Numbas

CHRISTIAN PERFECT, BILL FOSTER,
ANTHONY YOUD, STACEY ASTON

School of Mathematics & Statistics
Newcastle University
Newcastle upon Tyne
United Kingdom

Thu 04 Aug 13:00:27 2011

Contents

Chapter 1

Introduction

Numbas is a web-based e-assessment system developed at Newcastle University. It consists of a set of tools which produce SCORM-compliant exam packages (`http://scorm.com/`).

1.1 Features

- Simple installation. Unpack Numbas anywhere on your computer.
- Web-based, so it can run on a wide range of computers. In particular, Windows, Mac and Linux operating systems are supported, under Internet Explorer 8+, Mozilla Firefox 2.5+, Google Chrome and Safari.
- Runs entirely on the client computer in JavaScript. This means there is no backend server, and exams can be deployed in a variety of locations, for example, in Virtual Learning Environments (VLEs) or Learning Management Systems (LMS) such as Blackboard, DVDs, and even stand-alone on the web.
- Extensive support for questions of a mathematical nature. Answers to questions can be complicated mathematical expressions.
- Support for LaTeX, using MathJax (`http://www.mathjax.org/`).
- Write questions using simple markup with any text editor.
- Questions can be fully randomised.
- Because the exam runs in the browser, rich content such as videos and interactive graphs can be included.
- SCORM 2004 standards compliant, so the exams can be run in a VLE which supports this standard.
- Support for themes to change the look and user interface of exams.

- Support for extensions to add new features, such as new question types, or mathematical and statistical libraries.

1.2 System requirements

Numbas has modest system requirements, whether you are running an exam or if you want to author questions.

- A web browser on Windows, Linux, or Mac. Numbas has been tested and is known to work with Internet Explorer (versions 8 and higher), Mozilla Firefox (versions 2.5 and higher), Google Chrome, and Safari.
- If you wish to author questions, you will need a text editor, and an installation of the Python programming language (`http://www.python.org/`), version 3.1 or higher.

1.2.1 Important information about MathJax

Numbas uses MathJax to render mathematics using native web fonts, which are supported by all modern browsers. Due to a security setting in Mozilla Firefox, web fonts cannot be loaded by the exam when it is run locally within the browser (i.e. not from a web server), and MathJax falls back to rendering mathematics using images, which is slower. (The layout of content on the page is not affected.) This can be remedied, however, by installing a local copy of MathJax. See appendix A on SCORM (specifically §A.2).

1.3 Numbas structure

Numbas can be unpacked anywhere on your system, and does not require any further installation. Once unpacked, the directory structure is as follows.

bin: Python scripts for compiling an exam.

doc: The documentation for Numbas.

exams: Files which describe your exams.

extensions: You can extend the functionality of Numbas by writing JavaScript files and putting them in this directory.

output: Once an exam is compiled, the output goes in this directory.

runtime: The core JavaScript files which must be included with every Numbas exam.

scormfiles: Additional files necessary to create a SCORM-compliant exam — see appendix A.

themes: The look and feel of an exam can be customised by creating themes. Themes should go in this directory.

1.4 Organisation of this manual

The manual is designed to be read from beginning to end. Each chapter builds on the last. Chapters should not be skipped, otherwise certain concepts are unlikely to make sense.

The next chapter — chapter 2 — is intended as a quick start guide, by the end of which you should have a fully functioning exam, using the simple example exam provided. Subsequent chapters explain Numbas in detail.

- Chapter 3 describes the basics of the markup used to construct exams.
- Chapter 4 describes the `exam` object, which is used to define all aspects of an exam.
- Chapter 5 describes the `question` object, which is used to define questions with an exam.
- Chapter 6 describes the `part` object. Each question in your exam must have a part; this object is used to define the properties of a question part.
- Chapter 7 describes content blocks. These are used for prompting students to do something, and for the display of various aspects of the exam.
- Chapter 8 describes the part types that can be used within a question.
- Chapter 9 describes the Judged Mathematical Expression syntax, which is used to define variables, functions, and the answers to the JME part type.
- Appendix A explains how to produce SCORM objects, which could be included in a VLE, such as Blackboard.

Chapter 2

Quick start guide

This chapter is intended as a quick start guide for those who are unfamiliar with Numbas.

We shall describe the basics of constructing an exam and running it, without going into significant technical details. By the end of this chapter you should have a simple, yet fully functioning, exam.

2.1 Creating the example exam

All aspects of an exam, including questions, variables, marks, answers, etc., are described in a `.exam` file, which is a plain text file written in a simple markup language.

In the `exam` directory you will find a file called `example.exam` (see listing 2.1), which describes a simple exam with one question containing two parts. The example exam also includes an image in the question statement. This image is stored in the resources directory for the exam.

Listing 2.1: Example exam.

```
{
	// Define exam properties
	name: My First Exam
	duration: 1800

	resources: [example_files]

	// Define questions
	questions: [
		{
			name: Addition and subtraction
			statement: """
				Answer the following questions on addition and subtraction.
				<img src="resources/example_files/Love_math_1.jpg"/
```

```
            """

            variables: {
                a: "random(1..9)"
                b: "random(1..9)"
                c: "random(1..9)"
                d: "random(1..9)"
            }

            parts: [
                {
                    type: numberentry
                    prompt: "What is $\var{a}+\var{b}$?"
                    marks: 1
                    answer: "{a+b}"
                }

                {
                    type: numberentry
                    prompt: "What is $\var{c}-\var{d}$?"
                    marks: 1
                    answer: "{c-d}"
                }
            ]
            advice: """
                In part 1 simply add the two numbers, so that
                \[\var{a}+\var{b} = \var{a+b}.\]

                In part 2 simply subtract the two numbers, so that
                \[\var{c}-\var{d} = \var{c-d}.\]
            """
        }
    ]
}
```

We shall explain the various aspects of the markup in chapter 3 and subsequent chapters, but for now, we can run this file through the exam interpreter to “compile” the exam, and produce a collection of HTML and JavaScript files.

In a command prompt change to the directory where Numbas is unpacked, then run[1]

```
python bin/numbas.py exams/example.exam
```

[1] On some systems it is possible for multiple versions of Python to co-exist. If version 3.1 is not the default on your system, then you may need to use another command instead of `python`. For example, on Debian and Ubuntu you might need to use `python3.1`, provided that the package *python3* is installed.

When the compilation finishes, you will see a message telling you that the exam has been created in `output/example`.

Now, from a web browser, open the file `output/example/index.html`. You should then see a screen which looks like figure 2.1. Information about the exam,

Figure 2.1: The first screen you see when opening the example exam.

e.g. name, number of questions, available marks, time limits, etc., is shown on this page.

Click the *Start* button to begin the exam, and you will see the first question presented to you, as in figure 2.2. Each question of an exam is shown on a separate page,

Figure 2.2: The first question of the exam.

and navigation between questions is possible either by clicking the *Previous*/*Next* buttons, or by using the question list toward the left hand side of the page. In the question list, the up and down arrow icons, or the mouse scroll wheel, can be used to scroll through questions if there are too many to show.

The *Reveal* button is used to show a "worked solution" to the current question, written by the question author.

The question statement is shown in the green box at the beginning of the question, and question parts are shown underneath.

Enter the answers in the boxes. You can then either press the *Submit all parts* button at the bottom of the page, which will mark the entire question in one go or, alternatively, click the *Submit part* button next to each part, to have each part marked individually.

When a question or part is marked, the score for each part is displayed, the total question score is displayed at the bottom of the page, and the score in the question list is updated.

In addition to the scores, an icon showing a green tick, blue percent sign, or red cross is shown. The green tick means the question or part is correct, the blue percent sign means the question or part is partially correct, and the red cross means the question or part is incorrect. This lets you see at a glance where you have lost marks, if any.

Now click the *End Exam* button. You are presented with a screen similar to figure 2.3. This page shows the exam details again, and a breakdown of your per-

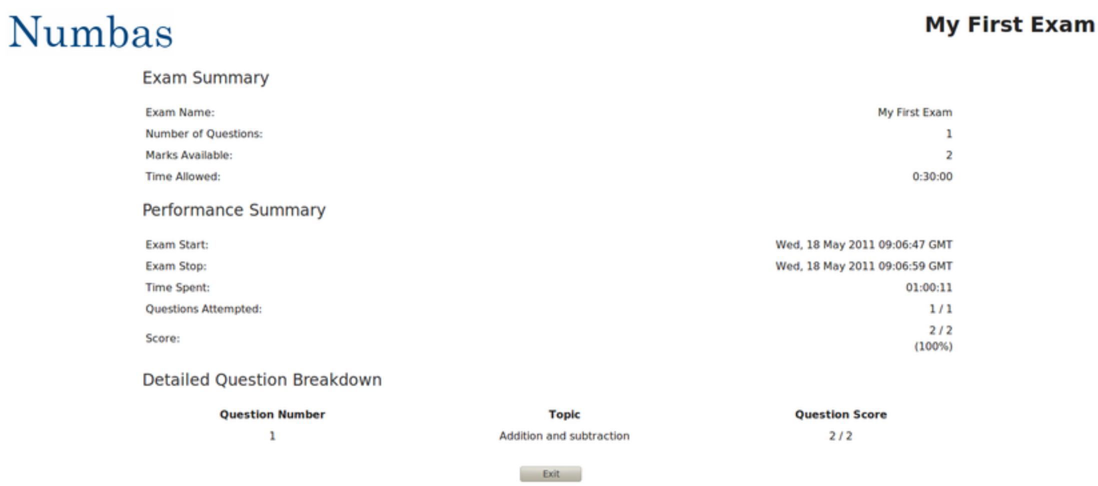

Figure 2.3: The performance summary page of the exam.

formance in the exam. Finally, clicking the *Exit* button will finish the exam, and you can then close the browser window.

