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

List of Figures

List of Tables

List of Listings

Acronyms

β Second letter of the greek alphabet. 33

α First letter of the greek alphabet. 33

GCD Greatest Common Divisor. 33

LCM Least Common Multiple. 33

Glossary

matrix a rectangular table of elements. 33

universal set the set of all things. 32

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 2.1, will be made an overview on previously developted work made on this subject, then, on section 2.2, will be made a characterization of the key concepts related to it. By last, on section 2.3, are presented and explained several technologies representative of the state of the Art on asynchronous data flow 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 flow key concepts and design alternatives

With the development of several approaches related to asynchronous data flow 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 flow, namely:

2.2.1 Synchronous versus Asynchronous

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

Asynchronous 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 process and can even be done in another machine. 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 operation before going forward, which limits the program efficiency if parallelism is applicable. To better visualize what was explained, we have the following examples:

```
// create a client
HttpClient client = HttpClient.newHttpClient();
// create a request
HttpRequest request = HttpRequest.newBuilder(
        URI.create("https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY"))
                .header("accept", "application/json")
                .build();

//not blocking http asynchronous request
var futureResponse = client
        .sendAsync(request, new JsonBodyHandler<>(DTO.class))
        .thenAccept(res -> System.out.println(res.body().get().title)); //
   consumer callback for when the operation finishes

futureResponse.get(); //blocks until the response is received
```

Listing (2.1) Asynchronous call example

```
// create a client
HttpClient client = HttpClient.newHttpClient();

// create a request
HttpRequest request = HttpRequest.newBuilder(

         URI.create("https://api.nasa.gov/planetary/apod?api_key=DEMO_KEY"))
         .header("accept", "application/json")
         .build();

// blocking http synchronous request
HttpResponse<Supplier<DTO>> response = client.send(request, new
   JsonBodyHandler<>(DTO.class));

// the response:
System.out.println(response.body().get().title);
```

Listing (2.2) Synchronous call example

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 remote operation is finished, consequently, the

caller operation is dependent from several variables to go forward e.g. : HTTP messaging latency, remote server operation speed or bandwidth issues. 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 the OS is ready to process the received responses from a remote server that handled the operation.

2.2.2 Push vs Pull

Another concept important to understand how asynchronous data flow 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:

```
Flowable<Long> flow = Flowable
                .interval(1, TimeUnit.SECONDS);

Iterator<Long> iterator = flow.blockingIterable().iterator();

while (iterator.hasNext()) {
    System.out.println(iterator.next());
}
//Output:
//0
//1
//2
//3
//.
//.
//9
```

Listing (2.3) Pull pattern example

```
Flowable<Long> flow = Flowable
                .interval(1, TimeUnit.SECONDS);

flow.blockingSubscribe(System.out::println);

//Output:
//1
//2
//3
//4
//5
```

Listing (2.4) Push pattern example

Figure 2.2: Example of Pull and Push data handling patterns

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 with *Reactive Streams* or asynchronous data processing in general, is the nature of the data flow. There are two main adjectives to name a data flow, `Hot` or `Cold`.

A `Cold` data flow, is a flow of information that is produced just when the stream pipeline in subscribed by an observer. In this case, the producer only starts sending/producing data when someone is interested in the data from that source. For example, when program uses a IO mechanism to lazily retrieve a sequence of words from a database, the IO mechanism will only start sending information just when a consumer subscribes that data flow. Usually, the data is sent to the consumer in unicast.

On the other hand, in a `Hot` data flow, the data is produced independently of existing any observer to that information. This mechanism usually work in broadcast and the data is continuously produced and sent to possible observers. In this case, when an observer subscribes to a publisher, exists the possibility of data items being already lost to that publisher while in the `Cold` flow, the consumer usually receives all items that were produced by the source. In the following examples, a number is produced each 100 milliseconds:

```
Flowable<Long> cold = Flowable.interval(100, TimeUnit.MILLISECONDS);
Thread.sleep(1000);
cold.blockingSubscribe(System.out::println); //prints: 0,1,2,3,4 .... no
   items lost
```

Listing (2.5) Cold Example

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

Listing (2.6) Hot Example

Figure 2.3: Example of Hot and Cold data streams

As we can see, in the `Hot` data flow example, the items are emitted from the moment the producer is created, independently of existing any subscriber or observer attached to that publisher. Notice that when a consumer is subscribed to the publisher, 1 seconds after the emittion started, the numbers from 0 to 10 were not printed.

