University of Antwerp

Chemo-informatics and computational drug design

Prof. Dr. Hans De Winter

University of Antwerp

Campus Drie Eiken, Building A

Universiteitsplein 1, 2610 Wilrijk, Belgium

Gefinancierd door de Europese Unie
NextGenerationEU

Table of Contents

Chapter 1. Introduction to Python

1. Introduction

Python is a general-purpose programming language that is very powerful yet also quite easy to learn, read and write. Many applications in Python exist, ranging from web development to machine learning and data science. Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. As Python supports modules and packages, it can be used for a variety of applications. Simply by loading different modules, the user can incorporate a wide variety of different and specific functions into its code base. Statistical, mathematical and machine learning methods are introduced by importing modules such as scikit-learn and scipy, while bioinformatics-approaches are largely governed by packages such as Biopython. Finally, plotting and data visualisation has been made easy with Python packages like Matplotlib.

To work with molecules and drugs, there exist Python packages like RDKit and OpenBabel to help you with that. In fact, as almost all molecule manipulations can be done with Python in combination with a package like RDKit, it is important for the interested chemo-informatician to learn and use Python. For this reason, this course will start with a short introduction on Python. In subsequent chapters, key concepts of the chemo-informatics package RDKit will be introduced.

2. Installing Python

Python can be installed and run from almost every computer, be it a Mac, Windows or Linux machine. There are even implementations of Python on the small Raspberry Pi systems, illustrating how widespread and powerful Python really is.

Python comes in two main flavors, namely version 2 and version 3+. The former has been discontinued since the beginning of 2020, so we will only focus on version 3+ (3.7, 3.8, and 3.9). In order to distinct between both flavors, the name Python2 is often used to denote Python version 2, while the name Python3 referes to Python version 3+. In this course, we will refer to Python 3+ when talking about Python as such.

Installation of Python can be achieved in many ways, but probably the easiest ways are through means of the Anaconda installer for a local installation of Python on your computer, or by using the Google Colab interface for a web-based usage of Python without the need to install software on your system. In the following sections, both approaches are illustrated.

2.1. Anaconda

Anaconda is a free, easy-to-install package manager, environment manager, and Python distribution with a collection of many open source packages. Installation of Anaconda is first required, and subsequently many Python packages can be installed in their own environments, making it safe to try-out different Python packages without introducing the risc that packages might conflict and break already installed packages.

An extensive and well-writing installation manual is available from the Anaconda documentation portal. This installation manual describes the required steps to take for insalling on Windows, macOS and Linux systems.

2.2. Google Colab

Perhaps the easiest way to learn Python is with the Google Colab application. This is a web-based application that allows one to write and execute Python code in your own browser and with no installation or required configuration settings. In order to use Google Colab for the first time, you can look at this nice tutorial and follow the different steps.

3. Python concepts

3.1. Importing modules

One of the powerful aspects of Python is its use of modules. With modules, the user can select what kind of functionality is needed by importing the corresponding modules that contain the required functionalities. Python modules are imported using the `import` statement:

```
>>> import math
>>> math.sqrt(25)
5.0
>>> math.sin(math.pi)
1.2246467991473532e-16
```

In this example, the `math` module is imported. This module contains many mathematical functions and constants (such as `math.pi`).

It is also possible to import only specific functions from the module, as illustrated in the following example in which only the `sqrt` function is loaded:

```
>>> from math import sqrt
>>> sqrt(25)
5.0
>>> math.sin(math.pi)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'math' is not defined
```

One may of course also import multiple functions at once:

```
>>> from math import sqrt, sin, pi
>>> sqrt(25)
5.0
>>> sin(pi)
1.2246467991473532e-16
```

Aliases (or synonyms) to the imported modules may be provided as desired:

```
>>> import math as mathematics
>>> mathematics.pi
3.141592653589793
```

To know which functions and constants are defined in the spcified module, one can use the `dir()` function:

```
>>> dir(math)
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e',
'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder', 'sin', 'sinh',
'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

3.2. Basic operations

Basic operations in Python include addition, substraction, multiplication and division. The exact effect of each of these operations depends on the underlying data type (see below), but in the case of numbers as data types the results are as expected:

```
>>> 10 + 4
14
>>> 10 - 4
6
>>> 10 * 4
40
>>> 10 ** 4
10000
```

```
>>> 10 / 4
2.5
>>> 5 % 4
1
>>> 10 // 4
2
```

Python works also with `boolean`s, which are data types that can have the value of `true` or `false`. Boolean operations include comparisons such as > (larger than), >= (larger or equal than), < (smaller than), <= (smaller or equal than) and == (equal) or `!` = (not equal). In addition, there exist also `and`, `or` and `not` as boolean operations. All data types can automaticallly be converted to a `boolean` type:

```
>>> 5 > 3
True
>>> 5 < 3
False
>>> 5 != 3
True
>>> 5 >= 3 and 10 > 3
True
>>> 5 >= 3 and not 10 > 3
False
>>> True
True
>>> not False
True
```

3.3. Data types

Standard Python contains already a multitude of data types, and these can be extended by the programmer using the `class` data type (below more on classes). The common data types in Python are integer numbers (`int`), fractional numbers (`float`), booleans (`bool`), text (`str`) and the `None` type. The type can be determined with the `type()` function:

```
>>> type(2)
<class 'int'>
>>> type(2.4)
<class 'float'>
>>> type("two")
<class 'str'>
>>> type('two')
<class 'str'>
>>> type(True)
<class 'bool'>
>>> type(None)
<class 'NoneType'>
```

In order to find out if a given object is of a given type, the `isinstance()` function is useful:

```
>>> isinstance(2.0, int)
False
>>> isinstance(2, int)
True
>>> isinstance(2, float)
False
>>> isinstance(2.0, float)
True
>>> isinstance(2.0, (float, int))
True
```

Objects often can be converted to other types (not always, as shown in the last example):

```
>>> int(2.1)
2
>>> float(2)
2.0
>>> str(2.9)
'2.9'
>>> int("two")
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'two'
```

With respect to conversion to a `boolean` type, `0`, `None` and empty containers (see below) are all converted to `False`, and all the other objects are converted to `True`:

```
>>> bool(0)
False
>>> bool(0.00000001)
True
>>> bool(None)
False
>>> bool(“”)
False
>>> bool(1)
True
```

3.4. Variables

Variables in Python are placeholders that can contain certain values. These values are assigned to the variables:

```
>>> a = 3
>>> a
3
>>> b = 7
>>> a + b
10
>>> b ** a
343
>>> b * 2
14
>>> c = a + b
>>> c
10
>>> a = 1
>>> a = 2
>>> a = 3
>>> a
3
>>> a = “another value into a”
>>> a
'another value into a'
```

3.5. Lists

As the name suggests, lists are lists of ordered and multiple data types. Lists are denoted with `[]`, in which `[` denotes the start of the list, and `]` the end. Elements can be added to the list using the `append()` method:

```
>>> my_list = []
>>> my_list
[]
>>> my_list.append(3)
>>> my_list
[3]
>>> my_list.append(4)
>>> my_list
[3, 4]
>>> another_list = ["This", "are", "five", "list", "elements"]
>>> another_list
['This', 'are', 'five', 'list', 'elements']
```

List elements can be accessed with their index (starting at zero). The last element can be accessed with the -1 index and the total number of elements is retrieved with the `len()` function:

```
>>> another_list[0]
'This'
>>> another_list[2]
'five'
```

```
>>> another_list[-1]
'elements'
>>> len(another_list)
5
```

Individual elements can be removed from a list using the `remove()` and `pop()` methods:

```
>>> another_list.remove('list')
>>> another_list
['This', 'are', 'five', 'elements']
>>> another_list.pop(1)
'are'
>>> another_list
['This', 'five', 'elements']
```

Lists can be sorted:

```
>>> another_list.sort()
>>> another_list
['This', 'elements', 'five']
```

There are many more functions and methods that can be used in conjunction with lists. For the interested student, please have a look at the W3 Schools webpage on lists.

3.6. Tuples

Tuples are like lists, but tuples cannot be modified with methods like `append()` or `remove()`. Tuples are initiated and remain then constant:

```
>>> digits = (1,2,3)
>>> digits
(1, 2, 3)
>>> digits[1]
2
>>> len(digits)
3
>>> digits * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> digits.append(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

3.7. Strings

Strings are sequences of characters. They share some similarities with lists such as accessing elements with indices and the `len()` method:

```
>>> txt = "this is a string"
>>> txt[1]
'h'
>>> txt[1:4]
'his'
>>> len(txt)
16
```

In addition, strings also possess some methods that are typical for text:

```
>>> txt2 = txt.upper()
>>> txt2
'THIS IS A STRING'
>>> txt2.lower()
'this is a string'
>>> txt.split()
['this', 'is', 'a', 'string']
```

The W3 Schools webpage on strings provides examples of many other methods related to strings.

3.8. Dictionaries

Dictionaries are structures which can contain multiple data types which are ordered as key-value pairs. For each (unique) *key*, the dictionary contains an associated *value*. Keys can be strings, numbers, or tuples, while the corresponding values can be any Python object:

```
>>> d = {}
>>> d
{}
>>> d = dict()
>>> d = {1: "one", "two": 2, 3: [1,2,3,4], "four": None}
{1: 'one', 'two': 2, 3: [1, 2, 3, 4], 'four': None}
```

The dictionary elements can be accessed as key/value pairs:

```
>>> d[1]
'one'
>>> d[105] = “OneHundredAndFive”
>>> d
{1: 'one', 'two': 2, 3: [1, 2, 3, 4], 'four': None, 105: 'OneHundredAndFive'}
>>> d[105] = None
>>> d
{1: 'one', 'two': 2, 3: [1, 2, 3, 4], 'four': None, 105: None}
```

As already shown in the previous example, dictionary elements can be added by providing the key/value pair using square brackets. Deletion of dictionary elements is possible with the `del()` function:

```
>>> del d[105]
>>> d
{1: 'one', 'two': 2, 3: [1, 2, 3, 4], 'four': None}
```

For more information, see the W3 Schools webpage on dictionaries.

3.9. Sets

Sets are containers very similar to dictionaries, but sets have only the unique keys and no values. Sets can be used to generate lists of unique elements:

```
>>> l = set()
>>> l.add(1)
>>> l.add(2)
>>> l.add(3)
>>> l.add(3)
>>> l.add(3)
>>> l
{1, 2, 3}
>>> l.remove(3)
>>> l
{1, 2}
```

For more information, see the W3 Schools webpage on sets.

3.10. Control statements

Conditional statements

Conditional statements (`if`, `else`, and `elif`) control the flow of a program depending on the outcome of certain conditions. These conditions are evaluated and depending on the outcome (`True` or `False`) a certain flow direction is followed:

```
>>> x = 3
>>> if x == 3:
...     print(“Three”)
... elif x == 4:
...     print(“Four”)
... else:
```

```
...     print(“Not 3 and not 4”)
...
Three
```

Loops

Repetitive processes are implemented using loops. There are two kinds of loops, namely the `for` loop and the `while` loop. Let’s start with the `for` loop to illustrate how one can iterate over all elements in a list:

```
>>> fruits = ["apple", "lemon", "strawberry"]
>>> for fruit in fruits:
...     print(fruit)
...     print(fruit.upper())
...
apple
APPLE
lemon
LEMON
strawberry
STRAWBERRY
```

The `range()` keyboard may come in handy in this context. With this keyword, a range of numbers is generated. For example, `range(10)`, which is equal to `range(0,10)`, returns a list of 10 integers ranging from 0 to 9 inclusive:

```
>>> for i in range(10):
...     print(i)
...
0
1
2
3
4
5
6
7
8
9
```

This feature may be exploited to loop over a list using list indices:

```
>>> for i in range(len(fruits)):
...     print(i, fruits[i])
...
0 apple
1 lemon
2 strawberry
```

The second keyword for looping is `while`. This keywords evaluates the expression given as argument, and as long as this experession converts to `True`, the block contained by the `while` keyword is executed:

```
>>> i = 1
>>> while i <= 4:
...     print(i)
...     i += 1
...
1
2
3
4
```

Breaks and continuation

To interrupt a loop, the `break` keyword may be used:

```
>>> for i in range(5):
...     print(i)
...     if i == 3:
```

```
...             break
...
0
1
2
3
```

When the `break` keyword is encountered, the loop terminates and the program flow moves out of the current loop. This is comparable to the `continue` keyword, however in that case the program continues at the beginning of the current loop:

```
>>> for i in range(5):
...     if i == 1 or i == 3:
...             print(i)
...     else:
...             continue
...
1
3
```

3.11. Functions

Functions are sets of instructions grouped together. Functions are launched when called, and they can have multiple input values (called the arguments) and a single return value. Functions come in handy when the same set of instructions has to be repeated on different vaiables or values. They are defined using the `def()` keyword:

```
>>> def sum(x,y):
...     s = x + y
...     return s
...
>>> sum(2,3)
5
>>> sum(100,4)
104
```

In the preceding example, a `sum()` function is defined that takes two arguments and returns a single value. It is also possible to define default values for the input arguments:

```
>>> def sum(x=1,y=2):
...     s = x + y
...     return(s)
...
>>> sum()
3
>>> sum(4,5)
9
```

Functions are defined and used extensively in Python, and form the basis of many packages that may be imported in your Python code. For example, the `math.sqrt()` that was mentioned on page 8 is actually a function that takes a single argument and returns the square root of that argument.

Although a function can only return a single value by definition, it is possible to circumvent this limitation by returning a tuple, list or dictionary:

```
>>> def sum_substraction_multiplication(a, b):
...     return(a+b, a-b, a*b)
...
>>> a = sum_substraction_multiplication(2,3)
>>> a
(5, -1, 6)
>>> print("2 + 3 = ", a[0])
2 + 3 =  5
>>> print("2 - 3 = ", a[1])
2 - 3 =  -1
>>> print("2 * 3 = ", a[2])
2 * 3 =  6
```

3.12. Reading and writing files

Writing to text files is achieved using a three-steps procedure: 1) open the file, 2) write to the file, 3) close the file:

```
>>> f = open("test.txt", "w")   # 'w': write to the file
>>> f.write("Hello\n")
6
>>> f.write("Line 2\n")
7
>>> f.write("\n")
1
>>> f.write("\n")
1
>>> f.write("\n")
1
>>> f.write("Last line\n")
10
>>> f.close()
```

This code generates a new file with filename `test.txt`. Should the `text.txt` file already exists, then this existing file is overwritten by the new data. This file contains six lines:

```
Hello
Line 2



Last line
```

Reading a file is also done in three steps: opening, reading, and closing:

```
>>> f = open("test.txt", "r")   # 'r': open the file to read
>>> for line in f.readlines():
...     print(line)
...
Hello

Line 2









Last line

>>> f.close()
```

It is common practice to remove any newline characters at the end of each line [using the `strip()` method], and to skip over empty lines:

```
>>> f = open("test.txt", "r")   # 'r': open the file to read
>>> for line in f.readlines():
...     line = line.strip()
...     if line is None or line == "": continue
...     print(line)
...
Hello
Line 2
Last line
>>> f.close()
```

3.13. Further reading

Python is a rich scripting language with many more functions and options than can ever be covered in this short chapter. Google is a perfect source of information to search for Python functions that you might need (for example, "how to read in text files in python"). In addition, there exist many tutorial sites on the internet, such as W3 Schools for Python.

4. Exercises

4.1. Exercise 1

Create a function `calc()` that acts as a simple calculator. Make sure to implement four operations: "add", "substract", "divide", "multiply". If the operation is not specified, default to addition. If the operation is misspecified, return an prompt message. For example, `calc(4,5,"multiply")` should return 20, `calc(3,5)` should returns 8, and `calc(1, 2, "something")` should return an error message.

4.2. Exercise 2

Given a list of numbers, create a function that returns a list where all adjacent duplicate elements have been reduced to a single element. For example, [1, 2, 2, 3, 2] returns [1, 2, 3, 2].

4.3. Exercise 3

Copy/paste the BSD 4 clause license (https://en.wikipedia.org/wiki/BSD_licenses) into a text file. Read the file and count the occurrences of each word within the file. Store the words' occurrence number in a dictionary.

4.4. Exercise 4

Write a program that enumerates all different configurations that are possible when positioning three 1's into a an array of ten cells. The following configurations illustrate two valid examples:

```
0 1 1 0 0 1 0 0 0 0 0
```

```
0 1 0 0 0 1 0 0 0 0 1
```

Note: with ten positions and three 1's to be placed, the total number of possible configurations (*n*) should be:

$$n = \frac{10!}{3!\,(10-3)!} = 120$$

Chapter 2. Representing molecules in the computer

1. Molecular formats

In order for computers to work with molecular information, molecules need to be converted in a format that computers can understand. During the last decades many different approaches have been developed, ranging from accurate quantum chemical representations with wavefunctions describing the position of electrons and the corresponding atom nuclei, up to the simplest representations in which only the molecular topology is described (Figure 1).

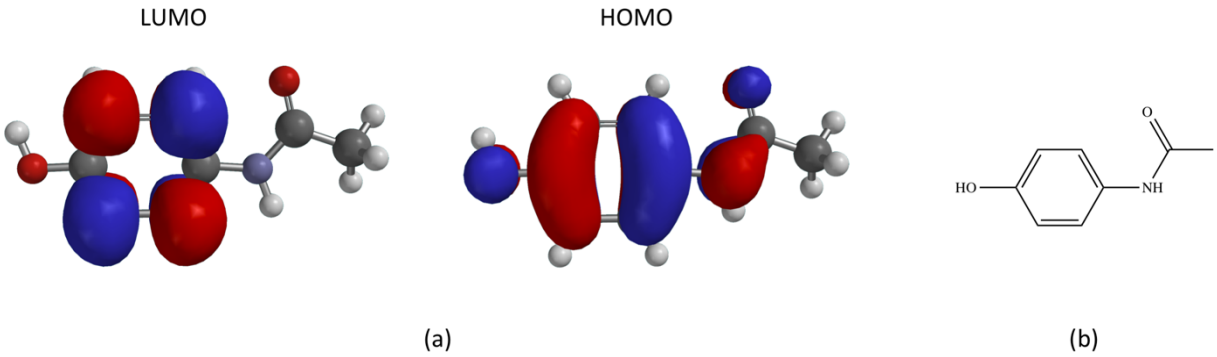


Figure 1. Different representations of the same molecule (paracetamol). (a) Two quantum chemical representations, the left one showing the lowest unoccupied molecular orbital (LUMO) and the right one showing the highest occupied molecular orbital (HOMO). (b) Example of a simple molecular representation in which the atom types and their connections are shown.

Although the fact that quantum chemical representations are considered to be the most accurate and contain the largest amount of *ab initio* information, in practice this approach is not very useful since it is extremely time consuming to generate and very expensive in terms of disk storage space. For this reason, alternative methods have been developed in which the majority of the chemical information is removed and considered to be implicitly present. For example, using the topological molecular representation shown in Figure 1b, it is implicitly assumed that:

1) each carbon, oxygen and nitrogen atom contain 6, 8 and 7 electrons, respectively,
2) the six-membered ring is aromatic,
3) each of the atoms is connected to an appropriate number of hydrogen atoms so that each atomic valence is correctly represented.
4) the oxygen atoms contain two lonepairs each and the nitrogen atom contains one lone pair,

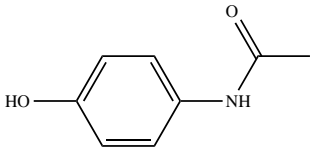
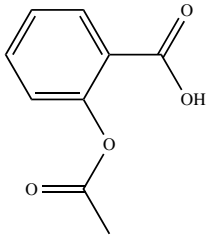
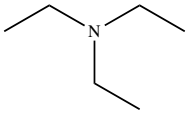
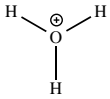
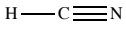
Also, the topological representation does not contain any information about the three-dimensional shape of the molecule, so in this case it is also implicitly assumed that the conformation of this molecule can be generated based on knowledge from other molecules and which has been collected using X-ray techniques (the Cambridge Structural Database is a comprehensive resource of almost a million X-ray structures of small molecules, see https://www.ccdc.cam.ac.uk). Hence, the majority of chemical representations assume some prior chemical knowledge; the main difference between most of the molecular formats is mainly in the amount of prior knowledge that is assumed. There are many chemical formats available (see https://en.wikipedia.org/wiki/Chemical_file_format for a complete overview), but the ones most frequently used are the SMILES, SDF and PDB formats.

1.1. SMILES

SMILES stands for Simplified Molecular Input Line Entry Specification and was developed in the 80's by David Weininger of Daylight Chemical Information Systems Inc. in the USA [1]. The format is a specification in the form of a line notation as it describes the structure of molecules using short sequences of characters (ASCII strings). A

complete format description is available at http://www.daylight.com/dayhtml/doc/theory/theory.smiles.html. Some examples are given in Table 1:

Table 1. Examples of SMILES representations

Name	Structure	Example SMILES
Paracetamol		`Oc1ccc(NC(C)=O)cc1`
Aspirin		`O=C(O)c1ccccc1OC(C)=O`
Triethylamine		`CCN(CC)CC`
Hydronium ion		`[H][O+]([H])[H]` `[OH3+]`
Hydrogen cyanide		`N#C`

SMILES notation consists of a series of characters containing no spaces. Hydrogen atoms may be omitted or included. Aromatic structures may be specified directly or in Kekulé form. There are five generic SMILES encoding rules, corresponding to specification of atoms, bonds, branches, ring closures, and disconnections.

Atoms

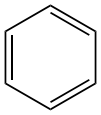
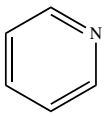
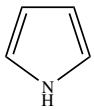
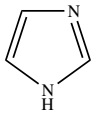
Atoms are represented by their standard abbreviation of the chemical elements and surrounded by square brackets, such as `[Ag]` for silver. In practice however, brackets are omitted in cases of atoms which:

- are in the organic subset of B, C, N, O, P, S, F, Cl, Br, or I, *and*
- have no formal charge, *and*
- have the number of hydrogens attached implied by their normal valences, *and*
- are the normal isotopes, *and*
- are not chiral centres.

All other elements or configurations must be enclosed in brackets, and have charges and hydrogens that are specified explicitly. For instance, the SMILES for water may be written as O or [OH2] or [H]O[H], but the hydronium ion can only be specified as `[OH3+]` or `[H][O+]([H])[H]`. When brackets are used, the symbol H should be added if the atom in brackets is bonded to one or more hydrogen, followed by the number of hydrogen atoms if this is greater than 1, then by the sign '+' for a positive charge or by '-' for a negative charge. For example, `[NH4+]` should be used to represent ammonium, although that `[H][N+]([H])([H])[H]` may also be used. If there is more than one charge, it is normally written as a digit like `[Ca+2]`.

Aromatic atoms, like in the benzene ring, may also be represented in their lower-case form. For the B, C, N, O, P and S atoms this becomes 'b', 'c', 'n', 'o', 'p' and 's', respectively. Aromatic nitrogen bonded to hydrogen, as found in pyrrole, must be represented as `[nH]` (Table 2):

Table 2. SMILES representation of aromaticity

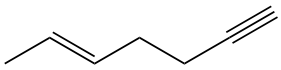
Name	Structure	Example SMILES
Benzene		`c1ccccc1` `C1=CC=CC=C1`
Pyridine		`n1ccccc1` `C1=CC=CN=C1`
Furan		`o1cccc1` `C1=COC=C1`
Pyrrole		`c1ccc[nH]1` `C1=CNC=C1`
Imidazole		`n1c[nH]cc1` `C1=CNC=N1`

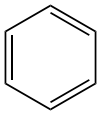
Bonds

A single bond is represented using the symbol '-'. Bonds between non-aromatic atoms are assumed to be single unless specified otherwise and are implied by the adjacency of the atoms in the SMILES string. Although single bonds may be written as "-", this is usually omitted. For example, the SMILES for ethanol may be written as `C-C-O`, `CC-O` or `C-CO`, but is usually written as `CCO` or `OCC`.

Double and triple bonds are represented by the symbols '=' and '#', respectively, as illustrated by the SMILES of carbon dioxide (`O=C=O`) and hydrogen cyanide (`C#N`). An aromatic 'one and a half' bond may be indicated with '**:**', although that the '**:**' may be omitted when lower-case atom symbols are used (Table 3).

Table 3. Different SMILES representations of bonds

Name	Structure	Example SMILES
5-hepten-1-yne		`CC=CCCC#C` `C-C=C-C-C-C#C`

Benzene		`c1ccccc1`
		`C1=CC=CC=C1`
		`C1=C-C=C-C=C-1`
		`C1:C:C:C:C:C:1`

Rings

Ring structures are written by breaking each ring at an arbitrary point to make an acyclic structure and adding numerical ring closure labels to show connectivity between non-adjacent atoms. For example, cyclohexane and dioxane may be written as `C1CCCCC1` and `O1CCOCC1` respectively. And methylcyclohexane and 1,2-dimethylcyclohexane are represented as `CC1CCCCC1` and `CC1CCCCC1C`, respectively (Figure 2).

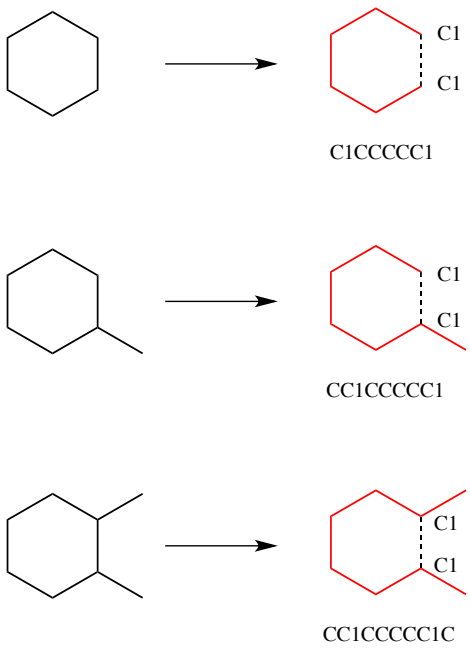


Figure 2. Illustration of how the ring labeling procedure works for the SMILES representation. Cyclic structures are represented by breaking one bond in each ring. The bonds are numbered in any order, designating ring opening (or ring closure) bonds by a digit immediately following the atomic symbol at each ring closure.

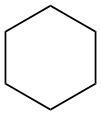
The choice of the numerical ring closure label is arbitrary, for example cyclohexane may just as well be represented as `C2CCCCC2` or `C0CCCCC0` (Table 4). For a second ring, the label will need to be different than the first label. For example, decalin (decahydronaphthalene) may be written as `C1CCCC2C1CCCC2`.

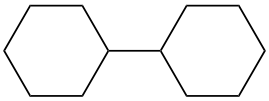
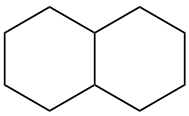
SMILES does not require that ring numbers be used in any particular order, and permits ring number zero, although this is rarely used. Also, it is permitted to re-use ring numbers after the first ring has closed, although this usually makes formulae harder to read. For example, bicyclohexyl is usually written as `C1CCCCC1C2CCCCC2`, but it may also be written as `C0CCCCC0C0CCCCC0`. Multiple digits after a single atom indicate multiple ring-closing bonds. For example, an alternative SMILES notation for decalin is `C1CCCC2CCCCC12`, where the final carbon participates in both ring-closing bonds 1 and 2.

Ring-closing digits may be preceded by a bond type. For example, cyclopropene is usually written `C1=CC1`, but if the double bond is chosen as the ring-closing bond, it may be written as `C=1CC1`, `C1CC=1`, or `C=1CC=1`.

Table 4. Different SMILES representations of rings

Name	Structure	Example SMILES

Cyclohexane		`C1CCCCC1`
		`C0CCCCC0`
		`C2CCCCC2`

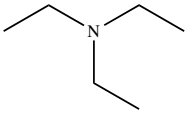
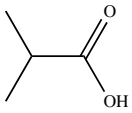
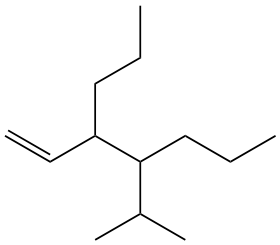
Bicyclohexyl		`C1CCCCC1C2CCCCC2` `C0CCCCC0C0CCCCC0`
Decalin		`C1CCCC2C1CCCC2` `C1CCCC2CCCCC12`

Branching

Branches are specified by enclosing them in parentheses, and can be nested or stacked. In all cases, the implicit connection to a parenthesized expression (a branch) is to the left. The first atom within the parentheses, and the first atom after the parenthesized group, are both bonded to the same branch point atom. Branches may be

written in any order. For example, bromochlorodifluoromethane may be written as `FC(Br)(Cl)F`, `BrC(F)(F)Cl`, `C(F)(Cl)(F)Br`, or the like. Examples of branches are given in Table 5.

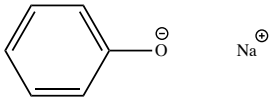
Table 5. Different SMILES representations of branches

Name	Structure	Example SMILES
Triethylamine		`CCN(CC)CC`
Isobutyric acid		`CC(C)C(=O)O`
3-Propyl-4-isopropyl-1-heptene		`C=CC(CCC)C(C(C)C)CCC`

Disconnected structures

Disconnected compounds are written as individual structures separated by a '**.**' (dot). The order in which the ions or ligands are listed is arbitrary. There is no implied pairing of one charge with another, nor is it necessary to have a net zero charge. If desired, the SMILES of one ion may be imbedded within another as shown in the example of sodium phenoxide (Table 6):

Table 6. Different SMILES representations sodium phenoxide illustrating disconnected structures.

Name	Structure	Example SMILES
Sodium phenoxide		`[Na+].[O-]c1ccccc1` `c1cc([O-].[Na+])ccc1`

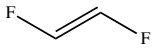
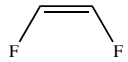
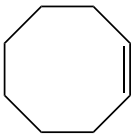
Stereochemistry

The SMILES representation permits, but does not require, specification of stereoisomers, which can be stereoisomerism around double bonds or stereoisomerism of asymmetric centres. The term *isomeric SMILES* collectively refers to SMILES written using these stereochemical rules.

Configuration around double bonds is specified using the characters '/' and '\ to show directional single bonds adjacent to a double bond. For example, `F/C=C/F` (see Table 7) is one representation of *trans*-1,2-difluoroethylene, in which the fluorine atoms are on opposite sides of the double bond, whereas `F/C=C\F` is one possible representation of *cis*-1,2-difluoroethylene, in which the F's are on the same side of the double bond.

Bond direction symbols always come in groups of at least two, of which the first is arbitrary. That is, `F\C=C\F` is the same as `F/C=C/F`. When alternating single-double bonds are present, the groups are larger than two, with the middle directional symbols being adjacent to two double bonds. For example, the common form of (2,4)-hexadiene is written `C/C=C/C=C/C` (Table 7).

Table 7. Different SMILES representations of double bond stereochemistry

Name	Structure	Example SMILES
Trans-1,2-difluoroethylene		`F/C=C/F` `F\C=C\F`
Cis-1,2-difluoroethylene		`F/C=C\F` `F/C=C\F`
(2*E*,4*E*)-hexa-2,4-diene		`C/C=C/C=C/C`
Cis-cyclooctene		`C1CCC/C=C\CC1`

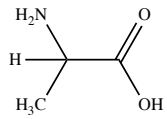
Configuration at tetrahedral carbon is specified by '`@`' or '`@@`'. SMILES uses a very general type of chirality specification based on local chirality. Instead of using a rule-based numbering scheme to order neighbour atoms of a chiral centre, orientations are based on the order in which neighbours occur in the SMILES string.

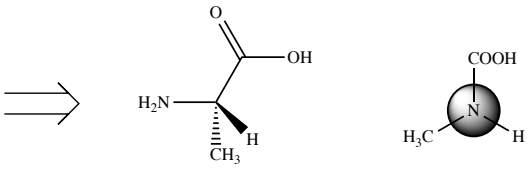
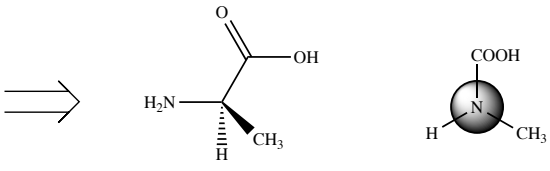
The simplest and most common kind of chirality is tetrahedral; four neighbour atoms are evenly arranged about a central atom, known as the chiral centre. Consider the four bonds in the order in which they appear, left to right, in the SMILES form. Looking toward the central carbon from the perspective of the first bond, the other three are either clockwise or counter-clockwise. These cases are indicated with '`@@`' and '`@`', respectively, as the `@` symbol itself is a counter-clockwise spiral. The symbol '`@`' indicates that the following neighbours are listed anticlockwise. '`@@`' indicates that the neighbours are listed clockwise.

Consider for example the amino acid alanine (Table 8). One of its SMILES forms is `NC(C)C(=O)O`, more fully written as `N[CH](C)C(=O)O`. This SMILES representation does not include stereochemical information. *L*-alanine however, the more common enantiomer, is written as `N[C@@H](C)C(=O)O`. Looking from the N-C bond, the hydrogen (`H`), methyl (`C`), and carboxylate (`C(=O)O`) groups appear clockwise. *D*-Alanine can be written as `N[C@H](C)C(=O)O`.

Table 8. Different SMILES representations of tetrahedral stereochemistry

Name	Structure	Example SMILES

Alanine		`NC(C)C(=O)O`
		`N[CH](C)C(=O)O`
		`N[CH](C)C(=O)[OH]`

L-Alanine		`N[C@@H](C)C(=O)O`
D-Alanine		`N[C@H](C)C(=O)O`

The SMILES specification includes elaborations on the `@` symbol to indicate stereochemistry around more complex chiral centres, such as trigonal bipyramidal molecular geometry. For a complete reference, please refer to the online manual at http://www.daylight.com/dayhtml/doc/theory/theory.smiles.html.

Canonical SMILES

Any given molecule can be represented by a wide variety of SMILES representations, all depicting the same molecule. For example, methane can be represented as `C`, `[H][C]([H])[H]`, or `[CH4]`. In order to overcome this 'random' behaviour of the SMILES representation, the concept of a canonical SMILES has been introduced. Canonicalization is a way to determine which of all possible SMILES will be used as the reference SMILES for a molecular graph. Suppose you want to find if a structure already exists in a data set. In graph theory this is the graph isomorphism problem. Using the canonical SMILES instead of the graphs reduces the problem to a simple text matching problem.

Hence canonicalization has some nice advantages but unfortunately there exists no universal canonical SMILES. Every toolkit or program uses a different algorithm, and sometimes the algorithm changes with different versions of the toolkit. There are even different forms of canonical SMILES, depending on if atomic properties like isotope are important for the result. Therefore, if one wants to compare molecules based on their canonical SMILES representations, care should be taken that all these canonical SMILES' have been generated by the same toolkit.

1.2. InChi

InChi stands for IUPAC International Chemical Identifier, and is a universal textual identifier or name for chemical compounds. It is designed to provide a unique and canonical way to encode the molecular topology into a text string, facilitating retrieval and storage of chemical information in molecular databases. The format and algorithms of the InChi format is freely available, ensuring thereby that there is only one standard. All information on the InChi standard can be found on http://www.iupac.org/home/publications/e-resources/inchi.html.

The InChi descriptor describes chemical substances in terms of layers of information: 1) the atoms and their bond connectivity, 2) tautomeric information, 3) isotope information, 4) stereochemistry, and 5) electronic charge information. Not all layers have to be provided; for instance, the tautomer or stereochemistry layers can be omitted if these types of information are not relevant or not defined. For example, the InChi description for ethanol is:

```
InChI=1S/C2H6O/c1-2-3/h3H,2H2,1H3
```

Every InChi starts with the string '`InChi=`', followed by the version number (which is currently `1`) and the letter '`S`' to denote standard InChi. The remaining information is structured as a sequence of layers and sublayers. The standard four layers are the 'main layer', the 'charge layer', the 'stereochemical layer' and the 'isotopic layer'.

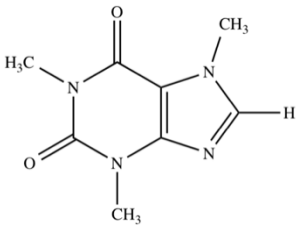
Main layer

The main layer provides information about the chemical formula (no prefix), the atom connections (prefix '`c/`'), and the hydrogen atoms (prefix '`h/`'):

- The *chemical formula* sublayer is the only layer that must occur in every InChi. Without this sublayer an InChi string is invalid.
- The *atom connection* sublayer is indicated by the 'c/' prefix. The atoms defined in the chemical formula sublayer are numbered in sequence (excluding the hydrogens), and this sublayer defines which atoms are connected to which other atoms.
- The *hydrogen atoms* sublayer is indicated by the 'h/' prefix, and describes how many hydrogen atoms are connected to each of the non-hydrogen atoms.

An example of the main layer is given above for ethanol. The chemical formula sublayer states that there are two carbon atoms, six hydrogens and one oxygen. From this chemical formula, the atom numbering for the non-hydrogen atoms can be extracted: C1, C2, O3. The atom connection sublayer (prefix '/c') of the ethanol InChi states that atom 1 is connected to atom 2, and atom 2 is connected to atom 3. Hence the molecular topology, excluding hydrogens so far, is C1-C2-O3. Finally, hydrogen information is defined in the hydrogen atoms sublayer (prefix '/h'), stating that O3 is connected to one hydrogen ('`3H`'), C2 connected to two hydrogens ('`2H2`'), and C1

connected to three hydrogens ('1H3'). Branches and ring closures are given in parentheses, as exemplified in Figure 3.



InChi=1S/C8H10N4O2/c1-10-4-9-6-5(10)7(13)12(3)8(14)11(6)2/h4H,1-3H3

Figure 3. InChi description of caffeine. Atom sequence is determined by a canonicalization algorithm to ensure that the atom sequence is not depending on how the structure was entered or drawn (for more information, see http://depth-first.com/articles/2006/08/12/inchi-canonicalization-algorithm/).

Charge layer

The charge layer consists of two sublayers:

- The *total charge* sublayer '/q' is the net charge of the core parent structure.
- The *protonation/deprotonation* sublayer '/p' indicates the net number of protons that need to be removed from (or added to) the structure when derived from its core parent. An example illustrating this is given in Figure 4:

InChI=1S/C6H13N/c7-6-4-2-1-3-5-6/h6H,1-5,7H2/p+1

InChI=1S/C9H20N/c1-10(2,3)9-7-5-4-6-8-9/h9H,4-8H2,1-3H3/q+1

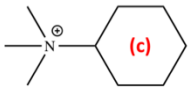


Figure 4. InChi notations of two charged compounds. The first compound (a) is the protonated form of the neutral core (b). For this reason, the InChi string of (a) is not containing a total charge sublayer '/q', since the total charge of its core structure (b) equals zero. However, the InChi of compound (a) contains a protonation/deprotonation sublayer '/p' to indicate that the compound contains an extra proton compared to its core structure (b). It is up to the processing software to determine where this proton should be positioned, as this is not defined in the hydrogen atoms sublayer '/h' (which indicates the presence of a single hydrogen on atom 6, two hydrogens on atoms 1-5 and 7, and two hydrogens on atom 7, the latter being the N). Compound (c) contains a formal charge which cannot be removed by deprotonation; hence the InChi of this compound contains a total charge sublayer '/q' defining a total charge of +1, and the InChi is lacking a protonation/deprotonation sublayer '/p'.

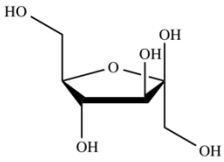
Other layers

The two main remaining layers include the stereochemical layer and the isotopic layer. The stereochemical layer contains sublayers representing double bond sp^2 stereochemistry ('/b') and tetrahedral sp^3 stereochemistry ('/t'). The isotopic layer (signified with the prefix '/i') identifies different isotopically labelled atoms. Exchangeable isotopic hydrogen atoms (deuterium and tritium) are listed separately.

1.3. InChiKey

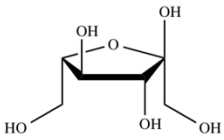
The length of an InChi string depends on the number of atoms in the molecule. Due to this variable length, InChi strings are not always the preferred manner to be used as a key in database queries. In order to tackle this shortcoming, the InChiKey was introduced in 2007. An InChiKey is a fixed-length (27 characters) condensed digital representation. The first part is 14 characters long and encodes the molecular skeleton (connectivity). After a hyphen, there is a second string of 10 characters, the first eight of which encode stereochemistry and isotopes, and after that, 'S' indicates that the key was produced from standard InChI and 'A' indicates that version 1 of InChI was used. The final character, 'N', means 'neutral' (Figure 5). Both parts of the InChIKey are based on a truncated SHA-256 hash of the corresponding InChI layers. For encoding of the data, only uppercase ASCII letters are used

which ensures that the indexing engines will not split the data and also avoids case-sensitivity problems. There is a finite, but extremely small probability of finding two structures with the same InChIKey.



D-Fructose

InChI=1S/C6H12O6/c7-1-3-4(9)5(10)6(11,2-8)12-3/h3-5,7-11H,1-2H2/t3-,4-,5+,6-/m1/s1
RFSUNEUAIZKAJO-ARQDHWQXSA-N



L-Fructose

InChI=1S/C6H12O6/c7-1-3-4(9)5(10)6(11,2-8)12-3/h3-5,7-11H,1-2H2/t3-,4-,5+,6+/m0/s1
RFSUNEUAIZKAJO-UNTFVMJOSA-N

Figure 5. Illustration of the InChiKey concept. Shown are D- and L-fructose with their corresponding InChi and InChiKey representations. Indicated in red are the differences between InChi and InChiKey of the two molecules. Since D- and L-fructose are stereoisomers of each other's, differences are only present in the stereochemistry sublayer of the InChi string. Use of InChiKey allows searches based solely on atom connectivity (the first 14 characters), as the stereoisomers D-fructose and L-fructose both have the same first block of 14 characters, RFSUNEUAIZKAJO.

1.4. MOL-block

The MOL-file (sometimes also called MOL-block) is a chemical file format capable of representing a single chemical structure and its associated data fields. The MOL format was developed by Molecular Design Limited (MDL). The molfile consists of a header and a connection table (CT). Below is a sample chemical record in MOL format of *L*-alanine, of which the corresponding structure is shown in **Error! Reference source not found.** below:

```
L-Alanine
ChemDraw
This is a comment line
  6  5  0  0  1  0            999 V2000
   -0.6622    0.5342    0.0000 C   0  0     0  0  0  0
    0.6222   -0.3000    0.0000 C   0  0     0  0  0  0
   -0.7207    2.0817    0.0000 C   1  0     0  0  0  0
   -1.8622   -0.3695    0.0000 N   0  3     0  0  0  0
    0.6220   -1.8037    0.0000 O   0  0     0  0  0  0
    1.9464    0.4244    0.0000 O   0  5     0  0  0  0
  1  2  1  0  0  0
  1  3  1  1  0  0
  1  4  1  0  0  0
  2  5  2  0  0  0
  2  6  1  0  0  0
M  CHG  2   4   1   6  -1
M  ISO  1   3  13
M  END
```

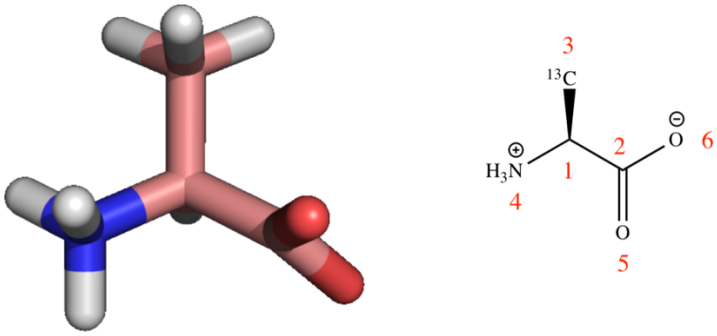


Figure 6. Structure of L-alanine of which the corresponding MOL representation is given above. This structure also shows the hydrogen atoms for clarity, but these hydrogen atoms have been omitted in the MOL-file above (in this example, only the non-H atoms are stored in the MOL-file).

There are currently two main versions that differ considerably, namely V2000 and V3000. V2000 is the oldest, yet still widely used. For that reason, we will only consider V2000 (the V3000 format is available online).

The V2000 version of a MOL-block is comprised of three parts: the header block, the connection table and the data items.

Header block

The first part of the SDF file consist of a three-line **header block** (the first three black lines in the example above). These three lines **may** contain:

- Name of the molecule (`L-alanine`)
- Details of the software used to generate the structure (`ChemDraw`)
- Comment (`Comment`)

The three lines in the header block may be empty; however, in that specific case there should be three empty lines (the software that reads MOL-files counts on the fact that there are three lines preceding the connection table, see below).

Connection table

The second part consists of the **connection table** (CT) which is composed of a *counts line* (highlighted in red in the example above), an *atom block* (blue), a *bond blocks* (green), and followed by a *properties block* (black).

1) The **counts line** is made up of twelve fixed-length fields - the first eleven are three characters long, and the last field is six characters long:

```
aaabbblllfffcccsssxxxrrrpppiiimmmvvvvvv
```

in which:

`aaa`	= number of atoms
`bbb`	= number of bonds
`lll`	= number of atoms lists (not used)
`fff`	= not used anymore
`ccc`	= chiral flag (0 = not chiral; 1 = chiral)
`sss`	= number of stext entries (not used)
`xxx, rrr, ppp, iii`	= not used anymore
`mmm`	= not used anymore, default set to 999
`vvvvvv`	= version (should be `V2000` in V2000 format)

The first two fields are the most critical, and specifiy the number of atoms and bonds of the compound. In the example given above, the *L*-alanine compound consists of 6 atoms and 5 bonds. Actually, the compound consists of more than 6 atoms and 5 bonds (13 atoms and 12 bonds), but the given numbers in the file denote the actual number of atoms and bonds presented in the SDF file (implicit hydrogen atoms are not provided in this example). *L*-Alanine is a chiral molecule, so the chiral flag is set to *on*. `V2000` in the counts line denotes the SDF version, with V2000 being the most widely used one and V3000 (the extended connection table) being a more recent version.

