INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

From asynchronous IO to reactive stream pipelines

Diogo Paulo de Oliveira Rodrigues

Licenciado em Engenharia Informática e de Computadores

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Júri:

Presidente: [Grau e Nome do presidente do juri]

Vogais: [Grau e Nome do primeiro vogal]
[Grau e Nome do segundo vogal]

Setembro, 2022

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Área Departamental de Engenharia de Electrónica e Telecomunicações e de Computadores

From asynchronous IO to reactive stream pipelines

Diogo Paulo de Oliveira Rodrigues

Licenciado em Engenharia Informática e de Computadores

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Doutor Fernando Miguel Gamboa de Carvalho

Júri:

Presidente: [Grau e Nome do presidente do juri]

Vogais: [Grau e Nome do primeiro vogal]
[Grau e Nome do segundo vogal]

Setembro, 2022

Aos meus ...

Acknowledgments

Os agradecimentos. Apesar de haver total liberdade no conteúdo e forma desta secção, normalmente inicia-se com os agradecimentos institucionais (orientador, instituição, bolsas, colegas de trabalho, ...) e só depois os pessoais (amigos, família, ...)

Abstract

The dissertation must contain two versions of the abstract, one in the same language as the main text, another in a different language. The package assumes the two languages under consideration are always Portuguese and English.

The package will sort the abstracts in the proper order. This means the first abstract will be in the same language as the main text, followed by the abstract in the other language, and then followed by the main text.

The abstract is critical because many researchers will read only that part. Your abstract should provide an accurate and sufficiently detailed summary of your work so that readers will understand what you did, why you did it, what your findings are, and why your findings are useful and important.

The abstract should not contain bibliography citations, tables, charts or diagrams. Abbreviations should be limited. Abbreviations that are defined in the abstract will need to be defined again at first use in the main text.

Finally, you must avoid the use of expressions such as "The present work deals with ... ", "In this thesis are discussed .... ", "The document concludes that .... ", "apparently and .... " etc.

The word limit should be observed, 300 words is the limit.

Abstracts are usually followed by a list of keywords selected by the author. Choosing appropriate keywords is important, because these are used for indexing purposes. Well-chosen keywords enable your manuscript to be more easily identified and cited.

Keywords: Keywords (in English) . . .

Resumo

Independentemente da língua em que está escrita a dissertação, é necessário um resumo na língua do texto principal e um resumo noutra língua. Assume-se que as duas línguas em questão serão sempre o Português e o Inglês.

O template colocará automaticamente em primeiro lugar o resumo na língua do texto principal e depois o resumo na outra língua.

Resumo é a versão precisa, sintética e selectiva do texto do documento, destacando os elementos de maior importância. O resumo possibilita a maior divulgação da tese e sua indexação em bases de dados.

A redação deve ser feita com frases curtas e objectivas, organizadas de acordo com a estrutura do trabalho, dando destaque a cada uma das partes abordadas, assim apresentadas: Introdução; Objectivo; Métodos ; Resultados e Conclusões

O resumo não deve conter citações bibliográficas, tabelas, quadros, esquemas.

E, deve-se evitar o uso de expressões como "O presente trabalho trata ...", "Nesta tese são discutidos....", "O documento conclui que....", "aparentemente é...."etc.

Existe um limite de palavras, 300 palavras é o limite.

Para indexação da tese nas bases de dados e catálogos de bibliotecas devem ser apontados pelo autor as palavras-chave que identifiquem os assuntos nela tratados. Estes permitirão a recuperação da tese quando da busca da literatura publicada.

Palavras-chave: Palavras-chave (em português) ...

Contents

1

Introduction

This package is distributed under GPLv3 License. If you have questions or doubts concerning the guarantees, rights and duties of those who use packages under GPLv3 License, please read http://www.gnu.org/licenses/gpl.html.

A margin-par note!

A a note in a line by itself.

Please note that

this package and template are not official for ISEL/IPL.

2

State of the Art

Firstly, on section **??**, will be made an overview on previously developted work made on this subject, then, on section **??**, will be made a characterization of the key concepts related to it. By last, on section **??**, are presented and explained several technologies representative of the state of the Art on asynchronous data flux in different programming realities, e.g. on Kotlin, JAVA and C#.

2.1 Background

From the end of 80´s to the beginning of the 2000´s, with the acceleration of Moores´s Law in hardware and network bandwidth development, the creation of the web as we know today through the wide spread of use of the HTTP protocol and the support from new operative systems to multithreading support, the necessity of high responsiveness servers started to grow. This increase in demand of new ways to handle data through parallelism, caused the necessity of design new programming models compatible with concurrent work.