2.2 Creating your own exams

As you can see from the example, creating an exam is as simple as writing a plain text file, compiling it with the `numbas.py` interpreter, and opening the resulting files in a web browser. A more complicated exam — `mathssample.exam` — demonstrating many aspects of the system, is also provided in the exams directory.

Our recommendation would be to start off by using `example.exam`, and modifying it to suit your needs. Since questions are self-contained objects, they can be reused simply by copying and pasting from other exam files.

2.2.1 numbas.py

This file is the interpreter that translates the markup in a `.exam` file into a usable exam. Running `numbas.py -h` will show the help message, explaining the options available.

Chapter 3

The exam format

As we briefly explained in chapter 2, an exam is constructed by writing a plain text `.exam` file, which consists of simple markup, describing all aspects of the exam. This chapter describes the fundamental aspects of the markup.

3.1 Data types

The markup is a JSON-like format (`http://www.json.org/`), with minimal punctuation, and three data types: objects, arrays, and literals. The language is purely declarative — there is no control code.

3.1.1 Arrays

Arrays take the form

```
[ data1, data2, ..., dataN ]
```

or, alternatively, across multiple lines

```
[
  data1
  data2
  ...
  dataN
]
```

in which case commas can be omitted.

3.1.2 Objects

Objects are lists of key–value pairs, and take the form

```
{ key1: value1, key2: value2, ..., keyN: valueN }
```

where each value can be of any data type, including other objects. Again, it is possible to write an object over multiple lines, optionally omitting the separating commas:

```
{
  key1: value1
  key2: value2
  key3: {
    key4: value4
    key5: value5
  }
  keyN: valueN
}
```

3.1.3 Literals and quoting

A literal is a text string, which may or may not need to be quoted. An unquoted literal ends with a new line or a comma, e.g.

```
{
  name: My First exam, age: 25
}
```

Properties of exams or questions or part types, or anything which is not marked as content, will very rarely need to be quoted. Pure number values, and boolean values also do not need to be quoted. Some examples of when to use quotes are shown below.

Single-quotes: Used only when defining string variables (variables are explained in later chapters). Consider defining a string variable `hello` as below. In this case, the value must be enclosed in single-quotes.

```
variables: {
  a: 'Hello'
}
```

Double-quotes: Used when the property value includes a new line or a comma. Content which includes braces or square brackets should also be enclosed in double-quotes. The value below includes both a new line and a comma, therefore double-quotes must be used.

```
{
  prompt1: "Hello, this is a
    long string"
}
```

Triple double-quotes (`"""`): Used when the property value itself includes double-quotes. The value below itself includes a double-quote, so it must be enclosed in triple double-quotes.

```
{
  prompt2: """
   Bob said "Hello" to Fred
      """
}
```

Chapter 4

The exam object

The fundamental data structure is the `exam` object, containing all the data necessary to define an exam. This chapter describes the `exam` object, and the data types and objects which can be used within it.

By default, all object properties are optional, and if they are not present in the exam file, they will take their default values. Table 4.1 describes the properties of the exam itself.

4.1 Event object

One of the properties of an object might be an `event` object. The `event` object defines what action to take when a particular event occurs. Table 4.2 describes the properties of the `event` object.

The `action` property can take various values, depending on the context under which the `event` object appears. The permissible values are described elsewhere, where an `event` object is a property of another object.

4.2 Navigation

It is possible to control how and when students are allowed to move between questions, for example, whether they are allowed to return to a question once they have completed it, whether they can jump between questions at will, or whether they must complete a question before moving on to another one.

Navigation is an `exam` property, and is controlled by the `navigation` object. A simple example might be the following:

```
navigation: {
  reverse: false
  browse: false
}
```

which means that the student is not allowed to return to a previous question (`reverse`), and is not allowed to jump between questions at will (`browse`), so the student can only move forward through the exam. By default, both of these properties are `true`.

Table 4.3 describes the properties of the `navigation` object.

4.2.1 Navigation events

The `onadvance`, `onreverse`, and `onmove` properties are `event` objects (see §4.1), which determine what action should be taken when the student attempts to navigate away from the current question, having not completed the current question. They take the default value of

```
{ action: none, message: "" }
```

so the student is allowed to move back or forward to another question, or jump between questions in the question list unhindered.

The permissible values for the `action` property of the `event` objects are:

- `none`: do nothing;
- `warnifunattempted`: warn the student that they have not completed the current question, but move on anyway;
- `preventifunattempted`: warn the student that they have not completed the current question, and do not allow the student to move on to the next question.

4.3 Timing

If the assignment is timed, by setting the exam property `duration` to a non-zero value, then the `timing` object can be used to define warning messages shown to the student when a certain amount of time is left, or when the time has expired. Table 4.4 describes the valid properties of the `timing` object.

4.3.1 Timing events

The `timeout` and `timedwarning` properties are `event` objects (see §4.1), which determine what action to take when a student has run out of time, or when there is a certain amount of time left (this is fixed at five minutes at the moment). The default for both properties is

```
{
  action: none
  message: ""
}
```

so no action is taken in either case. The valid values for the `action` property are

- `none`: do nothing;
- `warn`: display the warning `message`.

4.4 Feedback and advice

Numbas includes the facility to provide feedback and advice to the student taking the exam. The most common use of feedback is to provide the student with a fully worked solution to a question. The student can see this feedback by pressing the *Reveal* button (see the screen shot in figure 2.2). You can control when and how the student sees this feedback, by using the `feedback` object, which is an `exam` property. All valid properties of the `feedback` object are detailed in table 4.5.

4.4.1 Restricting advice

By default, the student is allowed to reveal the advice to a question at any time, by pressing the *Reveal* button. Doing so will remove all marks awarded for the current question (the student is warned that this will happen). To prevent the student being able to see the advice, set the `allowrevealanswer` property to `false`, e.g.

```
feedback: {
  allowrevealanswer: false
}
```

This will remove the *Reveal* button from the page.

4.4.2 Restricting mark information

By default, the student can see how many marks are available for the entire exam, and how many marks have been awarded so far. This behaviour can be altered by setting the `showtotalmark` and `showactualmark` properties, e.g.

```
feedback: {
  showtotalmark: false
  showactualmark: false
}
```

This will prevent the student from being able to see how many marks have been awarded, and how many are available for the exam.

4.4.3 Restricting the answer state

By default, the student will be shown whether the submitted answer is correct (green tick), incorrect (red cross), or — in the case of multiple submit boxes — partially correct (blue percent sign).

You can alter this behaviour with the `showanswerstate` property, e.g.

```
feedback: {
  showanswerstate: false
}
```

In this case, the student receives no indication of whether the submitted answer is correct.

Note: Be careful of how you use `showanswerstate` and `showactualmark`. If you set the former to `false`, but the latter to `true`, then students can still see whether their answers are correct by looking at the current mark total. In contrast, it is reasonable to set `showanswerstate` to `true`, but `showactualmark` to `false`, when you want to show whether the answer is correct, but not how many marks are gained.

4.4.4 Advice object

The `advice` object defines under which circumstances advice is shown to the student. Advice is a property of the `feedback` object. Table 4.6 describes the properties of the `advice` object.

As a property of the `feedback` object, the `advice` object determines when advice is shown to the student. The actual advice content shown to the student is set with the `advice` property of the `question` object — see §5.5.

4.4.5 Feedback examples

There are a number of feedback and advice options, and choosing the correct ones for the behaviour you desire can be tricky at first. Some examples are listed below.

4.4.5.1 All feedback

Set no feedback options explicitly; the default options will turn on all forms of feedback except threshold advice.

4.4.5.2 No feedback

Use a `feedback` object with the following properties set to `false`.

```
feedback: {
  showtotalmark: false
  showactualmark: false
```

```
    showanswerstate: false
    allowrevealanswer: false
  }
```

4.4.5.3 No Reveal button but automatic feedback when student scores less than a threshold

Use a `feedback` object with `showanswerstate` set to `false`. Also use the feedback `advice` object with `type` set to `threshold`, and `threshold` set to the threshold percentage.

```
feedback: {
  allowrevealanswer: false
  advice: {
    type: threshold
    threshold: n
  }
}
```

4.4.5.4 Reveal button and automatic feedback when student scores less than a threshold

Use a `feedback` object with an `advice` object. Set `type` to `threshold`, and `threshold` to the threshold percentage.

```
feedback: {
  advice: {
    type: threshold
    threshold: n
  }
}
```

4.5 Resources

If you want to include videos or images in your exams, then they must be part of the collection of files which make up the compiled exam.

This can be achieved by setting the `resources` property of the `exam` object to an array of directories or file names (relative to the directory in which the `.exam` file resides), which are to be included. These directories and files then appear under the `resources` directory of the compiled exam.

As an example, suppose you want to include the images `graph.png` and `ball.png` in an exam. You should set `resources` as

```
resources: [graph.png, ball.png]
```

Then, when you need to make reference to `graph.png` in a question, for example, the path to the image is `resources/graph.png`. Images and videos can be included in `content` blocks, which are described later in chapter 7.

4.6 Simplification

You can use standard in-built simplification rules or user-defined rules in order to simplify, rearrange or display mathematical expressions for feedback to the users. This is particularly useful when you are supplying complete solutions. More details can be found in chapter 7, sections 7.1.2, 7.1.3 and 7.1.4.

