Sequence Alignment Algorithms

Algoritmi per l'Allineamento di Sequenze

Relatore: Prof. Giancarlo Mauri
Correlatore: Prof. Gianluca Della Vedova

Tesi di Laurea di:
Mauro Baluda
Matricola 038208

Part of ALiBio project
`http://www.alibio.org/`

Preface

ALiBio Goal

In a few words the goal of ALiBio is to give libraries and efficient fundamental algorithms to be used when developing applications in the bioinformatics field.

The large amount of biological data available nowadays makes the need of well-implemented efficient algorithm really stringent. Various projects (Open Bioinformatics Foundation) are already giving a number of libraries to the developer community.

The focus of those projects has always been on the ease of use, especially for people coming from the Biology field, and efficiency issues have not been the top priority.

On the other hand ALiBio is targetting the developer community with a strong CS background. ALiBio is suited for developing highly optimized applications where efficiency is paramount.

Two more issues that will receive a lot of attention in this project are testing and documentation. All libraries and algorithms that will be included in ALiBio must have an extensive suite of regression tests and must be clearly documented. In fact we require the use of noweb[10] for developing code to be included in ALiBio.

The advantages of Free Software (cfr.2.1) are well-known. ALiBio is a free library, at the same time it can be used also for developing non-free software. While we encourage the development of free software, we do not require it in order to use ALiBio.

Sequence Alignment Algorithms

Mauro Baluda

September 29, 2006

Contents

Chapter 1

Introduction

1.1 Objective of the stage

The purpose of this stage is to realize an efficient implementation of algorithms for optimal resolution of the Peerwise Sequence Alignment problem which has a great biological interest being a key to finding important regions in the genome, determining its functions and uncovering evolutionary forces. This particular stage is part of the ALiBio project[10] which aim is to provide a collection of useful libraries for the development of bioinformatics programs, with special regard to the efficiency of the implementation.

When I started the stage, the data structures, useful to represent biological information such as sequences of DNA, RNA and proteins, were already implemented. I will base my algorithm implementation upon these data structures.

1.2 Sequence Alignment

In detail, I will implement algorithms for global alignment (cfr.2.1) local alignment (cfr.2.1) and semiglobal alignment (cfr.2.1) following the specifications of the project. This algorithms are designed to find the best possible alignment (cfr.2.1) between two given sequences of symbols. Alignment algorithms are used to compare the sequence homology between two protein or DNA sequences. These programs find the best match between the two sequences. Occasionally gaps need to be introduced to make the two sequences align.

Global alignment algorithms attempt to match sequences from end to end, even though parts of the alignment are not very convincing. for example:

```
CATTTAGATT-C
  X|| |X|| X
--GTT-GTTTAT
```

On the other hand local alignment algorithms search for segments of the two sequences that match well. Using the same sequences as above, we could get:

```
  caTTTAgattc
    ||||
gttgTTTAt
```

Semiglobal alignments instead try to align a prefix of one of the sequences with a suffix of the other giving

```
       CATttagattc
        ||
gttgttt-AT
```

1.3 Specifications of the project

Developing the stage I followed the guide-lines defined by the AliBio project.

Literate Programming

The first requisite is to produce the job using a tool for literate programming[9] called noweb[10]. Literate programming is a methodology that combines a programming language with a documentation language, thereby making programs more robust, more portable, more easily maintained, and arguably more fun to write than programs that are written only in a high-level language. The main idea is to treat a program as a piece of literature, addressed to human beings rather than to a computer. The program is also viewed as a hypertext document, rather like the World Wide Web.

Languages

As documentation language ALiBio uses the LaTeXwriting system, which produces high quality scientific and mathematical documentation.

ALiBio project is implemented with the C++ programming language which, along with C, is probably the most used by developers of scientific applications all over the world. This choice guarantees a high portability with all the compilers that respect the ISO standard, and therefore a high compatibility of the code with any hardware platform.

Coding Conventions

To obtain a sufficient uniformity in the produced code developers should comply with the following conventions:

- Filenames: All in lower case and no spaces
- C++ files suffix: .cpp (programs), .hpp (libraries)

- Code lines length: All code lines must have 80 character at most
- The code: Must be written in lower case, using underscore as word separator
- Variable names: Descriptive names are required for global variables (eg. `string name;`), short names for local vars (eg. `string s;`)
- Functions names should be active verbs (eg. `get_seq()` , not `seq()`)
- Chunk names must not contain a name between double square brackets
- Printing functions: Generally, the output should not begin with any newline, but should end with a newline.
- Indenting: Four spaces with respect to the block. Four blanks, must be used. tab is not allowed
- Indentations must be used for iterations (while, do), conditional instructions (if), methods and declarations (public, private, protected); Indentations must not be used for code included inside namespaces or classes;
- Block structures: The left brace `{` must be on the same line as the first instruction of the block, separated by a space; The right brace `}` must be on a new line, in the same column as the beginning of the block;
- Comments should not be used frequently, and only used for improving code readability. The real documentation must be in the documentation chunks
- Whenever possible, all assertions should be placed between the declarations and the algorithmic sections
- Blank lines should be used for readability reasons
- Namespaces must be specified for all instructions and not-built-in types (eg. `std::cout`, not `cout`)
- ALiBio's namespace is alibio. All produced code must be within such namespace
- In order to avoid possible multiple inclusion, all libraries must check inclusions using opportune preprocessor instructions:
- The template construct (along with the type parameters), should be placed on the line before the class definition
- Each member functions implementation should be preceded by the template line too. Typically, when dealing with a complex return value, that too should be placed on a separate line. Moreover, the function name should be preceded by the class name(and type parameters), on the same line and separated by the scope operator.

Programming Paradigms

Besides exploiting the fundamental characteristics of C++ such as object orientation, the project also embraces the concept of Generic Programming as offered by the Standard Template Library[12] that widely used all over the project. Generic programming is a datatype-independent way of programming computers. This means that the same source code can be used regardless of the datatype the code will be instantiated with or passed as parameters. For example the type `vector` of the STL allows to define a vector of elements, whose type can be freely chosen at compile time, and at the same time it offers a whole set of efficient functions for its management. This fact guarantees a high flexibility using the libraries.

Useful libraries

To develop the project another important library has been used: the Boost Library[13] which offers a series of optimized tools for managing data structures as multidimensional matrices, lists, graphs and trees but also mathematical functions, input/output functions and many more.

Licence

From a legal and distributive point of view, ALiBio adopts a free software licence named LGPL, making the code and the documentation accessible to everyone.

External developers who want to contribute to ALiBio are wellcome, anyway to maintain the necessary uniformity of the final work, everyone should follow the project specifications and the chosen conventions for code and documentation writing

Anyone can freely include the AliBio libraries inside its own program or freely redistributes the libraries under the terms of the GNU Lesser General Public License.

1.4 Phases of development

Learning the tools

Before being able to proceed with the coding phase, I dedicated a period of time studying the necessary tools for the realization of the project. All the software used (noweb, LaTeX, Gnu C++ Compiler, CVS, GAWK) is freely available for the Linux operating system, so I had the opportunity to deepen my knowledge of Linux as a development platform and of the GNU development toolchain, in particular the Make program.

Although I already knew LaTeX[14], I was completely new to noweb[10] and the literate programming in general. I also needed to improve my knownledge about the CVS[15] versioning system.

Concerning the programming part of the work, I already had some C++ knowledge but I had to study the principles on which the Standard Library has been developed in particular the Templates mechanism.

In my work I used some functions taken form the Boost Library[13] so I had to study their on-line documentation.

I also studied the parts of ALiBio that had been already realized, In this way I learned not only how to use the library I needed but also the peculiar coding and documenting style adopted in the project.

Study of the algorithms

Once learned the necessary tools for the realization of the project, I went on studying the algorithms that I had to implement using the original articles written by their authors and a lot of other informations found on the World Wide Web.

Algorithm implementation

To implement the alignment algorithms, I needed to implement a specific class named ⟨*score_matrix* 8⟩ to contain the scoring schema needed for the alignment calculation. The object instantiated from this class will contain a value for each possible couple of elements of the used alphabet in a bi-dimensional matrix and the costs for inserting and extending gaps in the alignment. The class also implements the needed functions to set or modify these values.

The alignment class ⟨*seq_alignment* 25⟩ contains the functions that implement the alignment algorithms and a function to print the calculated alignment. An alignment object is constructed from a ⟨*score_matrix* 8⟩ object and two sequences in various possible formats, the alignment functions (which are the heart of the algorithm) actually calculate the alignment and prepare the output lines that will be printed by ⟨*print_alignment* 34⟩.

Realization of example programs

To facilitate the job of the developers who want to use the library, I wrote an example program which show how it is possible to include and to exploit the functionalities of the library in their programs. The program shows how to create the needed ⟨*score_matrix* 8⟩ object and modify its scoring system, then we show how to instantiate a ⟨*seq_alignment* 25⟩ object with two arbitrary sequences and calculate their different alignments. The program also show how to print the obtained result on standard output.

Testing

In the project a lot of importance is given to the tests phase. Once the code has been written, it's necessary to write some special tests able to verify its correctness. The adopted technique

is the *automatic regression tests* that allows to compare the actual output of the test with the expected output or the output of a previous version of the implementation. The project adopts some conventions to write these tests. The purpose is to check the input and the output attended for every single test and to automate their execution accomplished by a dedicated *script*. Respecting these conventions, the program is written to develop the tests on every single element of the library. The tests are performed in a particular order trying to avoid, as far as possible, the test of a function that calls a function not yet tested.

In my studies, two test programs have been implemented: one for testing the correctness and functionalities of the ⟨*score_matrix* 8⟩, the other one to verify the correct execution of the ⟨*seq_alignment* 25⟩ (which implements the alignment algorithms). The files used as input of the *script* which executes the test programs contain the input for each test and the expected output. This will be compared with the actual output by the *script* which establishes the success rather than the failure of the execution.

The implemented algorithms make widely use of low-level access to memory, C++ doesn't guarantee from memory leaks so we used a specific tool to detect eventual improper memory allocation and access.

To ensure the correctness of the algorithms over a large set of inputs I developed a randomized test that calls the alignment functions over random sequences and verifies the output. These test revealed a series of error in the implementation that weren't visible previously.

The testing phase of the work took a great amount of time, comparable to the actual development phase.

Analysing performances

Other libraries already exist to solve the peerwise alignment problem so we could test our implementations against EMBOSS[16] (one of the most widely used) to compare results and performances.

Writing the documentation

The documentation of the job (written in English) is constituted by:

1. an introductory part that describes the global project,
2. a section containing the necessary definitions for a better understanding of the code,
3. a wide description of the code to explain its functionality,
4. the description of the example programs and the tests,
5. the description of the performance analysis carried out
6. the synopsis tables containing, for each function, a brief description, the necessary prerequisites and the effect of its execution.

Chapter 2

Sequence alignment algorithms

2.1 Definitions

alphabet

An alphabet Σ is a non-empty finite set of characters.

sequence

A sequence is a list of elements of a given alphabet arranged in a "linear" fashion, such that the order of the members is well defined and significant.

alignment

A one-to-one matching of two sequences so that each character in a pair of sequences is associated with a single character of the other sequence or with a gap.

alignment score

A numerical value that describes the overall quality of an alignment. Higher numbers correspond to higher similarity.

match

In sequence alignment, the existence of the same character in a homologous position in both sequences.

mismatch

In sequence alignment, the existence of different characters in a homologous position in the two sequences.

gap

An insertion or a deletion. In sequence alignment, a pair containing a special null character "-".

gap opening penalty

The length-independent cost of opening a gap in a sequence alignment.

gap extension penalty

The cost of extending by one character a pre-existing gap in a sequence alignment.

linear gap penalty

A scoring system for gaps within alignments that charges a penalty for the existence of a gap proportional to the gap's length.

affine gap penalty

A scoring system for gaps within alignments that charges a penalty for the existence of a gap and additional penalty proportional to the gap's length.

global alignment

An optimal alignment that includes all characters from each sequence but may miss short regions of high local similarity.

Global alignments are most useful for closely related sequences of known homology.

local alignment

An optimal alignment that includes the most similar local region but may include only short portions of the sequences that were used to calculate the alignment.

Local alignments are especially useful for distantly related sequences.

semiglobal alignment

An optimal alignment that includes a prefix of one of the two sequences and a suffix of the other, in other words a global alignment where initial end terminal gaps are free.

Free software

Free software is a matter of the users' freedom to run, copy, distribute, study, change and improve the software. More precisely, it refers to four kinds of freedom, for the users of the software:

1. The freedom to run the program, for any purpose (freedom 0).
2. The freedom to study how the program works, and adapt it to your needs (freedom 1). Access to the source code is a precondition for this.
3. The freedom to redistribute copies so you can help your neighbor (freedom 2).
4. The freedom to improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this.

A program is free software if users have all of these freedoms. Thus, you should be free to redistribute copies, either with or without modifications, either gratis or charging a fee for distribution, to anyone anywhere. Being free to do these things means (among other things) that you do not have to ask or pay for permission.

You should also have the freedom to make modifications and use them privately in your own work or play, without even mentioning that they exist. If you do publish your changes, you should not be required to notify anyone in particular, or in any particular way.

More info at http://www.gnu.org/.

2.2 Problem formalization

The Sequence Alignment Problem[1] can be formalized as follows: Let Σ be a k-letter alphabet, and let A and B be two sequences over Σ. An alignment of strings $A = a_0, \ldots, a_{m-1}$ and $B = b_0, \ldots, b_{m-1}$ is a $2 \times l$ matrix M^0 $(l \geq n, m)$, such that the first (second) row of M^0 contains the characters of A (B) in order interspersed with $l - n$ $(l - m)$ spaces. We assume that no column of the alignment matrix contain two spaces.

For every pair of symbols (a_i, b_j) from Σ is defined a similarity score, the score of the alignment is defined as the sum of the scores of its columns.

The Sequence Alignment problem is to find the alignment of sequences A and B of maximal score.

2.3 Classic algorithms

In 1970 Saul Needleman and Christian Wunsch proposed[3] an algorithm for solving the SA problem which used a technique called Dynamic Programming and guaranties to find the global alignment (cfr.2.1) with the maximum score.

In 1981 Temple Smith and Michael Waterman proposed[4] a modifications of the original algorithm to find the optimal local alignment (cfr.2.1).

These algorithms are quadratic in time and space complexity and are the fundamental of all the other algorithms we will discuss in this paper.

2.3.1 Dynamic Programming

Dynamic Programming[2] was invented by the mathematician Richard Bellman in 1953, it is a method for reducing the runtime of algorithms exhibiting the properties of *overlapping subproblems* and *optimal substructure* where *overlapping subproblems* means that the problem can be broken down into subproblems which are reused several times and *optimal substructure* means that optimal solutions of subproblems can be used to find the optimal solutions of the overall problem.

2.3.2 Needleman-Wunsch algorithm

Every possible combination of chars in two sequences are represented in a 2dimensional array M^0 with an extra row and column added to allow the alignment to begin with a gap of any length in either sequence.

M^0	-	b_0	...	b_{j-1}	b_j	...	b_{n-1}
-	$M^0_{0,0}$	$M^0_{0,1}$	...	$M^0_{0,j}$	$M^0_{0,j+1}$	...	$M^0_{0,n}$
a_0	$M^0_{1,0}$						
$\vdots$	$\vdots$						
a_{i-1}	$M^0_{i,0}$			$M^0_{i,j}$	$M^0_{i,j+1}$		
a_i	$M^0_{i+1,0}$			$M^0_{i+1,j}$	$M^0_{i+1,j+1}$		
$\vdots$	$\vdots$						
a_{m-1}	$M^0_{m,0}$						$M^0_{m,n}$

Scores are filled in for each square, starting from $M^0_{1,1}$, searching for the maximum possible value using the recurrence below in which $M^0_{i+1,j+1}$ is the score of an optimal alignment between $A_i = a_0, \ldots, a_i$ and $B_j = b_0, \ldots, b_j$

$$M^0_{i+1,j+1} = \max \begin{cases} M^0_{i,j+1} + score(a_i, -) \\ M^0_{i+1,j} + score(-, b_j) \\ M^0_{i,j} + score(a_i, b_j) \end{cases}$$

The first raw and column are initialized with the score of an initial gap, the optimal alignment score can be read in $M^0_{m,n}$

2.3.3 Smith-Waterman algorithm

The local alignment problem correspond to finding the best global alignment between all of the substrings of A and B. An efficient way to achieve this is non-considering alignments with negative scores, this is the modified recurrence:

$$M^0_{i+1,j+1} = \max \begin{cases} 0 \\ M^0_{i,j+1} + score(a_i, -) \\ M^0_{i+1,j} + score(-, b_j) \\ M^0_{i,j} + score(a_i, b_j) \end{cases}$$

The largest value found all over M^0 represents the score of the best local alignment between A and B.

2.4 Affine gap penalties

Mutations are usually manifestations of errors in DNA replications. Nature frequently deletes or inserts entire substrings as a unit, as opposite to deleting or inserting individual nucleotides.

It is natural to assume that the score of a gap consisting of x spaces is not just the sum of scores of x spaces. In the Affine gap penalties scheme, the score for a gap of length x is $-(g + xe)$, where $g > 0$ is the penalty for the introduction of the gap and $e > 0$ is the penalty for adding a symbol to the gap.

To calculate affine gap penalties we need two more matrix: M^1 for keeping the best alignment ending with a gap in A and M^2 for the best alignment ending with a gap in B, we calculate the scores using the following three recursions:

$$M^1_{i+1,j+1} = \max \begin{cases} M^1_{i,j+1} - g \\ M^0_{i,j+1} - e \end{cases}$$

$$M^2_{i+1,j+1} = \max \begin{cases} M^0_{i+1,j} - g \\ M^0_{i+1,j} - e \end{cases}$$

$$M^0_{i+1,j+1} = \max \begin{cases} M^0_{i,j} + score(a_i, b_j) \\ M^1_{i+1,j+1} \\ M^2_{i+1,j+1} \end{cases}$$

2.5 Linear space alignment

When computing optimal sequence alignments time is not the only limiting factor, the algorithms above use a 2dimensional matrix leading to a space complexity $O(mn)$. Aligning genomes of different organisms, each 1 billion pairs long, would far exceed the RAM in current computers.

In 1988 Eugene W. Myers and Webb Miller published an article[5] that showed how to use a method proposed[6] by Hirshberg in 1975 to develop a linear-space version of the Alignment algorithm with affine gap penalties.

