IDEAL Documentation

Release IDEAL 1.0, March 23, 2021

EBG MedAustron GmbH
ACMIT Gmbh
Medical University Vienna

Mar 23, 2021

CONTENTS

Independent **D**os**E** c**A**lculation for **L**ight ion beam therapy using Geant4/GATE. Also known as "pyidc" (python module for independent dose calculation).

This is the 1.0 release of the IDEAL project. This was developed between February 2018 and March 2021 at EBG MedAustron GmbH, in collaboration with ACMIT Gmbh and the Medical University of Vienna. This code has been tested to work correctly for treatment plans with the fixed beam lines of the MedAustron clinic. At the time of this release it has not yet been tested at any other clinic, but we hope that this software will be useful at other clinics as well.

If you wish to install this 1.0 release, please clone this code directly from GitHub on a shared disk of the submit node of your HTCondor cluster and follow the installation instructions.

In order to facilitate installation with `pip`, the code will be reorganized a bit. In particular the `ideal` python module will effectively be renamed `pyidc`, in order to avoid a name collision with another python module. This reorganized version of IDEAL will get release tag `1.1`.

This project will not work until it has been properly configured, regardless of whether you install this project by cloning or with `pip` (v1.1 and later). Please take your time and read the *Installation* and *Commissioning* sections of the documentation carefully.

CHAPTER ONE

CONTENTS

1.1 Overview

1.1.1 Synopsis

IDEAL is a set python modules and scripts that can be used to compute the dose distribution (in the patient or in a phantom) for a given treatment plan. The dose calculations are based on Geant4/Gate (specifically Gate-RTion).

1.1.2 Disclaimers

- IDEAL is **NOT** a medically certified product.
- This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License [GPLv3] for more details.
- IDEAL has been developed and tested for clinical use at EBG MedAustron GmbH, using the specific combination of beamlines, treatment planning software and computing infrastructure in that clinic. We wrote this software with the hope that it will be useful in other clinics as well, but we cannot guarantee that it will work correctly right out of the box.

1.1.3 Intended Use

Clinical use "IDEAL is a software system designed for dose calculation of treatment plans produced by a Treatment Planning System (TPS) for scanned ion beam delivery. IDEAL provides an independent estimate of the dose distribution expected for the proposed treatment plan. The evaluation and review of the dose distributions is done by the qualified trained radiation therapy personnel. The intended use is as a quality assurance tool only and not as a treatment planning device."

Research use IDEAL is intended as a research tool to study dose distributions in PBS ion beam therapy. Research topics could include the properties of passive elements, biological effect models, patient motion and much more.

1.1.4 Intended Users

IDEAL was written to be used by medical physicists and/or IT professionals in scanned ion beam therapy clinics. The user roles are:

clinical user Medical physicist in charge of patient specific quality assurance (PSQA). This user uses IDEAL to obtain an independent dose calculation, as described in *Intended Use*.

commissioning user Medical physicist in charge of validating and commissioning IDEAL. This user will provide all site specific data, such as beam models, geometrical details of nozzles and passive elements, Hounsfield lookup tables for all CT protocols, etc.

admin user Medical physicist or IT professional who installs and maintains the IDEAL software on a clinical site. The admin user makes sure that the software is correctly installed, assists the commissioning user with configuring the IDEAL with the correct and up to date commissioning data and manages the queue of calculation jobs on the cluster.

research user Medical physicist who uses IDEAL for "research" purposes, i.e. any purpose that is not the independent dose calculation of a clinical treatment plan.

The software currently allows to assign these roles to users, however in the 1.0 release these roles are not enforced. This is a TODO item.

1.1.5 User interfaces

IDEAL can be used through several user interfaces:

1. Command line interface: the *clidc.py* script ("command line independent dose calculation") can be used to trigger a dose calculation based on a given DICOM treatment plan file that was exported (together with the corresponding CT, structure set and TPS dose calculations) from the TPS to directory on a shared file system (IDEAL typically does not run on the same hardware as the TPS). The output (dose distributions for each beam in the plan) is saved in DICOM as well. This interface is useful for commissioning and research.
2. Via a custom PyQt GUI: only to be used for research.
3. Research scripting: IDEAL has been written in a modular way, such that its functionality is available to research users via python modules.

1.2 Installation

1.2.1 System requirements

Hardware

IDEAL will run CPU-intensive simulations. [GateRTion] is single threaded, so in order to get results within a clinically reasonable time frame, many instances of GateRTion need to run in parallel on sufficiently fast CPUs. While this can in principle be achieved with a single machine, in the following we assume a typical setup with a small/medium size cluster.