Property	Description	Default value
`name`	The name of the exam as it appears at the top of the page.	`Untitled exam`
`duration`	The time allowed for the exam, in seconds. A value of 0 means there is no time limit.	`0`
`percentpass`	The minimum percentage score to be classified as a pass.	`0`
`shufflequestions`	Determines whether the question order should be randomised.	`false`
`navigation`	An object defining navigation rules, i.e. how the student is allowed to move between questions (see §4.2).	`{}`
`timing`	An object defining the warning messages shown to the student when they have run out of time, or when a certain amount of time is left (see §4.3).	`{}`
`feedback`	An object defining feedback rules (see §4.4).	`{}`
`resources`	An array of directories or file names (relative to the directory containing the exam) to be included in the resources directory of a compiled exam (see §4.5).	`[]`
`questions`	An array of `question` objects (see chapter 5).	*no default*
`rulesets`	An object defining the user-defined rule sets to be used for simplification, rearrangement or display(see §7.1.2).	`{}`

Table 4.1: The valid properties of an exam. The `questions` property is required, as denoted by the italic typeface.

Property	Description	Default value
`action`	What to do when a particular event occurs (see below).	`None`
`message`	The message to display when an event occurs, providing `action` is not `None`.	*empty string*

Table 4.2: The valid properties of an event object.

Property	Description	Default value
showfrontpage	Whether to show the welcome page before beginning the exam. If `false`, the exam begins at the first question as soon as the page has loaded.	`true`
reverse	Whether the student is allowed to move to the previous question.	`true`
browse	Whether the student is allowed to move to any question, by using the question list.	`true`
onadvance	An `event` object describing what to do when a student attempts to move on to the next question, without completing the current question.	see §4.2.1
onreverse	An `event` object describing what to do when a student attempts to move to the previous question, without completing the current question.	see §4.2.1
onmove	An `event` object describing what to do when a student attempts to move on to any other question, without completing the current question.	see §4.2.1

Table 4.3: The valid properties of a navigation object.

Property	Description	Default value
timeout	An `event` object describing what to do when the student has run out of time.	see §4.3.1
timedwarning	An `event` object describing what to do when the student has five minutes left.	see §4.3.1

Table 4.4: The valid properties of a timing object.

Property	Description	Default value
`showactualmark`	Whether to show the student's score while the exam is in progress.	`true`
`showtotalmark`	Whether to show the total marks available for the exam, questions, and question parts.	`true`
`showanswerstate`	Whether to show the tick, cross, or percent sign when a student submits an answer.	`true`
`allowrevealanswer`	Whether the student is allowed to click the *Reveal* button to see the advice.	`true`
`advice`	An `advice` object defining under which circumstances advice is shown (see §4.4.4).	`{}`

Table 4.5: The valid properties of a feedback object.

Property	Description	Default value
`type`	When advice is shown to the student. If the value is `onreveal`, then advice is only shown to the student when the *Reveal* button is pressed. If the value is `threshold`, then advice is shown when the student scores below `threshold`%.	`onreveal`
`threshold`	Reveal advice if the student scores less than this percentage on a single question.	`0`

Table 4.6: The valid properties of an advice object.

Chapter 5

The question object

The `question` object defines the questions that make up an exam, and is a property of the `exam` object. Each question in an exam is made up of a number of parts, defined by the `part` object, which is described in detail in chapter 6. Each `question` object can have any or all of the properties listed in table 5.1.

Property	Description	Default value
`name`	Name of the question.	`Untitled question`
`statement`	*(Content)* Question statement. Displayed above the question parts. Tells the student what the setup of the question is.	*empty string*
`advice`	*(Content)* The text to display when the student presses the *Reveal* button, or enters an incorrect answer — see §5.5.	*empty string*
`parts`	An array of `part` objects (see chapter 6).	`[]`
`variables`	An object defining the variables used in the question — see §5.2.	`{}`
`functions`	An object defining the functions used in the question — see §5.4.	`{}`

Table 5.1: The valid properties of a question. The `statement` and `advice` properties are `content` blocks. See chapter 7 for more information on `content` blocks.

5.1 Question parts

Each question in an exam has a question statement, which tells the student the setup of the question, and a number of question parts, which are defined in the `parts` property. The `parts` property is an array of `part` objects, described in detail in chapter 6.

5.2 Declaring variables

Question variables are declared in a `variables` object, which is a property of the `question` object. They can therefore be used in all question parts contained within the `question` object. Each variable takes the key–value form `name: expr`, where `name` is the name of the variable, and `expr` is an expression defining the variable. The expression will sometimes needed to be quoted, as explained in §3.1.3.

The expression `expr` uses a special *Judged Mathematical Expression* (JME) syntax. Here, we shall provide a brief explanation of this syntax. It is explained in greater depth in chapter 9.

5.2.1 Simple variables

The value of a variable can be something as simple as a number, e.g.

```
variables: {
  x: 5
  y: 3.7
}
```

It is also possible to reference other variables which have already been defined within the question, e.g.

```
variables: {
  x: 5
  y: 3.7
  a: x+y
  b: a*x-7.4
  c: x/2+b^2
}
```

where we have also used simple mathematical operators (`+`, `-`, `*`, `/`, `^`) to combine variables. Note that variable definitions cannot be cyclical, e.g. `a: b+2`, `b: a-2` is not permitted.

Variables are not limited to being numerical; they can also be strings, e.g.

```
variables: {
  surname: 'Bloggs'
  forename: 'Joe'
  name: forename+" "+surname
}
```

A string variable must be enclosed in single-quotes (double-quotes should not be used). To concatenate strings use the `+` operator, as in `name` above.

Note: Variables cannot be named `pi`, `e`, or `i`, since these names are reserved for the built-in constants $\pi = 3.14\ldots$, $\mathrm{e} = 2.71\ldots$, and imaginary i.

5.2.2 Complex variables

Numbas supports complex numbers, and complex variables can be defined and used as you might expect, e.g.

```
variables: {
  z1: 3+5*i
  z2: 1-7*i
  z3: z1^2
  z4: z1-3*z2
  z5: re(z1)
  z6: im(z3)
}
```

The variables `z5` and `z6` make use of the functions `re()`, and `im()`, which return the real and imaginary parts of a complex number, respectively. Many of the other available functions can also be applied to complex numbers. See §9.2 for a full list of valid mathematical operators, constructs, and functions.

5.2.3 Randomisation

The variables in the previous section are static, i.e. whenever the exam is run, these variables will always take the same values. Much more flexibility can be gained by defining random variables, so that values change each time the exam is run.

The syntax for a random variable is either

```
name: random(start..end[#step])
```

or

```
name: "random(a1,a2,a3,...,aN)"
```

The syntax `start..end#step` means the variable will be chosen discretely from between `start` and `end` in steps of size `step` (and `step` is optional). If the step size is zero, then the variable will be chosen continuously from the range. If the step size is not set, then it defaults to one. The syntax `a1,a2,...,aN` means the variable will take one of the values `a1` to `aN`. Random variables of any data type (e.g. number, string, boolean) can be defined. The values `a1,a2,...,aN` can themselves be variables that have already been properly assigned. Some randomisation examples are shown in table 5.2. As with the simple variable declarations in the previous section, it is possible to reference other random variables which have already been declared, e.g. `v9: 2*v1+random(3..7)`, `v10: "v2*random(2,4,12)/2"`.

5.2.4 Conditional expressions

It is possible to declare variables conditionally, i.e. a variable is assigned a value depending on whether a condition is true or false. Numbas supports `if` statements and `switch` statements.

Randomisation	Possible values
`v1: random(1..5)`	1, 2, 3, 4, 5
`v2: random(10..20#2)`	10, 12, 14, 16, 18, 20
`v3: random(0..0.5#0.05)`	0, 0.05, 0.1, 0.15, . . . , 0.5
`v4: random(0..0.5#0)`	Any real number in the range $[0, 0.5]$
`v5: "random(1,3,6,24)"`	1, 3, 6, 24
`v6: random(-9..9)`	-9, -8, . . . , 8, 9
`v7: "random(v1,v2,v3)"`	The assigned value of either `v1`, `v2`, or `v3`
`v8: "random('Cat','Dog','Bird')"`	"Cat", "Dog", or "Bird"

Table 5.2: Example random variables.

5.2.4.1 if statements

The syntax for an `if` statement is

```
name: "if(condition, expr1, expr2)"
```

So, if `condition` is `true`, then `name=expr1`, otherwise, `name=expr2`. More complex conditions can be built up by using the `and`, `or`, and `not` operators. The expressions can include any valid JME syntax, including further `if` or `switch` statements (see below). Some examples are shown in table 5.3.

Variable	Value
`amount`	`random(5000..15000#1000)`
`rate`	`"if(amount<10000,0.1,0.15)"`
`tax`	`amount*rate`
`x`	`random(1..5)`
`y`	`random(2..10)`
`z`	`"if(x<3 and y>7,random(-5..-1),x*y^2)"`

Table 5.3: Examples of if statements.

5.2.4.2 switch statements

The syntax for a `switch` statement is

```
name: "switch(cond1,expr1, cond2,expr2, ...,
                                    condN,exprN, defaultExpr)"
```

Any number of `condition,expression` pairs is allowed. Each condition is checked sequentially; when a condition evaluates to `true`, the variable is assigned the value of the corresponding expression, and subsequent conditions are not checked. If none of the conditions is true, then `defaultExpr` is used, if present.

The expressions can include any valid JME syntax, including further `if` or `switch` statements. See table 5.4 for some examples.

Variable	Value
`a`	`random(-5..5)`
`ans`	`"switch(a<0,10, a=0,3, a>0,-7, -99)"`

Table 5.4: Examples of switch statements.

5.2.5 Lists

A list is a finite collection of elements of any type. A list can be defined explicitly using square bracket notation, e.g.

```
a: "[1,2,3,4,5]"
```

or implicitly using the `repeat` function, which evaluates a given expression a given number of times, e.g.

```
a: "repeat( random(1..10), 5 )"
```

This example will produce a list of five numbers in the range $[1, 10]$.

A particular element of the list can be retrieved by using another form of the square bracket notation, e.g.

```
a: "[1,3,5,7,9]"
b: a[2]
```

where indexing starts at zero.

