iradina manual

Christian Borschel

4th of August, 2014

Contents

1 Introduction

Iradina (**i**on **r**ange **a**nd **d**amage **i**n **na**nostructures) is a Monte Carlo program which simulates the transport of ions through nanostructures using the binary collision approximation (BCA). It works similar to other BCA ion transport simulators like the TRIM program [ZBL85], but in contrast to TRIM, *iradina* uses a 3-dimensional rectangular grid to define the target instead of layers. This allows a more accurate representation of nanostructures as in TRIM. *Iradina* is not dynamic, meaning that the target stays the same all the time, things like ion beam mixing cannot be simulated and the developement of a surface under FIB irradiation can also not be simulated. If you need that, look at TRI3DYN [Möl13]. *Iradina* borrows some extremely fast procedures from the nice and open *Corteo* code [Sch07, Sch08], allowing it to do calculations much faster than TRIM. However, the purpose of *iradina* is not to replace TRIM, but to offer an alternative for simulating ion beam irradiation of nanostructures.
Iradina itself has no graphical user interface (and no non-graphical user interface either). It just reads all its input parameters from some files, calculates something, and writes the results to some other files. However, the configuration files are human-readable and thus not so difficult to maintain. Apart from that, a separate user interface is available for Windows, which can be used for an easy definition of the target structure.

This manual mainly describes how to use the program, the underlying physics models are not discussed here. Detailed descriptions can be found in [BR11, Bor12]. If you are not familiar with Monte Carlo ion beam transport simulations, I strongly recommend to take a look into the following references [ZBL85, Sch07] or some other text on the topic. To quote N.P. Barradas: "Never trust a computer code". If you still consider using a computer code to simulate physics, you should at least be aware of how the code works to correctly interpret the simulation results. The real world can be very different from the simulation...

This manual refers to version 1.0.6 of *iradina* and version 1.0 of the user interface.

2 Installation

Iradina is released under the terms of the GNU general public license, version 3 [GPL07]. It is written in plain c so it should be able to run on most common computers and operating systems. Like in Corteo, the only restriction is that the computer must comply with the IEEE 754 standard to represent 32-bit float values (details explained in [Sch07]). Precompiled versions for Windows (32 and 64 bit) are available for download. Other than that, *iradina* has been tested successfully on various Linux systems starting from 2.6 and up using the GCC version 4.3.2 or later for compilation. *Iradina* uses the Corteo databases for stopping powers and solutions to the scattering integral. The use of databases makes the program very fast, however, this also means that *iradina* requires quite some memory. While about 100 MBytes may be enough for many applications, a complex simulation with many different element may require far more (see appendix B).

2.1 Installation procedure part I: obtaining the necessary files

You need two things to run *iradina*:

1. Corteo database files (the scattering matrix and electronic stopping data)
2. *Iradina* executable file
   Note: At the moment, there are two versions of the Corteo scattering matrix with different precision, the older one uses the four most siginificant bits (4-MSB) of the float mantissas and the new one uses the six most significant bits (6-MSB). *Iradina* can in principle use both, but depending on the number of used MSBs you need to use or compile the appropriate *iradina* executable. As yet, I have not observed any significant differences in *iradina* simulations when using the 4-MSB or 6-MSB-version. Thus, I recommend using the 4-MSB version at the moment because the 6-MSB version requires much more memory (which may quickly exhaust a common desktop PC).

1. There are different ways to obtain the Corteo database files:
   - The simplest way: download the packed files from the *iradina* homepage (`www.iradina.de`).
   - Download the files from the Corteo homepage [Sch07]. Note: Starting from version 20130715, Corteo uses the 6MSB scattering matrix. If you download that one, you must use the appropriate *iradina* executable.
   - Download Corteo and build the database yourself using "corteosetup" (follow the instructions on the Corteo website). This requires more time and work, however apart from learning something about the mechanism, you will be able to chose the interaction potential (the precomputed version use the "universal" potential [ZBL85]).
2. There are two ways to obtain the iradina executable:
   - Download the precompiled executables (Windows) from the *iradina* website.
   - Download the source from the *iradina* website and compile the executable from source (Linux, Mac, any operation system having a c compiler).

There are several precompiled executables, you should select the one that fits your needs best. The following table lists the precompiled version and explains how they can be compiled on Linux from source using `gcc` and `make`. If you want to compile on windows, you can use `mingw32-gcc` and type `mingw32-make` instead of make:

Windows	**Comments**	**How to compile this**
`iradina32.exe`	i386, 32bit, 4-MSB	`make -f makefile-32bit`
`iradina.exe`	x86-64, 64bit, 4-MSB	`make -f makefile`
`iradina32-6.exe`	i386, 32bit, 6-MSB	Open the file `fromcorteo.h`, locate the line `#include "indexvalues.h"` and replace it by `#include "indexvalues6bit.h"`. Then type: `make -f makefile-32bit`