2.5.1 Linear space cost-only alignment

It's easy to compute $M^0_{m,n}$ in linear space because only the scores from the previous row of the matrix are needed therefore the alignment scores in the rows before i can be discarded while computing alignment scores for row $i+1$. This observation leads to an algorithm for a cost-only alignment with a space complexity $O(n)$.

2.5.2 Track back the alignment

Finding the actual alignment requires backtracking through the entire matrix after all of the rows and columns have been filled. To compute both the optimal score and alignment, one option would be to use a divide et impera approach[2].

Let's call A^{rev} and B^{rev} the reverse of A and B. $M^{0rev}_{i,j}$ is the optimal score of aligning $A^{rev}_i = a_{m-1}, \ldots, a_i$ and $B^{rev}_j = b_{n-1}, \ldots, b_j$.

As explained above the values of $M^{0rev}_{i,j}$ can be computed in time $O(mn)$ and space $O(n)$.

We can prove true the following equivalence:

$$M^0_{m,n} = \max_{k=0\ldots n} M^0_{\frac{m}{2},k} + M^{0rev}_{\frac{m}{2},n-k}$$

It states that the middle point of A has to align to some point in B, k is the optimal breakpoint in B obtained by maximizing over all possible ways of bisecting the sequence B aligning the first part with A until the middle point and the second part with A after the middle point.

M^0	-	b_0	...	b_{k-1}	...	b_{n-1}
-	$M^0_{0,0}$					
a_0						
$\vdots$						
$a_{\frac{m}{2}-1}$				$M^0_{\frac{m}{2},k}$		
$\vdots$						
a_{m-1}						$M^0_{m,n}$

The square $M^0_{\frac{m}{2},k}$ divides the matrix into four parts; the same algorithms is called recursively over the submatrices $[M^0_{0,0}, M^0_{\frac{m}{2},k}]$ and $[M^0_{\frac{m}{2},k}, M^0_{m,n}]$. Keeping track of the values assumed by k, one can determine also the actual alignment in linear space.

2.5.3 Time and space analysis of the algorithm

To compute the middle point of an alignment between to sequences of size m an n:

- Space: $2n$
- Time: $c \times mn$ for some constant c

Then , left and right calls cost $c \times (\frac{m}{2} \times k + \frac{m}{2} \times (n-k)) = \frac{c \times mn}{2}$

All recursive calls cost:

- Space: $O(n+m)$ to store the optimal alignment
- Time: $c \times mn + \frac{c \times mn}{2} + \frac{c \times mn}{4} + \ldots = 2c \times mn$ that is $O(mn)$

Chapter 3

Implementation

3.1 Library structure

To implement sequence alignment algorithms we decided to divide the work in two different classes:

1. ⟨*score_matrix* 8⟩ class implementing a scoring system.
2. ⟨*seq_alignment* 25⟩ class implementing the alignment algorithms.

Here is the file main structure:

1 ⟨*two_sequence.hpp* 1⟩≡

```
⟨licence 2⟩
#ifndef ALIBIO_TWO_SEQUENCE_INCLUDED
#define ALIBIO_TWO_SEQUENCE_INCLUDED
⟨include 3⟩
⟨namespace 6⟩
{
  ⟨score_matrix 8⟩
  ⟨seq_alignment 25⟩
}
#endif //ALIBIO_TWO_SEQUENCE_INCLUDED
```

Root chunk (not used in this document).

3.2 The licence

As stated before, ALiBio project is published under a free software licence named LGPL, the program begins with the opportune references

2 ⟨*licence* 2⟩≡

```
//ALiBio: Algorithms Library for Bioinformatics
//Copyright (C) 2002-2006:
//Gianluca Della Vedova, Riccardo Dondi, Luca Fossati,
//Lorenzo Mariani, Francesco Rossi, Mauro Baluda.
//
//This library is free software; you can redistribute it and/or
//modify it under the terms of the GNU Lesser General Public
//License as published by the Free Software Foundation; either
//version 2.1 of the License, or (at your option) any later version.
//
//This library is distributed in the hope that it will be useful,
//but WITHOUT ANY WARRANTY; without even the implied warranty of
//MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
//Lesser General Public License for more details.
//
//You should have received a copy of the GNU Lesser General Public
//License along with this library; if not, write to the Free Software
//Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307  USA
//
//http://bioinformatics.org/ALiBio
//
//Lab. Bioinformatica
//DISCo, Univ. Milano-Bicocca
//via Bicocca degli Arcimboldi 8
//20126 Milano (Italy)
```

This code is used in chunks 1, 49, 61, 76, 78, 83, and 91–93.

3.3 Included libraries

For our implementation we used a lot of external libraries: Let's begin with libraries from the standard c++

3 ⟨*include* 3⟩≡

```
#include <stdlib.h>
#include <assert.h>
#include <iostream>
#include <new>
```

This definition is continued in chunks 4 and 5.
This code is used in chunk 1.

We also needeed some data structures that was already implemented as parts of the ALiBio project to represent biological structures:

4 ⟨*include* 3⟩+≡
```
#include <alibio/sequence.hpp>
#include <alibio/bio_string.hpp>
#include <alibio/empty.hpp>
```
This code is used in chunk 1.

From the boost project we used the Boost.MultiArray library which provides a generic N-dimensional array concept definition and common implementations of that interface.

5 ⟨*include* 3⟩+≡
```
#include <boost/multi_array.hpp>
```
This code is used in chunk 1.

We will use the namespace std

6 ⟨*namespace* 6⟩≡
```
using namespace std;
```
This definition is continued in chunk 7.
This code is used in chunk 1.

Every element of this library belongs to the namespace `alibio`.

7 ⟨*namespace* 6⟩+≡
```
namespace alibio
```
This code is used in chunk 1.

3.4 The score_matrix class

Let's separate class interface from implementation

8 ⟨*score_matrix* 8⟩≡
⟨*score_matrix interface* 9⟩
⟨*score_matrix implementation* 13⟩
This code is used in chunk 1.

3.4.1 score_matrix interface

Here is the declaration of the score matrix class: it's a template class in which the parameter T is the type of the costs: we could use a small type (ex. short) to save memory or a floating point type (ex. double) if we need decimal positions. The default type for T has been set to float because of its flexibility

9 ⟨*score_matrix interface* 9⟩≡
```
template <class T=float>
class score_matrix{
private:
⟨private variables of score_matrix 10⟩
⟨private functions of score_matrix 11⟩
public:
⟨functions exported by score_matrix 12⟩
};
```
This code is used in chunk 8.

Private variables of score_matrix

The ⟨*score_matrix* 8⟩ class relies on the boost::multi_array for representing the actual matrix data, the alphabet is memorized in a vector<char> object according to the alibio::alphabet implementation. It also keeps an int for the alphabet size.

10 ⟨*private variables of score_matrix* 10⟩≡
```
boost::multi_array<T, 2> *data;
vector<char> alphabet;
T gap_cost, extend_gap_cost;
int lnt;
```
This code is used in chunk 9.

Private functions of score_matrix

The ⟨*score_matrix* 8⟩ class stores the alphabet in a vector, a recursive private function is needed to find a char's position in logaritmic time.

11 ⟨*private functions of score_matrix* 11⟩≡
```
int find_char_pos(char letter, int low, int high) const;
```
This code is used in chunk 9.

Functions exported by score_matrix

The ⟨*score_matrix* 8⟩ class provides several functions which offer a useful interface to the internal data structures. The following is a brief description of each function:

- ⟨*score_matrix constructor* 14⟩: Creates a score matrix for the given alphabet: default scores are 1 for matching and -1 for gaps and substitution of a symbol.
- ⟨*score_matrix destructor* 15⟩: frees memory after a deallocation.
- ⟨*set_score* 17⟩: Sets the score of switching between two given chars.
- ⟨*get_score* 18⟩: Returns the score of switching between two given chars.
- ⟨*set_match_score* 19⟩: Sets the score of symbol matches.
- ⟨*set_mismatch_score* 20⟩: Sets the score of symbol mismatches.
- ⟨*set_gap_cost* 21⟩: Sets the cost of inserting and extending a gap.
- ⟨*get_gap_cost* 22⟩: Returns the cost of inserting a gap.
- ⟨*get_extend_gap_cost* 23⟩: Returns the cost of extending a gap.
- ⟨*print* 24⟩: Prints on the standard output the score matrix.

12 ⟨*functions exported by score_matrix* 12⟩≡

```
score_matrix(const alibio::alphabet& alph);
~score_matrix();
void set_score(char a, char b, T val);
T get_score(char a, char b) const;
void set_match_score(T val);
void set_mismatch_score(T val);
void set_gap_cost(T val);
void set_gap_cost(T val, T ext_val);
T get_gap_cost() const;
T get_extend_gap_cost() const;
void print() const;
```

This code is used in chunk 9.

3.4.2 score_matrix implementation

Here is the implementation of the declared functions:

13 ⟨*score_matrix implementation* 13⟩≡
⟨*score_matrix constructor* 14⟩
⟨*score_matrix destructor* 15⟩
⟨*find_char_pos* 16⟩
⟨*set_score* 17⟩
⟨*get_score* 18⟩
⟨*set_match_score* 19⟩
⟨*set_mismatch_score* 20⟩
⟨*set_gap_cost* 21⟩
⟨*get_gap_cost* 22⟩
⟨*get_extend_gap_cost* 23⟩
⟨*print* 24⟩

This code is used in chunk 8.

Constructors implementation

The constructor implementation has an `alibio::alphabet` object as parmeter. It adds the symbol - to the given alphabet for representing a gap in the aligned sequence and then creates the ⟨*score_matrix* 8⟩ object using the `boost::multi_array` class. The size of the matrix is taken from the alphabet size and is kept in the `lnt` variable. Finally it populates the matrix with some default values: 1 for matching, -1 for symbol switching. Gap cost is initialized to 1.

14 ⟨*score_matrix constructor* 14⟩≡

```
template <class T>
score_matrix<T>::score_matrix(const alibio::alphabet &alph){

    vector<alibio::symbol> symbols=alph.get_all_symbols();
    lnt=symbols.size();
    alphabet.reserve(lnt);

    vector<alibio::symbol>::iterator It;
    for(It=symbols.begin(); It!=symbols.end(); It++)
        alphabet.push_back((*It).get_id());

    try{
        data=new boost::multi_array< T, 2>(boost::extents[lnt][lnt]);
    }
    catch(const bad_alloc& x){
        cerr<< "Out of memory in score_matrix(alphabet alph): "<<x.what()<<"\n";
        abort();
    }

    //we sort the vector for fast (logaritmic) searching
    sort(alphabet.begin(),alphabet.end());

    vector<char>::iterator It1, It2;
    for (It1=alphabet.begin(); It1!=alphabet.end(); It1++)
        for (It2=alphabet.begin(); It2!=alphabet.end(); It2++)
            if(It1==It2)
                this->set_score(*It1,*It2,(T)1); //default score for match
            else
                this->set_score(*It1,*It2,(T)-1); //default score for mismatch

    gap_cost=(T)1; //default cost for gap opening
    extend_gap_cost=(T)1; //default cost for gap extending
}
```

This code is used in chunk 13.

Destructor implementation

It frees the memory allocated by the constructor

15 ⟨*score_matrix destructor* 15⟩≡

```
template <class T>
score_matrix<T>::~score_matrix(){
    delete data;
}
```

This code is used in chunk 13.

Finding a char

A function for finding a char in the alphabet with logaritmic cost.

We could get a constant cost with an hash table but alphabets are normally small and calculating the hashes would take long...

16 ⟨*find_char_pos* 16⟩≡

```
template <class T>
int score_matrix<T>::find_char_pos(char letter, int low, int high) const{
  if (high < low) return -1;

  int mid = (high + low) / 2;

  if ( alphabet[mid] == letter) return mid;

  if ( alphabet[mid] > letter) {
    return find_char_pos (letter, low, mid-1);
  } else {
    return find_char_pos (letter, mid+1, high);
  }
}
```

This code is used in chunk 13.

Setting the score

This function permits to set the score of an alignment beetween the chars `a` and `b` to the value `val` which are its parameters, if the chars don't belong to the alphabet, the function aborts.

17 ⟨*set_score* 17⟩≡

```
//abort if doesn't find the given char
template <class T>
void score_matrix<T>::set_score(char a, char b,  T val){
    int x=find_char_pos(a, 0, lnt);
    int y=find_char_pos(b, 0, lnt);

    if(x==-1){
        cerr << "Error: char "<< a<<" not present in the alphabet";
        abort();
    }
    if(y==-1){
        cerr << "Error: char "<<b <<" not present in the alphabet";
        abort();
    }

    (*data)[x][y]=val;
    return;
}
```

This code is used in chunk 13.

Getting the score

This function returns the score of alignment between the chars `a` and `b` passed as parameters, if the chars don't belong to the alphabet, the function aborts.

18 ⟨*get_score* 18⟩≡

```
//abort if doesn't find the given char
template <class T>
T score_matrix<T>::get_score(char a, char b) const{
    int x=find_char_pos(a, 0, lnt);
    int y=find_char_pos(b, 0, lnt);

    if(x==-1){
        cerr << "Error: char "<< a<<" not present in the alphabet";
        abort();
    }
    if(y==-1){
        cerr << "Error: char "<<b <<" not present in the alphabet";
        abort();
    }

    return (*data)[x][y];
}
```

This code is used in chunk 13.

Setting score for matching

This function permits to set the score of an alignment beetween any two identical chars to a desired value `val`.

19 ⟨*set_match_score* 19⟩≡
```
template <class T>
void score_matrix<T>::set_match_score(T val){
    for (int i=0;i<lnt;i++)
        (*data)[i][i]=val;
}
```
This code is used in chunk 13.

Setting score for mismatching

This function permits to set the score of an alignment beetween any two different chars to a desired value `val`.

20 ⟨*set_mismatch_score* 20⟩≡
```
template <class T>
void score_matrix<T>::set_mismatch_score(T val){
    for (int i=1;i<lnt;i++)
        for (int j=0;j<i;j++){
            (*data)[i][j]=val;
                (*data)[j][i]=val;
        }
}
```
This code is used in chunk 13.

Setting the cost for opening and extending gaps

These functions permit to set the cost for opening and extending a gap in the alignment, if no extending gap cost is passed as parameter the opening gap cost will be used.

21 ⟨*set_gap_cost* 21⟩≡
```
template <class T>
void score_matrix<T>::set_gap_cost(T val){
    gap_cost=val;
    extend_gap_cost=val;
}

template <class T>
void score_matrix<T>::set_gap_cost(T val, T ext_val){
    gap_cost=val;
    extend_gap_cost=ext_val;
}
```
This code is used in chunk 13.

Getting the cost for opening and extending gaps

22 ⟨*get_gap_cost* 22⟩≡
```
template <class T>
 T score_matrix<T>::get_gap_cost() const{
    return gap_cost;
}
```
This code is used in chunk 13.

23 ⟨*get_extend_gap_cost* 23⟩≡
```
template <class T>
T score_matrix<T>::get_extend_gap_cost() const{
    return extend_gap_cost;
}
```
This code is used in chunk 13.

Printing the score matrix

A function that prints on the standard output a visualizations of the alphabet used, of the score matrix and gap costs

24 ⟨*print* 24⟩≡
```
template <class T>
void score_matrix<T>::print() const{
    //printing the score matrice
    int m=lnt;

    cout << "\t";
    for (int j=0;j<m;j++){
        cout<< alphabet[j] << "\t";
    } cout << "\n";

    for (int i=0;i<m;i++){
        cout << alphabet[i] << "\t";
        for (int j=0;j<m;j++){
            cout <<  (*data)[i][j] << "\t";
        }
        cout <<  "\n";
    }

    cout << "gap cost=" << gap_cost << "\n";
    cout << "gap extension cost=" << extend_gap_cost << "\n";
}//end print()
```
This code is used in chunk 13.

3.5 Sequence alignment class

We need to separate class interface from implementation

25 ⟨*seq_alignment* 25⟩≡
⟨*seq_alignment interface* 26⟩
⟨*seq_alignment implementation* 30⟩
This code is used in chunk 1.

3.5.1 `seq_alignment` interface

Here is the declaration of the `seq_alignment` class: it's a template class in which the parameter `T` is the type of the costs: default value is `float`.

26 ⟨*seq_alignment interface* 26⟩≡
```
template <class T=float>
class seq_alignment{
    private:
⟨private variables of seq_alignment 27⟩
⟨private functions of seq_alignment 28⟩
    public:
⟨functions exported by seq_alignment 29⟩
};
```
This code is used in chunk 25.

Private variables of seq_alignment

The ⟨*seq_alignment* 25⟩ class uses the `boost::multi_array` class for storing the matrix needed for its calculations

27 ⟨*private variables of seq_alignment* 27⟩≡

```
//the score matrix
const score_matrix<T> *c_matrix;
//gap creation ad extention costs;
T gap_cost, extend_gap_cost;

//the sequences
const vector<char> *A, *B;
//sequence lengths
int m, n;

//Matrix for quadratic-space alignments
boost::multi_array<T, 2> *N;
//Matrix for affine-gap alignments
boost::multi_array<T, 3> *M;
//Second vectors for linear-space alignments
boost::multi_array<T, 3> *M1;

//start and end of the alignment
int min_m, min_n, max_m, max_n; //inizialized by constructors
//score of the alignment
T max_score;
//buffers for memorizing the alignment
vector<char> *line1, *line2, *line3;

//infinite negative for type T
T NEG_INFTY;
```

This code is used in chunk 26.

Private functions of seq_alignment

These three private functions make the actual calculations needed to build the alignments.

28 ⟨*private functions of seq_alignment* 28⟩≡

```
void build_alignment(int i, int j);
void build_affine_alignment(int i, int j);
void build_linear_alignment(int i1, int j1, int i2, int j2,
                            T A_begin_gap_cost, T A_end_gap_cost,
                            T B_begin_gap_cost, T B_end_gap_cost);
```

This code is used in chunk 26.