The `map` function applies a given expression to all elements of the list, e.g.

```
a: "[1,2,3,4,5]"
b: "map(x^2, x, a)"
c: "map(x^3, x, 1..3)"
```

Here, the variable `b` will be the list `[1,4,9,16,25]`. The definition of the variable `c` shows that `map` also works on ranges, so that `c` will be the list `[1,8,27]`.

Finally, the `abs` function can be used to obtain the number of elements in the list, e.g.

```
a: "[1,6,23]"
b: abs(a)
```

Here, `b` will be set to 3.

5.3 Using variables

Once variables have been declared, they can be used in most places within a question definition, in particular within part answers and `content` blocks (see chapter 7). To do this, enclose the variable name in braces. When the exam is displayed, the variable is evaluated according to its definition, and substituted in. Examples of variable usage are shown in table 5.5.

Usage	Result
`{a}`	2
`{b}`	3
`{a}+{b}`	2+3
`{a+b}`	5
`{3*a+2*b}`	12
`2*{a}+{3*b}`	2*2+9

Table 5.5: Examples of variable usage, having defined the variables `a: 2` and `b: 3`.

Note: There is one exception to using braces in this way, when formatting *displayed* mathematics using LaTeX. In this case, a slightly different syntax must be used — see §7.1.

5.4 Functions

As well as defining variables, it is possible to define your own functions, which can then be used in exactly the same way as the built-in functions, e.g. `cos(x)`, `exp(x)`, etc. . This enables you to write more complicated questions, than by defining variables alone.

5.4.1 Declaring functions

Just like variables, functions are declared in a `functions` object, as a property of a `question` object. Consider the following code which defines the factorial operator.

```
functions: {
  factorial: {
    parameters: [ [x,number] ]
    type: number
    definition: "if(x=0,1,x*factorial(x-1))"
  }
}
```

Each function has a name, a list of the parameters the function takes, the return type of the function, and its definition.

Note: There is already a built-in function to calculate the factorial, called `fact()`, so you do not need to write your own. This definition is used for illustrative purposes only.

5.4.1.1 Parameters

The `parameters` property defines the arguments of the function. Its value is an array of two-element arrays and it takes the the form

```
parameters: [ [arg1,type1], [arg2,type2], ..., [argN,typeN] ]
```

Where each `argi` is a dummy argument to be used in the function definition, and each `typei` is the type of the corresponding argument (valid types are described in §9.1).

5.4.1.2 Definition

The `definition` property defines what the function does. Any valid JME syntax is allowed here. Variables, and other functions which have already been defined, can be used within a function definition (notice how the example function even calls itself).

The resulting type of the calculations performed in the function definition must match the value of the `type` property.

5.4.2 Using functions

User-defined functions are called in exactly the same way as the built-in functions, e.g. `name(arg1,arg2,...argN)`.

5.5 Advice

When declared as a property of the `question` object, the `advice` property sets the content to be shown when the student presses the *Reveal* button. It is a `content` block (see chapter 7), and is usually used to provide a fully-worked solution to the question.

Chapter 6

The part object

The `part` object defines the parts that make up a question. It is a property of the `question` object. In addition to the properties listed in table 6.1, further properties are required, depending on the *type* of the part. Chapter 8 describes the supported part types, and the additional properties, in detail.

6.1 Part marks

There are three properties controlling the number of marks available for a part. The first is `marks`, which sets the maximum number of marks available for the part. The `minimummarks` and `enableminimummarks` properties are most useful when you have defined part steps — see §6.2.

Since it is possible for the student to lose marks by revealing steps, it is possible for the part to be negatively marked. For example, suppose there were 3 marks available for a part, and you decided that the student should lose 1 mark if they viewed steps. If the student did view the steps, and also answered the part incorrectly, then their net score for this part would be -1. To prevent this, you can set `minimummark` to some value, zero say, and also set `enableminimummark` to `true`.

6.2 Steps

Steps can be used as "intermediate" question parts, where you would like to provide the student with the option of taking extra steps to answer a question. Sometimes, the step may simply be informative, reminding the student of a formula, for example. Alternatively, it might be a series of sub-parts guiding the student to the final answer. Initially, the steps are hidden from the student. Clicking on the *Show steps* button will reveal the steps. You can also decide whether marks should be deducted if the student views the steps and can award partial marks if students give correct answers within steps.

Property	Description	Default value
type	The part type. Permissible values are `jme`, `numberentry`, `patternmatch`, `1_n_2`, `m_n_2`, `m_n_x`, `gapfill`, and `information`. Part types are described in detail in chapter 8.	*no default*
`marks`	The number of marks available for this part — see §6.1.	`0`
`minimummarks`	The minimum number of marks available for this part — see §6.1.	`0`
`enableminimummarks`	Whether there should be a minimum number of marks available for this part. Set in conjunction with `minimummarks` — see §6.1.	`false`
`prompt`	*(Content)* Text telling the student what they should do.	*empty string*
`steps`	An array of `part` objects, which the student can reveal. These parts can be used as intermediate steps in answering the question. You can decide whether to deduct marks if the student uses steps, by setting `stepspenalty` below. See §6.2 for more details on steps.	`[]`
`stepspenalty`	The number of marks to deduct from the total available for this part, if the student uses steps.	`0`

Table 6.1: The valid properties of a part. The `type` property is required, as denoted by the italic typeface. The `prompt` property is a `content` block — see chapter 7.

6.2.1 Creating steps

You define steps using the `steps` property of the `part` object. The `steps` property is itself an array of `part` objects, and so creating a step is exactly equivalent to creating a question part. All the part types explained in chapter 8 are available to use.

If you decide that marks should be deducted when the student views the steps, you can do so by setting the `stepspenalty` property to a non-zero value. This number is then subtracted from the maximum number of marks available for the part. Note that this procedure can result in negative marking for parts — see §6.1.

6.2.2 Steps examples

Below are two different examples of how steps could be used.

6.2.2.1 Informative steps

In this example viewing steps deducts no marks as it is simply a reminder.

```
{
  type: numberentry
  marks: 1
  prompt: "Convert $\var{a}$ km into metres."
  answer: "{a*1000}"
  steps: [
    {
      type: information
      prompt: "Recall: $1$ km = $1000$ m."
    }
  ]
}
```

6.2.2.2 Sub-part steps

In this example viewing steps deducts 1 mark from the student's total. If the student answers steps correctly but does not enter the correct answer overall they are given partial marks. However, if the student gives the correct overall answer the answer to steps is not considered when awarding marks.

```
{
   type: numberentry
   marks: 2
   prompt: "Find $x$ where $15 + 3x = 42$."
   answer: "9"
   steps: [
     {
       type: gapfill
             prompt: "Rearrange the equation and the solve for $x$.
                      So, $3x =$ [[0]]. You should now be able to enter
                      your final answer."'
             gaps: [
               { type: numberentry, answer: "27", marks: 0.5 }
             ]
     }
   ]
   stepspenalty: 1
}
```

Chapter 7

Content blocks

A number of the properties within the exam objects are `content` blocks, marked as *(Content)* in the various tables throughout this manual. These blocks are displayed as HTML when the exam is run, so any valid HTML is allowed.

It is not necessary for you to be familiar with HTML markup — simpler *Textile* markup (`http://textile.thresholdstate.com/`) is also permitted, and its use is encouraged. Only use HTML as a last resort, e.g. when you want to include a video in your exam. See the Textile website for an explanation of the markup.

When you want to write more complicated mathematics, the restrictions of HTML will be too great. This problem is overcome by using LaTeX for displaying mathematics, as explained in the next section.

7.1 LaTeX

It is possible to use LaTeX syntax to display mathematics in `content` blocks, e.g. in question `statements`, or part `prompts`.

Inline math-mode LaTeX can be used by enclosing content in dollar signs; display-mode LaTeX (i.e. on its own line, and in a larger font) can be achieved by enclosing content between escaped square brackets (`\[` and `\]`).

7.1.1 Variables

In §5.3 we explained how to use declared variables in `content` blocks and part answers. If you would like to use LaTeX to format content, then it is not possible to use the double-brace syntax within this content, because the braces conflict with LaTeX's syntax.

Instead, an entirely equivalent syntax is to use the `\var{}` command. Just as with braces, using `\var{}` will evaluate its argument according to any declared variables and substitute in the value. See table 7.1 for some examples.

Using LaTeX will allow you to display much more complicated mathematics than is possible with raw HTML. Questions involving differentiation or integration, for

Brace syntax	LaTeX syntax
`What is {a}+{b}?`	`What is $\var{a}+\var{b}$?`
`Calculate 2{a}-{b^2}`	`Calculate $2\var{a}-\var{b^2}$`

Table 7.1: Examples of variable usage within LaTeX content.

example, become easy to write, as shown in table 7.2.

Content	Display
`\[\frac{\partial}{\partial x} x^{\var{a}}y^{\var{b}}\]`	$\frac{\partial}{\partial x}x^2y^5$
`\[\int_{\var{a}}^{\var{b}}{x^{4}\mathrm{d}x}\]`	$\int_2^5 x^4\mathrm{d}x$

Table 7.2: Examples of variable usage within more complicated LaTeX content. Variables are declared as `a: 2`, `b: 5`.

7.1.2 Simplification, Rearrangement and Display

One of the most powerful features of Numbas is the ability to feedback to the user a full and detailed solution of the questions. Most such solutions require a step-by-step format so that the explanation is comprehensive and understandable. In order to do this Numbas includes powerful automatic simplification, rearrangement and display functionality enabling the author of the questions to provide such detailed solutions, displayed in the best possible mathematical format and so adding clarity to the explanation.
This functionality is provided through a set of "simplification rules." Note that the term simplification is used in this section to cover rearrangement and display functionality as well as standard simplification rules.
These include cancelling numerical factors in fractions, collecting numerical factors together, simplifying expressions involving 0 or 1, etc.
Since you do not necessarily know what values your random variables will take, the ability to automatically simplify expressions is very useful.
See table 7.3 for a list of standard rules and their simplification/rearrangement/display features.

