendors, at least of programs for scientific visualization, have therefore adopted the PostScript language as an output format. In fact, this language is also used as native instruction set for several types of laser printers.

Environment	Description
`Center`	center text
`Tabular`	build a horizontally centered table of text or numbers
`Table`	as `Tabular`, except that a caption and/or label may be added
`Itemize`	build a list of items preceded by a bullet symbol (•)
`Enumerate`	build a list of numbered items
`Figure`	insert a PostScript figure with caption and/or label
`Dispmath`	insert displayed mathematical text

Table 2.1: *Environment types supported by class* `Reporter`

Working with environments. Much inspired by LaTeX, the `Reporter` tool permits the document to be organized into different environments. By the term *environment* we here refer to the group of formatting instructions and text inserted between subsequent calls to `begin`***env*** and `end`***env***, where *env* can be replaced by any of the identifiers given in Table 2.1. The user is free to produce nested environments, e.g. by inserting displayed mathematics as an item in a list of type `Itemize` or `Enumerate`. Except for the `Tabular`, `Table` and `Figure` environments, there are no arguments passed to the corresponding scope delimiters `begin`***env*** and `end`***env***. These exceptions will be explicitly treated later on.

As mentioned above, the function `put` accepts a text string that is inserted into the document. However, this function is sensitive to the current choice of environment. At the initial stage, where no environment has been explicitly invoked, `put` simply writes the given text. In contrast, if `put` is used between calls to `beginItemize` and `endItemize`, it results in a new list item. After the environment is closed, `put` behaves as it did before the call to `beginItemize`. Similar redefinitions of `put`'s action occur inside the other environments as well.

When `put` is called, it receives a string consisting of text and/or hardcoded formatting instructions (e.g. LaTeX commands like `$x_{i}$`). All `put` operations will implicitly call the function `adaptString`, which tries to eliminate any conflicts between the given text and the instruction set used internally by the chosen report format. On some occasions it may be preferable to invoke `insert` rather than `put` in order to circumvent this filter mechanism.

To summarize, let us have a look at the prototypes for `put` and `insert`, as well as a few other functions for text generation:

```
// Resolving conflicts between text and instruction set
```

```
virtual String adaptString (const String& txt);

        // Inserting text

         Reporter& put    (const String& txt);
         Reporter& insert (const String& txt);


        // Shortcuts to center and dispmath environments

virtual Reporter& centerline  (const String& txt) =0;
        Reporter& displaymath (const String& txt);


        // Inline math mode

virtual Reporter& putmathline (const String& txt) =0;


        // Embedding text styles

virtual Reporter& beginGroup (TextStyle style);
virtual Reporter& endGroup   ();

virtual String    getGroup   (const String& txt, TextStyle style);
```

The functions `centerline` and `displaymath` represent compact ways of producing `Center` and `Dispmath` environments, respectively. In addition, there is a specialized version of the `put` operation that generates inline mathematical expressions. This option is typically used as in

```
rep->putmathline("z_{i} = x_{i}^{2}");
```

which results in the expression $z_i = x_i^2$ when `rep` refers to a `LaTeXReporter` object.

It is often useful to change the style of an important text segment in order to make it stand out from its surroundings. Such operations are possible when employing the concept of *text groups*. That is, any text generated between calls to `beginGroup` and `endGroup` will have the style attribute defined by the argument `style`. In some cases it will more convenient to use the function `getGroup` to change the style of a single word or a small text segment. This function does not produce output by itself, it only returns the argument `txt` with embedded formatting instructions corresponding to the parameter `style`. The returned string should then be passed on to a function like `put`. Since the `style` parameter is independent of the underlying markup language, this procedure is transparent with respect to the

`TextStyle`	`ASCIIReporter`	`LaTeXReporter`	`HTMLReporter`
`PLAIN`	sample text	sample text	sample text
`BOLD`	sample text	**sample text**	**sample text**
`ITALICS`	sample text	*sample text*	*sample text*
`TYPEWRITER`	sample text	`sample text`	`sample text`
`MATHTEXT`	z_{i} = x_{i}^{2}	$z_i = x_i^2$	*z_{i} = x_{i}^{2}*

Table 2.2: *The effects of different text styles.*

choice of reporter type. In Table 2.2 we list the style variations presently supported and indicate the corresponding effects for different members of the `Reporter` hierarchy. As seen in this table, grouping commands using the style `MATHTEXT` can be an alternative to `putmathline` and the `Dispmath` environment. Moreover, some of the text styles are not supported by all `Reporter` variants. These limitations are further discussed in the following sections that deal with specific document formats.

Composing tables. When reporting on the execution of a numerical simulation, it is often convenient to present input parameters and vital results in a tabular form. The `Reporter` tool offers two flavours of this option based on the `Tabular` and `Table` environments. As suggested by the interface specified below, these alternatives are almost identical, except that `endTable` accepts a table caption and a label for easy reference. The member functions dealing with tables are as follows:

```
    // For easy formatting of table cells

virtual String cellpad (const String& s) =0;


    // Put tables in a single operation

virtual Reporter& put (ArrayGenSimple(String)& tb);
virtual Reporter& put (ArrayGenSimple(real)& tb);

virtual Reporter& put (ArrayGenSimple(real)& tb,
                       const String& cell_format);

virtual Reporter& put (ArrayGenSimple(real)& tb,
                       ArrayGenSimple(String)& cell_format);
```

```
    // Tabulating environments

virtual Reporter& beginTabular (int ncols, const String& col_format);
virtual Reporter& endTabular   ();

virtual Reporter& beginTable   (int ncols, const String& col_format);
virtual Reporter& endTable     (const String& caption, const String& label);


    // For cross-referencing

virtual String makeLabelRef (const String& label) = 0;
```

When starting a new table, the argument `ncols` is required to specify the number of columns. We may also provide an optional argument `col_format` that determines how each column should be justified: left (`l`), right (`r`) or centered (`c`). If `col_format` is a one-character string, the specified justification is forced on all columns. The default choice is to center columns that are not explicitly given another layout. It should be noted that some report formats support only left justified columns due to restrictions of the underlying markup language.

We have briefly described the function `put` that is used for text insertion. However, when called within the scope of a tabulating environment, this function will produce a new line in the table. It is then the caller's responsibility to format each line, usually employing the utility function `cellpad` to line up the table entries properly. A simpler alternative is to record the table contents in an `ArrayGenSimple` object `tb` capable of holding `ncols` columns with entries of type `String` or `real`. The contents of such arrays can be inserted as table entries in a single call to `put`. In this case we activate overloaded versions of `put` that are tailored for table management. When generating a table with `real` entries in this way, we may include a second argument `cell_format` holding formatting instructions as traditional C or C++ style format strings. If this argument is a simple string, say `"%6.3f"`, it defines the format to be used for all entries in the accompanying array of numbers. More flexibility is provided when `cell_format` is an `ArrayGenSimple(String)` object of format strings. Usually, the `cell_format` array will have the dimensions of `tb`. As alternatives, `cell_format` may consist of a single row or column matching the corresponding dimension of `tb`. The ith format string will then be used for all entries in column i or row i, respectively.

When referring to a table from another document location, it is advisable to use the function `makeLabelRef`. Given the same argument `label` as accepted by `endTable`, this utility returns a valid table reference based on

the cross-referencing capabilities of the current markup language. It is also possible to draw horizontal lines across the full table width by the previously defined function `drawline`.