Submission Node

- At least 4 cores with a clock speed greater than 2GHz.
- At least 16 GiB of RAM.
- Local disk space for the operating system and HT Condor, 100 GiB should be sufficient.
- Shared disk: see below.

Calculation Nodes

- In total at least 40 cores (preferably 100-200) with a clock speed greater than 2GHz.
- At least 8 GiB of RAM per core[1].
- Local disk space for the operating system and HT Condor, 100 GiB should be sufficient.

Shared disk (internal)

- At least half a terabyte.
- Storage of all software, configuration and simulation data.
- Accessible by the submission and calculation nodes.
- Internal cluster network and storage hardware should provide at least O(10Gbit/second) read and write speed.
- Should support high rewrite rate. During a simulation, temporary results up are saved for all cores typically every two minutes, O(1Gib/core).

Network access to/from submission node

The submission node should be accessible by the user, or be connected with an external server that functions as the user interface. To this end, the submission node should be connected to an reasonably fast internal network that allows access to a shared directory system (typically CIFS) or HTTPS connections with at least one other server. Recommended data upload and download speed is 1Gbit/second or faster.

Mounting Windows file shares

A typical clinical computing environment is dominated by MS Windows devices, and if the environment includes a Windows File Share (CIFS) then it can be convenient to mount this from the submit node of the IDEAL cluster. This can then be used for DICOM input to and output from IDEAL.

Ask your local MS Windows system administrator which subfolder(s) on the file share you can use for IDEAL input/output, and with which user credentials. Some administrators prefer to use personal user accounts for everything (so they can track who did what, in case something went wrong), others prefer to define "service user" accounts that can be used by several users for a particular (limited) purpose. Create a new folder on the submit node (`/var/data/IDEAL/io` in the example below) and save the user credentials in a text file `secrets` with `-r--------` file permissions (readable only by you).

Then run the following script (or edit `/etc/fstab`, if you are comfortable doing that) to create "read only" mount point for reading input and a "read and write" mount point for writing output. The mount points *can* point to the same folder.

```
#!/bin/bash
set -x
set -e

# you need to define the names and paths here
ideal_remote="//servername.domainname/path/to/IDEAL/folder"
ideal_rw="/var/data/IDEAL/io/IDEAL_rw"
ideal_ro="/var/data/IDEAL/io/IDEAL_ro"
creds="/var/data/IDEAL/io/secrets.txt"

for d in "$ideal_rw" "$ideal_ro"; do
```

[1] The RAM requirement is driven mostly by Carbon ion simulation memory needs. For proton plans the RAM requirements are more relaxed.

```
        if [ ! -d "$d" ] ; then
                mkdir -p "$d"
        fi
done

# you need to provide the actual uid and gid here
creds_uid_gid="credentials=$creds,uid=montecarlo,gid=montecarlo"

rw_opts="-o rw,file_mode=0660,dir_mode=0770,$creds_uid_gid"
ro_opts="-o ro,file_mode=0440,dir_mode=0550,$creds_uid_gid"
sudo mount.cifs "$ideal_remote" "$ideal_rw" $rw_opts
sudo mount.cifs "$ideal_remote" "$ideal_ro" $ro_opts
```

Software

Operating System

For all cluster nodes: Linux. Any major modern operating system (e.g. [Ubuntu] 18.04 or later) should work.

Python

- Python [Python3] version 3.6 or later should be installed on all nodes.
- Submission node: *virtualenv* and *pip* are used to install modules that are not part of the standard library.
- In case the IDEAL cluster is not directly connected to the internet, the intranet should contain a repository that is accessible by the submission node and provides up to date release of the following python modules (versions are *minimum* versions):