Taking the wave initiated by the Gang of Four in [**gof**], where 23 patterns were compiled to deal with object-oriented problems, a group of researchers published the *Proactor Pattern* in the paper [**proactor**] to deal with asynchronous IO. In the document, are identified four properties that high-performance web server must have:

- Concurrency - The server must process multiple client requests simultaneously.

- Efficiency - The software design must be built aiming the use of least hardware resources as possible.

- Simplicity - The code of the solution must be easy to understand, modular and avoid own built design patterns as possible.

- Adaptability - The system must be totally decoupled from client implementations, allowing it to be easily used by any client independently of the underlying technologic realities. To achieve this, may be used standards e.g. [**REST**] or SOAP.

The authors propose the *Proactor Pattern*, because in their opinion, conventional concurrency models fail to fully achieve the enumerated properties. In the paper, before presenting the *Proactor Pattern*, are identified two major concurrency models, namely: *multithreading* and *reactive event dispatching*.

The paper refers that one of the most direct implementations of the multithreading approach, is the handling of multiple requests by creating a new thread every request. Each request will then be fully processed and the recently created thread is then be disposed after the work is finished.

This solution has several serious issues. Firstly, creating a new thread per request is highly costly in terms of computational resources, because are involved context switches between user and kernel modes; secondly, must be taken in account synchronization to maintain data integrity. Then, the authors warn about the fact that the IO retrieved data is mainly memory-mappped, wich rises the question: What happens when the data obtained through IO becomes greater than the system memory can hold? The system stalls until more memory becomes available!? On last, if the server receives a high demand of requests, the server easily blocks in the process of creating and disposing threads.

To avoid this issue, the authors, recommended the use of dynamic threadpools to process requests, where each request will be linked to a pre-existing thread,

avoiding all the overhead of creating and disposing a thread per request; however, issues related with memory-mapping and overhead due to the switching of data between different threads maintains.

Another traditional concurrency model identified by the authors of the paper, is the *Reactive Synchronous Event Dispatching* or more commonly known as `Reactor Pattern`. In this model, a *Dispatcher*, with a single thread in a loop, is constantly listening requests from clients and sending work requests to an entity named *Handler*. The *Handler*, will then process the IO work Synchronously and request a connection to the client in the *Dispatcher*. When the requested connection is ready to be used, the *Dispatcher* notifies the *Handler*. After the notification, the *Handler* asynchronously sends the data, that is being or has been obtained through IO, to the client.
Although the authors identifying that this approach is positive, because decouples the application logic from the dispatching mechanisms besided with the low overhead due the use of a single thread, the authors identify several drawbacks with this approach. Firstly, since IO operation are synchronous, the code for this approach is complex because must be set in place mechanisms to avoid IO blocking through hand off mechanisms. Then, if a request processing blocks, the processing of another requests may be impacted.

To keep the positive points but mitigating the identified issues of previous approaches, is suggested the *Proactor Pattern*. This pattern is very similar to the *Reactive Synchronous Event Dispatching*, however, after the requests processed by a single threaded *Completion dispatcher*, the IO work is then dispatched asynchronously to the underlying OS IO subsystems, where multiple requests can be processed simultaneously. For the result to be retrieved, is previously registered a callback in the *Completion Dispatcher* and the OS has the responsibility to queue the finished result in a well known place.

Finally, the *Completion Dispatcher* has the responsibility to dequeue the result placed by the OS and call the correct previously registered callback. With this, this model creates a platform that provides: decoupling between application and processing mechanisms, offers concurrent work without the issues inherent with the use of threading mechanisms and since IO is managed by the OS subsystems, is avoided code complexity in handling possible blocking and scheduling issues.

The *Proactor Pattern*, creates the ground for several models used by modern platforms that use a single/few threads to process client requests and parallel mechanisms to do the heavy work in the background; namely, for example: *Javascript*

NODE.JS, *Spring Webflux*, *vertx* and others.

From what was explained until now, is evident the tendency followed by software architects in terms of asynchronous processing from non-reactive to event driven approaches. Initially the systems were non-reactive, where each request had to be processed in a specific thread and that thread blocked until something got ready to go further. Then, with the asynchronous systems based on events with the introduction of callback systems inspired in patterns like the *Reactor* or *Proactor*; the software design started to become more event driven, allowing the servers to be more efficient in responsiveness, flexibility and resources optimization.

However, are some limitations in these asynchronous models. For example, if the data to be processed is bigger than the memory available or if the data to be calculated is from a source that produces data at a constant rate that must be processed in real time, these models work badly. The traditional models fail to comply these objectives because are mostly eager by design or not comply with the notion of a continuous source of information that requires to be processed in real time. Taken this in account, projects like project Reactor, Asynchronous Enumerable provided by Microsoft or papers like [**LAZYVSEAGER**], try to deal with these issues, by providing API's that merge the concepts of Fluent API´s, functional programming and code syntax that tries to resemble synchronous code, being the complexity inherent with asynchronous models implementations hidden from the programmer.