Importing PostScript figures. Usually, simulation results are best presented in terms of plots and figures. If we generate the result report with help of a `Reporter` derivative capable of handling PostScript code, figures can be inserted in a `Figure` environment:

```
    // Figure handling

virtual Reporter& beginFigure (const String& psfilename, real cmWidth = 10.0);
virtual Reporter& endFigure   (const String& caption, const String& label);

String getCurrPlotDir ();


    // Shortcut to figure environment
Reporter& figure (const String& psfilename, const String& caption,
                  const String& label, real cmWidth = 10.0);
```

The first argument `psfilename` handed to `beginFigure` defines the name of the involved PostScript file, while the optional parameter `cmWidth` decides the figure scaling. If the string `psfilename` contains one or more '`/`' characters, it is regarded as a complete path specification. Otherwise, it is assumed to be the name of a file located in a directory administered by the `Reporter` object itself. If the current report is written to a disk file named `reportfile.`*ext*[5], the directory `reportfile_figures` is automatically created. In this directory there will be a series of subdirectories named `sec1`, `sec2` and so on. In order to save PostScript files in the correct subdirectory at run-time, the member function `getCurrPlotDir` should be called to obtain valid paths. Since `Reporter` objects do not check the existence of the specified files, it is usually not necessary to supply the final figures at run-time. Naturally, this is not true when a run-time presentation of the current report is requested, see below.

Similar to the `Table` environment, the closing delimiter `endFigure` accepts two arguments that are used as figure caption and reference label. Usually, `Figure` environments are generated by a call to `figure`, which combines the actions of `beginFigure` and `endFigure` into one single operation. The `makeLabelRef` utility described for tables may be used in connection

[5]The actual file name extension *ext* will depend on the chosen report format, e.g. `.txt` (ASCII), `.tex` (LaTeX) or `.html` (HTML).

to figures as well. Finally, it should be mentioned that some classes in the `Reporter` hierarchy may lack the possibility of including PostScript figures due to limitations of the underlying markup language. In such cases an appropriate message will be issued.

Several excellent graphing tools are available that can turn simulation data into PostScript figures. In addition to relatively expensive commercial software, there is a considerable number of public domain programs, see [4]. We will not discuss the details of the different alternatives in this note. However, in the context of Diffpack, the combination of class `CurvPlotFile` and the plotting utility `Gnuplot`[6] has proved to be useful, see [9] for further information.

Postprocessing and presentation of reports. Usually, automatically generated reports will be processed separately after termination of the original application. However, certain member functions in class `Reporter` also offers the flexibility of integrated processing and report presentation:

```
        // Postprocessing and presentation of report

    virtual void process (BooLean postscript = dpTRUE) =0;
    virtual void print   (BooLean postscript = dpTRUE) =0;
    virtual void preview (BooLean postscript = dpTRUE) =0;
```

These three member functions issue the relevant Unix commands needed for such purposes. The argument `postscript` indicates whether the generated document includes PostScript segments, thus forcing special constraints on the choice of previewers and printing options. Depending on the chosen report format, the default system calls may be overridden by certain Unix environment variables. This option will be further discussed in the following subsections.

2.2 Using `ASCIIReporter` for simple text-based output

The simplest version of the `Reporter` tool, i.e. the class `ASCIIReporter`, generates plain ASCII text files. Although this format is very restricted in terms of layout and other document attributes, it may be useful for compact

[6] `Gnuplot` is a Diffpack tool built on top of the public domain program `gnuplot`. In contrast to many other pograms, `gnuplot` is capable of producing PostScript output from a data file, without requiring interactive user reponses.

reports with modest requirements to special features like tables and mathematical typesetting. Moreover, since most word processors and editors accept ASCII files, the output from `ASCIIReporter` can serve as a basis for further paste-up in document preparation systems that are not directly supported by the `Reporter` hierarchy.

Limitations. As mentioned above, the ASCII format is relatively restricted with respect to some of the advanced features defines as part of the `Reporter` interface. In particular, there is no support for mathematical expressions. The text written in math mode will be copied as it is passed to `putmathline` or similar commands, typically including LaTeX instructions. Moreover, the different text styles set by the grouping commands `beginGroup`, `endGroup` and `getGroup` are neglected.

When it comes to tabulating environments, all table cells are left justified. Other cell formats are simply ignored. In addition, PostScript figures are not supported. Attempts to include figure files will result in a warning message.

Postprocessing and presentation of reports. Normally, ASCII reports are previewed and printed using standard Unix commands such as `less` and `lp`. The default settings may be overruled by defining certain Unix environment variables. In Table 2.3 we have listed the names of the relevant environment variables together with their default values. The occurrences of the string "`%s`" are simply standard `printf` format masks for insertion of the current report name. For instance, to change the previewer from `less` to `more` we could make the definition

```
setenv ASCIIRep_PREVIEW_TXT_CMD "xterm -e more %s.txt &"
```

on the Unix command line. We also note that the environment variables with names containing the substring "`PS_CMD`" are used only when the `postscript` flag is raised. If the flag is set to false, the relevant variable names are those containing the substring "`TXT_CMD`". In Figure 2.2 we can see the first screen page of a typical ASCII report.

2.3 Using `LaTeXReporter` for LaTeX typeset output

Compared to the simple report format used by class `ASCIIReporter`, considerably more sophisticated documents can be processed when using LaTeX [5]. In fact, all features specified by the `Reporter` interface are fully supported by

Unix environment variable	Default value
ASCIIRep_CLEANUP_CMD	"rm -f %s.ps"
ASCIIRep_PROCESS_TXT_CMD	" "
ASCIIRep_PROCESS_PS_CMD	"mpage -2 -s -A -p -P %s.txt > %s.ps"
ASCIIRep_PRINT_TXT_CMD	"lp -s %s.txt"
ASCIIRep_PRINT_PS_CMD	"lp -s -opost %s.ps"
ASCIIRep_PREVIEW_TXT_CMD	"xterm -e /usr/local/bin/less %s.txt &"
ASCIIRep_PREVIEW_PS_CMD	"ghostview -quiet -a4 -seascape %s.ps &"

Table 2.3: *Environment variables and default settings for* `ASCIIReporter`.

the class `LaTeXReporter`, including typesetting of mathematical expressions, insertion of PostScript figures, different table layouts and various text styles.

Inserting PostScript figures. When `LaTeXReporter` generates an input file for LaTeX, it assumes that the macro package `psfig` is installed. This package, as well as its accompanying driver program `dvips`, is available on most computer systems supporting LaTeX. If you do not have access to these utilities, they can be downloaded from several ftp sites on the Internet. In particular, you can try an anonymous ftp to `ftp.uni-stuttgart.de`.

Any PostScript file that `psfig` finds acceptable[7] can be inserted by a call to the function `figure`. The arguments of this function and their meanings have been discussed in Section 2.1.

Postprocessing and presentation of reports. Generated LaTeX reports are saved in files with extension `.tex`. Such files have to be processed before the final document can be previewed or printed. Given the file `report.tex`, we can enter the command

```
latex report
```

on the Unix command line. This will result in a new file, `result.dvi`, which can be printed by the `dvips` program or some other driver for DVI (DeVice Independent) files. The `dvips` command will also ensure that eventual PostScript figures are correctly inserted in the printed document.

Once we have generated a DVI file, it can be previewed on screen. This is usually achieved by use of the program `xdvi`. However, if the document

[7] The `psfig` macro wants *conforming PostScript* code that satisfies certain conventions on structure. Most mainstream programs capable of producing PostScript output obey these conventions.

Figure 2.2: *A typical* `ASCIIReporter` *screen page previewed by* `less`.

includes PostScript figures, these will not show up in the previewer window. To overcome this problem, the complete document should be translated into PostScript code and loaded by a PostScript previewer like `ghostview`, see Figure 2.3 for a sample screen page. This translation can be done by specifying an output file for `dvips`, i.e. we issue the command

```
dvips -o report.ps report.dvi
```

to generate the PostScript file `report.ps`.

As mentioned before, the document handling discussed in this subsection can also be implicitly performed by calls to member functions in the `LaTeXReporter` class. The default commands issued by these functions may be overruled by defining certain Unix environment variables. In Table 2.4 we have listed the names of the relevant environment variables together with their default values. As before, we note that the environment variables with names containing the substring "`PS_CMD`" are used only when the `postscript` flag is raised. In other cases, the relevant variable names are those containing the substring "`DVI_CMD`".