In the `Cold` example, the producer only emits data when a subscription is done, and because of that, all the produced numbers were printed, in contrast with what happen in `Hot` stream, where data loss are almost certain.

2.2.4 Cancelables

As already stated above, Asynchronous operations may run outside the main program context. This implies, that the main program loses visibility and control on what happens in that asynchronous operations. Because of that, exists the need to put in place mechanisms of control that allow the main program to maintain control over an asynchronous operation to, for example, cancel the operation or put in place finishing logic that allow, for example: resource disposing, logging, decisions etc...

These mechanisms, are many times done through the concept of *cancelables*. Usually, a cancelable, is an entity that represents an operation that can canceled from an external entity, or, an entity that allows to set logic when an operation finishes by any reason.

In C#, a cancelable is an interface implemented by objects that represent asynchronous operations and provides the means to cancel asynchronous operations, on the fly. This is achieved through a mechanism named: `CancellationToken`, that is used to pass information through different execution threads.

In RXJava, a `Cancelable`, is a functional interface with the method `cancel()`. Then, the `Cancelable` can be associated to a data source representation, the `Observable`, by calling the method `Observable.setCancelable(Cancelable)`. When the `Observable` finishes or is canceled for any reason, the method `Cancelable.cance` will be called. This way, proper logic is put in place to handle an asynchronous operation cancelation.

As we saw, these two concepts of cancelable diverge. One, provides the means to cancel an operation on the fly and gives some control over the operation cancelation; the other, provides the means to control an operation cancelation independently of how it was cancelled.

2.2.5 Error Handling

In synchronous environments, usually, when something goes wrong, the way to handle an error in the majority of cases its by throwing an exception and propagate it until the proper code handles it, usually in a try/catch block. However, in cases when exists an asynchronous operation or when a continuous stream of data items are being received, that way of dealing with an error can imply several issues, e.g: exceptions not reaching main program, log losing or asynchronous flow blockage.

Since log losing or blocking a whole operation because of a badly handled error is unacceptable, the best way to deal with errors in asynchronous data flow it is to isolate the error. This way, the flow processing may continue in parallel while the error is properly handled.

The best way to handle this kind of errors, it is to have proper callbacks that are called when an error occurs on the stream item. This way, a function can handle the error properly, without the necessity to blocking any data stream processing, if avoidable and the proper logging and any additional measure to handle it can be put in place.

2.2.6 Intrisic Keywords

Asynchronous code is tendentiously harder to understand because, in opposition with what happens in synchronous code, the sequence of programming statements operations may not happen chronologically ordered. Because of that, many times it's difficult to debug and sustain asynchronous code.

For that reason, many languages started to add syntax techniques that allow the programmer to build asynchronous code that resembles the synchronous syntax. Under the hoods, the virtual machines that sustain these syntax mechanisms, handle the code bounded with that 'intrisic words'and builds asynchronous routines that the programmer will not be aware of; being this a way to abstract the programmer from the complexity of handling and sustaining complex asynchronous code.

One example of *intrinsic words* mechanisms, is the `async...await` keywords implemented in *Microsoft's .NET C#* and in *javascript*.

In the example 2.4 we can see an example of these keywords being used. Where, for example, we can observe in the line 15, a call to a asynchronous operation, and, by hadding the keyworkd `await`, the next statment although being to an asynchronous operation the statment orde looks like its synchronous.

Another example of 'intrisic words' used in asynchronous processing is the `for await...of`, used in javascript to proccess asynchronous enumerables.

On the sub chapters of 2.3 these examples will be explained with more depth.

2.2.7 Back-pressure

When the *pull* method is used to retrieve items from a source, the producer retrieves only the items it can process in the given time.

However, when the *push* approach is used as data retrieval method from asynchronous flows, the producers have the initiative to push items to its consumers. This can originate situations, where the producer emits items faster than the producers can handle, which can create problems like: unwanted loss of data, lack of responsiveness from consumers, etc...