Module	Version	Module	Version	Module	Version
alabaster	0.7.12	Babel	2.8.0	backcall	0.1.0
certifi	2020.6.20	chardet	3.0.4	cycler	0.10.0
decorator	4.4.2	docutils	0.16	filelock	3.0.12
htcondor	8.9.8	idna	2.10	imagesize	1.2.0
ipython	7.13.0	ipython-genutils	0.2.0	itk	5.0.1
itk-core	5.0.1	itk-filtering	5.0.1	itk-io	5.0.1
itk-numerics	5.0.1	itk-registration	5.0.1	itk-segmentation	5.0.1
jedi	0.17.0	Jinja2	2.11.2	kiwisolver	1.1.0
MarkupSafe	1.1.1	matplotlib	3.2.1	numpy	1.18.2
packaging	20.4	parso	0.7.0	pexpect	4.8.0
pickleshare	0.7.5	pip	20.2.4	pkg-resources	0.0.0
prompt-toolkit	3.0.5	ptyprocess	0.6.0	pydicom	1.4.2
Pygments	2.6.1	pyparsing	2.4.6	python-daemon	2.2.4
python-dateutil	2.8.1	pytz	2020.1	requests	2.24.0
scipy	1.4.1	setuptools	46.0.0	six	1.14.0
snowballstemmer	2.0.0	Sphinx	3.2.1	sphinxcontrib-applehelp	1.0.2
sphinxcontrib-devhelp	1.0.2	sphinxcontrib-htmlhelp	1.0.3	sphinxcontrib-jsmath	1.0.1
sphinxcontrib-qthelp	1.0.3	sphinxcontrib-serializinghtml	1.1.4	traitlets	4.3.3
urllib3	1.25.10	wcwidth	0.1.9	wheel	0.34.2

HTCondor

IDEAL relies on the [HTCondor] cluster management system for running many simulations in parallel[2]. Any recent release (e.g. 8.6.8) should work well. All major Linux distributions provide HTCondor as a standard package. The full documentation of HTCondor can be found on the HTCondor web page. Below some of the specific details for configuring and running HTCondor are described. These are meant as guidance, the optimal configuration may depend on the details of available cluster.

Configuration

Each (submit or calculation) node has HTCondor configuration files stored under `/etc/condor/`. The `/etc/condor/condor_config` file contains the default settings of a subset of all configurable options. This file should not be edited, since any edits may be overwritten by OS updates. The settings below may be added either to the `/etc/condor/condor_config.local` file, or in a series of files `/etc/condor/config.d/NNN_XXXXXX`, where `NNN` are numbers (to define the order) and `XXXXXX` are keywords that help you remember what kind of settings are defined in them.

The options described below are important for running IDEAL. The values of the settings are sometimes used in the definition of other settings, so be careful with the order in which you add them.

The configuration can be identical for all nodes, except for the daemon settings.

Condor host The submit node should be "condor host", which is declared by setting `CONDOR_HOST` to the IP address of the submit node:

```
CONDOR_HOST = w.x.y.z
```

Enable communication with other nodes The simplest way to configure this is to just enable communication ("allow write") for each node with all nodes in the cluster, including the node itself. The `ALLOW_WRITE` is a comma-separated list of all hostnames and IP addresses. For ease of reading, the nodes can be added one by one, like this:

```
ALLOW_WRITE = $(FULL_HOSTNAME), $(IP_ADDRESS), 127.0.0.1, 127.0.1.1
ALLOW_WRITE = $(ALLOW_HOST), submit_node_hostname, w.x.y.z
ALLOW_WRITE = $(ALLOW_HOST), calc_node_hostname, w.x.y.z
ALLOW_WRITE = $(ALLOW_HOST), calc_node_hostname, w.x.y.z
ALLOW_WRITE = $(ALLOW_HOST), calc_node_hostname, w.x.y.z
```

Which daemons on which nodes This is the only item that requires different configuration for submit and calculation nodes.

For the *submit* node:

```
DAEMON_LIST  = MASTER, COLLECTOR, NEGOTIATOR, SCHEDD, GANGLIAD
```

For the *calculation* nodes:

```
ALLOW_NEGOTIATOR = $(CONDOR_HOST) $(IP_ADDRESS) 127.*
DAEMON_LIST  = MASTER, STARTD, SCHEDD
```

Network and filesystem Make sure to configure the correct ethernet port name and the full host name of the submit node

```
BIND_ALL_INTERFACES = True
NETWORK_INTERFACE = ethernet_port_name
CUSTOM_FILE_DOMAIN = submit_node_full_hostname
FILESYSTEM_DOMAIN = $(CUSTOM_FILE_DOMAIN)
UID_DOMAIN = $(CUSTOM_FILE_DOMAIN)
```

[2] In future releases, other cluster management systems such as SLURM and OpenPBS may be supported as well.