2.2 Asynchronous flux key concepts and design alternatives

With the development of several approaches and implementations related to asynchronous data flux in several programming plataforms; a dictionary of properties, concepts and design alternatives started to grow by itself. In the following, are discussed several of the concepts related with asynchronous data flux, namely:

2.2.1 Synchrony versus Asynchrony

Before explaining more terms related with asynchronous data flux, it's important to clarify what is synchrony and asynchrony in programming.

Asynchrony in programming, is a call to a function or routine that returns immediately, not blocking the caller until the operation is finished. The operation processing, will be completely independent from the caller execution thread and can even be done in another machine and technology. This way, the caller is freed to do more work, even to start `N` more operations in parallel.

Meanwhile, a call to a synchronous function or routine, blocks the caller until the operation finishes. In this case, the caller has to wait for the completion of the synchronous call before going forward, which limits the program efficiency if parallelism is applicable. Usually, in this case, the execution `Thread` of the caller is

To better visualize what was explained, we have the following example:

```
1 CompletableFuture
2         .runAsync(()->
    System.out.println("I
    am asynchronous!"));
3
4 System.out.println("What
    you are!?");
5
6 //Output:
7 //What you are!?
8 //I am asynchronous!
```

Listing (2.1) Asynchronous call example

(a)

```
1 Stream.of("I am
    synchronous!")
2      .forEach(System.out::
    println);
3
4 System.out.println("Nice
    to know!");
5
6 //Output:
7 //I am synchronous!
8 //Nice to know!
```

Listing (2.2) Synchronous call example

(b)

Figure 2.1: Example of Synchronous and Asynchronous calls in JAVA

As we can see, in the synchronous call example, the operation return only happens after the whole subsequent operation is finished, consequently, the print of the message made in the synchronous routine happens right before the second statement call.

Meanwhile, in the asynchronous operation call, the return happens immediately after the call, however, the processing inherent with that operation will start just when the subsystem that handles the asynchronous function is ready to process that work, for example, when a worker thread from a threadpool is ready to handle that work. Because of that, the print made in the asynchronous operation will happen "out of order", being done just after the print of the second statement.

2.2.2 Push vs Pull

Another concept important to understand how asynchronous data flux is handled in programming, is the *Pull* and *Push* processing patterns. In *Pull* pattern, usually, exists a source of data and the program iterates over that source to operate over each item.

On the other hand, in the *Push* pattern, the items of the data source are "Pushed" to a routine that will operate over that item. To help to assimilate what was just explained, we have the following example:

```
1  Stream<Integer> stream =
       Stream.of(1,2,3,4,5);
2          Iterator it =
       stream.iterator();
3
4          while(it.hasNext
       ()) {
5                System.out.
       println(it.next());
6          }
7
8  //Output:
9  //1
10 //2
11 //3
12 //4
13 //5
```

Listing 2.3: Pull pattern example

```
1  Stream<Integer> stream =
       Stream.of(1,2,3,4,5);
2
3  stream.forEach(curr ->
       System.out.println(
       curr));
4
5
6
7  //Output:
8  //1
9  //2
10 //3
11 //4
12 //5
```

Listing 2.4: Push pattern example

As we can see, in the pull pattern, the items are "pulled" from a data source through an iteration mechanism.

In contrast with that, in the *Push* pattern, items are pushed to a consumer through a supplier.

2.2.3 Hot versus Cold

Another property that must be took in account when handling this subject is nature of a data flux. There are two main adjectives to name a data flux, Hot or Cold.

A cold data flux, is a sequence of information that is received by a consumer with the control of that consumer. For example, when program uses a stream to lazily retrieve a sequence of words from a buffer, the consumer has total control on when and how the next item will be proccessed.

On the other hand, in a `Hot` data flux, the consumer has no control on how many items will receive in the given moment. In a Hot data flux, the consumer can receive data from the moment the stream is open while in a Cold stream, the consumer will receive just when data is requested from the stream.

```
1 Stream<Integer> stream =
    Stream.of(1,2,3,4,5);
2 Thread.sleep(2000);
3 stream.forEach(System.out
    ::println); //
    1,2,3,4,5
```

Listing 2.5: Cold Example

```
1 ConnectableFlowable<Long> flux =
    Flowable
2                 .intervalRange(0, 100,
     0, 100, TimeUnit.MILLISECONDS)
3                 .publish();
4 flux.connect(); // Start emition
5 Thread.sleep(1000);
6 flux.blockingSubscribe(System.out::
    println); // 10,12,13,14,15,16 ...
```

Listing 2.6: Hot Example

As we can see, in the `Hot` data flux example, the items are emitted from the moment the emitter is created. In the example, the emitter emits a number per hundred milliseconds, where the number is 1 or the "repetition number" + 1. Since the stream is continuous, when the consumer starts to listen the pipepline 1 seconds after the emittion started, 10 items were already lost when a observer subscribes to that data source. However, in the `Cold` example, the consumer only receives the items when he manually consumes the pipeline.