Functions exported by seq_alignment

29 ⟨*functions exported by seq_alignment* 29⟩≡

```
    //constructors
    seq_alignment(const score_matrix<T>& scores,
                  const vector<char>& fst, const vector<char>& snd);
    seq_alignment(const score_matrix<T>& scores,
                  string fst, string snd);
    seq_alignment(const score_matrix<T>& scores,
                  alibio::bio_string fst, alibio::bio_string snd);
    //destructor
    ~seq_alignment();

    //prints the last calculated alignment
    void print_alignment();

//classic alignment functions
    //global - Needleman-Wunsch
    void nw_align();
    //local - Smiths-Waterman
    void sw_align();

//alignment functions using affine gap penalties
    //global
    void nw_affine_align();
    //local
    void sw_affine_align();
    //semiglobal
    void semiglobal_affine_align();

//alignment functions using linear space - Myers-Miller
    //global
    void nw_linear_align();
    //local
    void sw_linear_align();
    //semiglobal
    void semiglobal_linear_align();
```

This code is used in chunk 26.

3.5.2 `seq_alignment` implementation

The implementations of the declared functions

30 ⟨*seq_alignment implementation* 30⟩≡
⟨*seq_alignment constructors* 31⟩
⟨*seq_alignment destructor* 33⟩

⟨*print_alignment* 34⟩

⟨*nw_align* 35⟩
⟨*sw_align* 36⟩
⟨*build_alignment* 37⟩

⟨*nw_affine_align* 38⟩
⟨*sw_affine_align* 39⟩
⟨*semiglobal_affine_align* 40⟩
⟨*build_affine_alignment* 42⟩

⟨*nw_linear_align* 44⟩
⟨*sw_linear_align* 45⟩
⟨*semiglobal_linear_align* 46⟩
⟨*build_linear_alignment* 43⟩

This code is used in chunk 25.

Constructors implementation

The three constructors take a `score matrix` and `two sequences` and instantiate the needed memory. The `seq_alignment T` parameter must be of the same type of the corresponding `cost_matrix T` parameter, the compiler takes care of point out eventual misuses.

The constructors differs from the type of the sequences passed as parameters, respectively `const vector<char>&`, `string` and `alibio::bio_string`:

31 ⟨*seq_alignment constructors* 31⟩≡

```
template <class T>
seq_alignment<T>::seq_alignment(const score_matrix<T>& scores,
                                const vector<char>& fst, const vector<char>& snd){
    try{
        A=new vector<char>(fst);
        B=new vector<char>(snd);
    }
    catch(const bad_alloc& x){
        cerr << "Out of memory in seq_alignment constructor: "<< x.what()<<"\n";
        abort();
    }

    ⟨common initializations 32⟩
}//end seq_alignment(score_matrix, vector<char>, vector<char>)

template <class T>
seq_alignment<T>::seq_alignment(const score_matrix<T>& scores,
                                string fst, string snd){
    try{
        A=new vector<char>(fst.begin(),fst.end());
        B=new vector<char>(snd.begin(),snd.end());
    }
    catch(const bad_alloc& x){
        cerr << "Out of memory in seq_alignment constructor: "<<x.what()<<"\n";
        abort();
    }

    ⟨common initializations 32⟩
}//end seq_alignment(score_matrix, string, string)

template <class T>
seq_alignment<T>::seq_alignment(const score_matrix<T>& scores,
                                alibio::bio_string fst, alibio::bio_string snd){
    try{
        A=new vector<char>(fst.get_sequence().begin(),fst.get_sequence().end());
        B=new vector<char>(snd.get_sequence().begin(),snd.get_sequence().end());
    }
    catch(const bad_alloc& x){
        cerr << "Out of memory in seq_alignment constructor: "<<x.what()<<"\n";
        abort();
    }
```

⟨*common initializations* 32⟩
}//end seq_alignment(score_matrix, bio_string, bio_string)

This code is used in chunk 30.

All of the constructors need to initialize the same local variables, so we use a single code chunk

32 ⟨*common initializations* 32⟩≡

```
c_matrix=&scores;
gap_cost=c_matrix->get_gap_cost();
extend_gap_cost=c_matrix->get_extend_gap_cost();
NEG_INFTY=-numeric_limits<T>::max()/2;

m=A->size(); n=B->size();

//reinizialize min_m min_n max_m max_n max_score
min_m=0; max_m=m; min_n=0; max_n=n;
max_score=NEG_INFTY;

try{
    line1=new vector<char>();
    line2=new vector<char>();
    line3=new vector<char>();
}
catch(const bad_alloc& x){
    cerr << "Out of memory:" << x.what() << "\n";
    abort();
}
```

This code is used in chunk 31.

Destructor implementation

33 ⟨*seq_alignment destructor* 33⟩≡

```
template <class T>
seq_alignment<T>::~seq_alignment(){
    delete A;
    delete B;
    delete line1;
    delete line2;
    delete line3;
}//~seq_alignment()
```

This code is used in chunk 30.

Printing the alignment

This functions prints on the standard output the calculated alignment and it's score.

34 ⟨*print_alignment* 34⟩≡

```
template <class T>
void seq_alignment<T>::print_alignment(){

    vector<char>::iterator It;

    cout << min_m+1 << "\t";
    for (It=line1->begin(); It!=line1->end(); It++)
        cout << (*It);
    cout << "\t" << max_m << "\n";

    cout << "\t";
    for (It=line2->begin(); It!=line2->end(); It++)
        cout << (*It);
    cout << "\n";

    cout << min_n+1 << "\t";
    for (It=line3->begin(); It!=line3->end(); It++)
        cout << (*It);
    cout << "\t" <<  max_n << "\n";

    cout << "Alignment total score:" << max_score << "\n";

⟨debug_random 75⟩
}//end print_alignment()
```

This code is used in chunk 30.

3.5.3 Classic algorithms implementation

Needleman-Wunsch algorithm implementation

35 ⟨*nw_align* 35⟩≡

```
template <class T>
void seq_alignment<T>::nw_align(){

    //allocating matrix space
    try {
        N=new boost::multi_array< T, 2>(boost::extents[m+1][n+1]);
    }
    catch(const bad_alloc& x){
        cerr << "Out of memory in nw_align(): " << x.what() << "\n";
        abort();
    }
    //reinizialize min_m min_n max_m max_n max_score
    min_m=0; max_m=m; min_n=0; max_n=n;
    max_score=NEG_INFTY;

    (*N)[0][0]=(T)0;

    for (int j=1;j<=n;j++){
        (*N)[0][j]=-j*gap_cost;
    }

    for (int i=0;i<m;i++){
        int next_i=i+1;
        (*N)[next_i][0]=(*N)[i][0]-gap_cost;
        for (int j=0;j<n;j++){
            int next_j=j+1;
            (*N)[next_i][next_j]=max(max((*N)[i][j]+c_matrix->get_score((*A)[i],
                                                                        (*B)[j]),
                                         (*N)[i][next_j]-gap_cost),
                                     (*N)[next_i][j]-gap_cost);
        }
    }

    max_score=(*N)[m][n];

    //cleaning an eventual precalculated alignment
    (*line1).clear();
    (*line2).clear();
    (*line3).clear();
    build_alignment(max_m, max_n);
```

⟨*memory_line* 82⟩

```
    delete N;
}//end nw_align()
```

This code is used in chunk 30.

Smith-Waterman algorithm implementation

36 ⟨*sw_align* 36⟩≡

```
template <class T>
void seq_alignment<T>::sw_align(){

    //allocating matrix space
    try{
        N=new boost::multi_array< T, 2>(boost::extents[m+1][n+1]);
    }
    catch(const bad_alloc& x){
        cerr << "Out of memory in sw_align(): " << x.what() << "\n";
        abort();
    }
    //reinizialize min_m min_n max_m max_n max_score
    min_m=0; max_m=m; min_n=0; max_n=n;
    max_score=0;

    for (int i=0;i<=m;i++)
        (*N)[i][0]=(T)0;
    for (int j=0;j<=n;j++)
        (*N)[0][j]=(T)0;

    for (int i=0;i<m;i++){
        int next_i=i+1;
        for (int j=0;j<n;j++){
         int next_j=j+1;
        (*N)[next_i][next_j]=max(max(max((*N)[i][j]+c_matrix->get_score((*A)[i],
                                                                        (*B)[j]),
                                         (*N)[i][next_j]-gap_cost),
                                 (*N)[next_i][j]-gap_cost),
                             (T)0);
            if((*N)[next_i][next_j]>(*N)[max_m][max_n]){
                max_m=next_i;
                max_n=next_j;
            }
        }
    }

    max_score=(*N)[max_m][max_n];

    //cleaning an eventual precalculated alignment
    (*line1).clear();
    (*line2).clear();
    (*line3).clear();
    if (max_score>0)
        build_alignment(max_m, max_n);
    else
        max_score=0;
```

⟨*memory_line* 82⟩

```
    delete N;
}//end sw_align()
```

This code is used in chunk 30.

Tracking back the alignment

37 ⟨*build_alignment* 37⟩≡

```
template <class T>
void seq_alignment<T>::build_alignment(int i, int j){
    int prev_i=i-1;
    int prev_j=j-1;

    if((i>0) && (j>0) &&
        ((*N)[i][j]==((*N)[prev_i][prev_j]+c_matrix->get_score((*A)[prev_i],
                                                               (*B)[prev_j])))){
        build_alignment(prev_i, prev_j);
        line1->push_back((*A)[prev_i]);
        if((*A)[prev_i]==(*B)[prev_j])//matching
            line2->push_back('|');
        else
            line2->push_back('X');
        line3->push_back((*B)[prev_j]);
        return;
    }
    if((i>0) && ((*N)[i][j]==((*N)[prev_i][j]-gap_cost)) ){
        build_alignment(prev_i,j);
        line1->push_back((*A)[prev_i]);
        line2->push_back(' ');
        line3->push_back('-');
        return;
    }
    if((j>0) && ((*N)[i][j]==((*N)[i][prev_j]-gap_cost)) ){
        build_alignment(i,prev_j);
        line1->push_back('-');
        line2->push_back(' ');
        line3->push_back((*B)[prev_j]);
        return;
    }

    min_m=i;
    min_n=j;
}//end build_alignment(int, int)
```

This code is used in chunk 30.

3.5.4 Aligning using affine gap penalties

Global alignment implementation

38 ⟨*nw_affine_align* 38⟩≡

```
template <class T>
void seq_alignment<T>::nw_affine_align(){

    //allocating matrix space
    try{
        M=new boost::multi_array< T, 3>(boost::extents[m+1][n+1][4]);
    }
    catch(const bad_alloc& x){
        cerr << "Out of memory in nw_affine_align(): " << x.what() << "\n";
        abort();
    }
    //reinizialize min_m min_n max_m max_n max_score
    min_m=0; max_m=m; min_n=0; max_n=n;
    max_score=NEG_INFTY;

    //preparing first line of the 4 matrix
    (*M)[0][0][0]=(T)0;
    (*M)[0][0][1]=NEG_INFTY;
    (*M)[0][0][2]=NEG_INFTY;
    (*M)[0][0][3]=(T)3;//STOP HERE

    for (int j=0;j<n;j++){
        int next_j=j+1;
        (*M)[0][next_j][0]=-(gap_cost+j*extend_gap_cost);
        (*M)[0][next_j][1]=NEG_INFTY;
        (*M)[0][next_j][2]=-(gap_cost+j*extend_gap_cost);
        (*M)[0][next_j][3]=(T)2;//LEFT
    }

    for (int i=0;i<m;i++){
        int next_i=i+1;
        //preparing first column of the 4 matrix
        (*M)[next_i][0][0]=-(gap_cost+i*extend_gap_cost);
        (*M)[next_i][0][1]=-(gap_cost+i*extend_gap_cost);
        (*M)[next_i][0][2]=NEG_INFTY;
        (*M)[next_i][0][3]=(T)1;//UP

        for (int j=0;j<n;j++){
            int next_j=j+1;
            ⟨affine elaboration 41⟩
        }
    }

    max_score=(*M)[m][n][0];

    //cleaning an eventual precalculated alignment
```

```
    (*line1).clear();
    (*line2).clear();
    (*line3).clear();
    build_affine_alignment(m, n);
```

⟨*memory_line* 82⟩

```
    delete M;
}//end nw_affine_align()
```

This code is used in chunk 30.

Local alignment implementation

39 ⟨*sw_affine_align* 39⟩≡

```
template <class T>
void seq_alignment<T>::sw_affine_align(){

    //allocating matrix space
    try{
        M=new boost::multi_array< T, 3>(boost::extents[m+1][n+1][4]);
    }
    catch(const bad_alloc& x){
        cerr << "Out of memory in sw_affine_align(): " << x.what() << "\n";
        abort();
    }
    //reinizialize min_m min_n max_m max_n max_score
    min_m=0; max_m=m; min_n=0; max_n=n;
    max_score=0;

    //preparing first row of the 4 matrix
    (*M)[0][0][0]=(T)0;
    (*M)[0][0][1]=NEG_INFTY;
    (*M)[0][0][2]=NEG_INFTY;
    (*M)[0][0][3]=(T)3;//STOP HERE

    for (int j=0;j<n;j++){
        int next_j=j+1;
        (*M)[0][next_j][0]=(T)0;
        (*M)[0][next_j][1]=NEG_INFTY;
        (*M)[0][next_j][2]=-(gap_cost+j*extend_gap_cost);
        (*M)[0][next_j][3]=(T)3;//STOP HERE
    }

    for (int i=0;i<m;i++){
        int next_i=i+1;
        //preparing first column of the 4 matrix
        (*M)[next_i][0][0]=(T)0;
        (*M)[next_i][0][1]=-(gap_cost+i*extend_gap_cost);
        (*M)[next_i][0][2]=NEG_INFTY;
        (*M)[next_i][0][3]=(T)3;//STOP HERE

        for (int j=0;j<n;j++){
            int next_j=j+1;
            ⟨affine elaboration 41⟩

            if((T)0>(*M)[next_i][next_j][0]){
                (*M)[next_i][next_j][0]=(T)0;
                (*M)[next_i][next_j][3]=(T)3;//STOP HERE
            }

            if((*M)[next_i][next_j][0]>max_score){
                max_score=(*M)[next_i][next_j][0];
```

```
                max_m=next_i;
                max_n=next_j;
            }
        }
    }

    //cleaning an eventual precalculated alignment
    (*line1).clear();
    (*line2).clear();
    (*line3).clear();
    if (max_score>0)
        build_affine_alignment(max_m, max_n);
    else
        max_score=0;
```

⟨*memory_line* 82⟩

```
    delete M;
}//end sw_affine_align()
```

This code is used in chunk 30.

Semiglobal alignment implementation

40 ⟨*semiglobal_affine_align* 40⟩≡

```
template <class T>
void seq_alignment<T>::semiglobal_affine_align(){

    //allocating matrix space
    try{
        M=new boost::multi_array< T, 3>(boost::extents[m+1][n+1][4]);
    }
    catch(const bad_alloc& x){
        cerr <<"Out of memory in semiglobal_affine_align(): "<<x.what() << "\n";
        abort();
    }
    //reinizialize min_m min_n max_m max_n max_score
    min_m=0; max_m=m; min_n=0; max_n=n;
    max_score=0;

    //preparing first row of the 4 matrix
    (*M)[0][0][0]=(T)0;
    (*M)[0][0][1]=NEG_INFTY;
    (*M)[0][0][2]=NEG_INFTY;
    (*M)[0][0][3]=(T)3;

    for (int j=0;j<n;j++){
        int next_j=j+1;
        (*M)[0][next_j][0]=(T)0;
        (*M)[0][next_j][1]=NEG_INFTY;
        (*M)[0][next_j][2]=-(gap_cost+j*extend_gap_cost);
        (*M)[0][next_j][3]=(T)3;//STOP HERE
    }

    for (int i=0;i<m;i++){
        int next_i=i+1;
        //preparing first column of the 4 matrix
        (*M)[next_i][0][0]=(T)0;
        (*M)[next_i][0][1]=-(gap_cost+i*extend_gap_cost);
        (*M)[next_i][0][2]=NEG_INFTY;
        (*M)[next_i][0][3]=(T)3;//STOP HERE

        for (int j=0;j<n;j++){
            int next_j=j+1;
            ⟨affine elaboration 41⟩
        }
    }

    for (int i=1;i<=m;i++)
        if((*M)[i][n][0]>max_score){
            max_score=(*M)[i][n][0];
            max_m=i;
            max_n=n;
```

```
        }
    for (int j=1;j<=n;j++)
        if((*M)[m][j][0]>max_score){
            max_score=(*M)[m][j][0];
            max_m=m;
            max_n=j;
        }

    //cleaning an eventual precalculated alignment
    (*line1).clear();
    (*line2).clear();
    (*line3).clear();
    if (max_score>0)
        build_affine_alignment(max_m, max_n);
    else
        max_score=0;
```

⟨*memory_line* 82⟩

```
    delete M;
}//end semiglobal_affine_align()
```

This code is used in chunk 30.

Matrix elaboration

41 ⟨*affine elaboration* 41⟩≡

```
(*M)[next_i][next_j][0]=(*M)[i][j][0]+c_matrix->get_score((*A)[i],(*B)[j]);
(*M)[next_i][next_j][1]=max((*M)[i][next_j][0]-gap_cost,
                            (*M)[i][next_j][1]-extend_gap_cost);
(*M)[next_i][next_j][2]=max((*M)[next_i][j][0]-gap_cost,
                            (*M)[next_i][j][2]-extend_gap_cost);

(*M)[next_i][next_j][3]=(T)0;//upleft

if((*M)[next_i][next_j][1]>=(*M)[next_i][next_j][0] &&
   (*M)[next_i][next_j][1]>=(*M)[next_i][next_j][2]){
    (*M)[next_i][next_j][0]=(*M)[next_i][next_j][1];
    (*M)[next_i][next_j][3]=(T)1;//up
} else if((*M)[next_i][next_j][2]>=(*M)[next_i][next_j][0] &&
          (*M)[next_i][next_j][2]>(*M)[next_i][next_j][1]){
    (*M)[next_i][next_j][0]=(*M)[next_i][next_j][2];
    (*M)[next_i][next_j][3]=(T)2;//left
}
```

This code is used in chunks 38–40.