Resource limits HTCondor should try to use all CPU power, but refrain from starting jobs if the disk space, RAM or swap exceed some safe thresholds

```
SLOT_TYPE_1 = cpus=100%,disk=90%,ram=90%,swap=10%
NUM_SLOTS_TYPE_1 = 1
SLOT_TYPE_1_PARTITIONABLE = True
```

Resource guards Define what to do when some already running job exceeds its resource limits

```
MachineMemoryString = "$(Memory)"
SUBMIT_EXPRS = $(SUBMIT_EXPRS)  MachineMemoryString
MachineDiskString = "$(Disk)"
SUBMIT_EXPRS = $(SUBMIT_EXPRS)  MachineDiskString
SYSTEM_PERIODIC_HOLD_memory = MATCH_EXP_MachineMemory =!= UNDEFINED && \
                       MemoryUsage > 1.0*int(MATCH_EXP_MachineMemoryString)
SYSTEM_PERIODIC_HOLD_disc = MATCH_EXP_MachineDisk =!= UNDEFINED && \
                       DiskUsage > int(MATCH_EXP_MachineDiskString)
SYSTEM_PERIODIC_HOLD = ($(SYSTEM_PERIODIC_HOLD_disc)) || ($(SYSTEM_PERIODIC_
↪HOLD_memory))
SYSTEM_PERIODIC_HOLD_REASON = ifThenElse(SYSTEM_PERIODIC_HOLD_memory, \
                           "Used too much memory", ""), ifThenElse(SYSTEM_
↪PERIODIC_HOLD_disc, \
                           "Used too much disk space","Reason unknown")

MEMORY_USED_BY_JOB_MB = ResidentSetSize/1024
MEMORY_EXCEEDED = ifThenElse(isUndefined(ResidentSetSize), False, ( ($(MEMORY_
↪USED_BY_JOB_MB)) > RequestMemory ))
PREEMPT = ($(PREEMPT)) || ($(MEMORY_EXCEEDED))
WANT_SUSPEND = ($(WANT_SUSPEND)) && ($(MEMORY_EXCEEDED)) =!= TRUE
WANT_HOLD = ( $(MEMORY_EXCEEDED) )
WANT_HOLD_REASON = \
        ifThenElse( $(MEMORY_EXCEEDED), \
        "$(MemoryUsage) $(Memory) Your job exceeded the amount of requested␣
↪memory on this machine.",\
         undefined )
```

Miscellaneous

```
##########################################
COUNT_HYPERTHREAD_CPUS=FALSE
START = TRUE
SUSPEND = FALSE
PREEMPT = FALSE
PREEMPTION_REQUIREMENTS = FALSE
KILL = FALSE
ALL_DEBUG = D_FULLDEBUG D_COMMAND
POOL_HISTORY_DIR = /var/log/condor/condor_history
KEEP_POOL_HISTORY = True
MaxJobRetirementTime    = (1 *  $(MINUTE))
CLAIM_WORKLIFE = 600
MAX_CONCURRENT_DOWNLOADS = 15
MAX_CONCURRENT_UPLOADS = 15
```

ROOT

Any release with major release number 6 should work.

Geant4

GATE-RTion requires Geant4 version 10.03.p03, compiled **without** multithreading.

GATE-RTion

GATE-RTion [GateRTion] is a special release of Gate [GateGeant4], dedicated to clinical applications in pencil beam scanning particle therapy. If all nodes run the same hardware, then this can be compiled and installed once on the shared disk of the cluster and then be used by all nodes. If the different cluster nodes have different types of CPU then it can be good to compile Geant4, ROOT and Gate-RTion separately on all nodes and install it on the local disks (always under the same local path).

After installation, a short shell script should be created that can be "sourced" in order to set up the shell environment for running `Gate`, including the paths not only of `Gate` itself, but also of the Geant4 and ROOT libraries and data sets. For instance:

```
source "/usr/local/Geant4/10.03.p03/bin/geant4.sh"
source "/usr/local/ROOT/v6.12.06/bin/thisroot.sh"
export PATH="/usr/local/GATE/GateRTion-1.0/bin:$PATH"
```

1.2.2 IDEAL installation

Installing the IDEAL scripts

For the current 1.0rc release, IDEAL is obtained by cloning from GitLab or unpacking a tar ball, provided by at the IDEAL source repository: https://gitlab.com/djboersma/ideal. In a future release (1.0), we hope that the code can simply be installed with `pip install ideal` (which would then also perform some of the post-install steps). The code should be installed on the *shared disk of the IDEAL cluster*. The install directory will be referred to in this manual as the "IDEAL top directory". The IDEAL top directory has the following contents:

Table 1: IDEAL top directory contents

Name	Type	Description
bin	Folder	Executable scripts and `IDEAL_env.sh`
cfg	Folder	System configuration file(s)
docs	Folder	Source file for this documentation
ideal	Folder	Python modules implementing the IDEAL functionality
gpl-3.0.txt	File	Open Source license, referred to by the *LICENSE* file
LICENSE	File	Open Source license
RELEASE_NOTES	File	Summary of changes between releases

The *first_install.py* script

IDEAL will not function correctly immediately after a clean install (cloning it from GitLab or extracting it from a tar ball).

