The Art of Computational Science

The Kali Code

vol. 5

Documentation: Acsdoc

Piet Hut and Jun Makino

September 13, 2007

Contents

Preface

At the end of Volume 0, Alice and Bob briefly discussed the question of documentation. They then moved on, in Volume 1, to learn Ruby and to write their first 2-body code in Ruby. While doing the reseach described in Volume 1, they also made some notes about how to write Volume 1, and similarly in the following volumes. However, after moving on to the N-body system in Volume 4, things just got too complicated to keep track of, and they decided to settle on a more systematic way to record their adventures. After some discussion (see chapter 1 in this volume) they choose to use Ruby for documentation as well as for code writing.

The main idea of their writing system is to use a specially designed format, called *acsdoc*, which in turn is a stand-alone variant of *rdoc*[1]. From each acsdoc file, html or postcript or pdf files can be automatically generated.

For those readers who are interested in making a small contribution to our project, in the form of some code extensions or other small tools, it is not necessary to master the acsdoc format. However, for those of you who are interested in writing a whole volume, either by yourself or in collaboration with us, it is important to do so in acsdoc.

xxx (to be written:)

[In order to write in acsdoc, it is not necessary to go through this whole volume. Instead, you can follow the instructions in *A Quick Introduction to Acsdoc*, which can be found at xxxxx. Of course, if you really like to know what we did, how we did it, and most importantly, why we did it, by all means, read on, and follow all the details!]

0.1 Acknowledgments

We thank xxx, xxx, and xxx for their comments on the manuscript.

Piet Hut and Jun Makino

[1] `http://rdoc.sourceforge.net`

Chapter 1

Introduction

1.1 Documentation

Alice: Now that we got so far at to write actual N-body codes in Ruby, with a rich variety of integratorsk it is high time that we begin to write some documentation.

Bob: I don't like writing documentation.

Alice: How about reading documentation?

Bob: Well, it depends. If it is written well, and if it is useful, of course I like it. But so much documentation is neither written well nor useful, whether it is a manual that comes with a new DVD player or instructions to fill out your tax form. And software documentation especially seems to lack clarity, in many cases.

Alice: What do you find lacking?

Bob: For one thing, it is often too short to be useful. It is written as an afterthought, by someone who already has been working on a project for a long time. As a result, this person can no longer imagine what the original problems where, and probably not even what led him or her to the original design decisions. Typically, reading a software manual is like walking into a theater in the middle of a movie. Lots of action, but hard to say what is going on, or why.

Alice: So, let's do things better then. Why don't we keep notes of all of our conversations, and present those to the users of our software, our students or whomever else will find our codes on the web?

Bob: Surely you are joking. Notes of *all* of our conversations? That will quickly produce a few shelves worth of books!

Alice: As long as we keep it all online, who cares how many shelves would be stacked with printed versions. The point is to have it available, later, both for ourselves and others. And we don't have to write up every scrap of dialogue between us: we can certainly distill it a bit. But it should retain the flavor of a dialogue, rather than a laundry list of things-to-do and things-done.

Bob: You seem to be serious! Do you have any idea of how much work this will be?

Alice: It will be quite a bit of work, but I think that *not* doing so will be even more work.

Bob: Huh?

Alice: I mean, if we don't write notes about the whole process of code development, we are destined to spend more time later trying to reconstruct it. Just imagine what will happen. We start with a few nice toy models. Some students come and make some extensions. Before we know it, they or we or both use it in a little research project. Then, a few months later, we want to extend its use for another research project. By that time everyone will have forgotten the details of how the code was modified at which stage, and how the code was designed in the first place.

Now imagine the alternative. We keep notes about our discussions even *before* we type the first key stroke of a program. In doing so, we summarize what we think the problem is, and the way we think we can solve the problem. Then, while we write the code, and while we keep changing it, we also realize that the way we look at the problem keeps changing. So everything will be in flux: questions, answers, methods, approaches. But, because we keep some notes during each session, at any stage we can go back and check to see what happens.

So half a year later, a student comes into your office, and asks you questions about a piece of code. Instead of taking a deep breath, and steadying yourself for an hour of digging and trying to remember, you just smile and hand the student the URL of our conversation notes. The students happily leaves your office, and you happily continue your own work.