Tracking back the alignment

42 ⟨*build_affine_alignment* 42⟩≡

```
template <class T>
void seq_alignment<T>::build_affine_alignment(int i, int j){

    if (i>0 && j>0 && (*M)[i][j][3]==(T)0){
        build_affine_alignment(i-1, j-1);
        line1->push_back((*A)[i-1]);
        if((*A)[i-1]==(*B)[j-1])//matching
            line2->push_back('|');
        else
            line2->push_back('X');//mismatch
        line3->push_back((*B)[j-1]);

        return;
    }
    if(i>0 && (*M)[i][j][3]==(T)1){//it's the beginning of a gap in A
        int k=i;
        while (i>0 && (*M)[i-1][j][1]==(*M)[i][j][1]+extend_gap_cost){
            i--;//measuring the gap
        }

        if (i>0)
            build_affine_alignment(i-1, j);

        while (i<=k){
            line1->push_back((*A)[i-1]);
            line2->push_back(' ');
            line3->push_back('-');
            i++;
        }

        return;
    }
    if(j>0 && (*M)[i][j][3]==(T)2){//it's the beginning of a gap in B
        int k=j;
        while (j>0 && (*M)[i][j-1][2]==(*M)[i][j][2]+extend_gap_cost){
            j--;//measuring the gap
        }

        if (j>0)
            build_affine_alignment(i, j-1);

        while (j<=k){
            line1->push_back('-');
            line2->push_back(' ');
            line3->push_back((*B)[j-1]);
            j++;
        }
```

```
            return;
        }
        if((*M)[i][j][3]==(T)3){
            min_m=i;
            min_n=j;
        }
    }//end build_affine_alignment(int, int)
```

This code is used in chunk 30.

3.5.5 Aligning in linear space

Building the alignment using linear space

43 ⟨*build_linear_alignment* 43⟩≡

```
template <class T>
void seq_alignment<T>::build_linear_alignment(int i1, int j1, int i2, int j2,
                                              T A_begin_gap_cost, T A_end_gap_cost,
                                              T B_begin_gap_cost, T B_end_gap_cost){
    assert(i2 >= i1 && j2 >= j1);

    //best score of the subproblem
    T best_score=NEG_INFTY;

    //solving the base of the recursion
    if  (j1==j2){//only gap in A
        for (int i=i1;i<i2;i++){//gap in the second sequence
            line1->push_back((*A)[i]);
            line2->push_back(' ');
            line3->push_back('-');
        }

        if((i1==0) && (j1==0) && (i2==m) && (j2==n))
            max_score=-(gap_cost+(m-1)*extend_gap_cost);

        return;
    }
    if  (i1==i2){//only gap in B
        for (int j=j1;j<j2;j++){//gap in the second sequence
            line1->push_back('-');
            line2->push_back(' ');
            line3->push_back((*B)[j]);
        }

        if((i1==0) && (j1==0) && (i2==m) && (j2==n))
            max_score=-(gap_cost+(n-1)*extend_gap_cost);

        return;
    }

    int mid=(i1+i2)/2;//mid line

//computing maximum path scores form (i1,j1) to (mid,*)
    //preparing first line of the 4 matrix (gap in B)
    (*M)[i1%2][j1][0]=(T)0;
    (*M)[i1%2][j1][1]=NEG_INFTY;
    (*M)[i1%2][j1][2]=NEG_INFTY;
    (*M)[i1%2][j1][3]=(T)0;

    for (int j=j1;j<j2;j++){
        int next_j=j+1;
```

```
        (*M)[i1%2][next_j][0]=-(A_begin_gap_cost+(j-j1)*extend_gap_cost);
        (*M)[i1%2][next_j][1]=NEG_INFTY;
        (*M)[i1%2][next_j][2]=-(A_begin_gap_cost+(j-j1)*extend_gap_cost);
        (*M)[i1%2][next_j][3]=(T)2;//left
    }

    for (int i=i1; i<=mid; i++){

        //precalc i+1 assuring to use only 2 lines

        int next_i=(i+1)%2;
        //calculating the first column (gap in B)
        (*M)[next_i][j1][0]=-(B_begin_gap_cost+(i-i1)*extend_gap_cost);
        (*M)[next_i][j1][1]=-(B_begin_gap_cost+(i-i1)*extend_gap_cost);
        (*M)[next_i][j1][2]=NEG_INFTY;
        (*M)[next_i][j1][3]=(T)1;//up

        for (int j=j1;j<j2;j++){
            int next_j=j+1;
            (*M)[next_i][next_j][0]=(*M)[i%2][j][0]+
                                c_matrix->get_score((*A)[i],(*B)[j]);
            (*M)[next_i][next_j][1]=max((*M)[i%2][next_j][0]-gap_cost,
                                (*M)[i%2][next_j][1]-extend_gap_cost);
            (*M)[next_i][next_j][2]=max((*M)[next_i][j][0]-gap_cost,
                                (*M)[next_i][j][2]-extend_gap_cost);

            (*M)[next_i][next_j][3]=(T)0;//upleft

            if((*M)[next_i][next_j][1]>=(*M)[next_i][next_j][0] &&
               (*M)[next_i][next_j][1]>=(*M)[next_i][next_j][2]){
                (*M)[next_i][next_j][0]=(*M)[next_i][next_j][1];
                (*M)[next_i][next_j][3]=(T)1;//up
            } else if((*M)[next_i][next_j][2]>=(*M)[next_i][next_j][0] &&
                      (*M)[next_i][next_j][2]>(*M)[next_i][next_j][1]){
                (*M)[next_i][next_j][0]=(*M)[next_i][next_j][2];
                (*M)[next_i][next_j][3]=(T)2;//left
            }
        }
    }

//computing maximum path scores from (i2,j2) down to (mid,*)
    //preparing first line of the 4 matrix (gap in B)
    (*M1)[i2%2][j2][0]=(T)0;
    (*M1)[i2%2][j2][1]=NEG_INFTY;
    (*M1)[i2%2][j2][2]=NEG_INFTY;
    (*M1)[i2%2][j2][3]=(T)0;//STOP HERE

    for (int j=j2;j>j1;j--){
        int prev_j=j-1;
        (*M1)[i2%2][prev_j][0]=-(A_end_gap_cost+(j2-j)*extend_gap_cost);
        (*M1)[i2%2][prev_j][1]=NEG_INFTY;
```

```
    (*M1)[i2%2][prev_j][2]=-(A_end_gap_cost+(j2-j)*extend_gap_cost);
    (*M1)[i2%2][prev_j][3]=(T)2;//right
}

for (int i=i2; i>mid; i--){
    //precalc i-1 assuring to use only 2 lines
    int prev_i=(i-1)%2;

    //calculating the first column (gap in B)
    (*M1)[prev_i][j2][0]=-(B_end_gap_cost+(i2-i)*extend_gap_cost);
    (*M1)[prev_i][j2][1]=-(B_end_gap_cost+(i2-i)*extend_gap_cost);
    (*M1)[prev_i][j2][2]=NEG_INFTY;
    (*M1)[prev_i][j2][3]=(T)1;//down

    for (int j=j2;j>j1;j--){
        int prev_j=j-1;
        (*M1)[prev_i][prev_j][0]=(*M1)[i%2][j][0]+
                            c_matrix->get_score((*A)[i-1],(*B)[prev_j]);
        (*M1)[prev_i][prev_j][1]=max((*M1)[i%2][prev_j][0]-gap_cost,
                                (*M1)[i%2][prev_j][1]-extend_gap_cost);
        (*M1)[prev_i][prev_j][2]=max((*M1)[prev_i][j][0]-gap_cost,
                                  (*M1)[prev_i][j][2]-extend_gap_cost);

        (*M1)[prev_i][prev_j][3]=(T)0;//downright

        if((*M1)[prev_i][prev_j][1]>=(*M1)[prev_i][prev_j][0] &&
           (*M1)[prev_i][prev_j][1]>=(*M1)[prev_i][prev_j][2]){
            (*M1)[prev_i][prev_j][0]=(*M1)[prev_i][prev_j][1];
            (*M1)[prev_i][prev_j][3]=(T)1;//down
        }else if ((*M1)[prev_i][prev_j][2]>=(*M1)[prev_i][prev_j][0] &&
                  (*M1)[prev_i][prev_j][2]>(*M1)[prev_i][prev_j][1]){
            (*M1)[prev_i][prev_j][0]=(*M1)[prev_i][prev_j][2];
            (*M1)[prev_i][prev_j][3]=(T)2;//right
        }
    }
}

//finding a midpoint of the global alignment

//position of the best score
int best_score_pos=j1;

for (int j=j1;j<=j2;j++){
    T sum=((*M)[mid%2][j][0]+(*M1)[mid%2][j][0]);

    //if 2 gaps are in the same direction we must reduce the penalty
    sum=max(sum,
            (*M)[mid%2][j][1]+(*M1)[mid%2][j][1]+gap_cost-extend_gap_cost);
    sum=max(sum,
            (*M)[mid%2][j][2]+(*M1)[mid%2][j][2]+gap_cost-extend_gap_cost);
    //find the best j
```

```
        if(sum>best_score){
            best_score_pos=j;
            best_score=sum;
        }
    }

    //in the first recursion we archive the best global score in max_score
    if((i1==0) && (j1==0) && (i2==m) && (j2==n) && (best_score>max_score))
        max_score=best_score;

    //only one char left on seq A
    if(i2-i1==1){
        if ((*M)[i2%2][j2][3]==(T)0){
            build_linear_alignment(i1, j1, i2-1, j2-1,
                                   A_begin_gap_cost,A_end_gap_cost,
                                   B_begin_gap_cost,B_end_gap_cost);

            line1->push_back((*A)[i2-1]);
            if((*A)[i2-1]==(*B)[j2-1])//matching
               line2->push_back('|');
            else
                line2->push_back('X');//mismatch
            line3->push_back((*B)[j2-1]);
            return;
        }
        if((*M)[i2%2][j2][3]==(T)1){//print the char in A
                build_linear_alignment(i1, j1, i2-1, j2,
                                   A_begin_gap_cost,A_end_gap_cost,
                                   B_begin_gap_cost,B_end_gap_cost);

                line1->push_back((*A)[i2-1]);
                line2->push_back(' ');
                line3->push_back('-');
            return;
        }
        if((*M)[i2%2][j2][3]==(T)2){//it's the beginning of a gap in A
            int k=j2;
            while(j2>j1 && (*M)[i2%2][j2-1][2]==(*M)[i2%2][j2][2]
                                                        +extend_gap_cost){
                j2--;//measuring the gap
            }

            if (j2>j1)
                build_linear_alignment(i1, j1, i2, j2-1,
                                   A_begin_gap_cost,A_end_gap_cost,
                                   B_begin_gap_cost,B_end_gap_cost);

            while (j2<=k){
                line1->push_back('-');
                line2->push_back(' ');
                line3->push_back((*B)[j2-1]);
```

```
                j2++;
            }

            return;
        }

        if(m==1)//A had only 1 char
            max_score=(*M)[i2%2][j2][0];

        return;
    }

//operate on the 2 halves
    //if the pivot is in the middle of a long gap we divide in 3 parts
    //long gap in B
    if (best_score==(*M)[mid%2][best_score_pos][1]+
                    (*M1)[mid%2][best_score_pos][1]+
                    gap_cost-extend_gap_cost){

        build_linear_alignment(i1,j1,mid-1,best_score_pos,
                               A_begin_gap_cost,A_end_gap_cost,
                               B_begin_gap_cost,extend_gap_cost);

        line1->push_back((*A)[mid-1]);
        line2->push_back(' ');
        line3->push_back('-');
        line1->push_back((*A)[mid]);
        line2->push_back(' ');
        line3->push_back('-');

        build_linear_alignment(mid+1,best_score_pos,i2,j2,
                               A_begin_gap_cost,A_end_gap_cost,
                               extend_gap_cost,B_end_gap_cost);
        return;
    }
    //long gap in A
    if (best_score==(*M)[mid%2][best_score_pos][2]+
                    (*M1)[mid%2][best_score_pos][2]+
                    gap_cost-extend_gap_cost){
        build_linear_alignment(i1,j1,mid,best_score_pos-1,
                               A_begin_gap_cost,extend_gap_cost,
                               B_begin_gap_cost,B_end_gap_cost);

        line1->push_back('-');
        line2->push_back(' ');
        line3->push_back((*B)[best_score_pos-1]);
        line1->push_back('-');
        line2->push_back(' ');
        line3->push_back((*B)[best_score_pos]);

        build_linear_alignment(mid,best_score_pos+1,i2,j2,
```

```
                                extend_gap_cost,A_end_gap_cost,
                                B_begin_gap_cost,B_end_gap_cost);
        return;
    }

    build_linear_alignment(i1,j1,mid,best_score_pos,
                           A_begin_gap_cost,gap_cost,B_begin_gap_cost,gap_cost);
    build_linear_alignment(mid,best_score_pos,i2,j2,
                           gap_cost,A_end_gap_cost,gap_cost,B_end_gap_cost);
}//end build_linear_alignment()
```

This code is used in chunk 30.

Global alignment

44 ⟨*nw_linear_align* 44⟩≡

```
template <class T>
void seq_alignment<T>::nw_linear_align(){

    //Allocating needed space
    try{
        M=new boost::multi_array< T, 3>(boost::extents[2][n+1][4]);
        M1=new boost::multi_array< T, 3>(boost::extents[2][n+1][4]);
    }
    catch(const bad_alloc& x){
        cerr << "Out of memory in nw_linear_align(): " << x.what() << "\n";
        abort();
    }
    //reinizialize min_m min_n max_m max_n max_score
    min_m=0; max_m=m; min_n=0; max_n=n;
    max_score=NEG_INFTY;

    //cleaning an eventual precalculated alignment
    (*line1).clear();
    (*line2).clear();
    (*line3).clear();

    if (m>0||n>0)
        build_linear_alignment(0,0,m,n,gap_cost,gap_cost,gap_cost,gap_cost);
    else max_score=0;//both sequences empty
```

⟨*memory_line* 82⟩

```
    delete M;
    delete M1;
}
```

This code is used in chunk 30.

Local alignment

45 ⟨*sw_linear_align* 45⟩≡

```
template <class T>
void seq_alignment<T>::sw_linear_align(){
    //Allocating needed space
    try{
        M=new boost::multi_array< T, 3>(boost::extents[2][n+1][4]);
        M1=new boost::multi_array< T, 3>(boost::extents[2][n+1][4]);
    }
    catch(const bad_alloc& x){
        cerr << "Out of memory in sw_linear_align(): " << x.what() << "\n";
        abort();
    }
    //reinizialize min_m min_n max_m max_n max_score
    min_m=0; max_m=m; min_n=0; max_n=n;
    max_score=0;

//determining max_m max_n with an upward cost only alignment
    //preparing first line of the 4 matrix
    (*M)[0][0][0]=(T)0;
    (*M)[0][0][1]=NEG_INFTY;
    (*M)[0][0][2]=NEG_INFTY;

    for (int j=0;j<n;j++){
        int next_j=j+1;
        (*M)[0][next_j][0]=(T)0;
        (*M)[0][next_j][1]=NEG_INFTY;
        (*M)[0][next_j][2]=-(gap_cost+j*extend_gap_cost);
    }

    for (int i=0; i<m; i++){
        //precalc i+1
        int next_i=(i+1)%2;

        //calculating the first column
        (*M)[next_i][0][0]=(T)0;
        (*M)[next_i][0][1]=-(gap_cost+i*extend_gap_cost);
        (*M)[next_i][0][2]=NEG_INFTY;

        for (int j=0;j<n;j++){
            int next_j=j+1;
            (*M)[next_i][next_j][0]=(*M)[i%2][j][0]+
                                c_matrix->get_score((*A)[i],(*B)[j]);
            (*M)[next_i][next_j][1]=max((*M)[i%2][next_j][0]-gap_cost,
                                (*M)[i%2][next_j][1]-extend_gap_cost);
            (*M)[next_i][next_j][2]=max((*M)[next_i][j][0]-gap_cost,
                                (*M)[next_i][j][2]-extend_gap_cost);

            if((*M)[next_i][next_j][0]>(*M)[next_i][next_j][1] &&
               (*M)[next_i][next_j][0]>(*M)[next_i][next_j][2]){
```

```
            } else if((*M)[next_i][next_j][1]>(*M)[next_i][next_j][2]){
                (*M)[next_i][next_j][0]=(*M)[next_i][next_j][1];
            } else {
                (*M)[next_i][next_j][0]=(*M)[next_i][next_j][2];
            }

            if(0>(*M)[next_i][next_j][0]){
                (*M)[next_i][next_j][0]=(T)0;
            }

            if((*M)[next_i][next_j][0]>max_score){
                max_m=i+1;
                max_n=next_j;
                max_score=(*M)[next_i][next_j][0];
            }
        }
    }

//determining min_m min_n with a backward cost only alignment
    //preparing first line of the 4 matrix
    (*M1)[max_m%2][max_n][0]=(T)0;
    (*M1)[max_m%2][max_n][1]=NEG_INFTY;
    (*M1)[max_m%2][max_n][2]=NEG_INFTY;

    for (int j=max_n;j>0;j--){
        int prev_j=j-1;
        (*M1)[max_m%2][prev_j][0]=(T)0;
        (*M1)[max_m%2][prev_j][1]=NEG_INFTY;
        (*M1)[max_m%2][prev_j][2]=-(gap_cost+(max_n-j)*extend_gap_cost);
    }

    for (int i=max_m; i>0; i--){
        //precalc i+1
        int prev_i=(i-1)%2;

        //calculating the first column
        (*M1)[prev_i][max_n][0]=(T)0;
        (*M1)[prev_i][max_n][1]=-(gap_cost+(max_m-i)*extend_gap_cost);
        (*M1)[prev_i][max_n][2]=NEG_INFTY;

        for (int j=max_n;j>0;j--){
            int prev_j=j-1;
            (*M1)[prev_i][prev_j][0]=(*M1)[i%2][j][0]+
                                c_matrix->get_score((*A)[i-1],(*B)[prev_j]);
            (*M1)[prev_i][prev_j][1]=max((*M1)[i%2][prev_j][0]-gap_cost,
                                    (*M1)[i%2][prev_j][1]-extend_gap_cost);
            (*M1)[prev_i][prev_j][2]=max((*M1)[prev_i][j][0]-gap_cost,
                                    (*M1)[prev_i][j][2]-extend_gap_cost);

            if((*M1)[prev_i][prev_j][0]>(*M1)[prev_i][prev_j][1] &&
               (*M1)[prev_i][prev_j][0]>(*M1)[prev_i][prev_j][2]){
```

```
            } else if((*M1)[prev_i][prev_j][1]>(*M1)[prev_i][prev_j][2]){
                (*M1)[prev_i][prev_j][0]=(*M1)[prev_i][prev_j][1];
            } else {
                (*M1)[prev_i][prev_j][0]=(*M1)[prev_i][prev_j][2];
            }

            if((*M1)[prev_i][prev_j][0]==max_score){
                min_m=i-1;
                min_n=prev_j;
            }
        }
    }

    //cleaning an eventual precalculated alignment
    (*line1).clear();
    (*line2).clear();
    (*line3).clear();
    //if first string empty
    if (max_score>0)
        build_linear_alignment(min_m,min_n,max_m,max_n,
                               gap_cost,gap_cost,gap_cost,gap_cost);
    else
        max_score=(T)0;
```

⟨*memory_line* 82⟩

```
    delete M;
    delete M1;
}//end sw_linear_align_align()
```

This code is used in chunk 30.