`iradina-6.exe`	x86-64, 64bit, 6-MSB	Open the file `fromcorteo.h`, locate the line `#include "indexvalues.h"` and replace it by `#include "indexvalues6bit.h"`. Then type: `make -f makefile`
`iradina32_nw.exe`	i386, 32bit, 4-MSB, nanowire version	Open the file `iradina.h`, and uncomment the following line: `/* #define INCLUDE_SPECIAL_GEOMETRY */` Copy the files `geometry.c` and `geometry.h` from the geometries/nanowire directory to the source directory and type: `make -f makefile-32bit`
`iradina32_np.exe`	i386, 32bit, 4-MSB, nanoparticle version	Open the file `iradina.h`, and uncomment the following line: `/* #define INCLUDE_SPECIAL_GEOMETRY */` Copy the files `geometry.c` and `geometry.h` from the geometries/nanoparticle directory to the source directory and type: `make -f makefile-32bit`
`iradina_nw.exe`	x86-64, 64bit, 4-MSB, nanowire version	Open the file `iradina.h`, and uncomment the following line: `/* #define INCLUDE_SPECIAL_GEOMETRY */` Copy the files `geometry.c` and `geometry.h` from the geometries/nanowire directory to the source directory and type: `make -f makefile`
`iradina_np.exe`	x86-64, 64bit, 4-MSB, nanoparticle version	Open the file `iradina.h`, and uncomment the following line: `/* #define INCLUDE_SPECIAL_GEOMETRY */` Copy the files `geometry.c` and `geometry.h` from the geometries/nanoparticle directory to the source directory and type: `make -f makefile`

Note for windows users: the graphical user interface will run `iradina.exe`. If you want to use a different version, you should rename the executables accordingly. (The GUI automatically runs `iradina_np.exe` or `iradina_nw.exe` when the respective goemtries are selected in the GUI). If you are not sure, which version of *iradina* you have, then run *iradina* in the command line with the option `-i`. It will then print some information on the version.

2.2 Installation procedure part II: putting the files together

You should put the *iradina* executable in some local directory on your hard drive, for example `C:\iradina\` on windows. It is recommend not to use any blanks in the path. Next, you need to create a subdirectory named `data` in the same directory of the *iradina* executable. Now copy or move all the Corteo database files into that directory. Then you should be able to run *iradina* from a terminal. If you want to use the graphical user interface, refer to chapter 4.

2.3 Updating

In order to update from an older to the latest version, it should in most cases be sufficient simply to replace the executables files with the new versions.

3 How to use the program

Iradina is non-interactive. It can be run from a command line or it can be called by any other program or script to do some "background work". In order to run, *iradina* needs to be provided with four input files, which tell *iradina* what to do and which are described in section 3.1. Several command line options can be passed to *iradina* which are described in section 3.2. After *iradina* has finished the simulation it will store the results to a number of different files. These are described in 3.3.

3.1 Preparing the input files

Iradina reads the input from four input files: a general configuration file describing how the program is supposed to run (contains mostly simulation parameters etc.). The second file describes the material properties of all materials found in the target. The third file defines the structure of the simulation volume. The fourth file holds the the 3-dimensional description of the target (i.e. the shape of your nanosized object).
The first three files look similar to other configuration file types (and are mostly human-readable): empty lines or lines starting with a `#`–sign are ignored. All other lines can either denote the beginning of a new section by `[section name]` or can contain a parameter definition: `parname=value(s)`. Note that omitting any parameter might result in undefined behaviour or crashes of the program. Upper and lower case letters cannot be exchanged! Recommendation: Use existing example files and adapt them according to your needs.
Alternatively, it is possible to put all the information from the four input files into one combined input file, see details in section 3.1.5.

3.1.1 The configuration file

The contents of the configuration file will be explained using the following example and line-by-line explanations (do not include the line numbers in the actual file!):

```
10:   # Configuration file for iradina

20:   [IonBeam]
30:   ionZ=14
40:   ionM=28.0
50:   ionE0=50000
60:   ion_vx=1
70:   ion_vy=0
80:   ion_vz=0
90:   ion_distribution=0
100:  enter_y=20.6
110:  enter_z=20
120:  beam_spread=1.5
```

```
130:  [Simulation]
140:  max_no_ions=20000
145:  display_interval=100
150:  storage_interval=1000
155:  status_update_interval=1000
160:  store_transmitted_ions=1
162:  store_exiting_recoils=0
163:  store_exiting_limit=100
165:  store_energy_deposit=1
170:  store_ion_paths=0
180:  store_recoil_cascades=0
185:  store_path_limit=100
190:  simulation_type=0
200:  flight_length_type=0
210:  flight_length_constant=0.3
220:  detailed_sputtering=1
230:  min_energy=5
240:  seed1=39419293
250:  seed2=93145294
260:  OutputFileBaseName=output/test
265:  normalize_output=1
266:  transport_type=0
267:  multiple_collisions=0
268:  scattering_calculation=0
269:  do_not_store_damage=0
280:  store_range3d=0
285:  store_info_file=0

400:  [Target]
410:  straggling_model=3
420:  MaterialsFileName=Materials.in
430:  TargetstructureFileName=Structure.in
```

Lines **10**–**120** describe the parameters of the ion beam. `ionZ` is the number of protons of the ion, `ionM` the mass in atomic mass units, and `ionE0` the energy in units of eV.
Lines **60**–**80** desribe the initial unit velocity vector of the ion. All ions enter the target in the x-direction, but the ion beam can be tilted. Specifiying negative values for `ion_vx` makes no sense. If, for example, the ion beam should enter the target perpendicular to the $x = 0$ plane, set `ion_vx=1` and the other two to zero. If the ion beam should enter with an angle of 20° to this perpendicular direction and tilted towards the y-direction, then set `ion_vx` to $\cos(20) = 0.9397$ and `ion_vy` to $\sin(20) = 0.3420$. The velocity unit vector should always have the length 1, so: $\sqrt{v_x^2 + v_y^2 + v_z^2} = 1$ should be satisfied.
The "shape" of the ion beam is defined in lines **90**–**120**. There are four possibilities:

ion_distribution	means
0	Completely random distribution of entry positions of the ion on the $x = 0$ plane. The other three parameters are ignored (and can be omitted).
1	All ions enter at the center of the target (like in TRIM). The other three parameters are ignored (and can be omitted).
2	All ions enter at the defined (y, z) position in the $x = 0$ plane. The position must be specified by `enter_y` and `enter_z` in units of nm.
3	Like 2, but the ions are spread randomly around the entry point. `beam_spread` specifies how much the y and z position is spread around the defined entry point (in units of nm).