Bob: I must say, that sounds almost idyllic.

Alice: and here is another thought experiment. You yourself want to extend a piece of code you wrote a year ago. You vaguely remember that you stopped development on that code because something wasn't quite right. However, you can't remember what wasn't right. Was it that you had to give a conference presentation on another topic, so you had to stop working on the code, and never got back to it? Or was it that you realized that the underlying idea had some logical problems in some cases? Or did you just loose interest in the problem? Or did you find that another piece of code by someone else already did the job and you used that?

If you had gotten into the discipline of always making notes during each session of working on your code, you would never have to scratch your head in such a case; you'd just look up what happened. And what I just described is only the top layer. If you decide to continue working on the code, it would be wonderful to be able to refresh your memory about the many details that went into the design process in the first place.

Bob: I must admit, that also sounds good, but I'm afraid it sounds too good to be true. If it is a matter of just a few notes, the information will be hopelessly incomplete. On the other hand, if we had detailed notes for every session that I worked on a code, I'm not sure that I could retrieve the information I wanted.

So I don't think it will work. Just listing: "I did this, because of that, and then I did such in order to do so" does not generate a very interesting document. In order to make it useful, you'd better provide some good structure. But doing so takes time: it would be like writing a paper, each time you write a piece of code!

1.2 Dialogues

Alice: You put your finger on the problem. And your last point shows to me what is so nice about a dialogue. Just listing what happened will be dry and boring, and indeed hard to read later on. But if we recreate a dialogue between the two of us, we can tell our students and colleagues how and why we did what, in a natural flow of arguments.

Bob: Like Plato's dialogues?

Alice: Perhaps more like the dialogues that Galileo wrote. But let's leave out Simplicio, and just take two able researchers.

Bob: You and me?

Alice: Why not?

Bob: Hmmm. I doubt that it would be practical. You're not going to tape our conversations, are you?

Alice: Oh no, that would be too much work and also it would give far too much material. Instead, we'll make our written dialogues much shorter than the real-life ones.

Bob: So you want to distill our wisdom?

Alice: Yes, distilling is I guess how you could say it, but in the process we'll distill our whole experience, from wisdom to foolishness. It is essential to show what problems we ran into, how we found out that there were problems, how we traced the roots of those problems, and finally how we found solutions . . .

Bob: . . . with much trial and error.

Alice: Exactly. For students it will probably be interesting to see how we go about problem solving, and for colleagues at least it will give them a precise idea of why we choose our design details the way we did. When they can see the whole path of development, they have much more of a basis to agree or disagree. To the extent that they agree with our approach, they can then add more material to our codes in the same spirit. And to the extent they disagree, it will be easier for them to point out exactly where they part ways with us, and why.

Bob: Well, I'm sure it will be interesting for the students for another reason too.

Alice: What do you have in mind?

Bob: I'm sure they'll enjoy seeing us making mistakes! It will make them feel better, I'm sure.

Alice: Good point. I remember when I started programming, how stupid I often felt when something didn't work right away.

Bob: Until you found out that in programming almost always things dont' work right away.

Alice: Exactly. Okay, let's do it!

Bob: Wait. If we really want to put all our mistakes on paper, perhaps the students will enjoy seeing our first half dozen or so mistakes, but at some point even they will get bored.

Alice: So we'll have to be creative in our distillation process. For example, in the first few volumes we can show them how we stumbled here and there, trying to learn Ruby, for example. But further on, we can polish the presentation.

Bob: It will seem then that we will become very smart very quickly, in rather unrealistic ways.

Alice: Sure, but that's fine. For the record, let's just note here that all similarities between the Bob and Alice of our dialogues and actual astronomers present or past is completely accidental!

1.3 Presentation Format

Bob: I must say, you're making a strong case for writing dialogues. I'm willing to give it a try.

Alice: okay, let's get started right away! Remember the conversation we had when we saw each other over coffee, and asked each other what our research plans were? That's how we got this idea of writing some code for toy model simulations of dense stellar systems.