2) The **atom block** is made up of atom lines, one line per atom, with the following format:

```
xxxxxxxxxxyyyyyyyyyyzzzzzzzzzz·aaaddcccssshhhbbbvvv
```

in which:

`xxxxxxxxxx`	= *x*-coordinate of atom	(10 characters)
`yyyyyyyyyy`	= *y*-coordinate of atom	(10 characters)
`zzzzzzzzzz`	= *z*-coordinate of atom	(10 characters)
·	= space	(1 character)
`aaa`	= atom symbol	(3 characters)
`dd`	= mass difference from the value in the periodic table (default = 0)	(2 characters)
`ccc`	= charge (0 = 0, 1 = +3, 2 = +2, 3 = +1, 5 = -1, 6 = -2, 7 = -3)	(3 characters)
`sss`	= atom stereo parity (ignored when read)	(3 characters)
`hhh`	= ignored	(3 characters)
`bbb`	= ignored	(3 characters)
`vvv`	= valence (default = 0)	(3 characters)

Of these, the atom coordinates, atom symbol, mass difference and charge fields are the most important ones; the other fields are often ignored.

3) The **bond block** is made up of bond lines, one line per bond, with the following format:

```
111222tttsss
```

in which:

`111`	= first atom number (counting starts from 1)	(3 characters)
`222`	= second atom number	(3 characters)
`ttt`	= bond type (1 = S; 2 = D; 3 = T; 4 = A; 5 = S or D; 6 = S or A; 7 = D or A; 8 = any)	(3 characters)
`sss`	= bond stereo (default 0)	(3 characters)

The bond type symbols denote the following: S is a single bond, D is a double bond, T is a triple bond, A is an aromatic bond.

4) The **properties block** is made up of a number of additional properties, and is read line-per-line until an `M  END` line is encountered. Each line is the property block is identified by a prefix in the form of `M  XXX` with two spaces separating the `M` and `XXX`. The prefix `M  END` terminates the properties block. The most important properties are:

- `M··CHGnn8·aaa·vvv …`

  Denotes atomic formal **charges** [with `nn8` denoting the number of entries (1-8), `aaa` the atom number, and `vvv` the charge]. When present, this property supersedes all charge values in the atom block, forcing a zero charge on all atoms not listed in an `M  CHG` line.

- `M··RADnn8·aaa·vvv …`

  Denotes atomic **radicals** [with `nn8` denoting the number of entries (1-8), `aaa` the atom number, and `vvv` the radical property (0 = no radical; 1 = singlet; 2 = doublet; 3 = triplet)]. When present, this property supersedes all charge values in the atom block, forcing a zero charge on all atoms not listed in an `M  RAD` line.

- `M··ISOnn8·aaa·vvv …`

  Denote atomic isotopes [with `nn8` denoting the number of entries (1-8), `aaa` the atom number, and `vvv` the absolute mass of the atom isotope as a positive integer]. When present, this property supersedes all isotope values in the atom block. Default (no entry) means natural abundance.

- `M··END`

  Should be the last line in a MOL-file.

1.5. SDF

The Structure Data Format (SDF) is a chemical file format to represent multiple chemical structure records and associated data fields. It wraps the MOL-block format and extends this with a chemical data block. SDF has been developed by Molecular Design Limited (MDL) and has become the most widely used standard for importing and exporting information on chemicals. Below is a sample chemical record in SDF format of L-alanine, of which the structure is shown in Figure 6 above:

```
L-Alanine
ChemDraw
Comment
··6··5··0··0··1··0·············999·V2000
····-0.6622····0.5342····0.0000·C···0··0·····0··0··0··0
····0.6222···-0.3000····0.0000·C···0··0·····0··0··0··0
····-0.7207····2.0817····0.0000·C···1··0·····0··0··0··0
····-1.8622···-0.3695····0.0000·N···0··3·····0··0··0··0
····0.6220···-1.8037····0.0000·O···0··0·····0··0··0··0
····1.9464····0.4244····0.0000·O···0··5·····0··0··0··0
··1··2··1··0··0··0
··1··3··1··1··0··0
··1··4··1··0··0··0
··2··5··2··0··0··0
··2··6··1··0··0··0
M··CHG··2···4···1···6··-1
M··ISO··1···3··13
M··END
>··<Sample Ref.>
OC101-12

>··<Melting Point>
41.00 - 43.00

>··<B1 Record No.>
304

>··<ID>
304

$$$$
```

The SDF format is comprised of three parts: the header block, the connection table and the data items. The header block and connection table together form the MOL-block as described in the section before. The SDF-format to describe a single molecule consists therefore of the combination of a MOL-block followed by a number of data items (Figure 7):

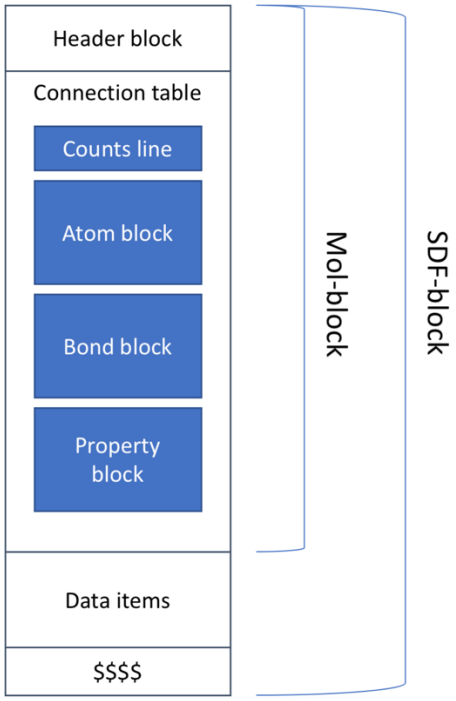


Figure 7. The relation between the different formats and blocks. The connection table consists of a counts line, and an atom, bond and property block (the MOL-block). The SDF-block consists of a MOL-blocks amended with data items and a closing $$$$ line.

Data items

The SDF format can include associated data items (the orange lines in the previous example of *L*-alanine). These data fields start with a header, which begins with a > character. On the same line, the name of the data field is written in angular brackets. After the header, a data field contains one or more lines of up to 200 characters of free text, which is the value of the data field. Each data field ends with a blank line.

It should be noted that the data items section is not obligatory. In cases when there are no data items to include, then this section may be ommitted.

Multiple molecule entries

An interesting property of the SDF-format is that it can contain multiple molecule entries. Each molecule is represented as a separate SDF-block. The end of each molecule entry is denoted with a $$$$ line.

1.6. PDB

The Protein Data Bank (PDB) format provides a standard representation for macromolecular structure data derived from X-ray diffraction and NMR studies. It is mainly focussed on the representation of protein structures but it may be used for other ligands as well. This representation was created in the 1970's and a large amount of software using it has been written in the meantime. The format is quite extensive and elaborated but the most important aspect is highlighted here.

PDB format consists of lines of information in a text file. Each line of information in the file is called a record. A PDB file generally contains several different types of records, arranged in a specific order to describe a structure. The PDB format of *L*-alanine (**Error! Reference source not found.**) is given here:

```
HETATM····1··N···LIG·············-0.944···-0.430···-0.803··0.00··0.00···········N1+
HETATM····2··C···LIG·············0.362···0.206···-0.576··0.00··0.00···········C
HETATM····3··C···LIG·············0.357···1.563···-1.160··0.00··0.00···········C
HETATM····4··C···LIG·············1.494···-0.664···-1.172··0.00··0.00···········C
HETATM····5··O···LIG·············0.534···2.579···-0.450··0.00··0.00···········O
HETATM····6··O···LIG·············0.159···1.741···-2.486··0.00··0.00···········O1-
CONECT····1····2
CONECT····2····1····3····4
CONECT····3····2
CONECT····3····5
CONECT····3····6
CONECT····4····2
CONECT····5····3
CONECT····6····3
END
```

In general, the most important record types are:

- ATOM: atomic coordinate record containing the X, Y, Z orthogonal Å coordinates for atoms in standard residues (amino acids and nucleic acids).
- HETATM: atomic coordinate record containing the X, Y ,Z orthogonal Å coordinates for atoms in nonstandard residues. Nonstandard residues include inhibitors, cofactors, ions, and solvent. The only functional difference from ATOM records is that HETATM residues are by default not connected to other residues.
- CONECT: specify connectivity between atoms for which coordinates are supplied. The connectivity is described using the atom serial number as shown in the entry. CONECT records are not mandatory for standard protein residues.
- END: marks the end of the PDB file.
- TER: indicates the end of a chain of residues. For example, a haemoglobin molecule consists of four subunit chains that are not connected. TER indicates the end of a chain and prevents the display of a connection to the next chain.

The **ATOM** record is made up, each atom on a separate line, of the following format (not all fields have to be supplied):

```
ATOM··nnnnn·mmmmArrr·cssss····xxxxxxxxyyyyyyyyzzzzzzzzoooooottttt··········ee
```

in which:

ATOM	= required field	(4 characters followed by two spaces)
nnnnn	= atom serial number	(5 characters)
··	= 2 spaces	(2 characters)
mmmm	= atom name	(4 characters)
·	= 1 space	(1 character)
A	= alternate location indicator	(1 character)
rrr	= residue name	(3 characters)
·	= 1 space	(1 character)
c	= chain identifier	(1 character)
ssss	= residue sequence number	(4 characters)
····	= 4 spaces	(4 characters)
xxxxxxxx	= X orthogonal Å coordinate	(8 characters)
yyyyyyyy	= Y orthogonal Å coordinate	(8 characters)

zzzzzzzz	= Z orthogonal Å coordinate	(8 characters)
oooooo	= occupancy	(6 characters)
tttttt	= temperature factor	(6 characters)
·····	= 10 spaces	(10 characters)
ee	= element symbol	(2characters)

The HETATM is identical as the ATOM record, with the exception that the first 6 characters of the ATOM record are replaced with HETATM.

The CONECT record obeys the following format:

```
CONECTnnnnnmmmmmmooooooppppppvvvvvv
```

in which (all six characters wide):

CONECT	= required field
nnnnnn	= atom serial number
mmmmmm	= serial number of bonded atom
oooooo	= serial number of bonded atom
pppppp	= serial number of bonded atom
vvvvvv	= serial number of bonded atom

These CONECT records occur in increasing order of the atom serial numbers they carry in columns 7-11. The target-atom serial numbers carried on these records also occur in increasing order. The connectivity list given by the CONECT records is redundant in that each bond indicated is given twice, once with each of the two atoms involved specified in columns 7-11.

The END record marks the end of the PDB file. Most software packages do not require an END record.

The TER record marks the end of a protein or nucleic acid chain. The TER records occur in the coordinate section of the entry, and indicate the last residue presented for each polypeptide and/or nucleic acid chain for which there are determined coordinates. For proteins, the residue defined on the TER record is the carboxy-terminal residue; for nucleic acids it is the 3'-terminal residue. According the official specifications, the format of a TER record is as follows:

```
TER···nnnnn·····rrr·ciiiix
```

in which:

TER	= required field	(3 characters)
···	= 3 spaces	(3 characters)
nnnnnn	= serial number	(6 characters)
·····	= 6 spaces	(6 characters)
rrr	= residue name	(3 characters)
·	= 1 space	(1 character)
c	= chain identifier	(1 character)
iiii	= residue sequence number	(4 characters)
x	= insertion code	(1 character)

However, in practice often only the TER keyword field is specified without any of the other fields.

The PDB format specifies many more keywords, many of these being specific for describing the primary and secondary structures of the polypeptide chains, information about the crystallographic setup and experimental details, and another set of keywords to specify the authors of the PDB file.

2. Molecular graphics

With molecular graphics, molecules and their properties are represented on graphical display devices such as a computer screen. Ever since Dalton's atoms and Kekulé's benzene, there has been a rich history of hand-drawn atoms and molecules, and these representations have had an important influence on modern molecular graphics. Many molecular graphics programs and systems exist and the majority of those graphics systems include editing commands or calculations for use in molecular modelling.

2.1. Visualisation software

There are many molecular graphics programs available, some of these being commercial while other programs are open source and/or free of charge for academic purposes. It is impossible to list all of these tools; hence we will limit ourselves to those open source tools that are most commonly used in our experience.

PyMol

PyMol (https://pymol.org/2/) is a free and open source molecular graphics system for visualization, animation, editing, and publication-quality imagery. **PyMOL** is scriptable and can be extended using the Python language. Supports Windows, Mac OSX, Unix, and Linux (Figure 8).

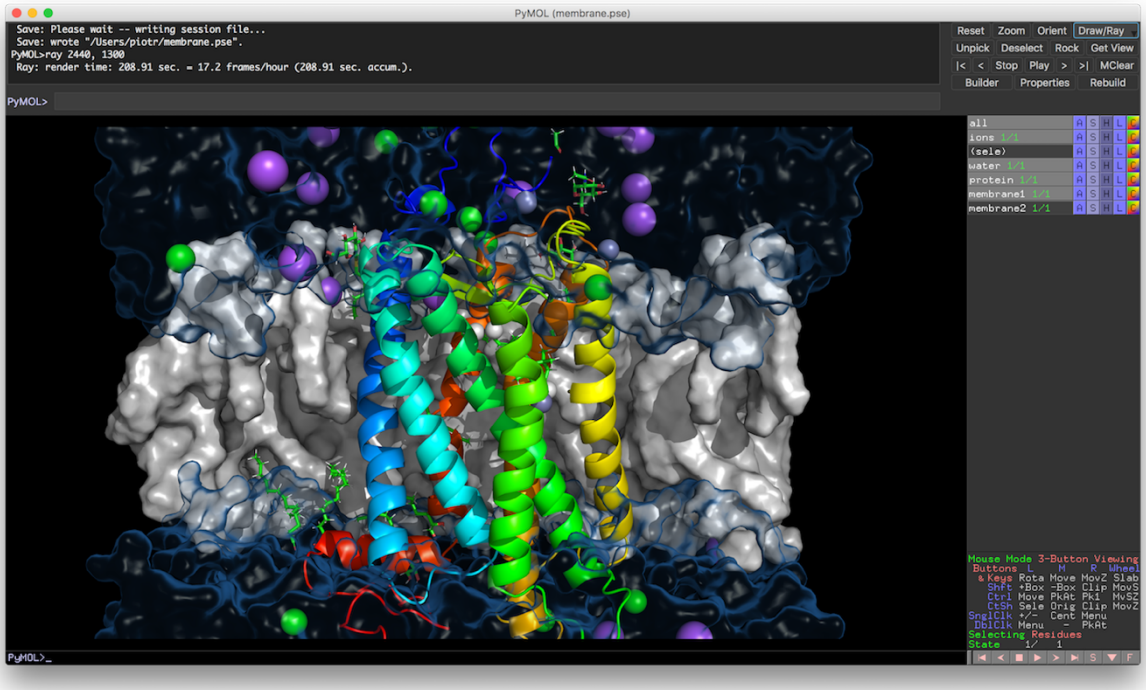


Figure 8. Screenshot of a PyMol session showing a cross-membrane protein represented as ribbons. The membrane part is colored gray, while the protein is colored yellow/green/blue.

PyMol can be extended with many add-ons and public-available scripts, downloadable from the PyMolWiki page (https://pymolwiki.org/index.php/Main_Page). This forum is a powerful resource for questions and answers, and illustrates the power of open source software that is supported by a broad user community.

For advanced visualization problems, **PyMol** also supports hardware stereographics using active shutters systems. Stereoscopy (also called stereoscopics, or stereo imaging) is a technique for creating or enhancing the illusion of depth in an image by means of stereopsis for binocular vision]. Most stereoscopic methods present two offset images separately to the left and right eye of the viewer. These two-dimensional images are then combined in the

brain to give the perception of 3D depth. An active shutter 3D system is a technique of displaying stereoscopic 3D images. It works by only presenting the image intended for the left eye while blocking the right eye's view, then presenting the right-eye image while blocking the left eye, and repeating this so rapidly that the interruptions do not interfere with the perceived fusion of the two images into a single 3D image. Modern active shutter 3D systems generally use liquid crystal shutter glasses (also called active shutter glasses). Each eye's glass contains a liquid crystal layer which has the property of becoming opaque when voltage is applied, being otherwise transparent. The glasses are controlled by a timing signal that allows the glasses to alternately block one eye, and then the other, in synchronization with the refresh rate of the screen. The timing synchronization to the video equipment may be achieved via a wired signal, or wirelessly by either an infrared or radio frequency transmitter. A well-known system for active shutter 3D visualization is the **3D Vision kit** of NVIDIA. It comes with 3D shutter glasses, a transmitter, and special graphics driver software. While regular LCD monitors run at 60 Hz, a 120 Hz monitor is required to use 3D Vision (Figure 9). A special 3D-enable graphics card is also required.

Figure 9. The 3D Vision kit from NVIDIA to display hardware-enable active stereo. A special 120 Hz monitor, 3D shutter glasses and a transmitter are needed.

VMD

VMD is a molecular visualization program for displaying, animating, and analysing large biomolecular systems using 3-D graphics and built-in scripting. It is an open-source software package that can be downloaded free of charge from http://www.ks.uiuc.edu/Development/Download/download.cgi?PackageName=VMD for Mac OS X, Unix or Windows. It is especially suited for advanced visualisation of molecular dynamics trajectories and can be fine-tuned by means of Tcl-Tk-based scripting (Figure 10).

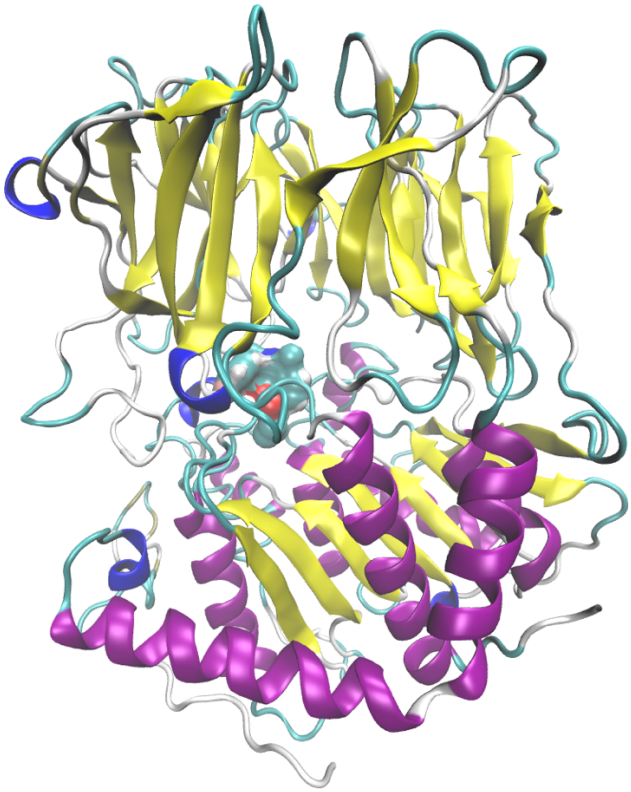


Figure 10. Structure of a propyl endopeptidase protein rendered by the VMD program. The protein peptide backbone is rendered as a ribbon-diagram (see below) and colored by secondary structure (helices in purple, β-sheets in yellow). The GSK-522 ligand is shown in the center of the cavity.

2.2. Molecular representations

CPK

CPK is a three-dimensional space-filling representation in which atoms are represented by spheres. The name CPK refers to the scientists Corey, Pauling and Koltun, who developed this representation concept into a useful form. The radius of each atoms sphere is normally correlated to its atomic number, but sometimes the spheres are drawn with a smaller radius and the bonds are drawn in a cylinder-type of representation (Figure 11).

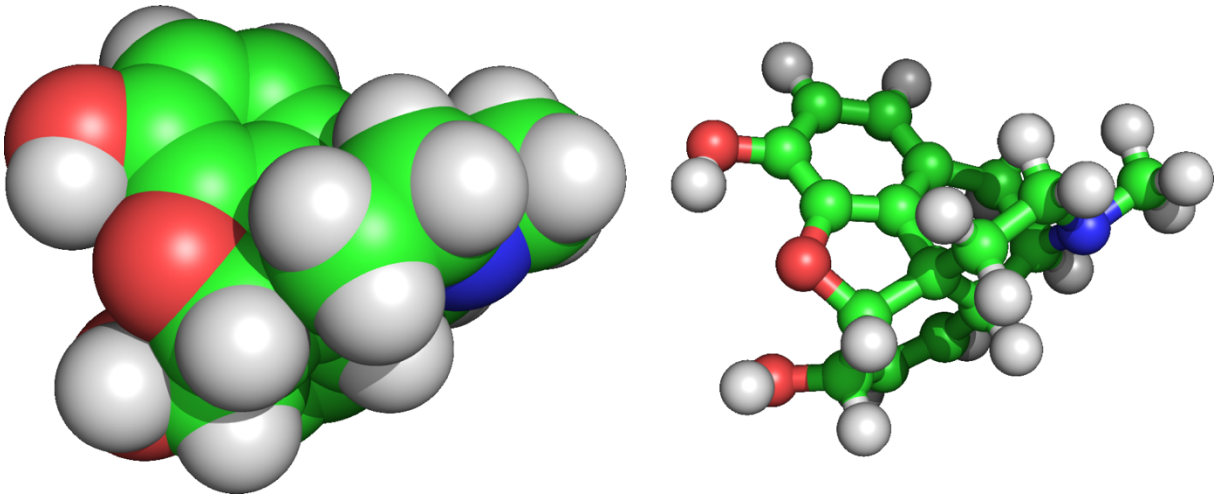


Figure 11. CPK representations of morphine. Left picture is a typical CPK representation in which the atoms are shown as spheres. The right picture shows a CPK model in which the atom radii are reduced and in which the bonds are drawn as cylinders.

Stick representation

In the CPK representation, focus is on the atoms and the corresponding volume the molecule occupies. In a stick representation, focus is rather on the connectivity between the atoms by representing the bonds as cylinders (Figure 12). For computational drug design, this representation is probably the most useful as it permits the researcher to investigate potential interactions between ligand and protein.

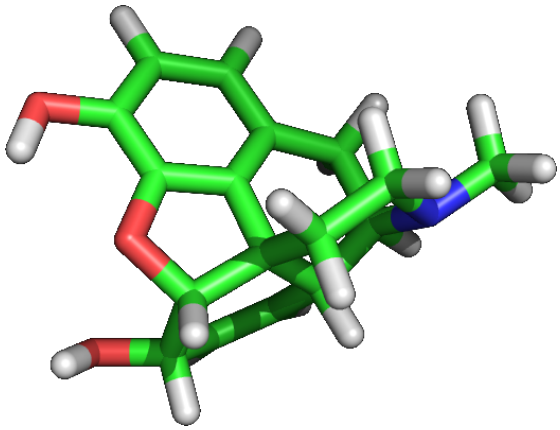


Figure 12. Stick representation of morphine. Only the bonds between atoms are shown, and colored according the participating atom types (C is green, O is red, H white and N blue).

Ribbon diagram

Ribbon diagrams, also known as Richardson diagrams, are 3D schematic representations of protein structure and are one of the most common methods of protein depiction used today. The ribbon shows the overall path and organization of the protein backbone in 3D, and are generated by interpolating a smooth curve through the polypeptide backbone. α-helices are shown as coiled ribbons or thick tubes, β-strands as arrows, and lines or thin tubes for non-repetitive coils or loops. The direction of the polypeptide chain is shown locally by the arrows, and may be indicated overall by a colour ramp along the length of the ribbon.

Figure 13. PyMol ribbon of the structure of the mouse brain tubby protein, which has a characteristic β-barrel fold with a central α-helix (PDB: 1C8Z). β-sheets are shown in yellow, and α-helices in red.

Ribbon diagrams are simple, yet powerful, in expressing the visual basics of a molecular structure (twist, fold and unfold). This method has successfully portrayed the overall organization of the protein structure, reflecting its 3-

dimensional information, and allowing for better understanding of a complex object both by the expert structural biologists and also by other scientists, students, and the general public.

Solvent-accessible surface

The solvent-accessible surface area (SASA) if the surface area of a molecule that is accessible to a solvent, in most cases water. The SASA was first described by Lee & Richards in 1971.**2** The SASA is typically calculated using the 'rolling ball' algorithm developed by Shrake & Rupley in 1973.**3** This algorithm uses a sphere (of solvent) of a particular radius to 'probe' the surface of the molecule. In the case of water as solvent, the radius that is most commonly used is 1.4 Å (Figure 14 and Figure 15).

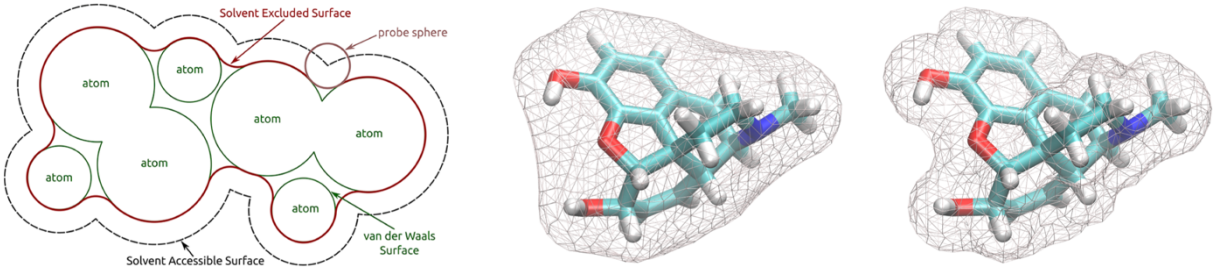


Figure 14. Left: illustration of the principle of the SASA, showing the relationship between probe radius, atom radii. Solvent-excluded surface and van der Waals surface. The solvent-accessible surface (black dotted line) is defined by the area that the center of probe sphere is traversing, while the solvent-excluded area (SES, red line) corresponds to the part of the van der Waals surface that can be touched by the probe, plus the reentrant surface. Middle and right: example of the solvent-excluded surface calculated using two different probe radii (8 Å for the left figure and 1 Å for the right figure).

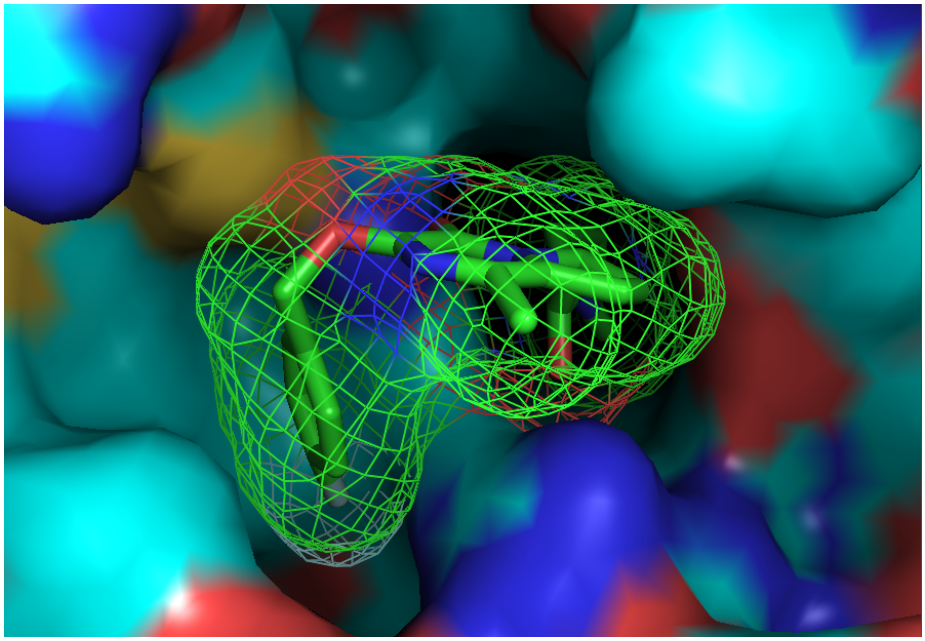


Figure 15. Illustrating the use of the SASA in analyzing shape complementary between ligand (wireframe SASA) and its surrounding receptor (solid surface).

Many programs are available to calculate SASA's, including the PyMol software. The SASA for a typical druglike molecule as shown in Figure 15 is around 300-500 $Å^2$, while for a protein like prolyl endopeptidase this area becomes 75,000 $Å^2$.

Chapter 3. Chemical informatics with RDKit

RDKit is an open-source programming API (Application Programming Interface) that allows users to write programs (called scripts) for the manipulation of small molecules. In this section, you will learn how to install RDKit on your computer, and how you can start working with molecules. In later sections, we will apply RDKit to illustrate the concepts of clustering, molecular similarity, and concepts of machine learning.

Since RDKit is open-source, this means that many modelling scientist are using this tool, and many more scientists are also contributing to improve the code behind it. The original link to RDKit is https://www.rdkit.org, and all code can be downloaded from the RDKit github repository: https://github.com/rdkit. You can download the RDKit software from this repository, but there are much easier methods to install the software on your computer. This is described in the following section. In that section, you will also learn how to make use of the Jupyter notebook for easy scripting and testing of our code.

1. Installing RDKit

There are many ways to install RDKit on your computer, but by far the easiest ways to install or use RDKit is 1) to use Anaconda for this purpose, or 2) to use Google Colab.

1.1. Installing with Anaconda

Anaconda is an environment and toolkit that equips you with tools to work with many open source packages and libraries. Anaconda is a distribution of the Python and R programming languages for scientific computing, that aims to simplify package management and deployment. The distribution includes data-science packages suitable for Windows, Linux, and macOS.

Installation of Anaconda is described in section 2.1. Once you have installed Anaconda on your computer, you can start using it and install RDKit. For this, open a terminal, and enter the following command to make sure that Anaconda has installed correctly:

```
$ conda --version
conda 4.8.5
```

If Anaconda installed correctly the version should be displayed; in this example it is version 4.8.5 but it might be different in your case.

It is not obligatory but advisable to create a separate conda environment in which RDKit will be installed. This environment will be using Python 3. For this example we will give this environment the name ‘rdkit’, but other names are also possible:

```
$ conda create --name rdkit python=3
```

Once created, you should activate this environment:

```
$ conda activate rdkit
```

Now install RDKit:

```
$ conda install -c conda-forge rdkit
```

To test the installation, start a Python session and use RDKit from there:

```
$ python
Python 3.9.1 (default, Dec 11 2020, 06:28:49)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from rdkit import Chem
>>> mol = Chem.MolFromSmiles("Cc1ccccc1")
>>> mol
<rdkit.Chem.rdchem.Mol object at 0x7fd50862da60>
```

```
>>> mol.GetNumAtoms()
7
>>> exit()
```

A useful extension to this conda installation is the installation of a Jupyter notebook. A Jupyter notebook is a Python environment that runs in your web browser and that provides you interactive sessions. It is an ideal environment to play around with the Python and RDKit modules, as it allows to test your code interactively and it also allows you to visualize images along your code. To install, just follow the following easy steps.

First make sure that you have activated your rdkit environment (done in the previous steps), so that the Jupyter notebook will be installed in the same conda environment as RDKit:

```
$ conda activate rdkit
```

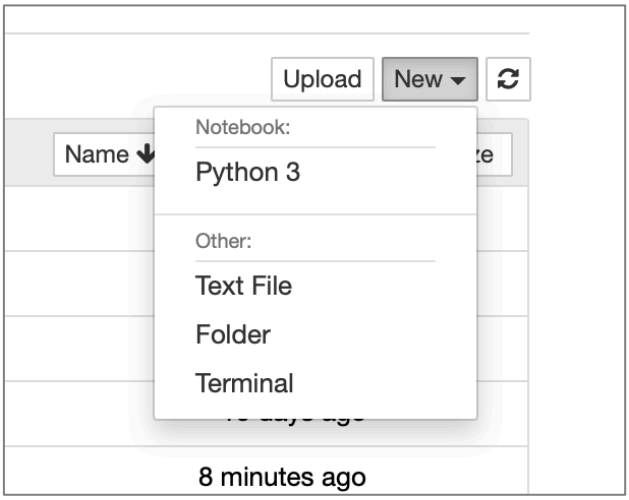
Now install Jupyter:

```
$ conda install jupyter
```

and create your first notebook:

```
$ jupyter notebook
```

This will open your web browser and you can now start a new Python 3 session:



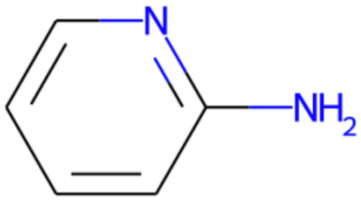
Now you can try your first notebook session. Enter the following:

```
>>> from IPython.display import SVG
>>> from rdkit import Chem
>>> mol = Chem.MolFromSmiles("c1cccnc1N")
>>> mol
```

Press the Run button and you should see a depiction of the molecule:

```
In [6]:
1 from IPython.display import SVG
2 from rdkit import Chem
3 mol = Chem.MolFromSmiles('c1cccnc1N')
4 mol
```

Out[6]:

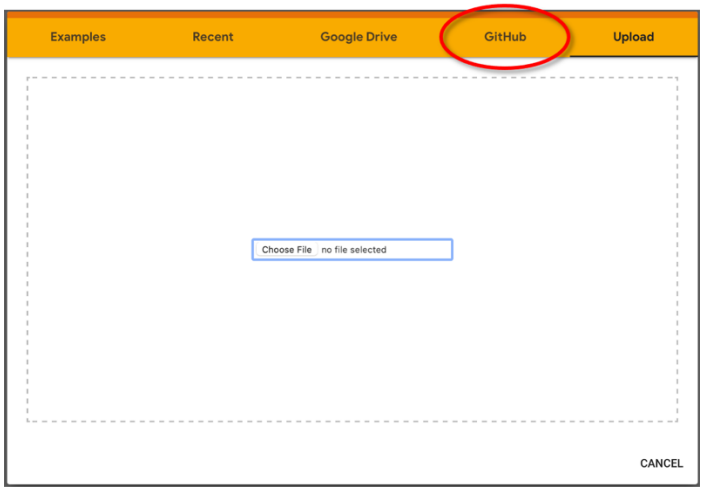


1.2. Installing in Google Colab

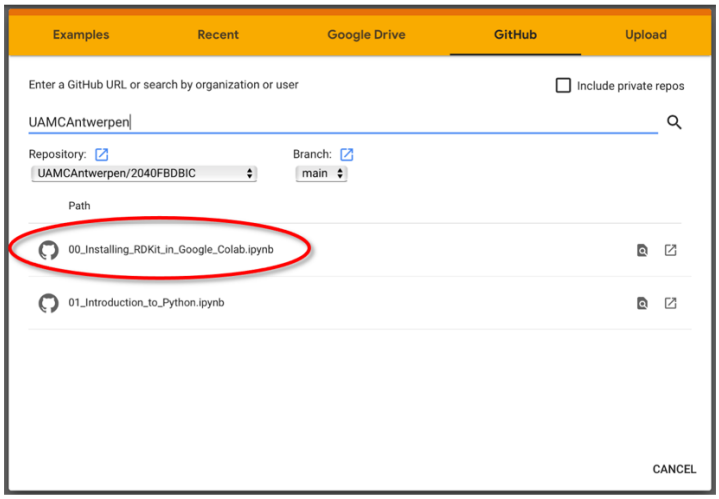
Google Colab can be considered as an online Jupyter notebook, running on the servers of Google and using your Google drive as a harddisk in the cloud. It has the advantage that one does not need to install Anaconda and Jupyter locally, making it easy to work with RDKit if you are not allowed to install software on your local machine. However, the disadvantage of Google Colab is that RDKit is not installed on Google Colab by default. This is something that you will have to take care off each time you will use Google Colab to run RDKit, and installaton of this might take a couple of minutes each time when starting a new workbook. In the following paragraphs we will explain how to do this.

How to start with Google Colab is explained in section 2.2. Please follow these steps and then you are ready to start with the installation of RDKit. This requires some code typing, but as all this code has been saved at the GitHub page of UAMCAntwerpen (the GitHub repository that contains many of the codes for this course), you don't need to type it.

First start a new workbook within Google Colab (`File > New notebook`). This brings you an empy notebook in which you can start typing Python code. However, we need to install RDKit first. For this, select `File > Upload notebook`, and select the GitHub tab from the menu:



Next, enter `UAMCAntwerpen` as the GitHub organization, select `UAMCAntwerpen/2040FBDBIC` as reporsitory (`main` branch), and select `00_Installing_RDKit_in_Google_Colab.ipynb` as the file to download:



This will download the RDKit installation code into your newly created notebook,[1] and now you need to press the arrows to run all the code cells and install RDKit (might take some minutes). Test the installation by creating a new code cell and entering:

```
>>> mol = Chem.MolFromSmiles("C")
>>> mol
```

This should visualize a depiction of a methane molecule.

2. Your first RDKit lines

After installing RDKit either in an Anaconda environment or on Google Colab, you can now start working with it. RDKit contains a number of modules, of which the `Chem` module being the core module. This one is required for almost all RDKit manipulations:

```
>>> from rdkit import Chem
```

Molecules are stored within RDKit as special `molecule` objects which can be created and modified in a wide variety of ways. An easy way of creating a new molecule is by converting a SMILES representation into a molecule object using the `MolFromSmiles()` method:

```
>>> mol = Chem.MolFromSmiles("CCCC")
>>> print(mol)
<rdkit.Chem.rdchem.Mol object at 0x7fb66742ea60>
```

With RDKit, one can count the number of atoms and bonds very quickly:

```
>>> print("number of atoms:", mol.GetNumAtoms())
number of atoms: 4
>>> nBonds = mol.GetNumBonds()
>>> print("number of bonds:", nBonds)
number of bonds: 3
```

Converting a molecule object back to its SMILES representation is done with the `MolToSmiles()` method:

```
>>> print(Chem.MolToSmiles(mol))
CCCC
```

Molecules can be valid (a correct processed molecule) or invalid (when something went wrong with processing, such as an invalid SMILES:

```
>>> for smiles in ["CCCC", "c"]:
...     print("Smiles:", smiles)
```

[1] In fact, the only code needed to install RDKit in Google Colab is `!pip install rdkit-pypi`

```
...     mol = Chem.MolFromSmiles(smiles)
...     print(mol)
...     print(mol is None)
...     print("")
...
Smiles: CCCC
<rdkit.Chem.rdchem.Mol object at 0x7fb66742eac0>
False

Smiles: c
[11:29:02] non-ring atom 0 marked aromatic
None
True
```

Hydrogen atoms are by default handled as being implicitly there, meaning that hydrogen atoms are not part of the general topology but can be generated upon request. When these hydrogen atoms have been generated, they are called being 'explicit'. The AddHs() and RemoveHs() functions within the AllChem module can be used to add (make explicit) or remove (convert back to implicit) hydrogens:

```
>>> from rdkit.Chem import AllChem
>>> mol = Chem.MolFromSmiles("C")
>>> print("number of atoms:", mol.GetNumAtoms())
number of atoms: 1
>>> print("number of bonds:", mol.GetNumBonds())
number of bonds: 0
>>> mol = AllChem.AddHs(mol)
>>> print("number of atoms:", mol.GetNumAtoms())
number of atoms: 5
>>> print("number of bonds:", mol.GetNumBonds())
number of bonds: 4
>>> mol = AllChem.RemoveHs(mol)
>>> print("number of atoms:", mol.GetNumAtoms())
number of atoms: 1
>>> print("number of bonds:", mol.GetNumBonds())
number of bonds: 0
```

Molecules can contain user-definable properties. Each property is defined by a name and a value. Properties are set with the `SetProperty()` method. One of the predefined properties is the molecular name. This name is defined with the `_Name` property value:

```
>>> mol = Chem.MolFromSmiles("c1ccccc1")
>>> mol.SetProp("_Name", "benzene")
>>> print(Chem.MolToMolBlock(mol))
benzene
     RDKit          2D

  6  6  0  0  0  0  0  0  0  0999 V2000
    1.5000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.7500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.5000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.7500    1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7500    1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  2  0
  2  3  1  0
  3  4  2  0
  4  5  1  0
  5  6  2  0
  6  1  1  0
M  END
```

3. Looping over atoms and bonds

Looping over the different atoms in a molecule is done with the `GetAtoms()` method:

```
>>> mol = Chem.MolFromSmiles('C1OC1')
>>> for atom in mol.GetAtoms():
```

```
...     print(atom.GetAtomicNum(), atom.GetIdx(), atom.GetSymbol(), atom.GetExplicitValence())
...
6 0 C 2
8 1 O 2
6 2 C 2
```

One can also access individual atoms by means of their index:

```
>>> for i in range(0, mol.GetNumAtoms()):
...     print(i, mol.GetAtomWithIdx(i).GetSymbol())
...
0 C
1 O
2 C
```

Information about each of the atoms neighbours is retrieved using the `GetNeighbors()` method:

```
>>> for atom in mol.GetAtoms():
...     neighbors = atom.GetNeighbors()
...     print(neighbors)
...     print(atom.GetIdx(), end = ": ")
...     for neighbor in neighbors: print(neighbor.GetIdx(), end="-")
...     print("")
...
(<rdkit.Chem.rdchem.Atom object at 0x7fb667482fa0>, <rdkit.Chem.rdchem.Atom object at
0x7fb6674480a0>)
0: 1-2-
(<rdkit.Chem.rdchem.Atom object at 0x7fb667448220>, <rdkit.Chem.rdchem.Atom object at
0x7fb667448280>)
1: 0-2-
(<rdkit.Chem.rdchem.Atom object at 0x7fb667448040>, <rdkit.Chem.rdchem.Atom object at
0x7fb6674480a0>)
2: 1-0-
```

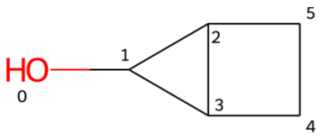
In a similar way, bonds can be looped over with the `GetBonds()` method:

```
>>> for bond in mol.GetBonds():
...     bt = bond.GetBondType()
...     bbi = bond.GetBeginAtomIdx()
...     bei = bond.GetEndAtomIdx()
...     print(bt, bbi, "-", bei)
...
SINGLE 0 - 1
SINGLE 1 - 2
SINGLE 2 - 0
```

4. Rings

RDKit has a wide variety of functions and methods to work with rings. Consider the following molecule:

```
>>> mol = Chem.MolFromSmiles('OC1C2C1CC2')
```



To prompt whether an atom is part of a ring or not, use the `IsInRing()` method:

```
>>> for atom in mol.GetAtoms():
        idx = atom.GetIdx()
        ring = atom.IsInRing()
        r3 = atom.IsInRingSize(3)
        r4 = atom.IsInRingSize(4)
        r6 = atom.IsInRingSize(6)
        print(idx, ring, r3, r4, r6)
```

```
0 False False False False
1 True True False False
2 True True True False
3 True True True False
4 True False True False
5 True False True False
```

The 'smallest set of smallest rings' (SSSR) can be obtained with the `GetSymmSSSR()` function within the `Chem` module. In a multi-ring system there are many cyclic paths; for example a naphthalene system has two paths of length six around the two obvious rings plus a path of length ten around the perimeter. Any two of these three rings would completely describe the ring system, but the shortest cyclic paths are what one normally calls the SSSR (Figure 16):

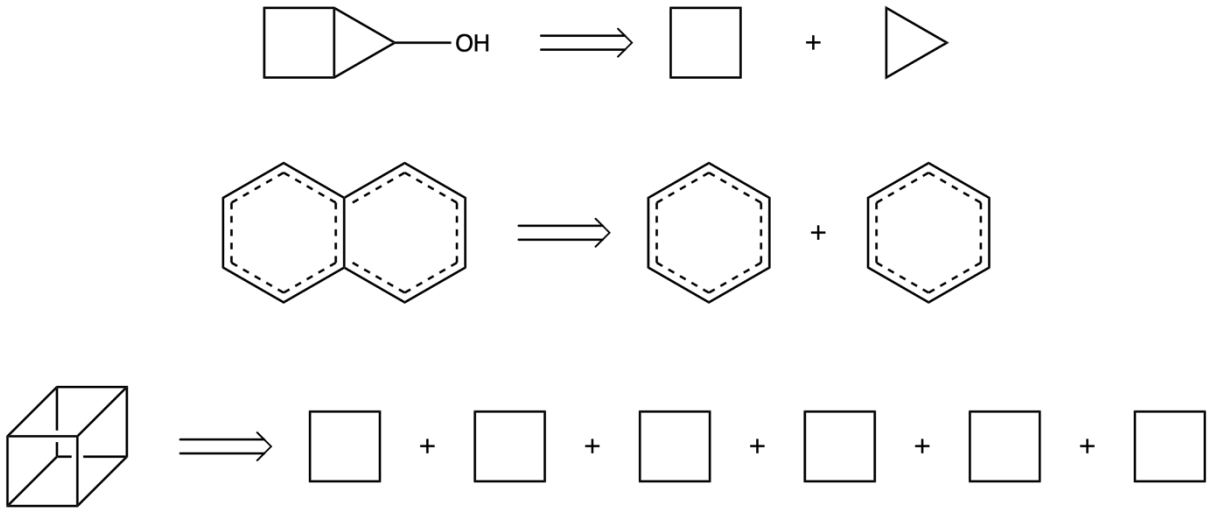


Figure 16. The smallest set of smallest rings (SSSR) depicted for a number of example molecules.

```
>>> mol = Chem.MolFromSmiles("OC1C2C1CC2")    # bicyclo-propane-butane
>>> smallestSetOfSmallestRings = Chem.GetSymmSSSR(mol)
>>> n_sssr = len(smallestSetOfSmallestRings)
>>> print(n_sssr)
2
>>> for i in range(n_sssr): print(list(smallestSetOfSmallestRings[i]))

[1, 2, 3]
[4, 5, 2, 3]
```

```
>>> mol = Chem.MolFromSmiles("c12ccccc1cccc2")    # napthalene
>>> smallestSetOfSmallestRings = Chem.GetSymmSSSR(mol)
>>> n_sssr = len(smallestSetOfSmallestRings)
>>> print(n_sssr)
2
>>> for i in range(n_sssr): print(list(smallestSetOfSmallestRings[i]))

[1, 2, 3, 4, 5, 0]
[6, 7, 8, 9, 0, 5]
```

```
>>> mol = Chem.MolFromSmiles("[C@H]12[C@@H]3[C@@H]4[C@H]1[C@H]5[C@@H]4[C@H]3[C@@H]25")
>>> smallestSetOfSmallestRings = Chem.GetSymmSSSR(mol)
>>> n_sssr = len(smallestSetOfSmallestRings)
>>> print(n_sssr)
6
>>> for i in range(n_sssr): print(list(smallestSetOfSmallestRings[i]))

[0, 3, 2, 1]
[0, 7, 6, 1]
[0, 7, 4, 3]
[1, 6, 5, 2]
[3, 4, 5, 2]
[7, 4, 5, 6]
```