The simulation options are set in lines **130**–**260**. It starts with the number of ions that should be simulated impinging on the target. In lines **145** and **150** you can instruct the program to display the progress every n ions and to store intermediate results every n ions. So, if the simulation takes too much time, you can kill the running program and see at least the results up to the last storing point. However, storing the data to disk takes some time, so you shouldn't set this interval too short.
By setting `status_update_interval` you can tell *iradina* how often it should generate a status file when the `-g` option is supplied via the command line (see section 3.2).
If `store_transmitted_ions` is set to 1, the program creates a file where it stores the exit position, direction and energy for each ion that leaves the target. The exit positions, direction and energy of recoils leaving the simulation volume can be stored as well, which is activated by setting `store_exiting_recoils` to 1. Since this takes up memory and cpu time, the number of leaving recoils to be stored can be limited by `store_exiting_limit` in line **163**. Setting the parameter `store_energy_deposit` to 1 in line **165** instructs the program to create and store two arrays, that sum up electronic and nuclear energy loss in each cell. With the two parameters in lines **170** and **180** you can instruct *iradina* to store the exact flying paths of ions, and of recoils respectively, to disk. For each collision, an entry is stored to the files that designates the momentary position of the projectile and its energy. This is helpful to visualize the ion paths or recoil cascades, but keep in mind that this options generates enourmous amounts of data and slows down the program significantly. I only recommend using these options if the total number of ions is very small, for example < 100. You can use the `store_path_limit` setting in line **185** to tell *iradina* to stop storing ion paths and exact recoils after a number of ions. This way you optain a few paths for visualization, but most of the simulation will be still be fast.
The `simulation_type` in line **190** tells *iradina* how detailed the simulation should be. There are two choices at the moment:

simulation_type	means

0	Full simulation including detailed following of all recoils in collision cascades and deatiled calculation of damage. (Similar to TRIM's "Detailed calculation with full damage cascades").
3	Ions distribution only. Recoils are not followed, damage is not calculated. Faster. This does only work when `transport_type` is set 1. Otherweise *iradina* always considers recoils.
other numbers	Undefined behaviour of program.

Iradina uses the binary collision approximation (BCA) and the random phase approximation (RPA). Between two collisions, the ions travels on a straight free flight path and only loses energy to electrons. In line **200** the way how this free flight length is calculated, can be specified:

`flight_length_type`	means
0	The flight length is varied randomly according to a Poisson distribution with an average flight length according to the mean interatomic distance.
1	The flight length is constant and always corresponds to the mean interatomic distance.
2	The flight length is constant and always exactly the number specified by the `flight_length_constant` parameter (in units of nm).

Note that the simulation time scales approximately with inverse flight length. However, flight lengths far larger than the interatomic spacing will generate mostly inaccurate results. I recommend using option 0, but this is something you might think about. This is also an example of "you should know how a program works, to correctly interpret its results".
With the `detailed_sputtering` parameter in line **220** you can specify how accurately sputtering should be taken into account. If this parameter is 0, *iradina* will not calculate reasonable sputter yields but will be faster. If set to 1, *iradina* does calculate sputtering in more detail. However, in order to get accurate sputter yield, one should combine this parameter with `transport_ytpe=0`, otherwise sputter yields will be inaccurate. See section 3.4 for more details.
The `min_energy` parameter in line **230** decides when projectiles have to stop (unit: eV). Any projectile having less than this value is immediately stopped and followed any further. While 5 eV may be a useful value in many applications, this causes errors in sputtering yields. The minimum energy should be smaller than the smallest surface binding energy. However, the smaller this value, the londer the simulation takes.
In lines **240** and **250**, the initial seeds for the PRNG (pseudo random number generator) are defined. *Iradina* is a Monte Carlo code meaning it uses a lot of random numbers. Most computers