2.4 Using `HTMLReporter` for hypertext output

The last report format that we will discuss in this note is based on HTML, which is a *hypertext* markup language. In such documents it is possible to link text segments or other entities to certain actions, e.g. a text string

Unix environment variable	Default value
LaTeXRep_CLEANUP_CMD	"rm -f %s.ps"
LaTeXRep_PROCESS_DVI_CMD	"latex %s.tex > /dev/null"
LaTeXRep_PROCESS_PS_CMD	"dvips -q -o %s.ps %s.dvi"
LaTeXRep_PRINT_DVI_CMD	"dvips -q %s.dvi"
LaTeXRep_PRINT_PS_CMD	"dvips -q %s.dvi"
LaTeXRep_PREVIEW_DVI_CMD	"xdvi -hush %s.dvi &"
LaTeXRep_PREVIEW_PS_CMD	"ghostview -quiet -a4 %s.ps &"

Table 2.4: Environment variables and default settings for `LaTeXReporter`.

indicating a file name may be linked to the file it describes. When the user selects this file name, typically by clicking the mouse on it, the file in question is displayed by some suitable previewer. In Diffpack applications the class `HTMLReporter` uses this hypertext mechanism to present included PostScript figures. Moreover, the generated HTML-based reports will start with a list of contents. By selecting the wanted section, the user get instant access to the requested information. Next to all section headings, there is a text saying "To Contents". When this text is selected, the list of contents is brought back into view.

Limitations. The current implementation of `HTMLReporter` does not have support for mathematical typesetting. Instead, the text inserted in math mode is highlighted by using a slanted font. Moreover, tables are forced to have left justified cells regardless of the formatting instructions provided in the function calls. Both situations are likely to be improved when the next level of the HTML language is generally available. This new standard, which is presently known as HTML+, defines LaTeX-like environments for producing tables and mathematical expressions.

Even today, depending on which HTML browser that is used, the generated reports will have the standard text styles such as italics and boldface.

Postprocessing and presentation of reports. Unlike the report formats presented above, there is no need for explicit processing of HTML documents. The HTML report can simply be loaded into a browser like `Mosaic` [13], which also acts as the previewer or printing utility. If `Mosaic` is started manually, the report file can be specified by typing

```
Mosaic report.html
```

on the Unix command line. Alternatively, `Mosaic` can be started without arguments and the file can be loaded by choosing the option "Open Local..." from the "File" menu. In Figure 2.4 we show a screen page from a HTML-based report previewed in `Mosaic`.

If `Mosaic` is not installed on your computer, it can be downloaded by anonymous ftp to `ftp.ncsa.uiuc.edu` or some other site. You may also try other HTML browsers that are distributed on the Internet.

Instead of starting an HTML browser explicitly, you may call `HTMLReporter`'s member function `preview`. This action will by default cause `Mosaic` to open the current report file. However, this choice of browser can be overruled by setting the Unix environment variable `HTMLRep_PREVIEW_CMD`. The default behaviour of `preview` is identical to what will be achieved if this variable is set to `"Mosaic %s.html &"`.

2.5 How to implement new report formats

It is possible to extend the `Reporter` hierarchy to cater for additional document formats. To do this, the new class, say `NewReporter`, should be derived from the base class `Reporter`. All functions that are initially defined as being pure virtual *must* be implemented. Moreover, you will probably need to reimplement most of the member functions that are specified as virtual. In order to maintain the possibility of nested environments and the automatic management of figure files, all `begin`*env* and `end`*env* functions should call the relevant base class implementations `Reporter::begin`*env* and`Reporter::end`*env* explicitly. The corresponding conventions are required also for the functions `header`, `trailer` and `section`.

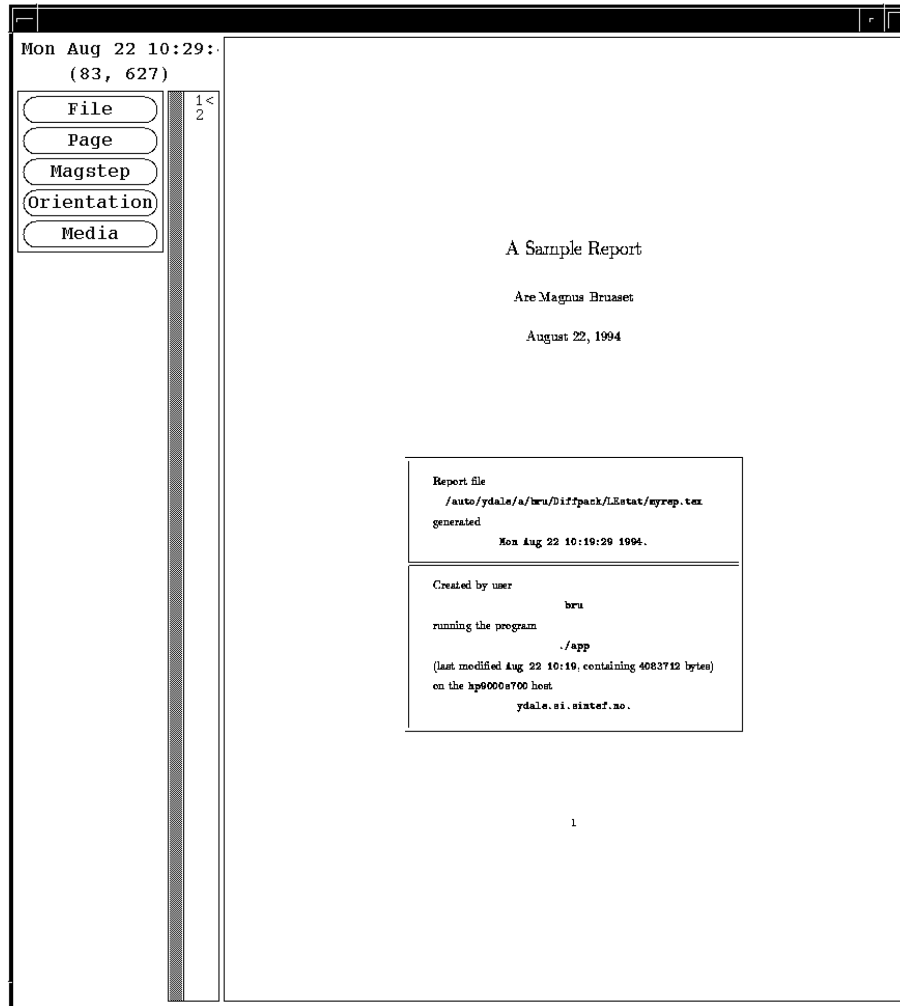

Figure 2.3: *A typical* `LaTeXReporter` *screen page previewed by* `ghostview`.