5. Reading and writing molecules

5.1. Single molecules

Molecules can be constructed from a wide variety of molecular formats as decribed in Chapter 2.1. Many examples on how to construct a molecule from a SMILES string with the `Chem.MolFromSmiles()` function have been given in the previous sections, and with the `Chem.MolFromMolFile()` function one can read directly from a SDF file to create a new molecule:

```
>>> mol = Chem.MolFromMolFile("input.sdf")
```

It is good practice to check the validity of the generated molecule, since it may happen that the input file (or SMILES string) contains errors which may lead to errors in the molecule construction phase. This can be done by checking whether the generated molecule is `None` or not:

```
>>> mol = Chem.MolFromSmiles("c1ccccc1")
>>> mol is None
False
>>> mol = Chem.MolFromSmiles("c1cCc1")
[12:52:32] Can't kekulize mol.  Unkekulized atoms: 0 1 3

>>> mol is None
True
>>> smiles = ['c1ccccc1', 'c1cCc1']
>>> mols = []
>>> for s in smiles:
...     mol = Chem.MolFromSmiles(s)
...     if mol is None:
...             continue
...     else:
...             mols.append(mol)
...
[12:55:20] Can't kekulize mol.  Unkekulized atoms: 0 1 3

>>> print(len(mols))
1
```

Molecules can also be created from an InChi string using the `MolFromInchi()` method, which is located in the `Chem.inchi` module:

```
>>> from rdkit.Chem import inchi
>>> mol = Chem.MolFromSmiles("C1CCNCC1")
>>> inchistring = inchi.MolToInchi(mol)
>>> print(inchistring)
InChI=1S/C5H11N/c1-2-4-6-5-3-1/h6H,1-5H2
>>> mol = inchi.MolFromInchi(inchistring)
>>> print(Chem.MolToSmiles(mol))
C1CCNCC1
```

MOL blocks (Chapter 2.1.4.) are also possible to write or read:

```
>>> mol = Chem.MolFromSmiles('C1CCC1')
>>> print(Chem.MolToMolBlock(mol))

     RDKit          2D

  4  4  0  0  0  0  0  0  0  0999 V2000
    1.0607    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.0000   -1.0607    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.0607    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    1.0607    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  1  0
  3  4  1  0
  4  1  1  0
M  END
```

Property data can be added with the `SetProp()` method of the molecule. In order to specify the name of the molecule, set the `_Name` property to the actual name

```
>>> mol.SetProp("_Name","cyclobutane")
>>> print(Chem.MolToMolBlock(mol))
cyclobutane
     RDKit          2D

  4  4  0  0  0  0  0  0  0  0999 V2000
    1.0607    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.0000   -1.0607    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.0607    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.0000    1.0607    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  1  0
  3  4  1  0
  4  1  1  0
M  END
```

This name, or any other property that has been set, can be retrieved with the `GetProp()` method:

```
>>> mol.GetProp("_Name")
'cyclobutane'
```

5.2. Sets of molecules

Multiple molecules can be written to a SDF file (Chapter 2.1.4.) using the `SDWriter()` function:

```
>>> smiles = ["C", "CC", "CO", "CCO", "C1OC1"]
>>> mols = []
>>> for s in smiles: mols.append(Chem.MolFromSmiles(s))
>>> writer = Chem.SDWriter("filename.sdf")
>>> for mol in mols: writer.write(mol)
>>> writer.close()
```

These molecules can be read in again with the corresponding `SDMolSupplier()` function:

```
>>> reader = Chem.SDMolSupplier("filename.sdf")
>>> for mol in reader:
...     if mol is None: continue
...     print(Chem.MolToSmiles(mol))
...
C
CC
CO
CCO
C1CO1
```

To generate a file containing the SMILES representations of molecules, you can use the standard Python `write()` function:

```
>>> f = open("file.smi", "w")
>>> for mol in mols: f.write(Chem.MolToSmiles(mol) + "\n")
...
2
3
3
4
6
>>> f.close()
```

6. Working with conformations

When molecules are created from input like SMILES, there is no information present that describes the position of each atom in three-dimensional space (conformations). RDKit contains a number of functions that allow the

user to generate the 3D-conformation(s) of molecules, and one of these is the `EmbedMolecule()` function within the `AllChem` module. Consider this example on aspirine:

```
>>> mol = Chem.MolFromSmiles("CC(=O)Oc1ccccc1C(=O)O")
>>> mol.SetProp("_Name", "aspirine")
>>> print(Chem.MolToMolBlock(mol))
aspirine
     RDKit          2D

 13 13  0  0  0  0  0  0  0  0999 V2000
    5.2500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    3.7500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    3.0000   -2.5981    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    3.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    1.5000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.7500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.5000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.7500    1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7500    1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.5000    2.5981    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7500    3.8971    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    3.0000    2.5981    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  2  0
  2  4  1  0
  4  5  1  0
  5  6  2  0
  6  7  1  0
  7  8  2  0
  8  9  1  0
  9 10  2  0
 10 11  1  0
 11 12  2  0
 11 13  1  0
 10  5  1  0
M  END
```

A number of observations can be made. First, the molecule does not contains hydrogens atoms (these are implicitely present but are not showing up in the connectivities). Second, the z-coordinates of the atoms (third column) are all set to 0, meaning that the molecule contains 2D- rather than 3D-information. To generate a conformation of aspirine, first the hydrogen atoms need to be made explicit:

```
>>> mol = Chem.AddHs(mol)
>>> print(Chem.MolToMolBlock(mol))
aspirine
     RDKit          2D

 21 21  0  0  0  0  0  0  0  0999 V2000
    5.2500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    3.7500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    3.0000   -2.5981    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    3.0000    0.0000    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    1.5000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.7500   -1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.5000    0.0000    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.7500    1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.7500    1.2990    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.5000    2.5981    0.0000 C   0  0  0  0  0  0  0  0  0  0  0  0
    3.0000    2.5981    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    0.7500    3.8971    0.0000 O   0  0  0  0  0  0  0  0  0  0  0  0
    6.7500   -1.2990    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
    5.2500   -2.7990    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
    5.2500    0.2010    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
    1.5000   -2.5981    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
   -1.5000   -2.5981    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
   -3.0000    0.0000    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
   -1.5000    2.5981    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
    1.5000    5.1962    0.0000 H   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
```

```
  2  3  2  0
  2  4  1  0
  4  5  1  0
  5  6  2  0
  6  7  1  0
  7  8  2  0
  8  9  1  0
  9 10  2  0
 10 11  1  0
 11 12  2  0
 11 13  1  0
 10  5  1  0
  1 14  1  0
  1 15  1  0
  1 16  1  0
  6 17  1  0
  7 18  1  0
  8 19  1  0
  9 20  1  0
 13 21  1  0
M  END
```

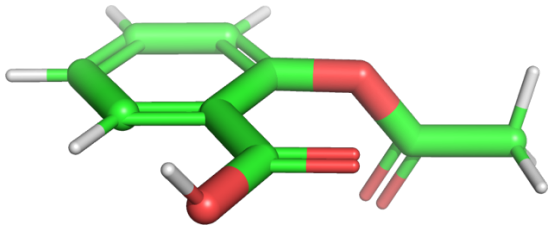
Second, a conformation needs to be generated with the `EmbedMolecule()` function:

```
>>> AllChem.EmbedMolecule(mol)
0
>>> print(Chem.MolToMolBlock(mol))
aspirine
     RDKit          3D

 21 21  0  0  0  0  0  0  0  0999 V2000
   -2.8028   -2.2971    0.2528 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.9196   -1.1193    0.3821 C   0  0  0  0  0  0  0  0  0  0  0  0
   -1.9380   -0.4309    1.4053 O   0  0  0  0  0  0  0  0  0  0  0  0
   -1.0329   -0.6957   -0.5799 O   0  0  0  0  0  0  0  0  0  0  0  0
   -0.2038    0.3759   -0.5076 C   0  0  0  0  0  0  0  0  0  0  0  0
   -0.5008    1.6592   -0.9114 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.4068    2.6946   -0.8011 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.6647    2.4889   -0.2762 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.9781    1.2093    0.1330 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.0695    0.1765    0.0217 C   0  0  0  0  0  0  0  0  0  0  0  0
    1.4778   -1.1541    0.4786 C   0  0  0  0  0  0  0  0  0  0  0  0
    0.6542   -2.1230    0.3851 O   0  0  0  0  0  0  0  0  0  0  0  0
    2.7241   -1.3647    1.0002 O   0  0  0  0  0  0  0  0  0  0  0  0
   -2.9017   -2.5297   -0.8210 H   0  0  0  0  0  0  0  0  0  0  0  0
   -3.7860   -2.0195    0.6858 H   0  0  0  0  0  0  0  0  0  0  0  0
   -2.4405   -3.1762    0.7986 H   0  0  0  0  0  0  0  0  0  0  0  0
   -1.4956    1.7914   -1.3196 H   0  0  0  0  0  0  0  0  0  0  0  0
    0.1087    3.6835   -1.1386 H   0  0  0  0  0  0  0  0  0  0  0  0
    2.3837    3.2960   -0.1858 H   0  0  0  0  0  0  0  0  0  0  0  0
    2.9838    1.0330    0.5554 H   0  0  0  0  0  0  0  0  0  0  0  0
    3.5700   -1.4979    0.4428 H   0  0  0  0  0  0  0  0  0  0  0  0
  1  2  1  0
  2  3  2  0
  2  4  1  0
  4  5  1  0
  5  6  2  0
  6  7  1  0
  7  8  2  0
  8  9  1  0
  9 10  2  0
 10 11  1  0
 11 12  2  0
 11 13  1  0
 10  5  1  0
  1 14  1  0
  1 15  1  0
  1 16  1  0
  6 17  1  0
  7 18  1  0
  8 19  1  0
  9 20  1  0
 13 21  1  0
```

```
M  END
```

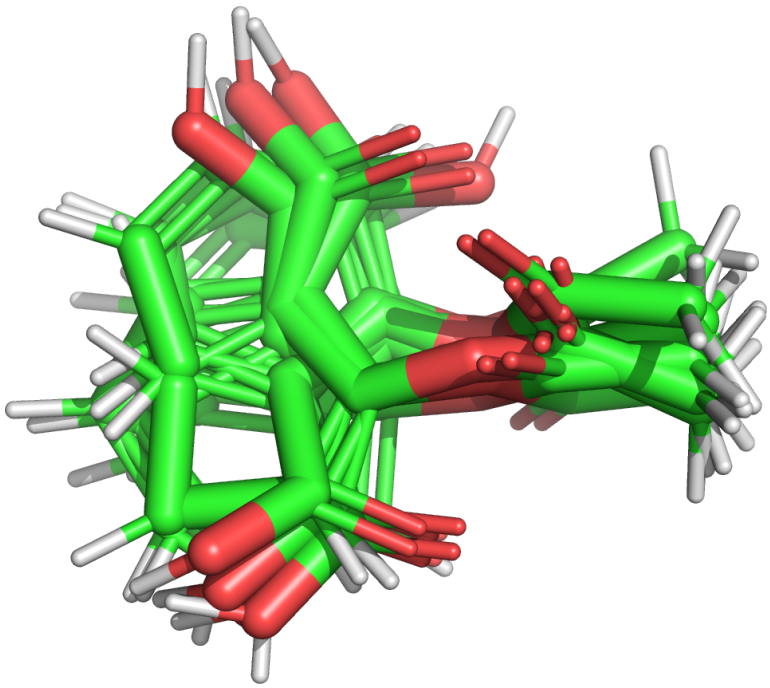
This is better visualised using a molecular graphics visualisation (section 1.2):



Often a single conformation is not enough, as most drug-like molecules contain a significant number of flexible bonds around which the molecule can rotate. Therefore, it is more realistic to generate multiple conformations of the same molecule. This can be achieved with the `EmbedMultipleConfs()` function within `AllChem`:

```
>>> mol = Chem.MolFromSmiles("CC(=O)Oc1ccccc1C(=O)O")
>>> mol = Chem.AddHs(mol)
>>> conformationIds = AllChem.EmbedMultipleConfs(mol, numConfs=10)
>>> print(len(conformationIds))
10
>>> w = Chem.SDWriter("aspirin.sdf")
>>> for cid in conformationIds: w.write(mol, confId = cid)
>>> w.close()
```

This script tries to generate 10 different aspirin conformations and writes these to a file called "`aspirin.sdf`". The 10 conformations in this file can be visualised with a graphical program like PyMol:

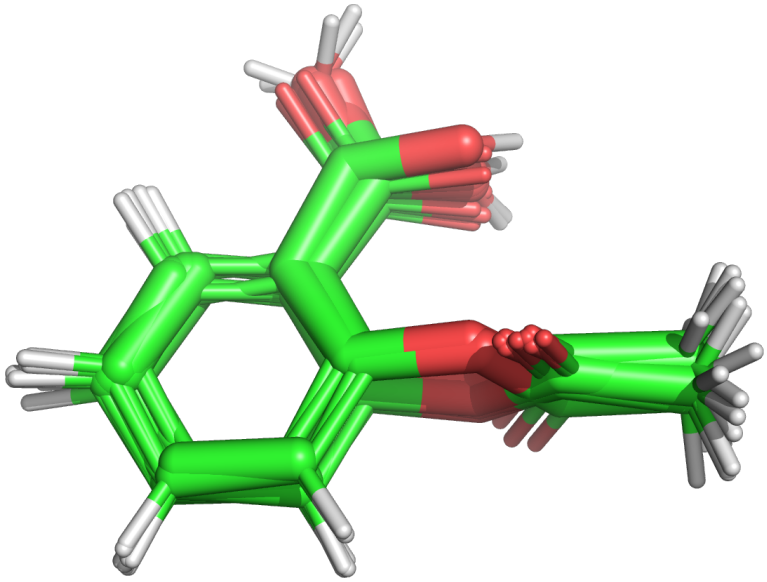


As can be seen, the different conformations are positioned randomly in space, and it would be better to align the structures onto each other for easier visualisation:

```
>>> mol = Chem.MolFromSmiles("CC(=O)Oc1ccccc1C(=O)O")
>>> mol = Chem.AddHs(mol)
>>> conformationIds = AllChem.EmbedMultipleConfs(mol, numConfs=10)
>>> rmslist = []
>>> AllChem.AlignMolConformers(mol, RMSlist = rmslist)
>>> for rms in rmslist: print(rms)

1.017382225703262
1.3937357171422475
1.0824690989074286
1.2752681581629701
```

```
1.0768188563130232
1.4597310140042876
1.3692269740893361
1.4493429134502234
1.5222762797711171
>>> w = Chem.SDWriter("aspirin.sdf")
>>> for cid in conformationIds: w.write(mol, confId = cid)
>>> w.close()
```



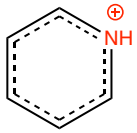
7. Substructure searching: SMARTS

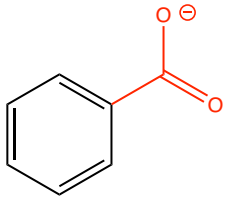
Substructure searching is a very useful application of chemo-informatics. Substructure searches can be used to identify user-defined fragments (substructures) in query molecules. In order to define the substructure that one wants to use, the SMARTS line notation is used (SMARTS stands for **SM**ILES **ar**bitrary **t**arget **s**pecification). SMARTS is related to the SMILES line notation. It was originally developed by David Weininger and colleagues at Daylight Chemical Information Systems. The most comprehensive descriptions of the SMARTS language can be found in Daylight's SMARTS theory manual, tutorial and examples. A useful website to validate your SMARTS is SMARTS-PLUS.

7.1. SMARTS syntax

Atoms

Atoms are specified by their symbol or by their atomic number. An aliphatic carbon is matched by the SMARTS pattern `[C]`, aromatic carbon by `[c]` and any carbon by `[#6]` or `[C,c]` (the comma denotes the `or` operator). The wild card symbols `*`, `A` and `a` match any atom, any aliphatic atom and any aromatic atom respectively (the `*` can also be written as `[A,a]`). Implicit hydrogens are considered to be a characteristic of atoms but these hydrogens can be specified if desired or required. For example, the SMARTS for an amino group can be written as `[NH2]`, but `[N]` can also match an amino group if the nitrogen contains two hydrogens. Atomic formal charge is specified by the descriptors `+` and `-` as exemplified by the SMARTS `[nH+]` of a protonated aromatic nitrogen atom and `[O-]C(=O)c` (deprotonated aromatic carboxylic acid):

[nH+] matches 

[O-]C(=O)c matches 

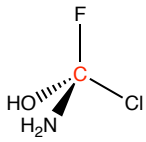
Bonds

Bonds types can be specified with the - (single), = (double), # (triple), : (aromatic) and ~ (any) tokens. Similar to SMILES, when a bond is not specified, then a single (in the case of aliphatic flanking atoms) or aromatic bond type (in the case of aromatic flanking atoms) is assumed.

Connectivity

The X and D descriptors are used to specify the total numbers of connections (including implicit hydrogen atoms) and connections to explicit atoms, respectively. As an example, [CX4] matches carbon atoms with bonds to any four other atoms while [CD4] only matches quaternary carbons:

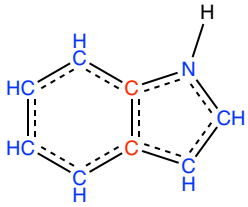
CH_4 is matched by [CX4]



is matched by [CX4] and [CD4]

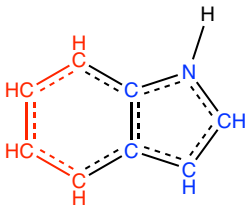
Cyclicity

As originally defined by Daylight, the R descriptor is used to specify ring membership. In the Daylight model for cyclic systems, the smallest set of smallest rings (SSSR) is used as a basis for ring membership. For example, indole is perceived as a 5-membered ring fused with a 6-membered ring rather than a 9-membered ring. The two carbon atoms that make up the ring fusion would match [cR2] and the other carbon atoms would match [cR1]:



match [cR1] [cR2]

Lower case r specifies the size of the smallest ring of which the atom is a member. The carbon atoms of the ring atoms in indole would both match [cr5]:



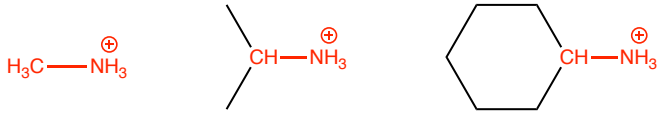
match [cr5] [cr6]

Bonds can be specified as cyclic, for example C@C matches directly bonded atoms in a ring.

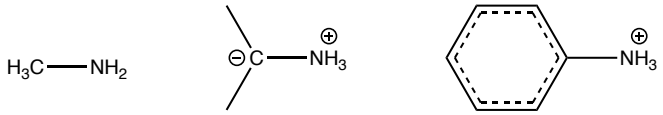
Four logical operators allow atom and bond descriptors to be combined:

Operator	Symbol
and	;
or	,
and (high priority)	&
not	!

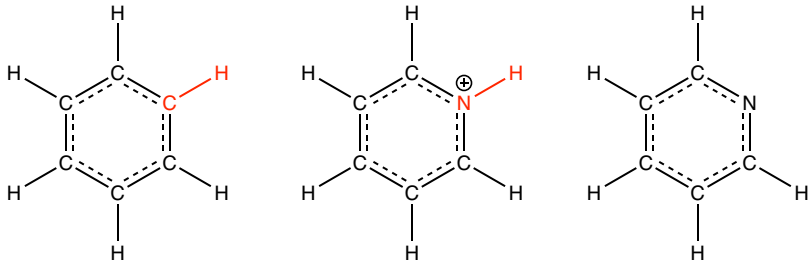
The *and* operator (`;`) can be used to define a protonated primary amine as `[N;H3;+][C;X4]`. This reads as a nitrogen atom (`N`) with (`;`) three hydrogens (`H3`) and (`;`) a positive charge (`+`), bonded with a single bond to an aliphatic carbon atom (`C`) that has (`;`) four implicit or explicit connections (`X4`). The following examples all match this SMARTS pattern:



but these examples do not:

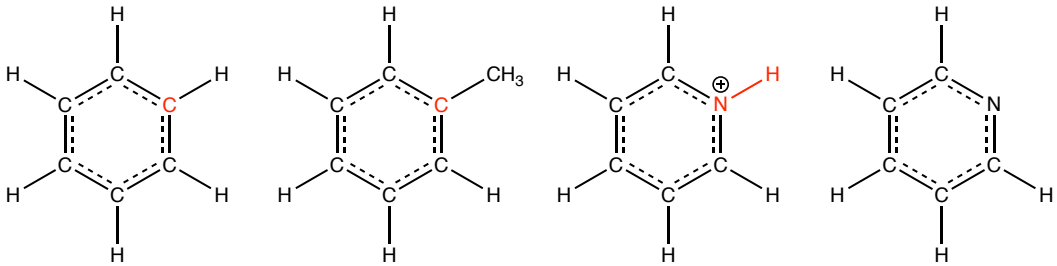


The *or* operator (`,`) has a higher priority than the normal *and* (`;`), so `[c,n;H]` defines an aromatic carbon or an aromatic nitrogen, each possibility bonded to a hydrogen atom:



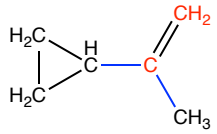
match match no match

The *and* operator (&) has higher priority than *or* (`,`) so `[c,n&H]` defines an aromatic carbon, or an aromatic nitrogen with hydrogen atom connected to:



match match match no match

The *not* operator (`!`) can be used to define unsaturated aliphatic carbon as `[C;!X4]` and acyclic single bonds as `*-!@*`:



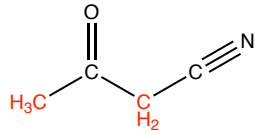

```
[C;!X4]
*-!@*
```

Recursive SMARTS

Recursive SMARTS allow detailed specification of an atom's environment. It is indicated with a dollar sign (`$`). This SMARTS feature allows you to define an "atomic enviroment" that is required for matching. The "enviroment" atoms will not be included into result match. Any SMARTS expression may be used to define an atomic environment by writing a SMARTS starting with the atom of interest in this form:

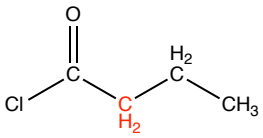
```
$(SMARTS)
```

For example `[C&!$(C=O)&!$(C#N)]` will match any aliphatic carbon not double bonded to an oxygen and not triple bonded to a nitrogen:



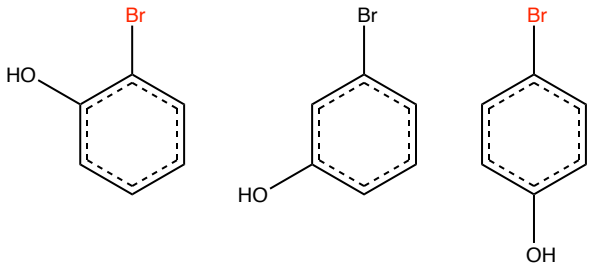
match

while `C[$(C=O)]` will match any aliphatic carbon that is on the α-position of an carbonyl function:



match

As another example, the pattern `Br[$(c1c([OH])cccc1),$(c1ccc([OH])cc1)]` will match any bromo atom that is located on a phenol ring in ortho- or para-position:

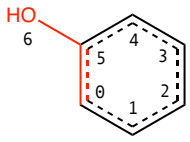


match no match match

7.2. Working with SMARTS

Substructure matching can be done using query molecules built from SMARTS:

```
>>> m = Chem.MolFromSmiles('c1ccccc1O')
>>> smartsMol = Chem.MolFromSmarts('ccO')
>>> m.HasSubstructMatch(smartsMol)
True
>>> m.GetSubstructMatch(smartsMol)
(0, 5, 6)
```



What is returned are the indices of the atoms in the target molecule (phenol), ordered according the atoms in the SMARTS pattern. To get all matches, use the `GetSubstructMatches()` method:

```
>>> m.GetSubstructMatches(smartsMol)
((0, 5, 6), (4, 5, 6))
```

To illustrate the bromophenol example above, we can write the following code:

```
>>> bromophenols = ["Oc1ccccc1Br", "Oc1cccc(Br)c1", "Oc1ccc(Br)cc1"]
>>> mols = []
>>> for bp in bromophenols:
...     mols.append(Chem.MolFromSmiles(bp))
...
>>> p = Chem.MolFromSmarts("Br[$(c1c([OH])cccc1),$(c1ccc([OH])cc1)]")
>>> for mol in mols:
...     if mol.HasSubstructMatch(p):
...             print(Chem.MolToSmiles(mol), "True")
...     else:
...             print(Chem.MolToSmiles(mol), "False")
...
Oc1ccccc1Br True
Oc1cccc(Br)c1 False
Oc1ccc(Br)cc1 True
```

8. Excercises

8.1. Exercise 1

Write a writer function that takes as input a list of molecules and a filename, and that writes these molecules out as SMILES (each molecule on a separate line).

8.2. Exercise 2

Write a function that takes as input both a filename and a molecule, and that writes the molecule to the file in MOL-block format (there is a function in RDKit that does this, but this exercise is about writing this function yourselves).

8.3. Exercise 3

Write a program that reads a list of molecules (represented in SMILES), calculates their molecular weight, and prints this SMILES together with the molecular weight on the same line. Have a look at the RDKit page to find out which functions you need. The MW can be calculated using RDKit with the `MolWt()` function in the `Descriptors` module:

```
>>> from rdkit.Chem import Descriptors
>>> m = Chem.MolFromSmiles('c1ccccc1C(=O)O')
>>> Descriptors.MolWt(m)
122.12299999999998
```

8.4. Exercise 4

Write a program that takes as input a SDF-file of molecules and that calculates the Lipinski's rule-of-five on these molecules. The rule-of-five is a simple criterium to predict whether a molecule is drug-like or not. A molecule confers to the rule-of-five when all of the following criteria are met:

- Molecular weight <= 500 Da. The MW can be calculated using RDKit with the `MolWt()` function in the `Descriptors` module.
- Calculated logP <= 5. The logP can be calculated with RDKit using the `MolLogP()` function in the `Descriptors` module:

```
>>> from rdkit.Chem import Descriptors
>>> m = Chem.MolFromSmiles('c1ccccc1C(=O)O')
>>> Descriptors.MolLogP(m)
1.3848
```

- Number of hydrogen bond donors <= 5. Any O or N atom connected to at least one H is counted as a hydrogen bond donor. Use substructure searches with SMARTS to implement this.
- Number of hydrogen bond acceptors <= 10. Any O or N atom in the molecule is counted as a hydrogen bond acceptor. Use substructure searches with SMARTS to implement this.

Chapter 4. Fingerprints, molecular similarity and clustering

1. Molecular fingerprints

In order for computers to handle, search and compare molecules, it is necessary that these molecules are represented in a computer-readable form. Molecular fingerprints are a way of encoding the structure of a molecule, and which involve turning the molecule into a sequence of bits that can then be easily compared between molecules. There are several types of molecular fingerprints depending on the method by which the molecular representation is transformed into a bit string. Most methods use only the 2D molecular graphs (the topology) and are thus called 2D fingerprints. The best known of these 2D fingerprints are the Daylight, Morgan and MACCS fingerprints.

1.1. Linear path-based: Daylight fingerprints

The Daylight fingerprints were originally developed by Daylight Chemical Information Systems, Inc., a US-based company founded in 1987 by the Weininger brothers (http://www.daylight.com/about/index.html). Their fingerprint technology is categorised as topological or path-based fingerprints, and work by analysing all the fragments of the molecule following a linear path up to a certain number of bonds, and then hashing every one of these paths to create the fingerprint (Figure 17). The Daylight fingerprints consist of up to 2,048 bits and encode all possible connectivity pathways through a molecule up to a given length (mostly 7 atoms). Most software packages implement these fingerprints or fingerprints based on them.

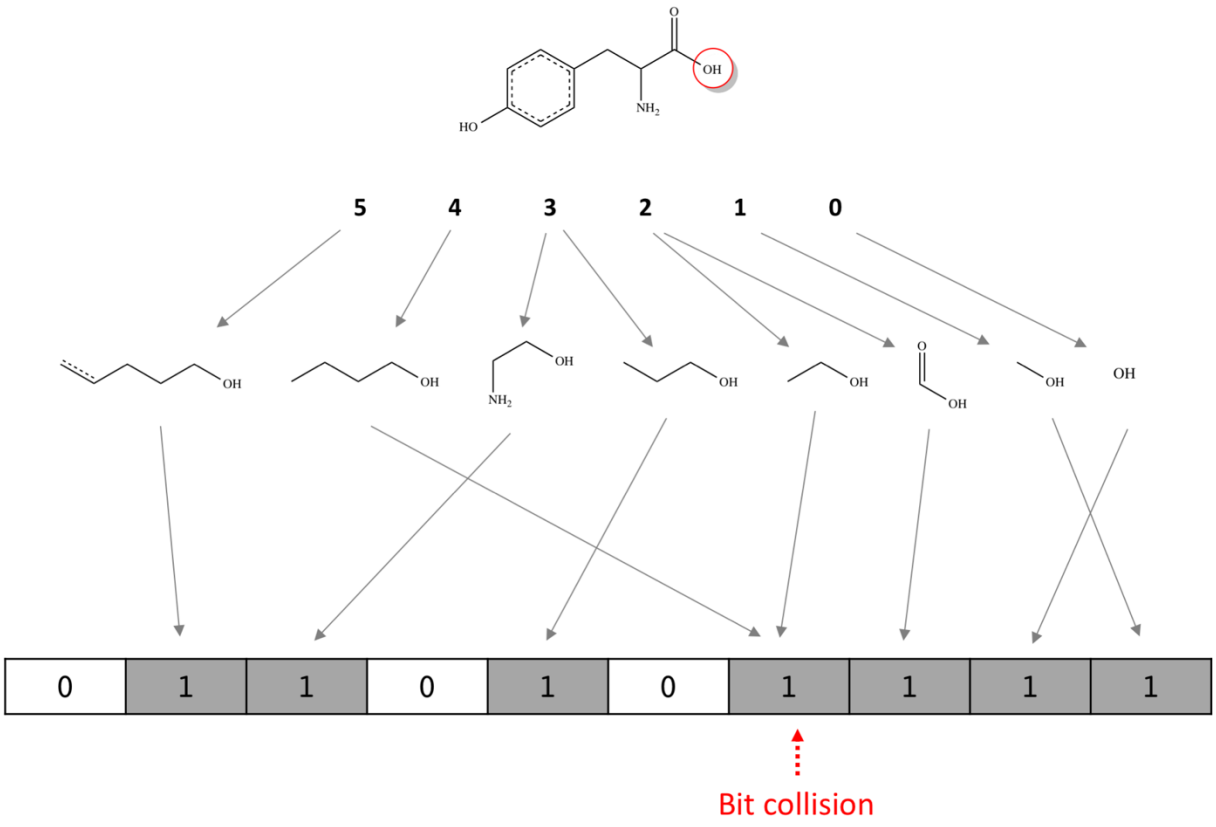


Figure 17. A representation of a hypothetical 10-bit topological fingerprint (a real Daylight fingerprint consists of 2,048 bits), in this case a linear path-based fingerprint with fragments up to a length of 5 (in the Daylight implementation this is normally a length of 7 atoms). All fragments found from the starting atom (circled) are shown, and the fragment length and corresponding bit in the fingerprint are indicated. There is one-bit collision, which are bits that are set by more than one fragment; these are likely in fingerprints with a reduced number of bits. Only fragments and bits for a single starting atom are shown; for the full fingerprint, this process would be carried out for every atom in the molecule. Adapted from reference 4.

1.2. Circular path-based: Morgan fingerprints

Circular fingerprints are also hashed topological fingerprints, but they are different in that instead of looking for paths in the molecule, the environment of each atom up to a determined radius is recorded. They are widely used for full structure similarity searching. The Morgan fingerprints are also called ECFP fingerprints (Extended-Connectivity Fingerprint). They represent circular atom neighbourhoods and produce fingerprints of variable length. They are most commonly used with a diameter of 4 and referred to as ECFP4. A diameter of 6 (ECFP6) is also commonly used.

ECFPs have two typical representations (Figure 18):

- List of integer identifiers. The natural and accurate representation of ECFPs is by means of varying-length lists of integer identifiers. Each identifier represents a particular substructure, more precisely, a circular atom neighbourhood, which is present in the molecule. The list of integer identifiers is sorted in ascending order. These identifiers can also be interpreted as indexes of bits in a huge virtual bit string. Each position in this bit string accounts for the presence or absence of a specific substructure feature. Since this virtual bit string is extremely large and sparse, it is not stored explicitly, but the indexes of the 1 bits are recorded in a varying-length list. In spite of this interpretation, the feature identifiers are stored as signed values due to technical reasons, that is, they can be either positive or negative. By default, this integer list representation contains only one instance of each identifier. However, in particular applications, it could be beneficial to consider the frequency count of the ECFP features, that is, to record each identifier as many times as the represented feature occurs in the molecule. This variation of ECFP is often denoted as ECFC.
- Fixed-length bit string. Traditional representation of binary molecular fingerprints is by means of fixed-length bit strings. This representation can also be applied to ECFPs by "folding" the underlying virtual bit string into a much shorter bit string of specified length (for example 1,024). Compared to the identifier lists, this representation simplifies the comparison and similarity calculation of ECFPs and it could reduce the required storage space, especially for large molecules. On the other hand, the applied folding operation increases the likelihood of collision, that is, two (or more) different substructure features could be represented by the same bit position. As a result, a certain amount of information is usually lost, which worsens both the quality and interpretability of this representation.

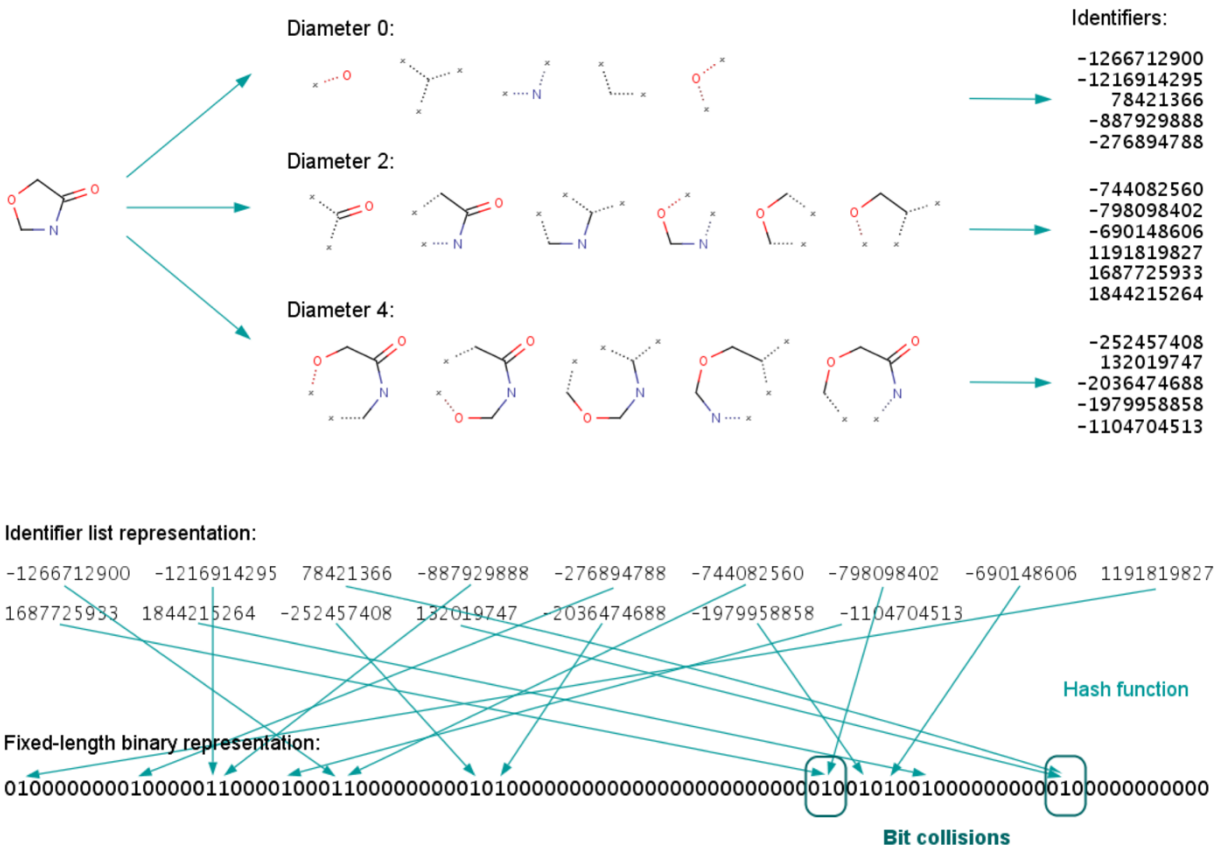


Figure 18. Upper part: ECFP generation process. Lower part: generation of the fixed-length bit string by 'folding'.

1.3. Substructure-based: MACCS keys

Substructure keys-based fingerprints set the bits of the bit string depending on the presence in the compound of certain substructures or features from a given list of structural keys. This usually means that these fingerprints are most useful when used with molecules that are likely to be mostly covered by the given structural keys, but not so much when the molecules are unlikely to contain the structural keys, as their features would not be represented. Their number of bits is determined by the number of structural keys, and each bit relates to presence or absence of a single given feature in the molecule (Figure 19), which is not the case with path-based fingerprints.

The MACCS-key comes in two variants, one with 960 and the other with 166 structural keys based on substructure patterns. The shorter one is the most commonly used, as it is relatively small in length but covers most of the interesting chemical features for drug discovery and virtual screening. The larger variant contains proprietary substructures and cannot be processed by all software packages.

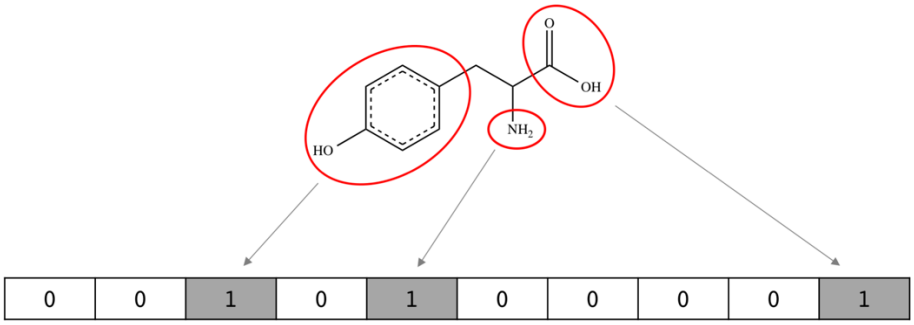


Figure 19. A representation of a hypothetical 10-bit substructure fingerprint, with three bits set because the substructures they represent are present in the molecule (circled). Adapted from reference 4.

2. Similarity metrics

Quantifying the similarity of two molecules is a key concept and a routine task in cheminformatics. Its applications encompass a number of fields, mostly medicinal chemistry-related, such as virtual screening. A virtually infinite number of methods have been described to calculate similarity and dissimilarity, but the Tanimoto index and Euclidean distance metrics being the most widely used.

2.1. Tanimoto

The Tanimoto similarity metric is commonly used in conjunction with bitwise fingerprints, such as the linear hash-based Daylight and the fixed-length Morgan fingerprints. Represented as a mathematical equation:

$$T(a,b) = \frac{N_c}{N_a + N_b - N_c}$$

with N_a and N_b representing the number of bits set (state 1) in fingerprints ***a*** and ***b***, respectively, and N_c the number of common bits set. The Tanimoto distance runs from 0 to 1.

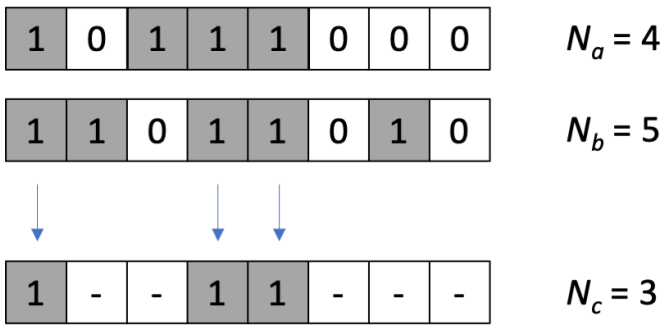


Figure 20. Illustration of the Tanimoto index on a hypothetical example of two bit strings of 8 bits each. T(a,b) = 3 / (4 + 5 – 3) = 0.5. Perfect similarity results in an index of 1, while perfect dissimilarity results in an index of 0.

2.2. Tversky

The Tversky similarity measure is an asymmetrical metric, meaning that the resulting value depends on the order of the two fingerprints:

$$T(a,b) = \frac{N_c}{\alpha O_a + \beta O_b + N_c}$$

with O_a and O_b being the number of bits that are only set (state 1) in the fingerprints of ***a*** and ***b***, respectively. N_c is defined as above. The factors α and β are weighing factors that can be used to put more weight on the fingerprints ***a*** and ***b***, respectively. Setting the parameters $\alpha = \beta = 1.0$ is identical to using the Tanimoto measure. The Tversky metric runs from 0 to 1, with 0 indicating total dissimilarity and 1 indicating equality.

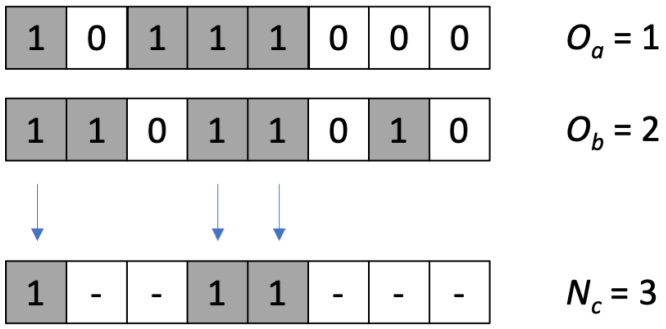


*Figure 21. Illustration of the Tversky index on a hypothetical example of two bit strings of 8 bits each. With $\alpha = 0.9$ and $\beta = 0.1$, T(a,b) = 3 / (0.9 * 1 + 0.1 * 2 + 3) = 0.73. However, with $\alpha = 0.1$ and $\beta = 0.9$, T(a,b) = 3 / (0.1 * 1 + 0.9 * 2 + 3) = 0.61. Perfect similarity results in an index of 1, while perfect dissimilarity results in an index of 0.*

The actual choice of the α and β values depends on the research question one wants to answer. If fingerprint $\boldsymbol{a}$ is the fingerprint of a query molecule, and $\boldsymbol{b}$ is the fingerprint of a molecule in a database in which one wants to look for molecules that are similar to $\boldsymbol{a}$, then setting α to a large value (e.g. 0.9) will give large values of *T(a,b)* for database molecules that are superstructures of the query $\boldsymbol{a}$, while setting α to a small value (e.g. 0.1) will give large values of *T(a,b)* for database molecules that are rather substructures of $\boldsymbol{a}$.

2.3. Application of similarity metrics

Similarity or distance measures have been used widely to calculate the similarity or dissimilarity between two samples of dataset. Cheminformatics is known as the domain that dealing with chemical information and both similarity and distance coefficient have been an important role for matching, searching and classification of chemical information. Similarity metrics are applied in a number of tasks:

- Selection of a set of diverse compounds for *in vitro* screening. In case one wants to purchase a small subset of chemical compounds from a large database of potential structure, it is often advisable to select a small but diverse subset of these structures in order to increase the likelihood of identifying a potential hit compound that is active against the *in vitro* screen one is investigating. Diversity-based selection is often performed by means of compound clustering, an approach that is further described in section 3.
- Selection of a set of similar compounds for hit conformation and the development of quantitative structure-activity relationships (QSAR). The chemical similarity principle, which states that compounds with similar structure will probably have similar bioactivities, is an underlying assumption of similarity-based virtual screening.
- Development of machine learning models in QSAR. Machine learning techniques can be broadly classified as supervised or unsupervised learning.

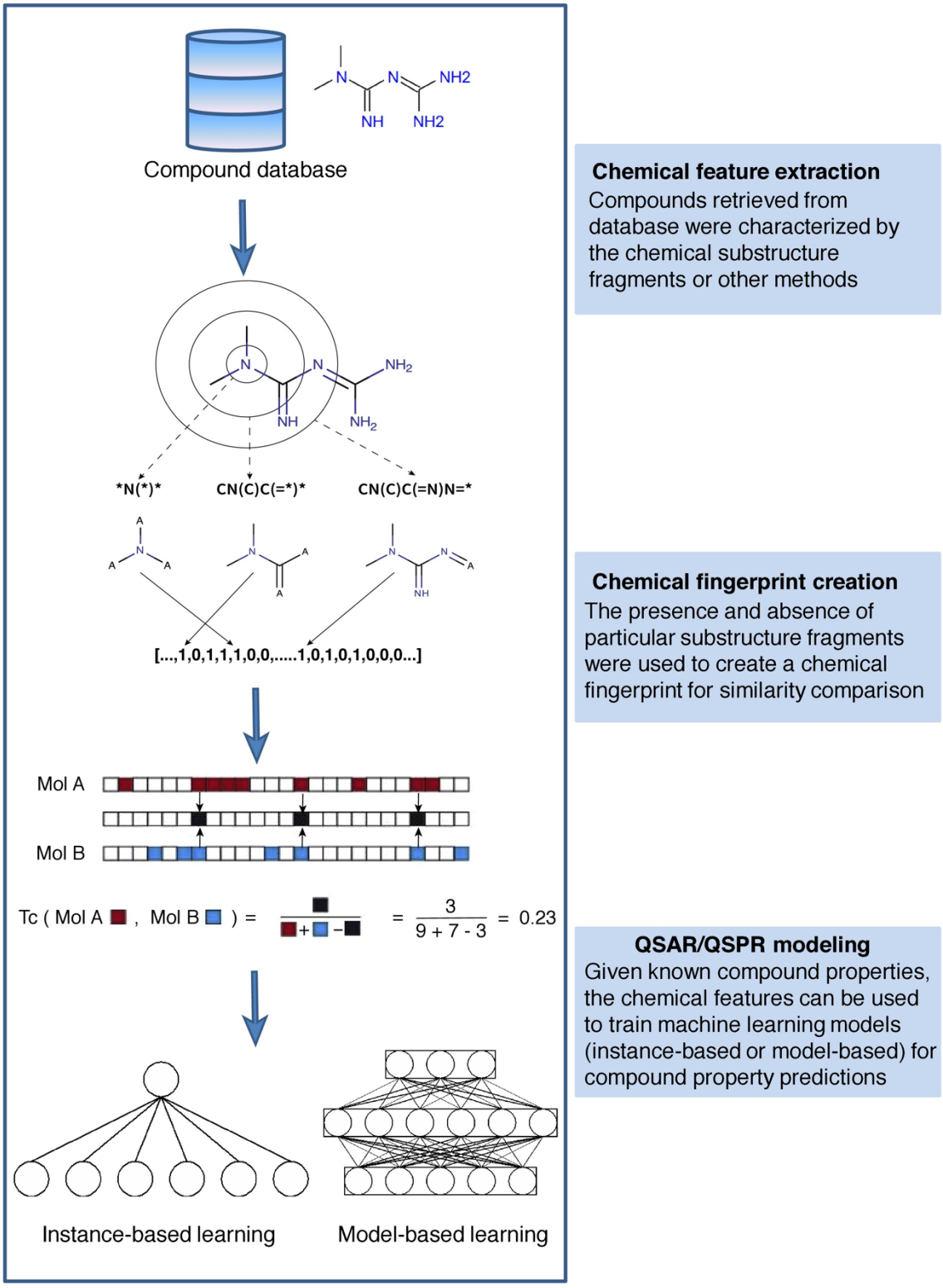


Figure 22. Overview of the process that is commonly used in pharmaceutical drug discovery. Compound databases are converted in chemical fingerprints, which are in turn used to search for similar compounds or for QSAR model building.