2.3 State of the Art

The thesis style includes the following options, that must be included in the options list in the `\documentclass[options]{thesisisel}` line at the top of the `template.tex` file.

The list below aggregates related options in a single item. For each list, the default value is prefixed with a *.

2.3.1 Language Related Options

You must choose the main language for the document. The available options are:

1. ***pt** — The text is written in Portuguese (with a small abstract in English).
2. **en** — The text is written in English (with a small abstract in Portuguese).

The language option affects:

- **The order of the summaries.** At first the abstract in the main language and then in the foreign language. This means that if your main language for the document in english, you will see first the abstract (in english) and then the 'resumo' (in portuguese). If you switch the main language for the document, it will also automatically switch the order of the summaries.
- **The names for document sectioning.** E.g., 'Chapter' vs. 'Capítulo', 'Table of Contents' vs. 'Índice', 'Figure' vs. 'Figura', etc.
- **The type of documents in the bibliography.** E.g., 'Technical Report' vs. 'Relatório Técnico', 'MSc Thesis' vs. 'Tese de Mestrado', etc.

No mater which language you chose, you will always have the appropriate hyphenation rules according to the language at that point. You always get portuguese hyphenation rules in the 'Resumo', english hyphenation rules in the 'Abstract', and then the main language hyphenation rules for the rest of the document. If you need to force hyphenation write inside of `\hyphenation{}` the hyphenated word, e.g.
`\hyphenation{op-ti-cal net-works}`.

2.3.2 Class of Text

You must choose the class of text for the document. The available options are:

1. **bsc** — BSc graduation report.
2. **prepmsc** — Preparation of MSc dissertation. This is a preliminary report graduate students at ISEL/IPL must prepare to conclude the first semester of the two-semesters MSc work. The files specified by (a) `\dedicatoryfile` and (b) `\acknowledgmentsfile` are ignored, even if present, for this class of document.
3. **msc** — MSc dissertation.

2.3.3 Printing

You must choose how your document will be printed. The available options are:

1. **oneside** — Single side page printing, and
2. ***twoside** — Double sided page printing.

The article 50th, of Decree-Law No. 115/2013, requires the deposit of a digital copy of doctoral thesis and master's dissertations in a repository that is part of the RCAAP repository[1], https://www.rcaap.pt. This deposit aims to preserve scientific work, as well as providing Open Access to scientific production is not restricted object or embargo.

For the reason explained above, we include the option to format your thesis in a way that presents well on screen and/or on paper. But always remember that your work will be stored in the RCAAP portal in electronic format.

The available options are:

1. **onpaper** — Format your thesis in a way that presents on paper or,
2. ***onscreen** — on screen.

2.3.4 Font Size

You must select the encoding for your text. The available options are:

1. **11pt** — Eleven (11) points font size.
2. ***12pt** — Twelve (12) points font size. You should really stick to 12pt...

2.3.5 Text Encoding

You must choose the font size for your document. The available options are:

1. **latin1** — Use Latin-1 (ISO 8859-1) encoding. Most probably you should use this option if you use Windows;
2. **utf8** — Use UTF8 encoding. Most probably you should use this option if you are not using Windows.

[1] Repositórios Científicos de Acesso Aberto de Portugal

2.3.6 Examples

Let's have a look at a couple of examples:

- BSc graduation report, in portuguese, with 11pt size and to be printed one sided (I wonder why one would do this!)
  `\documentclass[bsc,pt,11pt,oneside,latin1]{thesisisel}`
- Preparation of MSc thesis, in portuguese, with 12pt size and to be printed one sided (I wonder why one would do this!). Note that, `pt` is declared by default, so it can be omitted:
  `\documentclass[prepmsc,12pt,oneside,latin1]{thesisisel}`
- MSc dissertation, in english, with 12pt size and to be printed double sided on screen. Note that, `twoside` and `12pt` are declared by default, so it can be omitted:
  `\documentclass[msc,en,utf8,onscreen]{thesisisel}`

The present document is defined according to the following settings:

```
\documentclass[msc,pt,twoside,12pt,a4paper,utf8,onscreen,h
↪ yperref=true,listof=totoc]
↪ {thesisisel}
```

2.4 How to Write Using LaTeX

Please have a look at Chapter **??**, where you may find many examples of LaTeX constructs, such as sectioning, inserting figures and tables, writing equations, theorems and algorithms, exhibit code listings, etc.