Semiglobal alignment

46 ⟨*semiglobal_linear_align* 46⟩≡

```
template <class T>
void seq_alignment<T>::semiglobal_linear_align(){
    //Allocating needed space
    try{
        M=new boost::multi_array< T, 3>(boost::extents[2][n+1][4]);
        M1=new boost::multi_array< T, 3>(boost::extents[2][n+1][4]);
    }
    catch(const bad_alloc& x){
        cerr<< "Out of memory in semiglobal_linear_align(): "<< x.what()<<"\n";
        abort();
    }
    //reinizialize min_m min_n max_m max_n max_score
    min_m=0; max_m=m; min_n=0; max_n=n;
    max_score=0;

//determining max_m max_n with an upward, cost only, alignment
    //preparing first line of the 4 matrix
    (*M)[0][0][0]=(T)0;
    (*M)[0][0][1]=NEG_INFTY;
    (*M)[0][0][2]=NEG_INFTY;

    for (int j=0;j<n;j++){
        int next_j=j+1;
        (*M)[0][next_j][0]=(T)0;
        (*M)[0][next_j][1]=NEG_INFTY;
        (*M)[0][next_j][2]=-(gap_cost+j*extend_gap_cost);
    }

    for (int i=0; i<m; i++){
        //precalc i+1
        int next_i=(i+1)%2;

        //calculating the first column
        (*M)[next_i][0][0]=(T)0;
        (*M)[next_i][0][1]=-(gap_cost+i*extend_gap_cost);
        (*M)[next_i][0][2]=NEG_INFTY;

        for (int j=0;j<n;j++){
            int next_j=j+1;
            (*M)[next_i][next_j][0]=(*M)[i%2][j][0]+
                                c_matrix->get_score((*A)[i],(*B)[j]);
            (*M)[next_i][next_j][1]=max((*M)[i%2][next_j][0]-gap_cost,
                                (*M)[i%2][next_j][1]-extend_gap_cost);
            (*M)[next_i][next_j][2]=max((*M)[next_i][j][0]-gap_cost,
                                (*M)[next_i][j][2]-extend_gap_cost);

            if((*M)[next_i][next_j][1]>=(*M)[next_i][next_j][0] &&
               (*M)[next_i][next_j][1]>=(*M)[next_i][next_j][2]){
```

```
                (*M)[next_i][next_j][0]=(*M)[next_i][next_j][1];
            } else if((*M)[next_i][next_j][2]>=(*M)[next_i][next_j][0] &&
                      (*M)[next_i][next_j][2]>(*M)[next_i][next_j][1]){
                (*M)[next_i][next_j][0]=(*M)[next_i][next_j][2];
            }

            if((i+1==m || next_j==n) && (*M)[next_i][next_j][0]>max_score){
                max_m=i+1;
                max_n=next_j;
                max_score=(*M)[next_i][next_j][0];
            }
        }
    }

//determining min_m min_n with a backward, cost only, alignment
    //preparing first line of the 4 matrix
    (*M1)[max_m%2][max_n][0]=(T)0;
    (*M1)[max_m%2][max_n][1]=NEG_INFTY;
    (*M1)[max_m%2][max_n][2]=NEG_INFTY;

    for (int j=max_n;j>0;j--){
        int prev_j=j-1;
        (*M1)[max_m%2][prev_j][0]=-(gap_cost+(max_n-j)*extend_gap_cost);
        (*M1)[max_m%2][prev_j][1]=NEG_INFTY;
        (*M1)[max_m%2][prev_j][2]=-(gap_cost+(max_n-j)*extend_gap_cost);
    }

    for (int i=max_m; i>0; i--){
        //precalc i+1
        int prev_i=(i-1)%2;

        //calculating the first column
        (*M1)[prev_i][max_n][0]=-(gap_cost+(max_m-i)*extend_gap_cost);
        (*M1)[prev_i][max_n][1]=-(gap_cost+(max_m-i)*extend_gap_cost);
        (*M1)[prev_i][max_n][2]=NEG_INFTY;

        for (int j=max_n;j>0;j--){
            int prev_j=j-1;
            (*M1)[prev_i][prev_j][0]=(*M1)[i%2][j][0]+
                                  c_matrix->get_score((*A)[i-1],(*B)[prev_j]);
            (*M1)[prev_i][prev_j][1]=max((*M1)[i%2][prev_j][0]-gap_cost,
                                         (*M1)[i%2][prev_j][1]-extend_gap_cost);
            (*M1)[prev_i][prev_j][2]=max((*M1)[prev_i][j][0]-gap_cost,
                                         (*M1)[prev_i][j][2]-extend_gap_cost);

            if((*M1)[prev_i][prev_j][1]>=(*M1)[prev_i][prev_j][0] &&
               (*M1)[prev_i][prev_j][1]>=(*M1)[prev_i][prev_j][2] ){
                (*M1)[prev_i][prev_j][0]=(*M1)[prev_i][prev_j][1];
            } else if((*M1)[prev_i][prev_j][2]>=(*M1)[prev_i][prev_j][0] &&
                      (*M1)[prev_i][prev_j][2]>(*M1)[prev_i][prev_j][1]){
                (*M1)[prev_i][prev_j][0]=(*M1)[prev_i][prev_j][2];
```

```
            }

            if((i-1==0 || prev_j==0 ) &&
               ((*M1)[prev_i][prev_j][0]==max_score)){
                min_m=i-1;
                min_n=prev_j;
                break; //the first found is OK
            }
        }
    }

    //cleaning an eventual precalculated alignment
    (*line1).clear();
    (*line2).clear();
    (*line3).clear();

     if (max_score>0)
        build_linear_alignment(min_m,min_n,max_m,max_n,
                               gap_cost,gap_cost,gap_cost,gap_cost);
    else
        max_score=(T)0;
```

⟨*memory_line* 82⟩

```
    delete M;
    delete M1;
}//end semiglobal_linear_align()
```

This code is used in chunk 30.

Chapter 4

Tests

4.1 Regression tests

In order to test the current and the possible future versions of the produced libraries, we adopt the technique of "automate regression testing", which performs a sequence of tests to compare the expected behaviour of the classes with their real behaviour or to compare the new versions of our classes implementation with the old ones. This is done by comparing the output of some appropriate test programs executed on the latest versions of the classes, against the expected output or against the output of the same test programs executed on the old implementation of the classes.

The test requires the following tools:

- A test program (one for each class), which contains a main function implementing tests for all the features of the tested class;
- A behaviour file (one for each class), which contains the tests inputs and the corresponding expected outputs. The inputs provided are supposed to cover all the possible situations (even the unexpected ones) that can rise using the class;
- An output file (one for each class), which contains the actual outputs produced by the test program executed on the latest version of the class;
- A script program that executes the test programs, performs the comparison between the expected and the actual outputs, and reports the possible differences;

Then, the programmer must make sure that the behaviour of the classes hasn't changed, except in expected ways.

4.1.1 Structure of the test program

Description of the main function

The test program contains a main function that implements all the suitable tests. Each test is identified by a number code (starting from 1), and can be executed independently from others.

The main function reads a code as parameter and selects the corresponding test. Then it reads the requested test inputs from standard input, executes the test and prints the results on standard output. If incorrect parameters or input values are found, the program reports an error and, if it's impossible to continue, ends the test and returns the 0 value.

Description of the variables

The `main` function uses two variables:

- `line`, of `string` data type, used to store each line read from the standard input.
- `code`, of `int` type, used to store the codes of the tests to execute.

47 ⟨*main function for test: head* 47⟩≡

```
#define SUCCESS 0
#define CODE_ERROR 1
#define INPUT_ERROR 2
#define SYNTAX_ERROR 3

int main (int argc, char *argv[]){
    int code;

    //check the correctness of the parameters

    if(argc<2) {
        std::cout << "Usage: " << argv[0] << " TEST_CODE\n";
        return SYNTAX_ERROR;
    }
    code=atoi(argv[1]);
    if(code==0) {
        std::cout << "Error while testing: incorrect code.\n";
        return CODE_ERROR;
    }
```

This code is used in chunks 49 and 61.

48 ⟨*main function for test: tail* 48⟩≡

```
    return SUCCESS;
}
```

This code is used in chunks 49 and 61.

4.1.2 Structure of the test file

The test file must respect the following syntax:

- the first line of each test must begin with the special sequence "====="; everything after that is ignored and can, therefore, be viewed as a comment;
- after that, a test identifier (a numerical code) is expected;
- the third line must begin with the "***** sequence"; everything after that is ignored and can, therefore, be viewed as a comment;
- in the following lines, input data must be specified (each input on a different line);
- after all the input data have been entered, the program expects a line beginning with the "***** sequence"; everything after that is ignored and can, therefore, be viewed as a comment;
- in the following lines, the expected outputs must be specified;
- after all the expected outputs have been entered, either another test or the end of the file is expected.

Test file example

```
=====set_score(char, char, T) - 2 chars 1 T=====
3
*****
G
C
5
*****
5
=====set_match_score(T) - 1 T=====
4
*****
3
*****
      A     C     G     T
A     3     -1    -1    -1
C     -1    3     -1    -1
G     -1    -1    3     -1
T     -1    -1    -1    3
gap cost=1
gap extension cost=1
```

4.1.3 Test program for `score_matrix` class

This program implements all the suitable tests for the ⟨*score_matrix* 8⟩ class.

49 ⟨*score_matrix.cpp* 49⟩≡

```
⟨licence 2⟩
#include <iostream>
#include <stdlib.h>
#include <alibio/two_sequence.hpp>

⟨main function for test: head 47⟩
switch(code) {
    case 1:
        ⟨testing score_matrix(alibio::alphabet) constructor 51⟩
        break;
    case 2:
        ⟨testing set_score(char, char, T) function 52⟩
        break;
    case 3:
        ⟨testing get_score(char, char) function 53⟩
        break;
    case 4:
        ⟨testing set_match_score(T) function 54⟩
        break;
    case 5:
```

```
        ⟨testing set_mismatch_score(T) function 55⟩
        break;
    case 6:
        ⟨testing set_gap_cost(T) function 56⟩
        break;
    case 7:
        ⟨testing set_gap_cost(T,T) function 57⟩
        break;
    case 8:
        ⟨testing get_gap_cost() function 58⟩
        break;
    case 9:
        ⟨testing get_extend_gap_cost() function 59⟩
        break;
    case 10:
        ⟨testing print() function 60⟩
        break;
    default:
        std::cout << "Error while testing: code '" << code << "' not found";
}
⟨main function for test: tail 48⟩
```

Root chunk (not used in this document).

Preparing `score_matrix` object

50 ⟨*preparing score_matrix object* 50⟩≡

```
//declaring an alphabet for DNA
alibio::alphabet dna("DNA");

//adding symbols to the alphabet
dna.add_symbol(alibio::symbol('A',"Adenine"));
dna.add_symbol(alibio::symbol('C',"Cytosine"));
dna.add_symbol(alibio::symbol('T',"Thymine"));
dna.add_symbol(alibio::symbol('G',"Guanine"));

//creating the score matrix
alibio::score_matrix<float> myscores(dna);
```

This code is used in chunks 51–60 and 62–65.

Testing `score_matrix` constructor

The test creates an `alibio::alphabet` object and uses it for instantiating a ⟨*score_matrix* 8⟩ object

51 ⟨*testing score_matrix(alibio::alphabet) constructor* 51⟩≡

```
{
    ⟨preparing score_matrix object 50⟩
}
```

This code is used in chunk 49.

Testing `set_score` function

The test reads from the input the size of the distance matrix that will be created using the generic constructor `score_matrix(int dim)` and prints it. Then reads a second size and resize the object using `resize(int dim)`. Finally prints the resized object on the standard output using `print()` function.

52 ⟨*testing set_score(char, char, T) function* 52⟩≡

```
{
    ⟨preparing score_matrix object 50⟩

    //reading 2 chars from stdin
    std::string line1;
    std::string line2;

    //reading 1 float from stdin
    std::string line3;

    std::getline(std::cin,line1,'\n');
    std::getline(std::cin,line2,'\n');
    std::getline(std::cin,line3,'\n');
    myscores.set_score(line1[0], line2[0], atof(line3.c_str()));
    cout << myscores.get_score(line1[0], line2[0]);
}
```

This code is used in chunk 49.

Testing `get_score` function

The test creates an `alibio::alphabet` and a ⟨*score_matrix* 8⟩ object, it reads from the standard input two characters (one per line) and prints the score of switching between the first to the second getting the value from the ⟨*score_matrix* 8⟩ object. If one or booth of the chars are not listed in the alphabet, the test prints an error message.

53 ⟨*testing get_score(char, char) function* 53⟩≡

```
{
    ⟨preparing score_matrix object 50⟩

    //reading 2 chars from stdin
    std::string line1;
    std::string line2;
    float score;

    std::getline(std::cin,line1,'\n');
    std::getline(std::cin,line2,'\n');
    score=myscores.get_score(line1[0], line2[0]);
    cout << score;
}
```

This code is used in chunk 49.

Testing `set_match_score` function

The test creates a ⟨*score_matrix* 8⟩ object

54 ⟨*testing set_match_score(T) function* 54⟩≡

```
{
    ⟨preparing score_matrix object 50⟩

    //reading 1 double from stdin
    std::string line1;

    std::getline(std::cin,line1,'\n');
    myscores.set_match_score(atof(line1.c_str()));
    myscores.print();
}
```

This code is used in chunk 49.

Testing `set_mismatch_score` function

The test creates a ⟨*score_matrix* 8⟩ object

55 ⟨*testing set_mismatch_score(T) function* 55⟩≡

```
{
    ⟨preparing score_matrix object 50⟩

    //reading 1 double from stdin
    std::string line1;

    std::getline(std::cin,line1,'\n');
    myscores.set_mismatch_score(atof(line1.c_str()));
    myscores.print();
}
```

This code is used in chunk 49.

Testing `set_gap_cost` function

The test creates a ⟨*score_matrix* 8⟩ object.

56 ⟨*testing set_gap_cost(T) function* 56⟩≡

```
{
    ⟨preparing score_matrix object 50⟩

    //reading 1 double from stdin
    std::string line1;

    std::getline(std::cin,line1,'\n');
    myscores.set_gap_cost(atof(line1.c_str()));
    cout << myscores.get_gap_cost() << endl;
    cout << myscores.get_extend_gap_cost();
}
```

This code is used in chunk 49.

57 ⟨*testing set_gap_cost(T,T) function* 57⟩≡

```
{
    ⟨preparing score_matrix object 50⟩

    //reading 2 double from stdin
    std::string line1, line2;

    std::getline(std::cin,line1,'\n');
    std::getline(std::cin,line2,'\n');
    myscores.set_gap_cost(atof(line1.c_str()),atof(line2.c_str()));
    cout << myscores.get_gap_cost() << endl;
    cout << myscores.get_extend_gap_cost();
}
```

This code is used in chunk 49.

Testing `get_gap_cost` function

The test creates a ⟨*score_matrix* 8⟩ object.

58 ⟨*testing get_gap_cost() function* 58⟩≡

```
{
    ⟨preparing score_matrix object 50⟩

    cout << myscores.get_gap_cost();
}
```

This code is used in chunk 49.

Testing `get_extend_gap_cost` function

The test creates a ⟨*score_matrix* 8⟩ object.

59 ⟨*testing get_extend_gap_cost() function* 59⟩≡

```
{
    ⟨preparing score_matrix object 50⟩

    cout << myscores.get_extend_gap_cost();
}
```

This code is used in chunk 49.

Testing `print` function

The test creates a ⟨*score_matrix* 8⟩ object.

60 ⟨*testing print() function* 60⟩≡

```
{
    ⟨preparing score_matrix object 50⟩

    myscores.print();
}
```

This code is used in chunk 49.

4.1.4 Test program for `seq_alignment` class

This program implements the tests for the ⟨*seq_alignment* 25⟩ class.

61 ⟨*seq_alignment.cpp* 61⟩≡

⟨*licence* 2⟩

```
#include <iostream>
#include <stdlib.h>
#include <alibio/two_sequence.hpp>

⟨main function for test: head 47⟩
switch(code) {
    case 1:
        ⟨testing seq_alignment vector constructor 62⟩
    case 2:
        ⟨testing seq_alignment string constructor 63⟩
        break;
    case 3:
        ⟨testing seq_alignment bio_string constructor 64⟩
        break;
    case 4:
        ⟨testing nw_align() function 66⟩
        break;
    case 5:
        ⟨testing sw_align() function 67⟩
        break;
    case 6:
        ⟨testing nw_affine_align() function 68⟩
        break;
    case 7:
        ⟨testing sw_affine_align() function 69⟩
        break;
    case 8:
        ⟨testing semiglobal_affine_align() function 70⟩
        break;
    case 9:
        ⟨testing nw_linear_align() function 71⟩
        break;
    case 10:
        ⟨testing sw_linear_align() function 72⟩
        break;
    case 11:
        ⟨testing semiglobal_linear_align() function 73⟩
        break;
    case 12:
        ⟨testing nw_affine_align() with a different alphabet 74⟩
        break;
    default:
        std::cout << "\nError while testing: code '" << code << "' not found";
}
⟨main function for test: tail 48⟩
```

Root chunk (not used in this document).

Testing `seq_alignment` constructors

This test verifies the ⟨*seq_alignment* 25⟩ vector constructor using the DNA alphabet and the default score matrix

62 ⟨*testing seq_alignment vector constructor* 62⟩≡

```
{
    ⟨preparing score_matrix object 50⟩

    //reading 2 strings from stdin
    std::string line1;
    std::string line2;

    std::getline(std::cin,line1,'\n');
    std::getline(std::cin,line2,'\n');

    vector<char> A(line1.begin(),line2.end());
    vector<char> B(line1.begin(),line2.end());
    //creating the alignment matrix
    alibio::seq_alignment<float> my_alignment(myscores, A, B);
}
```

This code is used in chunk 61.