3. Maximum common substructure

The maximum common substructure (MCSS) is the largest substructure (graph) which can be identified between two or more molecules.

To illustrate the MCSS concept, consider the example in Figure 23. In this example, the MCSS between morphine, codeine and heroine is shown in red.

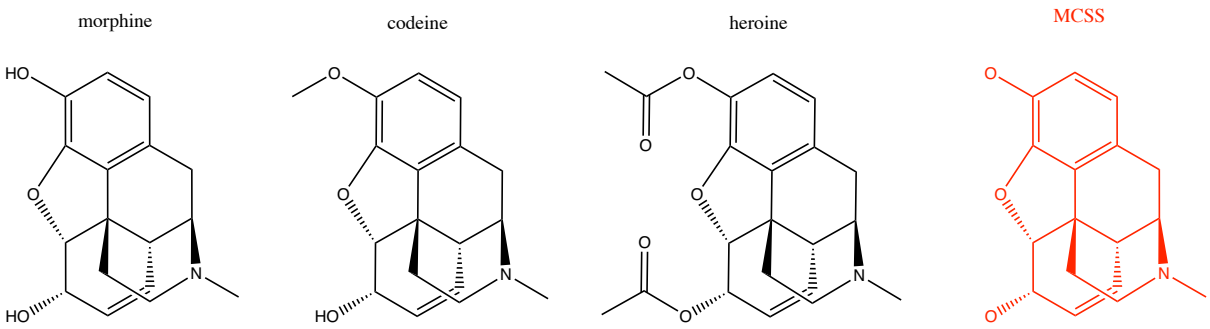


Figure 23. The maximum common substructure (red) between morphine, codeine and heroine. RDKit was used to calculate this MCSS.

The RDKit code to calculate the MCSS is rather straightforward but may take some time for complex molecule sets (normally in the order of microseconds, but it may go to minutes in some rare cases):

```
from rdkit.Chem import rdFMCS

morphine = Chem.MolFromSmiles("CN1CC[C@]23C4=C5C=CC(O)=C4O[C@H]2[C@H](C=C[C@H]3[C@H]1C5)O")
codeine = Chem.MolFromSmiles("CN1CC[C@]23[C@@H]4[C@H]1CC5=C2C(O[C@H]3[C@@H](O)C=C4)=C(OC)C=C5")
heroine =
Chem.MolFromSmiles("CN([C@H](CC(C=C1)=C23)[C@@H]4C=C[C@@H]5OC(C)=O)CC[C@]43[C@H]5OC2=C1OC(C)=O")

mols = [morphine, codeine, heroine]
mcss = rdFMCS.FindMCS(mols)
```

There exist many variations in the algorithms to calculate the MCSS of molecules, but the majority is based on some kind of backtracking approach in which an exhaustive search on all combinations is performed. Each molecule is converted into a graph representation (a graph is a set of edges [bonds] and nodes [atoms]) that is traversed iteratively to identify common edges and nodes.

4. Clustering

With the increase in the amount of chemical information available, methods that can organize chemical structures and their associated data are essential. Cluster analysis refers to a group of statistical methods that are used for identifying groups ('clusters') of similar items in multidimensional space. They require a measure of similarity between items, hence the Tanimoto or Euclidean distance measures are wide used for this. In cheminformatics, clustering methods are used for three main purposes:

- Grouping compounds into chemical series (or something approximating to this), as a way of organizing large datasets. For example, it is easier for a chemist to browse through 500 clusters (where the molecules in a cluster are similar) than 50,000 arbitrarily ordered compounds
- Identifying new bioactive molecules: if a compound with unknown activity is in a cluster that is biased towards compounds with known activity, we can make a prediction of the probability of activity of the unknown compound (for example, if 75% of the compounds in the cluster are active, we might say the probability of activity is 75%).
- Picking representative subsets: if we cluster a set of compounds, we can then take one compound from each cluster as a 'representative' of this cluster, and the total set of representative compounds as a representative subset of the whole dataset. This is sometimes more useful than random selection.

4.1. Hierarchical clustering

Clustering methods can be either **hierarchical** or **non-hierarchical**. Hierarchical clustering creates a tree of clusters, with, at the bottom level, every item in its own cluster, and at the top level all items in one cluster. Algorithmically,

this can be done either by starting at the bottom and progressively merging clusters (agglomerative) or starting at the top and breaking up clusters (divisive).

Mostly, the hierarchical agglomerative methods work in the same algorithmic fashion, but differ in the way that they decide which clusters to merge at each level.

1) The **Ward's method** has been used widely in cheminformatics, and is distinguished by merging clusters which, when merged, have the smallest increase in variance from the mean (i.e. create the 'tightest' cluster when merged).

2) Other methods include **single linkage** (the clusters are merged with the minimum distance between the nearest two points in each cluster); complete linkage (the clusters are merged with the minimum distance between the farthest points in each cluster); and group average (the minimum value of the mean distance between all pairs in the two clusters). Due to their computational complexity and memory requirements, hierarchical methods do not scale well to very large datasets, and thus they are giving way to faster, non-hierarchical methods. In order to create a partitioned grouping of a dataset (i.e. where every item is in one and only one cluster, or is a singleton), one must select a horizontal slice from this tree (Figure 24).

In order to extract a partition from the hierarchy (i.e. a grouping of compounds where every compound is in one and only one cluster), we need to select a 'level' from this hierarchy. This can be done intuitively or by using some algorithms.

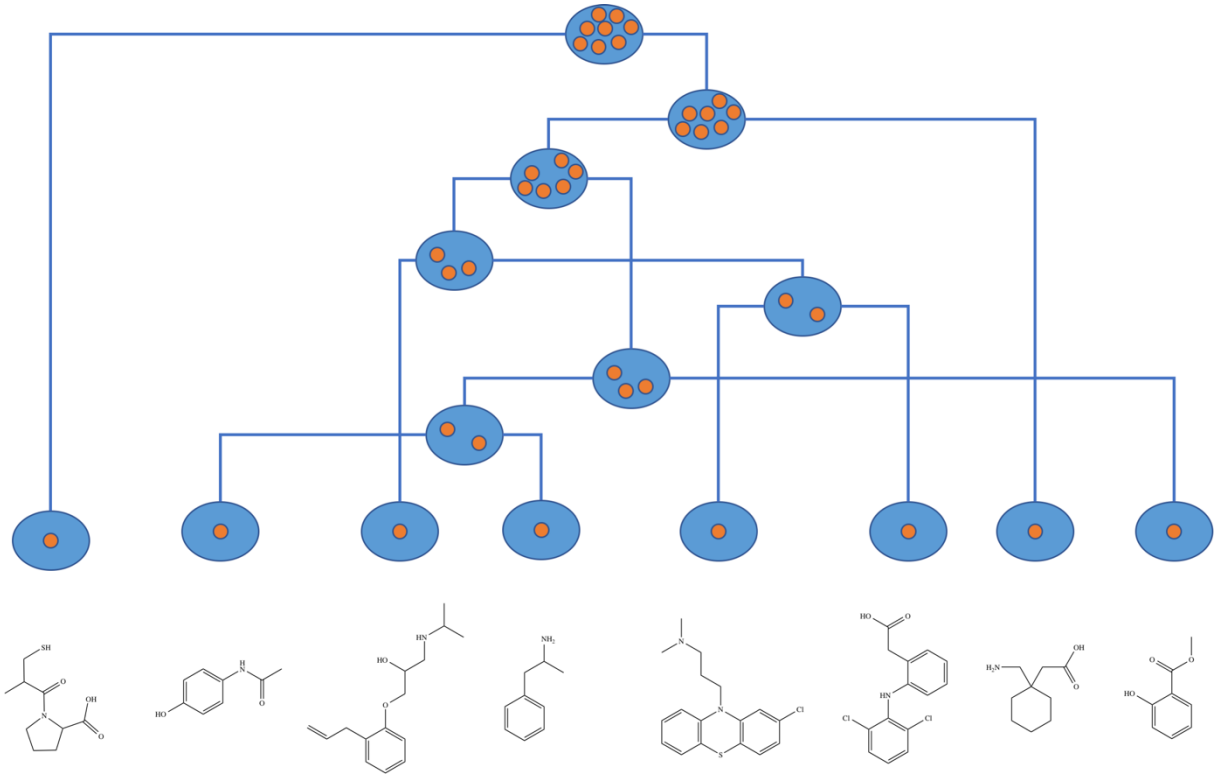


Figure 24. Hierarchical clustering work by initially putting every item in a cluster by itself, at the bottom level (with n clusters, where n is the number of points). It then identifies the two clusters to merge (depending on the methods as described above) and merges them to form a new cluster at the next level. The next level up will therefore consist of one cluster with two points, and all the rest of the points in clusters by themselves (i.e. there will be n-1 clusters). The process repeats until there is just one cluster at the top containing all the points, resulting in a cluster hierarchy.

4.2. Non-hierarchical clustering

Non-hierarchical methods can use a variety of algorithms, but they generally all produce a single partitioning of the dataset into clusters (versus a tree which can result in many partitions). Some of the more common non-hierarchical methods are Jarvis-Patrick, *K*-means and *K*-medoids.

1) **Jarvis-Patrick (JP)** is a non-hierarchical method where, for each compound in a set, the *j* nearest neighbours (that is the *j* other compounds in the dataset that are the most similar) are identified. Two compounds cluster

together if they 1) are in each other's list of *j* nearest neighbours, and 2) have k_{min} of their *j* nearest neighbours in common (Figure 25). This method doesn't require level selection, but does require *j* and k_{min} to be predefined. For example, one might choose a nearest neighbour list of size 20 (*j*) and put molecules in the same cluster if they share more than 10 nearest neighbours (k_{min}). Depending on the types of fingerprints used, Tanimoto is usually used as the measure of similarity. JP is fast, but has had mixed results in cheminformatics in terms of quality.

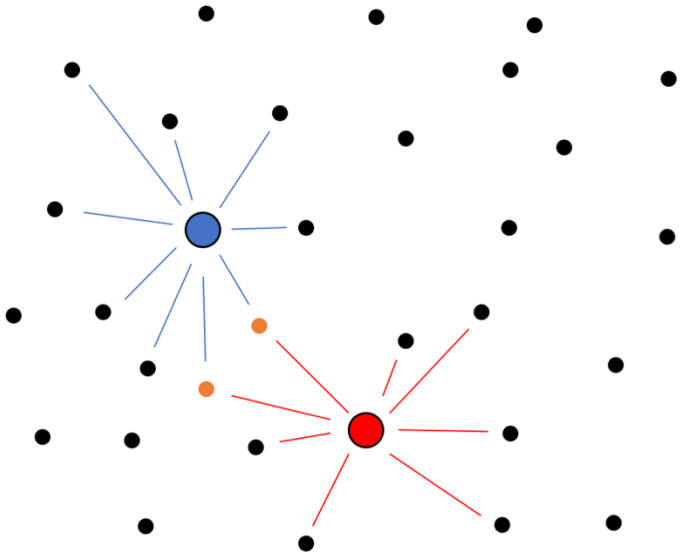


Figure 25. Illustration of the Jarvis-Patrick algorithm. Each dot represents a molecule and the distance between the dots corresponds to the Tanimoto similarity between the compounds (larger distance, lower similarity). For each of the blue and red compounds, the j=8 nearest neighbors are shown using lines. The two compounds that are shared by both nearest neighbor lists are shown in orange. The blue and red compound would cluster together if k_{min} would be set to 1 or 2, but not if $k_{min} > 2$.

2) ***K*-means** clustering is more widely used than JP. It requires that the number of desired clusters *k* be known in advance. The algorithm proceeds by alternating between these steps:

- Assignment step: assign each observation to the cluster whose mean is nearest;
- Update step: calculate the new cluster means to be the centroids of all molecules in the cluster.

The algorithm has converged when the assignments no longer change, however there is no guarantee that the optimum is found using this algorithm and the result may depend on the initial clusters. Generally, only a few (<100, often <10) iterations are required to settle (Figure 26).

The assignment of the initial *k* cluster centroids is normally done in a *random fashion*. However, alternative methods exist, such as the *Random Partition method* in which each molecule is initially assigned random to one of the *k* clusters, and then calculating the mean from each of the cluster's randomly assigned points to get the initial cluster centres. A consequence of the Random Partition method is that the initial cluster centres are all close to the centre of the dataset (Figure 27).

3) ***K*-medoids** is a derivative of K-means, but differs from *K*-means in that it uses real compounds, or 'medoids', to represent cluster centres, rather than centroids.

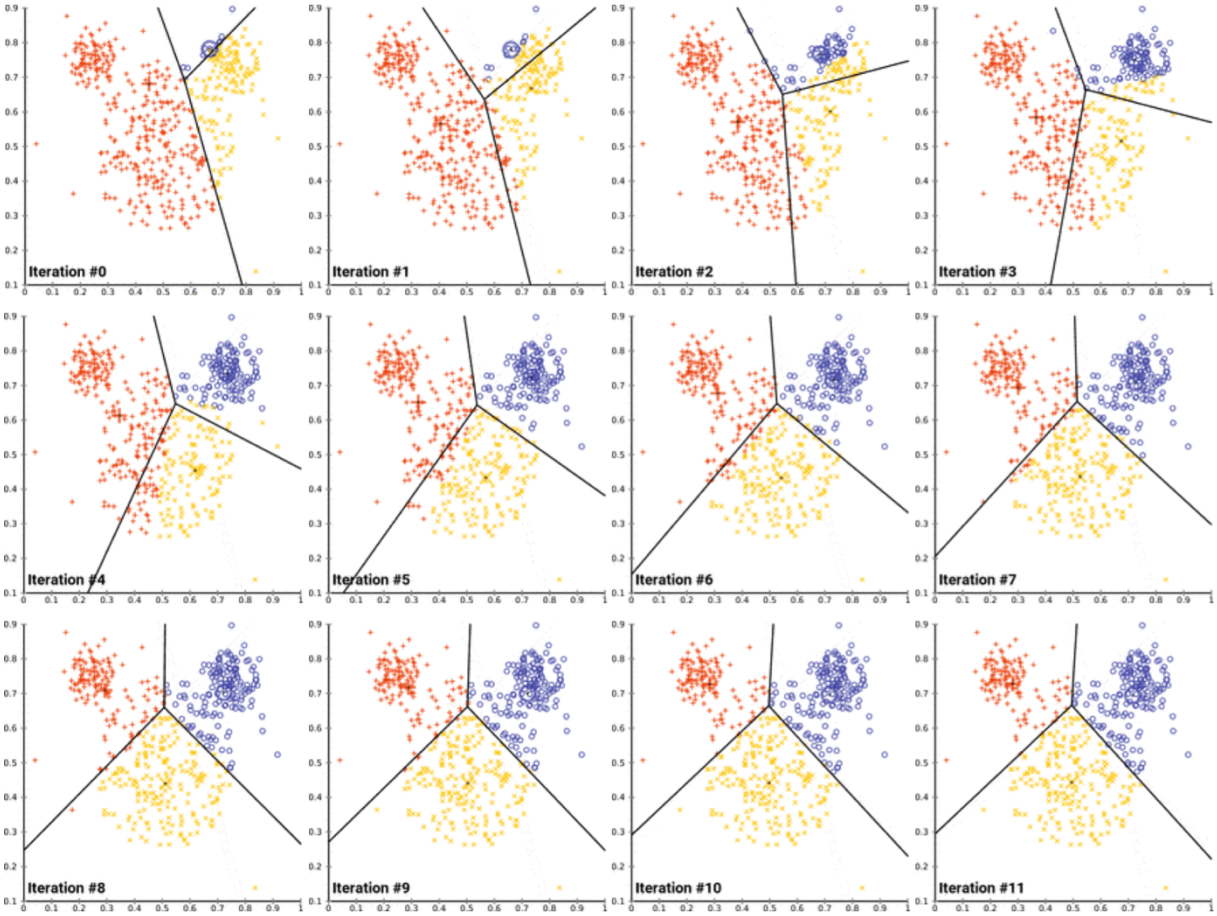


Figure 26. Illustrating the iterative process of the k-means clustering algorithm (in this case, k=3). The centroids of each cluster are shown as black dots.

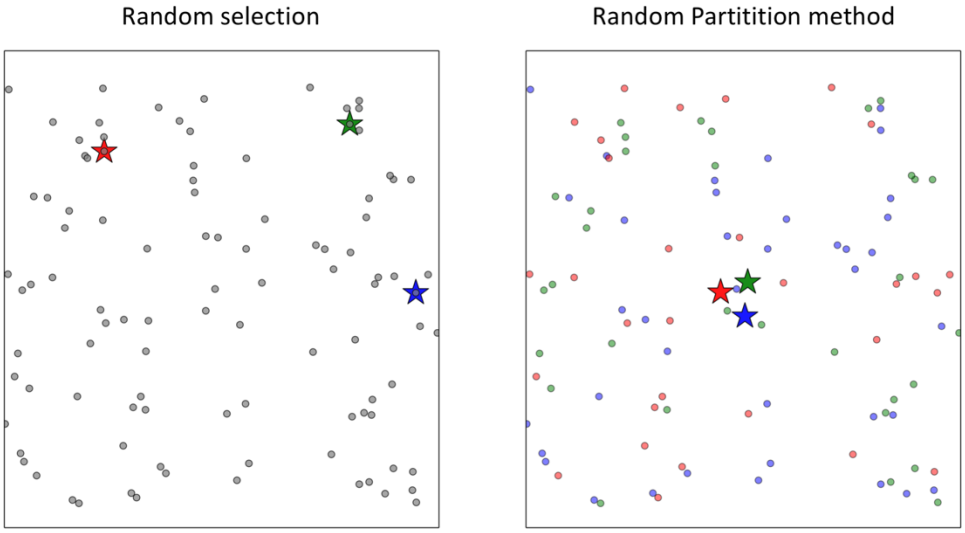


Figure 27. Initialization step of the k-means clustering method. The random selection method is nothing more than a random selection of k cluster centers from the dataset, while the 'Random Partition' method assigns each molecule initially random to one of the k *clusters, and then calculating the mean from each of the cluster's randomly assigned points to get the initial cluster centres.*

4.3. Self-organising maps (SOM) or Kohonen networks

SOM's are a type of artificial neural networks that are trained using unsupervised learning to produce a two-dimensional and discretised representation of the input molecules. The artificial neural network introduced by the Finnish professor Kohonen in the 1980s and is therefore sometimes called a Kohonen map or network. Kohonen

maps can be used in conjunction with spectrophore fingerprints, which makes it an attractive method to cluster compounds based on their spectrophore similarities (Figure 28).

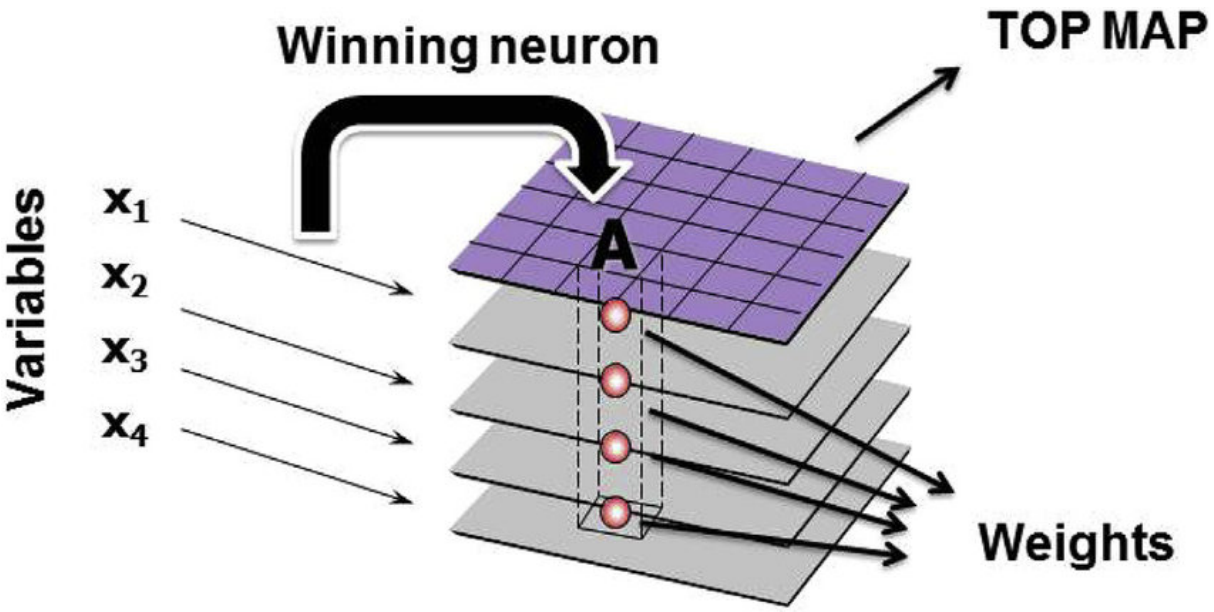


Figure 28. Illustration of a SOM network consisting of 6 x 6 cells, each represented by a weight vector of size 4. Adapted from https://goo.gl/images/15mSAo .

- Initialisation phase. The user needs to define the desired map size. In most cases, a grid of 10 by 10 cells is often sufficient, but other size may be tested as well. Each grid cell is composed of a vector of 48 values (the size of the spectrophore vector), and all values of these 100 vectors are initially assigned with random numbers.
- Step 1. From the database of compounds that need to be clustered, an input molecule (spectrophore) is selected.
- Step 2. The Euclidean distance between the input spectrophore and each of the 100 cells is calculated, after which the cell with the smallest distance is selected.
- Step 3. The 48 values of the selected cell are updated according a user-defined update function. Many flavours exist for this function, but a good start is to use an average function (each of the values in the cell are assigned a new value which is nothing more than the average between the old value and the corresponding value of the input spectrophore).
- Step 4. Stop if the maximum number of iterations has been reached, otherwise continue with step 1.

5. Diversity analysis

Diversity analysis gained popularity in the late 1990's in response to the following needs in the pharmaceutical industry:

- There was much interest as to how well the corporate collections of compounds held by pharmaceutical companies 'covered' possible chemistry/drug space (Figure 29).
- Combinatorial chemistry experiments were producing many new compounds, and people wanted to know if these compounds added anything new (in terms of chemical or biological functionality) to their corporate collections, i.e. if they made the datasets more diverse, or just replicated what was already in there?
- Libraries of thousands of compounds became available for purchase – are they worth the money?

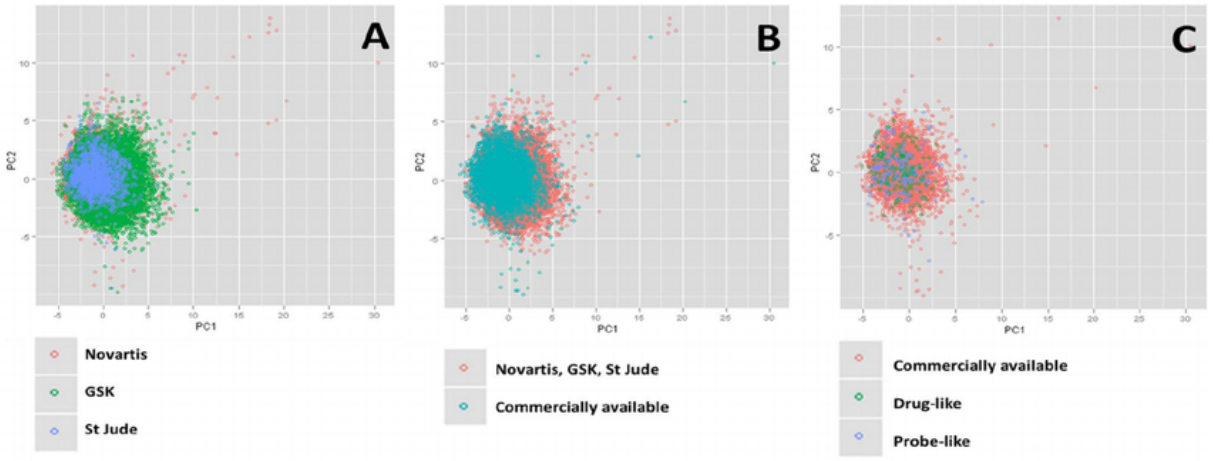


Figure 29. Principal Component Analysis plots. Chemical diversity of the GSK, Novartis and St-Jude libraries displayed (panel A); Overlap in chemical diversity of the combined datasets and the commercially available compounds (panel B); Overlap in chemical diversity of the commercially available compounds where the drug-like and probe-like chemotypes were annotated (panel C). Adapted from reference 5.

6. Excercises

6.1. Exercise 1

Write a clustering algorithm that reads as input a sdf-file of compounds, calculates the Daylight fingerprints from the molecules, and performs a clustering on these molecules to retain 50 different clusters. Print, for each cluster, the names of the molecules within the cluster, and print also the corresponding SMILES. Hint: sdf-files can be downloaded from different vendors, see for example the company Life Chemicals. Alternatively, you can use the `compounds_10k.smi` or `compounds_100k.smi` databases from the UAMC Githib repository.

6.2. Exercise 2

Write a program that takes the SMILES of a query molecule, reads a sdf-file of compounds, and then calculates the Tanimoto similarity of the query molecule against each of the database molecules. Plot a histogram of the calculated similarity indices, and writes out the 10 most similar compounds in SMILES format. For an example of database molecules, see for example the company Life Chemicals. Alternatively, you can use the `compounds_10k.smi` or `compounds_100k.smi` databases from the UAMC Githib repository.

6.3. Exercise 3

Write a program that performs a self-organising maps clustering of a set of molecules. For this purpose, create a 10x10 map, leading to 100 clusters. For an example of database molecules, see for example the company Life Chemicals. Alternatively, you can use the `compounds_10k.smi` or `compounds_100k.smi` databases from the UAMC Githib repository.

Chapter 5. Quantitative structure-activity relationship models

1. Some general concepts

1.1. Model building strategy

Apart from similarity searches, diversity analysis and clustering, chemo-informatics approaches are often used to develop sophisticated models that are capable to predict biological activities for molecules that have never been tested before (QSAR: quantitative structure-activity relationship). The methods that underly these models are called "machine learning" methods, and may vary from simple linear regression approaches up to the more sophisticated deep-learning and so-called "artificial intelligence" methods.

The purpose of each developed model is to predict, given a molecular structure as input, some kind of property from that molecular structure. If this property is a biological activity, then the model is called a QSAR model. If the property is rather a physicochemical property, such as aqueous solubility, then one talks about QSPR models (quantitative strutucre-property relationship). In order to build a QSAR or QSPR model, each model has to be trained using some existing property data of interest, and the corresponding molecular structures. A model building process generally consists in three stages: 1) *model generation*, 2) *model validation*, and 3) *production run* (Figure 30).

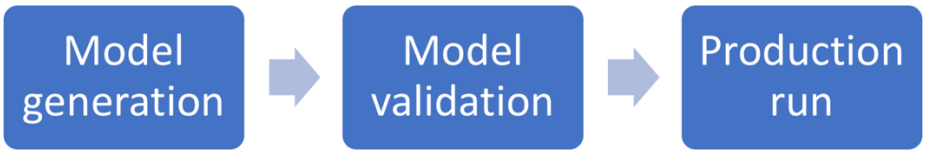


Figure 30. The three stages in a model building experiment.

Once a suitable model has been generated, its predictive power and quality should be validated. This validation is done in the second phase, which is called the validation phase. If it turns out that the quality of the model is not sufficient enough, then a new model has to be generated and again validated, until the final model reaches the desired quality. In this case, the actual production stage may start. Validation is a crucial aspect of any virtual screening process. It is the process by which the reliability and relevance of a procedure are established for a specific purpose. The different validation methods and metrics will be discussed in the following sections.

A QSAR model is nothing more than a mathematical description that links whatever kind of chemical descriptor (such as a molecular fingerprint) to the desired property that one want to model. To illustrate this concept, consider a simple linear regression model in which property *y* (for example the molecular lipophilicity) is linked to descriptor *x* (for example the molecular weight of the molecule):

$$y = ax + b$$

In order to be able to link *y* to *x*, coefficients *a* and *b* need to be known, and this is achieved by training the linear regression equation with a set of trainingsdata that contain many y and x as examples. Using simple mathematics, it is then possible to derive *a* and b in order to generate a linear regression model.

This example may be somewhat too simple in real situations, but it illustrates well the concept of model building within the world of chemo-informatics and computational drug design. In practice, one is often interested in the prediction of biological activities (*y*) using chemical fingerprints as input descriptor (*x*). Instead of the simple linear regression model as given in this example, more sophisticated methods are used to correlated *y* to *x*. In this section, we will cover some of these methods that are used to build models, but we will start by describing the basics that are needed to validate the quality of these models using performance metrics.

1.2. Classification and regression models

A model can either indicate whether a molecule is predicted to be 'active' or 'inactive' (1 versus 0), or it can return a predicted binding affinity value as a real number. In the former case, the returned data type is binary (or class), while in the latter case the returned data type is continuous (Figure 31). Pharmacophore searches (see later) are often binary classification methods, as the molecule may fit the pharmacophore or not. Docking screens on the contrary often return a scoring value and are therefore continuous models.

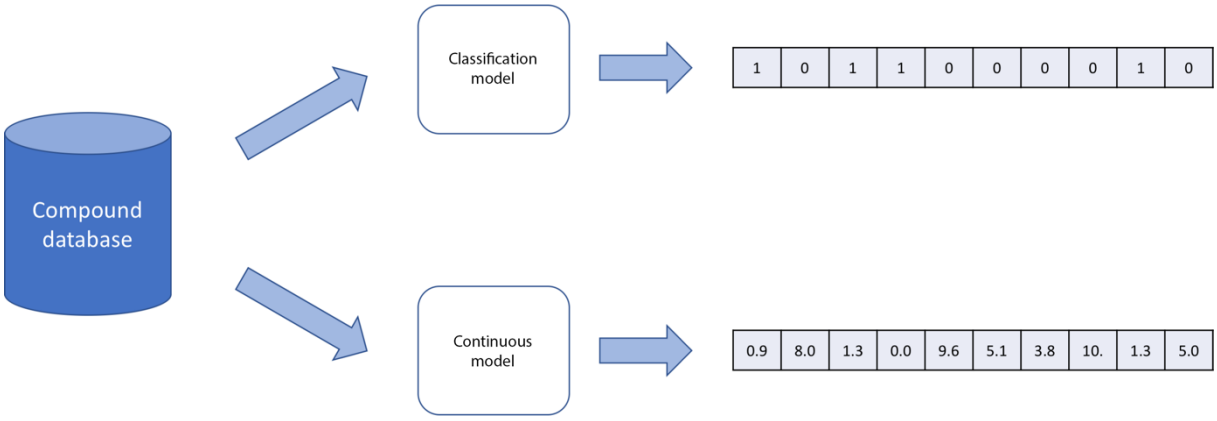


Figure 31. Two types of data generated by virtual screening methods: classification models (binary models) produce binary data ('active' or 'inactive', '1' or '0'), while continuous models produce continuous data (for example the value calculated by a docking scoring function, or the biological activity expressed in pIC_{50} units).

When the model performs a classification, one calls this model a classification model. Such a model is able to predict the class of the unknown datapoint (for example a new molecule) to which this datapoint belongs (for example, class "ACTIVE" or class "INACTIVE"). On the other side of the spectrum, regression models are able to predict a continuous property value for the unknown datapoint, such as a biological activity or aqueous solubility.

1.3. Supervised and unsupervised learning

Within machine learning, there are two basic approaches: supervised learning and unsupervised learning. The main difference is that in supervised learning one uses labeled data to build models, while this is not the case for unsupervised learning.

Supervised learning

Supervised learning is a machine learning approach that is defined by its use of labeled datasets. In supervised learning, the model is trained using data which is labeled. It means some data is already tagged with the correct answer. It can be compared to learning which takes place in the presence of a supervisor or a teacher. As an example of such a labeled dataset, consider a set of compounds of which the biological activities for a given target is known. In this case, the compounds can be considered to be "labeled" with these affinity data, and the model can be trained to predict the affinities of novel compounds.

Supervised learning can be separated into two types of problems when data mining: classification and regression:

- *Classification* problems use an algorithm to accurately assign test data into specific categories, such as separating active compounds from inactive ones. Classification problems are solved using classification models (see the section before). Linear classifiers, support vector machines, decision trees, neural networks and random forest are all examples of classification algorithms. In this course, we will mainly focus on decision trees and random forest methods.
- *Regression* is another type of supervised learning method that models the relationship between the dependent (the observation one wants to predict, such as the biological activity) and independent variables (for example the molecular fingerprint bits). Regression problems are solved using regression models. Such regression models are helpful for predicting numerical values based on different data

points. Some popular regression algorithms are linear regression, some flavors of neural networks, logistic regression and polynomial regression. In this course, we will mainly examplify linear regression.

Unsupervised learning

Unsupervised learning uses machine learning algorithms to analyze and cluster unlabeled data sets. These algorithms discover hidden patterns in the data itself without any assigned labels.

Unsupervised learning models are used for two main tasks: clustering and dimensionality reduction.

- *Clustering* is a data mining technique for grouping unlabeled data based on their similarities or differences. For example, *K*-means clustering algorithms assign similar data points into groups, where the *K* value represents the size of the grouping and granularity. In this course, we have already covered hierarchical and non-hierarchical clustering methods (see before).
- *Dimensionality reduction* is a learning technique used when the number of features (or dimensions) in a given dataset is too high. It reduces the number of data inputs to a manageable size while also preserving the data integrity. Often, this technique is used in the preprocessing data stage, hence before the actual model building step takes place. A typical example of dimensionality reduction is principle component analysis (PCA). In this course, we will not cover any of the dimensionality reduction methods.

1.4. Descriptive and predictive analytics

Apart from the model types and learning methods, we can distinguish between two types of data analytics:

- Descriptive analytics
- Predictive analytics

Descriptive analytics

In descriptive analytics, one tries to answer the question of 'what happened?' In descriptive analytics, one looks at the data and tries to obtain a detailed description of the data itself. In descriptive analytics one mainly looks at the variables separately. Example of descriptive analytics algorithms are the creation of tables with frequencies or percentages of the different values or groups of values, summary statistics like the mean (for continuous variables) or the median. These two well known summary statistics tell us something about the "middle" of the data, but there are also summary statistics regarding the spread (i.e. are different data points close to each other in value or not) like the variance or interquartile range. Graphical representations of the data play a key role as they give insight into the complete distribution of the variable. These are often closely related to the summary statistics, like in the case of a boxplot. The most commonly known graphical representations are bar charts for all non-continuous data and histograms for the continuous variables. Special interest goes to two odd kinds of values. There often are outliers, cases which are so far removed from the rest of the data that they might be mistakes. If not properly identified if these values are indeed valid they can have a disproportionately big impact on all further analyses. Apart from outliers one also needs to look out for missing values. In case of missing values, the issue must be resolved before one can continue with the analysis. In any given case missing values should not plainly be converted to a zero as this would of course result in incorrect statistics and conclusions.

Predictive analytics

Prediction is main focus of all methods that are covered in this course, as it tells us what would happen given new data based on data from the past. In the case of computational drug design, models are derived that allows the researcher to look for other compounds with the desired property. Linear regression is already a form of prediction. Confidence intervals are often used here as they give an expected range rather than only a single value.

2. Building QSAR models

There exist many machine learning approaches that can be used to generate models with. Many machine learning approaches can be used to generate both classification or continuous models. In this section, a number of the more commonly used methods are presented and their use explained. We will not focus on the mathematical implementations; rather we will highlight some important features of each of the methods and explain the advantages and disadvantages.

For all machine learning methods in this course, the common and open source package scikit-learn is used to demonstrate the concepts of QSAR model building. It is by default available from within Google Colab, so there is no need for additional installations unless you need scikit-learn on your own local system. In that case, installation instructions are provided on the scikit-learn website.

In order to be able to build machine learning models, there are two important data sources that should be available:

- The molecules in a computer-readable format. Often, standard 1024- or 20480-bit fingerprints are used for this, but other representations, such as the logP, molecular weight, number of atoms, are also possible.
- The data that should be modeled. In the case of building QSAR models, this is often some kind of biological activity that one wants to model (for example, represented as IC_{50} or pIC_{50} values). In the case of QSPR models, this can be the physicochemical property such as the to-be-modeled aqeous solubility (for example, expressed in g/L).

The combination of molecules and corresponding data is called the "training set". The quality of this training set is of utmost importance for the quality of the generated model. When the datapoints (for example, the biological activities) are of lesser quality, then the corresponding model will also suffer from lesser quality.

Molecular structures cannot be used by the computer as such. Rather, they need to be converted in a format that the computer can understand. Very often, standard fingerprints, such as the standard Daylight-, Morgan-fingerprint or MACCS-keys are used. Once all molecules are converted in such a representation, this training set is then fed into the machine learning application to calculate a model.

As an example of a typical training set, consider the following snippet from a file with compounds that were tested on their potential DPP4-inhibition capabilities:

```
ACTIVE      CN[C@@H](C1CCC(CC1)NS(=O)(=O)c2ccc(F)cc2F)C(=O)N3CC[C@H](F)C3
ACTIVE      CN(C)C(=O)[C@H]([C@H](N)C(=O)N1CC[C@H](F)C1)C2CCC(CC2)N(C)C(=O)c3cccc(F)c3
ACTIVE      C[C@@H](B(O)O)N1CC[C@H](N)C1=O
ACTIVE      CN(C)C(=O)[C@H]([C@H](N)C(=O)N1CC[C@H](F)C1)C2CCC(CC2)NS(=O)(=O)c3ccc(F)cc3F
ACTIVE      CN(C)C(=O)[C@H]([C@H](N)C(=O)N1CC[C@H](F)C1)c2ccc(cc2)N(C)C(=O)c3ccc(F)cc3
...
INACTIVE    Cn1nnnc1SCc1ccc(C(=O)Nc2ccccc2F)cc1
INACTIVE    O=C1/C(=C/c2ccccc2Br)Oc2cc(OCc3ccc(F)cc3)ccc21
INACTIVE    Cc1nn(C(C)C(=O)NCc2ccccc2Cl)c2ccccc12
INACTIVE    CCCOc1ccccc1C(=O)Nc1ccc(NC(=O)C(C)C)cc1
INACTIVE    OCCNc1nc(Nc2ccccc2O)nc2ccccc12
...
```

The first column indicates whether the compound is an inhibitor of DPP4 ("`ACTIVE`" when $pIC_{50} > 4$), or not ("`INACTIVE`"). Each compound is represented by its SMILES representation. As this dataset contains only `ACTIVE` or `INACTIVE` as classes, it can be used to build classification models but it is not suitable to build continuous models from. Other datasets can include the actual pIC_{50} values as property, and these datasets can then be used to build regression models.

2.1. Linear regression models

In linear regression, the target value (*e.g.* the property that needs to be predicted) is modeled as a linear combination of the molecular features (*e.g.* the individual fingerprint bits). Linear regression is probably the most simple of all machine learning models, but it offers interpretability and ease of calculation.

Linear regression fits a linear model with coefficients $w = (w_1,\ldots,w_p)$ to minimize the residual sum of squares between the observed targets (properties) in the dataset, and the targets predicted by the linear approximation. Mathematically it solves a problem of the form:

$$\min_w \|Xw - y\|_2^2$$

as it is common to use the sum of squared errors $\|\varepsilon\|_2^2$ as a measure to minimize.

Good practices require the use of a training set and a test set. This is done by randomly diving the entire dataset into these two sets using a certain fraction, for example 70% of the dataset becomes the training set and the remaining 30% becomes the test set. The model is then trained using the data in the training set, and the resulting model is subsequently validated using the test set. A common metric to evaluate a linear regression model is the mean squared error of the test set, defined as:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$$

with n the number of datapoints in the test set, $\hat{Y}_i$ the predicted value of datapoint *i*, and Y_i the real value of datapoint *i*. In other words, the *MSE* is the mean of the squares of the differences between real values and predictions. If this difference is close to 0, then the *MSE* will also be close to 0.

A second obvious validation of the generated model is the visual inspection of the plot that scatters the predicted values against the real values. There should be a clear correlation between both, as is shown in Figure 32. If this correlation is absent, then the generated model has no predictive value and is of little use in a typical QSAR setting.

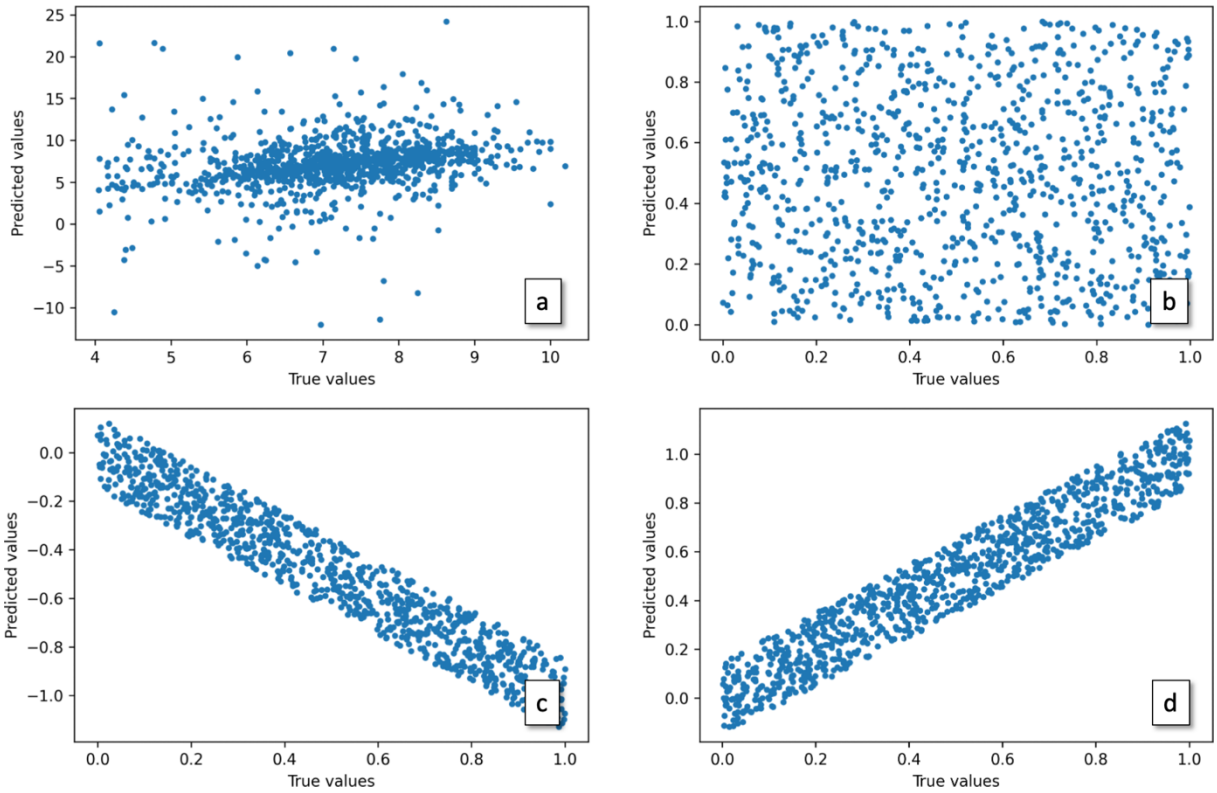


Figure 32. Examples of different correlation plots. (a) Example of a weak correlation with large variations in the predicted values. (b) No correlation at all. (c) Negative but good correlation between the true and predicted datapoints, indicating a good predictive model. (d) Positive and good correlation between the true and predicted values.