can of course not generate real random numbers, so the program uses the algorithm proposed by L'Ecuyer [L'E88] to generate pseudo random numbers. These seeds can be important: if you choose the same seeds for the same simulation, the program will always generate exactly the same results! If you want to run the program again and add up results from multiple runs to obtain better statistics, you must change these seeds!
The simulation will possibly generate a lot of output data and store these to a number of different files. With `OutputFileBaseName` (line **260**) you can decide where the data are stored. This parameter may contain a subdirectory to the current path from which you start the simulation. So for this example, *iradina* will store results to the subdirectory `output` and all output file names will begin with the word `test`.
By setting `normalize_output` to 1 in line **265**, you can instruct *iradina* to store output results in units of $(1/cm^3)$ per $(ions/cm^2)$. This allows the direct calculation of the concentration of implanted ions or some defect type for a specific fluence. Note, this is a little different from TRIM: *iradina* calculates $ions/cm^2$ by assuming that the plane in cm^2 is perpendicular to the ion beam. In TRIM, cm^2 corresponds to sample surface.
In line **266**, the transport algorithm is selected. There are two different versions implemented. Using "0" is much more accurate (especially important for sputter calculations!). Transport mode "1" is much faster (more similar to the computing flow in Corteo), yields similar ion and damage distributions but much worse sputter yields!
267: Let projectiles do more than one collision per flightpath (in annular cylinders). This number defines the number of extra collisions. So, zero means just the one "normal" collision. For details see [Eck91, p.92ff].
268: Setting `scattering_calculation` to 0 makes iradina use the fast database method for calculation of the scattering angle. "1" makes iradina use MAGIC. Note, using MAGIC only works when `transport_type` is set to "0".
269: When `do_not_store_damage` is set to 1, then *iradina* does not store the files that describe the various distribution of damage. The damage is still calculated, just not stored. This is useful, if you are just interested in sputtering for example, where damage must be correctly calculated but you do not need all the output to cram your precious disc space.
280: When `store_range_3d` is set to 1, then *iradina* creates a file that lists the final position (exact coordinates, not cells!) of each implanted ion (similar to the Range3D file created by TRIM). This file can be used to extract the distribution of implanted ions independently from the cell structures.
285: When `store_info_file` is set to 1, then *iradina* will create a file with additional information on the simulation each time results are stored. This information includes, for instance, the version of iradina, the time of the simulation, the number of completed ions, compiled options, etc.
The straggling model (line **410**) can be selected to be one of the following:

`straggling_model`	means
0	No straggling.
1	Bohr straggling [Boh48].
2	With Chu corrections [Chu76].
3	Chu and Yang's corrections [YOW91]

Use number 3, it is not perfect but the best implemented. All models need the same simulation time, as values are precalculated and tabulated anyway.
The last two parameter in lines **420** and **430** tell *iradina* where to find the definition of the materials properties and the target structure, see following sections.

3.1.2 The materials file

All materials that can be found in the target must be defined in a materials definition file. Again, the contents of this file will be explained using the following example and line-by-line explanations (do not include the line numbers in the actual file!):

```
10:  [GaAs]
20:  IsVacuum=0
30:  ElementCount=2
40:  Density=4.43e22
50:  ElementsZ=31,33
60:  ElementsM=69.72,74.92
70:  ElementsConc=0.5,0.5
80:  ElementsDispEnergy=20.0,25.0
90:  ElementsLattEnergy=3.0,3.0
100: ElementsSurfEnergy=2.0,1.2
105: IonSurfEnergy=2.0

110: [Vacuum]
120: IsVacuum=1
130: ElementCount=0
140: Density=1
150: ElementsZ=1
160: ElementsM=1
170: ElementsConc=1
180: ElementsDispEnergy=1
190: ElementsLattEnergy=1
200: ElementsSurfEnergy=1
```

Every material definition starts with the name of the material in square brackets (for example lines **10** and **110**). You can choose any name you like up to 20 characters, its just important that it is in square brackets. For every material, 9 parameters must be defined: If the material is vacuum (yes, vacuum is a kind of material in *iradina*) set `IsVacuum` to 1. For all other materials set this to zero. All of the following parameters are ignored for vacuum. Next, you need to specify the number of elements in this material by `ElementCount` (it must be specified before the other parameters, otherwise *iradina* might crash). The density of the material must be specified in atoms per cm^3. `ElementsZ` must be a comma-separated list of the proton numbers

of all the elements in the current material. `ElementsM` contains the masses in atomic mass units. The concentrations of the various elements in this material is defined in `ElementsConc`. The numbers do not need to add up to 1, *iradina* normalizes these values, so its easier to define odd stoichiometries like X_7Y_{13}. The next three parameters contain comma-separated lists of various energy barriers for each element, namely the displacement energy, the lattice binding energy and the surface binding energy. You can extract these values from TRIM but you should know that these may be different for compounds. The last parameter in line **105** defines the surface binding energy of the ion. The ions gains this, when entering the material and looses it when exiting. Under oblique angles, the ion is refracted. This parameter may be omitted as it is usually not of great importance.
Iradina internally assigns consecutive numbers to each material found in the definition file, starting with zero. Note: if different isotopes of one element are important to you, you can specify them as different elements. However, *iradina* uses the mass of the most abundant isotope for the calculation of the scattering event.
Important note: it is strongly recommended NOT to create one file with all materials that you know for all of your simulations! You should just include the materials you really need for the current simulation, because *iradina* will create 2.6 MByte scattering matrices for every possible combination of two elements in the target! So the memory usage increases with the square of the number of different elements! If your materials contain 92 different elements, *iradina* needs 22 GByte of memory in the 4-MSB version or 352 GByte in the 6-MSB version. . . .