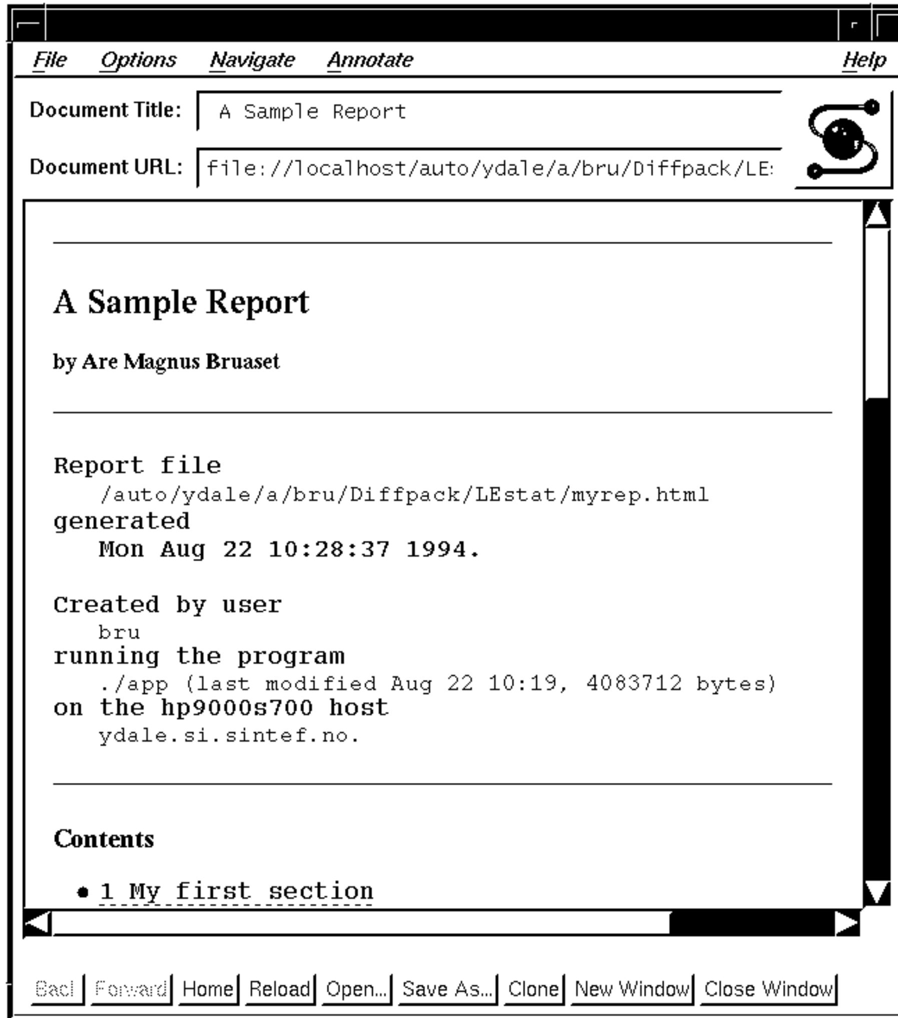

Figure 2.4: *A typical* `HTMLReporter` *screen page previewed by* `Mosaic`.

Chapter 3

A brief look at the Diffpack menu system

In order to prepare for the case study in Section 4, we will briefly review the use of the Diffpack menu system. For a more exhaustive description of this menu system we refer to the manual [11]. The main purpose of this section is to see how Diffpack menus can be put to work together with a `Reporter` object.

3.1 User dependent menu code

Any user-defined class that is supposed to interface to the menu system must be derived from the abstract base class `MenuUDC`[1]. In this way, the new class is equipped with certain member functions that are used to build and parse its own submenu. The `MenuUDC` definition includes these functions:

```
  // Basic MenuUDC functionality

virtual void define (MenuSystem& menu, int level = MAIN) =0;
virtual void scan   (MenuSystem& menu)                   =0;

virtual void adm    (MenuSystem& menu);

static void makeSubMenuHeader
  (
        MenuSystem&  menu,
   const String&     description,
```

[1]The acronym "`UDC`" refers to User Dependent Code. All Diffpack class names that end with this acronym refer to base classes providing a uniform interface to user-supplied code segments that are likely to change for different applications and problems.

```
    const String&      command,
          int&         level,
          char         hot_key
  );
```

The function define receives a MenuSystem object and an integer stating the current menu level. When returning from this function, new user-defined items and/or submenus have been added to the specified MenuSystem. The initial part of this definition phase should call makeSubMenuHeader to set certain attributes of the new submenu. Most applications will use the predefined MenuSystem object global_menu and just add their own contributions to this instance.

Once the menu is complete, it can be presented to the user by invoking the current MenuSystem's member function prompt. When the user has responded to this prompt, the menu items should be scanned to record the relevant menu choices. That is, the user-defined class has to supply the function scan that reads out the value for each item added in define. The combined process of defining a menu, prompting the user and scanning the menu choices is by default availabe as the single function adm.

By giving certain command line options to a Diffpack program, it is possible to choose between different menu modes. The default mode uses a simple text-based menu prompt. Other possibilities inlude a graphical user interface based on X/Motif, which is invoked by supplying the Unix command line option +isg to the program at start-up. However, in many situations it is more convenient to stick with the default text-based menu, since the menu choices can be taken directly from an ASCII file. Given an application app and a corresponding file of menu choices input_file.i, this can be done by redirection, i.e. by issuing the command

```
app < input_file.i.
```

Another alternative is to use the -D command line option, i.e. typing

```
app -D input_file.i.
```

The latter approach loads the contents of the input file as new default values for the specified menu items. The user is then free to make additional menu changes interactively, before accepting the current values by entering the menu command ok.

When running a program based on the Diffpack menu system, a file is generated that contains a description of the complete menu tree together with the actual choices used in that execution. This information can easily

provide a manual describing the user interface of the application in question. To produce such a manual typeset in LaTeX, simply enter

```
input_manual_in_latex
latex manual
latex table
```

on the Unix command line. The file `manual.tex` contains the manual itself, while `table.tex` is used to generate a compact overview of the menu choices used in the last execution.

3.2 Multiple loops

Numerical simuations are usually run for different sets of input parameters, typically affecting properties of the underlying mathematical model or the behaviour of the numerical algorithms. To simplify the management of input data, the Diffpack menu system provides a parameter analysis module. This feature allows the input parameters to be automatically changed according to given rules when running the simulation inside a multiple loop. A new set of parameters is then constructed for each pass of the loop. Any program that is supposed to be comaptible with the parameter analysis module, has to follow certain design rules. Most important, the numerical simulator should be implemented as a separate class derived from `MenuUDC`, possibly utilizing other classes as internal data structures or by multiple inheritance. In particular, this user-defined class must implement certain member functions inherited from `MenuUDC`:

```
    // These are used by MenuSystem::multipleLoop :

virtual void solveProblem ();

virtual void openReport () {}
virtual void resultReport ();
virtual void closeReport () {}

  // For monitoring progress of multiple loops

int getMultipleLoopLimit () { return ncomb; }
int getMultipleLoopIndex () { return icomb; }
```

In addition to the previously discussed procedures that define and scan the menu, we also need an implementation of `solveProblem`. This function is the entry point to the numerical simulator and is called once for each pass

of the loop. If the application is supposed to generate some kind of report on the results obtained for each iteration, the necessary output statements should be coded inside `resultReport`. Moreover, the functions `openReport` and `closeReport` are called immediately before and after the multiple loop, respectively. As indicated by their names, these functions give the user an opportunity for initializing and cleaning up objects and file structures used for report management.

To keep track of the loops administered by the menu system, two additional utility functions have proved to be handy. The function `getMultipleLoopIndex` returns the current loop index, starting with one and terminating when the value of `getMultipleLoopLimit` is reached. In most cases it is a good habit to include the current value of the loop counter as part of the output generated by `resultReport`. It is then possible to uniquely identify the paramter set used for this particular simulation.

Once the member functions inherited from `MenuUDC` are given a suitable implementation, the menu system can take control of the parameter variation. Instead of the standard sequence of calls to `define`, `prompt` and `scan`, this is typically achieved by invoking

```
global_menu.multipleLoop(MySimulator);
```

where `MySimulator` is an instance of the user-defined class derived from `MenuUDC`. Let us take a quick look at an example where we vary certain input parameters required by a program that solves a linear system of equations by some preconditioned iterative method. If the menu system is given the command

```
set basic method = {BiCGStab & TFQMR } ,
```

it will make two passes of the loop — one for each value of the item `basic method`. Including the instruction

```
set RILU relaxation parameter = { 0.0:0.8,0.2 & 0.9 & 0.95 }
```

the computations will be performed for 10 values of the RILU parameter. Seen in relation to the previous parameter variation, this adds up to 20 passes of the loop.