Bob: I guess we can reconstruct a dialogue along those lines. That will be a good place to start, since we were just chatting. It will become more complicated once we try to reconstruct how we learned to work with Ruby, how we coded up the two-body problem, how I got carried away with adding yet more integrators, and so on.

But already with the first volume, a pure dialogue, how do you want to write that, in what form? Plain text is not very helpful. At the very least we want to put that material on the web, to make it accessible to our students and others who may be interested.

Alice: But I don't like writing everything in HTML. For one thing, mathematical equations don't lend themselves very well to that medium. For another, I like to make printouts, especially of complex codes and of text with many equations, and I far prefer LaTeX for that purpose.

Bob: There are programs that can translate LaTeX to HTML, but from what I've seen, they are not doing a very good job. Perhaps a better alternative would be to start with a third medium, from which we can then produce both postscript output, like with LaTeX, and HTML output, for web sites.

Alice: Do you have any particular example in mind.

Bob: XML is being advertised as a *lingua franca* for exactly these purposes, but I'm a little hesitant to make a jump in that direction. Everything I've seen is rather cumbersome, at least on the level of writing XML directly, even though the basic idea is fine. The problem is that the XML source is almost unreadable, with even more markers and flags and begin-this and end-that than you already have in HTML.

Alice: How do people then work with XML in practice?

Bob: I presume they use an editor that allows them to add all the extra information, where needed, without showing it all on the screen.

Alice: A bit like the WYSIWYG, what-you-see-is-what-you-get, way in which many text editors nowadays show you what will be printed, rather than what the special symbols are that you type in order to get there?

Bob: Yes. Unlike emacs or vi, which show the full LaTeX commands, you could imagine working with an editor that would produce the LaTeX output directly in a window, while giving you a menu from which you can pick the directives.

Alice: Probably somebody has already written such a thing.

Bob: I bet. However, I'm used to good old ascii coding in editors where you can exactly see what you put in.

Alice: Me too. I really don't like having to click a mouse at every point along the way. Besides, if the input is in ascii, you know for sure that you will still be able to read that material ten years from now. If you write it in some form of 'easier' format, it is quite likely that some day you won't be able to decode anymore what you wrote!

Bob: I couldn't agree more. So I must admit, I don't like the XML solution very much, if it would mean working with a WYSIWYG editor. Besides, what would such an editor show you on the screen? HTML output format? LaTeX output format? A third type of output format? None of them would cover all the bases.

Alice: Have you tried writing XML that way?

Bob: No, though I came close. Recently, I was thinking about using some XML for documenting some of my codes, so I looked around on the web, to see what kind of editors were available. Unfortunately, the good ones all seemed te be commercial products, and I couldn't find anything in the open-source domain that looked appealing.

Alice: There is nothing wrong with paying for a software package.

Bob: Indeed, I'd be happy to pay for a good piece of software. However, when I do so, I need to be convinced that I will use it often enough to make it worth the investment. And even more importantly, I do want to have access to the source code, if I'm going to use it as a foundation for some of my own code development.

Alice: Those are good points. If we are going to use XML, we would want our students to join in and write extensions of what we're doing. If they first need to buy an expensive editor before they can even start working with our software, that wouldn't be ideal. And yes, it would be much better if they would have direct access to the source text of our dialogues as well as the source of our codes.

Bob: So let me look what alternatives there are available. There must be others who came up to the same dilemmas as we have just summarized. I'll do a web search, and with a little luck I may come across a better system. I'll let you know!

Chapter 2

Rdoc

2.1 Simple Elegance

xxx

nil

Chapter 3

End of old version

This is as far as we wrote this volume, originally.

In the following part, we present a first draft of a new introduction.

Introduction to ACSDOC
Piet Hut and Jun Makino

Contents

Chapter 4

Preface

This is a short introduction to ACSDOC, a document-processing system which can generate latex or html files from a common, easy-to-write text.

Chapter 5

Simple Example

The following is a simple example.

```
  = Simple Example

  This is a simple example.
```

This one creates the output shown in figure 5.1

Here, what is shown is the LaTeX output created by

```
   acsdoc.rb --tolatex simplexample.ok
   latex simplexample.tex
```

You can also create an HTML page by

```
   acsdoc.rb simpleexample.ok
```

This command create a single HTML file, simpleexample.html, at the same directory as the source .ok file lives.