3.1.3 The structure file

The simulation volume is always a rectangular box (or a "cuboid"). The simulation volume is divided into a possibly large number of small cells using a 3-dimensional rectangular grid. The cells are also cuboid-shaped, but can have different sizes in each of the three directions. Each cell may contain any of the materials defined in the materials file. Every cell contains several counters, to sum up implanted ions and the various kinds of damage in each cell. Again, the contents of the structure file will be explained using the following example and line-by-line explanations (do not include the line numbers in the actual file!):

```
10:  # Structure definition file for iradina
20:  [Target]
30:  cell_count_x=50
40:  cell_count_y=50
50:  cell_count_z=4
60:  cell_size_x=1
70:  cell_size_y=1
80:  cell_size_z=10
90:  periodic_boundary_x=0
100: periodic_boundary_y=0
110: periodic_boundary_z=0
120: CompositionFileType=0
130: CompositionFileName=testwire.conc.in
140: UseDensityMultiplicator=0
```

```
150: DensityMultiplicatorFileName=none
160: special_geometry=0
```

Lines **30** to **50** define how many cells there are in each directions. These must be integer numbers >0. Note that ions enter the target through the $x = 0$ plane. The size of the cells in each direction is defined as shown in lines **60** to **80** (in units of nm). The cell dimensions should be larger than monolayer distances. Otherwise, this could cause trouble in sputter yield calculations. Cell sizes below 1 nm are definitely not recommended! In each of the three directions you can activate periodic boundary conditions (PBC) by settings the relavant parameter to 1 (lines **90-110**). This may allow you to significantly decrease the necessary simulation volume. If PBC is activated for one direction, then all ions that leave the simulation volume in this direction are transferred to the opposite side of the simulation volume. Otherwise, ions leaving the simulation volume are not followed any further.
There are two different formats for composition files, called "0" and "1", details in section 3.1.4. The format is selected by the `CompositionFileType` parameter. The name of the composition file is designated by `CompositionFileName`. If various parts of your target consist of the same material but have different densities you do not need to define different materials. Instead you can activate the `UseDensityMultiplicator` option by setting it to 1. Then you may specify an additional composition file (name designated by `DensityMultiplicatorFileName`), which contains a multiplicator for the density of each cell in the target.
If *iradina* has been compiled with a non-standard target geometry, this special geometry can be switched off by setting `special_geometry` to 0 (line **160**). For details about non-standard geometries, see section C.

3.1.4 The composition file

The composition file describes the 3-dimensional composition of your target, so it defines the shape of the nanoobject that you want to irradiate. For each cell, the composition file must contain one number that selects the material in this cell. The materials must have been defined in the materials file. Be aware that the numbering of materials starts with zero.
If `CompositionFileType=0` is selected, the composition file must contain four columns. The first three columns are the x–, y–, and z–coordinate of a cell and the fourth column is the material number for this cell. The allowed range for the x–, y–, and z–values depends on the structure definition file. The range extends from 0 to `cell_count`-1 in each direction. The order of the data lines in the composition file can be arbitrary. The composition file must not contain anything else. An example might look like this (with different materials 0, 1 and 2):

```
0  0  0  2
0  0  1  2
0  0  2  2
0  0  3  2
0  1  0  1
0  1  1  0
0  1  2  0
0  1  3  0
```

```
1   0   0   2
1   0   1   2
...
```

Note, this example composition requires three different materials to be defined (numbers 0, 1 and 2). The example material definition from section 3.1.2 will not work with this composition, because it defines only two different materials.
The second available format for composition files (selected by `CompositionFileType=1`) needs less disk space but is more difficult to read: it simply omits the first three columns and assumes that the material values are given in the order as illustrated in the above example. There is no binary format at the moment (which would save even more space), just text files.

3.1.5 Combined input file

The combined file format is intended for simpler use in combination with the user interface. Basically, the combined file is like a config file with the other three input files appended to it with some separation markers in between. *Iradina* looks at the first line in the file in order to determine whether a combined input file is provided. The combined file with its markers should looks like this:

```
#<<<BEGIN CONFIGFILE
  ... contents of config file ...
#<<<BEGIN STRUCTUREFILE
   ... contents of the structure file ...
#<<<BEGIN MATFILE
   ... contents of the materials definition file ...
#<<<BEGIN COMPFILE
   ... contents of the composition file ...
```

Note that the separation markers look like comments but are not treated as such.

3.2 Running the program

Iradina can be run from a command line or it can be invoked by another process or you might just start it by clicking its icon in the file browser of your choice. I recommend using a command line first, so that you can check the output. If no parameters are provided, *iradina* assumes that there is a configuration file named `Config.in` in the current directory and loads the configuration from this file. However, you can provide various command line arguments (they are all optional):

`-h`	Prints the help (basically like this table).
`-l`	Display the license file.
`-c FILENAME`	Instructs *iradina* to use FILENAME as the configuration file instead of the default `Config.in`
`-p NUMBER`	Specify how much info to print to console. -2 means very little, 2 means a lot, with various possibilities in between. Default is 0.
`-n NUMBER`	Specifies how many ions are simulated (overrides the setting specified in the config file).

`-E NUMBER`	Sets the energy of the ions. This option overrides the setting from the config file.
`-w`	Wait for return key before exiting (useful if started from another program and you still want to see output).
`-m`	Do not simulate anything, only estimate memory usage (roughly).
`-d`	Print details for memory usage (only useful when `-m` option also supplied).
`-g STRING`	Generate and update status file while running. See details below.
`-i`	Print info on this version of iradina.