Chapter 4

A simple case study

In the previous sections we have outlined the report management in Diffpack and how this mechanism can work tightly coupled to the menu system. To fully demonstrate these versatile tools, we will now look at a specific application designed to solve the second-order elliptic boundary value problem

$$-\nabla \cdot (\nabla u) + v\,(\cos(\theta), \sin(\theta)) \cdot \nabla u = 0$$

in some box-shaped domain Ω subject suitable boundary conditions on $\partial\Omega$. When this problem is discretized by a finite element procedure, we face the problem of solving a set of n linear algebraic equations. Since this linear system is sparse and nonsymmetric, we would like to apply a preconditioned Krylov subspace method. However, the performance of iterative solvers for nonsymmetric problems is usually hard to estimate in advance. For this reason, we would like the application to compute the numerical solution for some values of v and θ, using different iterative algorithms. Motivated by the discussion in Section 3, we want to realize this experiment by running a multiple loop controlled by the menu system.

We start by creating a new class CDFEM that implements the simulator that we need in order to solve the given problem. The following code segment shows the member functions in CDFEM related to the menu system:

```
class CDFEM : public FEM, public MenuUDC, public Store4Plotting
{
protected:
  // ...
public:
  // ...
  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void scan   (MenuSystem& menu); // read and intialize data
  virtual void adm    (MenuSystem& menu);
```

```
  virtual void solveProblem ();
  virtual void openReport ();
  virtual void resultReport ();
  virtual void closeReport ();
};
```

We notice that this class is derived from the finite element framework `FEM` and the menu base class `MenuUDC`[1]. In particular, we have to implement the functions `define` and `scan` used to build menu items and reading the corresponding user responses, respectively. We will not go into details about the contents of these functions, the interested reader should instead look at the application code[2].

In order to have access to multiple loops for parameter analysis, we have to define the action that shall take place for a given set of input parameters. This is done by implementing the virtual function `solveProblem` inherited from `MenuUDC`. By default, the body of this function is empty, and we redefine it to call `driver` that is the computational kernel of class `CDFEM`:

```
//-----------------------------------------------------------------------------
void CDFEM:: singleProblem ()
//-----------------------------------------------------------------------------
{
  driver();
}
```

At this point, we may run the simulations without any report generation. This can be achieved by the following main program:

```
//-----------------------------------------------------------------------------
main (int nargs, const char** args)
//-----------------------------------------------------------------------------
{
  initDIFFPACK (nargs, args);
  global_menu.init ("Linear equation test solver", "LinEq-experiment");

  CDFEM problem;   // make a simulator object, called problem

  global_menu.multipleLoop (problem);
}
```

[1]`CDFEM` is also derived from the base class `Store4Plotting` that administers the storage of data files used for plotting. Since this aspect is of no interest in this particular example, it is not discussed further.

[2]The complete code listing of class `CDFEM` is given in Appendix A.

In order to let CDFEM produce its own result reports, we have to implement a version of the function resultReport. To demonstrate different possibilities, we want to use three alternative approaches. First, we want to print a compact summary of the solver's performance to the standard output stream. We would also like to produce a file containing these data in tabular form, and finally, we would like a complete report generated by a class in the Reporter hierarchy. To fulfill these requirements, we need the initialization procedure openReport that is automatically called by MenuSystem's multipleLoop before entering the parameter loop:

```
//-----------------------------------------------------------------------------
void CDFEM:: openReport ()
//-----------------------------------------------------------------------------
{
  rep.rebind(*createReporter(rep_format,"reportfile"));
  rep->header("Linear Equation Test Solver","K{\\aa}re Dump",dpTRUE);

  summary_file.rebind("FILE=result.tex");
  LinEqStatBlk:: openLaTeXtable(summary_file);
}
```

In this implementation, the string rep_format defines which type of reporter that is to be used. This information is typically read from the menu, and is used to allocate a Reporter object referred by the handle rep. Moreover, we open a file named result.tex that will contain a table of performance data. In our application, the performance data will automatically be collected by the class LinEqStatBlk, which also knows how to use the Reporter mechanism. Further details on the use of LinEqStatBlk is given in [2], but we should note the existence of static member functions for opening and closing LaTeX tables.

For each set of input parameters MenuSystem::multipleLoop will call the function resultReport given below:

```
//-----------------------------------------------------------------------------
void CDFEM:: resultReport ()
//-----------------------------------------------------------------------------
{
  LinEqStatBlk state;

  lineq.getStatistics(state);

  state.putLaTeXline(summary_file);
  state.print(s_o);
```

```
  rep->section(aform("Performance of solve #%d",
             getMultipleLoopIndex()).chars());
  rep->subsection("Problem dependent parameters");
  rep->beginItemize();
  rep->put(oform("Computational domain       : %s",geometry.chars()));
  rep->put(oform("Partitioning into grid     : %s",partition.chars()));
  rep->put(oform("Velocity magnitude (v)     : %e",v));
  rep->put(oform("Velocity direction (theta) : %e",theta));
  rep->endItemize();

  state.print(rep(),verb_level);
}
```

First, we allocate a `LinEqStatBlk` object to hold the performance data collected from the `LinEqAdm` object `lineq`. Then we print a compact summary of this information in LaTeX format to the output stream `os`. The same information is also written in ASCII form to the standard output stream `s_o`. The rest of `resultReport` is concerned with the `Reporter` object. It starts by generating a new section, for which the heading includes the current value of the loop counter. We also make a subsection for reporting on problem dependent parameters such as the current geometry and the value of `theta` (θ). This information is written by calls to `rep->put`. However, since these statements are surrounded by the delimiters of an `Itemize` environment, each `put` results in a new list item. Finally, we hand the `Reporter` object over to `LinEqStatBlk::print`, which uses it for a complete report on the solver's performance. The argument `verb_level` decides the level of details included in this report, see [2].

When the parameter loop terminates, the following function is called:

```
//-----------------------------------------------------------------------------
void CDFEM:: closeReport ()
//-----------------------------------------------------------------------------
{
  LinEqStatBlk:: closeLaTeXtable(summary_file);
  summary_file->flush();

  rep->trailer();
  rep->flush();
}
```

After closing the LaTeX table residing in the file represented by `os` and the report generator referred by `rep`, all relevant files are flushed. At this point the recorded information is available for further processing and presentation. To

demonstrate the capabilities of different report formats, we have included two appendices showing a sample input data file and the corresponding reports generated by `ASCIIReporter`, `LaTeXReporter` and `HTMLReporter`.

Appendix A

Source listing of the test program

A.1 The interface: CDFEM.h

```
// solver for v*(cos(theta),sin(theta))*grad u = div(grad u)

#include <FEM.h>            // FEM algorithms
#include <FieldFE.h>        // finite element grid and scalar field
#include <DegFreeFE.h>      // mapping: nodal values -> unknowns in linear sys.
#include <LinEqAdm.h>       // linear systems, storage and solution
#include <PreproBox.h>      // preprocessor for box-shaped grid
#include <MenuUDC.h>        // needed for automatic parameter analysis
#include <SimResFile.h>     // tool for reading a reference solution from file
#include <UpwindFE.h>       // upwind weighting functions (for high Pe numbers)
#include <Store4Plotting.h>

class CDFEM : public FEM, public MenuUDC, public Store4Plotting
{

protected:
  // general data:
  Handle(GridFE)    grid;  // finite element grid
  Handle(DegFreeFE) dof;   // trivial mapping here: nodal values -> unknowns
  Handle(FieldFE)   u;     // finite element field, the primary unknown
  LinEqAdm          lineq; // linear system, storage and solution
  PreproBox         p;     // preprocessor for box meshes
  UpwindFE          Wpert; // upwind weighting functions functions
  int               PG_tp; // indicator for Wpert method
  Ptv(real)         velocity,
                    diffusion;  // help var. for Wpert, set in scan

  real v;                  // velocity in the equation
```

```
  real theta;             // direction of velocity in the equation

  real              error_L2;    // estimation of the L2-error of u
  SimResFile        refsol;      // reference solution for comp. error_L2
  FieldFE           u_ref;       // reference solution field
  FieldFE           u_error;     // error field

  Handle(Reporter)  rep;         // report with more detailed results
  String            rep_format;  // ascii, LaTeX or html format for rep
  VerbosityLevel    verb_level;  // verbosity level of the rep report
  Os summary_file;               // file with compact summary of results

  BooLean           make_plotmtv_plot;

  String geometry;               // input to preprocessor
  String partition;


public:

  CDFEM ();
 ~CDFEM () {}
  virtual void driver ();             // main driver routine

  // --- FEM computations: ---
  virtual void fillEssBC ();   // set boundary conditions
  virtual void integrands      // evaluate weak form in the FEM-equations
    (ElmMatVec& elmat, FiniteElement& fe);
  real errorL2 ();             // empirical L2 error (compares u and u_ref)

  // --- menu system: ---
  virtual void define (MenuSystem& menu, int level = MAIN);
  virtual void scan   (MenuSystem& menu); // read and intialize data
  virtual void adm    (MenuSystem& menu);

  // --- automatic parameter analysis: ---
  virtual void solveProblem ();
  virtual void openReport ();
  virtual void resultReport ();
  virtual void closeReport ();
};
```