To illustrate the process of QSAR model building, we will consider an example in which the inhibitory activity of chemical compounds against the dipeptidyl-peptidase IV (DPP4) is modeled using a linear regression model. The ChEMBL database was used to extract all small molecule compounds having biological activity data against DPP4 (stored as pIC_{50} values). In total, 3,858 of such compounds were retrieved, and their chemical structures (stored as SMILES strings) were converted into Daylight fingerprints consisting of 2,048 bits per fingerprint:

```
url = "https://raw.githubusercontent.com/UAMCAntwerpen/2040FBDBIC/main/dpp4.pIC50.txt"
data = requests.get(url).text.split("\n")
fps = []
pic50 = []
for d in data:
  fields = d.split()
  if len(fields) < 1: continue
  pic50.append(float(fields[1]))
  mol = Chem.MolFromSmiles(fields[0])
  fp = np.zeros((0,), dtype=np.int8)
  DataStructs.ConvertToNumpyArray(Chem.RDKFingerprint(mol), fp)
  fps.append(fp)
```

Prior to building a model, the dataset is first split into a training set (70% of the compounds) and a test set (30%). The training set is used to train the model, while the test set is kept aside and used later on to validate the generated model:

```
pic50_train, pic50_test, fps_train, fps_test = train_test_split(pic50, fps, test_size=0.3)
model = linear_model.LinearRegression()
model.fit(fps_train, pic50_train)
```

Finally, the generated linear regression model is validated with the test set. For this purpose, the generated model is applied on the fingerprints of the test set, and the hence generated pIC_{50} values (the predictions) are compared with the true pIC_{50} values as given by the test set. This comparison is done with the *MSE* metric:

```
pic50_pred = model.predict(fps_test)
print("MSE = ", mean_squared_error(pic50_test, pic50_pred))
```

The calculated *MSE* is 9.3, meaning that the average squared difference between the real and predicted pIC_{50} values is around 9.3. Given that the pIC_{50} values in the dataset range between 4-10, a *MSE* of 9.3 is therefore a large error, indicating a bad predictive model. This is also obvious when plotting the predicted valaues against the corresponding real values:

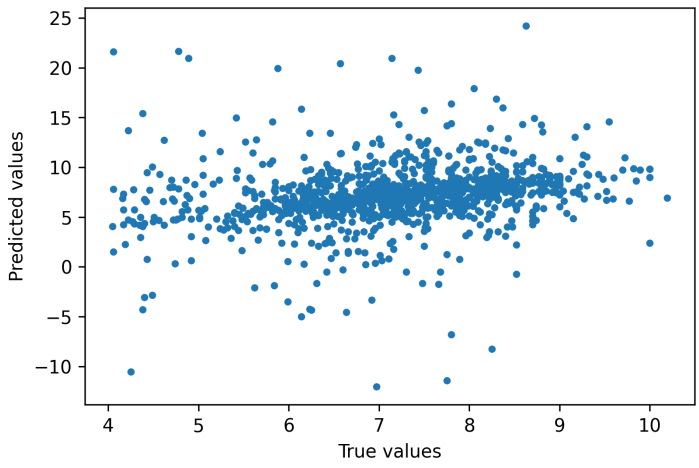


Figure 33. Correlation plot between the true and predicted pIC50 values of a linear regression model of DPP4 inhibition. The lack of obvious correlation is indicative of a bad predictive model, also confirmed by the MSE which is around 9.3.

Linear regression models are models of continuous datasets, like the prediction of biological activity data such as IC_{50} values

Linear regression models are typically used to predict numerical properties such as biological activities or physicochemical properties. However, **logistic regression** is a regression analysis technique that can be applied on **classification problems**. Please consult this Youtube video if you want to learn more about this useful technique in machine learning (8'47" in length).

2.2. Neural network models

Neural networks form the basis of deep learning and artificial intelligence approaches. It has undergone a huge transformation with the recent availability of powerful graphical processing units (GPU's) which are optimally suited to perform calculations needed for neural networks. Since deep learning and neural networks are available in many flavors, going from recursive deep learning, convolutional deep learning and reinforcement deep learning, we will only very briefly explain the basic concepts of neural networks in this section.

A neuron, in the context of neural networks, is a fancy name for a "function". A function takes some input *x*, applies some logic *f()*, and outputs the result *f(x)*:

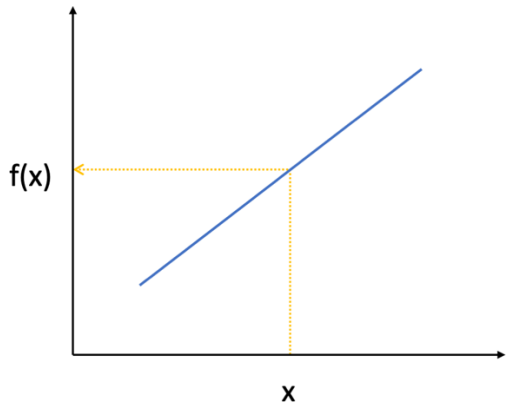


Figure 34. A neuron represented as a simple mathematical function, taking some input x and returning a result f(x).

Hence, since that a neuron is nothing more than a function, a neural network is nothing more than a network of functions:

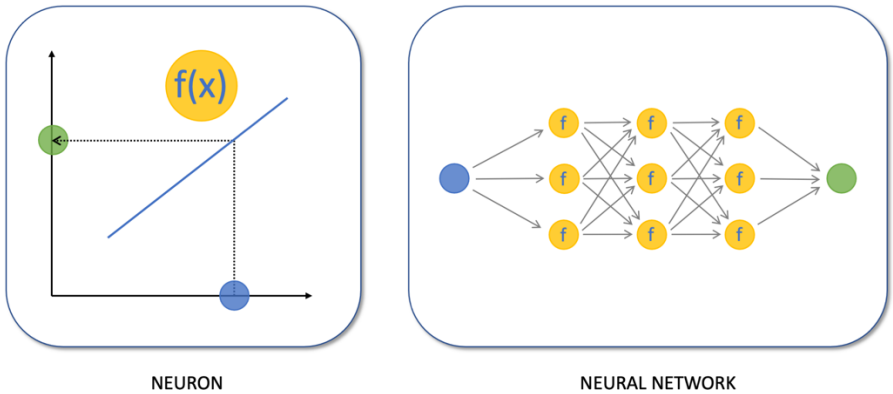


Figure 35. Relation between a neuron and a neural network. A neural network consists of a network of functions (neurons), each function is linked by its input and output to the other functions. Neural networks mainly differ in 1) the arrangement (connectivities) between the functions and 2) the type of function f(x).

As can be seen from Figure 35, each neuron in the neural network is connected to a number of other neurons. Each neuron in a given column of neurons receives its input from each of the neurons in the column on the left side, and sends its output, after transformation with f(x), to each neuron of the column on its right side. In Figure

36, the connection between a single neuron and its *n* preceding neurons is highlighted. The figure depicts a neuron connected with *n* other neurons and thus receives *n* inputs (x_1, x_2, x_3, …, x_n). Each input signal is weighted with a specific weight for that particular connection. This configuration is called a perceptron. All the inputs are individually weighted, added together and passed into the activation function. There are many different types of activation functions but one of the simplest is the step function. A step function will typically output 1 if the input is larger than a certain threshold, otherwise it will output a 0. The inputs (x_1, x_2, x_3, …, x_n) and weights (w_1, w_2, w_3, …, xw_n) are real numbers and can be positive or negative. Hence, a perceptron consists of weights, a summation processor and an activation function.

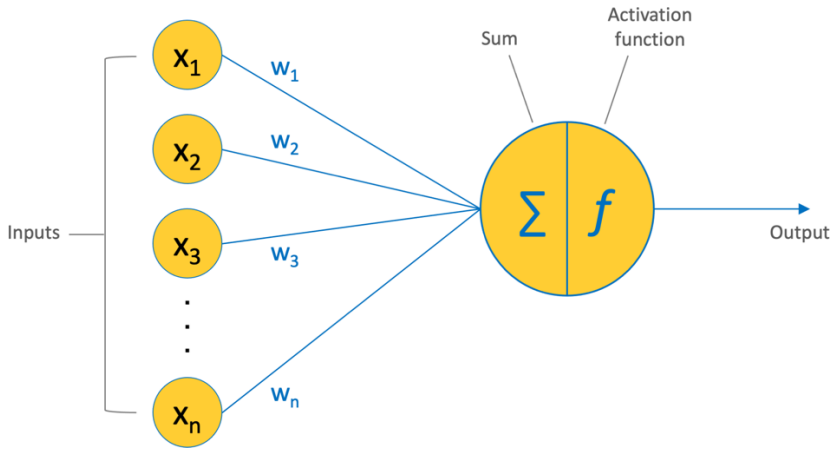


Figure 36. A single neuron connected to n other neurons. The neuron receives n inputs (x_1, x_2, x_3, …, x_n). Each input is weighted with a corresponding weight (w_1, w_2, w_3, …, w_n). Such a configuration is called a perceptron.

The training of a neural network is nothing more than the optimisation of the network so that the difference between the true and predicted values is minimised. Optimisation of the network includes the optimisation of the connectivities between the neurons (weights of the connections and the type of connectivities), the optimisation of the function parameters and the functions themselves, and other stuff which is depending on the flavor of deep learning network. How this learning process works is beyond the scope of this course, but to learn more about this Youtube video is more than worth looking at (length is 11'18").

Neural networks can be used both on continuous and classification datasets, like the prediction of numerical properties (for example, IC_{50}) or the prediction of a certain class (for example, "ACTIVE" versus "INACTIVE")

2.3. Random forest models

The random forest classifier is a useful machine learning tool that fits a number of decition tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. It is mainly used for classification applications, but there exist flavors of the random forest technology that can be used in regression applications as well. In this section, it will be explained how basic decision trees work, how individual decisions trees are combined to make a random forest, and ultimately discover why random forests are so good at what they do.

Decision trees as they are the building blocks of random forest models. The concept of a decision tree is quite straigthforward and is best illustrated using an example (Figure 37). The dataset consists of different shapes in different colors. There are two squares in red, five circles of which one is red and four are colored blue. As such, the features are 'shape' and 'color'. So how can we do this?

In first instance, color seems like a pretty obvious feature to split by as the majority of the shapes are colored blue. So we can use the question, 'Is color red?' to split our first node. You can think of a node in a tree as the point where the path splits into two - observations that meet the criteria go down the 'YES' branch and ones that don't

go down the 'NO' branch. At this point, the 'NO' branch (the blues) contains nothing but circles so we are done there, but our 'YES'branch can still be split further. This can be done based on the second feature and ask "Is it a square?" to make a second split. The two squares go down the 'YES' subbranch and the circle goes down the 'NO' subbranch and we are all done.

In this example, our decision tree was able to use the two features to split up the data perfectly. Obviously in real life our data will not be this clean but the logic that a decision tree employs remains the same. At each node, it will ask: "*what feature will allow me to split the observations at hand in a way that the resulting groups are as different from each other as possible (and the members of each resulting subgroup are as similar to each other as possible)?*"

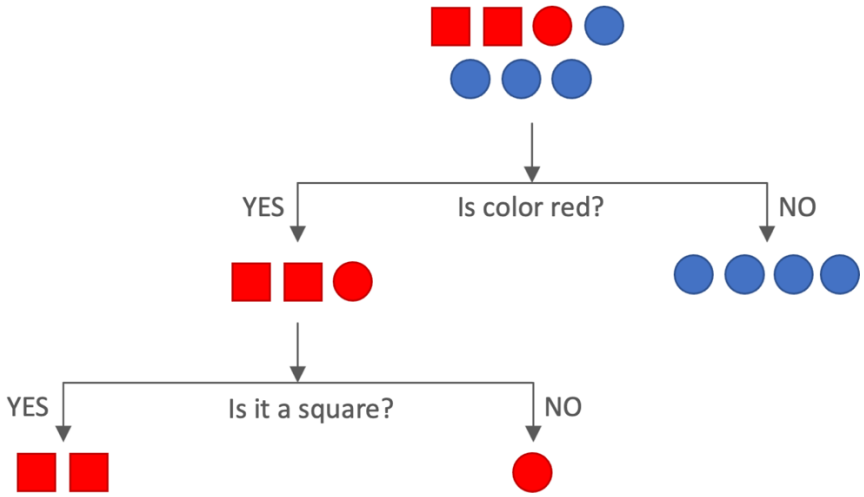


Figure 37. Example of a decision tree that was modeled to separate three different objects based on their shape and color.

Training (building) a decision tree comes down to finding the best cutoff parameters to separate each dataset onto two separate sets. Obviously, a decision tree only takes into account the provided parameters with each data sample. For example, in the case of molecules characterised by their 1024-bits fingerprints, a possible question could be: "Is the bit at position 245 a 1 or a 0?"

Random forests, like its name implies, consist of a large number of individual decision trees that operate as an ensemble. Each individual tree in the random forest spits out a class prediction and the class with the most votes becomes the prediction of the random forest (see Figure 38). The fundamental concept behind random forest is based on "the wisdom of crowds". In data science terms, the reason why the random forest models work so well is that a large number of uncorrelated models (the individual decision trees) that operate as an ensemble will outperform any of the individual constituent models.

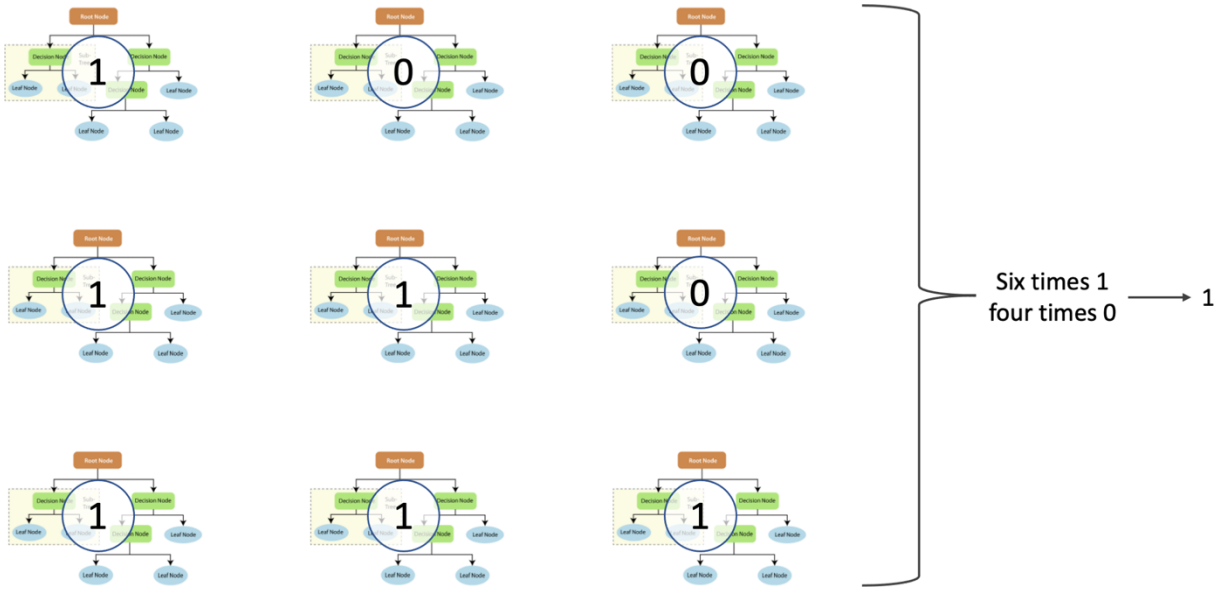


Figure 38. Illustration of a random forest model constituted out of nine uncorrelated decision trees. Six decision trees predict the value to be 1, and four trees predict a 1. Hence, the prediction made by the random forest is 1.

The low correlation between the individual models is the key concept behind the success of random forests. Uncorrelated models can produce ensemble predictions that are more accurate than any of the individual predictions. The reason for this effect is that the trees protect each other from their individual errors. While some trees may be wrong, many other trees will be right, so as a group the trees are able to move in the correct direction. So the prerequisites for random forest to perform well are:

- There needs to be some actual signal in our features so that models built using those features do better than random guessing.
- The predictions (and therefore the errors) made by the individual trees need to have low correlations with each other.

Random forest models ae most often used for classification problems, but there are random forest flavors that are designed to handle continuous datasets

---

3. Validation

3.1. Validation of classification models

There exist many metrics to define the quality of a model, and the actual choice of a metric depends on the question one wants to be answered. Sometimes one needs a model that is capable of selecting *all* active compounds that are contained in the database, but at the cost of having to test many compounds from that database. At other times, one might just be interested to identify only a few active compounds from the database, with the risk of missing out many others. In order to quantify these properties, a number of performance definitions and metrics have been described.

The confusion matrix

The confusion matrix is one of the most intuitive and easiest metrics used for finding the correctness and accuracy of the model. It is used for classification problems where the output can be of two or more types of classes, for example a compound is predicted by the model to be *active* (1) or *inactive* (0).[2]

The confusion matrix in itself is not a performance measure as such, but almost all of the performance metrics are based on confusion matrix and the numbers inside it (Figure 39).

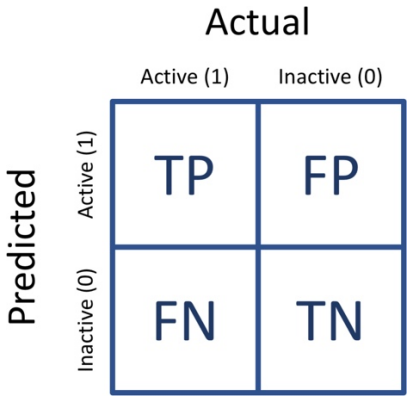


Figure 39. The confusion matrix is a table with two dimensions ('Actual' and 'Predicted'), and sets of 'classes' in both dimensions. The 'actual' classifications are columns and the 'predicted' ones are rows. TP: true positives, TN: true negatives, FP: false positives, FN: false negatives.

True positives (TP) – True positives are the compounds when the real measured biological activity of the compound is 'active' and the predicted activity also 'active'.

True negatives (TN) – True negatives are the cases when the real measured biological activity of the compound is 'inactive' and the predicted activity also 'inactive'.

False positives (FP) – False positives are compounds for which the measured biological activity is 'inactive' but the predicted activity is 'active'.

False negatives (TF) – False negatives are the cases for which the real biological activity is 'active' but the predicted activity is 'inactive'.

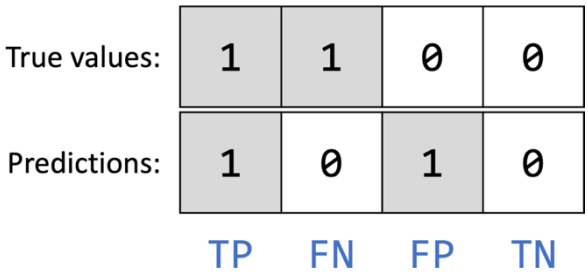


Figure 40. Illustration of the concepts of true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN).

The ideal model should give zero false positives and zero false negatives, but due to the inherent error in all models this goal is never achieved. We know that there will be some error associated with every model that is used for predicting the true class of the target compound. This will result in false positives and false negatives. There is no hard rule that says what should be minimized in all the situations, and depends on the business needs and the context of the problem one is trying to solve. Based on that, we want our model to minimize either the false positives or the false negatives:

1. Minimizing false positives. If one wants to mine a compound database for compounds that are active against a particular therapeutic target, but one has limited resources to physically purchase these

[2] In case of a continuous model such as the scoring function of a docking experiment, these continuous data first have to be transformed into binary data as explained later.

compounds so that only a limited number can be acquired and biologically tested, then one needs a model that minimizes false positives. By minimizing false positives, the researcher can be relatively 'sure' that any compound that is predicted by the model as 'active' is actually also active, hence limiting waste of financial resources on the purchase of inactive compounds. The drawback of a such model is that many 'active' compounds in the database will not be picked up, hence potentially missing out some real good active compounds.

2. Minimizing false negatives. A model that minimizes false positives will pick up 'all' active compounds from the database, however at the cost of having to experimentally screen a large number of compounds including many false positives. This model is useful if one expects that the database will not contain many active compounds and one wants to retrieve as much as possible of these active ones.

True positive rate: sensitivity or recall

The true positive rate (also called *sensitivity*, *recall*, *hit rate* or *power*) is the probability that an actual positive value will be predicted as being positive by the model. A model with a large *TPR* is capable of retrieving the majority of the real positives in the database (small number of *FN*), however at the cost of retreiving also many false positives (Figure 42a). In this context, a model that predicts all compounds to be active (irrespective whether they really are active) has a sensitivity of 1. As such a model is useless in practical applications, a second metric, in addition to the sensitivity metric, is warranted for model validation. An example of a metric useful to include could be the *FPR* (*vide infra*). By combining the *TPR* and *FPR* metrics one obtains the *accuracy* metric (*vide infra*): it optimizes the *TPR* and minimizes the *FPR* (Figure 41).

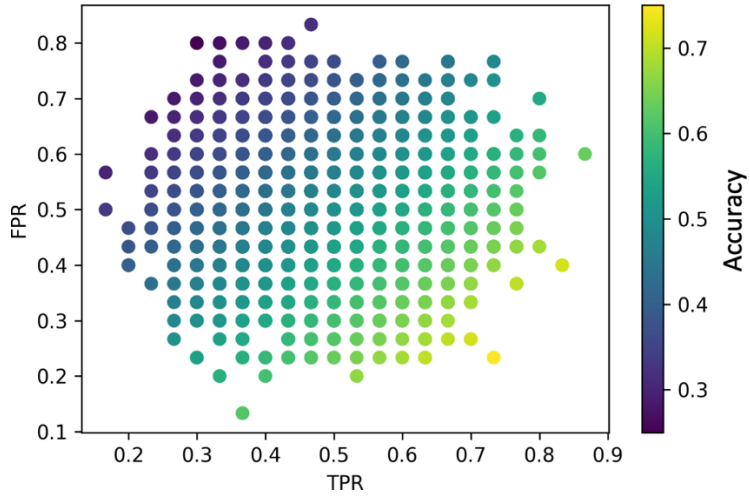


Figure 41. Relation between the true positive rate (TPR), false positive rate (FPR) and accuracy (color-coded; large accuracy value are yellow, small accuracy values are blue). Figure was created by generating 10,000 random models and calculating the appropriate metrics from these models. In a typical virtual screening application, a model with a large TPR and a small FPR is often desired (lower-right corner). Such models have a high accuracy (yellow dots).

The recall is a measure that tells what proportion of real active compounds are predicted by the algorithm as active. Since the real actives can be defined as *TP* and *FN*, and the compounds predicted by the model as actives equals to *TP*, sensitivity or recall can be written as:

$$Sensitivity = recall = \frac{TP}{TP + FN} = TPR$$

True negative rate: specificity

The true negative rate (*TNR*, also called *specificity*) is the probability that an actual negative value will be predicted as being negative. A model with a high specificity is capable of retrieving the majority of the real negatives in the database, however at the cost of retrieving also many false negatives (Figure 42b).

As specificity is the measure that tells what proportion of the compounds that were predicted as being inactives are really inactive, its value depends in the the actual negatives (equalling to *FP* + *TN*) and the compounds predicted to be inactive (equalling to *TN*). Hence, specificity becomes:

$$Specificity = \frac{TN}{TN + FP} = TNR$$

Specificity is the exact opposite of sensitivity. If a model is altered to increase sensitivity, then the specificity of that model will decrease, and *vice versa*.

False positive rate: fall-out

The false positive rate (*FPR*) is the probability that a false alarm will be generated by the model. Models that are characterised with a high *FPR* will suggest a large number of positive hits, while these predictions will prove to be negative once these are tested in the lab. The false positive rate is calculated from following equation:

$$Fall\ out = \frac{FP}{FP + TN} = FPR$$

The false positive rate can also be defined as:

$$FPR = 1 - specificity$$

False negative rate: miss rate

The false negative rate (*FNR*, also called the *miss rate*) is the probability that a true positive hit will be missed by the model. Hence, if it is crucial to retrieve all hits from the database, then a model with a small *FNR* should be used. The false negative rate is calculated from following equation:

$$FNR = \frac{FN}{FN + TP}$$

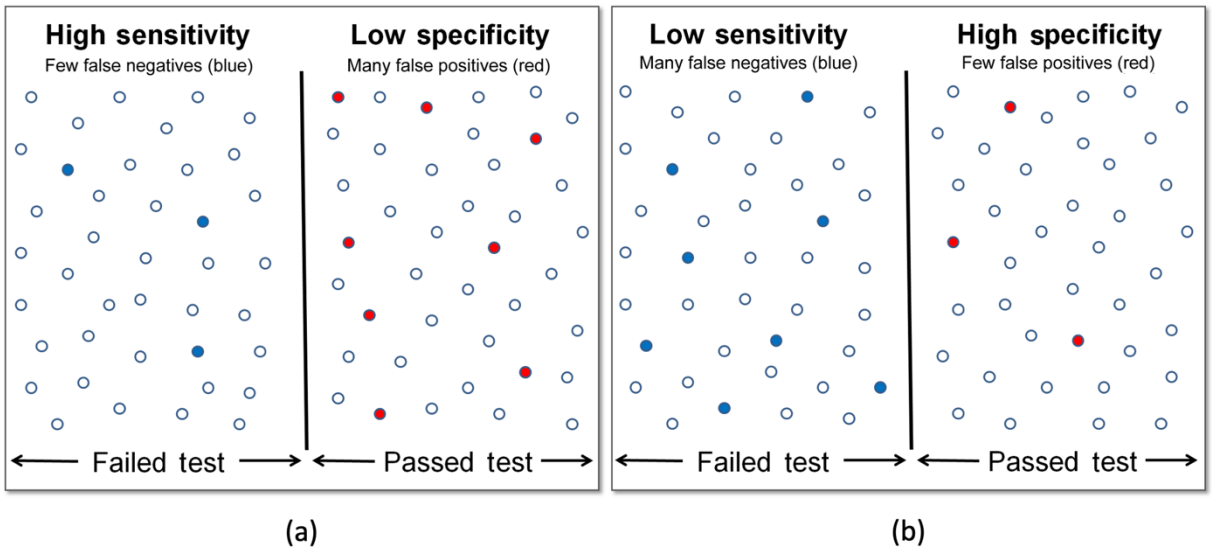


Figure 42. Trade-off between specificity (TNR) and sensitivity (TPR). Panel (a) shows the results for a model with high sensitivity and low specificity. This model will generate few false negatives at the cost of retrieving a large fraction of false positives. Panel (b) illustrates a model characterised by a low sensitivity and a high specificity. Such a model retrieves only few false positives but this comes at a cost of retrieving many false negatives. Figure adapted from https://commons.wikimedia.org/w/index.php?curid=14502690.

Accuracy and precision

An accurate test is a test in which systematic errors are reduced, while a precise test is a test in which random error are limited (Figure 43).

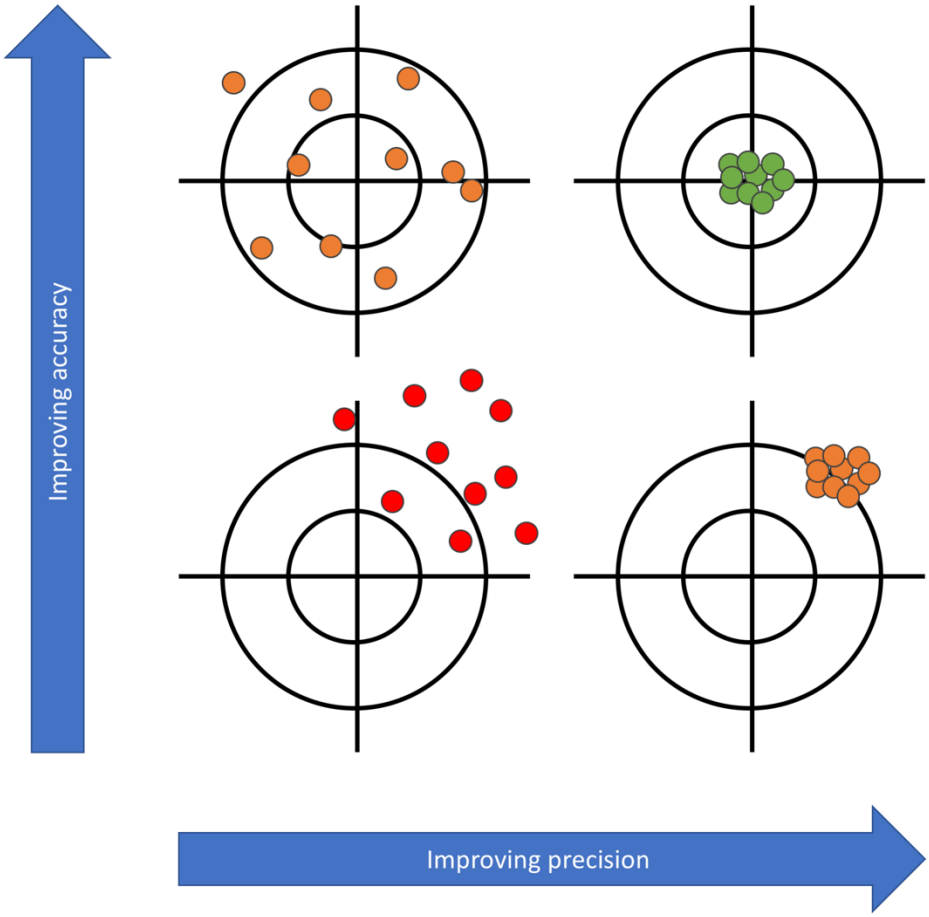


Figure 43. Schematic depiction of accuracy and precision. Precision is improved by decreasing random errors, while accuracy is improved by decreasing systematic errors.

In classification problems however, such as virtual screening or QSAR models, **accuracy** corresponds to the number of correct predictions (true positives and true negatives) made by the model over all kinds predictions made:

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

Accuracy is a good measure when the target variable classes in the data are nearly balanced, for example a compound database with 60% real actives and 40% real inactives. Given that high accuracies are indicative of models with a high TPR and a low FPR (see Figure 41), accuracy is a good metric to use in virtual screening experiments in which only limited budget is available to acquire novel compounds.

Precision in classification problems is a measure that tells what proportion of the predicted actives are real actives. The predicted actives that are real actives equals to *TP*, and the total number of actives in the database is *TP* + *FN*. Therefore, precision can be defined as:

$$Precision = \frac{TP}{TP + FP}$$

The sensitivity or recall gives information about a model's performance with respect to the false negatives (how many actives did we miss out), while precision gives information about its performance with respect to false positives (how many actives were retrieved from the database). Precision is about being precise: even if the database contained only one active compound, and if this active one was captured correctly, the method would be 100% precise. In contrast, recall is not so much about capturing cases correctly but more about capturing all actives. So basically, if we want to focus more on minimizing false negatives, recall should be as close to 100% as possible without precision being too bad. If one wants to minimize false positives, then focus should be to make precision as close to 100% as possible.

F1-score

The F1 -core is a single score that represents both precision and sensitivity in a single number:

$$F1\ score = \frac{2 * precision * recall}{precision + recall}$$

3.2. Validation of continuous models

In order to be able to use the different performance metrics (see the previous section), continuous data need to be converted into a set of classifications. This is done by ranking the set from 'good' to 'bad' (or the reverse), and then applying a cutoff to divide the set into 'good' and 'bad' data (or 'active' versus 'inactive' compounds) (Figure 44).

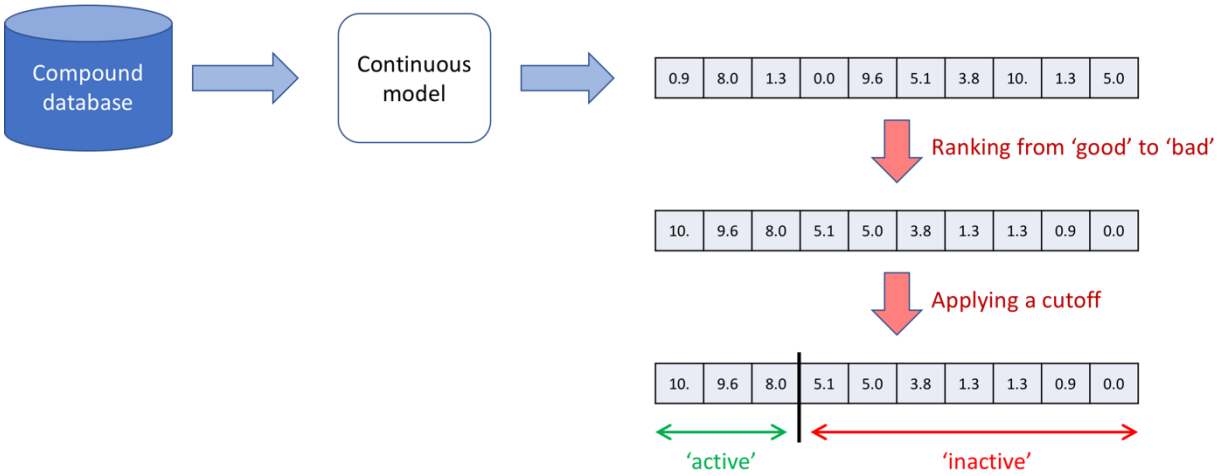


Figure 44. Transforming continuous data into binary data. The first step is the ranking of the data according some user-defined criterium, and the second step is the categorization of the data into two classes by applying a cutoff. This cutoff value is user-defined and may affect the outcome of the performance metrics.

The actual cutoff value has influence on the performance metrics such as specificity, accuracy, and all others, and should therefore be treated as part of the model parameters. As an example, consider two models (a bad and a good model) with their predicted activity of 10 compounds (of which the real experimental activities are known) and using 8 different cutoff values (Figure 45). The corresponding performance metrics are given in Table 9. From these it can be seen that the derived performance metrics are depending on the actual cutoff value, hence each time a performance metric is reported then the corresponding cutoff value should be given as well.

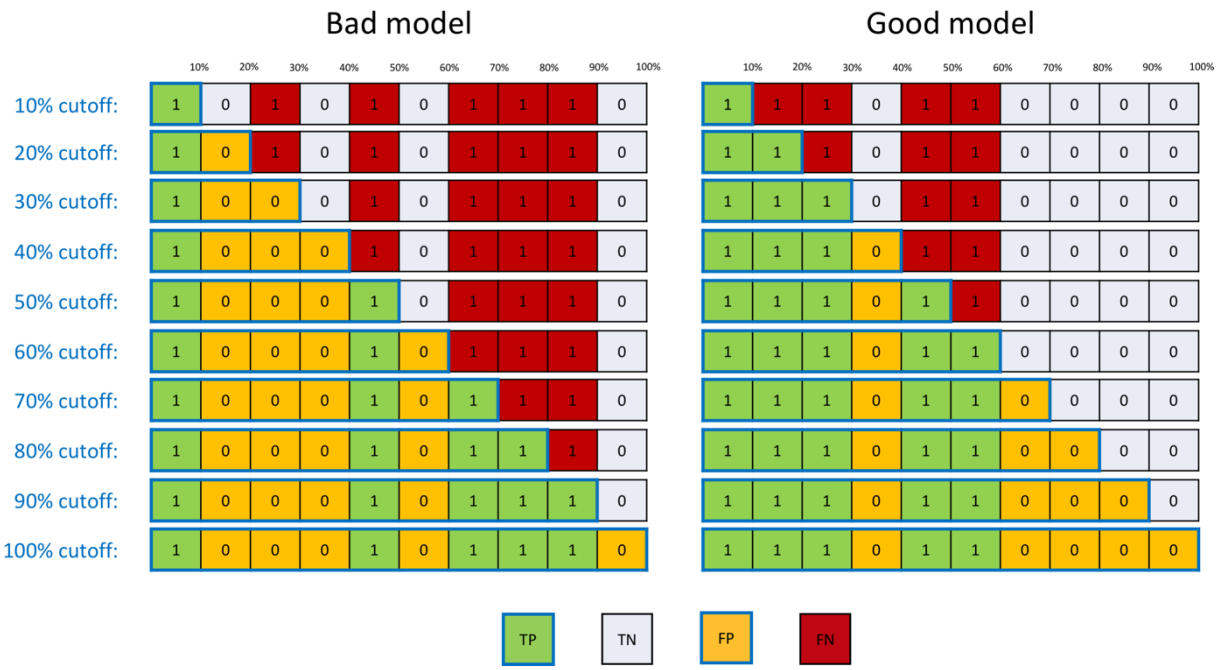


Figure 45. True and false positives, and true and false negatives for two kinds of models. Left the bad model, right the good model. The cutoff ratio is also given. The '1' or '0' in each cell represents the real value of the cell ('1' indicates 'active', '0' is 'inactive'). All values within the defined cutoff are predicted 'active' by the model.

Table 9. Calculated performance metrics for the bad and good model shown in Figure 45.

Cutoff	TP	TN	FP	FN	ACC	PRE	SPE	TPR	FPR	F1	EF	AUC
						Bad model						
10%	1	4	1	5	0.45	0.50	0.80	0.17	0.20	0.25	0.92	0.48
20%	1	3	2	5	0.36	0.33	0.60	0.17	0.40	0.22	0.61	0.38
30%	1	3	3	4	0.36	0.25	0.50	0.20	0.50	0.22	0.55	0.35
40%	1	2	3	4	0.30	0.25	0.40	0.20	0.60	0.22	0.50	0.30
50%	2	2	4	3	0.36	0.33	0.33	0.40	0.67	0.36	0.73	0.37
60%	2	1	4	3	0.30	0.33	0.20	0.40	0.80	0.36	0.67	0.30
70%	3	1	4	2	0.40	0.43	0.20	0.60	0.80	0.50	0.86	0.40
80%	4	1	4	1	0.50	0.50	0.20	0.80	0.80	0.62	1.00	0.50
90%	5	1	4	0	0.60	0.56	0.20	1.00	0.80	0.71	1.11	0.60
100%	5	1	5	0	0.55	0.50	0.17	1.00	0.83	0.67	1.10	0.58
						Good model						
10%	1	5	0	4	0.60	1.00	1.00	0.20	0.00	0.33	2.00	0.60
20%	2	5	0	3	0.70	1.00	1.00	0.40	0.00	0.57	2.00	0.70
30%	3	5	0	2	0.80	1.00	1.00	0.60	0.00	0.75	2.00	0.80
40%	3	4	1	2	0.70	0.75	0.80	0.60	0.20	0.67	1.50	0.70
50%	4	4	1	1	0.80	0.80	0.80	0.80	0.20	0.80	1.60	0.80
60%	5	4	1	0	0.90	0.83	0.80	1.00	0.20	0.91	1.67	0.90
70%	5	3	2	0	0.80	0.71	0.60	1.00	0.40	0.83	1.43	0.80
80%	5	2	3	0	0.70	0.63	0.40	1.00	0.60	0.77	1.25	0.70
90%	5	1	4	0	0.60	0.56	0.20	1.00	0.80	0.71	1.11	0.60
100%	5	0	5	0	0.50	0.50	0.00	1.00	1.00	0.67	1.00	0.50

Enrichment factor (EF)

The enrichment factor EF measures by how much the model is able to 'enrich' the number of actives in the predicted set of actives when compared to how many actives there exist in the entire dataset:

$$EF = \frac{TP(TP + TN + FN + FP)}{(TP + FP)(TP + FN)}$$

Mean squared error (MSE)

The mean squared error is a simple metric that describes how well the predicted values match with the true values. It is calculated by taking the mean of all squared differences between the true values and the corresponding predictions:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

with n the number of datapoints in the test set, $\hat{Y}_i$ the predicted value of datapoint *i*, and Y_i the real value of datapoint *i*. In other words, the *MSE* is the mean of the squares of the differences between real values and predictions. If this difference is close to 0, then the *MSE* will also be close to 0, indicative of a good predictive model.

AUC-ROC

AUC-ROC stands for Area Under the Curve – Receiver Operating Characteristics. It is one of the most important evaluation metrics for checking any classification model's performance. It is also sometimes written as AUROC (Area Under the Receiver Operating Characteristics).

AUC-ROC curve is a performance measurement for classification problem at various thresholds settings. ROC is a probability curve and AUC represents degree or measure of separability. It tells how much model is capable of distinguishing between actives and inactives. Higher the AUC, better the model is at predicting actives as actives and inactives as inactives. By analogy, the higher the AUC, the better the model is at distinguishing between inactive and active compounds.

The ROC curve is plotted with the true positive rate (TPR) against the false positive rate (FPR), where TPR is on *y*-axis and FPR is on the *x*-axis (Figure 46):

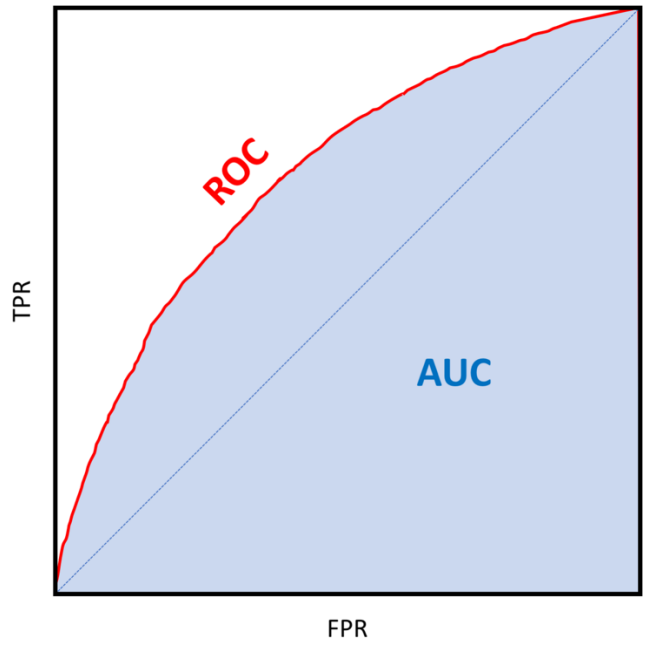


Figure 46. Illustration of the AUC-ROC curve. The dotted blue line corresponds to an AUC = 0.5, and the red line has an AUC larger than 0.5.

An excellent model has an AUC near to one which means it has good measure of separability. A poor model has AUC near to zero which means it has worst measure of separability. In fact, a model with AUC = 0 is reciprocating the result as it is predicting real actives as inactives and real inactives as actives. And when AUC is 0.5, it means that the model has no class separation capacity whatsoever.

Calculating the AUC from a single [FPR,TPR] pair of values obtained from the confusion matrix requires some area calculations. It is the sum of three areas as shown in Figure 47.

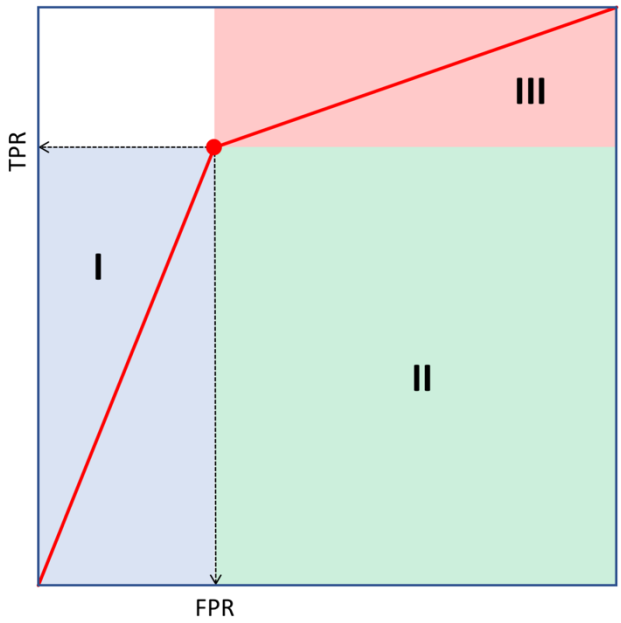


Figure 47. Calculating the AUC-ROC value when only a single [TPR,FPR] pair of values is given. It becomes the sum of half the area of I, the entire area of II, and half the area of III.

Mathematically:

$$AUC = \frac{TPR \times FPR}{2} + (1 - FPR) \times TPR + \frac{(1 - FPR) \times (1 - TPR)}{2}$$

which becomes:

$$AUC = \frac{TPR - FPR + 1}{2}$$

3.3. Cross-validation

Cross-validation or out-of-sample testing is a model validation technique for assessing how the results of a statistical analysis will generalize to an independent data set. It is mainly used in settings where the goal is prediction, and one wants to estimate how accurately a predictive model will perform in practice. In a prediction problem, a model is usually given a dataset of known data on which training is run (training dataset), and a dataset of unknown data (or first seen data) against which the model is tested (called the validation dataset or testing set). The goal of cross-validation is to test the model's ability to predict new data that were not used in estimating it, in order to flag problems like overfitting and to give an insight on how the model will generalize to an independent dataset.

A particular widely-used method of cross-validation is *k*-fold cross-validation. With *k*-fold cross-validation the complete dataset is divided into *k* disjoint parts of the same size. Subsequently *k* different models are built on *k*-1 parts each while those models are always validated on the remaining part of data. The picture below shows how a 10-fold cross-validation works (Figure 48):

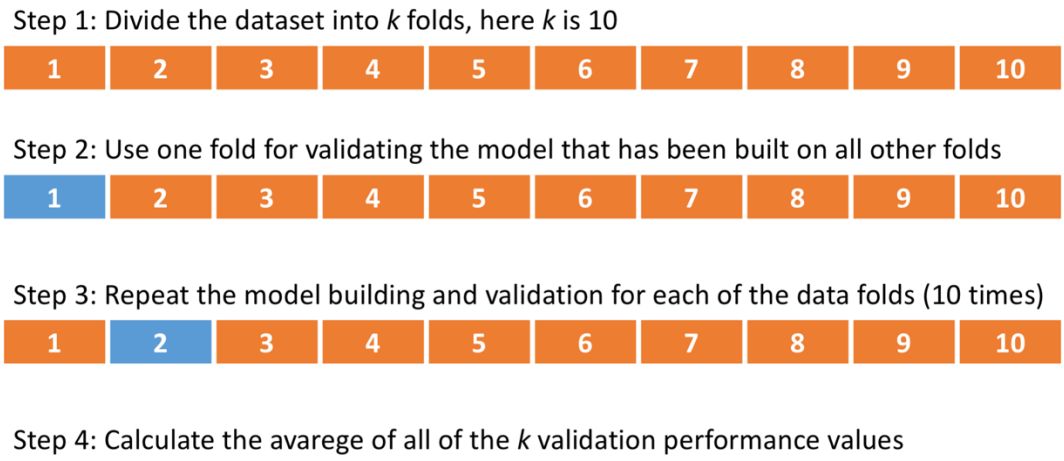


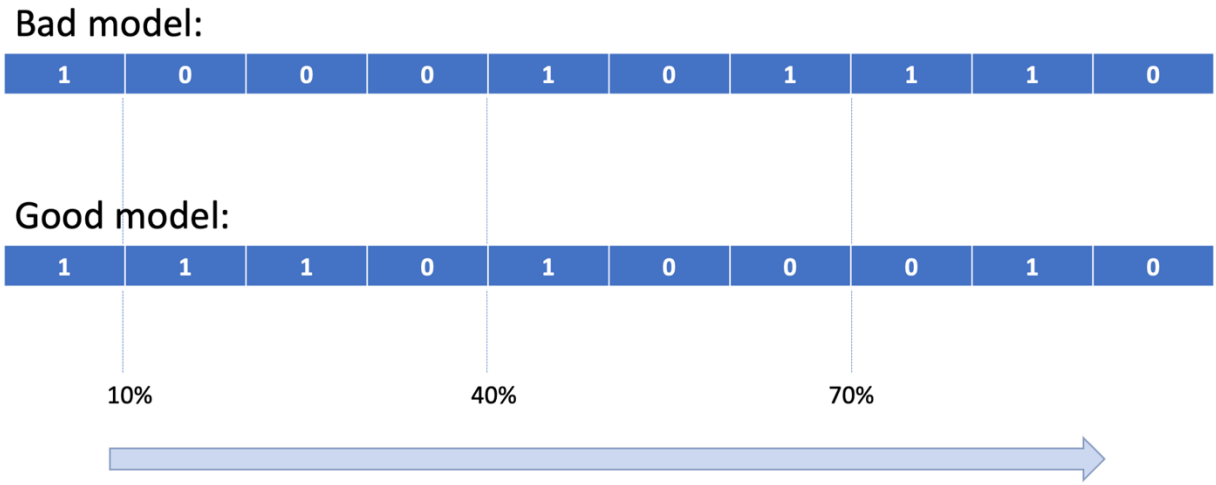
Figure 48. Principle of a k-fold cross-validation. Divide the data into k disjoint parts and use each part exactly once for testing a model built on the remaining parts.