The program mostly does not check whether the input parameters make any sense. In case some parameters are missing or completely out of range, the program will most likely crash or create strange results.
The `-g` option is useful when letting *iradina* do background work for other programs. It instructs *iradina* to output its current status to a file, which can be monitored by other programs (for example by a user interface). If it is set, every 200 ions *iradina* writes exactly four lines to the file `ir_state.dat` (the output frequency of 200 can be changed by the `status_update_interval` parameter in the config file). The first line of the status file contains the word `iradina` and the version of *iradina*. The second line contains the string submitted on the command line after the `-g` option. The third line contains a status word. The fourth line contains the number of ions that have been simulated. The status words and their meanings are:

`init0`	*iradina* has started.
`init1`	The config files have been read.
`init2`	Everything is initialized and ready.
`sim`	Simulation is running.
`simend`	Simulation has finished.
`end`	Simulation results have been stored and *iradina* will terminate immediately.

3.3 The output files

Iradina produces a lot of output files, which are listed in the table below. All file names begin with the name specified in the configuration file. A descriptor is added to that name, which tells you what to find in that file. The descriptor can consist of one or more parts separated by a dot. Some descriptors end with an element descriptor (`<ED>`) that specifies an element of a specific material defined in the materials file (section 3.1.2) by its proton number, mass, material number and element index within that material. These files are created for each element from each material in the target. If the `<ED>` is `sum` this file contains the sum of all corresponding element-specific files. Output files can have different formats. Many have the same format as the composition file (see section 3.1.4), indicated by "comp.":

Descriptor	Format	Contents

.ions.total	comp.	The number of implanted ions in each cell (interstitials + replacing ions).
.ions.replacements	comp.	If the ion is of the same element as a target element, it may transfer most of its energy to such element in a collision, remain on that lattice site and replace the recoiling atom. This file stores the number of replacing ions in each cell.
.ions.range3d	specific	The final position of each implanted ion (coordiantes in nm)
.int.<ED>	comp.	The number of recoil interstitials of the type specified by <ED> in each cell.
.vac.<ED>	comp.	The number of vacancies of the type <ED> in each cell.
.disp.<ED>	comp.	The number of displacements of atoms of type <ED> in each cell.
.repl.<ED>	comp.	The number of replacements that happened when a projectile of type <ED> replaced a recoil of type <ED> in each cell.
.leaving.<ED>	comp.	For each cell: the number of recoils of type <ED> that originated from this cell and left the simulation volume.
.leaving_directions.<ED>	specific	This file lists how many recoils of type <ED> left the simulation volume separated by the 6 possible leaving directions. Note: the file leaving_directions.sum does not include leaving ions.
.transmitted.ions	specific	Contains one line for each ion leaving the simulation volume. There are seven columns: The first three are the coordinates of the exit position of the ion (more specifically of the first collision that would take place outside the simulation volume if there were material there). The next three numbers denote the velocity unit vector (from which you can obtain the direction of the ion after leaving the target) and the last columns contains the remaining energy.
.leaving_recoils.<ED>	specific	Contains one line for each recoil leaving the simulation volume (up to store_exiting_limit). The format is the same as for transmitted ions.

`.cascades`	specific	Can be used to visualize recoil cascades. Contains one line for each collision of a recoil atom. Each line contains four numbers, the first three being the coordinates of the collision (in nm) and the last number denoting the momentary energy of the recoil in eV.
`.ionpaths`	specific	Stores exact path of each ion from collision to collision. Format like the `.cascades`–file.
`.energy.electronic`	comp.	The amount of energy deposited in each cell by electronic energy loss (a.k.a. "ionization" or "inelastic energy loss"). Note: this includes electronic energy loss of recoils, not only of the ions itself. This file is only created if `store_energy_deposit=1` is set in the config file.
`.energy.phonons`	comp.	The amount of energy deposited in each cell into the phononic system. (The remaining energy, when projectiles "get stuck" as replacements or interstitials). This is a little less than the primary nuclear energy loss of the ion, because recoils lose some of their energy to electrons. This file is only created if `store_energy_deposit=1` is set in the config file.
`.information`	specific	Additional information on the simulation (*iradina* version, time, ...).

Note that the output by default has no unit. If you simulate 1000 ions, then the counter of each cell will tell you how many of the initial 1000 ions will be end up in that cell. If you want *iradina* to store results in the much more convenient unit $(1/\mathrm{cm}^3)/(\mathrm{ions}/\mathrm{cm}^2)$, then set the `normalize_output` parameter to 1 in the config file.

3.4 Sputtering

Calculating sputter yields from MC simulations is not trivial. Read for example [MP10]. For reasonable results, its recommended to do the following:

- Use the following settings:
  `transport_type=0`
  `detailed_sputtering=1`
  `multiple_collisions=2`
- Set lattice binding energies to zero.
- Select reasonable surface binding energies.
- If your target has surfaces, which are not perpendicular to one of the grid axes: use small cells (like 1 nm^3).

- Important note: make sure, that all relevant surfaces from which sputtering can occur are not directly adjacent to the edge of the simulation volume. In such cases, a vacuum layer should be used for separation. This is important because the surface calculations are only activated when a projectile crosses from a material cell to a vacuum cell or vice versa. If you use *iradina* to calculate bulk sputter yields, always have a layer of at least one cell (or more than one nm) with vacuum at the entry side of the target.

The sputter yields can be extracted from the `leaving`-output files (see section 3.3).
In any case, accurate sputter yields are not guaranteed.