Right after the install, it is recommended to run the `bin/first_install.py` script. This script will attempt to create a minimal working setup:

- Some additional python modules installed (using `virtualenv`) in a so-called "virtual environment" named `venv`.
- Folders for commissioning data (definitions of beam lines, CTs, phantoms), logs, temporary data and output need to be created.
- The available resources and the simulation preferences need to be specified in a "system configuration" file `cfg/system.cfg` in the IDEAL install directory.

The script tries to perform all the trivial steps of the installation. Simple examples of a beam line model, CT protocols and a phantom are provided. These examples are hopefully useful to give an idea of where and how you should install your own beam models, CT protocols and phantoms. The details are described in the *Commissioning* chapter.

This script is supposed to be run after all previous steps have been performed. Specifically:

- A Linux cluster is available running the same OS on all nodes (e.g. Ubuntu 18.04) and with a fast shared disk that is accessible by all cluster nodes and has at least 200 GiB of free space.
- Geant4, ROOT and GateRTion should all be installed on the shared disk. A `gate_env.sh` shell script is available to configure a shell environment (`source /path/to/gate_env.sh`) such that these are usable. Specifically, the `Gate --version` command should return the version blurb corresponding to "GateRTion 1.0".
- HTCondor is installed and configured. All nodes on the Linux cluster run the same OS (e.g. Ubuntu 18.04).
- Python version 3.6 or newer and `virtualenv` are installed.

The `first_install.py` script will thoroughly check these assumptions but the checks are not exhaustive.

The minimum input for the script is the file path of the `gate_env.sh` script. It is recommended to also give the name of the clinic (with the -C option). Many more options are available, see the script's '–help' output.

Installing necessary python modules

The installation step described in this section is performed by the *first_install.py script*.

If you did *not* run the `first_install.py` script, then please read the rest of this section.

IDEAL needs several external python modules that are not included in a default python installation. In order to avoid interference with python module needs for other applications, the preferred way of installing these modules is using a virtual environment called `venv` in the IDEAL top directory. This may be done using the following series of commands (which may be provided in an install script in a later release of IDEAL) in a bash shell after a `cd` to the IDEAL top directory:

```
virtuelenv -p python3 --prompt='(IDEAL 1.0) ' venv
source ./venv/bin/activate
pip install filelock htcondor itk matplotlib numpy pydicom
pip install python-daemon python-dateutil scipy
deactivate
```

(The modules `ipython`, `Sphinx` and `PyQt5` are optional. The first enables interactive, python-based analysis, the second enables you to generate these docs yourself, and the third enables the somewhat clunky `sokrates.py` GUI interface.)

If you decide to install the virtual environment under a different path, then you need to edit the `bin/IDEAL_env.sh` script to use the correct path for `source /path/to/virtualenv/bin/activate` line, or to remove that line altogether.

Installing additional python modules

You can of course add extra modules with `pip`. There are three modules in particular that might be desirable when working with IDEAL:

- `ipython`: a python command line program, which can be useful for debugging (e.g., query DICOM files using the `pydicom` module)
- `Sphinx`: enables you to generate these docs yourself (`cd docs; make html`).
- `PyQt5`: enables running the `sokrates.py` GUI. It's a bit clunky, but some users like it.

In a fresh shell, `cd` to the IDEAL install directory and then run:

```
source ./venv/bin/activate
pip install ipython Sphinx PyQt5
deactivate
```

Alternatively, in a shell in which you already ran `source bin/IDEAL_env.sh`, you can directory run `pip install ipython Sphinx PyQt5`.

Set up the data directories

Like the virtual environment, this installation step may be automated in an installation step in the next release. IDEAL needs a couple of folder to store logging, temporary data and output, respectively. In a bash shell after a `cd` to the IDEAL top directory, do:

```
mkdir data
mkdir data/logging
mkdir data/workdir
mkdir data/output
mkdir data/MyClinicCommissioningData
```

The subdirectories of `data` are described in more detail below.

`logging`

The `logging` directory is where all the debugging level output will be stored. In case something goes wrong, these logging files may help to investigate what went wrong. When you report issues to the developers, it can be useful to attach the log file(s).