Simplification is performed in LaTeX content with the `\simplify` command. Simplification can also be performed on the displayed answers in a JME part, using the `answersimplification` property of the JME part (see §8.2 for more information on the JME part type). The syntax of the simplification command is

```
\simplify[rules]{expression}
```

where `expression` is the mathematical expression you want to simplify, and the optional `rules` argument further refines how the expression should be simplified.
The `rules` argument is a list of rules, either derived from the standard rules or by defining new rulese.g.

```
$\simplify[constantsFirst,zeroPower]{expression}$
```

This simplifies the expression by applying only the constantsFirst and the zeroPower rules as given in the full list of standard simplification rules. The rules are applied from left to right.
If you omit the `rules` argument i.e. write `$\simplify{expression}$` then this is the same as `$\simplify[all]{expression}$` where `all` refers to the first 15 rules in table 7.3.
As an example, consider the code `$\simplify{ ({a}*x)/({b}*y) }$`. Assuming $a = 2$ and $b = -1$, the result will be displayed as $-\frac{2x}{y}$ according to the latex expression `$-\frac{2x}{y}$`.
However, if you write `$\simplify[]{expression}$` then none of the rules in table 7.3 is applied.
If you do not want to use a rule in `all` , then include it with prefixed by an exclamation mark.
For example, if you do not want `collectNumbers` switched on then you can write `$\simplify[all,!collectNumbers]{expression}$`
As another example, you may not want to use `collectNumbers`, `sqrtSquare` in the simplification of an expression but want to use `fractionNumbers` (which is not in `all`) then this would be achieved by:

```
\simplify[all,!collectNumbers,!sqrtSquare,fractionNumbers]{expression}
```

Note that this means that the term "standard rule" includes all those found in table 7.3 as well as all their negations.
The `basic` rule found in table 7.3 is like `all` in that it a collection of rules, but these are the usual "tidying up" and presentation rules which we normally use in writing mathematics and they are normally always applied. You can see the collection of `basic` rules in the **appendix**. It is possible to turn these off by using `$\simplify[!basic]{expression}$`

7.1.3 Rule Sets from Standard Rules

You can define your own simplification rules in an exam by using the exam property `rulesets`.
You can use the existing standard rules to define a new rule, very useful if you use such a rule regularly e.g.
`rulesets: { std: [all, !collectNumbers, fractionNumbers]}`
defines a new ruleset `std` which you can use in subsequent questions.

Rule name	Meaning
unitFactor	cancel `1*x` to `x`
unitPower	cancel `x^1` to `x`
unitDenominator	cancel `x/1` to `x`
zeroFactor	cancel `0*x` to `0`
zeroTerm	cancel `x+0` to `x`
zeroPower	cancel `x^0` to `1`
collectNumbers	collect `1*2*3` to `6`
simplifyFractions	cancel `(a*b)/(a*c)` to `b/c`
zeroBase	cancel `0^x` to `0`
constantsFirst	rearrange `x*3` to `3*x`
sqrtProduct	simplify `sqrt(a)*sqrt(b)` to `sqrt(a*b)`
sqrtDivision	simplify `sqrt(a)/sqrt(b)` to `sqrt(a/b)`
sqrtSquare	simplify `sqrt(x^2)` to `x`
trig	simplify various trigonometric values e.g. `sin(n*pi)` to `0`
otherNumbers	simplify `2^3` to `8`
fractionNumbers	display all numbers as fractions instead of decimals
all	all of the rules above except for fractionNumbers
basic	the basic rules for the display of mathematics.

Table 7.3: The available standard simplification rules, in order. Note that the rule names are case-sensitive, and a rule will not be applied if it does not appear exactly as in the table. Also the negations of these rules are also standard.

This is the type of rule or a variant you would typically use in a step-by-step solution. For example, you would use the above rule in the following:
`\simplify[std]{({a}+{b})/{n}={(a+b)/n}}`
where you do not want `({a}+{b})/{n}` to be evaluated to the left of the = sign, so you switch off `collectNumbers`, and you want `{(a+b)/n}` to the right of the = sign displayed as a fraction and not a decimal so you switch on `fractionNumber`.
You can define as many rulesets as you like in an exam, e.g.

```
rulesets: {
std: [all, !collectNumbers, fractionNumbers]
cancelId: [unitFactor,unitPower,unitDenominator,zeroFactor,zeroTerm,zeroPower]
fracCancel: [cancelId, collectNumbers, fractionNumbers]
}
```

The general format is: `\simplify[A,B,...]{expression}` where `A,B,..` are rule sets.
Note that the rules are read from left to right so that for example the following ruleset: `rule1: [all,!collectNumbers, fractionNumbers,collectNumbers]` would have the `collectNumbers` rule turned on.

7.1.4 Defining non-standard Rules

You can define rules which are not part of the standard set as given in table 7.3. You use the powerful pattern-matching capability of Numbas. For example, if you want to improve the display of surds then the following rule could be used:

```
rulesets: {
        surdDisplay: [
                        {
                                pattern: "a/sqrt(b)"
                                result: "(sqrt(b)*a)/b"
                        }
                        ]
        }
```

This rule will if applied to an expression such as $\frac{x+1}{\sqrt{3}}$ would display it as $\frac{\sqrt{3}(x+1)}{3}$. This, as with all such rules, need to be used carefully as for example $\frac{2}{\sqrt{3}}$ would be displayed as $\frac{\sqrt{3}2}{3}$ and the usual method of displaying this would be $\frac{2\sqrt{3}}{3}$. It is possible to define rules to take care of these display requirements. See **appendix** for more detail.

Chapter 8

Question parts and types

Each exam consists of a number of questions which, in turn, consist of a number of parts. A part will generally be where the student will need to provide some input as their answer.

Numbas supports a variety of part types. This chapter describes each part type in detail, explaining under which circumstances you might use each type, and how to define the properties of a `part` object.

Each `part` object is an element of the `parts` array, within a `question` object. Parts will typically be defined as follows (see, for example, listing 2.1):

```
parts: [
  {
    type: <parttype>
    <other_properties>
  }
]
```

The `type` property is mandatory; other properties are optional. Table 6.1 lists the `part` object properties which can be used independently of the part type. Properties which are specific to particular part types are explained in the following sections.

8.1 Number entry

In a *Number entry* part, the student's answer should be a number (integer or real), which either matches the correct answer exactly, or lies within a particular range of the correct answer. The correct answer cannot be a fraction or a complex number; use the JME part type for these cases — see §8.2.

For a number entry part, set `type` to `numberentry`. Table 8.1 lists the properties of this part type. (All properties are optional, unless otherwise stated, and if not set, will take the default value listed.)

Declared variables and functions can be used in the `answer`, `minvalue`, and

Property	Description	Default value
`minvalue`*	The student's answer must be greater than or equal to this value for the answer to be considered correct.	`0`
`maxvalue`*	The student's answer must be less than or equal to this value for the answer to be considered correct.	`0`
`answer`*	The student's answer must be exactly equal to this value for the answer to be considered correct.	`0`
`integeranswer`	This specifies whether the answer must be an integer.	`false`
`partialcredit`	This specifies the percentage of the total part mark to be awarded, if `integeranswer` is set, and the student's answer is within the allowable range, but not an integer.	`0`
`inputStep`	The amount to change the entered number by when the student clicks on the up or down arrows in the input field.	`1`

Table 8.1: The valid properties of a number entry part. *Note that if `answer` is set, then `minvalue` and `maxvalue` need not be. If `answer` is not set, then both `minvalue` and `maxvalue` must be.

`maxvalue` properties, and the result of any variable evaluation must be a pure number (integer or real).

The example exam (listing 2.1) includes two number entry parts, where exact answers are required. Another example of a number entry part is the following:

```
{
  type: numberentry
  marks: 1
  prompt: "What is $\sqrt{2}$?"
  minvalue: 1.41
  maxvalue: 1.415
}
```

Here, the student's answer must lie in the range $[1.41, 1.415]$ for it to be marked correct, and one mark awarded.

8.2 Judged mathematical expression

In a *Judged Mathematical Expression (JME)* part, the student must enter a mathematical expression, which is equivalent to an expression defined to be correct. You should use this part type when the answer needs to be a mathematical expression, e.g. rearranging algebra, integrating a function, stating an identity, or when you need to use complex numbers.

For a JME part, set the part `type` to `jme`. Table 8.2 lists the properties of the JME part object, and table 8.3 lists the properties of the `restriction` object, which is used to define constraints on the student's answer in a JME part. How this object is used is described later.

Subsequent sections will describe various aspects of the JME part in greater detail.

8.2.1 Answer comparison

To determine whether the student's answer matches the correct answer, both expressions are evaluated several times, by substituting a set of randomly chosen values for the unknowns in each expression. These randomly chosen values are controlled by the `vsetrangepoints` and `vsetrange` properties.

As an example, suppose that the correct answer were $3x^2$, where the numbers 3 and 2 may or may not have been randomly chosen. In this expression x is the only unknown. The default comparison check is to randomly choose five values from the range $[0, 1]$, and substitute them for x in the expression. The same is done for the student's answer (using the same five values); for each evaluation, the student's answer must be sufficiently close to the correct answer for it to be considered correct.

Because evaluation in this way can lead to rounding errors, there are a number of available "checking methods," to overcome any possible problems. You can also decide that the comparison is allowed to fail up to a certain number of times before the student's answer is deemed incorrect. See §8.2.1.3 for more on this.

A consequence of this method of comparison is that any mathematical expression which is equivalent to the correct answer is permitted. For example, if the correct answer were $3x^2$ (`3*x^2` in JME syntax), then the following student answers would all be marked correct: `2*1.5*x^2`, `3*x*x`, `2*x^2+x^2`.