3

A Short LaTeX Tutorial with Examples

This Chapter aims at exemplifying how to do common stuff with LaTeX. We also show some stuff which is not that common! ;)

Please, use these examples as a starting point, but you should always consider using the Big Oracle (aka, Google, your best friend) to search for additional information or alternative ways for achieving similar results.

3.1 Document Structure

In engineering and science, a thesis or dissertation is the culmination of a master's or Ph.D. degree. A thesis or dissertation presents the research that the student performed for that degree. From the student's perspective, the primary purpose of a thesis or dissertation is to persuade the student's committee that he or she has performed and communicated research worthy of the degree. In other words, the main purpose of the thesis or dissertation is to help the student secure the degree.

From the perspective of the engineering and scientific community, the primary purpose is to document the student's research. Although much research from theses and dissertations is also communicated in journal articles, theses and dissertations stand as detailed documents that allow others to see what the work was and

how it was performed. For that reason, theses and dissertations are often read by other graduate students, especially those working in the research group of the authoring student [**gustavii2016write**, **glasman2010science**, **chicago**, **strunk**].

With a thesis or dissertation, the format also encompasses the names of the sections that are expected:

1. Thesis Cover
2. Acknowledgments (if exist)
3. Abstract (Portuguese and English)
4. Index
5. List of Figures
6. List of Tables
7. Nomenclature/List of Abbreviations (if exists)
8. Glossary (if exists)
9. Introduction
10. State-of-the-Art or Related work
11. Proposed method
12. Experiment result
13. Conclusion and Future work
14. References, and
15. Appendix (if exists)

3.1.1 State-of-the-Art

State-of-the-Art (SoTA) is a step to demonstrate the novelty of your research results. The importance of being the first to demonstrate research results is a cornerstone of the research business[1].

Besides demonstrating the novelty of your research results, a SoTA has other important properties:

[1] "Why and how to write the state-of-the-art", by Babak A. Farshchian, May 22, 2007

1. It teaches you a lot about your research problem. By reading literature related to your research problem you will learn from other researchers and it will be easier for you to understand and analyze your problem;
2. It proves that your research problem has relevance;
3. It shows different approaches to a solution;
4. It shows what you can reuse from what others have done.

3.1.2 Related work

In the *Related Works* section, you should discuss briefly about published matter that technically relates to your proposed work[2]

A short summary of what you can include (but not limited to) in the Related Works section:

1. Work that proposes a different method to solve the same problem;
2. Work that uses the same proposed method to solve a different problem;
3. A method that is similar to your method that solves a relatively similar problem;
4. A discussion of a set of related problems that covers your problem domain.

3.2 Glossary and Nomenclature/List of Symbols

Many technical documents use terms or acronyms unknown to the general population. It is common practice to add a glossary to make such documents more accessible. A *glossary* is a nice thing to have in a report and usually very helpful. As you probably can imaging, it is very easy to create in Latex.

As with all packages, you need to load glossaries with `\usepackage`, but there are certain packages that must be loaded before glossaries, if they are required: `hyperref, babel, polyglossia, inputenc` and `fontenc`.

```
\usepackage{glossaries}
\makenoidxglossaries
```

[2] https://academia.stackexchange.com/questions/68164/how-to-write-a-related-work-section-in-computer-science

Once you have loaded `glossaries`, you need to define your terms in the preamble (or, separated file) and then you can use them throughout the document.

Next you need to define the terms you want to appear in the glossary. Again, this must be done in the preamble. This is done using the command

```
\newglossaryentry{<label>}{<key-val list>}
```

The first argument `<label>` is a unique label to allow you to refer to this entry in your document text. The entry will only appear in the glossary if you have referred to it in the document using one of the commands listed later. The second argument is a comma separated `<key>=<value>` list.

Inside the text you just need to use the command `\gls{name}` or `\glspl{name}` (plural name) to call it. For example, the following defines the term 'set' and assigns a brief description. The term is given the label set. This is the minimum amount of information you must give:

```
\newglossaryentry{set} % the label
{ name=set,             % the term
  description={a collection of objects}  % a brief description
}
```

Other example, now the glossary associated with a symbol, universal set:

```
\newglossaryentry{U}   % the label
{ name={universal set}, % the term
  description={the set of all things}  % a brief description
  symbol={\ensuremath{\mathcal{U}}}   % the associated symbol
}
```

Here's a simple example:

```
\usepackage{glossaries}
\newglossaryentry{ex}{name={sample},description={an example}}
\newacronym{svm}{SVM}{support vector machine}
\newacronym{beta}{$\beta$}{Second letter of the greek alphabet}
\newacronym{alpha}{$\alpha$}{First letter of the greek alphabet}
```

```
\begin{document}
Here's my \gls{ex} term. First use: \gls{svm}.
Second use: \gls{svm}.

\textit{I want the \gls{beta} to be listed after the \gls{alpha}}.
\end{document}
```

This produces: *Here's my sample term. First use: support vector machine (SVM).*
Second use: SVM.
I want the Second letter of the greek alphabet (β) to be listed after the First letter of the greek alphabet (α).