`workdir`

The `workdir` directory will contain a subfolder for every time you use IDEAL to perform a dose calculation. The unique name of each subfolder is composed of the user's initials, the name and/or label of the plan and a time stamp of when you submitted the job. The subfolder will contain all data to run the GATE simulations, preprocessing the input data and postprocessing the output data:

- The GATE directory with the `mac` scripts and data needed to run the simulations.
- Temporary output, saved every few minutes, from all condor subjobs running this simulation.
- Files that are used or generated by HTCondor for managing all the jobs.
- Three more IDEAL-specific log files, namely: `preprocessor.log`, `postprocessor.log` and `job_control_daemon.log`.

The temporary data can take up a lot of space, typically a few dozen GiB, depending on the number of voxels in CT (after cropping it to a minimal bounding box containing the "External" ROI and the TPS dose distribution) and on the number of cores in your cluster. After a successful run, the temporary data is archived in compressed form, for debugging analysis in case errors happened or if there are questions about the final result.

Note: When an IDEAL job runs unsuccessfully, the temporary data is **NOT** automatically compressed/archived, since the user may want to investigate. Do not forget to delete or compress these data after the investigation has concluded, to avoid inadvertently filling up the disk too quickly.

After compressed archiving job work directory still still occupies up to a few GiB per plan, which will add up when running IDEAL routinely for many plans[3].

`output`

The `output` directory will contain a subfolder of each IDEAL job, using the same naming scheme as for the work directories. In IDEAL's *system configuration file* the user (with admin/commissioning role) can define which output will actually be saved, e.g. physical and/or effective dose, DICOM and/or MHD. This output directory serves to store the original output of the IDEAL job. If the path of a second output directory is given in the *system configuration file*, then the job output subfolder will be copied to that second location (e.g. on a CIFS file share, where it can be accessed by users on Windows devices).

Commissioning Data Directory

In the example `MyClinic` could be replaced by the name of your particle therapy clinic. If you are a researcher who studies plans from multiple different clinics, may want to create a commissioning data directory for each clinic.

This directory will contain the commissioning data for your particle therapy clinic. The details are laid out in *the commissionig chapter*.

1.3 Commissioning

1.3.1 System configuration file

The configuration file contains the specifications of the computing environment, the preferences of the users and much more. IDEAL cannot run without these specifications. The standard location of the system configuration file is `cfg/system.cfg` relative to the installation directory of IDEAL.

The syntax of the configuration file is similar to what is found in Microsoft Windows IN files. The configuration is divided in `[sections]` which contain lists of settings of the form `label = value`. The labels are case insensitive. The example `system.cfg` file provided with the IDEAL source contains many explanatory comments that should help the commissioning user to define the correct values.

`[directories]`

`input dicom` Full path of the top directory for finding DICOM input data. This is mainly used by the research GUI, to position the starting location for the input file browser.

`tmpdir jobs` Full path of the "work directory" on the shared disk of the cluster. For each IDEAL job, a subfolder will be created in this directory to store all the job specific data, including the intermediate and final results of the cluster subjobs. Upon successful completion of the job, most of the bulky data in the subfolder will be compressed, but not deleted.

`first output dicom` Full path of the output directory on the shared disk of the cluster. For each IDEAL job, a subfolder will be created in this directory to store the final output (DICOM dose files and a text summary of the job).

[3] A future release of IDEAL might run an automatic clean up of the oldest temporary data in order to ensure that the new jobs will not run out of disk space.

second output dicom Full path of the directory on an external file system (typically a Windows file share, mounted with SAMBA). The for each job, a copy of the output subfolder is stored here. This setting is optional: leaving this option empty avoids the copy.

logging Full path of the logging directory. For every IDEAL command, a logging file with all debug level logging output will be stored here.

commissioning Full path of the so-called "commissioning" directory. This contains site specific calibration and modeling data, such as beam models and HLUT tables, which are described in more detail below.

[mc stats]

There are three possible statistical goals for an IDEAL job: "number of primaries", "average uncertainty" and "maximum runtime". The ranges of allowed values and the default values of these goals are specified here, as well as which goals are enabled by default. This information is primarily used by the GUI.

Four values should be specified for each goal, to specify the minimum, default, maximum, and stepsize value, respectively. All values should be positive and the default value should be within the allowed range. It is recommended to choose the stepsize identical to the minimum value. For the goal(s) that should be enabled by default, the word "default" should be put at the end of the configuration line. At least one goal should should be enabled.

For instance:

```
n minutes per job   =    5        20       10080     5
n ions per beam     =  100    1000000 1000000000   100    default
x pct unc in target =    0.1        1.         99.    0.1
```

In this example, by default the goal is to simulate 1000000 primaries. The valid values for the number of primaries are in the range from 100 to 1000000000, and in when a GUI user clicks to increment or decrement, then the stepsize is 100. Since Gate uses a signed integer to count primaries, the maximum should be less than the number of cores in the cluster times 2^{31}-1=2147483647.