For the reasons discussed above, a *k*-fold cross-validation is recommended whenever you want to validate the future accuracy of a predictive model. It is a simple method which guarantees that there is no overlap between the training and test sets. It also guarantees that there is no overlap between the *k* test sets which is good since it does not introduce any form of negative selection bias. And last but not least, the fact that you get multiple validation errors for different test sets allows you to build an average and standard deviation for these test errors. This means that instead of getting a test error like 15% you will end up with an error average like 14.5% +/- 2% giving you a better idea about the range the actual model accuracy will likely be when you put into production.

4. Excercises

4.1. Exercise 1

Calculate and compare the accuracy and precision metrics for either a good and a bad model at each of the three different cutoff values (10%, 40% and 70%), illustrated as follows:



4.2. Exercise 2

Experiment with some of the regression and classification methods that are available in scikit-learn. Use the dataset that are on the UAMC Github repository. Also play around with different molecular fingerprints and note how the fingerprint types have an effect on the model quality

Chapter 6. Molecular mechanics and conformational analysis

1. Force fields

A force field refers to the functional form and parameter sets used to calculate the potential energy of a molecule. It is nothing more than a set of functions that take as input the coordinates of the atoms, and returns an energy value out of these. *All-atom* force fields provide parameters for every type of atom in a system, including hydrogen, while *united-atom* force fields treat the hydrogen and carbon atoms in each methyl group and each methylene bridge as one interaction center, thereby reducing the number of particles in the calculation with a significant calculation speedup as result.

1.1. Functional form of a force field

The basic functional form of potential energy in molecular mechanics includes 1) bonded terms for interactions of atoms that are linked by covalent bonds, and 2) nonbonded terms that describe the long-range electrostatic and van der Waals forces between atoms that are not bonded by covalent bonds. The specific decomposition of the terms depends on the force field, but a general form for the total energy in an additive force field can be written as:

$$E_{total} = E_{bonded} + E_{nonbonded}$$

and in which the bonded and nonbonded terms can be further defined as:

$$E_{bonded} = E_{bonds} + E_{angles} + E_{dihedrals}$$

$$E_{nonbonded} = E_{electrostatic} + E_{vdw}$$

with E_{bonds}, E_{angles} and $E_{dihedrals}$ describing the energy contributions of all covalent bonds, angles and dihedral angles, respectively, and $E_{electrostatic}$ and E_{vdw} the terms describing the nonbonded interactions between atoms due to the atomic partial electrostatic charges and the van der Waals interactions. In the following sections, each of these terms are described in more detail.

1.2. Bond potential

In standard force fields, the contribution by covalent bonds to the total relative energy is given by a Hooke's law in which bonds are treated as springs:

$$E_{bonds} = \sum_{all\ bonds} k(b - b_o)^2$$

with k being a bond force constant, b the actual bond length for a given bond, and b_0 the reference length. For example, in the case of a C-C bond, k could be 100 kcal/mol/Å^2 and b_0 is 1.54 Å. Hence, when the bond is at its reference value of 1.54 Å, then the relative energy contribution equals 0.0 kcal/mol, but when the bond is stretched or compressed by 0.1 Å, then the energy rises by 1.0 kcal/mol.

The force constant k and reference bond length b_0 depends on the actual bond type (single, double, triple) and on the constituting atoms (Figure 49).

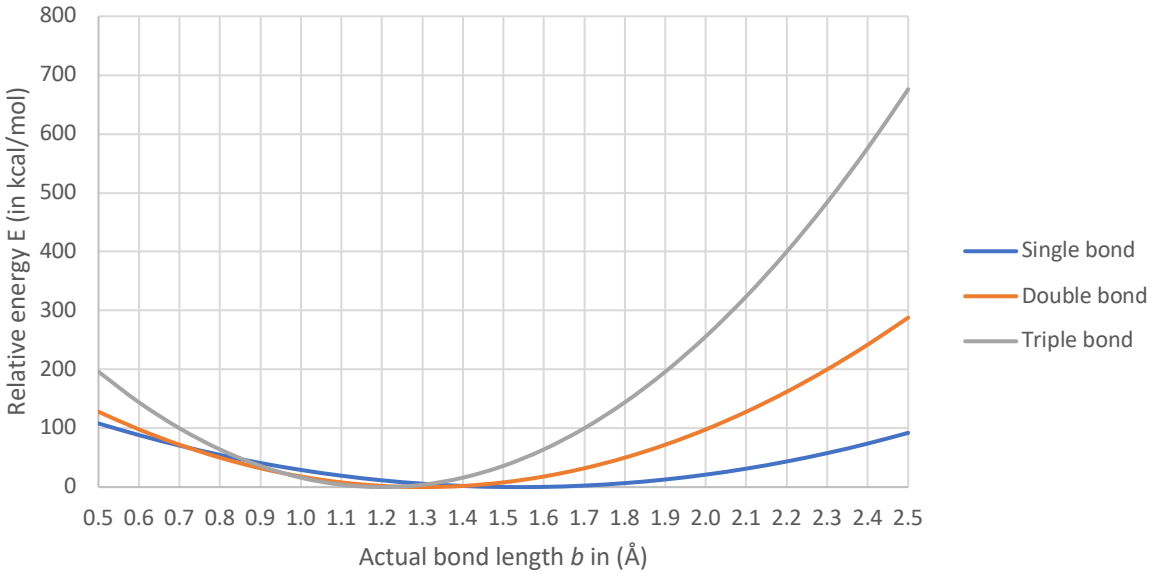


Figure 49. Bond potential illustrated for a single, double and triple C-C bond. Force field parameters are 1.54, 1.3 and 1.2 Å for b_0, and 100, 200 and 400 kcal/mol/Å² for k, respectively. The triple bond is more rigid than the single bond, as the amount of compression or stretching leads to a higher increase in relative energy.

1.3. Angle potential

The contribution of angles to the total energy of a molecule is similar to that from bonds:

$$E_{angles} = \sum_{all\ angles} k(\theta - \theta_0)^2$$

which sums over all angles and in which *k* being the angle force constant, θ the actual angle for a given bond angle, and θ_0 the reference angle. Again, the actual values for *k* and θ_0 depend on the constricting atoms that make up the bond angle.

1.4. Dihedral angle potential

A dihedral angle is an angle that is defined by four points, *in casu* four atoms. It is the angle between the planes that are defined by two sets of three atoms each, having two atoms in common (Figure 50):

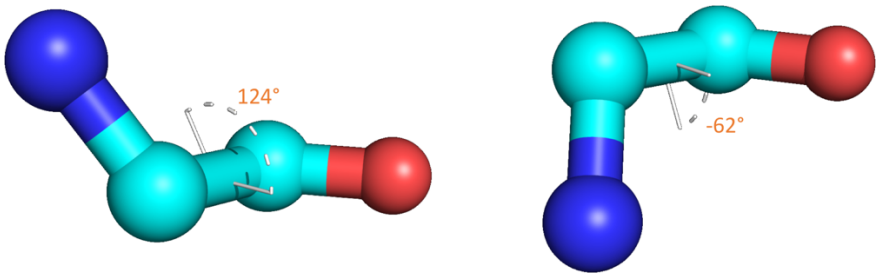


Figure 50. The dihedral angle for the N (blue) – C (cyan) – C (cyan) – O (red) sequence is defined as the angle between the plane through N-C-C and the plane through C-C-O.

Dihedral angle potentials are a bit more complex than bond or angle potentials, since the dihedral angle potential is defined as a periodic function with an optional phase shift:

$$E_{dihedrals} = \sum_{all\ dihedrals} k(1 + \cos(n\phi - \delta))$$

in which *n* is an integer ranging from 1, 2 or 3, ϕ being the actual dihedral angle and δ the phase shift (normally this is a value of 0° or 180°) (Figure 51).

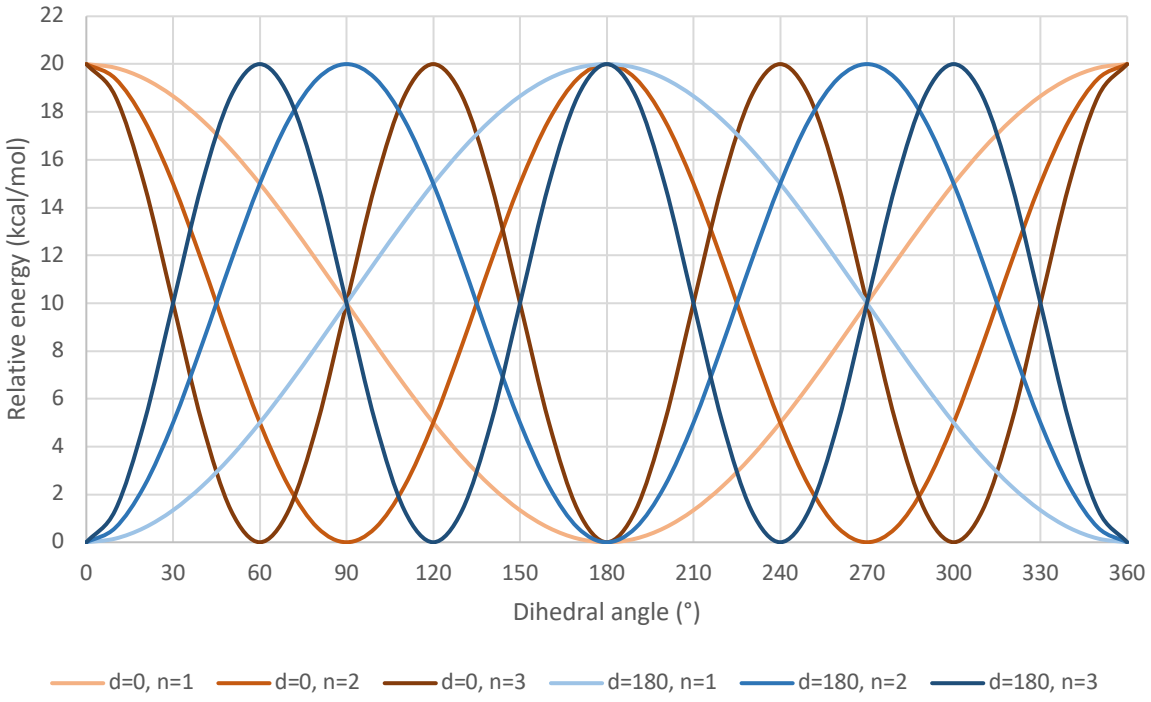


Figure 51. Illustration of a dihedral potential and its dependence on the force field parameters. In this particular example, k was set to 10 kcal/mol, and both δ (shown as d in the legend) and n were varied as indicated. A phase shift δ of 0° is often the most appropriate, since this gives energy minima near 60, 90 and/or 180°.

1.5. Electrostatic potential

The first nonbonded potential is the electrostatic potential, which is derived from the interaction between the partial atomic charges between all nonbonded atom pairs:

$$E_{electrostatic} = \sum_{all\ nonbonded\ atom\ pairs} \frac{q_i q_j}{kDr_{ij}}$$

with q_i and q_j the atomic partial charges of atom *i* and atom *j*, r_{ij} the actual distance between atom *i* and atom *j*, *k* a constant and *D* the dielectric constant. The atomic partial charges can be calculated using a wide variety of methods, including quantum chemical calculations from small molecule systems, and going up to less accurate approaches from protein systems. In most cases, fixed values for the atomic partial charges are used, with electronegative atoms such as N and O being assigned a negative value (for example -0.3 electrons), and carbons and hydrogen atoms a positive value. Depending on the sign of the atomic partial charge, atom pairs may attract or repel each other (Figure 52).

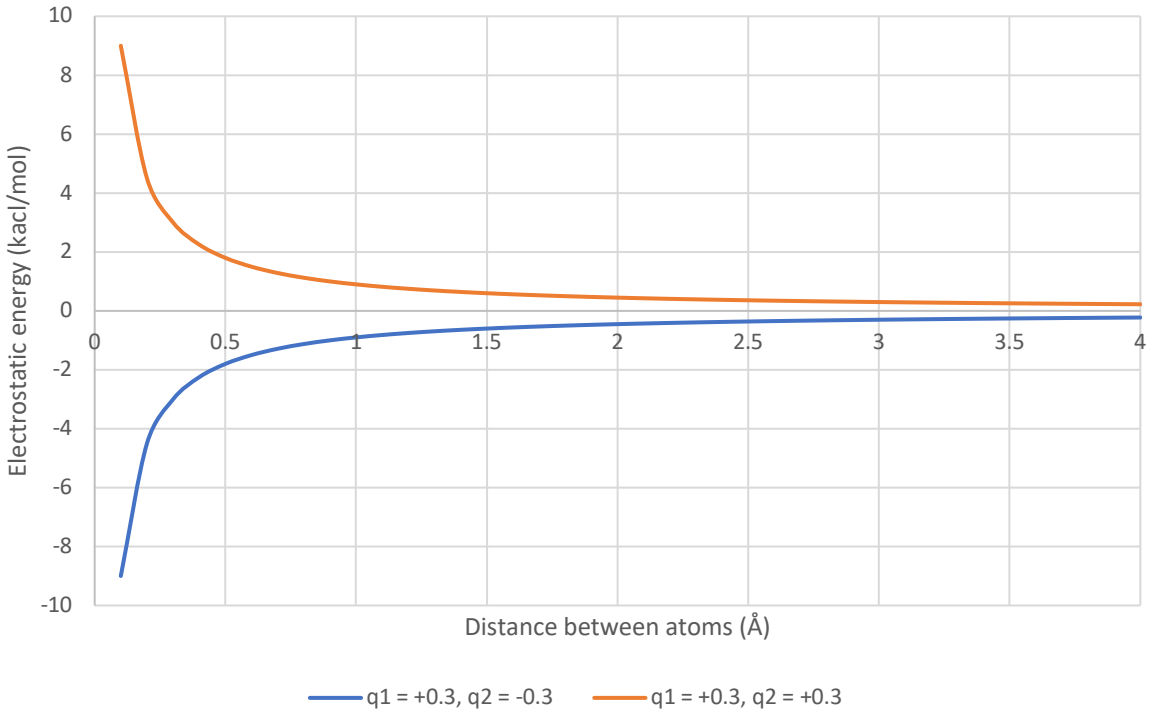


Figure 52. Illustration of the electrostatic energy as a function of the distance between the two atoms. Two cases are shown: the first case (blue) shows the change in electrostatic energy when both atoms carry an opposite charge (+0.3 and -0.3), while the second case shows the change in energy when both atoms carry a partial charge of the same sign (here +0.3 and +0.3). In the former case, the electrostatic energy is attractive, while in the latter situation the potential is repulsive. Note that the potential slowly limits a value of zero at long distances between the two atoms.

1.6. Van der Waals potential

The second nonbonded potential is the van der Waals (VDW) interaction, named after the Dutch scientist Johannes Diderik van der Waals. In contrast to the electrostatic interactions, these attractions are comparatively weak and vanish quickly at longer distances. Van der Waals interactions between two atoms arise from the balance between repulsive and attractive forces. Repulsion is due to the overlap of the electron clouds of both atoms, while the interactions between induced dipoles result in an attractive component. There are many mathematical models to describe the van der Waals interaction, but the 12-6 Lennard-Jones (LJ) potential is the one which is most often used to represent these interactions:

$$E_{vdw} = \sum_{all\ nonbonded\ atom\ pairs} \varepsilon_{ij} \left[\left(\frac{R_{min,ij}}{r_{ij}} \right)^{12} - 2 \left(\frac{R_{min,ij}}{r_{ij}} \right)^{6} \right]$$

with r_{ij} the distance between atom *i* and *j*, $R_{min,ij}$ the reference distance at which the interaction is strongest, and ε_{ij} a constant. At interatomic distances shorter than $R_{min,ij}$, the VDW interaction potential is increasing and leads to repulsion between the two atoms. At distances longer than $R_{min,ij}$, the VDW potential also increases and will approach zero at long distances (Figure 53).

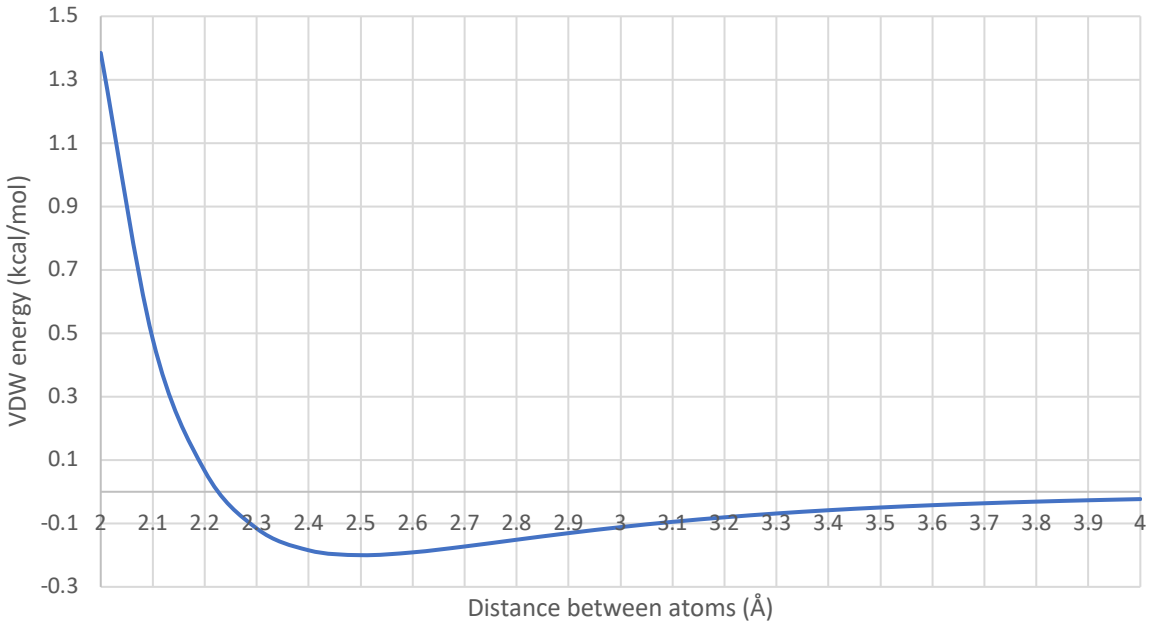


Figure 53. Illustration of the van der Waals potential between two atoms. The distance at which the potential reaches its minimum (in this case 2.5 Å) is called the van der Waals contact distance. This distance is dependent on the atom types; in general atoms with a higher atomic number also have a larger van der Waals radius.

1.7. Popular force fields

In the preceding sections, an overview was given of the different functions that are commonly used in force fields. The majority of the current force fields all rely on the same kind of functionalities although that small differences exists in both the functional form and/or the parameters used. There exist therefore a number of different force fields and some of them are listed here:

- AMBER – 'Assisted Model Building and Energy Refinement'. Widely used force fields for protein and DNA/RNA.
- CHARMM – 'Chemistry at HARvard Molecular Mechanics'. Originally developed at Harvard, nowadays maintained by Alexander MacKerrell in Baltimore. Widely used for both small molecules and macromolecules.
- GROMOS – 'GROningen MOlecular Simulation'. A force field that comes as part of the GROMOS software, a general-purpose molecular dynamics computer simulation package for the study of biomolecular systems. GROMOS force field A-version has been developed for application to aqueous or apolar solutions of proteins, nucleotides, and sugars. The B-version to simulate gas phase isolated molecules is also available.
- OPLS – 'Optimized Potential for Liquid Simulations'. Developed by William Jorgensen at the Yale University Department of Chemistry. OPLS variants include OPLS-AA, OPLS-UA, OPLS-2001 and OPLS-2005.
- UFF – 'Universal Force Field'. A general force field with parameters for the full periodic table up to and including the actinoids, developed at Colorado State University.
- MMFF – 'Merck Molecular Force Field'. Developed at Merck and suitable for a broad range of molecules.
- MARTINI - A coarse-grained potential developed by Marrink and coworkers at the University of Groningen, initially developed for molecular dynamics simulations of lipids, later extended to various other molecules. The force field applies a mapping of four heavy atoms to one CG interaction site and is parameterized with the aim of reproducing thermodynamic properties.

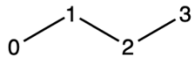
2. From 1/2D to 3D: Conformation generation with distance geometry

As already described before, molecules are often represented in 1D using the SMILES format, or in 2D with SD-format. However, in order to be able to explore the conformation of molecules, for example within the context of molecular dynamics or docking (see below), these 1D or 2D structures need to be converted into a real 3D representation, with the location in space of each atom represented with a *x*-, *y*-, and *z*-coordinate. There are many approaches that can be used for this purpose, but distance geometry is very often used and quite robust. In this section, we will briefly explain the rationale behind distance geometry, however without going into the mathematical details.

One way to describe the 3D-conformation of a molecule is in terms of the distances between all pairs of atoms. For a molecule that consists of N atoms, there are *N* (*N*-1) / 2 interatomic distances in the molecule, which can be described using a *N* x *N* symmetrical matrix. In this matrix, both the elements (*i*,*j*) and (*j*,*i*) contain the distance between atoms *i* and *j*. The diagonal elements are by definition all zero. The crucial feature about distance geometry is that this matrix cannot contain random distances; rather there are some constraints that can be defined to restrict the distances to a small set of potential solutions. For example, given a simple molecule such as butane, each cell within the distance matrix can be filled with both a lower and a upper distance, as examplified in Figure 54:

MIN	0	1	2	3
0	0	1.5	3.6	3.6
1	1.5	0	1.5	3.6
2	3.6	1.5	0	1.5
3	3.6	3.6	1.5	0

MAX	0	1	2	3
0	0	1.5	3.0	4.5
1	1.5	0	1.5	3.0
2	3.0	1.5	0	1.5
3	4.5	3.0	1.5	0



LIMITS	0	1	2	3
0	0	1.5	3.0/3.6	3.6/4.5
1	1.5	0	1.5	3.0/3.6
2	3.0/3.6	1.5	0	1.5
3	3.6/4.5	3.0/3.6	1.5	0

Figure 54. The principle of distance geometry explained using butane (four carbon atoms) as an example. The bond length between each of the bonded atoms is 1.5 Å, and the VDW radius of each carbon atom is 1.8 Å, implying that the closest distance that is possible between a pair of unbonded carbon atoms is 3.6 Å. The longest distance possible between any pair of unbonded carbon atoms is equal to the number of bonds between them, multiplied by the bond lengths. In practice, there are many additional restraints that can be generated to limit the boundaries in the distance matrix. For example, the minimal and maximal distances between atoms 0 and 2 is not 3.0/3.6 as is shown in the figure, but rather a single 2.45 Å given the fact that these two atoms are 1-3 linked and that the bond angle of 109.5° between 0-1-2 imposes additional limitations on the distance between 0 and 2. Using chemical knowledge many additional restraints can be formulated, like in the case of ring systems or between atoms separated by a torsion angle.

With a completed distance matrix at hand (containing upper and lower bounds), random values are assigned to each interatomic distance between its upper and lower bonds. In the following step, the distance matrix is converted into a trial set of Cartesian coordinates using a series of matrix operations, which are then refined in the final step.

Conformation generation with distance geometry is standard incorporated in RDKit, and can be called very conveniently to generate 3D-conformations from SMILES representations:

```
from rdkit import Chem
from rdkit.Chem import AllChem
mol = Chem.MolFromSmiles("C1CCOCC1NC=O")
mol = Chem.AddHs(mol)
AllChem.EmbedMolecule(mol)
```

It is common to refine the generated conformation(s) using some kind of energy minimization procedure.

3. Energy minimization

Energy minimization (also called energy optimization, geometry minimization, or geometry optimization) is the process of finding a molecular conformation, according to the energy calculated by the used force field, the net inter-atomic force on each atom is acceptably close to zero and the position on the potential energy surface is a (local) minimum. The motivation for performing a geometry optimization is the physical significance of the obtained structure: optimized structures often correspond to a substance as it is found in nature and the geometry of such a structure can be used in a variety of experimental and theoretical investigations such as quantitative structure-activity relationships.

Starting from a random conformation, and because most minimization algorithms can only go downhill, each energy minimization process results in the identification of the nearest local minimum; however, this is not always the global energy minimum (Figure 55).

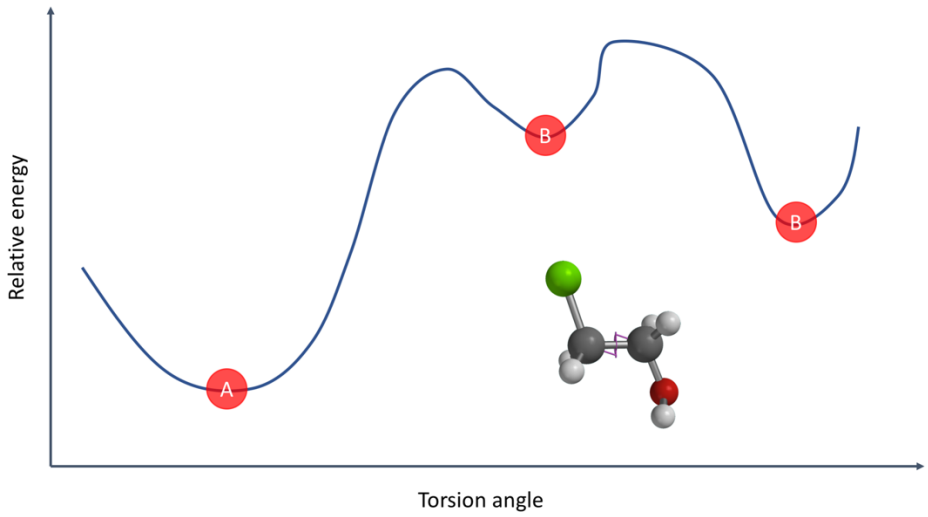


Figure 55. The global minimum (A) and two local minima (B) on the energy profile when rotation along the indicated torsion angle. The relative energy is calculated using a molecular mechanics force field.

A number of methods are available to calculate to local minimum, the steepest descent and conjugate gradients algorithm are two of the most commonly used ones. Of these two methods, steepest descent is several times faster than conjugate gradients, but the latter method converges after a smaller number of iterations and it therefore more productive (Table 10).

Table 10. Comparison between the steepest descent and conjugate gradient methods for energy minimisation of netropsin, a molecule with 13 flexible torsion angles. Metrics for an initial minimisation and a stringent minimisation are given. From reference 6.

Method	Initial minimisation (gradient < 1 kcal/Å²)		Stringent minimisation (gradient < 0.1 kcal/Å²)	
	CPU time (s)	Number of iterations	CPU time (s)	Number of iterations
Steepest descent	67	98	1,405	1,893
Conjugate gradient	149	213	257	367

4. Conformational analysis

Conformation generation is the process of transforming the 1D- or 2D-structure of a molecule into a 3D-structure, hence a structure in all atoms of the molecule have *x*-, *y*-, and *z*-coordinates assigned. Examples of 1D-representations include SMILES and InChi, and 2D-representations can include 'flat' SDF or PDB-formats (with 'flat' meaning that the file does not contain *z*-coordinates). Since the majority of molecules are conformationally flexible (rotation around single bonds), multiple conformations exist for a given molecule and conformational analysis is the process of generating these conformations. Conformational analysis can therefore be regarded as the analysis of the conformations that molecules can adopt as a result of single bond rotations, with the intention to locate the global energy minimum and several other minima.

A molecule can adopt an equilibrium between several such minima, the relative abundance of which is determined by the Boltzmann distribution, and which in turn is merely determined by the relative free energy of each pose. The shape of a molecule is not static but is a dynamic equilibrium between a number of conformations, the preferred ones being those we would encounter more times than any other if we were to take a series of snapshots of the population, because they have lower free energies. Conformational analysis can be considered to consist of two parts (Figure 56):

- The equilibrium **distribution** of the different energy minima of a molecule is driven by the **thermodynamics** (the differences in Gibbs free energies between the minima):

$$K_e = e^{-\frac{\Delta G}{RT}}$$

- The **rate** of interconversion between the different minima of a molecule is driven by the **kinetics** (the height of the free energy of activation barrier):

$$\ln\left(\frac{k}{T}\right) = 23.76 - \frac{\Delta G^{\ddagger}}{RT}$$

where $\Delta G^{\ddagger}$ is the free energy of barrier and *k* the rate constant (in s^{-1}).

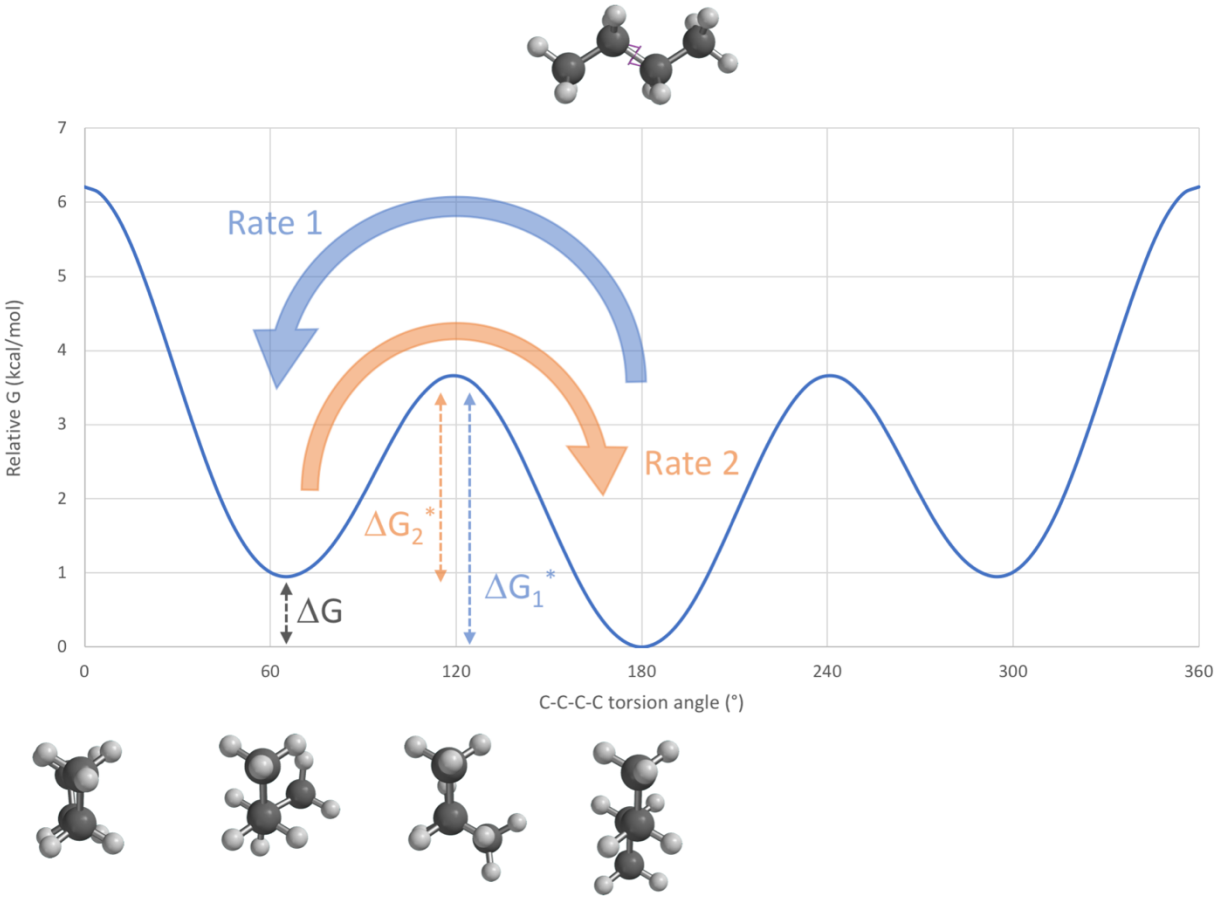


Figure 56. Conformational profile of n-butane, thereby focusing on the C-C-C-C bond. The different conformers for butane are, from left to right, syn-planar (0°), gauche (60°), anti-clinal (120°) and staggered (180°). The free energy difference between the gauche and staggered conformations is about 1 kcal/mol, corresponding to 16% of the conformations in gauche and 84% in staggered. Conversion from staggered to gauche will be slower than the corresponding conversion from gauche to staggered, as can be seen from the differences in activation free energies $\Delta G^{\ddagger}$.

Conformational analysis normally consists of two steps, and which might be iterated over and over again until a satisfactory solution has been identified:

- Energy calculation process.
- Exploration of the conformational space.

4.1. Systematic search

A systematic search is conceptually the simplest of all conformational analysis methods. Using a starting 3D-structure, torsion angles are varied in regular increments and at each step the corresponding energy is calculated. The conceptual simplicity of the systematic search method is in sharp contrast to the combinatorial complexity of its calculation. If A is the torsion angle increment and T is the number of rotatable torsion angles in the molecule, then the total number of possible conformers is $(360/A)^T$. The relative growth in the number of energy calculations is given Table 11.

Table 11. Relative computational complexity of a systematic search as a function of torsion angles and angle increment.

Angle increment (°)	Number of torsions			
	5	10	20	40
30	1	9.1×10^{5}	2.1×10^{17}	3.2×10^{39}
15	3.2×10^{1}	9.1×10^{8}	2.2×10^{23}	3.5×10^{51}
8	7.4×10^{2}	5.0×10^{11}	6.5×10^{29}	2.9×10^{62}
4	2.4×10^{4}	5.0×10^{14}	6.8×10^{35}	3.2×10^{72}
2	7.6×10^{5}	5.3×10^{17}	7.1×10^{40}	3.5×10^{86}

As a result of this computational complexity, systematic search methods to identify the global minimum can only be used in case of molecules having a limited number of flexible torsion angles, making the method less useful in practice. For this reason, other methods have been developed, including the Monte Carlo method and genetic algorithms.

4.2. Monte Carlo

Monte Carlo represents a technique to find a good solution to an optimization problem by trying random variations of the current solution. A worse variation is accepted as the new solution with a probability that decreases as the computation proceeds. Monte Carlo is not exhaustive however, meaning that some predefined heuristics are needed to define a suitable endpoint.

In its simple form, starting from a given starting structure random alterations are made and the internal energy of the resulting structure is calculated and an energy minimization step is undertaken. The new conformer produced is saved if the energy or the difference in energies between the new and the best conformer is less than the threshold optimum value. The search is automatically terminated after a user-defined number of unproductive attempts.

The Metropolis Monte Carlo approach amplifies the changes of finding the global minimum. It involves number of sequences where the Monte Carlo algorithm is run at different temperatures. The first phase runs at temperature T and an assortment of conformations are generated. The most stable conformation is used as starting origin for next phase where the temperature is set at a lower temperature. The process is repeated until the probability equation becomes selective towards which structure is accepted. Thus, a small part of the conformation space is meticulously investigated. The Metropolis Monte Carlo method introduces a probabilistic criteria to accept new conformers:

$$p = e^{-\Delta E / T k_b}$$

with ΔE being the energy difference between the new conformation and the initial starting one, T the temperature and K_b the Boltzmann constant. The acceptance test is performed by choosing a random number r (between 0 and 1) which is then compared to p. If $r < p$, the change is accepted and the new conformations is taken as the new starting point.

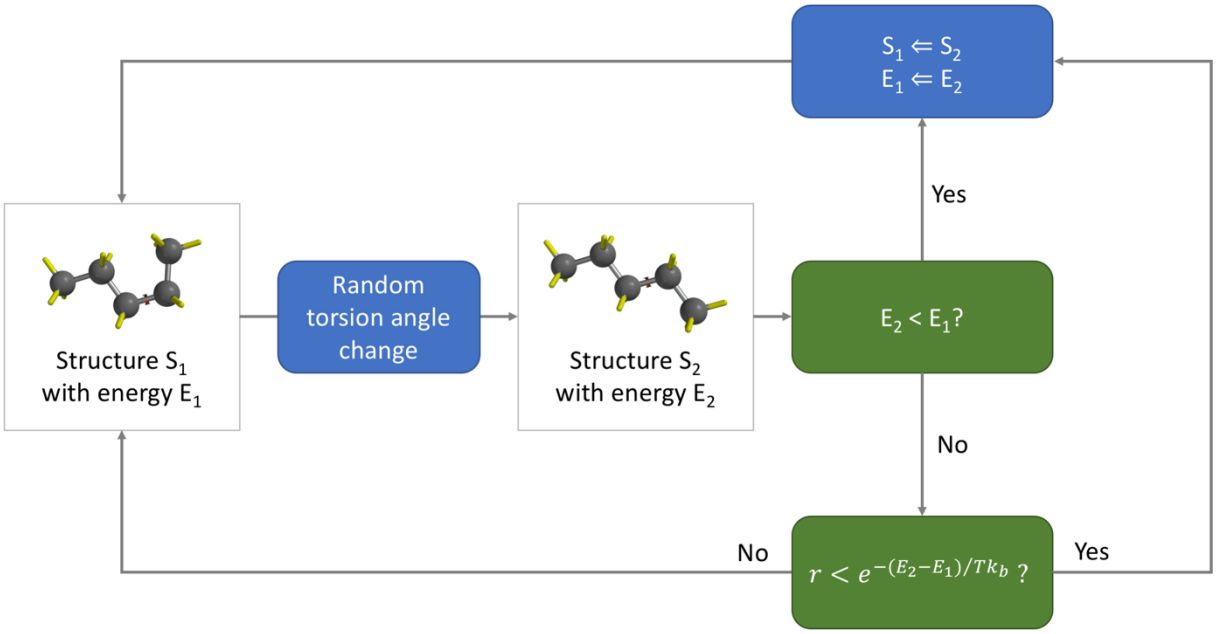


Figure 57. The Monte Carlo method for conformational analysis. The procedure is stopped after a user-defined number of cycles. The random number is defined as r, T the temperature and k_b the Boltzmann constant. In the Metropolis approach, the temperature T is slowly decreased as the number of runs increase.

4.3. Genetic algorithm

A genetic algorithm (GA) is search heuristic optimization method that is based on various computational models of Darwinian evolution. The genetic algorithm is a large-scale optimization algorithm mimicking a biological evolution in a randomly generated population. The algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation. A number of conformations forms this population. New chromosomes are generated by modifying some of the torsion angles, and a new population is created in accordance to operators (crossover and mutation). The process is repeated until it converges to a minimum energy structure.

Five phases are considered in a genetic algorithm:

1. Initial population
2. Fitness function
3. Selection
4. Crossover
5. Mutation

Initial population

The process begins with a set of individuals which is called a population. In the context of conformational analysis, each individual is a set of torsion angle values representing a single conformation of the molecule of which one wants to find the global minimum (Figure 58).

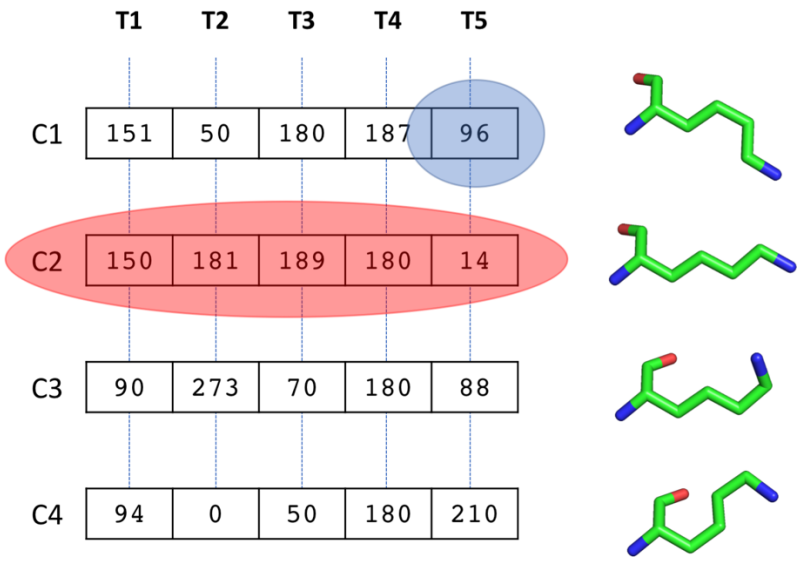


Figure 58. Genes (blue region), chromosomes (red region) and the population (all chromosomes). A single chromosome defines a single conformer, while each gene represents a single torsion angle in the molecule (in this example, each conformation is represented by five torsion angles).

Fitness function

The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score. In the case of a genetic algorithm for conformational analysis, the fitness function is simply the total molecular mechanics energy of each molecule, as calculated by the force field.

Selection

The idea of selection phase is to select the fittest individuals and let them pass their genes to the next generation. Two pairs of individuals (parents) are selected based on their fitness scores (energies). Individuals with high fitness (in casu the lowest energies) have more chance to be selected for reproduction.

Crossover

Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes. Offspring are created by exchanging the genes of parents among themselves until the crossover point is reached, and the new offspring are added to the population (Figure 59).

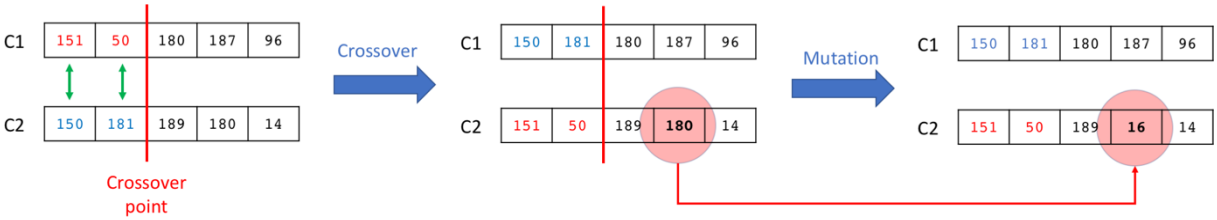


Figure 59. The workflow of the crossover and mutation operators in a genetic algorithm, illustrated on a population of two chromosomes.

Mutation

In certain new offspring formed, some of their genes can be subjected to a mutation with a low random probability. This implies that some of the torsion angles can be changed randomly (between 0 and 359°). Mutation occurs to maintain diversity within the population and to prevent premature convergence (Figure 59).

Pseudocode

```
START
Generate the initial population
Compute energy
REPEAT
       Selection
       Crossover
       Mutation
       Compute energy
UNTIL population has converged
STOP
```

Chapter 7. Molecular dynamics

1. Newton's 2nd law of motion

Molecular dynamics (MD) is a method to simulate the physical movement of atoms and molecules using a series of mathematical calculations. Commonly, this is done by numerically solving Newton's equations of motion for the system of interacting atoms and bonds, in which the forces and the potential energies between these atoms are calculated from the standard molecular mechanics force fields (Chapter 6).

A typical MD simulation consists of an iterative process which involves a number of discrete steps:

1. Given a set of atomic positions and a defined molecular mechanics force field, the first step consists of the calculation of the forces on each atom.
2. In the second step, the acceleration on each atom can be deduced from the force that acts on each atom.
3. In the third step, each atom is repositioned according the acceleration that is acting on it. For this purpose, small time steps are required to ensure numerical stability.
4. If enough steps have been performed, the simulation will step. If not, the simulation will continue in step 1.

These steps are described in more detail in the following sections.

1.1. Step 1: force calculation

The mathematical principles behind a MD simulation are quite simple, and involve the calculation of the forces between the atoms from the potential energy function that defines the system:

$$\vec{F} = -\nabla V(\vec{r})$$

with $\vec{F}$ being the forces on the system and $V(\vec{r})$ the potential energy of the system as function of the atomic positions $\vec{r}$.

Using the potential energy function as shown on page 86, calculating the corresponding first derivatives to obtain the forces is actually quite simple:

- Bond forces: $F_b = -2k(b - b_0)$
- Angle forces: $F_a = -2k(\theta - \theta_0)$
- Torsion angle forces: $F_d = kn\sin(n\phi - \delta)$
- Electrostatic forces: $F_e = q_i q_j / Dkr_{ij}^2$
- Van der Waals forces: $F_{vdw} = -12\varepsilon_{ij}R_{min,ij}^6\left(r_{ij}^6 - R_{min,ij}^6\right)/r_{ij}^{13}$

Plots of the potentials and corresponding forces are shown in Figure 60.