Do not use `\gls` in chapter or section headings as it can have some unpleasant side-effects. Instead use `\glsentrytext` for regular entries and one of `\glsentryshort`, `\glsentrylong` or `\glsentryfull` for acronyms. Alternatively use glossaries-extra which provides special commands for use in section headings, such as
`\glsfmtshort{<label>}`.

The plural of the word "matrix" is "matrices" not "matrixs", so the term needs the plural form set explicitly:

```
\newglossaryentry{matrix}% the label
{ name=matrix, % the term
  description={a rectangular table of elements},
plural=matrices % the plural
}
```

Given a set of numbers, there are elementary methods to compute its Greatest Common Divisor, which is abbreviated GCD. This process is similar to that used for the Least Common Multiple (LCM).

3.3 Importing Images

3.4 Floats Figures and Tables, and Captions

The tabular environment can be used to typeset tables with optional horizontal and vertical lines. LaTeX determines the width of the columns automatically. The first line of the environment has the form: `\begin{tabular}[pos]{table spec}`

table spec tells `LaTeX` the alignment to be used in each column and the vertical lines to insert.

pos can be used to specify the vertical position of the table relative to the baseline of the surrounding text.

The number of columns does not need to be specified as it is inferred by looking at the number of arguments provided. It is also possible to add vertical lines between the columns here.

Some notes are important to followed, such as present in Table **??**:

i) Not defined vertical lines;

ii) The legend must be on top;

iii) Use `\toprule`, `\midrule` and `\bottomrule` to draw horizontal lines.

Table 3.1: Table's rules.

Item		
Animal	Description	Price ($)
Gnat	per gram	13.65
	each	0.01
Gnu	stuffed	92.50
Emu	stuffed	33.33
Armadillo	frozen	8.99

There are two ways to incorporate images into your `LaTeX` document, and both use the graphicx package by means of putting the command `\usepackage{graphicx}` near the top of the `LaTeX` file, just after the documentclass command.

The two methods are

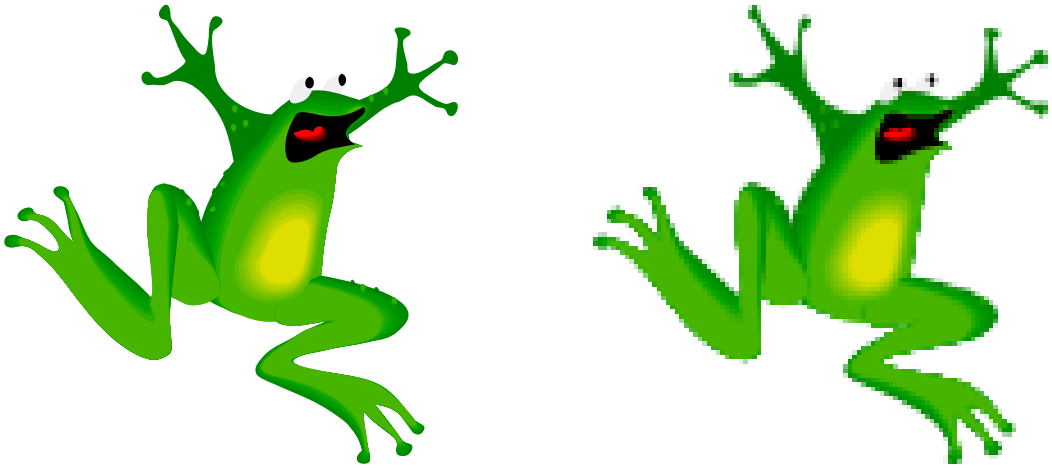
- include only PostScript images (esp. 'Encapsulated PostScript') if your goal is a PostScript document using dvips;
- include only PDF, PNG, JPEG and GIF images if your goal is a PDF document using pdflatex, `TeXShop`, or other PDF-oriented compiler.

Some PNG images within my `LaTeX` document. The quality of the image files is sufficient and the result using `LaTeX` and viewing the resulting DVI file is quite looks good.

To get the best quality of the images in PDF files I'd recommend using vector-based graphics for images. The best format to save images in is .pdf, see Figure **??**. With programs like `Inkscape`, you can draw as you would in MS Paint (and do much more), and because the images are vector-based instead of pixel-based, their quality should be preserved when converting to PDF in any way.