Here[1] is the created HTML file.

[1] `../examples/simpleexample.html`

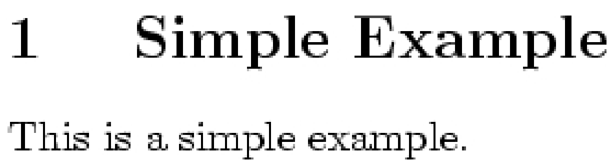

Figure 5.1: TeX output of simpleexample.ok

Chapter 6

Installation and other requirements.

6.1 Installing acsdoc

Acsdoc is provided as part of the ACS software system. It can be installed as a stand-alone software. acsdoc.rb is a single Ruby source program which require no other Ruby library files or whatsoever. So you can just copy it to your preferred location.

Documents are all written in acsdoc format, and can be created by

```
make documents
```

at the directory where the files are extracted from the archive.

We plan to offer some more "packaged" way to install the command and documents.

6.2 System requirements

Acsdoc has been tested on some distributions of Linux. It depends on UNIX operating system, and uses following commands/softwares

- mv, rm, cp, cat, csh
- convert from ImageMagick
- latex, dvips

- latex ”subsubsection” package

`Dvips` needs to understand options -E, -l, -x. If these options are not available with your dvips, well, you need to modify ascdoc.rb to supply appropriate options.

Chapter 7

Running acsdoc.

Syntax to invoke acsdoc is the following.

```
  % acsdoc.rb [options] [source files] infile ...
```

Here, `[options]` are command line options, `[source files]` are name of the program source file used in listing, `infile` is the input text file.

In the following I'll describe the meaning of command line arguments.

7.1 Input text file.

`Infile` is the input text file. Its format is described in more details in chapter 8. One can supply multiple input files in the case of the HTML generation. For Latex, only one file can be used. The name of an input text file should end with `.ok`.

7.2 Source files.

`Source files` are the files used in including partial codes. The use of them is described in sections 8.8 and 8.7. Currently, files with extention `.rb` are recognized as Ruby source file, `.c` C program, `.C` and `.cc` C++ programs. Support on C/C++ programs is rather primitive and might not serve your need.

7.3 Options

`--directory dirname`

This option makes the generate HTML file and additional image files etc to be moved to directory `dirname`. Option argument `dirname` must exist. If directory `dirname` does not exist, it is created.

Note that this option is ignored if `--tolatex` option is specified.

`--keep-dot-files`

This option is retained for backward compatibility. It has no effect.

`--reuseoutput`

This option controls if the output of inline commands (see section 8.9 for details) will be reused from previous run of acsdoc. If this option is not specified, all commands are newly ran on the fly. If this option is specified, and if the command appears in the same line of the text input file in the same form, the output of previous ran is reused.

`--tolatex`

Generate a Latex output instead of HTML. Default is HTML. If this option is specified with multiple text input files, the result might not correct. Latex mode is meant to be used with single input file.

If you want to process multiple input files for generating single Latex file, just create one input file by using `cat`.

7.4 Output files.

By default, one HTML file is created for each of one text input file. If `--tolatex` option is specified, one Latex file is created for one text input file, and you are not supposed to give multiple input files.

There are quite a few other files generated. The image files will be stored in `.imgs` directory. Thus, if you have multiple input files which are processed separetely, *i.e.,* if you do

```
% acsdoc.rb text1.ok
% acsdoc.rb text2.ok
```

In one directory, The content of `.imgs` directory created in the first command is overwritten by that of the second command. To avoid this, use `--directory` option (see section 7.3 for more details) to put output files to subdirectories.

Acsdoc also create a CSS file, `.acsdoc-style.css`, which is referenced from HTML file. By default, the content of this CSS file is the same for any output, but it would change when the version of acsdoc.rb changes.

Finaly, acsdoc creates a number of fragment files from source files specified in the command line. For example, for file `foo.rb`, there will be a number of files with name starting with `.foo.rb` (since the name of generated files start with a ".", they do not appear when you do normal `ls` without `-a` option).