To resolve these issues, were created strategies and design patterns that are commonly referred as *Backpressure*. There are four main approaches which the majority of *Backpressure* strategies are designed from and can be resumed as:

1. **DROP:** Producer drops items after a retrieving buffer gets full.
2. **Buffer everything:** A buffer, keeps all unprocessed items that are received. Usually, this strategy is used when all received items are critical for the business development and memory managment has flexibility to handle the increase of storage needs.
3. **Error:** An error is thrown when the buffer threshhold is reached, usally all items received after the threshhold is reached are discarded.
4. **Lastest:** Only the last received item in the given moment is kept.
5. **Missing:** No back-pressure strategy it is in place, all items that ca not be processed on arrival, are discarded.

2.3 State of the Art

In this section, we will be explaining the different frameworks that are the *State of the Art* in asynchronous data processing in several programming languages.

First of all, in the section 2.3.1, will be presented the multi-language `reactivex.io` project, and how the `Observer Pattern` is used in this project to implement real time asynchronous processing with and without *back-pressure*.

Secondly, in the section 2.3.2, will be presented the *.NET Async Enumerables* and how this approach diverges from approaches made in *reactivex.io*, *Javascript*, and in Kotlin with *Kotlin Flow*. In the sections 2.3.3 and 2.3.4, will be respectively presented the Javascript and the Kotlin Flow strategies for asynchronous data processing.

Finally, in the section 2.3.5, will be made an overview and taken conclusions about the different technologies presented, and how each approach can be used for different problems and objectives. Then, will be made a theoretical prediction on how each technology behaves in several know circumstances.

2.3.1 ReactiveX.io Project

After the historical context explained in the chapter 2.1, many groups of developers started to implement several solutions to handle asynchronous data flows. One of the most ambitious results of this proactivity is the *reactiveX* project.

This project gathers several ideas and known design patterns to implement an asynchronous data flow mechanism. The most positive points about this project implementation is that the mechanism is implemented in several programming languages using an open source approach and the solution design is done in a way, that all the complexity inherent with asynchronous programming is hidden from the programmer, making the code very intuitive and iterative an easily understandable for programmer that are not used to handle with asynchronous enviroments. In this dissertation, we will analyze the implementation of reactiveX made for Java, which is named: RXJava

2.3.1.1 RxJava architecture

To implement an asynchronous mechanism that can handle `Hot` and `Cold` data flows with and without back-pressure, the project adopted the *Observer pattern* allied with the concepts described in the *Publishes/subscriber pattern*. The designers justify this choice by saying that this model allows treating streams of asynchronous events with the same simplicity made in the synchronous counterparts e.g. in `Collection` and `Iterator`. The designer complement the justification by saying that, using this approach, the code complexity inherent with low-level threading, non-blocking IO, thread-safety etc... are completely abstracted from the programmer.

In the implementation, the data source/publisher of an asynchronous stream of events is represented by the classes `Observable` and `Flowable`. In this case, the only difference between `Observable` and `Flowable`, is that the class `Flowable` supports back-pressure through different bufferization strategies.

On a side note, it is important to refer that in RxJava, the same way `Observable` and `Flowable` represent a stream of N events, exists a single event is represented by a class named `Single`.

One of the great advantages of using this framework, is that the `Observable` and `Flowable`, although representing asynchronous data source, offer fluent API's that allows chaining operation fluently, like it is done in e.g, on java `Stream` implementation. In these fluent API's, stream processing operations like: `filter`, `flatMap` or `distinct` are available, allowing to treat these asycnhronous sequence of items like is made with synchronous streams.

On the other hand, the observers/subscribers are represented by consumers that are attached to `Observable` through the method `Observable.subscribe()`. The consumers can be either implementations of the functional interface `Consumer<T>` or implementations of the interface `Observer`.

The `Observer` interface is built with the intent to offer an extra control over the stream processing through error handling, which a `Consumer<T>` implementation lacks. The `Observer` can be viewed as the asynchronous representation of the java util interface `Iterator`, the resemblance can be seen by seeing the interface `Observer` methods:

1. `onSubscription()`: Method called right after the subscription is made with an `Observable`

2. `onNext(T item)`: Method called when an item is emmitted by the asynchronous data source.