In all cases, each image must be in an individual 1-image file; no animation files or multipage documents.

There are two different ways to place two figures/tables side-by-side. More complicated figures with multiple images. You can do this using subfigure environments inside a figure environment. Subfigure will alphabetically number your subfigures and you have access to the complete reference as usual through `\ref{fig:figurelabel}`, Figure **??**, or Figure **??** using `\ref{fig:subfigurelabel}`.



(a) (b)

Figure 3.1: Subfigure example with vectorial and no-vectorial images

Using the package listings you can add non-formatted text as you would do with `\begin{verbatim}` but its main aim is to include the source code of any programming language within your document. If you wish to include pseudocode or algorithms see LaTeX/Algorithms_ and_Pseudocode, as Listing **??**.

```
59     private static void printSet(Set set) {  // comment
60         int[] elements = set.getElements();
61         System.out.print('{');
62         for (int i = 0; i < elements.length; i++) {
63             System.out.print(elements[i] + (i == elements.length
    -1 ? "" : ","));
64         }
```

Listing 3.1: Static method - SetApp

```
1  # comentário
2  square <- function(x) {
3      x^2
4      % |$x^{2}$|
5  }
6
7  # nerv
8  x <- c(1:100)
9  y <- square(x)
```

Listing 3.2: R-Code (Test).

3.5 Generating PDFs from LaTeX

3.5.1 Generating PDFs with pdflatex

You may create PDF files either by using `latex` to generate a DVI file, and then use one of the many DVI-2-PDF converters, such as `dvipdfm`.

Alternatively, you may use `pdflatex`, which will immediately generate a PDF with no intermediate DVI or PS files. In some systems, such as Apple, PDF is already the default format for LaTeX. I strongly recommend you to use this approach, unless you have a very good argument to go for `latex` + `dvipdfm`.

A typical pass for a document with figures, cross-references and a bibliography would be:

```
$ pdflatex template
$ bibtex template
$ pdflatex template (twice)
```

You will notice that there is a new PDF file in the working directory called `template.pdf`. Simple :)

Please note that, to be sure all table of contents, cross-references and bibliography citations are up-to-date, you must run `latex` once, then `bibtex`, and then `latex` twice.

3.5.2 Dealing with Images

You may process the same source files with both `latex` or `pdflatex`. But, if your text include images, you must be careful. `latex` and `pdflatex` accept images in different (exclusive) formats. For `latex` you may use EPS ou PS figures. For `pdflatex` you may use JPG, PNG or PDF figures. I strongly recommend you to use PDF figures in vectorial format (do not use bitmap images unless you have no other choice).

3.5.3 Creating Source Files Compatible with both latex and pdflatex

Do not include the extension of the file in the `\includegraphics` command, use: `\includegraphics{evolution_steps}`, and not:
`\includegraphics{ evolution_steps.png}`.
In the first form, `latex` or `pdflatex` will add an appropriate file extension.

This means that, if you plan to use only `pdflatex`, you need only to keep (preferably) a PDF version of all the images. If you plan to use also `latex`, then you also need an EPS version of each image.

To be included in the sections above

If you are writing only one or two documents and aren't planning on writing more on the same subject for a long time, maybe you don't want to waste time creating a database of references you are never going to use. In this case you should consider using the basic and simple bibliography support that is embedded within `LaTeX`.

`LaTeX` provides an environment called `thebibliography` that you have to use where you want the bibliography; that usually means at the very end of your

document, just before the `\end{document}` command. Here is a practical example:

```
\begin{thebibliography}{9}
\bibitem{lamport94}
  Leslie Lamport,
  \emph{\LaTeX: A Document Preparation System}.
  Addison Wesley, Massachusetts,
  2nd Edition,
  1994.
\end{thebibliography}
```

In this document, the bibliography is in a separate document: `bibliography.bib` where information is entered from https://scholar.google.pt/, as show Figure **??**.