7.5 Examples

```
 % acsdoc.rb introduction.ok
```

createss a single HTML file `introduction.html`.

```
 % acsdoc.rb -d documents introduction.ok
```

creates a single HTML file `introduction.html` and store it to directory `documents`.

```
 % acsdoc.rb test.rb segmentsample.rb  introduction.ok
```

process the source files `test.rb` and `segmentsample.rb` to prepare fragment files used in introduction.ok, and then creates an HTML file.

```
 % acsdoc.rb  introduction.ok sample.ok
```

creates two HTML files, introduction.html sample,html, and add navigation links to these HTML files. The section (or chapter) numbers are as if these two HTML files are part of a single document.

Chapter 8

Tour over acsdoc funtionalities.

Markups in acsdoc is largely similar to that of Rdoc or RD, but not exactly the same. Here we overview what is available with acsdoc.

8.1 Sections.

One can start a new section (or subsection or chapter) by "=" (multiple "=" such as "==" or "===" result in deeper level, like 1.1 or 1.1.1)

The following is a sample for deep sections.

```
= First level section

This must be section 1

== 2nd level section

Here is section 1.1

=== 3rd level section

Here is section 1.1.1


= More sections

Here is section 2.
```

Here[1] is the created HTML file.

Up to five levels are supported, at least with HTML document. Number of levels available in Latex document depends on what is available on Latex. By default it is three.

8.2 Itemized list

What you can do with

```
<ul> or <ol>
```

in HTML, or \begin{itemize} or \begin{enumerate} in Latex, you can do in a simpler way.

For example,

```
Sample list
* Item 1.
  More text for Item 1.
* Item 2
  More on Item 2.
```

gives

Sample list

- Item 1. More text for Item 1.
- Item 2 More on Item 2.

A nested list can be made in the following way

```
Nested list
* Item 1.
  More text for Item 1.
  * nested item 1
  * nested item 2
* Item 2
  More on Item 2.
```

[1] `examples/sectionssample.html`

gives

Nested list

- Item 1. More text for Item 1.
  - nested item 1
  - nested item 2
- Item 2 More on Item 2.

```
 Numbered list
 1. Item 1
 2. Item 2
```

gives:

Numbered list

1. Item 1
2. Item 2

Note that the identifier for numbered list is numnber + ".". The number itself is not used in actual numberiing. Thus,

```
 Wrongly numbered list
 1. Item 1
 1. Item 2
```

gives

Wrongly numbered list

1. Item 1
2. Item 2

8.3 "as is" text

Text lines which start with a space, where this space is not followed by *, -, or a number + ".", appear as is.

Example:

```
  This is as-is text
```

This can be used to show program list etc.

8.4 Holizontal line

Three or more ”-” characters

```
---
```

will be converted to

Note that it should start at first column. If any space is before ”-”, it becomes ”as is” text.

8.5 Including file

```
#:include: test.rb
```

(without "#") gives:

```
## test.rb
  def test
     p test
  end
test
```

Note that this is exeption for the as-is text, since this `:include:` directive is interpreted even when it appears with preceeding space characters. Also, space characters before the :include: directive are added to each line of the included file. Thus

```
             #:include: test.rb
```

(without "#") gives

```
             ## test.rb
               def test
                  p test
               end
             test
```

8.6 Including program listing

You can use :inccode: in place of :include:. This may shows the included text in slightly different way.

Include:

```
## test.rb
  def test
     p test
  end
test
```

Inccode:

```
## test.rb
  def test
     p test
  end
test
```

In HTML, currently there is no difference. In Latex, :inccode: gives two horizontal lines marking the included code.

8.7 Including functions from source files

For C/C++ or Ruby sources, an automatic way to include one function from source code is provided. In Ruby, to include the listing of function buz fom class (or module) Bar in file foo.rb, you can write

```
##  :include: .foo.rb+buz+Bar
```

For exaple, "`:include:  .acsdoc.rb+wordmarkup+Acsdoc`" gives

```
 def wordmarkup(instr)
   @@wordreplace.each do |x| instr.gsub!(x[0]) do |word|
       $1 + x[1]+ " " + $2 +x[2] + $3
     end
   end
   instr
 end
```

If `buz` is the only function with that name in that file, or if buz is the top-level function, you can omit the class name as

```
##  :include: .foo.rb+buz
```

The names of source files should be given to acsdoc.rb as command-line arguments. They should appear before real .ok files in the argument list.