A.2 The implementation: CDFEM.C

```
#include <CDFEM.h>
#include <ElmMatVec.h>
#include <FiniteElement.h>
```

```
#include <DrawFE.h>
#include <LinEqStatBlk.h>
#include <createReporter.h>
#include <ErrorEstimator.h>


//-----------------------------------------------------------------------------
CDFEM:: CDFEM ()
//-----------------------------------------------------------------------------
  : lineq (EXTERNAL_SOLUTION)
{}



//-----------------------------------------------------------------------------
void CDFEM:: fillEssBC ()
//-----------------------------------------------------------------------------
{
  dof->initEssBC ();
  int nno = grid->getNoNodes();
  for (int i = 1; i <= nno; i++) {
    if (grid->BoNode (i,1))    // is node i subjected boundary indicator no 1?
      dof->fillEssBC (i, 0.0); // u=0
    else if (grid->BoNode (i,2))
      dof->fillEssBC (i, 1.0); // u=1
  }
}



//-----------------------------------------------------------------------------
void CDFEM:: integrands (ElmMatVec& elmat, FiniteElement& fe)
//-----------------------------------------------------------------------------
{
  int i,j;
  const int nbf = fe.getNoBasisFunc(); // no of nodes (or basis functions)
  const real detJxW = fe.detJxW();     // det J times numerical itg.-weight
  Wpert.calcWeightingFunction (fe, velocity, diffusion, 0.0, dpTRUE, PG_tp);
  real Wi;

  for (i = 1; i <= nbf; i++) {
    Wi = fe.N(i);
    for (j = 1; j <= nbf; j++)
      elmat.A(i,j) += (Wi*(velocity(1)*fe.dN(j,1) + velocity(2)*fe.dN(j,2))
       + (fe.dN(i,1)*fe.dN(j,1) + fe.dN(i,2)*fe.dN(j,2)) )*detJxW;
    elmat.b(i) += 0;
  }
}
```

```
//-----------------------------------------------------------------------------
void CDFEM:: driver () // main routine
//-----------------------------------------------------------------------------
{
  fillEssBC ();              // set essential boundary conditions
  makeSystem (dof(), lineq); // calculate linear system
  lineq.solve();             // solve linear system
  error_L2 = errorL2();      // estimate error (wrt a reference solution)
  dump (u(), NULL, oform("v=%g, theta=%g",v,theta));
  lineCurves (u());//, NULL, oform("v=%g, theta=%g",v,theta));
  // plot for test:
  if (make_plotmtv_plot)
    DrawFE::plotmtvScalar (u(), CaseName+".mtv",BINARY,
   oform("%c comment=\"v=%g theta=%g %d nodes\"",
 '%',v,theta,grid->getNoNodes()));
}



//-----------------------------------------------------------------------------
void CDFEM:: adm (MenuSystem& menu)
//-----------------------------------------------------------------------------
{
  define (menu);
  menu.prompt();
  scan (menu);
}



//-----------------------------------------------------------------------------
real CDFEM:: errorL2 ()
//-----------------------------------------------------------------------------
{
  real error = DUMMY;
  if (refsol.ok()) {   // is the reference solution available?
    // load reference solution:
    if (readField (u_ref, refsol, u->getFieldname(), DUMMY, dpFALSE)) {
      error = ErrorEstimator::L2Norm (u(), u_ref);
      ErrorEstimator::errorField (u(), u_ref, u_error);
      dump (u_error, NULL, oform("v=%g, theta=%g",v,theta));
    }
  }
  return error;
}
```

```
//-----------------------------------------------------------------------------
void CDFEM:: singleProblem ()
//-----------------------------------------------------------------------------
{
  driver();
}



//-----------------------------------------------------------------------------
void CDFEM:: openReport ()
//-----------------------------------------------------------------------------
{
  rep.rebind(*createReporter(rep_format,"reportfile"));
  rep->header("Linear Equation Test Solver","K{\\aa}re Dump",dpTRUE);

  summary_file.rebind("FILE=result.tex");
  LinEqStatBlk:: openLaTeXtable(summary_file);
}



//-----------------------------------------------------------------------------
void CDFEM:: resultReport ()
//-----------------------------------------------------------------------------
{
  LinEqStatBlk state;                   // statistics of linear solvers
  lineq.getStatistics(state);
  state.putLaTeXline(summary_file);     // print a compact line of results
  state.print(s_o);                     // print a copy to the screen

  // more verbose presentation of results in the report:
  rep->section(aform("Performance of solve #%d",
     getMultipleLoopIndex()).chars());
  rep->subsection("Problem dependent parameters");
  rep->beginItemize();
  rep->put(oform("Computational domain       : %s",geometry.chars()));
  rep->put(oform("Partitioning into grid     : %s",partition.chars()));
  rep->put(oform("Velocity magnitude (v)     : %e",v));
  rep->put(oform("Velocity direction (theta) : %e",theta));
  rep->endItemize();
  state.print(rep(),verb_level);
}



//-----------------------------------------------------------------------------
void CDFEM:: closeReport ()
```

```
//-----------------------------------------------------------------------------
{
  LinEqStatBlk:: closeLaTeXtable(summary_file);
  summary_file->flush();
  rep->trailer();
  rep->flush();
}


//-----------------------------------------------------------------------------
void CDFEM:: define (MenuSystem& menu, int level)
//-----------------------------------------------------------------------------
{
  menu.addItem (level,
              "geometry",
              "geometry",
              "computational domain",
              "d=2 [0,1]x[0,1]",
              "S",          // valid answer: string
              'g','g');
  menu.addItem (level,
              "no of elements in each direction",
              "nel_x",
              "square grid with nx=ny",
              "30",
              "R[0:1000]1",          // valid answer: Real number
              'n','n');
  menu.addItem (level,
              "element type",
              "elm_tp",