This behaviour can be prevented by restricting the student's answer in various ways, e.g. by defining string restrictions, length restrictions, and accuracy settings.

8.2.1.1 String restrictions

It is possible to restrict the characters and strings a student can enter as part of their answer. In contrast, you can also specify that the student's answer must contain particular characters or strings. This is done through the `notallowed` and `musthave` properties of the JME `part` object. These properties are `restriction` objects as shown in table 8.3. Consider the following example:

```
musthave: {
  strings: [hello,goodbye]
  message: Your answer is missing some words.
```

```
}
notallowed: {
  strings: [+,*,-]
  message: You have included forbidden characters in your answer.
  showstrings: true
}
```

For this particular part, the student's answer must contain the strings `hello` and `goodbye`, and must not include `+`, `*`, or `-`. If the student's answer violates any of these restrictions, then the appropriate message is shown, and in the `notallowed` case, the student will also be shown which strings are not allowed. In both cases the student would receive no marks for the part, since the `partialcredit` property is set to `0` by default.

8.2.1.2 Length restrictions

In certain circumstances, you might decide that the student's answer must be shorter or longer than a set number of characters.

Consider asking the student to expand $(x+1)^2$. The correct answer is x^2+2x+1, but due to the comparison method, the student could enter the original expression, and be awarded the marks, since the two expressions are equivalent.

One way to solve the problem (not necessarily the best) is to require that the student's answer is longer than seven characters, which is the number of characters required to enter `(x+1)^2` (in JME syntax). You could also combine the length restriction with a string restriction, forbidding left or right parentheses. The JME part object would then contain the following:

```
minlength: {length: 8}
notallowed: {strings: [(,)]}
```

In this case, the other properties of the `restriction` object take their default values.

8.2.1.3 Accuracy

As briefly mentioned earlier, evaluation and comparison of the correct answer and student's answer can lead to rounding errors. Several checking methods are available to help alleviate this problem. The `checkingtype` property controls the method, of which there are four — relative difference (`reldiff`), absolute difference (`absdiff`), decimal points (`dp`), and significant figures (`sigfig`). The `checkingaccuracy` property controls the tolerance involved in the method. See table 8.4 for a summary of the checking methods.

In each of the checking methods described below, the comparison is performed a number of times (determined by the `vsetrangepoints` property, for which the default is 5). The student's answer is deemed incorrect as soon as a comparison fails, unless the `failurerate` property is set to a value other than the default of 1.

In this case the comparison is allowed to fail a certain number of times before the student's answer is deemed incorrect. You might do this if you know the comparison will probably fail so many times over the range `vsetrange`.

Be wary of setting a comparison range including points on which the correct answer is not defined. For example, if the correct answer were $1/(2x-1)$, then the default comparison range of $[0, 1]$ is not acceptable, because the expression is not defined on $x = 1/2$.

The *relative difference* checking method calculates

$$|(a_{\mathrm{c}} - a_{\mathrm{s}})\,/a_{\mathrm{s}}|$$

(where a_{c} is the correct answer and a_{s} is the student's answer), and compares this value to `checkingaccuracy`. If it is strictly greater than `checkingaccuracy` (for which the default is 0.0001) then the comparison has failed, and the student's answer is deemed incorrect.

The *absolute difference* checking method is very similar to the relative difference method, except the value compared to `checkingaccuracy` is

$$|a_{\mathrm{c}} - a_{\mathrm{s}}|.$$

If this value is strictly greater than `checkingaccuracy` (for which the default is again 0.0001), then the comparison fails, and the student's answer is deemed incorrect.

In the *decimal points* checking method, the two answers a_{c} and a_{s} are both rounded to `checkingaccuracy` decimal points (the default is 5 decimal points). If, after rounding, the two answers are not exactly equal, the comparison has failed, and the student's answer is deemed incorrect.

The final checking method, *significant figures*, works in much the same way as the decimal points method, except the two answers are rounded to `checkingaccuracy` significant figures (the default is 5 significant figures). If the two rounded answers are not exactly equal, then the comparison has failed, and the student's answer is deemed incorrect.

8.2.2 Revealed answer simplification

The simplification rules applied to the displayed answer in a JME part can be specified using the `answersimplification` property of the JME part. The available rules are identical to those explained in §7.1.2. The property itself is defined as

```
answersimplification: 1111111011111011
answersimplification: "sqrtProduct,unitPower"
```

where the string of 1s and 0s shown here defines the default set of rules.

8.2.3 JME example

The following example of a JME question part brings together a number of the topics explained in the preceding sections.

```
{
  type: jme
  marks: 2
  prompt: "What is the length of the vector $(x,y)$?"
  answer: sqrt(x^2+y^2)
  checkingtype: absdiff
  checkingaccuracy: 0.01
  failurerate: 1
  vsetrangepoints: 5
  vsetrange: [-10,10]
  maxlength: {
    length: 20
    partialcredit: 50
    message: Your answer shouldn't be very complicated.
  }
  notallowed: {
    strings: [-,/]
    message: Please write your answer in the usual way.
  }
}
```

This JME part asks the question "What is the length of the vector (x, y)?", for which the correct answer is $\sqrt{x^2 + y^2}$. Two marks are awarded for a correct answer. The absolute difference method of comparison will be used; this difference should be less than 0.01. If any comparison does not match, the student's answer is deemed incorrect. Five points are randomly chosen over the range $[-10, 10]$ for the comparison.

The student's answer can be no longer than 20 characters; if it is, then a message is shown, and only 50% of marks will be awarded for an otherwise correct answer.

The student's answer cannot include `-` or `/`; if it does, then a message is shown, and no marks are awarded.

8.3 Pattern match

In a *Pattern match* part the student must enter a string which matches a regular expression. The valid properties of the pattern match part are given in table 8.5.

An example of a pattern match part might be the following.

```
{
  type: patternmatch
```

```
  marks: 1
  prompt: Write the name of a Top Gear presenter.
  answer: (James( May)?)|(Richard( Hammond)?)|(Jeremy( Clarkson)?)
  displayanswer: Jeremy
  casesensitive: false
}
```

In this example, the valid answers are `James`, `Richard`, `Jeremy`, `James May`, `Richard Hammond`, `Jeremy Clarkson`. The matches to the correct answers are not case-sensitive. Example invalid answers include `JamesMay`, `Richard Clarkson`, and `Hammond Richard`.

When the correct answer is revealed, the name `Jeremy` is shown. It is up to you to define a `displayanswer` which describes the possible correct answers.

Note: How to write regular expressions is beyond the scope of this manual, but when using this part type, be absolutely sure that your regular expression is correct. It is very easy to construct a regular expression which looks correct, but in fact, does not capture what you intend.

8.4 Multiple choice

There are three part types which can be thought of as *Multiple choice*.

1_n_2 There are several answers, of which the student must pick one. (For example, a list of statements, only one of which is true.)

m_n_2 There are several answers, of which the student can pick any number. (For example, a list of statements, several of which are true.)

m_n_x There is an $M \times N$ matrix of choice/answer pairs. The student will either have to select one cell from each row, or select any number of cells. (This type is equivalent to the previous two types, but repeated for multiple situations/objects.)

In each case a marking matrix must be defined, which determines how many marks should be awarded for a particular choice. For each choice that the student selects, the corresponding marks in the marking matrix are added to (or subtracted from) the score for the part. Table 8.6 details the available properties for this part type.

An example of an `m_n_2` part type might be the following.

```
{
  type: m_n_2
  marks: 2
  prompt: Which of the following are true?
  choices: [
    "$\sqrt{4} = 2$"
```

```
    "$x^2 = x$"
    "$x^2+x+1 = 0$ has no real solutions"
  ]
  matrix: [1,-1,1]
  minimummarks: 0
  shufflechoices: true
  displaycolumns: 3
}
```

Here, we have chosen the `m_n_2` part type, so the student must select a number of choices. If the student chooses the first and third choices (as defined in the question), then they will be awarded one mark for each. Choosing the second choice will result in the student having one mark deducted. The student will see the choices displayed in a random order, and the student's mark for this part cannot be negative, owing to the `minmummarks` property being set to zero (the default).

8.5 Gap fill

The *Gap fill* part type presents a block of content to the student, with "gaps" which they must fill in. Each gap is defined in the same way as any other part. This part type makes it possible to combine several other part types in a single question part. The following example will better illustrate how to use this part type.

```
{
  type: gapfill
  prompt: "
    The equation $x^2 + 3x + 2 = 0$ has [[0]] linear factors.

    The least solution is $x = $[[1]]

    The greatest solution is $x = $[[2]]
  "
  gaps: [
    { type: numberentry, answer: 2, marks: 1 }
    { type: jme, answer: x + 1, marks: 1 }
    { type: jme, answer: x + 2, marks: 1 }
  ]
}
```

The gap fill part type has only one property, namely `gaps`, which is an array of `part` objects. In the example above there are three gaps. The first is a number entry part, which corresponds to the placeholder given by `[[0]]`. In other words, the student will be expected to enter an answer in the input box which will appear between the words *has* and *linear*.

Similarly, input boxes will appear in place of `[[1]]` and `[[2]]`, for which the student is expected to enter JME expressions.

The total number of marks available for the part is calculated automatically from the marks available for each gap.