This test verifies the ⟨*seq_alignment* 25⟩ string constructor using the DNA alphabet and the default score matrix

63 ⟨*testing seq_alignment string constructor* 63⟩≡

```
{
    ⟨preparing score_matrix object 50⟩

    //reading 2 strings from stdin
    std::string line1;
    std::string line2;

    std::getline(std::cin,line1,'\n');
    std::getline(std::cin,line2,'\n');

    //creating the alignment matrix
    alibio::seq_alignment<float> my_alignment(myscores, line1, line2);
}
```

This code is used in chunk 61.

This test verifies the ⟨*seq_alignment* 25⟩ `bio_string` constructor using the DNA alphabet and the default score matrix

64 ⟨*testing seq_alignment bio_string constructor* 64⟩≡

```
{
    ⟨preparing score_matrix object 50⟩

    //declaring 2 sequences with "bio_string" class as primary structure and
    //"empty" class as secondary and tertiary structures
    alibio::sequence<alibio::bio_string,alibio::empty,alibio::empty> seq_one;
    alibio::sequence<alibio::bio_string,alibio::empty,alibio::empty> seq_two;

    //setting the alphabet (working on "seq_one.primary")
    seq_one.primary.set_alphabet(dna);
    seq_two.primary.set_alphabet(dna);

    //reading 2 strings from stdin
    std::string line1;
    std::string line2;
    std::getline(std::cin,line1,'\n');
    std::getline(std::cin,line2,'\n');

    //setting the sequence (working on "seq_one.primary")
    seq_one.primary.set_sequence(line1);

    //setting the sequence (working on "seq_two.primary")
    seq_two.primary.set_sequence(line2);

    //creating the alignment matrix
    alibio::seq_alignment<float> my_alignment(myscores,
                                              seq_one.primary, seq_two.primary);
}
```

This code is used in chunk 61.

Preparing `seq_alignment` object

In this section we prepare a ⟨*seq_alignment* 25⟩ object that will be used for all the alignment tests

65 ⟨*preparing seq_alignment object* 65⟩≡

```
⟨preparing score_matrix object 50⟩

//modifying score matrix
myscores.set_match_score(5);
myscores.set_mismatch_score(-4);
myscores.set_gap_cost(10, 0.5);

//reading 2 strings from stdin
std::string line1;
std::string line2;
std::getline(std::cin,line1,'\n');
std::getline(std::cin,line2,'\n');

//creating the alignment object
alibio::seq_alignment<float> my_alignment(myscores, line1, line2);
```

This code is used in chunks 66–73.

Testing `nw_align` function

66 ⟨*testing nw_align() function* 66⟩≡

```
{
    ⟨preparing seq_alignment object 65⟩
    my_alignment.nw_align();
    my_alignment.print_alignment();
}
```

This code is used in chunk 61.

Testing `sw_align` function

67 ⟨*testing sw_align() function* 67⟩≡

```
{
    ⟨preparing seq_alignment object 65⟩
    my_alignment.sw_align();
    my_alignment.print_alignment();
}
```

This code is used in chunk 61.

Testing `nw_affine_align` function

68 ⟨*testing nw_affine_align() function* 68⟩≡

```
{
    ⟨preparing seq_alignment object 65⟩
    my_alignment.nw_affine_align();
    my_alignment.print_alignment();
}
```

This code is used in chunk 61.

Testing `sw_affine_align` function

69 ⟨*testing sw_affine_align() function* 69⟩≡

```
{
    ⟨preparing seq_alignment object 65⟩
    my_alignment.sw_affine_align();
    my_alignment.print_alignment();
}
```

This code is used in chunk 61.

Testing `semiglobal_affine_align` function

70 ⟨*testing semiglobal_affine_align() function* 70⟩≡

```
{
    ⟨preparing seq_alignment object 65⟩
    my_alignment.semiglobal_affine_align();
    my_alignment.print_alignment();
}
```

This code is used in chunk 61.

Testing `nw_linear_align` function

71 ⟨*testing nw_linear_align() function* 71⟩≡

```
{
    ⟨preparing seq_alignment object 65⟩
    my_alignment.nw_linear_align();
    my_alignment.print_alignment();
}
```

This code is used in chunk 61.

Testing `sw_linear_align` function

72 ⟨*testing sw_linear_align() function* 72⟩≡

```
{
    ⟨preparing seq_alignment object 65⟩
    my_alignment.sw_linear_align();
    my_alignment.print_alignment();
}
```

This code is used in chunk 61.

Testing `semiglobal_linear_align` function

73 ⟨*testing semiglobal_linear_align() function* 73⟩≡

```
{
    ⟨preparing seq_alignment object 65⟩
    my_alignment.semiglobal_linear_align();
    my_alignment.print_alignment();
}
```

This code is used in chunk 61.

Testing nw_affine_align with a different alphabet

This test runs the nw_affine_align using a different alphabet

74 ⟨*testing nw_affine_align() with a different alphabet* 74⟩≡

```
{
    //declaring an alphabet for DNA
    alibio::alphabet prot("Protein");

    //adding symbols to the alphabet
    prot.add_symbol(alibio::symbol('A'));
    prot.add_symbol(alibio::symbol('R'));
    prot.add_symbol(alibio::symbol('N'));
    prot.add_symbol(alibio::symbol('D'));
    prot.add_symbol(alibio::symbol('C'));
    prot.add_symbol(alibio::symbol('Q'));
    prot.add_symbol(alibio::symbol('E'));
    prot.add_symbol(alibio::symbol('G'));
    prot.add_symbol(alibio::symbol('H'));
    prot.add_symbol(alibio::symbol('I'));
    prot.add_symbol(alibio::symbol('L'));
    prot.add_symbol(alibio::symbol('K'));
    prot.add_symbol(alibio::symbol('M'));
    prot.add_symbol(alibio::symbol('F'));
    prot.add_symbol(alibio::symbol('P'));
    prot.add_symbol(alibio::symbol('S'));
    prot.add_symbol(alibio::symbol('T'));
    prot.add_symbol(alibio::symbol('W'));
    prot.add_symbol(alibio::symbol('Y'));
    prot.add_symbol(alibio::symbol('V'));
    prot.add_symbol(alibio::symbol('B'));
    prot.add_symbol(alibio::symbol('Z'));
    prot.add_symbol(alibio::symbol('X'));

    //creating the score matrix for "prot" alphabet
    alibio::score_matrix<float> prot_scores(prot);

    //reading 2 strings from stdin
    std::string line1;
    std::string line2;
    //reading match score from stdin
    std::string line3;
    //reading gap cost from stdin
    std::string line4;
    //reading extending gap cost from stdin
    std::string line5;
    //reading mismatch score from stdin
    std::string line6;

    std::getline(std::cin,line1,'\n');
    std::getline(std::cin,line2,'\n');
```

```
    std::getline(std::cin,line3,'\n');
    std::getline(std::cin,line4,'\n');
    std::getline(std::cin,line5,'\n');
    std::getline(std::cin,line6,'\n');

    //modifying score matrix
    prot_scores.set_match_score(atof(line3.c_str()));
    prot_scores.set_gap_cost(atof(line4.c_str()),atof(line5.c_str()));
    prot_scores.set_mismatch_score(atof(line6.c_str()));

    //creating the alignment matrix
    alibio::seq_alignment<float> prot_alignment(prot_scores, line1, line2);

    //printing alignment
    prot_alignment.nw_affine_align();
    prot_alignment.print_alignment();
}
```

This code is used in chunk 61.

4.2 Memory leaks test

Because of the great amount of operations over arrays that are used in our library we though that there was great probabilities of memory leaks in our software so we decided to test it using a Free Software (cfr.2.1) project called Valgrind that includes a powerfull memory analyzer.

Using this software is very easy: we just had to launch our example program ⟨*two_sequence_example.cpp* 78⟩ with the command

```
$ valgrind --leak-check=yes ./two_sequence_example.bin
```

4.3 Random tests

We decided to test our functions against random generated sequences hoping to detect errors in algorithms implementation that didn't appear in our previous tests

We wrote code for calculating alignment scores by printed alignment, this way we could detect inconsistencies between them and find critical sequences for further tests. Decommenting the following debugging code we obtain additional output that will be used in ⟨*random_test.cpp* 76⟩.

The code reads the alignment memorized in `line1`, `line2` and `line3` and in the first part calculates the alignment score supposing we did an affine gap penalty (cfr.2.1) alignment, while in the second part it does the same for linear gap penalty (cfr.2.1) alignments.

75 ⟨*debug_random* 75⟩≡

```
//decomment for random test
/*
T true_score=0;
It=line2->begin();
for (int i=0;It!=line2->end();i++){
    if ((*line2)[i]==' '){//gap
        if ((*line1)[i]=='-'){//gap in A
            if (i>0 && (*line1)[i-1]=='-')
                true_score-=extend_gap_cost;
            else true_score-=gap_cost;
        } else {//gap in B
            if (i>0 && (*line3)[i-1]=='-')
                true_score-=extend_gap_cost;
            else true_score-=gap_cost;
        }
    }else true_score+=c_matrix->get_score((*line1)[i],(*line3)[i]);
    It++;
}

T lin_score=0;
It=line2->begin();
for (int i=0;It!=line2->end();i++){
    if ((*line2)[i]==' '){//gap
        if ((*line1)[i]=='-'){//gap in A
```

```
            lin_score-=gap_cost;
        } else {//gap in B
            lin_score-=gap_cost;
        }
    }else lin_score+=c_matrix->get_score((*line1)[i],(*line3)[i]);
    It++;
}

std::cout<<"True alignment score:"<<true_score<<" score:"<<lin_score << "\n";
//*/
```

This code is used in chunk 34.

The actual program generates random sequences of random length but not longer than the value in `max_seq_length`. It creates them upon a specified alphabet whose length is given in `alph_size` variable, then it calls all the alignment functions.

The whole procedure is repeated as many times as indicated in `repetitions` variable.

76 ⟨*random_test.cpp* 76⟩≡

```
⟨licence 2⟩
#include <iostream>
#include <stdlib.h>
#include <alibio/two_sequence.hpp>

int main(){
    int alph_size=4;
    int repetitions=2000;
    int max_seq_length=30;

    //Getting internal clock time for random
    srand(time(NULL));
    //declaring an alphabet
    alibio::alphabet my_alph("my alphabet");
    //adding symbols to the alphabet
    for (int i=0; i<alph_size; i++){
        my_alph.add_symbol(alibio::symbol('A'+i));
    }
    alibio::score_matrix<> myscores(my_alph);
    //changing scores
    myscores.set_match_score(5);
    myscores.set_mismatch_score(-4);
    myscores.set_gap_cost(10, 0.5);

    for (int n=0;n<repetitions;n++){
        vector<char> fst, snd;
        int n=rand() % max_seq_length;
        int m=rand() % max_seq_length;
        fst.reserve(n);
        snd.reserve(m);

        //generating strings randomly
        for (int i=0; i<n; i++){
            fst.push_back('A'+(rand() % alph_size));
            cout << fst[i];
        } cout << endl;
        for (int j=0; j<m; j++){
            snd.push_back('A'+(rand() % alph_size));
            cout << snd[j];
        } cout << endl;
        //creating the alignment matrix
        alibio::seq_alignment<> my_alignment(myscores, fst , snd);

        //do the alignments
        cout << "Optimal global alignment (linear gap penalty)\n";
```

```
        my_alignment.nw_align();
        my_alignment.print_alignment();

        cout << "Optimal local alignment (linear gap penalty)\n";
        my_alignment.sw_align();
        my_alignment.print_alignment();

        cout << "Optimal global alignment (affine gap penalty)\n";
        my_alignment.nw_affine_align();
        my_alignment.print_alignment();

        cout << "Optimal local alignment (affine gap penalty)\n";
        my_alignment.sw_affine_align();
        my_alignment.print_alignment();

        cout << "Optimal semiglobal alignment (affine gap penalty)\n";
        my_alignment.semiglobal_affine_align();
        my_alignment.print_alignment();

        cout << "Optimal global alignment (affine gap penalty-linear space)\n";
        my_alignment.nw_linear_align();
        my_alignment.print_alignment();

        cout << "Optimal local alignment (affine gap penalty-linear space)\n";
        my_alignment.sw_linear_align();
        my_alignment.print_alignment();

        cout<<"Optimal semiglobal alignment (affine gap penalty-linear space)\n";
        my_alignment.semiglobal_linear_align();
        my_alignment.print_alignment();
    }
    return 0;
}
```

Root chunk (not used in this document).

To analyze the huge output of our test program we wrote a GAWK script that verifies not only the correctness of reported alignment scores but also that affine alignments gave the same results of the corresponding linear-space ones. The script needs only score values to work, so we need to filter the ⟨*random_test.cpp* 76⟩ output with the `UNIX` command `grep` and then give it as input to the script ⟨*random_test.awk* 77⟩. We did it with the following command assuming we compiled ⟨*random_test.cpp* 76⟩ as `random_test.bin`

```
$ ./random_test.bin | grep score | ./random_test.awk
```

The script writes a line for each error found, if no errors are found nothing is printed.

77 ⟨*random_test.awk* 77⟩≡

```
#!/usr/bin/awk -f
BEGIN {}
{   //for each alignment we memorize the alignment score in the variable "old"
    if ((NR%16)%2==1) old=$3;
    if ((NR%16)%2==0){
        //in case of linear gap penalty we compare with the corresponding value
        if (NR%16==2 || NR%16==4){
            if ($4!=old) print old "<-->" $4 " ERROR " NR%16 "\n";
        }
        //in case of affine gap penalty we do the same
        else if($3!=old) print old "<-->" $3 " ERROR " NR%16 "\n";
    }
    else if (NR%16==5) var5=$3;//nw_affine_align score
    else if (NR%16==7) var7=$3;//sw_affine_align score
    else if (NR%16==9) var9=$3;//semiglobal_affine_align score
    else if (NR%16==11) {//comparing nw_affine and nw_linear
        if ($3!=var5) print var5 "<-->" $3 " GLOBAL\n";
    } else if (NR%16==13) {//comparing sw_affine and sw_linear
        if ($3!=var7) print var7 "<-->" $3 " LOCAL\n";
    } else if (NR%16==15) {//comparing semiglobal_affine and semiglobal_linear
        if ($3!=var9) print var9 "<-->" $3 " SEMIGLOBAL\n";
    }
}
END{}
```

Root chunk (not used in this document).

This test permitted to discover a number of subtle software bugs that couldn't be found with the previous tests.

Chapter 5

Examples

The following example program can be used as a quick tutorial on how the alignment functions can be used in conjunction with other `ALiBio` parts: let's start declaring an `alibio::alphabet` and adding symbols to it:

78 ⟨*two_sequence_example.cpp* 78⟩≡
⟨*licence* 2⟩

```
#include <iostream>
#include <stdlib.h>
#include <alibio/two_sequence.hpp>

int main(){
    //declaring an alphabet for DNA
    alibio::alphabet dna("DNA");

    //adding symbols to the alphabet
    dna.add_symbol(alibio::symbol('a',"Adenine"));
    dna.add_symbol(alibio::symbol('c',"Cytosine"));
    dna.add_symbol(alibio::symbol('t',"Thymine"));
    dna.add_symbol(alibio::symbol('g',"Guanine"));
```

This definition is continued in chunks 79–81.
Root chunk (not used in this document).

Then we create two `alibio::sequence` objects and assign to their primary structures the `alibio::alphabet` just created and two strings based on it:

79 ⟨*two_sequence_example.cpp* 78⟩+≡

```
//declaring 2 sequences with "bio_string" class as primary structure and
//"empty" class as secondary and tertiary structures
alibio::sequence<alibio::bio_string,alibio::empty,alibio::empty> seq_one;
alibio::sequence<alibio::bio_string,alibio::empty,alibio::empty> seq_two;

//setting the alphabet (working on "seq_one.primary")
seq_one.primary.set_alphabet(dna);
seq_two.primary.set_alphabet(dna);

//setting the sequence (working on "seq_one.primary")
seq_one.primary.set_sequence("ATTAGCATTAGACT");

//setting the sequence (working on "seq_two.primary")
seq_two.primary.set_sequence("GCCGTCTTC");
```

We need to create a ⟨*score_matrix* 8⟩ object constructing it from the same alphabet, now we can modify the scores as we like: by category (match, mismatch, gap) or by specifying a specific char couple:

80 ⟨*two_sequence_example.cpp* 78⟩+≡

```
alibio::score_matrix<float> myscores(dna);

//modifying the score_matrix
myscores.set_match_score(5);
myscores.set_mismatch_score(-4);
//modifying the score_matrix
myscores.set_gap_cost(5,0.5);
//modifying the score more selectively
myscores.set_cost('A','T',5);
//printing the score_matrix
myscores.print();
//creating alignment object
alibio::seq_alignment<float> my_alignment(myscores,
                                          seq_one.primary, seq_two.primary);
```

Finally we can call the alignment methods and print the generated alignments:

81 ⟨*two_sequence_example.cpp* 78⟩+≡

```
    cout << "Optimal global alignment (linear gap penalty)\n";
    my_alignment.nw_align();
    my_alignment.print_alignment();

    cout << "Optimal local alignment (linear gap penalty)\n";
    my_alignment.sw_align();
    my_alignment.print_alignment();


    cout << "Optimal global alignment (affine gap penalty)\n";
    my_alignment.nw_affine_align();
    my_alignment.print_alignment();

    cout << "Optimal local alignment (affine gap penalty)\n";
    my_alignment.sw_affine_align();
    my_alignment.print_alignment();

    cout << "Optimal semiglobal alignment (affine gap penalty)\n";
    my_alignment.semiglobal_affine_align();
    my_alignment.print_alignment();


    cout << "Optimal global alignment (affine gap penalty - linear space)\n";
    my_alignment.nw_linear_align();
    my_alignment.print_alignment();

    cout << "Optimal local alignment (affine gap penalty - linear space)\n";
    my_alignment.sw_linear_align();
    my_alignment.print_alignment();

    cout << "Optimal semiglobal alignment (affine gap penalty - linear space)\n";
    my_alignment.semiglobal_linear_align();
    my_alignment.print_alignment();

    return 0;
}
```

Chapter 6

Performance analysis

This chapter describes how to measure time and space performance of our algorithms, it also includes the scripts we used to produce the graphs presented in section 6.2.