4 The graphical user interface

Iradina itself has no user interface (UI) at all. An independent UI for Windows exists, called *iraUI*. "Independent" means that the user interface is in no way technically connected to *iradina* and can in principle be used without it or for other purposes (though that wouldn't make much sense). *IraUI* simply helps the user in creating all the necessary input files for *iradina* and in analyzing the output files. This way, simulations can be started with a few clicks, and without plunging through all the config files. Nevertheless, I recommend reading this manual also when using the UI, because it helps in understanding what all the parameters mean.
The following short manual corresponds to version 1.0 of *iraUI*.

4.1 Installation and first start

IraUI requires the dot-net framework 3.5 SP1 or higher to be installed. If this is already installed, it should be possible to simply download the archive with the executable file, extract all the files to a directory of your liking and start the program by running `iraUI.exe`. Otherwise, you can download the *iraUI*-installer, which should automatically download and install the dot-net framework and then *iraUI*.
IraUI must be told where it can find the *iradina* executable file. After the first start, click on `Simulations->General settings` and select the *iradina* working directory, where the *iradina* executables must reside.

4.2 Using iraUI

The user interface should be mostly self-explanatory, but here is a short guide line how to work with it:

- Start a new simulation project. Such a "project" contains all revelant information for the simulation.
- Define the ion beam parameters by clicking on `Experiment->Ion Beam Parameters` or on the respective symbol. If you want to create implantation profiles with different energies, you can set the different ion energies here. In that case *iraUI* will run multiple simulations automatically.
- Define the simulation parameters next by clicking on `Experiment->Simulation Parameters`.

- Then you should define the materials, which are present in the target. *IraUI* can use a database (DB) of predefined materials. In the `General settings` dialog you can specify the location of this DB. The DB file has the same format, as the materials definition file. In the `Materials`-dialog you can select materials from the DB and also add materials to the DB.

- After definition of the materials, you can define the target structure. First, define the number and size of cells in each direction (note that the ion beam travels in positive x-direction!). Then click on `Accept new structure`.

- Click on `Define target...`

- Here you can define your target in two ways: you can define logical objects (like blocks, cylinders or spheres), or you can edit the grid manually. After you have defined the objects, click on `Create target grid structure from objects` in order to scan the objects and create a grid out of them, which can be used by *iradina*. If two or more objects overlap at one grid point, the object further down the list has higher priority. Note that it is often useful to define one big block of vacuum, which fills the complete simulation volume. This ensures that all cells are defined. Then add your objects of choice, which will "overwrite" the vacuum at their respective positions.
  The grid can be visualized on the third tab of the dialog. Yoy may also edit the grid manually on the second tab of the dialog

- When everything is defined, save your project. When saving for the first time, *iraUI* will ask your where the output files of *iradina* should be stored.

- Use the `Run simulation` dialog to start the simulation. *Iradina* will be run in a console window and you can follow its output or watch the progress in *iraUI*.

- When the simulation has finished, select `Tools` and `Load results`. Now click on `Update` and the program will search for result files in the output directory of the current project. A list of the available results will be displayed. Double click on a field to load the results. A new window showing the resulting data will pop up. Right click on the window for further options.
  Note, the data are usually scalar fields with three spatial coordinates. The display shows the scalar value for a 2d plane from the 3d data. You can select the plane that is shown by right-clicking on the window and selecting `Properties`. In the property window you can also adjust the color scale and see more detailed information on the currently selected plot.

In the current version, *iraUI* cannot be used to analyze sputter yields. These can be read manually from the output files.
Note: *iraUI* always completely rewrites project files upon saving them. If you have manually created a config file and open it with *iraUI*, all your comments, structure and special options might be lost. Thus, it is recommended to make a backup copy of manually created input files before opening them with *iraUI*.

5 Costs, license and source code

Iradina is released under the GNU general public license (GPL), version 3 [GPL07], so you can use it for free. You can further obtain the source code and change it to your liking as long as you respect the GPL. The source code can be downloaded from `http://www.iradina.de`. If you use results obtained from *iradina* simulations in a publication, I kindly ask you to cite reference [BR11].
The independent user interface is currently freeware but not open source.

A How does the program work?

A short description of how the program works is given in reference [BR11]. However, *iradina* has been improved since [BR11] has been published and current versions work a little different.

B Optimization

Here are some hints how to reduce memory usage and how to increase simulation speed. *Iradina* is basically optimized for speed at the cost of memory.
Memory optimization:

1. Use as few chemical elements as possible in your target. Every possible combination of two elements requires additional 2.6 MBytes of memory, so this memory usage scales with n^2, with n being number of elements in the target.

2. Use as few cells as possible. Example: 100 cells in each direction are 10^6 cells in total. Each cell has about 6 counters for each element, each counter occupying 4 bytes of memory. So at 5 different elements in the target, 10^6 cells result in ≈120 MByte. These are also stored to disk at every storage step. If the composition file type is set to 0 you have about 10 bytes per cell, leading to ≈300 MByte output to disk.
   Try to use a small number of cells: For example: if your target is a nanowire and you need the cross-sectional distribution of ions in the nanowire, it is sufficient to use only one cell in the axial NW direction and apply PBC in this direction! A grid of 50×50 cells in the cross section might provide enough accuracy. With "only" 2500 cells, memory usage for the cells is 400 times less than for 10^6 cells so just 300 kByte instead of 120 Mbyte for the above example.