8.6 Information

The *Information* part is not really a part type in the same sense as the others. It is simply a block of content with no special properties, often used to give students hints in more difficult questions, e.g.

```
{
  type: information
  prompt: "Pythagoras' Theorem states: $a^2 = \sqrt{b^2 + c^2}$"
}
```

Property	Description	Default value
answer	A JME expression defining the correct answer (see chapter 9).	*no default*
answersimplification	The simplification rules that should be applied to the revealed answer. For display purposes only. A string of 0s and 1s, or the names of simplification rules to enable	see §8.2.2
checkingtype	The method to use when comparing the student's answer to the correct answer (see §8.2.1.3).	reldiff
checkingaccuracy	The inaccuracy tolerance when checking the student's answer against the correct answer. What value this takes depends on the checkingtype (see §8.2.1.3).	see §8.2.1.3
failurerate	The number of comparison failures to allow, before the student's answer is deemed incorrect (see §8.2.1).	1
vsetrangepoints	The number of values to randomly choose from vsetrange, for each unknown, when evaluating variables in an answer comparison (see §8.2.1).	5
vsetrange	A 2-element array defining the start and end points from which vsetrangepoints values should be chosen, in an answer comparison (see §8.2.1).	[0,1]
maxlength	A restriction object defining the maximum length, in characters, which a student's answer can be, or zero if there should be no restrictions (see §8.2.1.2).	{}
minlength	A restriction object defining the minimum length, in characters, which a student's answer can be, or zero if there should be no restrictions (see §8.2.1.2).	{}
musthave	A restriction object defining the strings which a student's answer must contain (see §8.2.1.1).	{}
notallowed	A restriction object defining the strings which a student's answer must not contain (see §8.2.1.1).	{}

Table 8.2: The valid properties of a JME object. The answer property is required, as denoted by the italic typeface.

Property	Description	Default value
`message`	*(Content)* A message to display to the student, when this particular restriction is violated.	*no default*
`partialcredit`	Percentage of the available marks to award, if this restriction is violated. Note that `partialcredit` is compounded if multiple restrictions are in use.	`0`
`length`	If this is a length restriction, then the length restriction of the student's answer, in characters.	`0`
`strings`	If this is a string restriction, then an array of strings, which must, or must not, be included in the student's answer.	`[]`
`showstrings`	If this is a string restriction, determines whether the list of `strings` is displayed to the student, if `strings` is set, and the student violates the string restriction.	`false`

Table 8.3: The valid properties of a restriction object. The `message` property is a `content` block — see chapter 7.

	Correct answer: a_c; Student answer: a_s	
Checking method	Description	When comparison fails
`reldiff`	Relative difference	$\|(a_\mathrm{c} - a_\mathrm{s})/a_\mathrm{s}\| >$ `checkingaccuracy`
`absdiff`	Absolute difference	$\|a_c - a_s\| >$ `checkingaccuracy`
`dp`	Decimal points	a_c and a_s rounded to `checkingaccuracy` decimal points. Failure if the rounded numbers are not exactly equal.
`sigfig`	Significant figures	a_c and a_s rounded to `checkingaccuracy` significant figures. Failure if the rounded numbers are not exactly equal.

Table 8.4: The available checking methods, and the methods for determining when the comparison between the correct answer and student's answer fails (see §8.2.1.3).

Property	Description	Default value
`answer`	A regular expression against which to test the student answer.	*no default*
`displayanswer`	*(Content)* The string to display as the correct answer, when the correct answer is revealed.	*no default*
`casesensitive`	Whether the use of upper or lower case characters matters.	`false`
`partialcredit`	The percentage of the original part mark to award, if `casesensitive` is `true`, and the student is only incorrect because of a difference in character case.	`0`

Table 8.5: The valid properties of a pattern match part. The `answer` and `displayanswer` properties are required, as denoted by the italic typeface. In addition `displayanswer` is a `content` block — see chapter 7.

Property	Description	Default value
choices	*(Content)* An array of strings representing the available choices.	*no default*
answers	*(Content)* If the part type is m_n_x, then an array of strings representing the possible answers to choose.	[]
matrix	The marking matrix. An array of numbers giving the marks awarded for selecting each choice. Negative numbers are allowed. If the part type is m_n_x, then the marking matrix must be an array of arrays, where each array corresponds to the marks for a row of choices.	*no default*
maxmarks	The maximum number of marks which can be awarded for this part. A value of 0 means no restriction.	0
maxanswers	The maximum number of answers the student can select for each choice. A value of 0 means no restriction.	0
minanswers	The minimum number of answers the student can select for each choice.	0
shufflechoices	Determines whether the choices should be displayed randomly.	false
shuffleanswers	Determines whether the answers should be displayed randomly.	false
displaytype	How to display the choices. Possible values are radiogroup, dropdownlist, and checkbox.	radiogroup
displaycolumns	Choices in the 1_n_2 and m_n_2 part types are spread over this many columns, or displayed in a single column list if the value is 0.	0

Table 8.6: The valid properties of a multiple choice part. The choices and matrix properties are required, as denoted by the italic typeface. The choices and answers properties are content blocks — see chapter 7.

Chapter 9

JME syntax

JME syntax is used to define variables, functions, and the answers to JME question parts, and whenever you want to perform calculations. Simple examples of JME syntax are shown in the examples in §5.2.

The syntax uses infix notation, like you would see in many common programming languages, such as C or Fortran, e.g. `x*y`, `2+z^2`, `cos(x)`, `sqrt(abs(y^3))`. There is no flow or control code in the syntax, so constructs like loops and conditional expressions are not possible. There is, however, the facility for basic conditional statements in variable declarations, as explained in §5.2.4.

9.1 Data types

All variables and values have a type. The types are

number A number can be integer (e.g. `1`), real (e.g. `1.467`), or complex (e.g. `3+5i`). Fractions can either be expressed in the usual way, e.g. `a/b`, or as a real number with the simplification rule `fractionnumbers` enabled.

string To write a string, enclose it in single-quotes, e.g. `'Hello'`.

boolean A boolean value is either `true` or `false`.

name Variables have a name, e.g. `x`, `surname`, `answer`. Names should start with a letter, which can be followed by any number of letters and digits. The names `pi`, `e`, and `i` are reserved to represent special constants, and so cannot be used as user-defined variable names. Names are not case-sensitive.

function There is an extensive set of functions available, such as `cos()`, `abs()`, `round()`, etc. A full list of all functions is given in §9.2. You can also define your own functions (see §5.4).

op Besides the usual mathematical operators (`+`, `-`, `*`, `/`), there are several more, as listed in §9.2.

range A *range* represents a discrete set of numbers evenly spaced between two other numbers. A range is defined as `a..b#n` If a step size is not defined, it defaults to 1. Ranges are most often used when defining random variables — see §5.2.3.

list A *list* is a finite collection of elements of any type — see §5.2.5.

9.2 JME operations and functions

The following tables list all supported operations and functions. Note that all the trigonometric functions work in radians.

<table>
<tr><th>Operator</th><th>Meaning</th><th>Example usage</th></tr>
<tr><td><code>+</code></td><td>Addition ($a+b$)</td><td><code>a+b</code></td></tr>
<tr><td><code>-</code></td><td>Subtraction ($a-b$)</td><td><code>a-b</code></td></tr>
<tr><td><code>*</code></td><td>Multiplication ($a \times b$)</td><td><code>a*b</code></td></tr>
<tr><td><code>/</code></td><td>Division ($\frac{a}{b}$)</td><td><code>a/b</code></td></tr>
<tr><td><code>^</code></td><td>Exponentiation (a^b)</td><td><code>a^b</code></td></tr>
<tr><td><code>..</code></td><td>Define a range ($\{x \in \mathbb{N} | a \leq x \leq b\}$)</td><td><code>a..b</code></td></tr>
<tr><td><code>#</code></td><td>Add a step size n, to the range $[a, b]$</td><td><code>a..b#n</code></td></tr>
<tr><td><code><</code></td><td>Less than comparison ($a < b$)</td><td><code>a<b</code></td></tr>
<tr><td><code>></code></td><td>Greater than comparison ($a > b$)</td><td><code>a>b</code></td></tr>
<tr><td><code><=</code></td><td>Less than or equal to comparison ($a \leq b$)</td><td><code>a<=b</code></td></tr>
<tr><td><code>>=</code></td><td>Greater than or equal to comparison ($a \geq b$)</td><td><code>a>=b</code></td></tr>
<tr><td><code>|</code> or <code>divides</code></td><td>Divides ($a|b$, <code>true</code> if $b = am$, $a, b, m \in \mathbb{Z}$)</td><td><code>a|b</code></td></tr>
<tr><td><code><></code></td><td>Inequality comparison ($a \neq b$)</td><td><code>a<>b</code></td></tr>
<tr><td><code>=</code></td><td>Equality comparison ($a == b$)</td><td><code>a=b</code></td></tr>
<tr><td><code>!</code> or <code>not</code></td><td>logical NOT ($\neg a$)</td><td><code>!a</code></td></tr>
<tr><td><code>&</code> or <code>and</code></td><td>logical AND ($a \wedge b$)</td><td><code>a and b</code></td></tr>
<tr><td><code>||</code> or <code>or</code></td><td>logical OR ($a \vee b$)</td><td><code>a or b</code></td></tr>
<tr><td><code>xor</code></td><td>logical XOR ($a \oplus b$)</td><td><code>a xor b</code></td></tr>
<tr><td><code>isa</code></td><td>Examine data type of a value</td><td><code>a isa 'number'</code></td></tr>
</table>

Table 9.1: The operations allowed in JME syntax.

Function	Argument type	Result type	Meaning
`abs(x)`	number	number	Absolute value of a number $\lvert x \rvert$
`arg(z)`	number	number	Argument of a complex number $\arg(z)$
`conj(z)`	number	number	Complex conjugate z^*
`re(z)`	number	number	Real part of a complex number $\Re(z)$
`im(z)`	number	number	Imaginary part of a complex number $\Im(z)$
`isint(x)`	number	boolean	`true` if x is an integer; `false` otherwise
`sqrt(x)`	number	number	Square root of a number $\sqrt{x}$
`root(x,n)`	number, number	number	nth root of a number $x^{\frac{1}{n}}$
`ln(x)`	number	number	Natural logarithm of a number $\ln(x)$
`log10(x)`	number	number	Logarithm to base 10 of a number $\log_{10}(x)$
`exp(x)`	number	number	Exponential function e^x
`max(x,y)`	number, number	number	Maximum of two numbers $\max(x, y)$
`min(x,y)`	number, number	number	Minimum of two numbers $\min(x, y)$
`sign(x)`	number	number	Sign of x: -1, 0 or $+1$

Table 9.2: Basic functions for dealing with numbers.