              "name in ElmDef hirarchy",
              "ElmB4n2D",
              "S/ElmT3n2D/ElmB4n2D/ElmTensorProd1/ElmB9n2D/ElmTensorProd2/"
              "ElmB8n2D/",
              'n','n');

  menu.addItem (level,
              "velocity magnitude",
              "velmag",
              "magnitude of velocity in the equation",
              "2.5",
              "R",
              'm','m');

  menu.addItem (level,
              "velocity direction",
              "veldir",
              "direction of velocity in the equation (in radians)",
              "0.53",
```

```
                "R",
                'd','d');

  LinEqAdm::fullMenu (ON,2);  // save extra scan job for it. methods
  LinEqAdm::defineStatic (menu, level+1);

  menu.addItem (level,
                "report format",
                "repform",
                "class name in Reporter hierarchy",
                "LaTeXReporter",
                validationString(hierReporter()),
                'R','R');

  menu.addItem (level,
                "verbosity level",
                "vlev",
                "indicator for detail level in report",
                "COMPACT",
                "S/COMPACT/EXTENDED/DETAILED/",
                'V','V');
  menu.addItem (level,
                "reference dataset basename",
                "refsol",
                "SimResFile file with a ref. sol. (no v/t info in name)",
                "NONE",
                "S",
                'R','R');
  menu.addItem (level,
                "upwind method",
                "upwind",
                "0: none, 1: std Brooks&Hughes, 2: temporal B&H",
                "1",
                "I[0:2]1",
                'u','u');
  menu.addItem (level,
                "plotmtv plot",
                "plotmtv",
                "",
                "OFF",
                "S/ON/OFF/",
                'p','p');
}


//-----------------------------------------------------------------------------
void CDFEM:: scan (MenuSystem& menu)
//-----------------------------------------------------------------------------
```

```
{
  geometry = menu.get ("geometry");
  p.geometryBox().scan (geometry);

  String element_tp = menu.get ("element type");
  int n = menu.get ("no of elements in each direction").getInt();
  partition = aform("d=2 et=%s [%d,%d] g=[1,1]",element_tp.chars(),n,n);
  p.partitionBox().scan (partition);

  grid.rebind (new GridFE());
  p.generateMesh (grid());     // run the preprocessor

  // modify boundary indicators to u=0 at y=0 and x=0, 0<y<1/4, u=1 at the
  // "rest" of x=0, while du/dn=0 at the two lines x=1 and y=1.

  grid->redefineBoInds ("nb=2 names= zero unity 1=(4) 2=(3)");
  grid->addBoIndNodes ("n=1 b1=[0,0]x[0,0.25]");

  v = menu.get ("velocity magnitude").getReal();
  theta= menu.get ("velocity direction").getReal();
  velocity.redim(2); diffusion.redim(2);
  velocity(1) = v*cos(theta); velocity(2) = v*sin(theta); diffusion.fill(1.0);
  PG_tp = menu.get ("upwind method").getInt();
  String plot = menu.get ("plotmtv plot"); assignEnum (make_plotmtv_plot,plot);

  // make curve plots normal to the front, all lines go through (0.5,0.5):
  line1_start.redim(2); line1_stop.redim(2);
  const real tanexpr = tan(M_PI-theta);
  line1_start(1) = ( 0.5 + 0.5*tanexpr)/tanexpr;
  line1_stop (1) = (-0.5 + 0.5*tanexpr)/tanexpr;
  line1_start(2) = 1;
  line1_stop (1) = 0;

  // --- allocate data: ---
  u.rebind (new FieldFE (grid(), "u"));    // allocate space for u
  dof.rebind (new DegFreeFE (grid(), 1));  // 1 for 1 unknown per node
  u->fill (0.0);                           // initial solution for lineq
  lineq.attach (u->values());              // lend u's nodal values to lineq
  lineq.scan (menu);

  rep_format = menu.get ("report format");
  String v_lev = menu.get ("verbosity level");
  assignEnum(verb_level,v_lev);

  // to dump to u to different files for each multiple loop execution, call
  // Store4Plotting::scan or Store4Plotting::open, we call the latter
  // since we make a filename containing v and theta (that makes it easier
  // to read the correct reference solution in a multiple loop):
  Store4Plotting::open (aform("%s-v%g-t%g",CaseName.chars(),v,theta));
```

```
  String reference_dataset = menu.get ("reference dataset basename");
  if (!reference_dataset.contains("NONE"))
    // when v and theta changes during a multiple loop, new reference solution
    // files must be opened:
    refsol.open (aform("%s-v%g-t%g",reference_dataset.chars(),v,theta));

/*
   Normally, the SimRes files have names base on CaseName only. In multiple
   loop executions this results in .mycase_m1.field etc. files.
   If these are to be used as reference solutions for error estimation
   it may be difficult in a multiple loop to load the proper .mycase_m*.field
   file from the reference solution dataset. In the present example we have
   solved this problem in the following way.
   We add the important physical parameters to the SimRes file name, here
   v and theta, giving names on the form mycase_m5-v0.1-t0.23. When reading
   this as a reference solution we still have the prblem with the _m*
   part of the name. This part can be removed by the Mvname script:

     Mvname '_m[0-9]*' '' .Tref*.grid
     Mvname '_m[0-9]*' '' .Tref*.field
     Mvname '_m[0-9]*' '' .Tref*.grid_ix
     Mvname '_m[0-9]*' '' .Tref*.field_ix

   The name of the reference dataset is now mycase-v0.1-t0.23.
   Why not construct the names on basis of CaseNameOrig and v and theta?
   This is what we want when we generate a reference solution. However, in
   an ordinary multiple loop simulation (not generating a reference solution)
   we would like to examine the results from, in principle, every execution
   in the loop. Using CaseNameOrig (or Mvname) overwrite files: For a given
   v and theta solutions, e.g. corresponding to different iterative solvers,
   will overwrite each other.
*/
}
```