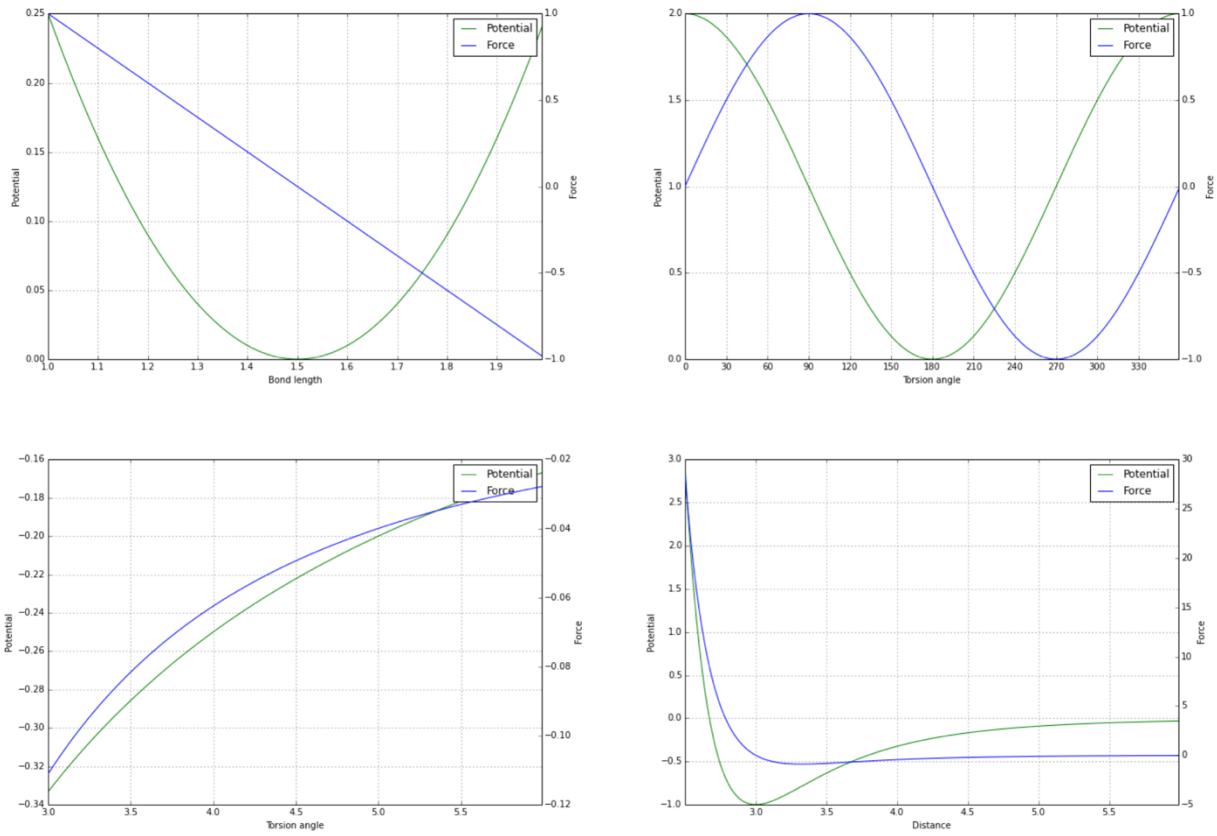


Figure 60. Illustration of the different force field potentials and the corresponding forces that are derived thereof. A) Upper left: bond potential and force with b_0 = 1.5 and k = 1. B) Upper right: torsion angle potential and force with n = 1, k = 1, δ = 0. C) Lower left: electrostatic potential and force with q_i = +1 and q_j = -1, D =1, k = 1. D) Lower right: Van der Waals potential and force with ε_{ij} = 1, $R_{min,ij}$ = 3.

1.2. Step 2: atom acceleration

Once the force acting on each atom is known, it is possible to calculate the acceleration for each atom from Newton's 2nd law of motion:

$$\vec{F} = m\vec{a}$$

Therefore:

$$\vec{a}_i = \frac{\vec{F}_i}{m_i}$$

with *i* being the index of the particular atom *i*.

1.3. Step 3: new atom positions

Several methods have been developed to calculate new atoms positions from the acceleration on each atom. The most important ones are the Verlet and leap-frog algorithms.

Verlet algorithm

The Verlet algorithm is based on the Taylor series expansion:

$$f(x_0 + x) = f(x_0) + \frac{x^1}{1!}f'(x_0) + \frac{x^2}{2!}f''(x_0) + \frac{x^3}{3!}f'''(x_0) + \cdots + \frac{x^n}{n!}f^{(n)}(x_0)$$

Taylor series are often truncated after the term involving the second derivative. Applied to our MD simulation, we can rewrite the new set of coordinates as $r(t + \delta t)$, and expand this as a Taylor series with *t* as x_0 and δt as *x* (or -δt as *x*):

$$r(t+\delta t) = r(t) + \frac{\delta t^1}{1} r'(t) + \frac{\delta t^2}{2} r''(t) + \cdots$$

$$r(t-\delta t) = r(t) - \frac{\delta t^1}{1} r'(t) + \frac{\delta t^2}{2} r''(t) + \cdots$$

which yields the following considering that $r'(t) = v(t)$ and $r''(t) = a(t)$:

$$r(t+\delta t) = r(t) + \delta t\, v(t) + \frac{1}{2}\delta t^2\, a(t)$$

$$r(t-\delta t) = r(t) - \delta t\, v(t) + \frac{1}{2}\delta t^2\, a(t)$$

with *v(t)* being the velocities on the atoms and *a(t)* the accelerations. Adding these two equations gives:

$$r(t+\delta t) + r(t-\delta t) = r(t) + \delta t\, v(t) + \frac{1}{2}\delta t^2\, a(t) + r(t) - \delta t\, v(t) + \frac{1}{2}\delta t^2\, a(t)$$

which can be rewritten as:

$$r(t+\delta t) = 2r(t) + \delta t^2\, a(t) - r(t-\delta t)$$

Implementation of the Verlet algorithm is straightforward. Calculation of a MD trajectory [*r(t+δt)*] requires only a timestep δt, the positions at timestep -δt [*r(t-δt)*], and the accelerations *a(t)* which in turn are derived from the forces.

Leap-frog algorithm

The leap-frog algorithm uses the following relationships:

$$r(t+\delta t) = r(t) + \delta t\, v\left(t + \frac{1}{2}\delta t\right)$$

$$v\left(t + \frac{1}{2}\delta t\right) = v\left(t - \frac{1}{2}\delta t\right) + \delta t\, a(t)$$

The velocities $v\left(t + \frac{1}{2}\delta t\right)$ are first calculated from the velocities at $t - \frac{1}{2}\delta t$ and the accelerations at time *t*. From this, the coordinates $r(t+\delta t)$ are then derived from the velocities at $t - \frac{1}{2}\delta t$ with the positions at the current frame. Hence, the velocities thus 'leap-frog' over the positions, which explains the name of the algorithm (Figure 61).

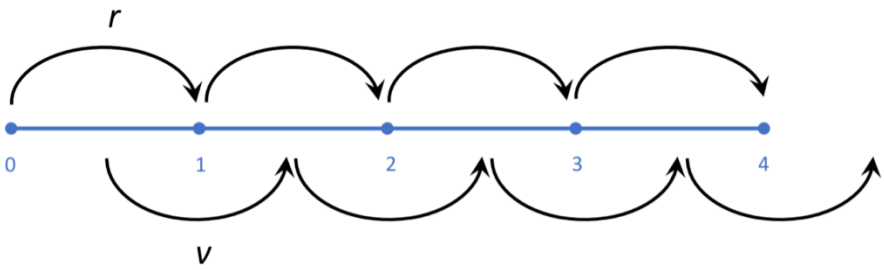


Figure 61. Sketch showing the structure of the leap-frog method. Once the algorithm has started with r(0) and $v(\frac{1}{2}\delta t)$, the algorithms continues by r and v leap-frogging over each other.

Choosing a timestep δt

An important parameter in the equations of motion of each MD simulation is the timestep δt. Using a large timestep would speed up the simulations significantly, but this would lead to algorithmic instabilities as the calculated atom relocations would be too large (Figure 62).

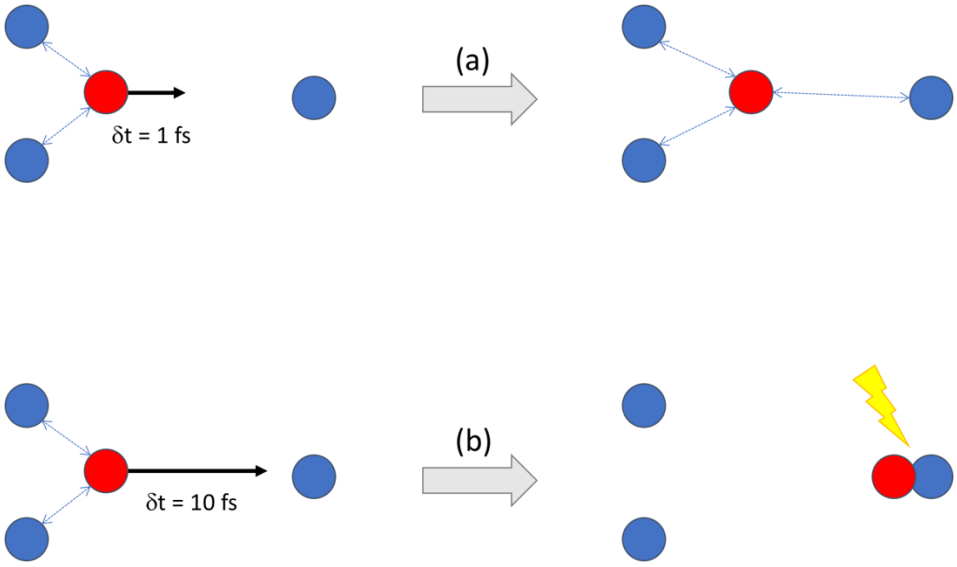


Figure 62. Illustration of the importance of using a suitable timestep. Blue spheres are atoms which are fixed in space, while the red sphere is the atom that is able to relocate due to the forces (blue dotted arrows) imposed by the surrounding blue atoms. (a) Relocation of the red atoms using a timestep of 1 fs. This yields a new situation in which the red atom is moved somewhat closer to the 3rd blue atom, but remains at a distance which is still far enough to avoid sterical clashes. (b) With a timestep of 10 fs, the red atom is relocated too far which results in a sterical clash with the 3rd blue atom.

It has been shown that the most optimal timestep should be 10 times smaller than the period of the highest vibrational frequencies of a molecule. In practice, a timestep of 1 fs is often used, and this number can be doubled to 2 fs in cases where the highest frequencies, *in casu* the C-H bond stretching vibrations, can be freezed out by constraining these bonds to their equilibrium values. Examples of such constraints are the SHAKE [7], LINCS [8] and RATTLE [9] algorithms.

2. Running a simulation

2.1. Setting up the protein and its environment

Obtaining protein coordinates

The basic ingredient to start a molecular dynamics simulations is a structure coordinate file of the molecule one wants to study. In case this is a protein or oligonucleotide structure, these structure coordinates can be downloaded from the Protein Databank (PDB, http://www.rcsb.org/pdb/home/home.do). The PDB contains thousands of high resolution protein structures that are derived by X-ray/neutron scattering and NMR methods. Once the PDB is available and downloaded, it is necessary to pre-format it depending on the presence of ligands or water molecules in the structure. It occurs that ligand coordinates are present in the PDB file and the MD software doesn't recognize the ligand. In this case, ligand chemistry needs to be explicitly defined by extracting the coordinates of the ligand and separated from main PDB file. A separated topology is constructed manually for the ligand and the information is added in the main topology file. It is also advised to remove external water molecules that are present in the PDB file.

Cleaning the protein structure

1) **Missing loops.** Not all the structures of the PDB are complete; some structures are missing some loops in the protein due to extensive motions of these loops. This leads to smearing out the electron density of these regions and therefore lack of visibility in the corresponding electron density maps. Several websites are online that provide tools to model these missing loops:

- ModLoop (https://modbase.compbio.ucsf.edu/modloop/)
- RAPPER (http://mordred.bioc.cam.ac.uk/~rapper/ca_trace.php)

- FALC-Loop (http://falc-loop.seoklab.org/)
- GalaxyWEB (http://galaxy.seoklab.org)
- ArchPRED (http://manaslu.fiserlab.org/loopred/)
- SuperLooper (http://bioinf-applied.charite.de/superlooper/)
- RCD+ (http://rcd.chaconlab.org)
- ROSIE (http://rosie.rosettacommons.org)

2) **Protonation state.** Next to having all missing residues fixed, another thing that should be taken care of is the protonation state of the acidic and basic protein residues, such as histidine (HIS), aspartic acid (ASP), glutamic acid (GLU), lysine (LYS), arginine (ARG), cysteine (CYS) and tyrosine (TYR). Depending on the local pH, these residues can reside in their acidic (deprotonated) or in their basic (protonated) form (Figure 63). The pK_a's of these seven residue are given in Table 12.

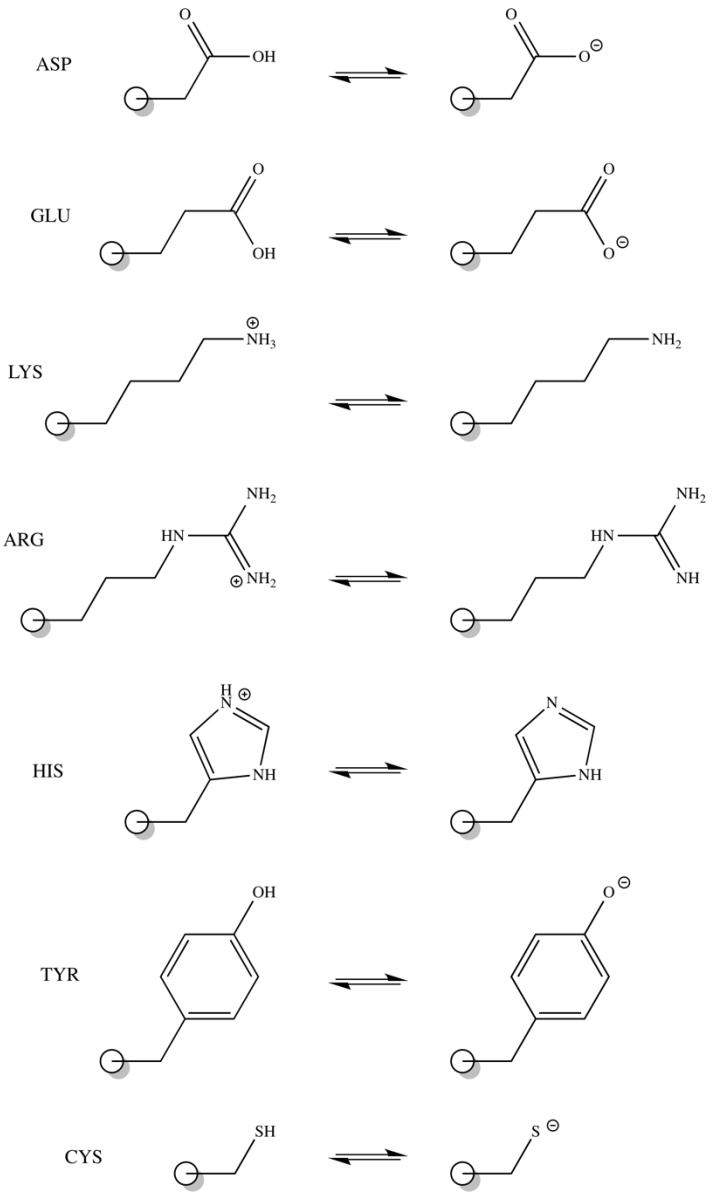


Figure 63. The seven residues which can occur either in their acidic form (left side) or basic form (right side).

Table 12. pK_a values of the acidic form of the seven residues which may occur in either acidic or basic form. Depending on the pH of the local environment (which may differ significantly from the pH of the global environment), residues may be protonated, deprotonated, or reside in both forms. The prevailing form at pH 7.4 can be calculated from the Henderson-Hasselbalch equation $pH = pK_a + log([base]/[acid])$.

Residue	pK_a	Prevailing form at pH 7.4
Aspartic acid	3.65	100% basic (anion)
Glutamic acid	4.25	100% basic (anion)
Lysine	10.53	100% acidic (cation)
Arginine	12.48	100% acidic (cation)
Histidine	6.00	96% basic (neutral), 4% acidic (cation)
Tyrosine	10.07	100% acidic (neutral)
Cysteine	8.33	10% basic (anion), 90% acidic (neutral)

It is important to calculate the pK_a of each ionisable residue in its protein environment, since the local environment of each residue may alter its pK_a significantly. There exist a number of online tools for the calculation of these pK_a's:

- Karlsberg+ (http://agknapp.chemie.fu-berlin.de/karlsberg/)
- H++ (http://biophysics.cs.vt.edu)
- PDB2PQR (http://nbcr-222.ucsd.edu/pdb2pqr_2.0.0/)

3) **Asparagine, glutamine and histidine flips**. Explicit hydrogens are needed for many of the MD force fields to function correctly, and structures determined by crystallography almost never include these hydrogens. As a consequence, hydrogen atoms need to be added prior to starting off a MD simulation, and this process is often automatized by most MD programs. However, a common problem is that the sidechain ends of asparagine (ASN), glutamine (GLN) and HIS residues are easily fitted 180° backwards, since the electron density alone cannot usually distinguish the correct choice of orientation. There are online tools available that can automatically diagnose and correct these types of systematic errors by considering all-atom steric overlaps as well as hydrogen bonding within each local network. The use of such tool is strongly recommended prior to adding any hydrogens:

- MolProbity (http://molprobity.biochem.duke.edu)

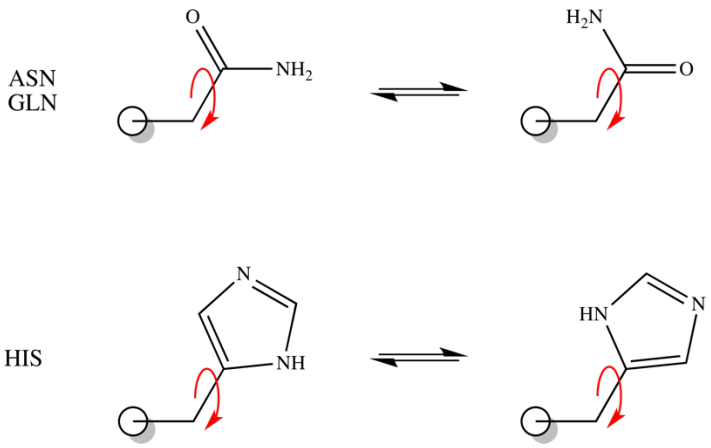


Figure 64. Sidechain flips of ASN, GLN and HIS illustrated. The evidence of such flips can only be deduced from inspection of the local environment around each of these residues.

Setup the box for simulations

1) **Choosing the periodic boundary box**. When simulating biomolecules in a water environment, periodic boundary conditions (PBC) are used to avoid artefacts resulting from boundary effects caused by finite size, and make the system more like an infinite one. During the simulation, only the properties of the original simulation box need to

be recorded and propagated. The minimum-image convention is a common form of PBC particle bookkeeping in which each individual particle in the simulation interacts with the closest image of the remaining particles in the system (Figure 65).

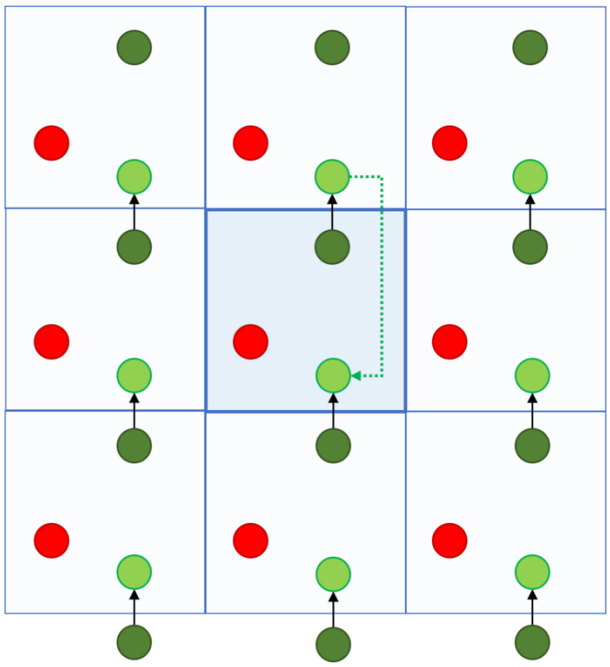


Figure 65. Periodic boundary condition in two dimensions. The blue square denotes the PBC box which contains three particles (red, dark green and light green). In computer simulations, this blue PBC box is the original simulation box, and the others are copies which are called images. A particle which has passed through one face of the simulation box should re-enter through the opposite face, as shown by the dotted arrow.

PBC requires the unit cell to be a shape that will tile perfectly into a three-dimensional crystal. Thus, a spherical or elliptical droplet cannot be used. A cube or rectangular prism is the most intuitive and common choice, but can be computationally expensive due to unnecessary amounts of solvent molecules in the corners, distant from the central macromolecules. An alternative that requires less volume is the truncated octahedron (Figure 66).

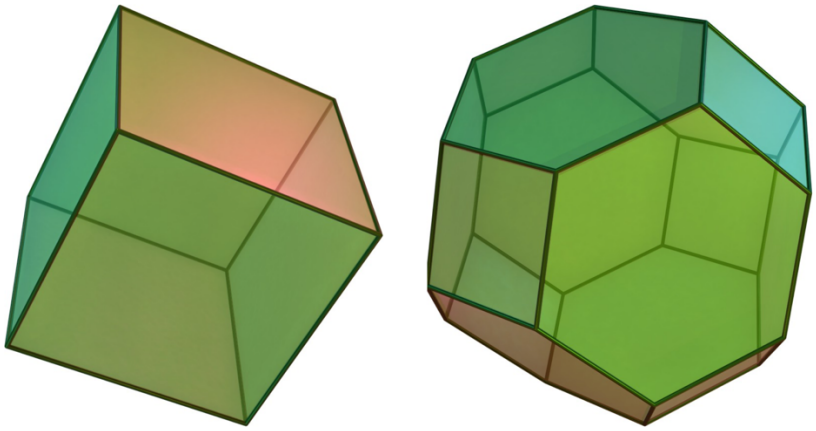
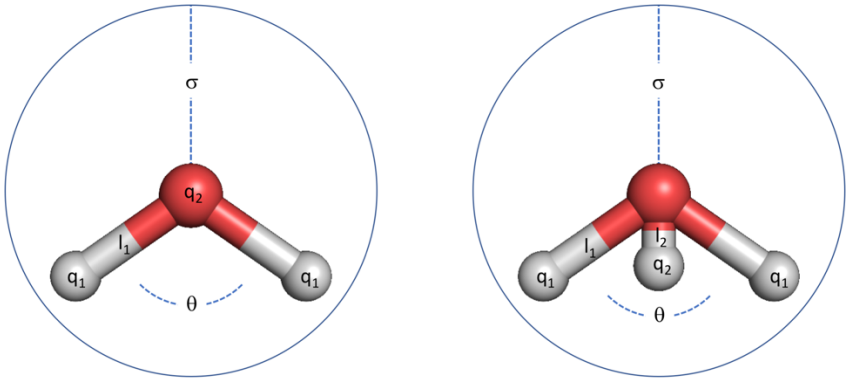


Figure 66. Two commonly used periodic box unit cells. Left: rectangular box. Right: truncated octahedron. This latter shape resembles more a spherical shape and is therefore often more 'economical' in terms of the number of solvent molecules that are needed to fill up the entire volume.

2) **Adding solvent**. MD simulations of biomolecular systems are performed in an environment containing water, which implies that a suitable water model has to be selected. There are a number of water models available (Figure 67), and it is important that the selected water model force field parameters are compatible with the force field parameters of the biomolecular system. In most cases, the TIP3P water model is usually fine.



Model a Model b

Figure 67. Parameters for some water models. 1) ***TIP3P*** *model: topology a, σ = 3.15 Å, θ = 104.5°, q_1 = +0.42, q_2 = -0.84, l_1 = 0.96 Å; 2)* ***SPC*** *model: topology a, σ = 3.17 Å, θ = 109.5°, q_1 = +0.41, q_2 = -0.82, l_1 = 1.00 Å; 3)* ***TIP4P*** *model: topology b, σ = 3.15 Å, θ = 104.5°, q_1 = +0.52, q_2 = -1.04, l_1 = 0.96 Å, l_2 = 0.15 Å.*

3) **Charge-neutralising the system**. The addition of ions is often needed to charge-neutralise the system (total charge of the entire box should be zero) and to reproduce an experimental environment in terms of salt concentration (for example 150 mM NaCl). Most MD software package contain tools to achieve this in a rather automated fashion.

2.2. Maintaining temperature and pressure: ensembles

Canonical ensemble: NVT

In a canonical ensemble, temperature, box volume and the number of particles is kept constant. The box volume is kept constant by keeping the unit cell dimensions unaltered. Temperature is maintained through one of the multiple thermostats that are available. The instantaneous value of the temperature of an unconstrained system is related to the kinetic energy via the particles' linear momenta:

$$T = \frac{2}{Nk_b}\sum_{i=1}^{N}\frac{|\vec{\boldsymbol{p}}_i|^2}{2m_i}$$

where N is the number of atoms, m_i the atomic mass of atom i, k_b the Boltzmann constant, and $\vec{\boldsymbol{p}}_i$ the translational momentum of atom i.

An obvious way to alter the temperature of the system is **velocity scaling**. If the temperature at time t is $T(t)$ and the velocities are multiplied by a factor λ, then the associated temperature change ΔT can be calculated as:

$$\Delta T = \frac{2}{Nk_b}\sum_{i=1}^{N}\frac{(m_i\lambda\vec{v}_i)^2}{2m_i} - \frac{2}{Nk_b}\sum_{i=1}^{N}\frac{(m_i\vec{v}_i)^2}{2m_i}$$

$$\Delta T = \frac{2\lambda^2}{Nk_b}\sum_{i=1}^{N}\frac{(m_i\vec{v}_i)^2}{2m_i} - \frac{2}{Nk_b}\sum_{i=1}^{N}\frac{(m_i\vec{v}_i)^2}{2m_i}$$

$$\Delta T = \lambda^2 T(t) - T(t) \quad \text{hence} \quad \Delta T = (\lambda^2 - 1)T(t)$$

$$T_0 - T(t) = \lambda^2 T(t) - T(t)$$

$$T_0 = \lambda^2 T(t)$$

$$\lambda = \sqrt{\frac{T_0}{T(t)}}$$

The simplest way to control the temperature is thus to multiply the velocities at each time step by the factor $\lambda = \Delta T_0/T(t)$, where $T(t)$ is the current temperature as calculated from the kinetic energy and T_0 being the desired temperature.

A weaker formulation of this approach is the **Berendsen thermostat**. To maintain the temperature the system is coupled to an external heat bath with fixed temperature T_0. The velocities are scaled at each timestep δt, such that the rate of change of temperature is proportional to the difference in temperature:

$$\frac{\delta T(t)}{\delta t} = \frac{1}{\tau}\big(T_0 - T(t)\big)$$

where τ is the coupling parameter which determines how tightly the bath and the system are coupled together. This method gives an exponential decay of the system towards the desired temperature. The change in temperature between successive time steps is:

$$\Delta T = \frac{\delta t}{\tau}\big(T_0 - T(t)\big)$$

and after substituting $(\lambda^2\text{-}1)T(t)$ for ΔT, the scaling factor for the velocities can be deduced :

$$\lambda^2 T(t) - T(t) = \frac{\delta t\big(T_0 - T(t)\big)}{\tau}$$

$$\lambda^2 T(t) = \frac{\delta t\big(T_0 - T(t)\big)}{\tau} + T(t)$$

$$\lambda^2 = \frac{\delta t\big(T_0 - T(t)\big)}{\tau T(t)} + 1$$

$$\lambda^2 = \frac{\delta t}{\tau}\left(\frac{T_0}{T(t)} - 1\right) + 1$$

If τ is large, then the coupling will be weak. If τ is small, then the coupling will be strong. If τ is chosen the same as the timestep δt, the Berendsen thermostat is nothing else than the simple velocity scaling. Values of $\tau \simeq$ 0.1-0.4 ps are typically used in MD simulations of condensed-phase systems, giving $\delta t/\tau \simeq$ 0.01-0.0025.

Velocity scaling artificially prolongs any temperature differences among the components of the system, which can lead to 'hot solvent, cold solute' phenomena, in which the temperature of the solvent is warmer than that of the solute, even though the overall temperature of the system may be at the desired value. One solution is to couple a particular set of particles (for example the solute as one set, and the solvent as a second set) to a separate thermostat. In addition, both the velocity scaling method as well as the Berendsen thermostat do not give rigorous canonical averages. Two methods that do generate rigorous canonical ensembles are the stochastic collisions methods and the extended system method.

In the **stochastic collisions** methods, a particle is randomly chosen at intervals and its velocity is reassigned by random selection from the Maxwell-Boltzmann distribution at the desired temperature. The method is also known as the **Anderson** method. It can be viewed as a method in which the system is in contact with a heat bath that randomly emits 'heat particles' which collide with the atoms in the system.

The **extended system** method was developed by Nose and Hoover, and is therefore also known as the **Nose-Hoover** temperature coupling bath. The mathematical derivatisation will not be covered here.

Isothermal-isobaric ensemble: NPT

In an isothermal-isobaric ensemble, temperature, pressure and the number of particles is kept constant. Many experimental measurements are made under conditions of constant temperature and pressure, and so simulations in the isothermal-isobaric ensemble are most directly relevant to experimental data. Temperature is maintained through one of the multiple thermostats that are available and which are explained in the previous section.

The pressure fluctuates more than other quantities such as the temperature. This can be explained by the fact that the pressure is related to the forces and positions of the particles:

$$p = \frac{k_b T N}{V} + \frac{1}{3V}\overline{\sum_{i<j} \overrightarrow{f_{ij}}\overrightarrow{r_{ij}}}$$

where **p** is the pressure, *T* is the temperature, **V** is the volume and k_b is the Boltzmann constant. The overline is an average, which is a time average in molecular dynamics, $\overrightarrow{f_{ij}}$ is the force on particle *i* exerted by particle *j*, and $\overrightarrow{r_{ij}}$ is the vector going from *i* to *j*: $\overrightarrow{r_{ij}} = \vec{r_j} - \vec{r_j}$. This quantity changes more quickly with the positions than the temperature, hence the greater fluctuations in pressure.

A macroscopic systems maintains constant pressure by changing its volume; hence a simulation in isothermal-isobaric conditions also maintains constant pressure by changing its volume of the PBC unit cell. The amount of volume fluctuation is related to the isothermal compressibility **κ**:

$$\kappa = -\frac{1}{V}\left(\frac{\delta V}{\delta P}\right)_T$$

Larger values of **κ** indicate more easily compressible substances, with larger volume fluctuations that occur at a given pressure than in less compressible substances.

Volume changes in an isobaric ensemble are achieved by changing the size of the unit cell in one, two or all three dimensions (normally in all directions). The relation between the isothermal compressibility **κ** and the box volume is given by:

$$\kappa = \frac{1}{k_b T}\frac{\langle V^2\rangle - \langle V\rangle^2}{\langle V^2\rangle}$$

with $\langle V^2\rangle$ being the time average of the squared volumes, and $\langle V\rangle^2$ the square of the time average of the volumes. The fraction $\frac{\langle V^2\rangle - \langle V\rangle^2}{\langle V^2\rangle}$ corresponds to the mean square volume displacement. Fluctuations in pressure and temperature for a typical MD simulation of a protein in a water box are given in Figure 68.

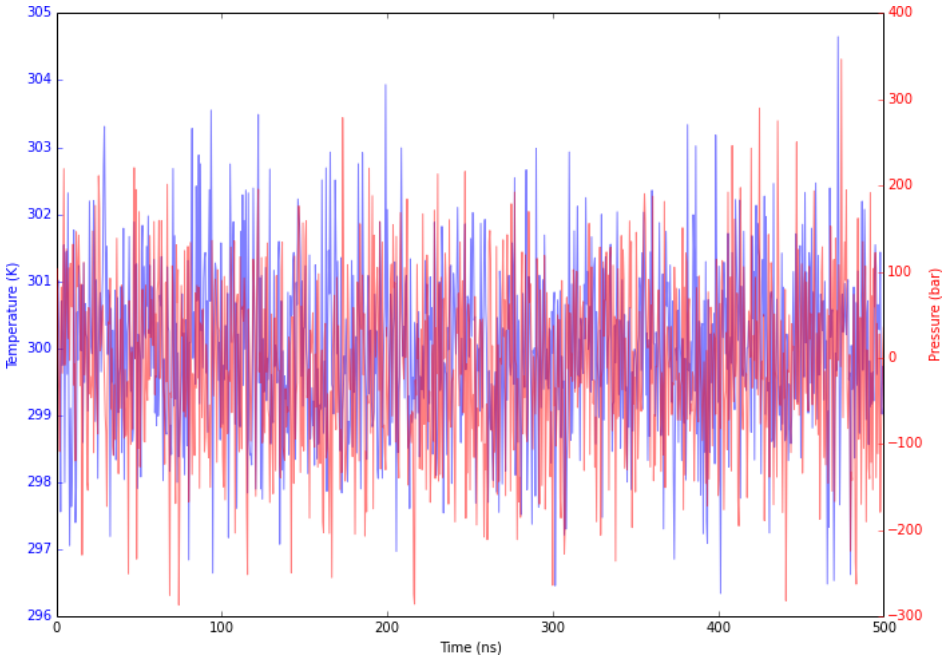


Figure 68. Temperature and pressure fluctuations in a MD simulation of a protein in a water box of dimensions 81.8 x 86.6 x 107.9 Å³. Average simulation temperature is 300.0 K with a standard deviation of 1.2 K, and average simulation pressure is -9.0 bar with a standard deviation of 100.6 bar.

Many of the methods to control pressure are similar to the ones to control temperature. Thus, pressure can be maintained at a constant value by scaling the volume. An alternative is to use a pressure bath in analogy to the temperature bath as found in the Berendsen thermostat. The rate of pressure change is given by:

$$\frac{dP(t)}{dt} = \frac{1}{\tau_p}\Big(P_{ref} - P(t)\Big)$$

with τ_p being the coupling constant, P_{ref} the reference pressure of the bath, and $P(t)$ the actual pressure at time t. To regulate pressure, the volume of the simulation unit cell is scaled by a factor λ, which is achieved by scaling the atomic coordinates towards the centre of the box by a factor $\lambda^{1/3}$. This factor λ can be derived according:

$$\lambda = 1 - \kappa\frac{\delta t}{\tau_p}\big(P - P_{ref}\big)$$

and the new centre of mass coordinates are given by:

$$\vec{r}_i' = \lambda^{1/3}\vec{r}_i$$

2.3. Short- and long-range interactions

Short-range

The most time-consuming part of a MD simulation is the calculation of the non-bonded energies and forces. The number of bond potential, angle potentials and torsion angle potentials in a force field model are all proportional to the number of atoms (order N), but the number of non-bonded terms that need to be evaluated at each step relates to the square of the number of atoms and is thus of order N^2.

Although in principle the non-bonded interactions should be calculated between every pair of atoms in the system, fortunately this is not always justified since the 12-6 Lennard-Jones potential falls off very quickly with distance. At a distance of 3 times $R_{min,ij}$, the van der Waals interaction has fallen to a value which is less than 1% of the corresponding value at a distance of $R_{min,ij}$ (Figure 69).

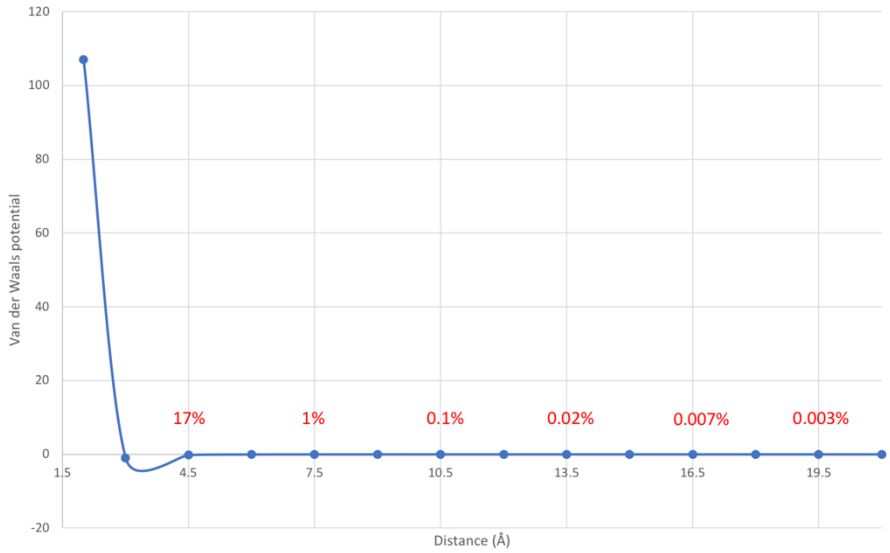


Figure 69. Fall-off of the non-bonded van der Waals interaction between a pair of atoms as a function of distance. $R_{min,ij}$ is set to 3 Å. Values in red is the relative reduction in van der Waals potential compared to the value at 3 Å. At a distance of 9 Å, the van der Waals interaction has fallen to a value which is less than 1% of the corresponding value at a distance of 3 Å.

A common approach to deal with the non-bonded interactions is to use a non-bonded cut-off and to apply the minimum image convention. When a cut-off is applied, the interactions between all pairs of atoms that are further apart than the cut-off value are set to zero. In this minimum image convention, each atom interacts with at most one image of every other atom in the system. When periodic boundary conditions are being used, the cut-off should be less than half the length of the cell, because otherwise a particle could interact with its own image and thus interact with same atom twice (Figure 70). Depending on the size of the biomolecule and the unit cell, a cut-off value of 10 Å or larger is recommend.

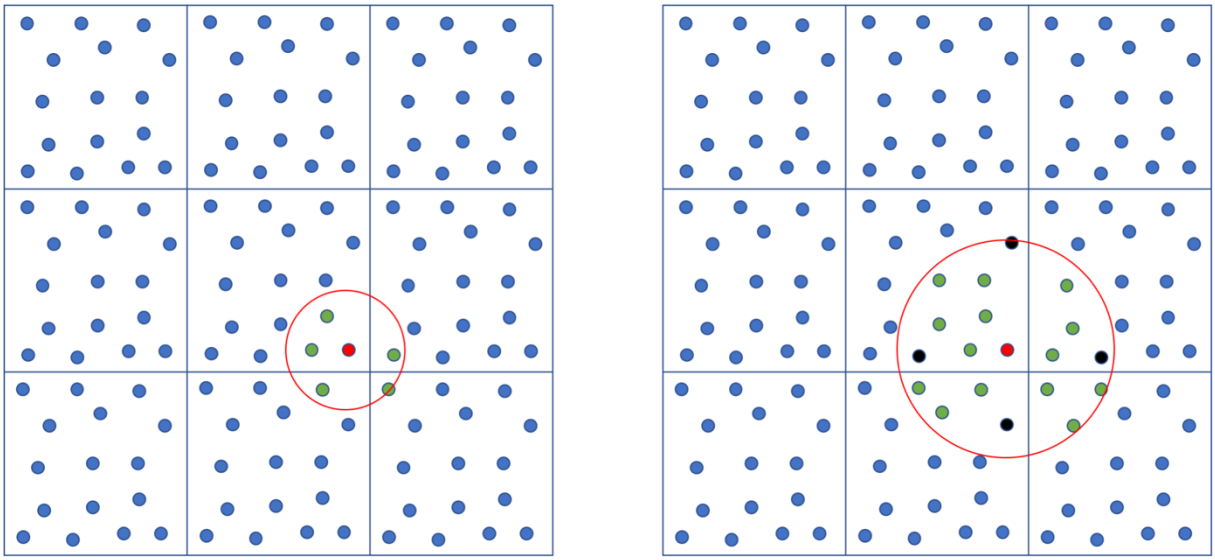


Figure 70. The spherical cutoff and the minimum image convention. Left: spherical cutoff (red circle) that is smaller than half of the unit cell dimension. The red atom is able to interact with all atoms that are positioned within the cutoff radius (shown in green). Right: spherical cutoff with a radius larger than half of the initial cell dimension. This results in interactions with multiple images of the same atoms, as shown in black spheres.

The cut-off distance defines the radius of the sphere containing all atoms for which the interaction with the originating atom is included. Therefore, when implemented as such, it is a hard limit that imposes a discontinuity in the potential energy and forces nearby the cut-off value, and hence may lead to truncation errors and issues with the total energy conservation. To circumvent this, a switching function may be used near the cut-off distance that smooths the van der Waals interaction to zero (Figure 71).

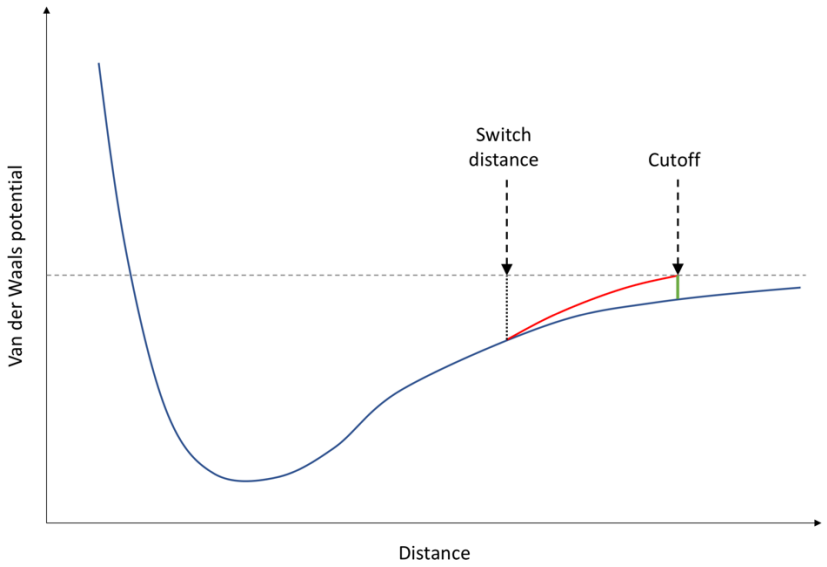


Figure 71. Van der Waals potential (red) and without (green) switching function.

Long-range

The electrostatic potential decays as r^{-1} and can therefore not be truncated at the same cut-off distance that is applicable for the van der Waals interactions. It is important to properly model these long-range non-bonded interactions, and a number of methods have been developed to account for this:

- **Ewald summation** takes advantage of ideas from harmonic analysis to reduce the complexity of the non-bonded interactions the order of $O(N^{3/2})$, which is often still impractical in the case of large systems;
- The **Particle-Mesh Ewald (PME)** method uses Fast Fourier Transform to bring the complexity down to the order of $O(N \log N)$, and this algorithm has been a staple of molecular simulation since its appearance;
- The **Smooth Particle-Mesh Ewald (SPME)** improves on PME by giving a sufficiently smooth energy function whose derivative can be obtained analytically, which results in more realistic simulations due to improved energy conservation.

2.4. Equilibration

In MD simulations, atoms of the macromolecules and of the surrounding solvent have to undergo a relaxation period that usually lasts for tens or hundreds of picoseconds before the system reaches a stationary state. The initial nonstationary segment of the simulated trajectory is typically discarded in the calculation of equilibrium properties. This stage of the MD simulation is called the equilibration stage.

Some equilibration protocols involve gradually increasing the temperature in a stepwise fashion while other more aggressive approaches simply use a linear temperature gradient and heat the system up to the desired temperature. Therefore, equilibration protocols exist in multiple flavours and it is not always straightforward to indicate the most optimal one, although that a gentle equilibration is often better. In Table 13, an example protocol is shown as a matter of illustration.

Table 13. Illustration of a typical equilibration protocol. Position restraints are often placed on the solute heavy atoms to prevent unphysical movements during the initial simulation stages.

Stage	Time (ps)	Temperature (K)	Restraints (kcal/mol/Å²)	Remarks
Minimization	1,000-10,000 steps	0	10	In this stage, steric clashes and close contacts are removed by a steepest descent minimization phase that consists of 1,000-10,000 steps.
Heating	5	0 → 300	10	Atoms are assigned an initial velocity according the desired temperature. To prevent blow-up of the system, temperature (kinetic energy) is added to the system only gradually, for example in steps of 5 K for a period of 5 ps each, resulting in a heating phase that lasts 300 ps (5 ps * 300 ps / 5) in total.
Relax restraints	450	300	10 → 0	The restraints on the heavy atoms are slowly relaxed, going from 10 kcal/mol/Å² to none. This relaxation is achieved over a total time of 450 ps.
Equilibrate	500	300	0	The last stage of the equilibration consists of a normal run at 300 K and without restraints, lasting for at least 500 ps.

3. Analysis

3.1. Production run

The optimal length of a MD production run is hard to define and depends on the size of the system and the timescales of the intrinsic movements one wants to investigate. Some protein folding events may occur on a 1-10 ns timescale, while other folding events may last at least 100-1,000 ns. Also, in order to decide on the optimal length of a MD simulation to study the binding or unbinding event of protein-ligand complex, experimental binding kinetics data of the event might help on deciding.

The output of a typical MD simulation consists of a trajectory file, which is a file containing the time-evolving coordinates of the MD system. Trajectories are sequential snapshots of the simulated molecular system which represents atomic coordinates at specific time periods. Not every timestep is recorded to a trajectory file, but rather at a user-defined interval which depends on the total length of the simulation and the amount of hard disk space that is available. In a typical run, the coordinates are saved to the trajectory file in intervals of 1-100 ps. Each coordinate set in the trajectory file is called a frame, hence the number of frames correspond to the number of coordinate sets. For example, consider a production run of 1,000 ns (which corresponds to 1 μs) with the coordinates saved every 100 ps, this will result in a trajectory file containing 1,000,000 ps / 100 ps = 10,000 coordinate sets or frames.

3.2. Common analysis methods

Visualization

The visualization of the movements of the system is extremely useful, and many software visualization tools have been developed to achieve this goal. Two of these are open-source and provide tools to generate high-quality movies of these movements:

- VMD (Visual Molecular Dynamics) is a product of the Group of Theoretical Biophysics from the University of Illinois (http://www.ks.uiuc.edu/Research/vmd/). It was developed specially for visualization and analysis of such biological systems as proteins, nucleic acids, and molecular systems on the basis of lipids (for example, components of cell membranes). The program compatible with the PDB format and allows user to use various methods of visualization and colouring molecules. VMD is suitable for animation and analysis of phase trajectories obtained from MD simulation.
- PyMol (https://github.com/schrodinger/pymol-open-source) is a molecular visualization system created by Warren DeLano. It is user-sponsored, open-source software, released under the Python License.

PyMOL can produce high-quality 3D images of small molecules and biological macromolecules, such as proteins. The *Py* part of the software's name refers to that it extends, and is extensible by, the programming language Python.

Root-mean square deviation (RMSD)

For each timepoint, the RMSD measures the deviation of the molecule relative to a reference set of coordinates, which is typically the first frame of the trajectory. In order to calculate the deviation, the atom coordinates of all frames are first aligned onto the corresponding coordinates of the first frame, and then the RMSD at each timepoint is calculated:

$$RMSD(t) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(\vec{r}_i(t) - \overrightarrow{r_{i,ref}}\right)^2}$$

in which *N* being the number of atoms that are used in the RMSD calculation at timepoint *t* (this could be all atoms, the heavy atoms, backbone atoms, the Cα atoms, or any other subset of atoms), $\vec{r}_i(t)$ the coordinate of atom *i* at timepoint *t*, and $\overrightarrow{r_{i,ref}}$ the coordinate of the corresponding atom in the reference set.