Speed optimization:

1. Optimize memory usage. Compared to the CPU cache, the system memory is slow. If a larger fraction of the program data fit into the cache, it will be faster. Do not attempt to use more memory than you have available in RAM (meaning to use the swap memory on disk, if your physical memory is too small). *Iradina* does a lot of random access to the memory and the disk is far too slow for that. Also, this stresses your disk and shortens its lifetime.

2. Use `transport_type=1` if the accuracy is sufficient.

3. Do not use `detailed_sputtering`, if not needed.
4. Do not store the exact ion paths and recoil cascades (only for a very small number of ions).
5. Use a large storage interval (storing to disk takes time).
6. Select a large free flight path (but be careful - this makes the results inaccurate!).

C Special Geometries

Special geometries can be used for example to represent nanowires as cylinders in the code and not as a number of cells. This allows accurate definition of curved surfaces, which improves sputter yield results. Support for special geometries must be compiled into the program by putting the line
`#define INCLUDE_SPECIAL_GEOMETRY`
in the file `iradina.h`. Which geometry is actually compiled into the program depends on the files `geometry.c` and `geometry.h` in the source directory. Geometries are at the moment available for nanowires (cylinders) and for nanoparticles (spheres). The corresponding geometry files must be copied from the `geometry`-subdirectory to the source base directory prior to compilation.
For Windows, different versions exist with support for nanowires and nanoparticles precompiled into the programs. The user interface uses the nanowire version `iradina_nw.exe`, if irradiation of nanowires is simulated. Please note that nanowire irradiation happens always from the nanowire side not from top.
For the special geometries "nanowire" and "nanoparticle", the target is described analytically and not by the rectangular grid. Nevertheless, the grid is used for counting implanted ions and damage and for all the output files. When using the nanowire geoemtry, *iradina* needs an additional input file called `nw_structure.in`, which must reside in the current directory. The file must contain one line specifying the diamater of the nanowire, i.e. for 50 nm diameter:

```
NW_diameter=50
```

For nanoparticles, *iradina* requires an input file called `np_structure.in`, which must contain a line like this: `NP_diameter=50`
If you want to create your own special geometry (like some other target shape, which can be described analytically), you have to create your own `geometry.c` and `geometry.h`. These files must contain certain functions. It is recommended to use a copy of the existing geomerty files (for example from the nanowire geomerty) and adjust them to your liking.

D Problems and Troubleshooting

- **Problem** on Windows: the program cannot be run from a network drive. When starting the program from the windows console or from within the GUI, windows claims, that the "command or program" cannot be found albeit being in the correct directory.
  Solution: work on a local drive.

- **Problem**: *Iradina* seems to read all the impact parameters wrongly and strange things happen.
  Solution: Make sure that your system decimal separator is set to a full stop.

- **Problem**: *iradina* ends immediately after started with the following error message: "Header element X does not correspond to current parameters in corteo.mat. Error load corteo scattering matrix. Initialization error: -4014. Aborting."
  Solution: This happens when you have a 6-MSB corteo scattering matrix but are using a 4-MSB *iradina* version or vice versa. Read section 2.1 of this manual and chose or compile the correct version of *iradina*.

References

[Boh48] Niels Bohr. The penetration of atomic particles through matter. *K. Dan. Vidensk. Selsk. Mat.-Fys. Medd.*, 18(8), 1948.

[Bor12] Christian Borschel. *Ion-Solid Interaction in Semiconductor Nanowires*. Dissertation, University of Jena, available online from the following url: `http://www.db-thueringen.de/servlets/DocumentServlet?id=20026`, 2012.

[BR11] C. Borschel and C. Ronning. Ion Beam Irradiation of Nanostructures - A New 3D Monte Carlo Simulation Code. *Nuclear Instruments and Methods in Physics Research B*, 269:2133, 2011.

[Chu76] W. K. Chu. Calculation of energy straggling for protons and helium ions. *Physical Review A*, 13(6):2057–2060, 1976.

[Eck91] Wolfgang Eckstein. *Computer Simulation of Ion-Solid Interactions*. Springer Series in Materials Science. Springer, Berlin, Heidelberg, New York, 1991.

[GPL07] GNU general public license, 2007.

[L'E88] Pierre L'Ecuyer. Efficient and portable combined random number generators. *Communications of the ACM*, 31:742–751, 1988.

[Möl13] Wolfhard Möller. TRI3DYN Collisional computer simulation of the dynamic evolution of 3-dimensional nanostructures under ion irradiation. *Nuclear Instruments and Methods in Physics Research B*, 322:23–33, 2013.

[MP10] W. Möller and M. Posselt. TRIDYN_FZR User Manual. Technical Report FZR-317, Helmholtz-Zentrum Dresden-Rossendorf, 2010.

[Sch07] F. Schiettekatte. *Some notes on Corteo*, available from `www.lps.umontreal.ca/~schiette/index.php?n=Recherche.Corteo`, 2007.

[Sch08] F. Schiettekatte. Fast Monte Carlo for ion beam analysis simulations. *Nuclear Instruments and Methods in Physics Research B*, 266:1880–1885, 2008.

[YOW91] Q. Yang, D. J. O'Connor, and Z. Wang. Empirical formulae for energy loss straggling of ions in matter. *Nuclear Instruments and Methods in Physics Research B*, 61:149–155, 1991.

[ZBL85] J. F. Ziegler, J. P. Biersack, and U. Littmark. *Stopping and Range of Ions in Solids*. Pergamon, New York, 1985.