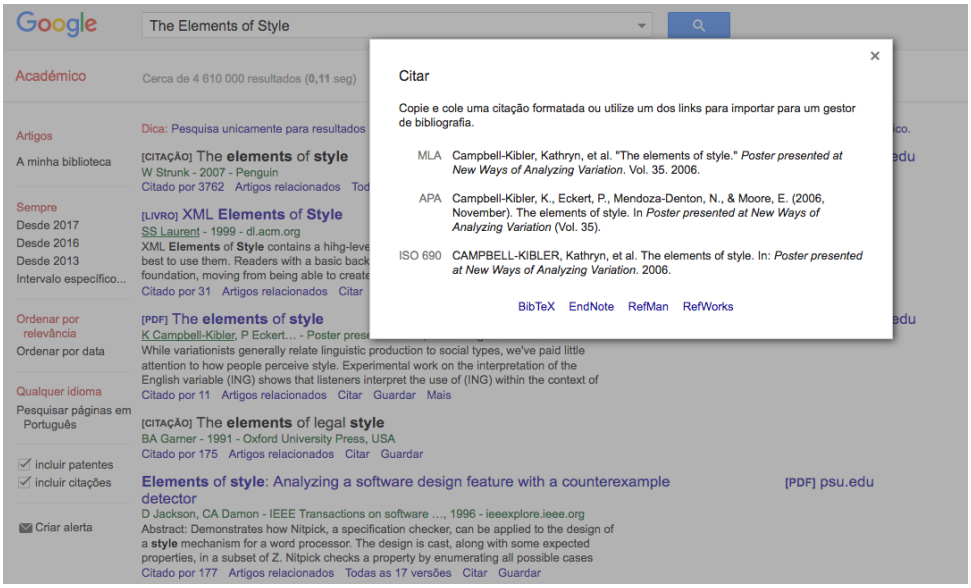


Figure 3.2: Screenshot from Scholar Google

To actually, cite a given document is *very* easy. Go to the point where you want the citation to appear, and use the following: `\cite{citekey}`, where the `citekey` is that of the `bibitem` you wish to cite, e.g `~\cite{lamport94}`. When `LaTeX` processes the document, the citation will be cross-referenced with the bibitems and replaced with the appropriate number citation. The advantage here, once again, is that `LaTeX` looks after the numbering for you.

When a sequence of multiple citations are needed, you should use a single `\cite{}` command. The citations are then separated by commas. Note that you must

not use spaces between the citations. Here's an result example [**strunk**, **chicago**, **texbook**].

Footnotes are a very useful way of providing extra information to the reader. Usually, it is non-essential information which can be placed at the bottom of the page. This keeps the main body of text concise.

The footnote facility is easy to use: `\footnote{Simple footnote}`[3].

3.6 Equations

Typesetting mathematics is one of `LaTeX`'s greatest strengths. It is also a large topic due to the existence of so much mathematical notation. It is recommend to read the following document available in Short Math Guide for LaTeX - AMS - American Mathematical Society.

3.7 Page orientation

The default page layout is "portrait", but sometimes it is still useful/necessary to have the whole document or only single pages changed to "landscape". The latter might be due to a large table or figure. If you want to make appear the left side up, better readable on screen, the pdflscape-package will do it: `\usepackage{pdflscape}`

and again:

```
\begin{landscape}
...
\end{landscape}
```

or, `\includepdf[landscape=true,pages={1}]{example.pdf}`

to put the page in "landscape", while the rest will remain in "portrait" orientation. Nevertheless, the header/footer will also be changed in orientation.

Written by Matilde Pós-de-Mina Pato with collaboration of Nuno Datia, 2012 October (1st version)

Written by Matilde Pós-de-Mina Pato, December 21, 2021 – version 2.5.3 (last version)

[3] Simple footnote

Applied Survival Analysis by Hosmer and Lemeshow

Stata Textbook Examples Applied Survival Analysis by Hosmer and Lemeshow
[newtest]

The data files used for the examples in this text can be downloaded in a zip file from the Wiley FTP website or the Stata Web site.

```
# The R package(s) needed for this chapter is the survival package.
# We currently use R 2.0.1 patched version. You may want to make sure
# that packages on your local machine are up to date. You can perform
# updating in R using update.packages() function.

# url: http://www.ats.ucla.edu/stat/r/examples/
# data set is hmohiv.csv.
hmohiv<-read.table("http://www.ats.ucla.edu/stat/r/examples/asa/hmohiv.
    csv", sep=",", header = TRUE)
attach(hmohiv)
hmohiv

# using the hmohiv data set. To control the type of symbol, a variable
    called psymbol is created.
# It takes value 1 and 2, so the symbol type will be 1 and 2.
psymbol<-censor+1
table(psymbol)

```

```
17 plot(age, time, pch=(psymbol))
18 legend(40, 60, c("Censor=1", "Censor=0"), pch=(psymbol))
19
20 age1<-1000/age
21 plot(age1, time, pch=(psymbol))
22 legend(40, 30, c("Censor=1", "Censor=0"), pch=(psymbol))
23
24 # Package "survival" is needed for this analysis and for most of the
      analyses in the book.
25 library(survival)
26 test <- survreg( Surv(time, censor) ~ age, dist="exponential")
27 summary(test)
28
29 pred <- predict(test, type="response")
30 ord<-order(age)
31 age_ord<-age[ord]
32 pred_ord<-pred[ord]
33 plot(age, time, pch=(psymbol))
34 lines(age_ord, pred_ord)
35 legend(40, 60, c("Censor=1", "Censor=0"), pch=(psymbol))
```