The average uncertainty in the target is defined as the average of the relative uncertainty in the voxels that have a dose larger than the a configurable fraction of the mean dose maximum. We are not using the absolute dose maximum in a voxel, since this tends to fluctuate too much. Instead we find the N highest dose values in the distribution, the mean of those values is the "mean dose maximum". The number N should be be configured in the system configuration file like this:

```
n top voxels for mean dose max = 100
```

The fraction of this "mean dose maximum" that serves as the threshold to mask the voxels that should contribute to the "average uncertainty" is configured like this:

```
dose threshold as fraction in percent of mean dose max = 50.
```

In this example configuration, the "mean dose maximum" is computed from the 100 highest dose values and the threshold is set at 50% of that maximum mean dose.

[simulation]

gate shell environment Full path to a shell file that will be sourced to set the environment variables correct for running Gate (GateRTion). Typically this shell script contains lines like `source /path/to/ROOT/bin/thisroot.sh` and `source /path/to/Geant4/bin/geant4.sh` to set the ROOT and Geant4 variables, plus a line like `export PATH="/path/to/Gate/bin:$PATH"`. If GateRTion was compiled with `USE_ITK` then an additional line adding the ITK library directory may need to be prepended to `LD_LIBRARY_PATH` as well.

number of cores This is the number of cores that will be used to simulate a single beam. On a small cluster you will typically set this to the total number of available cores. On a medium/large cluster (>100 cores) you could set it to a smaller number, for instance if you would to leave cores free for other use, if you would like to have several simulation jobs run in parallel or if for some reason there is limited disk space available for the temporary job data (depending on dose grid size, up to a gigabyte per core).

`proton physics list` Geant4 physics list for protons. Recommended setting: `QGSP_BIC_HP_EMZ`

`ion physics list` Geant4 physics list for ions. Recommended setting: `Shielding_EMZ`

`rbe factor protons` Usually 1.1

`air box margin [mm]` During preprocessing, the CT image is cropped down to enclose the minimum bounding box around the air-padded External ROI and TPS dose distribution. The GATE simulation uses different cut parameters and step sizes inside and outside the cropped CT volume: crude simulation outside, detailed simulation inside. The air padding serves to ensure that the detailed simulation will always start a little bit *before* the particles enter the External ROI volume. The default margin is 10 mm.

`remove dose outside external` Outside the external (in the air, typically) the dose is about the same as inside, but most TPS never show this. If you set this option then IDEAL will mask the final dose output, all voxels outside of the external are forced to have dose equal to zero. This is e.g. useful to avoid artifacts in gamma analysis.

`gamma index parameters dta_mm dd_percent thr_gray def` If leave this setting empty, then IDEAL will not attempt any gamma index calculations. If you provide four nonzero values, then IDEAL will try to compute the gamma index distribution with the TPS dose as the reference image, but only for the voxels that are a dose above a given threshold.

- `dta_mm`: distance parameter in mm
- `dd_percent`: relative dose difference parameter in percent
- `thr_gray`: threshold value in Gray
- `def`: default gamma value for target voxels with dose values below threshold

`stop on script actor time interval [s]` On each core, the simulation periodically saves the intermediate result for the dose distribution and for the simulation statistics (including the number of primaries simulated so far), and checks if the job control daemon has set a flag to indicate that the statistical goal (number of primaries, average uncertainty, and/or time out) has been reached and that the simulation should stop. This setting specifies the time interval between such save & check moments. Setting this too short will result in a slow down due to network overload, setting it too long will result in overshooting the statistical goals. Two minutes is a reasonable value for this setting. For medium/large number of cores (>100) it could possibly be good to choose longer times.

`htcondor next job start delay [s]` When HTCondor "stages" the GateRTion jobs (starts the jobs on the calculation nodes), it starts them not all at the same time, but rather with a small delay between each job and the next. This is done on purpose, because all jobs will start by reading lots of input and configuration data. We want to avoid or at least reduce the stampede effect in which all these read requests are clogging up the network. The best value for this delay needs to found empirically on your network. On a fast network (and with fast network cards on all nodes), that is 10Gbit/s or faster, with about 50 physical cores, 1 second delay seems enough, but on a slower network, e.g. 1Gbit/s, it is advisable to choose a larger delay, for instance 10 seconds. It is advisable to make sure that this delay value times the number of cores is less than the `stop on script actor time interval [s]`.