3. `onError()`: In contrast with Iterator, the RxJava `Observer` is equipped to support error handling. This callback will be called when an error occurs.

4. `onComplete()`: Method called when the data source closes or if the subscription finishes.

Since its possible to make a relation between synchronous and its asynchronous logic counterparts in many languages, on next, we will take as example the relation between `Java` library and how these objects can be directly related to their asynchronous counterparts in `RXJava`. On the next table, it is possible to see how each object type can be mapped

	Single Item	Multiple Items
Java	`T getData()`	`Iterable<T> getData()`
RxJava	`Single<T> getData()`	`Observable<T> getData()`

2.3.2 Asynchronous Enumerables

To better understand what an asynchronous enumerable is, firstly, we must define what Enumerables are.

An enumerable, is the logical representation of a collection that can be iterated. In synchronous programming, this is usually made by creating a Collection class, that implements a mechanism that enables iteration over its items.

On the other side, in the asynchronous world, many times exists situations where a stream of data items are constantly arriving from a remote source, the commonly named *asynchronous Flows* or *asynchronous streams*.

The software designers saw, that did not exist many logical differences between a collection and real time data sources; the only difference was that a collection items could be retrieved on demand from memory, while the data flows are iterated over time when items become available for consumption.

Taking all this in account, has born the idea of creating the notion of the asynchronous enumerable, where the asynchronous enumerable is the representation of a constant source of readable items. With the use of this concept, were implemented several mechanisms, e.g. the AsyncEnumerables in C#, that simplify the iteration over asynchronous data sources. Making the code similar to synchronous statements, easier to maintain, and, once again, abstracting the programmer from many complex details inherent with asynchronous processing.

2.3.2.1 .NET Asynchronous Enumerables

`IEnumerable<T>` in C#, is a interface that implements a method `GetEnumerator()` that returns a `IEnumerator<T>`. In its turn, `IEnumerator<T>`, is a interface that allows to iterate over items of a Collection by offering a method `T getCurrent()`, that allows to retrieve the current item and a method `bool MoveNext()`, that allows to iterate to the next item of the sequence and check if the iteration has finished. `IAsyncEnumerable<T>`, keeps the same logic of `IEnumerable<T>`, but instead of returning a synchronous `IEnumerator<T>` with a synchronous `MoveNext()` method, returns a `IAsyncEnumerator<T>` where its method `bool MoveNextAsync()` is asynchronous.

On the other side, the keyword `yield`, is a syntax sugar introduced in **C# 2.0** that allows to avoid the implementation of `IEnumerable<T>` and `IEnumerator<T>` interfaces by the programmer, allowing iteration and the lazy return of items. All the work of implementing iteration interfaces is handled by `.NET` virtual machine in background. On top of that, the use of `yield` is compatible with asynchronous iteration involving `IAsyncEnumerable<T>`, which simplifies interation of asynchronous flows in C#, as we can see in the next example:

```
static async Task Main(string[] args)
{
    Console.WriteLine($"{DateTime.Now.ToLongTimeString()}: Start
    ");
    IAsyncEnumerable<int> enumerable = FetchItems(1000);
    int i = 0;
    await foreach (var item in enumerable)
    {
        if (i++ == 10){ break;}
        Console.WriteLine($"{DateTime.Now.ToLongTimeString()}: {
    item}");
    }
    Console.WriteLine($"{DateTime.Now.ToLongTimeString()}: End")
    ;
}

static async IAsyncEnumerable<int> FetchItems(int delay)
{
    int count = 0;
    while(true)
    {
        await Task.Delay(delay);
        yield return count++;
    }
}

//Output:
//01:00:01: Start
//01:00:02: 1
//01:00:03: 2
//01:00:04: 3
//01:00:05: 4
//.....
//01:00:05: 9
//01:00:05: End
```

Figure 2.4: C# Asynchronous Enumerables example

2.3.2.2 Javascript asynchronous iterables

Javascript tries to provide a functional and weakly typed approach to asynchronous flow processing, opposed to what we saw in the C# solution.