6.1 Generating data

The following `c++` program generates the files `time.dat` and `space.dat` which contain time and memory used by ⟨*nw_align* 35⟩, ⟨*nw_affine_align* 38⟩ and ⟨*nw_linear_align* 44⟩ functions when operating over sequences of increasing length (from 0 to 3000 chars).

To obtain memory data, we need to decomment the following line which calls the `UNIX` command `ps` and transforms the result from KBytes to Bytes before writing it on the file `space.dat`.

The command supposes we compiled the ⟨*performance.cpp* 83⟩ source file to `performance.bin`.

82 ⟨*memory_line* 82⟩≡

```
//decomment for memory tests
/*
system("echo `/bin/ps --no-header -o vsz -C performance.bin`*1024 |\
        bc -l >> space.dat");
//*/
```

This code is used in chunks 35, 36, 38–40, and 44–46.

The program begins declaring some local variable whose meaning is explained in the code comments

83 ⟨*performance.cpp* 83⟩≡

```
⟨licence 2⟩
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <time.h>
#include <alibio/two_sequence.hpp>

int main(){
    //number of symbols in the alphabet
    int alph_size=4;
    //max length of the sequences
    int seq_size=3000;
    //variables for storing time measures
    time_t start,end;
    double dif;
    //output buffer for writing time measures
    ofstream out_time;
```

This definition is continued in chunks 84 and 85.
Root chunk (not used in this document).

Then the program generates an alphabet of `alph_size` size and deletes the files `time.dat` and `space.dat` if they exist. It also initialize the random number generator with a seed obtained by PC's clock.

84 ⟨*performance.cpp* 83⟩+≡

```
    //declaring an alphabet
    alibio::alphabet my_alph("my alphabet");
    //adding symbols to the alphabet
    for (int i=0; i<alph_size; i++){
        my_alph.add_symbol(alibio::symbol('A'+i));
    }
    alibio::score_matrix<int> myscores(my_alph);

    if( remove( "time.dat" ) == -1 )
        perror( "Error deleting file" );
    if( remove( "space.dat" ) == -1 )
        perror( "Error deleting file" );

    //Getting internal clock time for random seading
    srand(time(0));
```

The program then generates randomly the sequences and calls the alignment functions on them printing the used time on `time.dat` while the decommented ⟨*memory_line* 82⟩ prints memory usage on `space.dat`. Everything is repeated until sequence size equals `seq_size`.

85 ⟨*performance.cpp* 83⟩+≡

```
    for (int n=0;n<=seq_size;n+=1000){
        vector<char> fst, snd;
        fst.reserve(n); snd.reserve(n);

        //generating strings randomly
        for (int i=0; i<n; i++){
            fst.push_back('A'+(rand() % alph_size));
            cout << fst[i];
        }
        cout << endl;
        for (int i=0; i<n; i++){
            snd.push_back('A'+(rand() % alph_size));
            cout << snd[i];
        }
        cout << endl;
        //creating the alignment matrix
        alibio::seq_alignment<int> my_alignment(myscores, fst , snd);

//GLOBAL ALIGNMENT - LINEAR GAP PENALTY
        time (&start);//reset timer
        my_alignment.nw_align();//do the alignment
        my_alignment.print_alignment();
        //writing time elapsed in time.dat
        time (&end);
        dif = difftime (end,start);

        out_time.open("time.dat", ofstream::out | ofstream::app);
        out_time << dif << endl; out_time.close();

//GLOBAL ALIGNMENT - AFFINE GAP PENALTY
        time (&start);//reset timer
        my_alignment.nw_affine_align();//do the alignment
        my_alignment.print_alignment();
        //writing time elapsed in time.dat
        time (&end);
        dif = difftime (end,start);

        out_time.open("time.dat", ofstream::out | ofstream::app);
        out_time << dif << endl; out_time.close();

//GLOBAL ALIGNMENT - AFFINE GAP PENALTY - LINEAR SPACE
        time (&start);//reset timer
        my_alignment.nw_linear_align();//do the alignment
        my_alignment.print_alignment();
        //writing time elapsed in time.dat
        time (&end);
```

```
        dif = difftime (end,start);

        out_time.open("time.dat", ofstream::out | ofstream::app);
        out_time << dif << endl; out_time.close();
      }
      return 0;
}
```

The produced files are then formatted by the following GAWK script creating `gnuplot_time.dat` and `gnuplot_space.dat` with the `UNIX` commands

```
$ performance.awk time.dat >> gnuplot_time.dat
$ performance.awk space.dat >> gnuplot_space.dat
```

86 ⟨*performance.awk* 86⟩≡

```
#!/usr/bin/awk -f
BEGIN {ORS=" "; nchar="0";}
{
    if (NR==1) base=$0;

    if (NR%3==1) {
        print "\n" nchar "\t" $0;
        nchar+=10;
    }
    else print "\t" $0;
}
END{print "\n";}
```

Root chunk (not used in this document).

The data files obtained are then plotted with the following Gnuplot script.

87 ⟨*performance.plot* 87⟩≡

```
#!/usr/bin/gnuplot
set terminal postscript eps enhanced color
set xlabel "Input size (chars)"

    set out 'gnuplot_time.eps'
        #set xlabel "Input size (chars)"
        set ylabel "Time (sec)"
        set title "Time performance"
        plot \
            'gnuplot_time.dat' using 1:2 title 'nw align' \
                               smooth csplines lt 1 linewidth 3, \
            'gnuplot_time.dat' using 1:3 title 'nw affine align' \
                               smooth csplines lt 2 linewidth 3, \
            'gnuplot_time.dat' using 1:4 title 'nw linear align' \
                               smooth csplines lt 3 linewidth 3

    set out 'gnuplot_space.eps'
        #set xlabel "Input size (chars)"
        set ylabel "Space (Bytes)"
        set format y "%.0s{/Symbol \327}10^{%S}"
        set title "Space performance"
        #unset logscale
        plot \
            'gnuplot_space.dat' using 1:2 title 'nw align' \
                                smooth csplines lt 1 linewidth 3, \
            'gnuplot_space.dat' using 1:3 title 'nw affine align' \
                                smooth csplines lt 2 linewidth 3, \
            'gnuplot_space.dat' using 1:4 title 'nw linear align' \
                                smooth csplines lt 3 linewidth 3

set logscale xy
set xtics (1000,1500,2000,2500,3000)

    set out 'gnuplot_time_compl.eps'
        #set xlabel "Input size (chars)"
        set ylabel "Time (sec)"
        set format y "10^{%L}"
        set title "Time complexity (log axes)"
        plot [1000:] \
            'gnuplot_time.dat' using 1:2 title 'nw align' \
                               smooth csplines lt 1 linewidth 3, \
            'gnuplot_time.dat' using 1:3 title 'nw affine align' \
                               smooth csplines lt 2 linewidth 3, \
            'gnuplot_time.dat' using 1:4 title 'nw linear align' \
                               smooth csplines lt 3 linewidth 3, \
            (x**2)/100000 title 'quadratic' smooth csplines lt -1

    set out 'gnuplot_space_compl.eps'
        #set xlabel "Input size (chars)"
```

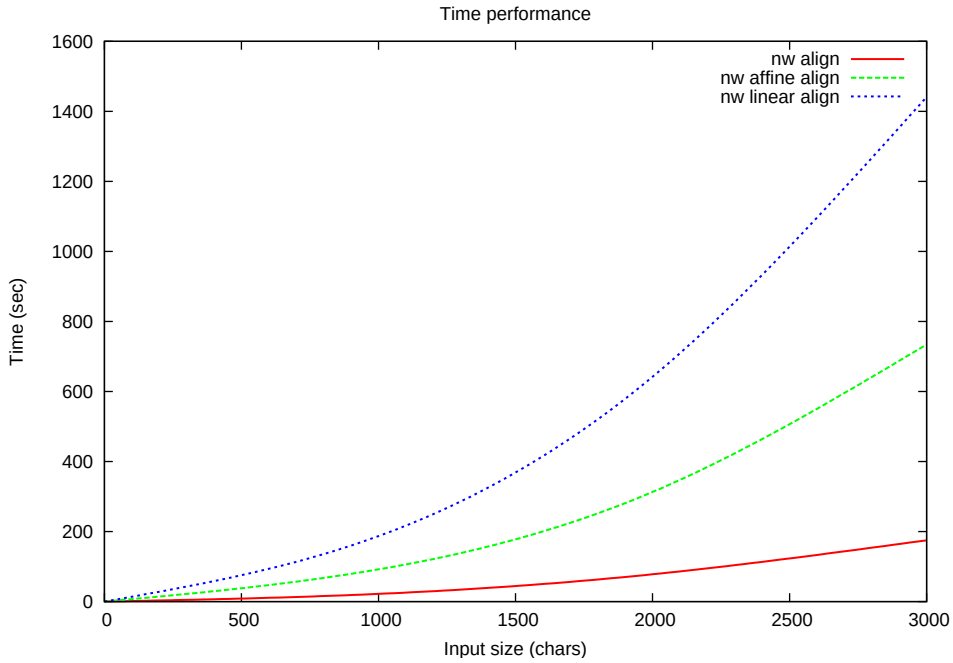
```
set ylabel "Space (Bytes)"
#set xtics (1000,1500,2000,2500,3000)
set format y "%.0s{/Symbol \327}10^{%S}"
set title "Space complexity (log axes)"
plot [1000:3000][:500000000] \
    'gnuplot_space.dat' using 1:2 title 'nw align'\
                        smooth csplines lt 1 lw 3,\
    'gnuplot_space.dat' using 1:3 title 'nw affine align' \
                        smooth csplines lt 2 lw 3, \
    x**2 title 'quadratic' smooth csplines lt -1
```

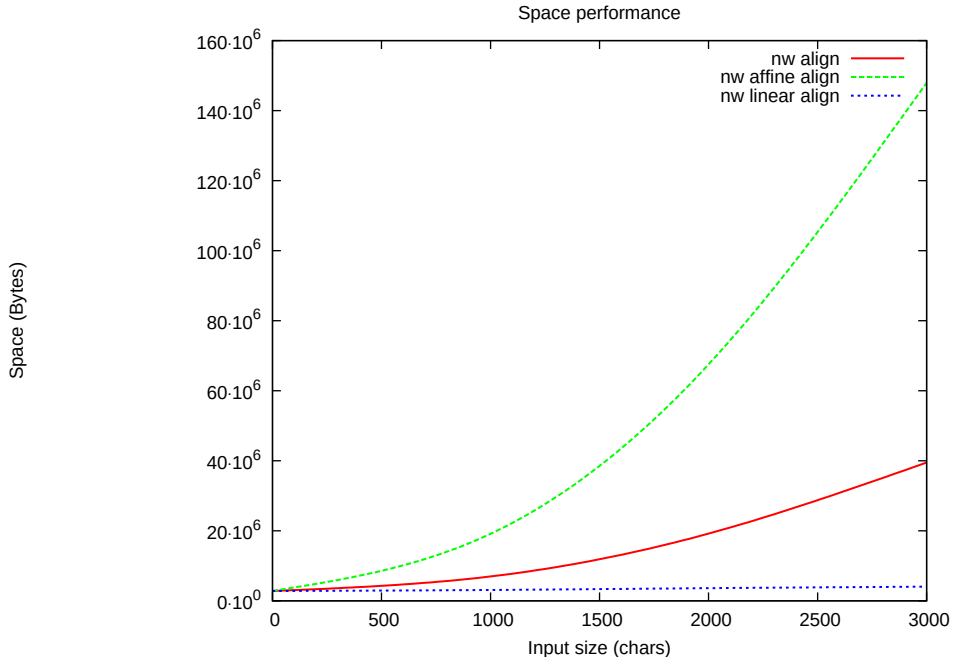
Root chunk (not used in this document).

6.2 Results

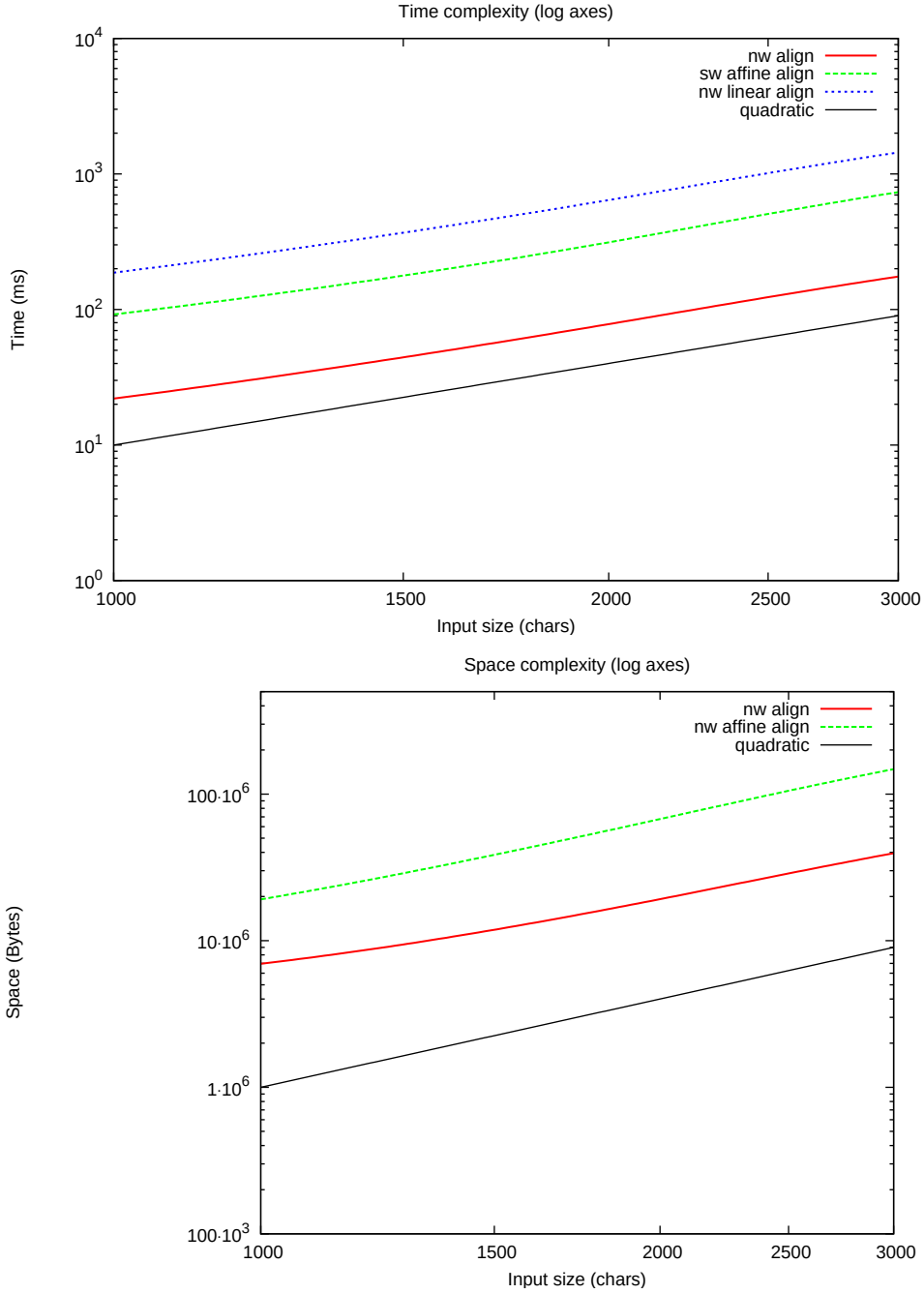
The following graphs represent time and space performances of our implementations, from them we can obtain informations about which algorithms are faster or memory-hungrier.

We can already notice that the space complexity of ⟨*nw_linear_align* 44⟩ is linear while the others are polynomial.





To have a visual indication of the polynomial functions rank we use logaritmic scale for both axes, now polynomials are straight lines whose slope depends on rank: if a function looks parallel to x^2 we can assure that its implementation is really quadratic. As expected, all the functions are quadratic-time, ⟨*nw_align* 35⟩ and ⟨*nw_affine_align* 38⟩ are also quadratic-space.



6.3 Performance comparison

In this section we compare the performances of our functions against the equivalent ones obtained from a well known project named EMBOSS[16] which is also Free Software (cfr.2.1). We used a Valgrind[17] tool called Massif which is a memory profiler that generates an easily understandable graph of memory usage and time compsunction for a given program.

In order to test the program over a realistic input we downloaded two uncorrelated nucleotide sequences from the European Bioinformatics website: embl:AF177870 and embl:L07770.

We also used a very common score matrix which corresponds to the EDNAFULL matrix in the EMBOSS package:

```
    A   T   G   C
A   5  -4  -4  -4
T  -4   5  -4  -4
G  -4  -4   5  -4
C  -4  -4  -4   5
```

We tested the Affine gap penalties versions of the algorithms using 10 as gap opening cost and 1 as extending cost.

The programs

All of the programs share a common part that takes care of creating the needed `alibio::seq_alignment` object:

88 ⟨*object_preparing* 88⟩≡

```
//declaring an alphabet for DNA
alibio::alphabet dna("DNA");

//adding symbols to the alphabet
dna.add_symbol(alibio::symbol('a',"Adenine"));
dna.add_symbol(alibio::symbol('c',"Cytosine"));
dna.add_symbol(alibio::symbol('t',"Thymine"));
dna.add_symbol(alibio::symbol('g',"Guanine"));

//declaring 2 sequences with "bio_string" class as primary structure and
//"empty" class as secondary and tertiary structures
alibio::sequence<alibio::bio_string,alibio::empty,alibio::empty> seq_one;
alibio::sequence<alibio::bio_string,alibio::empty,alibio::empty> seq_two;

//setting the alphabet (working on "seq_one.primary")
seq_one.primary.set_alphabet(dna);
seq_two.primary.set_alphabet(dna);

//setting the sequence (working on "seq_one.primary")
⟨embl:AF177870 89⟩
```

```
//setting the sequence (working on "seq_two.primary")
⟨embl:L07770 90⟩

alibio::score_matrix<int> myscores(dna);

//modifying the score_matrix
myscores.set_match_score(5);
myscores.set_mismatch_score(-4);
//modifying the score_matrix
myscores.set_gap_cost(10,1);

//creating alignment object
alibio::seq_alignment<int> my_alignment(myscores,
                                        seq_one.primary, seq_two.primary);
```

This code is used in chunks 91–93.