8.8 Including code flagments.

It is also possible to inclde a specified region from a source file. The region is (in the source file foo.rb) marked by `"# :segment start: bar"` and `"# :segment end: bar"` (here, bar can be some arbitrary name, without space or other special characters), and is included by

```
## :include: .foo.rb-bar
```

For example, if the source file segmentsample.rb is the following:

```
class Test
  def test
     p "test called"
  end
end

# :segment start: body
a= Test.new
a.test
# :segment end: body
```

By

```
 #  :include: .segmentsample.rb-body
```

We can get the following:

```
 a= Test.new
 a.test
```

8.9 Including the output of some program

There are followng seven directives to run commands

- :output:
- :command:
- :commandoutput:
- :commandinput:
- :commandinputoutput:

- :commandinputoutputnoecho:
- :commandinputoutputinteractive:

and one additional directive

- :prompt:

The directives to run commands can have variations with ”save”, like

- :commandoutput:
- :commandoutputinputoutput:

When these ”save” variations are used and acsdoc is invoked with ”–reuseoutput” option, acsdoc.rb look for the output of the same command previously executed (from its hidden data directory), and if the same command line is found at the same location of the input .ok file, corresponding output is taken from the saved result of previous run of acsdoc.rb.

Directive :output: echo foo

gives

```
foo
```

Directive :commandoutput: setenv LANG C ; date

```
|gravity> setenv LANG C ; date
Thu Sep 13 23:07:54 JST 2007
```

Directive :command: echo test

gives nothint as output, but it is stull executed. Thus, it can be used to do whatever things you like.

Directive :commandoutput: echo test gives

```
|gravity> echo test
test
```

Directive :commandinput: cat > aho END requires actual input data followed by ”END”, like

```
#  :commandinput: cat > aho END
#aaa
#bbb
#ccc
#END
```

Running this (without #) shows the input and in this case create a file "aho"

You can check if the file "aho" is made by:

```
#  :commandoutput: ls -al aho; pwd
#  :commandoutput: cat aho
# :command: rm aho
```

The result is:

```
|gravity> ls -al aho; pwd
-rw-r--r--    1 makino   makino         12 Sep 13 23:07 aho
/home2/makino/acs/kali/vol/documentation
|gravity> cat aho
aaa
bbb
ccc
```

Directive :commandinputoutput: cat END is similar to :commandinput:, but shows the result in text.

```
# :commandinputoutput: cat END
# aaa
# bbb
# ccc
# END
```

This (without #) gives you

```
|gravity> cat
aaa
bbb
ccc
aaa
bbb
ccc
```

Finally, :prompt: xxx> changes the prompt to ”xxx>”.

Thus,

```
# :prompt: yebisu>
# :commandoutput: echo test
```

gives

```
yebisu>echo test
test
```

The ”interactive” variant shows the input data at the location of corresponding ruby ”gets”. It works only with a ruby program which uses ”gets” function to read input from STDIN. Here is one example:

```
 yebisu>ruby testinteractive.rb
 enter x:1
 enter y:2
 1.0   2.0---
```

This is made with:

```
  : commandinputoutputinteractivesave : ruby testinteractive.rb END
  1
  2
  END
```

The file testinteractive.rb is

```
STDERR.print "enter x:"; x =gets.chomp.to_f
STDERR.print "enter y:"; y =gets.chomp.to_f
print x, "   "
print y
```

8.10 Boldface, italic, and typewriter font

Boldface, *italic*, and `typewriter font` are available. This part is generated from:

```
<b>Boldface</b>, <em>italic</em>, and <tt>typewriter font</tt>
```

For single word (without no space), you can use a more compact form

```
Sample *boldface*, _italic_, and +typewriter+ fonts.
```

which will look like:

Sample **Boldface**, *italic*, and `typewriter` fonts.

These markups (probably) do not work within listing. It should work with itemized list.

- **boldface** in an item.