Function	Argument type	Result type	Meaning
`ceil(x)`	number	number	Round up to nearest whole number
`floor(x)`	number	number	Round down to nearest whole number
`round(x)`	number	number	Round to nearest whole number
`trunc(x)`	number	number	Integer part of x
`fract(x)`	number	number	Fractional part of x
`precround(x,n)`	number, number	number	Round x to n decimal places
`siground(x,n)`	number, number	number	Round x to n significant figures

Table 9.3: Functions for rounding numbers.

Function	Argument type	Result type	Meaning
`sin(x)`	number	number	Sine
`cos(x)`	number	number	Cosine
`tan(x)`	number	number	Tangent
`cosec(x)`	number	number	Cosecant
`sec(x)`	number	number	Secant
`cot(x)`	number	number	Cotangent
`arcsin(x)`	number	number	Inverse sine
`arccos(x)`	number	number	Inverse cosine
`arctan(x)`	number	number	Inverse tangent
`sinh(x)`	number	number	Hyperbolic sine
`cosh(x)`	number	number	Hyperbolic cosine
`tanh(x)`	number	number	Hyperbolic tangent
`cosech(x)`	number	number	Hyperbolic cosecant
`sech(x)`	number	number	Hyperbolic secant
`coth(x)`	number	number	Hyperbolic cotangent
`arcsinh(x)`	number	number	Inverse hyperbolic sine
`arccosh(x)`	number	number	Inverse hyperbolic cosine
`arctanh(x)`	number	number	Inverse hyperbolic tangent
`degrees(x)`	number	number	Convert radians to degrees
`radians(x)`	number	number	Convert degrees to radians

Table 9.4: Trigonometry functions.

Function	Argument type	Result type	Meaning
`fact(x)`	number	number	Factorial of an integer $x!$
`mod(x,n)`	number, number	number	Modulo $x \mod n$
`perm(n,r)`	number, number	number	Permutations ${}^nP_r$
`comb(n,r)`	number, number	number	Combinations ${}^nC_r$
`gcd(a,b)`	number, number	number	Greatest common divisor of a and b
`lcm(a,b)`	number, number	number	Lowest common multiple of a and b

Table 9.5: Number-theoretical operations.

Function	Argument type	Result type	Meaning
`repeat(f,n)`	expr, number	list	Evaluate an expression a given number of times
`map(f,x,[])`	expr, var, list	list	Map an expression to a list
`abs([])`	list	number	Size of a list

Table 9.6: List operations.

Function	Argument type	Result type	Meaning
`random(a..b#n)`	range	number	Choose discrete random variable from range $[a, b]$ in steps of n. If $n = 0$, then choose continuous random variable from range $[a, b]$.
`random(a1,a2,...,aN)`	any	number	Choose a_i randomly from $(a_1, a_2, \ldots, a_N)$
`award(n,b)`	number, boolean	number	n if b is `true`, 0 otherwise
`if(cond,expr,else)`	boolean, any, any	any	If/Else (see §5.2.4)
`switch(cond1,expr1,...)`	Pairs of boolean, any	any	Select between cases (see §5.2.4)

Table 9.7: Functions for choosing between values

Chapter 10

Extensions

It has already been mentioned that it is possible to define new functions to be used when writing an exam. When you unpack Numbas you will find that there are already some JavaScript files in the `extensions` directory.

10.1 Statistics Extension

Table 10.1 details all statistical functions that have already been defined as part of the `stats.js` extension file, to generate random variables and calculate PMFs, PDFs and CDFs for various distributions. Table 10.2 details all other statistical functions. The functions generating certain types of random variables would most likely be used inside a `variables` object. For example, the following defines X as a Bernoulli random variable with success probability p.

```
variables: {
  p: "random(0..1#0.01)"
  X: "randomBernoulli(p)"
}
```

There is also a command to perform linear regression. However, we must output the gradient and intersect of the line of best fit individually. The following prints the line of best fit for the data set $(1,3),(3,12),(7,23),(19,34),(36,40)$.

```
variables: {
  x: [1,3,7,19,36]
  y: [3,12,23,34,40]
  a: "precround(regression(x,y,1),3)"
  b: "precround(regression(x,y,2),3)"
}

statement: """
  The line of best fit is $y = \var{a} + \var{b}x$.
"""
```

Further examples can be found in the `stats-extension.exam`, located in the `exams` directory.

Function	Argument type	Result type	Meaning
`randomBernoulli(p)`	number	number	Random Bernoulli variable, probability p
`randomBinomial(n,p)`	number, number	number	Random Binomial variable, parameters n, p
`pmfBinomial(x,n,p)`	number, number, number	number	Binomial variable PMF, parameters n, p
`randomUniform(a,b)`	number, number	number	Random Uniform variable, parameters a, b
`pmfUniform(x,a,b)`	number, number, number	number	Uniform variable PMF, parameters a, b
`cdfUniform(x,a,b)`	number, number, number	number	Uniform variable CDF, parameters a, b
`randomPoisson(lambda)`	number	number	Random Poisson variable, parameter λ
`pmfPoisson(x,lambda)`	number, number	number	Poisson variable PMF, parameter λ
`randomGeometric(p)`	number	number	Random Geometric variable, probability p
`pmfGeometric(x,p)`	number, number	number	Geometric variable PMF, probabiltiy p
`cdfGeometric(x,p)`	number, number	number	Geometric variable CDF, probability p
`randomExponential(lambda)`	number	number	Random Exponential variable, parameter λ
`pdfExponential(x,lambda)`	number, number	number	Exponential distribution PDF, parameter λ
`cdfExponential(x,lambda)`	number, number	number	Exponential distribution CDF, parameter λ
`randomNormal(mu,sigma)`	number, number	number	Random Normal variable, mean μ, variance σ^2
`pdfNormal(z,mu,sigma)`	number, number, number	number	Normal distribution PDF, mean μ, variance σ^2
`cdfNormal(z)`	number	number	Normal distribution CDF, mean μ, varience σ^2
`randomGamma(n,lambda)`	number, number	number	Random Gamma variable, parameters n, λ
`pdfGamma(x,n,lambda)`	number, number, number	number	Gamma distribution PDF, parameters n, λ

Table 10.1: Functions to do with certain statistical distributions.

Function	Argument type	Result type	Meaning
`mean([])`	list	number	Mean of a list of values
`variance([])`	list	number	Variance of a list of values
`standardDev([])`	list	number	Standard deviation of a list of values
`regression([],[],z)`	list, list, number	number	Uses *Method of Least Squares* to find a line of best fit. $z = 1$ returns intersect, $z = 2$ returns gradient.
`zTest([],mu)`	list, number	number	Z-test for a sample of variables from a Normal distribution with mean μ.

Table 10.2: Statistical functions.

Appendix A

SCORM objects

The `numbas.py` script will, by default, produce a directory containing files to run an exam locally in a browser.

It is also possible to compile your exam into a SCORM 2004 compliant object (`http://scorm.com`), so that it could theoretically be used within a VLE, such as Blackboard or Moodle. At the time of writing, many VLEs (including Blackboard and Moodle) do not support the SCORM 2004 standard well or at all, making it difficult to integrate the Numbas SCORM objects into them.

Nevertheless, the SCORM objects that Numbas does produce adhere to the standard. When the VLEs catch up, the SCORM objects can be used.

A.1 MathJax

By default, Numbas will use a copy of MathJax that is available on the MathJax Content Delivery Network (CDN). Part of the SCORM standard states that an exam object must be self-contained, and in particular, that there should be no dependence on a network connection. In order to overcome this limitation, SCORM compliant exams must include a local copy of MathJax.

A.2 Installing a local copy of MathJax

Follow these steps to install a local copy of MathJax into a new *theme*. This is necessary for producing SCORM objects, and for overcoming the Mozilla Firefox security issue with web fonts, as explained in §1.2.1.

- From the top-level Numbas directory, change to the `themes` directory, where you should see a directory called `default`. This is the directory which controls the default theme. Copy the contents of `default` to a new directory called `local`.

- Download a copy of MathJax from `http://www.mathjax.org`. (MathJax is also available as a git repository.) Unpack the contents into a directory called

MathJax, within `local`, at the same level as `index.html`. Note that this will take a good 10 minutes on a Windows machine, due to the large number of very small files in the MathJax archive.

- Edit the file `index.html` in `local/files`, and change the line which reads

  ```
  <SCRIPT SRC="http://cdn.mathjax.org/mathjax/latest/MathJax.js">
  ```

  so that it now reads

  ```
  <SCRIPT SRC="MathJax/MathJax.js">
  ```

The result of the above steps is to create a new theme called `local`, which includes a local copy of MathJax. Exams produced using this theme will have no dependence on a network connection.

A.2.1 Using the new theme

To use the new `local` theme, compile your exams using the `-t` option, which takes the name of the theme as an argument. Therefore, to compile the example exam, you would type

```
python bin/numbas.py -t local exams/example.exam
```

You can check that the resulting exam will use a local copy of MathJax by ensuring that the `MathJax` directory from the theme has been copied to `output/example`, and that `index.html` has the alteration stated above.

A.3 Creating a SCORM object

The procedure for creating a SCORM object is exactly as in the previous section, except you must include the `-s` option, so that the interpreter knows to include the extra files necessary to produce a valid object, e.g.

```
python bin/numbas.py -t local -s exams/example.exam
```

Incorporating the resulting object into a VLE is beyond the scope of this manual.