The lines of code that assign the downloaded sequences to the object

89 ⟨*embl:AF177870* 89⟩≡

```
seq_one.primary.set_sequence("gaacgcgaatgcctc ... aagaggttttcacag");
```

This code is used in chunk 88.

90 ⟨*embl:L07770* 90⟩≡

```
seq_two.primary.set_sequence("ggtagaacagcttca ... aaatttctttgcaagt");
```

This code is used in chunk 88.

Here are the actual programs used for graph creation:

91 ⟨*semiglobal_affine.cpp* 91⟩≡

```
⟨licence 2⟩
#include <iostream>
#include <stdlib.h>
#include <alibio/two_sequence.hpp>

int main(){
    ⟨object_preparing 88⟩

    cout << "Optimal semiglobal alignment (affine gap penalty)\n";
    my_alignment.semiglobal_affine_align();
    my_alignment.print_alignment();

    return 0;
}
```

Root chunk (not used in this document).

92 ⟨*sw_affine.cpp* 92⟩≡
⟨*licence* 2⟩
```
#include <iostream>
#include <stdlib.h>
#include <alibio/two_sequence.hpp>

int main(){
    ⟨object_preparing 88⟩

    cout << "Optimal local alignment (affine gap penalty)\n";
    my_alignment.sw_affine_align();
    my_alignment.print_alignment();

    return 0;
}
```
Root chunk (not used in this document).

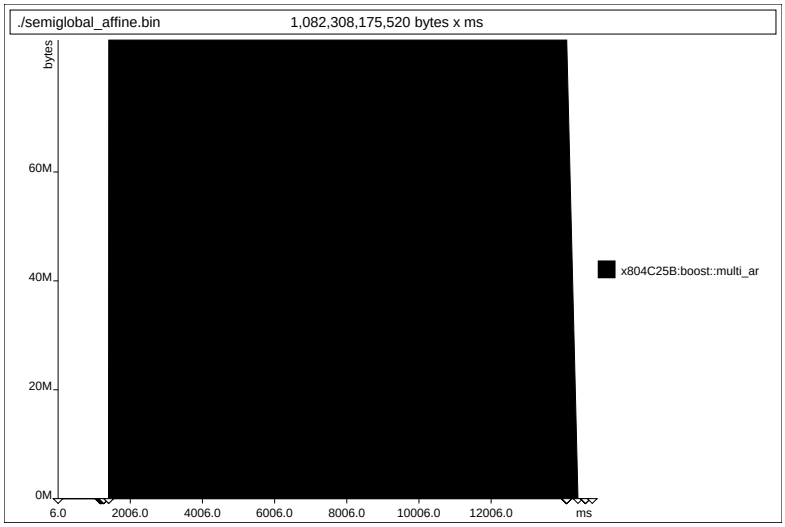
93 ⟨*nw_linear.cpp* 93⟩≡
⟨*licence* 2⟩
```
#include <iostream>
#include <stdlib.h>
#include <alibio/two_sequence.hpp>

int main(){
    ⟨object_preparing 88⟩

    cout << "Optimal global alignment (affine gap penalty - linear space)\n";
    my_alignment.nw_linear_align();
    my_alignment.print_alignment();

    return 0;
}
```
Root chunk (not used in this document).

6.3.1 Results

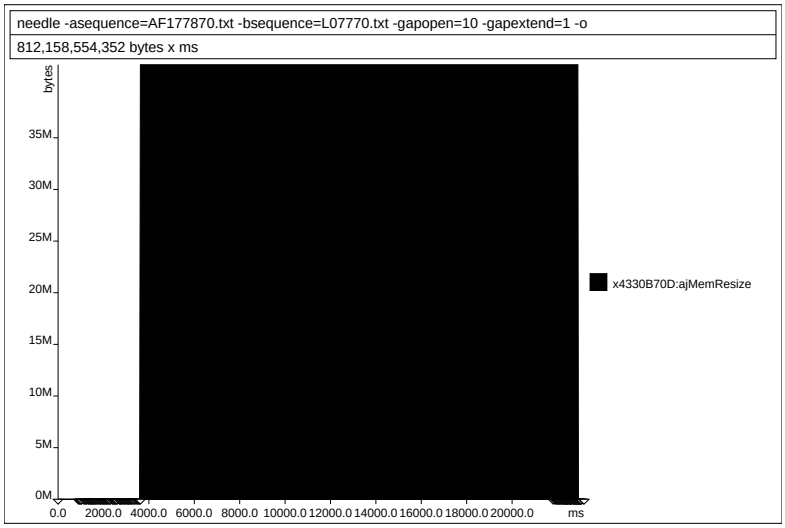
The graphs we obtained represent time (x axes) and space occupation (y axes) of the tested functions. We used the following command line where the actual `PROGRAM` tested is indicated at the top of each graph.

```
$ valgrind --tool=massif PROGRAM
```

Semiglobal alignment



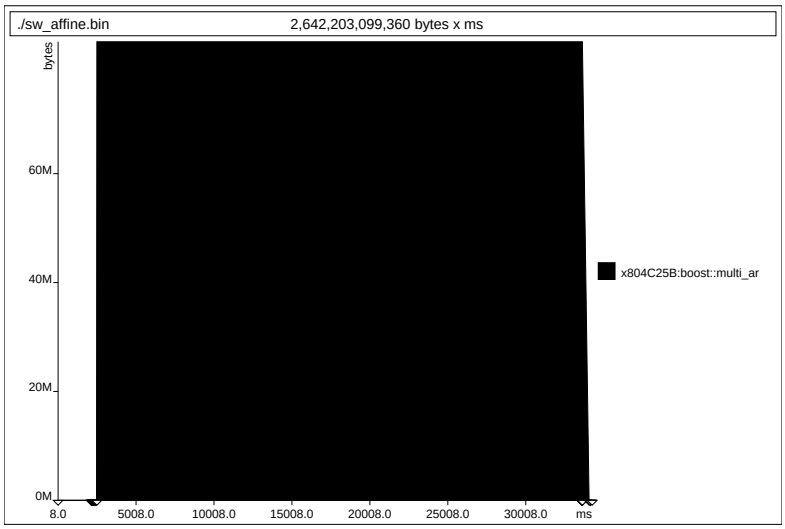
ALiBio ⟨*semiglobal_affine_align* 40⟩ function:
time=14sec space=80MB alignment score=1102



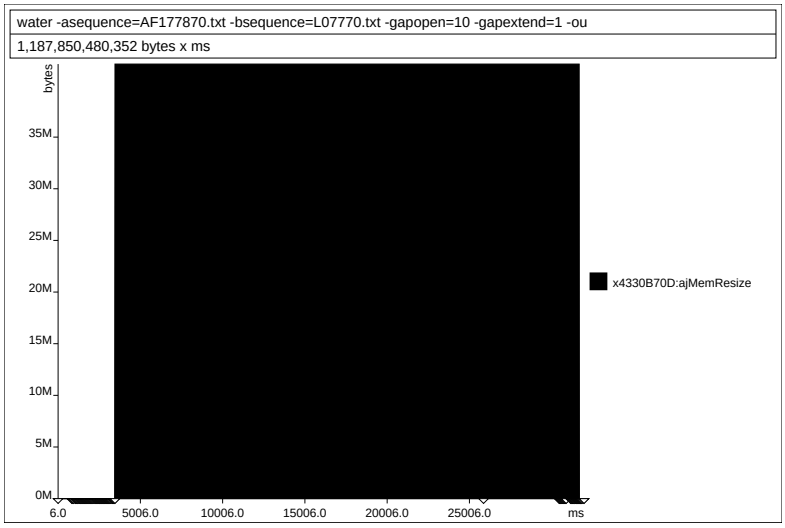
EMBOSS needle program:
time=22sec space=40MB alignment score=1102

While AliBio function is quite faster, EMBOSS program uses half space

Local alignment



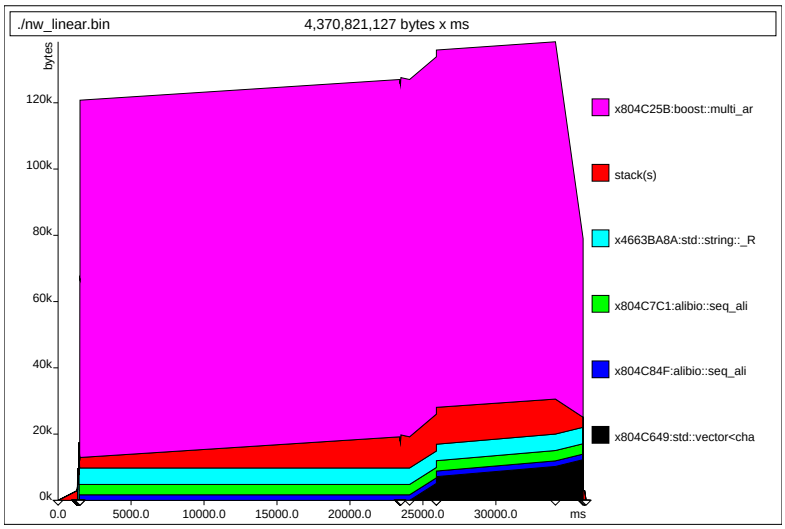
ALiBio ⟨*sw_affine_align* 39⟩ function:
time=35sec space=80MB alignment score=1113



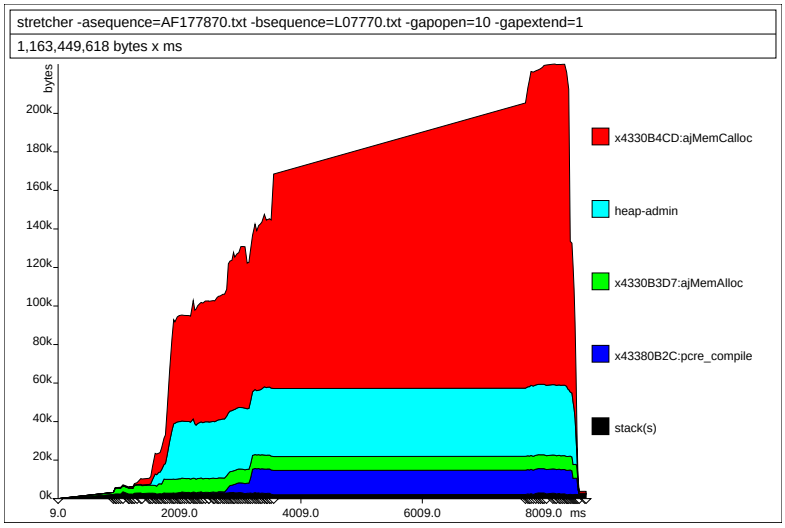
EMBOSS water program:
time=30sec space=40MB alignment score=1113

While time performances are quite similar, EMBOSS program uses half space

Linear-space global alignment



ALiBio ⟨*nw_linear_align* 44⟩ function:
time=35sec space=140kB alignment score=668



EMBOSS stretcher program:
time=9sec space=220kB alignment score=668

ALiBio function uses half space than EMBOSS program but is also 5 times slower.

Different color sections are associated with the different functions that actually reserve memory as reported in the caption.

Chapter 7

Conclusions

Once finished the stage period, the proposed objectives were reached. The requested algorithm implementation was produced, along with its documentation according to the project standard, also the implementation resource occupation is in line with the expected one.

The free software nature of the ALiBio project is an incentive for its improvement. In fact the bioinformatics software developers can freely use ALiBio inside their own programs, but also contribute with their own improvements and their expansions in developing the ALiBio libraries.

7.1 Possible future improvements

- On the performance side some improvements can be made to the given implementations, in particular the memory footprint can probably be furthermore reduced.
- Some modifications for better time performances could be added to the implemented algorithms, in particular a tecnique known as "Four Russians"[7] could reduce the time complexity by a factor of $log^2 n$.
- In the past years other algorithms have been developed for the peerwise alignment problem, in particular to find sub-optimal solutions; their implementation could be added to the present library.
- The ALiBio library has been thought to be useful for developers of bioinformatics applications, anyway the presented alignment functions could be easily used by end users if a Command Line Interface were developed.

  From that CLI, using pre-existing Free Software (cfr.2.1) like Pise[18] and Kaptain, a web interface and a graphical one can be obtained very quickly.

Chapter 8

Quick Methods reference

In this chapter we give a quick reference for all the methods implemented in the alignment library

8.1 `score_matrix` Class

`score_matrix (const alibio::alphabet&)` $\mapsto$ `void`
Description: ⟨*score_matrix constructor* 14⟩ creates a new object for the given `alibio::alphabet`
Precondition: -
Postcondition: - `~score_matrix (void)` $\mapsto$ `void`
Description: ⟨*score_matrix destructor* 15⟩ deallocates memory used by the ⟨*score_matrix* 8⟩
Precondition: -
Postcondition: - `set_score (char;char;T)` $\mapsto$ `void`
Description: ⟨*set_score* 17⟩ sets the score of mismatch between the two `char` to the given `T` value
Precondition: -
Postcondition: - `get_score (char;char)` $\mapsto$ `T`
Description: ⟨*get_score* 18⟩ returns the score of a mismatch between the two given `char`
Precondition: -
Postcondition: - `set_match_score (T)` $\mapsto$ `void`
Description: ⟨*set_match_score* 19⟩ sets the score for matching chars
Precondition: -
Postcondition: - `set_mismatch_score (T)` $\mapsto$ `void`
Description: ⟨*set_mismatch_score* 20⟩ sets the score for mismatching chars
Precondition: -
Postcondition: - `set_gap_cost (T;T)` $\mapsto$ `void`
Description: ⟨*set_gap_cost* 21⟩ sets the score for gap insertions
Precondition: -
Postcondition: - `print (void)` $\mapsto$ `void`
Description: ⟨*print* 24⟩ prints the score matrix
Precondition: -
Postcondition: -

8.2 seq_alignment Class

`seq_alignment (const score_matrix&;const vector<char>&;const vector<char>&)` ↦ `void`
Description: ⟨*seq_alignment constructors* 31⟩ create a new object for the given ⟨*score_matrix* 8⟩ and `vector<char>`
Precondition: -
Postcondition: -

`seq_alignment (const score_matrix&;biostring;biostring)` ↦ `void`
Description: ⟨*seq_alignment constructors* 31⟩ create a new object for the given ⟨*score_matrix* 8⟩ and `alibio::biostring`s
Precondition: -
Postcondition: -

`seq_alignment (const score_matrix&;const string&;const string&)` ↦ `void`
Description: ⟨*seq_alignment constructors* 31⟩ create a new object for the given ⟨*score_matrix* 8⟩ and `strings`
Precondition: -
Postcondition: -

`~seq_alignment (void)` ↦ `void`
Description: ⟨*seq_alignment destructor* 33⟩ deallocates memory used by the ⟨*seq_alignment* 25⟩
Precondition: -
Postcondition: -

`print_alignment (void)` ↦ `void`
Description: ⟨*print_alignment* 34⟩ prints the last calculated alignment
Precondition: -
Postcondition: -

`nw_align (void)` ↦ `void`
Description: ⟨*nw_align* 35⟩ calculates a global alignment using linear gap penalties
Precondition: -
Postcondition: -

`sw_align (void)` ↦ `void`
Description: ⟨*sw_align* 36⟩ calculates a local alignment using linear gap penalties
Precondition: -
Postcondition: -

`nw_affine_align (void)` ↦ `void`
Description: ⟨*nw_affine_align* 38⟩ calculates a global alignment using affine gap penalties
Precondition: -
Postcondition: -

`sw_affine_align (void)` ↦ `void`
Description: ⟨*sw_affine_align* 39⟩ calculates a local alignment using affine gap penalties

Precondition: -
Postcondition: -

`semiglobal_affine_align (void)` ↦ `void`
Description: ⟨*semiglobal_affine_align* 40⟩ calculates a semiglobal alignment using affine gap penalties
Precondition: -
Postcondition: -

`nw_linear_align (void)` ↦ `void`
Description: ⟨*nw_linear_align* 44⟩ calculates a global alignment using affine gap penalties and linear memory
Precondition: -
Postcondition: - `sw_linear_align (void)` ↦ `void`
Description: ⟨*sw_linear_align* 45⟩ calculates a local alignment using affine gap penalties and linear memory
Precondition: -
Postcondition: - `semiglobal_linear_align (void)` ↦ `void`
Description: ⟨*semiglobal_linear_align* 46⟩ calculates a semiglobal alignment using affine gap penalties and linear memory
Precondition: -
Postcondition: -

Bibliography

[1] Dan Gusfield. Algorithms on Strings, Trees and Sequences. University of California, Davis

[2] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford. Introduction to Algorithms, second edition, MIT Press and McGraw-Hill. ISBN 0-262-53196-8.

[3] Needleman, S. B. and Wunsch, C. D. (1970) J. Mol. Biol. 48, 443-453.

[4] Smith TF, Waterman MS (1981) J. Mol. Biol 147(1);195-7

[5] E. Myers and W. Miller, "Optimal Alignments in Linear Space," CABIOS 4, 1 (1988), 11-17.

[6] Hirshberg, D.S., A linear space algorithm for computing maximal common subsequences, Comm. Assoc. Comput. Mach., 18(6), 341-343, (1975).

[7] Four Russian Algorithm: Alrazarov, Dinic, Kronvod, Faradzev, 1970.

[8] ALiBio: Algorithm Library for BIOinformatics

[9] Literate Programming: Donald E. Knuth (Stanford, California: Center for the Study of Language and Information, 1992), xvi+368pp.

[10] NOWEB: Ramsey, N. (1994, September). Literate programming simplified. IEEE Software 11(5), 97–105.

[11] The C++ Programming Language: Addison-Wesley, ISBN 0-201-88954-4 and 0-201-70073-5.

[12] Alexander Stepanov and Meng Lee, The Standard Template Library. HP Laboratories Technical Report 95-11(R.1), November 14, 1995.

[13] BOOST: C++ libraries.

[14] LaTeX: Tobias Oeitker: *The not so short introduction to LaTeX*, Copyright ©1998 Tobias Oeitker and all contributors

[15] CVS: Concurrent Versions System.

[16] EMBOSS: The European Molecular Biology Open Software Suite (2000) Rice,P. Longden,I. and Bleasby,A.

[17] Valgrind: Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. Electronic Notes in Theoretical Computer Science 89 No. 2, 2003.

[18] Pise: Letondal C. A Web interface generator for molecular biology programs in Unix, Bioinformatics, 17(1), 2001, pp 73-82.