The keyworkd to provide the mechanism to iterate over asynchronous streams in relativity similar to what we saw in C#, with the implementation of the intrisic keywords `async...await` and `for await...of`.

With the keyword pair `async...await`, its provided the syntax "sugar" that enables to make asynchronous calls and the code syntax resembling synchronous. The `async` mark is used in an asynchronous function and enables the use of the keyword `await`. The `await` keyword is then used in an asynchronous call to another asynchronous function

On the other side, the keywords `for await...of`, provides support for iteration over asynchronous enumerables, as we can see in the next example:

```
async function* streamAsyncIterable(stream) {
    const reader = stream.getReader();
    try {
      while (true) {
        const { done, value } = await reader.read();
        if (done) {
          return;
        }
        yield value;
      }
    } finally {
      reader.releaseLock();
    }
  }

async function getResponseSize(url) {
    const response = await fetch(url);
    // Will hold the size of the response, in bytes.
    let responseSize = 0;

    const iterable = streamAsyncIterable(response.body);

    for await (const chunk of iterable) {
      // Incrementing the total response length.
      responseSize += chunk.length;
    }
    return responseSize;
}
```

Figure 2.5: Mozilla's Javascript Asynchronous Enumerables example

In this example, the function `streamAsyncIterable` read a stream of bytes and has the responsibility to lazily return the items as they become available.

On the other side, the function `getResponseSize`, the keyword `await` its followed by an asynchronous HTTP request and the mechanism `for wait...of` it is used to iterate over an asynchronous iterable stream. As we can observe, the code syntax is very similar to a synchronous iteration.

With the implementation of this mechanism, javascript simplifies in a very subtle way, how asynchronous streams are handled, allowing the code to be easily

readable and maintainable and abstracting the programming from: e.g. callbacks, tasks or completion tokens.

This solution is usually used in front-end environments, for example, to load several items from remote sources while a web page loads.

2.3.3 Kotlin Flow

Kotlin flow in a **??** compliant solution. Similarly to RxJava, Kotlin strongly typifies a stream of N events through the class `Flow<T>`.

Like we saw in RxJava, `Flow<T>` provides a fluent API of intermediate operators that allow item processing through the traditional stream processing push methods like: filter, map and . On othe se executed recurring to different execution contexts, for exemple, to change the running thread from the main programm to a custom thread scheduler.

Flows in kotlin have the caracteristic of always being cold, this means that the emission of items just start when the method `Flow.collect()` is called, as we can see in the next figure:

```
fun simple(): Flow<Int> = flow {
    println("Flow started")
    for (i in 1..3) {
        delay(100)
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    println("Calling simple function...")
    val flow = simple()
    println("Calling collect...")
    flow.collect { value -> println(value) }
    println("Calling collect again...")
    flow.collect { value -> println(value) }
}


Calling simple function...
Calling collect...
Flow started
1
2
3
```

Figure 2.6: Kotlin flow example

Relatively to the use of intrisic words, Kotlin offer the `suspension` key word that as the same

uses the function `emit` to do the same trick that `yield return` does in C#, which is to lazily return an item on the fly, when the item becomes available to be returned.

On next, we have an example of Kotlin Flow usage:

```
fun simple(): Flow<Int> = flow { // flow builder
    for (i in 1..3) {
        delay(100) // pretend we are doing something useful here
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
    launch {
        for (k in 1..3) {
            println("I'm not blocked $k")
            delay(100)
        }
    }
    simple().collect { value -> println(value) }
}

//output:
//I'm not blocked 1
//1
//I'm not blocked 2
//2
//I'm not blocked 3
//3
```

Figure 2.7: Kotlin flow example

As we can see, the fluent API is widely used to operate the `Flow`. This solution is often used in Android development for retrieving remote data.

2.3.4 Tecnhlogies comparation

As we saw, each technology have a set of properties that help to caracterize the solution. To help the caracherization of each technology documented, we have a relation between the caracteristics saw in the chapter 2.2.