8.11 Inline tex code.

Any tex code fragment can be embedded using the following form

```
<tex> tex codes </tex>
```

It can span to multiple lines. For example,

```
Here, <tex>$a= b$</tex> is a sample inline tex code.
```

gives

Here, $a = b$ is a sample inline tex code.

Since the use of tex codes is mostly to embed math formulae, one can write

```
<$ tex codes$>
```

instead of

```
<tex>$ tex codes $</tex>
```

Thus,

```
Here, <$a= b$> is a sample inline tex code.
```

gives

Here, $a = b$ is a sample inline tex code.

8.12 Numbered equations

The following form:

```
#  :equation:
# \label{equationlabel}
# a=b.
```

(without "#") gives

$$a = b. \tag{8.1}$$

The texts after :equation: directive (untill a blanck line) will be processed in Latex equation environment. Equation numbers are maintained within acsdoc. Anything which can be written in Latex equation environment can be used. To refer to the above equation, you can write

```
   equation ref(equation label)
```

which gives "equation 8.1." The label cannot contain space.

Latex eqnarray environment can be used as

```
# :eqnarray:
# \label{arrayeq}
# a &=& b,\nonumber\\
# c &=&d.
```

$$\begin{array}{rcl} a & = & b, \\ c & = & d. \end{array} \tag{8.2}$$

Note that the equation number counter of acsdoc assumes that only one number is used even when eqnarray is used. Thus, if you do

```
# :eqnarray:
# \label{arrayeq}
# a &=& b,\\
# c &=&d.
```

You get:

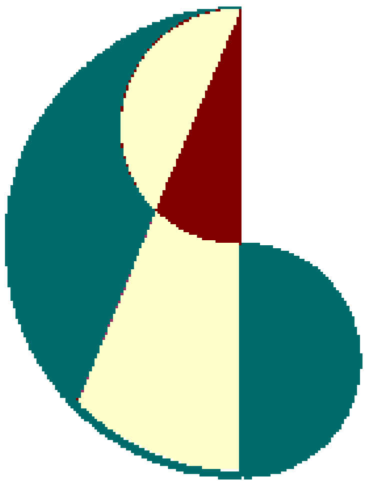

Figure 8.1: The ACS logo

$$a = b, \tag{8.3}$$
$$c = d. \tag{8.4}$$

However, the next equation will have wrong number like:

$$E = mc^2. \tag{8.5}$$

8.13 Figures

As in latex, one can make numbered figures. The syntax is

```
# :figure: sample.gif 5cm acslogo
# The ACS logo
```

Which gives figure 8.1.

Here, *sample.gif* is the name of the image file, which can have any format which is understood by "convert" command of ImageMagick. *5cm* is the horizontal size of the image (meaningful in Latex conversion only), and *acslogo* is the label. The text in the next and following lines, untill a blank line appear, become the figure caption. Thus, this figure can be refereed to by

```
Figure ref (acslogo)
```

(without space between ref and (acslogo)). Here is reference to this figure: figure 8.1.

Note that the size of an embedded postscript file, say foo.eps, may be too small for the html output. In such a case, you can for example double the size of the figure by converting it to a gif file, with the following command:

```
convert -density 200x200 foo.eps foo.gif
```

8.14 More on references

A section (or chapter) can be labeled by the :label: directive as

```
 #  :label: label
```

For examle, this section is labeled as

```
# :label: moreonreferences
```

And can be referenced as

```
# Section ref (sect:moreonreferences) is this section.
```

Here is the output:

Section 8.14 is this section.

8.15 Multiple input files

When acsdoc is applied to multiple input (.ok) files, multiple HTML files (one for one .ok file) is created, but each file will have navigation links to ”previous” and ”next” files. The order of the files is simply the order given in the command line argument.

The navigation link has ”Up” entry, which by default does not point to an URL. It can be specified by setting a value in the initialization file as

```
 @@toppagefilename= "some_file_name"
```

For example,

```
 @@toppagefilename= "../index.html"
```

would point to `index.html` in the parent directory.

8.16 Table of contents

The :tableofcontents: directive creates the table of contents. Example:

```
 # :tableofcontents:
```

It should appear as only word in one line.