Appendix B

Input file used by the test program

The following file lists the input used to generate the reports given in appendix C. The lines beginning with an exclamation mark (‘!’) are just comments.

```
! Problem dependent parameters (default geometry and partitioning)
set velocity (magnitude) = 2.5
set velocity (direction) = 0.53

! Set parameters connected to the linear solver
sub LinEqAdm

sub prm(Matrix)
set matrix type = MatSparse
ok

sub prm(LinEqSolver)
set basic method =  {BiCGStab & TFQMR}
ok

sub prm(Precond)
set preconditioning type = PrecRILU
set RILU relaxation parameter = 0.9
ok

ok

! Choose a report format
set Report format = ASCIIReporter
!set Report format = LaTeXReporter
!set Report format = HTMLReporter
```

```
! Defines the level of details reported by LinEqStatBlk
set Verbosity level = DETAILED

ok
```

Appendix C

Output generated by the test program

C.1 ASCII format

This report was generated by the input data given in appendix B, setting the option `Report format` equal to `ASCIIReporter`.

```
################################################################################

                          Linear Equation Test Solver

                                      by

                                 K{\aa}re Dump


################################################################################
```

```
================================================================================

     Report file

                    /auto/ydale/a/bru/tst2/reportfile.txt

     generated

                          Tue Oct  4 16:10:14 1994.

--------------------------------------------------------------------------------

     Created by user
                                      bru

     running the program
                                     ./app

             (last modified Oct 4 16:07, containing 5271552 bytes)

     on the hp9000s700 host

                             ydale.si.sintef.no.

================================================================================

 1 Performance of solve #1
==========================

 1.1 Problem dependent parameters
---------------------------------

     * Computational domain       : d=2 [0,1]x[0,1]
     * Partitioning into grid     : d=2 elm_tp=ElmB4n2D div=[30,30] grading=[1,1]
     * Velocity magnitude (v)     : 2.500000e+00
     * Velocity direction (theta) : 5.300000e-01


 1.2 System statistics
----------------------

     * Number of unknowns in system : 961
     * Block dimensions             : 1 x 1
     * Final residual in l2 norm    : 2.635435e-04
     * System representation        : LinEqSystemPrec
```

```
    * Block structure of A: doubleMatSparse (961 x 961) NOT_TRANSPOSED
    * Block structure of x: doubleArrayGen (961 x 1)
    * Block structure of b: doubleVec (961 x 1)



 1.3 Solver statistics
----------------------

   * Linear solver          : BiCGStab (Stabilized Biconjugate Gradients)
   * Number of iterations   : 7 (of max. 300)
   * Start vector mode      : USER_START
   * Converged              : dpTRUE
   * Total elapsed CPU time : 9.400000e-01 seconds
   * Termination state      : Convergence reached within the maximum number of iterations


 1.4 Preconditioning statistics
-------------------------------

   * Preconditioning mode : LEFT_PREC
   * Left preconditioner  :

     * Applied from left    : PrecRILU (Relaxed incomplete LU factorization)
     * Fill-in level        : 0
     * Relaxation parameter : 9.000000e-01

   * Right preconditioner :

     * Applied from right   : PrecNone (No preconditioner, i.e. the identity operator)



 1.5 Convergence statistics
---------------------------

   * Compound convergence test:
                           if ( C_{1} ) then stop
   * Convergence monitor #1, (see Figure fig1:1)

Sorry, can not include figures in ASCII report


     * Convergence monitor: CMRelResidual


 \parallel \! r^{k} \!\parallel/
\parallel \! r^{0} \!\parallel  \leq \varepsilon
```

```
    * Given tolerance \varepsilon          : 1.000000e-04
    * Monitoring residual type               : ORIGINAL_RES
    * Last base value                        : 1.239216e+01
    * Last monitored value using the l2 norm : 2.126696e-05
    * Use spectral estimates for scaling     : dpFALSE
    * Acting as stopping criterion           : dpTRUE
    * Recording convergence history          : dpTRUE
    * Inserted at list front (relational operator CM_AND)



 1.6 Work and storage estimates
-------------------------------

   * Work cost    :

     * system   : 3754709 flops
     * internal : 42 flops
     * total    : 3754751 flops

   * Storage cost :

     * system   : 3.169937e-01 Mb
     * internal : 0.000000e+00 Mb
     * total    : 3.169937e-01 Mb




 2 Performance of solve #2
==========================

 2.1 Problem dependent parameters
---------------------------------

   * Computational domain       : d=2 [0,1]x[0,1]
   * Partitioning into grid     : d=2 elm_tp=ElmB4n2D div=[30,30] grading=[1,1]
   * Velocity magnitude (v)     : 2.500000e+00
   * Velocity direction (theta) : 5.300000e-01


 2.2 System statistics
----------------------

   * Number of unknowns in system : 961
   * Block dimensions             : 1 x 1
   * Final residual in l2 norm    : 1.255124e-03
   * System representation        : LinEqSystemPrec
```

```
      * Block structure of A: doubleMatSparse (961 x 961) NOT_TRANSPOSED
      * Block structure of x: doubleArrayGen (961 x 1)
      * Block structure of b: doubleVec (961 x 1)



 2.3 Solver statistics
----------------------

    * Linear solver          : TFQMR (Transpose-free Quasi-minimal Residuals)
    * Number of iterations   : 14 (of max. 300)
    * Start vector mode      : USER_START
    * Converged              : dpTRUE
    * Total elapsed CPU time : 1.040000e+00 seconds
    * Termination state      : Convergence reached within the maximum number of iterations


 2.4 Preconditioning statistics
-------------------------------

    * Preconditioning mode : LEFT_PREC
    * Left preconditioner  :

      * Applied from left    : PrecRILU (Relaxed incomplete LU factorization)
      * Fill-in level        : 0
      * Relaxation parameter : 9.000000e-01

    * Right preconditioner :

      * Applied from right   : PrecNone (No preconditioner, i.e. the identity operator)



 2.5 Convergence statistics
---------------------------

    * Compound convergence test:
                           if ( C_{1} ) then stop
    * Convergence monitor #1, (see Figure fig2:1)

Sorry, can not include figures in ASCII report


      * Convergence monitor: CMRelResidual


 \parallel \! r^{k} \!\parallel/
\parallel \! r^{0} \!\parallel  \leq \varepsilon
```

```
      * Given tolerance \varepsilon          : 1.000000e-04
      * Monitoring residual type             : ORIGINAL_RES
      * Last base value                      : 1.331102e+01
      * Last monitored value using the l2 norm : 9.429210e-05
      * Use spectral estimates for scaling   : dpFALSE
      * Acting as stopping criterion         : dpTRUE
      * Recording convergence history        : dpTRUE
      * Inserted at list front (relational operator CM_AND)


 2.6 Work and storage estimates
-------------------------------

    * Work cost    :

      * system   : 3762990 flops
      * internal : 462 flops
      * total    : 3763452 flops

    * Storage cost :

      * system   : 3.169937e-01 Mb
      * internal : 0.000000e+00 Mb
      * total    : 3.169937e-01 Mb



                     ***  E N D   O F   R E P O R T  ***
```

C.2 LaTeX format

This report was generated by the input data given in appendix B, setting the option `Report format` equal to `LaTeXReporter`.

C.3 HTML format

This report was generated by the input data given in appendix B, setting the option `Report format` equal to `HTMLReporter`. In this case the screen pages have been converted to PostScript code. The hypertext links that appears as colored text on the screen are now indicated as underlined text.

Appendix D

Multi-format report generation

In many applications it is desirable to produce a variety of reports with different formats and detail levels. Based on the Reporter hierarchy, the Diffpack tool MultipleReporter is designed to handle such requirements. The interface of this class is very similar to the one used by Reporter, but includes some additional features for generation of summary reports.

The MultiReporter class administer seven simultaneously generated reports. Three of these are full reports in ASCII, LaTeX and HTML formats as produced by ASCIIReporter, LaTeXReporter and HTMLReporter, respectively. Moreover, compact summary reports are produced using these three document formats in addition to a simple ASCII table.

For further details on the design and use of MultipleReporter we refer to the corresponding man page, for which the source is found in the header file

```
$BTR/src/libs/bt2/report/MultipleReporter.h.
```

A demonstration of the MultiReporter's capabilities can be found in the directory

```
$TIMR/doc/reporter/Report-Demo,
```

see class CDFEM2.

Bibliography

[1] E. Acklam and H. P. Langtangen. Getting started with Diffpack revisited. World Wide Web document: Diffpack v1.4 Report Series, SINTEF & University of Oslo, 1996.
URL: *http://www.nobjects.com/diffpack/reports.*

[2] A. M. Bruaset and H. P. Langtangen. Getting started with numerical linear algebra in `Diffpack`. SINTEF report, in preparation.

[3] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[4] H. J. S. Feder and H. P. Langtangen. Public domain plotting and visualization software. Report STF33 A93022, SINTEF Applied Mathematics, 1993.

[5] L. Lamport. *LaTeX Users Guide & Reference Manual.* Addison-Wesley, 1986.

[6] H. P. Langtangen. Basic concepts in Diffpack. World Wide Web document: Diffpack v1.4 Report Series, SINTEF & University of Oslo, 1996.
URL: *http://www.nobjects.com/diffpack/reports.*

[7] H. P. Langtangen. Getting started with finite element programming in Diffpack. World Wide Web document: Diffpack v1.4 Report Series, SINTEF & University of Oslo, 1996.
URL: *http://www.nobjects.com/diffpack/reports.*

[8] H. P. Langtangen. Smart pointers. World Wide Web document: Diffpack v1.4 Report Series, SINTEF & University of Oslo, 1996.
URL: *http://www.nobjects.com/diffpack/reports.*

[9] H. P. Langtangen. Plotting of curves in Diffpack. The Numerical Objects Report Series #1997:4, Numerical Objects AS, Oslo, Norway, October 6, 1997. See *ftp://ftp.nobjects.com/pub/doc/NO97-04.ps.gz.*

[10] H. P. Langtangen and B. F. Nielsen. Getting started with Diffpack. World Wide Web document: Diffpack v1.4 Report Series, SINTEF & University of Oslo, 1996.
URL: *http://www.nobjects.com/diffpack/reports.*

[11] H. P. Langtangen and G. Pedersen. Simple and flexible input data handling in C++ programs. The Numerical Objects Report Series #1997:2, Numerical Objects AS, Oslo, Norway, October 6, 1997. See *ftp://ftp.nobjects.com/pub/doc/NO97-02.ps.gz.*

[12] S. B. Lippman. *C++ Primer*. Addison-Wesley, 2nd edition, 1992.

[13] NCSA. The NCSA Mosaic home page. World Wide Web document:
URL: *http://www.ncsa/uiuc.edu/SDG/Software/Mosaic/NCSAMosaicHome.html.*