The RMSD calculation is useful for judging overall motions and equilibration progress, as the RMSD typically first increases and then equilibrates to a steady-state (Figure 72). The *x*-coordinate (abscise) of a RMSD plot is typically the simulation time.

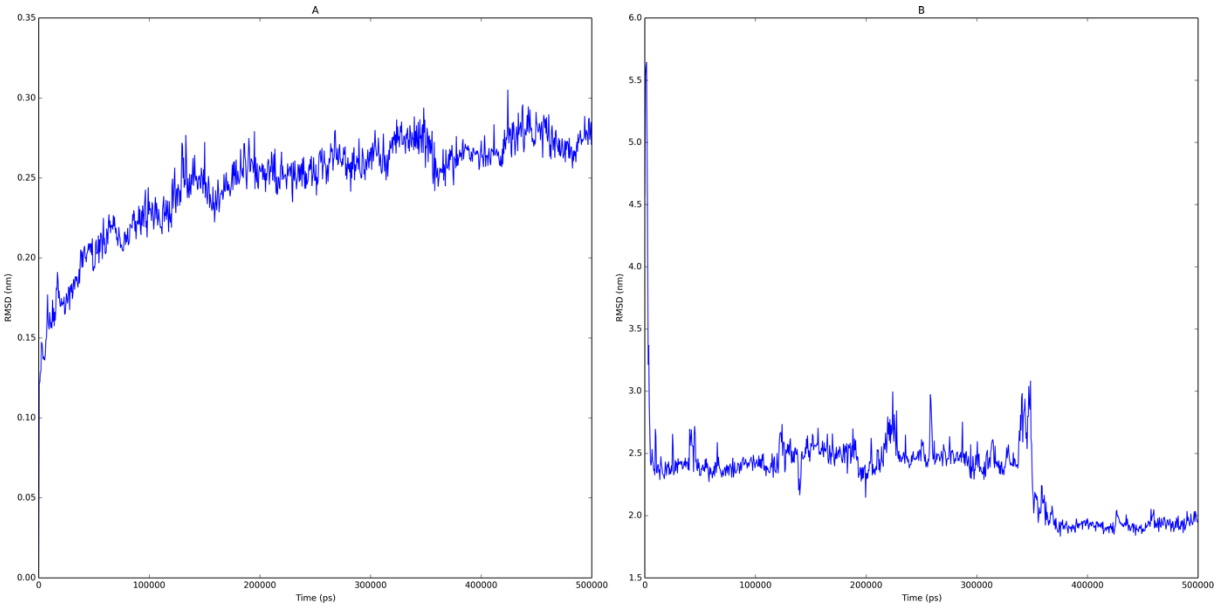


Figure 72. Left: RMSD calculated on the Cα atoms for the PREP protein after fitting the Cα atoms onto those of the first frame. RMSD starts at 0 nm (first frame) and levels out at approximately 0.30 nm. Right: RMSD calculated on atoms of the ligand after fitting the Cα atoms onto those of the first frame. The RMSD of the ligand is much larger (with a maximum of approximately 5.5 nm) compared to the Cα atoms, indicating a larger flexibility and motion.

Root-mean square fluctuation (RMSF)

For each atom, the RMSF measures the fluctuations about a point over an entire simulation, after first aligning the atom coordinates of all frames onto the corresponding coordinates of the first frame:

$$RMSF(i) = \sqrt{\frac{1}{T}\sum_{i=1}^{T}\left(\vec{r}_i(t) - \overrightarrow{r_{i,ref}}\right)^2}$$

with *T* being the total number of frames, $\vec{r}_i(t)$ the coordinate of atom *i* at frame *t*, and $\overrightarrow{r_{i,ref}}$ the coordinate of the corresponding atom in the reference set.

The RMSF is useful for judging which areas of the protein are dynamic or static (Figure 73).

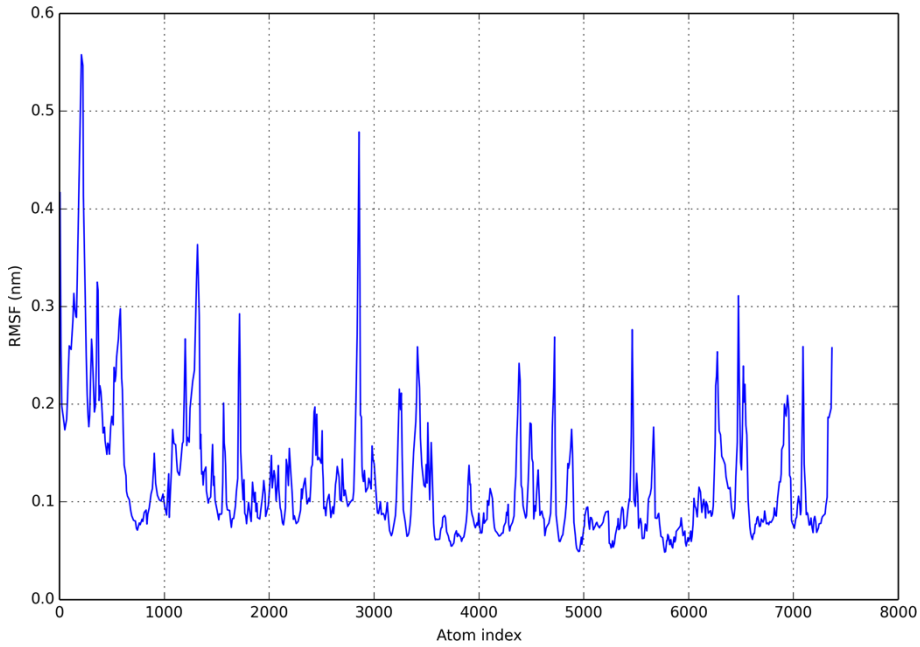


Figure 73. Illustration of a RMSF plot calculated from a 500 ns MD simulation of the PREP protein. The x-axis shows the atom indices of the Cα-carbons of the protein, and the y-axis shows the corresponding calculated RMSF after fitting all frames onto the first frame.

Solvent-accessible surface area (SASA)

The SASA measures the variation of the solvent-accessible surface area of the protein as a function of simulation time, providing information about potential folding or unfolding processes (Figure 74):

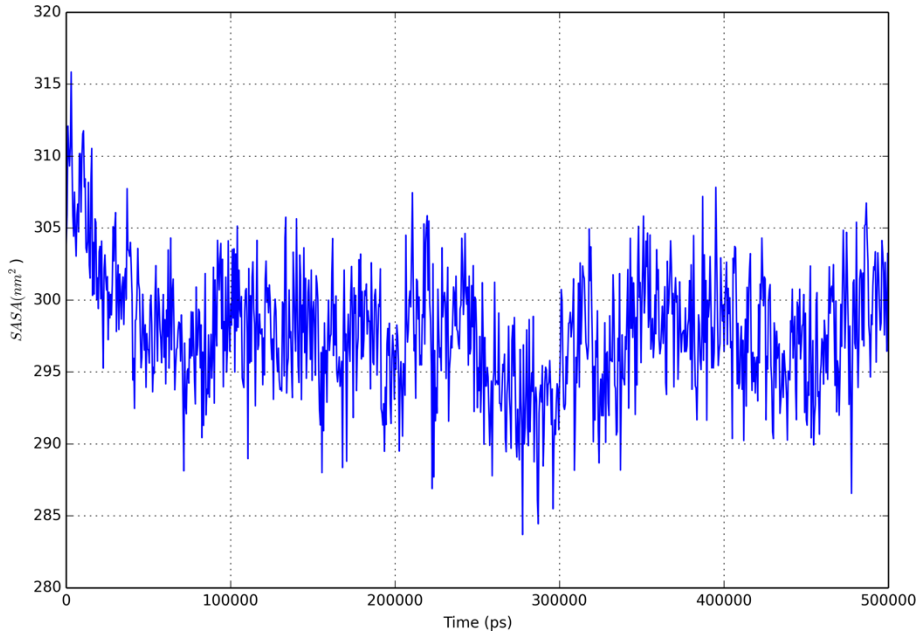


Figure 74. Plot of the SASA as a function of time calculated from a 500 ns MD simulation of the PREP protein. The total SASA (in nm²) is given on the y-axis, and the time (in ps) is given on the x-axis.

Radius of gyration (ROG)

The radius of gyration is the root mean square distance of the protein atoms parts from the molecule's center of mass:

$$ROG(t) = \frac{1}{N}\sum_{i=1}^{N}(\vec{r_i}(t) - \overrightarrow{r_{mean}}(t))^2$$

with *N* being the number of atoms in the protein, $\vec{r_i}(t)$ the coordinate of atom *i* at time *t*, and $\overrightarrow{r_{mean}}(t)$ the mean coordinate of all atoms at time *t*. The larger the value, the 'larger' the molecule. Just like the SASA, the ROG also provides information about potential folding or unfolding processes (Figure 75):

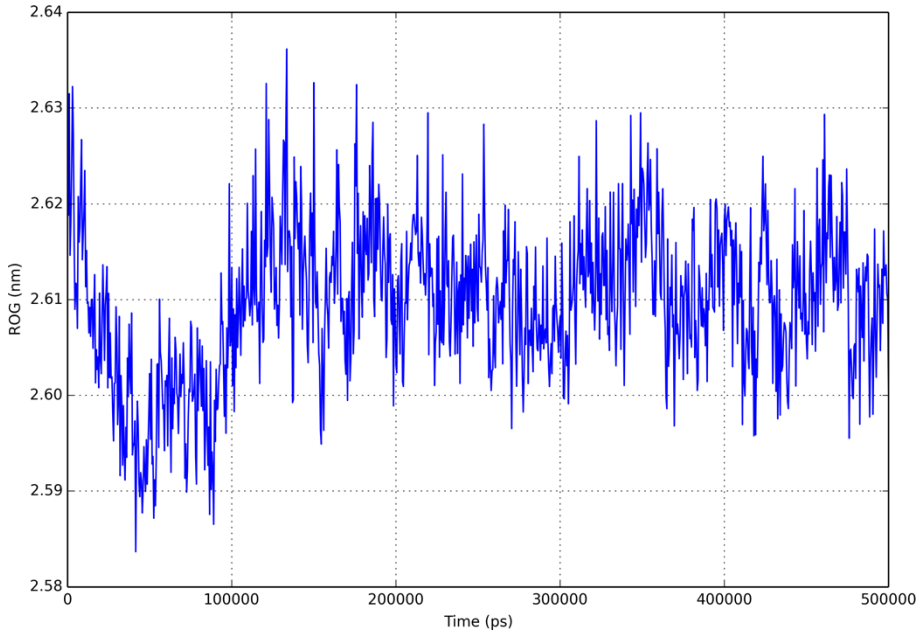


Figure 75. Plot of the ROG as a function of time calculated from a 500 ns MD simulation of the PREP protein. The total ROG (in nm) is given on the y-axis, and the time (in ps) is given on the x-axis. One can see that the protein shows a little 'shrinking around 50,000 ps (50 ns), and then returns to its normal size.

Chapter 8. Virtual screening

1. Pharmacophore searching

1.1. What is a pharmacophore?

A pharmacophore is an abstract description of molecular features that are necessary for molecular recognition of a ligand by a biological macromolecule. IUPAC defines a pharmacophore to be 'an ensemble of steric and electronic features that is necessary to ensure the optimal supramolecular interactions with a specific biological target and to trigger (or block) its biological response'. A pharmacophore is defined as an ensemble of these interactions, or more specific the corresponding chemical features and their relative positions and orientations. It can be seen as a powerful abstraction or representation of small molecule binding to proteins. Essential interactions, corresponding to chemical features, are hydrogen bonding, charge transfer, steric and electrostatic characteristics, and lipophilic interactions. The strength of feature-based pharmacophore models lies in the adequate definition of the pharmacophore points

1.2. Pharmacophore features

Typical pharmacophore features include hydrophobic centers, aromatic rings, hydrogen bond acceptors and donors, positive charges and negative charges. Combinations of these are also possible (Figure 76).

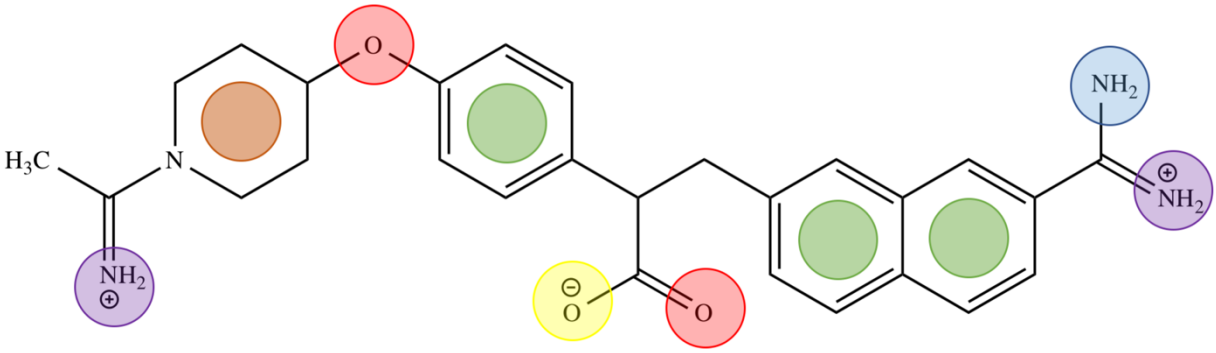


Figure 76. Schematic representation of a pharmacophore model of a ligand. The different pharmacophore types or features are indicated by different colors: purple, positive charge center and hydrogen bond donor; blue, hydrogen bond donor; brown, lipophilic; yellow, negative charge center and hydrogen bond acceptor; red, hydrogen bond acceptor; green, aromatic and lipophilic.

The program Pharao is a widely used pharmacophore searching program [10]. Pharao uses the following pharmacophore point definitions:

- *Aromatic rings*: the identification of aromatic ring pharmacophore points includes ring detection and aromaticity detection. A ring is labeled as aromatic if it is planar, has no exocyclic double bonds and satisfies Hückel's 4n + 2 rule;
- *Hydrogen bond donors*: hydrogen bond donor pharmacophore points correspond to atoms fulfilling the following conditions:
    - atom is nitrogen or oxygen;
    - formal charge of atom is not negative;
    - atom has at least one covalently attached hydrogen atom.
- *Hydrogen bond acceptors*: for the generation of hydrogen bond acceptor points, the following criteria need to be met:
    - atom is nitrogen or oxygen;

- formal charge of atom is not positive;
- atom has at least one available 'lone pair';
- atom is 'accessible'.

- *Charge centers*: atoms with a positive charge will correspond to a positive charge pharmacophore point, while atoms with a negative charge will correspond to a negative charge pharmacophore point.
- *Lipophilic spots*: to generate lipophilic pharmacophore points, a three-step procedure based on the method of Greene et al. [13] is used. First, every atom is assigned a 'lipophilic contribution'. This value is the product of a topology-dependent term *t* and the accessible surface fraction *s*. Term *t* is obtained using some simple heuristic rules that are based on the atom type, and fraction *s* is calculated with a solvent-accessible surface algorithm. Second, when a lipophilic contribution has been assigned to every atom, the next step is to group atoms into lipophilic regions or spots. Grouping atoms into spots is a simple procedure: (1) atoms in a ring of size 7 or less form a group; (2) atoms with three or more bonds, together with their neighbors and not bonded to any other non-hydrogen atom, form a group; (3) the remaining atoms are divided in chains on the basis of their connectivity, and each chain is defined as another group. Rings larger than seven atoms also count as chains. In the third step the lipophilic contribution for every spot is calculated as the summation of the contributions of the individual atoms belonging to that group or spot. If this value exceeds a predefined threshold, the spot corresponds to a lipophilic pharmacophore point for which the center coincides with the geometric center of this spot.

Combination of these six features results in a set of eight different pharmacophore types:

- AROM: aromatic ring
- HDON: hydrogen bond donor
- HACC: hydrogen bond acceptor
- LIPO: lipophilic region
- POSC: positive charge center
- NEGC: negative charge center
- HYBH: HDON + HACC
- HYBL: AROM + LIPO

Each pharmacophore point is represented by its type (eight possibilities as listed above) and a volume, representing the size of the pharmacophore point (Figure 77). The volume is modeled as a Gaussian 3D volume, calculated as:

$$V = \int e^{\left(-\frac{|\vec{m}-\vec{r}|^2}{\sigma}\right)} dr$$

with $\vec{m}$ as the center of the pharmacophore, and σ its spread (related to the volume).

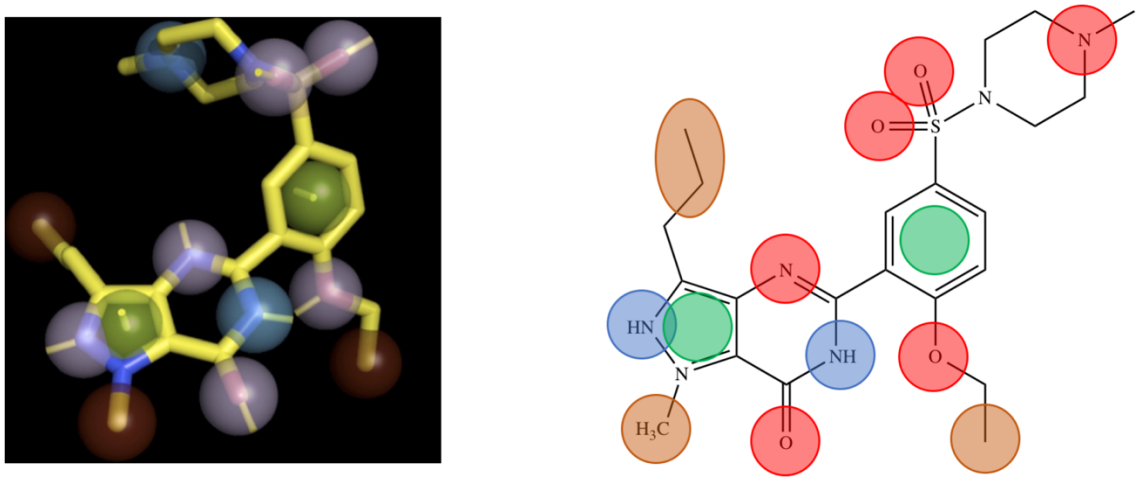


Figure 77. Example of a pharmacophore. The pharmacophore contains 14 pharmacophore points: two AROM points, three LIPO points, two HDON points and six HACC points.

1.3. Pharmacophore alignment

The quantification of the similarity between two pharmacophores can be computed from the overlap volume of the Gaussian volumes of the respective pharmacophores. The goal is to find the subset of matching functional groups in each pharmacophore that gives the largest overlap.

The procedure to compute the volume overlap between two pharmacophores is done in two steps. In the first step, a list of all feasible combinations of overlapping pharmacophore points is generated. In the second step, the corresponding features are aligned with each other using an optimization algorithm. The combination of features that gives the maximal volume overlap is retained to give the resulting score.

Step 1. Feature mapping

To compute the overlap between two pharmacophores, the first step is to define which points from the first pharmacophore can be mapped onto points from the second pharmacophore. A mapping of two pharmacophores consists of a list of points both pharmacophores where corresponding points have a compatible functional group and the internal distance between points is within a given range requirement. This range, as defined by the parameter ε, controls the feasibility of a combination of pharmacophore points.

The procedure starts by generating a list of all feasible pairs of features. First, two points from the first pharmacophore are selected (*a* and *b*) and the distance between them is calculated (d_{ab}). Next, two points with matching features are selected from the second pharmacophore (*c* and *d*) and the distance between these two points is also calculated (d_{cd}). The difference between the two distances is then compared to the sum of the sigmas of the four pharmacophore points. If

$$\varepsilon < \frac{|d_{ab} - d_{cd}|}{\sigma_a + \sigma_b + \sigma_c + \sigma_d}$$

then the combination of the two pairs is said to be feasible. This is also illustrated in Fig. 2. When ε is set to 1.0, this relates to the hard sphere atom model where the spheres are only touching each other and do not overlap. Smaller values of ε indicate that more overlap is required and becomes as such a more stringent selection criterion. Normally, ε is set to 0.5.

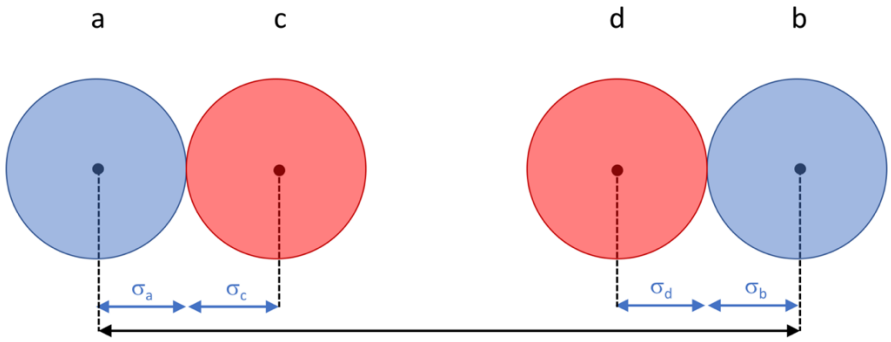


Figure 78. Illustration of the ε parameter. In this example, the difference between d_{ab} (black line) and d_{cd} (red line) is equal to the sum of the four sigmas (blue lines) of the pharmacophore points. When ε is smaller than 1.0, this implies that $|d_{ab} - d_{cd}|$ should be smaller than the sum of the four sigmas and thus the pharmacophore points should overlap.

Once the list of feasible pairs is constructed, they can be combined into larger feasible combinations. A combination of *n* pairs can be extended with any other pair if that pair is feasible and compatible with all the *n* pairs of the combination. This process is combinatorial in nature and the number of possible combinations grows very fast with the number of pharmacophore points. The ε parameter leads to a reduction of the number of feasible combinations.

Step 2. Alignment phase

Starting from the set of feasible combinations, the combination that gives the largest volume overlap is searched for. For every combination, the procedure starts by orienting the first and second pharmacophore subsets such that their geometric centre and their principal axes of inertia coincide. Next, by applying a constrained gradient-ascent to the rigid-body rotation of the second pharmacophore, the maximal volume overlap is determined between the two subsets. The rotational part is implemented using quaternion algebra. The use of the Gaussian representation of pharmacophore points offers an elegant way to compute the gradient and Hessian of the volume overlap. The result of the optimization procedure is the rotational angle and axis that gives the optimal overlap, and an alignment score which quantifies this overlap.

The complete alignment procedure starts from the subset with the largest number of matching points and computes the optimal volume overlap of this combination. Next, the smaller combinations are processed until the highest volume overlap so far is higher than the maximum volume overlap any smaller combination hypothetically can achieve. The rationale is that the volume overlap has an upper boundary that depends on the number of features to align. If the current best overlap is larger than this upper bound then there is no need to compute the alignment of smaller subsets since the score will never be larger than the current best.

Calculating the similarity between a pair of pharmacophores, a number of different measures can be used depending on the desired outcome. The most important are:

$$TANIMOTO = \frac{V_{overlap}}{V_1 + V_2 - V_{overlap}}$$

$$TVERSKY = \frac{V_{overlap}}{V_1}$$

with $V_{overlap}$ being the volume overlap of the matching subset of pharmacophores points, V_1 the volume of the first pharmacophore, and V_2 the volume of the second pharmacophore. The *TANIMOTO* measure is well known from bit vector comparison and is the default measure to score similarity between pharmacophores. The *TVERSKY* measure is to identify compounds having a pharmacophore that is a superset of the first pharmacophore (V_1). All two metrics are returning a score between 0 and 1.

2. Docking

Docking is a method which predicts the preferred orientation of one molecule to a second when bound to each other to form a stable complex. Knowledge of the preferred orientation in turn may be used to predict the strength of association or binding affinity between two molecules using a scoring function. Molecular docking is one of the most frequently used methods in structure-based drug design, due to its ability to predict the binding-conformation of small molecule ligands to the appropriate target binding site. Characterisation of the binding behaviour plays an important role in rational design of drugs as well as to elucidate fundamental biochemical processes.

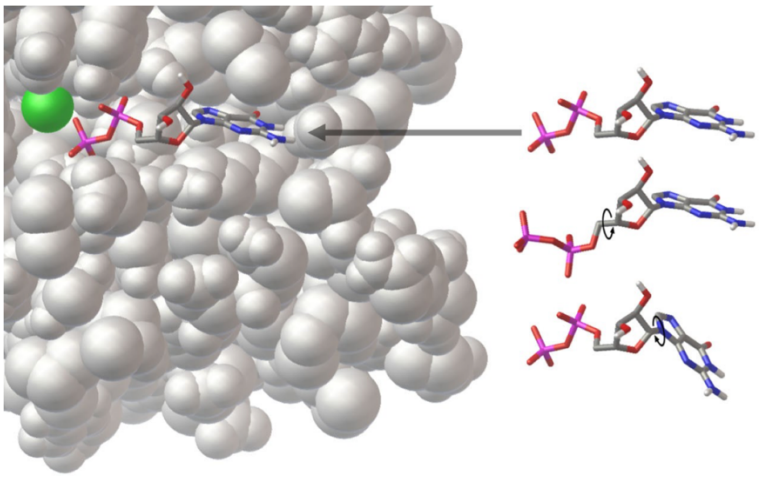


Figure 79. One can think of molecular docking as a problem of 'lock-and-key', in which one wants to find the correct relative orientation of the 'key' which will open up the 'lock'. Here, the protein can be thought of as the 'lock' and the ligand can be thought of as a 'key'. Molecular docking may be defined as an optimization problem, which would describe the 'best-fit' orientation of a ligand that binds to a particular protein of interest.

Molecular docking research focusses on computationally simulating the molecular recognition process. It aims to achieve an optimized conformation for both the protein and ligand and relative orientation between protein and ligand such that the free energy of the overall system is minimized (Figure 79).

To perform a docking screen, the first requirement is a structure of the protein of interest. Usually the structure has been determined using a biophysical technique such as x-ray crystallography or NMR spectroscopy, but can also derive from homology modeling construction. This protein structure and a database of potential ligands serve as inputs to a docking program. The success of a docking program depends on two components: the search algorithm and the scoring function (Figure 80).

Figure 80. Docking is the repeated process of searching and scoring, until certain predefined criteria are met.

2.1. Search algorithms

The search space consists of all possible orientations and conformations of the protein paired with the ligand. However, in practice with current computational resources, it is impossible to exhaustively explore the search space - this would involve enumerating all possible distortions of each molecule (molecules are dynamic and exist in an ensemble of conformational states) and all possible rotational and translational orientations of the ligand

relative to the protein at a given level of granularity. Most docking programs in use account for the whole conformational space of the ligand (flexible ligand), and several attempt to model a flexible protein receptor. Each 'snapshot' of the pair is referred to as a pose.

A variety of conformational search strategies have been applied to the ligand and to the receptor. These include:

- Shape-complementarity methods
- Molecular dynamics simulations
- Genetic algorithms

Shape-complementarity methods

As being the most common technique used in many docking programs, shape-complementarity methods focus on the match between the receptor and the ligand in order to find an optimal pose. Programs include:

- DOCK (http://dock.compbio.ucsf.edu)
- FRED (https://docs.eyesopen.com/oedocking/fred.html)
- GLIDE (https://www.schrodinger.com/glide), and
- SURFLEX (https://omictools.com/surflex-dock-tool).

Most methods describe the molecules in terms of a finite number of descriptors that include structural complementarity and binding complementarity. Structural complementarity is mostly a geometric description of the molecules, including solvent-accessible surface area, overall shape and geometric constraints between atoms in the protein and ligand. Binding complementarity considers features like hydrogen bonding interactions, hydrophobic contacts and van der Waals interactions to describe how well a particular ligand will bind to the protein. Both kinds of descriptors are conveniently represented in the form of structural templates which are then used to quickly match potential compounds that will bind well at the active site of the protein.

As one of the most established docking programs in the last decades, DOCK will be used as an example to explain the principles behind shape-complementarity methods. The DOCK procedure consists of three steps [11]:

1. Representation of the receptor and ligand structures;
2. Matching of the receptor and ligand representations;
3. Optimization of the ligand within the binding site.

The **representation step** consists of generating a set of spheres that fill all pockets and grooves on the surface of the receptor molecule. These spheres are collected into a number of presumptive 'binding' sites, and each site is then examined for geometric matching with the ligand. The ligand molecule is also represented by a set of spheres that approximately fill the space occupied by the ligand.

The molecular surfaces of receptor and ligand are required for this step. This surface is a collection of points and the vectors normal to the surface at each point (Figure 81).

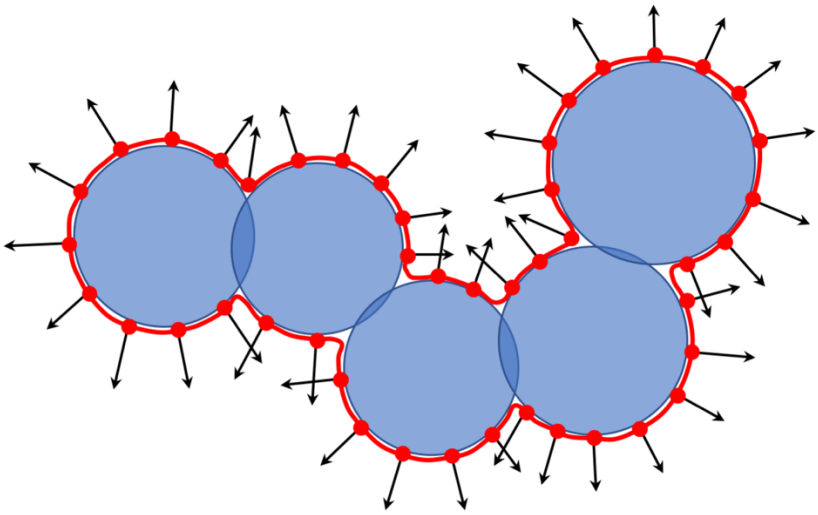


Figure 81. The blue spheres describe a receptor composed of five atoms, the red line the surface of the receptor, and the red dots are a set of points equally distributed on the surface. The black arrows are the normals of the surface at each point.

To reduce the number of points on the surface area, a more compact representation is then generated by constructing a set of spheres with the following properties:

a) Each sphere touches the molecular surface at two points (*i*, *j*) and has its centre on the surface normal from point *i* (Figure 82);

b) Each receptor sphere lies on the outside of the surface and each ligand sphere lies on the inside of the surface.

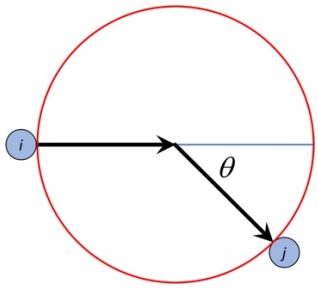


Figure 82. Sphere generated tangent to the surface points i and j with its center on the surface normal at point i.

It can be shown that, for a surface composed of *n* points, *n*-1 spheres can be constructed at each of the surface points. Therefore, the spheres that are generated in this way are reduced in two ways. First, of each of the *n*-1 spheres at each surface point, only the sphere with the smallest radius is kept. Second, for each smallest sphere, the angle formed by the two surface points, *i* and *j*, and the sphere centre, is calculated (Figure 82), and only those spheres are retained for which the angles are less that 90°.

The **matching rule** is based on the comparison of the internal distances in both receptor and ligand. A ligand sphere can be paired with a receptor sphere if the internal distances of all ligand spheres can be matched with all the internal distances of the receptor set, within some error limit on each distance. However, the complexity of the matching problem is that not all combinations can be tried, and for this reason a more pragmatic approach has been taken. First, each ligand sphere *i* is paired with each receptor sphere *k*, and all distances between ligand sphere *i* and the remaining ligand spheres *j* (d_{ij}) is calculated, as well as all distances between receptor sphere *k* to all remaining receptor spheres *g* (d_{kg}). A second pair of spheres ($j = g$) is assigned so that a maximum number of spheres obey the condition:

$$\left|d_{ij} - d_{kg}\right| < 1$$

in which the number '1' is a distance cut-off that has been chosen experimentally.

Once the best second pair of ligand-receptor spheres has been identified, a third pair of spheres is selected subject to the additional constraint that the distances from the new spheres to the previously assigned pairs must also obey the error check. This procedure continues until no further pairs can be assigned.

In the third phase, the **optimization stage**, all suggested pairings are explored. It carries out the rotation of the ligand spheres onto the corresponding receptor spheres using a least-squares minimization algorithm. The generated rotation/translation matrix is then applied to all ligand atoms and the resulting conformations are subjected to a user-defined scoring function for ranking and classification (see further).

Molecular dynamics simulations

In this approach, proteins are typically held rigid, and the ligand is allowed to freely explore their conformational space. The generated conformations are then docked successively into the protein, and an MD simulation consisting of a simulated annealing protocol is performed. This is usually supplemented with short MD energy minimization steps, and the energies determined from the MD runs are used for ranking the overall scoring. Although this is a computer-expensive method (involving potentially hundreds of MD runs), it has some advantages as no specialized energy/scoring functions are required. MD forcefields can typically be used to find poses that are reasonable and can be compared with experimental structures.

Because in this approach the protein is kept rigid, one might miss important protein conformations that are involved in ligand binding. One approach that aims to address this issue is ensemble-based docking. With this technique, ligands are docked to an ensemble of rigid protein conformations. Molecular dynamics simulations are used to generate the ensemble of protein conformations for the subsequent docking.

Genetic algorithms

Two of the most used docking programs belong to this class of docking approaches:

- GOLD (https://www.ccdc.cam.ac.uk/solutions/csd-discovery/components/gold/)
- AutoDock (http://autodock.scripps.edu)

Genetic algorithms allow the exploration of a large conformational space - which is basically spanned by the protein and ligand jointly in this case - by representing each spatial arrangement of the pair as a 'gene' with a particular energy. The entire genome thus represents the complete energy landscape which is to be explored. The simulation of the evolution of the genome is carried out by cross-over techniques similar to biological evolution, where random pairs of individuals (conformations) are 'mated' with the possibility for a random mutation in the offspring. These methods have proven very useful in sampling the vast state-space while maintaining closeness to the actual process involved.

Although genetic algorithms are quite successful in sampling the large conformational space, many docking programs require the protein to remain fixed, while allowing only the ligand to flex and adjust to the active site of the protein. Genetic algorithms also require multiple runs to obtain reliable answers regarding ligands that may bind to the protein. The time it takes to typically run a genetic algorithm in order to allow a proper pose may be longer, hence these methods may not be as efficient as shape complementarity-based approaches in screening large databases of compounds. Recent improvements in using grid-based evaluation of energies, limiting the exploration of the conformational changes at only local areas (active sites) of interest, and improved tabling methods have significantly enhanced the performance of genetic algorithms and made them suitable for virtual screening applications.

2.2. Scoring functions

The search algorithms of the docking programs generate a large number of potential ligand poses, of which some can be immediately rejected due to clashes with the protein. The remainder are evaluated using some scoring function, which takes a pose as input and returns a number indicating the likelihood that the pose represents a favourable binding interaction and ranks one ligand relative to another. Scoring functions can be categorized into three types:

- Forcefield-based scoring functions
- Empirical scoring functions
- Knowledge-based scoring functions

Forcefield-based scoring functions

Most scoring functions are physics-based molecular mechanics force fields that estimate the energy of the pose within the binding site. The various contributions to binding can be written as an additive equation:

$$E = W_{VDW}\sum_{i,j}\left(\frac{A_{ij}}{r_{ij}^{12}}+\frac{B_{ij}}{r_{ij}^{6}}\right) + W_{hbond}\sum_{i,j}p(\theta)\left(\frac{C_{ij}}{r_{ij}^{12}}+\frac{D_{ij}}{r_{ij}^{10}}\right) + W_{elec}\sum_{i.j}\frac{q_i q_j}{r_{ij}} + W_{sol}\sum_{i.j}(S_i V_j + S_j V_i)e^{\left(-r_{ij}^2/2\sigma^2\right)}$$

The components in this example consist of van der Waals (W_{VDW}) and electrostatic (W_{elec}) interactions, solvent effects (W_{sol}) and hydrogen bonding interactions (W_{hbond}).

The **van der Waals interaction term** describes the non-bonded interactions between non-polar atoms such as hydrophobic interactions. It is a standard Lennard-Jones potential function and the W_{VDW} is a scaling factor to bring the van der Waals contributions on the same scale as the other parameters.

Hydrogen bond interactions are modelled in a similar way as the van der Waals interactions, with the exception that a 12-10 Lennard-Jones function is used with different parameters, and that a penalty function $p(\theta)$ has been added to account for deviations of the hydrogen bond from ideal geometry (Figure 83).

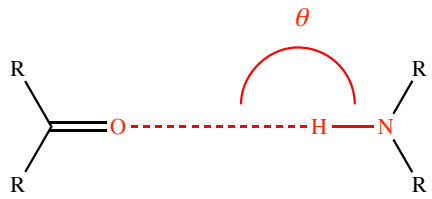


Figure 83. Illustration of the θ angle to calculate hydrogen bond geometry. Shown is a hydrogen bond between the O and H-N. In a hydrogen bond of optimal strength, this O-H-N angle should be close to 180°. Deviations from this optimal value results in weaker hydrogen bonding interactions.

Electrostatic interactions are normally represented as a constant W_{elec} term multiplied by the product of the partial charges and divided by the atomic distances.

Finally, **solvent effects** can be modelled in a variety of ways; the method which is shown in the equation above is based on the surfaces of ligand and receptor atoms (S_i and S_j), scaled by a solvent parameter (V_i and V_j) and the distance between the atoms (r_{ij}).

Empirical scoring functions

Empirical scoring functions are based on counting the number of various types of interactions between the two binding partners. Counting may be based on the number of ligand and receptor atoms in contact with each other or by calculating the change in solvent accessible surface area (ΔSASA) in the complex compared to the uncomplexed ligand and protein. The coefficients of the scoring function are usually derived by fitting using multiple linear regression methods, in which experimental structures and corresponding binding affinities are used as model system. These interactions terms of the function may include for example:

$$\Delta G = f_{hbonds}\Delta G_{hbonds} + f_{polar-apolar}\Delta G_{lpolar-apolar} + f_{nrot}\Delta G_{nrot} + f_{apolar-apolar}\Delta G_{apolar-apolar}$$

with f_{hbonds} a function representing the number of hydrogen bonds and ΔG_{hbonds} the corresponding parameter resulting from the least-squares fit, $f_{polar\text{-}apolar}$ and $\Delta G_{polar\text{-}apolar}$ the function and parameter accounting for the unfavourable hydrophobic-hydrophilic contacts between receptor and ligand, $f_{apolar\text{-}apolar}$ and $\Delta G_{apolar\text{-}apolar}$ the function and parameter accounting for the favourable hydrophobic-hydrophobic contacts, and f_{nrot} and ΔG_{nrot} the function and parameter representing the number of rotational bonds on the ligand (accounting for the loss of entropy upon ligand binding).

Knowledge-based scoring functions

An third scoring function approach is to derive a knowledge-based statistical potential for interactions from a large database of protein-ligand complexes, such as the Protein Data Bank (https://www.rcsb.org) or the Cambridge Structural Database (https://www.ccdc.cam.ac.uk/solutions/csd-system/components/csd/), and evaluate the fit of the pose according to this inferred potential. Knowledge-based scoring functions are based on statistical observations of intermolecular close contacts in large 3D databases which are used to derive 'potentials of mean force'. This method is founded on the assumption that close intermolecular interactions between certain types of atoms or functional groups that occur more frequently than one would expect by a random distribution are likely to be energetically favourable and therefore contribute favourably to binding affinity.

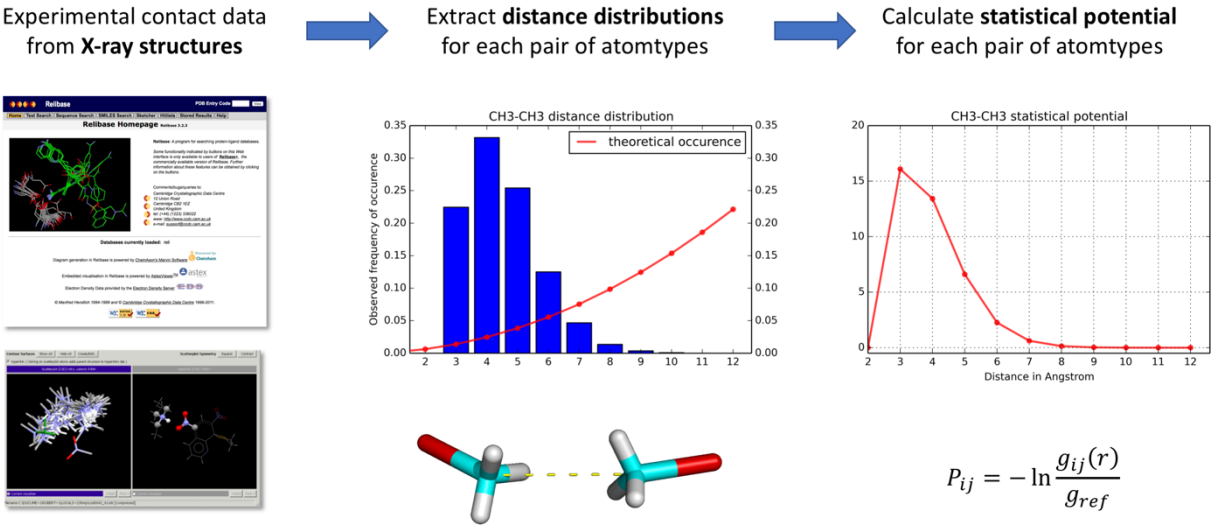


Figure 84. Development of a knowledge-based scoring function explained for the methyl-methyl interaction. Knowledge about the particular interaction type is initially extracted from structural databases from the number of methyl-methyl interactions is counted and also the corresponding geometrical features, such as the carbon-carbon distance. This experimental methyl-methyl distance distribution is subsequently compared to a theoretical distribution, i.e. 'what should be the distribution from a pure theoretical point of view, without the presence of favorable or unfavorable interaction terms'. Dividing the observed distribution g_{ij} by the reference distribution g_{ref} yields the potential of mean force that describes the potential between two methyl groups as a function of distance.

Scope of the scoring functions

Docking experiments are used for pose prediction or for compound selection. In the case of **pose prediction**, the question that needs to be answered is 'given a ligand and a protein, how does the ligand bind to the protein?'. Put in other words, the docking program needs to predict a valid binding conformation but the corresponding predicted binding affinity should not be accurate. Emphasis in this case is on predicting the correct binding pose rather than on predicting binding affinity values. However, in the case of **compound selection**, the scope is different. In this situation, one often wants to select from a large compound database those compounds that will bind to the protein under study, and therefore emphasis is on predicting correct binding affinities rather than correct binding poses.

Forcefield-based scoring functions are most suitable for correct pose prediction but less suited for compound selection. The reason for this is that the value of the scores generated by a forcefield-based function are dependent on the ligand type and its size, and therefore more difficult to compare between different ligands. However, the calculated scores are accurate enough to compare different docking poses of the same ligand, and for that reason forcefield-based scoring functions are suitable for pose prediction questions.

Empirical scoring functions are better suited for selection of compounds from a database screen. First of all, because of their simplicity, these scoring functions are fast to calculate and therefore suited to screen millions of compounds, and secondly, because these functions were derived by least-square fitting against experimental binding affinities, the resulting scores are also less depending on the ligand type and size and therefore

comparable between different ligands. The simplicity of the function has a drawback in the sense that the results are less sensitive to small variation in the docking poses, hence less suitable for pose prediction questions.

Knowledge-based scoring functions have been shown to be useful in both pose prediction as well as database screening cases. These functions are accurate enough to pick up small variations in the actual binding poses, and the score values are rather independent on the ligand type and size, hence also applicable to database screening. The main issue with these kind of scoring functions is the fact that there are a large number of structures from X-ray crystallography for complexes between proteins and high affinity ligands, but comparatively fewer for low affinity ligands as the later complexes tend to be less stable and therefore more difficult to crystallize. Scoring functions trained with this high affinity data can therefore dock high affinity ligands correctly, but as a consequence they will also give plausible docked conformations for ligands that do not bind. This gives a large number of false positive hits, i.e., ligands predicted to bind to the protein that actually don't when tested experimentally.

References

1 Weininger, D. (1988) 'SMILES, a chemical language and information system. 1. Introduction to methodology and encoding rules', *J. Chem.Inf. Comput. Sci.* **28**, 31-36.

2 Lee, B.; Richards, F.M. (1971). 'The interpretation of protein structures: estimation of static accessibility'. *J. Mol. Biol.* **55**, 379-400.

3 Shrake, A.; Rupley, J.A. (1973). 'Environment and exposure to solvent of protein atoms. Lysozyme and insulin'. *J. Mol. Biol.* **79**, 351-371.

4 Cereto-Massagué, A.; Ojeda, M.J.; Valls, C.; Mulero, M.; Garcia-Vallvé, S.; Pujadas, G. (2015). 'Molecular fingerprint similarity search in virtual screening', *Methods* **71**, 58-63.

5 Spangenberg, T.; Burrows, J.N.; Kowalczyk, P.; McDonald, S.; Wells, T.N.C.; Willis, P. (2013) 'The Open Access Malaria Box: A drug discovery catalyst for neglected diseases', *PLoS ONE* **8**, e62906.

6 Leach, A. (2001) 'Molecular modelling. Principles and applications'. 2nd Ed., Pearson Education Ltd. ISBN 0-582-38210-6.

7 Ryckaert, J.P.; Ciccotti, G.; Berendsen, H.J.C. (1977) 'Numerical integration of the cartesian equations of motion of a system with constraints: molecular dynamics of n-alkanes', *J. Comput. Phys.* **23**, 327-341.

8 Hess, B.; Bekker, H.; Berendsen, H.J.C.; Fraaije, J.G.E.M. 'LINCS: A linear constraint solver for molecular dynamics', *J. Comput. Chem.* **18**, 1463-1472.

9 Andersen, H.C. (1983) 'Rattle: a "velocity" version of the shake algorithm for molecular dynamics simulations', *J. Comput. Phys.* **52**, 24-34.

10 Taminau, J.; Thijs, G.; De Winter, H. (2008) 'Pharao: pharmacophore alignment and optimization', *J. Mol. Graph. Model.* **27**, 161-169.

11 Kuntz, I.D.; Blaney, J.M.; Oatley, S.J.; Langridge, R.; Ferrin, T.E. (1982) 'A geometric apporach to macromolecule-ligand interactions', *J. Mol. Biol.* **161**, 269-288.