8.17 Links to external URLs.

One can use <web> tag to make links to external URL. For example,

```
 < web >http://www.artcompsci.org|ACS homepage</web>
```

(whithout spaces) creates a link: *ACS homepage*[2].

If you specify just like <web>name</web>, it is assumed same name is specified for text and url, like `acsdoc.rb`, which is generated from

```
 < web >acsdoc.rb< /web >
```

(whithout spaces).

8.18 Inline image

If you want to have an inline image, you can use the form

```
  link: image_file
```

(without space after ":"). For example,

```
  link: sample.gif
```

gives

[2]`http://www.artcompsci.org`

Note that ”link: ...” should appear as single line without other words.

Note that here the link is a direct link to the image file. Thus, if you move the generated HTML file by ”–directory” option or by hand, you need to guarantee that image files are in the correct location.

8.19 Comments.

Lines starting with `#` in the first colums are treated as comments and not further processed.

8.20 References

As in latex with natbib style file, one can make two types of citetation in the text, such as

```
   < citet >MakinoHut2006</ citet>
   < citep >MakinoHut2006</ citep>
```

(without whitespaces). Actual reference entries should have the form:

```
  < REF >

  Aarseth, S. J. | 1963 |Aarseth1963| MNRAS, 126, 223

  Makino, J.; Hut, P. | 2006 |MakinoHut2006| ACS

  Barnes, J. E.; Hut, P. | 1986 |BarnesHut1986|Nature, 324, 446

  < /REF >
```

Each of the reference entries consists of four fields, separated by ”—”. One entry can consist of multiple lines, since the separator between reference entries are two consective newlines.

The first field of a reference entry is the list of authors. Authors must be separated by ”;”, and the familiy name, with single ”,” at the end, should appear first.

The second field is the publication year.

The third field is the tag, to be refered in the text.

The 4th fieled is the actual bibiliographic data.

Here is the citettion to Knuth: Knuth (1992).

8.21 Examples.

All of the functionalities described in this document is used in this document. So the best place to look at examples is the *input file itself*[3].

[3] `ch05.ok`

Chapter 9

Initialization file

Initialization file is searched in the order of \$ACSDOCINITRC, ./.acsdocinitrc, and ∼/.acsdocinitrc. The things you can write in the initialization file is Ruby statement. A typical way to use is something like:

```
# acsdoc initialization file
print "Loading the initialization file for ascdoc\n"
@@addtional_preambles_for_inline_tex = "\\usepackage{epsf}"
@@addtional_commands_for_inline_tex = "\\input macros"
```

This one allows the use of epsf package (style file), and macros.tex is included after \begin{document}.

Chapter 10

Wish lists

- tables

Chapter 11

Tips

11.1 Change the document class for Latex

To change the document class, supply appropriate value to variables `@@basic_preambles_for_tex` and `@@headers` in the initialization file (.acsdocinitrc). For example,

```
@@basic_preambles_for_tex = <<END
\\documentclass{article}
\\usepackage{graphicx}
END
@@header=[
"chapter","section","subsection","subsubsection",
"subsubsubsection", "subsubsubsubsection"]
```

is appropriate for article class. First item of `@@header` is not used. Default is ”book” class.

Chapter 12

Known problems

12.1 tags in listing mode

Most tags show themselves up correctly in listing (lines with preceeding space) mode, but

```
<tex>
```

and

```
<web>
```

are two exeptions, at least with current (as of Nov 21 2005) version of acsdoc.rb. You need to type

```
\<tex>
```

or

```
\<web>
```

even in listing mode.

12.2 section header just after the figure

section header (=== etc) just after figure entry seems to be processed incorrectly. If you add one more newline between figure data and section header, it works fine. (May 2 2006)

Chapter 13

Sandbox

Hmm, $, ?, _ ...

Are, does blank line still work as new paragraph?

This should be a new paragraph.

This also.

```
   This is as-is text with some tags <xxx>, <tt>, <b>, \begin{xxx}
```

Hmm.
nil nil nil nil nil

Bibliography

Knuth, D. E., 1992, *Literate Programming*, Center for the Study of Language and Information - Lecture Notes