	RxJava	**Kolin Flow**	**Javascript**	**C#**
Pull			x	x
Push	x	x		
Cancelable	x	x	x	x
Error Handling	x	x	x	x
Backpressure	x	Not aplicable	Not aplicable	Not aplicable
Intrisic words			x	x

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 3.1:

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

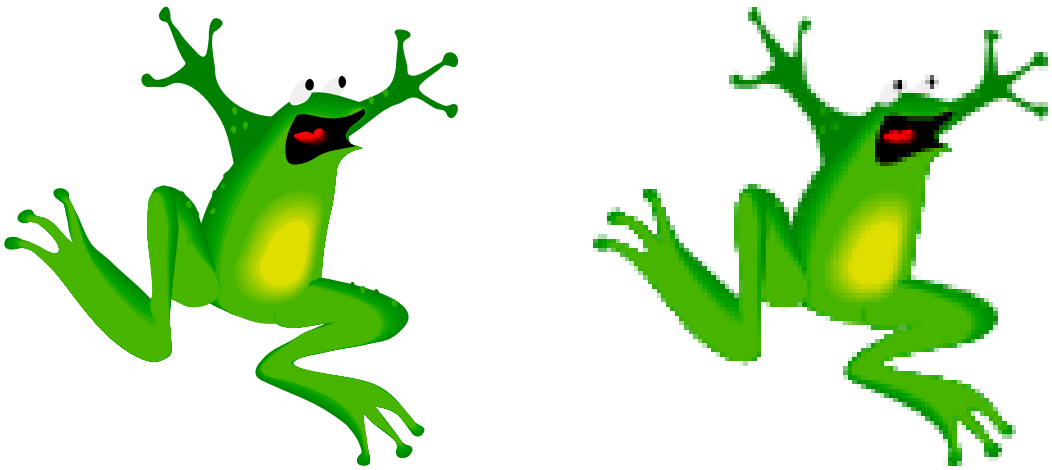
- include only PostScript images (esp. 'Encapsulated PostScript') if your goal is a PostScript document using dvips;
- include only PDF, PNG, JPEG and GIF images if your goal is a PDF document using pdflatex, `TeXShop`, or other PDF-oriented compiler.

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 3.1a. 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 3.1, or Figure 3.1b 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 3.1.

```
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 3.2.

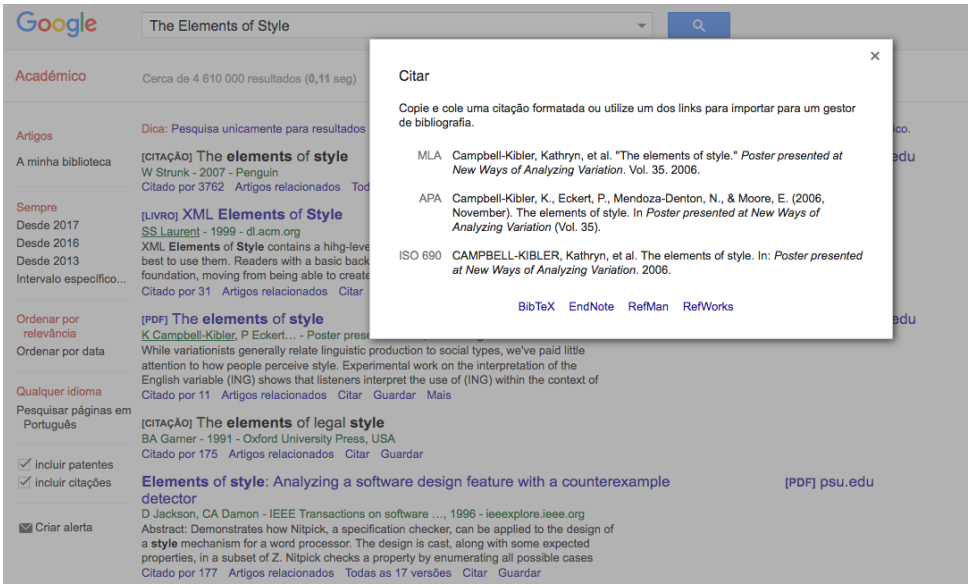


Figure 3.2: Screenshot from Scholar Google

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, February 10, 2022 – 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))
```