`minimum dose grid resolution [mm]` The user can configure a dose resolution that is different from the TPS dose resolution by changing the number of voxels. Too fine grained resolution will be costly on resources (RAM, disk space) so there is a limit for this, defined by the minimum size that a dose voxel should have in each dimension.

This may not be the optimal way to guard against resource overusage, but it's simple and intuitive. In later releases we could define other safeguards, maybe a maximum for the total number of voxels in the dose distribution.

Output options The dose distribution can be saved in several stages of the calculation and in various formats. You can configure which ones you would like to have:

- `write mhd unscaled dose`: sum of the dose distributions from all simulation jobs, computed in the CT geometry (cropped to a minimal box around the TPS dose distribution and the External

ROI). Since the total number of simulated primaries is much smaller than the total number of particles planned, this dose is much lower than the planned dose. This dose can be useful for debugging purposes and if this option is set then this dose will be exported in MHD format.

- `write mhd scaled dose`: this is the unscaled dose multiplied with the '(tmp) correction factor' (see below) and with the ratio of the number of planned particles over the number of simulated particles. For example, if the correction factor is 1.01, 10^{11} particles were planned for each of 30 fractions, and 10^{8} particles were simulated, then the scaling factor is 30300.
- `write mhd physical dose`: this is the scaled dose, resampled (using mass weighted resampling) to the same dose grid as the TPS dose distribution. Saved in MHD format.
- `write dicom physical dose`: this is the scaled dose, resampled (using mass weighted resampling) to the same dose grid as the TPS dose distribution. Saved in DICOM format.
- `write mhd rbe dose`: for protons, the "Relative Biological Effect" dose is estimated by mulitplying the physical dose by the *RBE factor for protons* setting (typically 1.1). Saved in MHD format. IDEAL can currently not compute RBE dose for other particles than protons.
- `write dicom rbe dose`: Save the RBE dose (for protons) in DICOM format.
- `write mhd plan dose`: Compute the plan dose (sum of physica/RBE beam doses) and save in MHD format.
- `write dicom plan dose`: Compute the plan dose (sum of physica/RBE beam doses) and save in DICOM format.

[(tmp) correction factors]

The computed dose distribution for a given treatment plan may need to be corrected by a constant factor to correct for a normalization error that can be due to various causes. Typically this factor is determined using plans and absolute dose measurements on (water) phantoms. In future releases of GateRTion this correction factor will be integrated in the beam model, hence the '(tmp)' in the section name. The correction factor can be defined for each beam line and each radiation type separately, using "TreatmentMachineName RadiationType" as the label and a floating point number (typically close to 1.0) as the value. For combinations of beamlines and radiation types that are not explicitly configured, the "default" value will be used. The radiation type is "proton" for protons and "ion_Z_A_Q" for ions, where Z is the number of protons, A is the atomic number (number of protons plus number of neutrons) and Q is the electric charge of the ion in units of `e`. The radiation type for carbon ions is `ION_6_12_6` and for helium ions it is `ION_2_4_2`. For example:

```
default = 1.0
IR2HBL ION_6_12_6 = 1.0
IR2HBL proton = 0.97371
IR2VBL proton = 1.00
IR3HBL proton = 0.97371
```

[condor memory]

For typical IDEAL jobs, all core will be used. But for calculations with a large CT, high resolution dose distribution and/or many pencil beams, the RAM usage each Gate process can be so high that it's better not to run on all cores. An IDEAL job submission to HTCondor includes a "memory request". This will do two things:

1. HTCondor will assume that the job is going to use the requested amount of RAM at some point during the run. In order to avoid oversubscribing the RAM and cause swapping, HTCondor will not start running any new jobs if the sum of the requested amounts of RAM would exceed the available RAM.
2. If a running job uses *more* than the requested amount of RAM for a too long period of time, then that job will be killed or set on hold by HTCondor (the exact policy details can be configured in the *HTCondor configuration files*).

You can specify a minimum, default and maximum value of the memory request, in units of megabyte. IDEAL makes an estimate of the required RAM with a simple linear formula: `RAM = offset + cA*A + cA*B +`

`....` Here `A`, `B` are quantities that are expected to impact the memory usage of the Gate simulation, such as the number of dose voxels, number of CT voxels and the number of spots. The estimate will be different for CT and for phantom simulations (because the phatom does not have "CT voxels"), and for different particles (the cross section tables, which are responsible for an important part of the RAM usage, are significantly larger for carbon ions than for protons). The linearity coefficients (`cA`, `cB`, etc.) can be set by the user, based on system observations during a series of test runs with differently sized CTs, phantoms, dose resolutions and plans. The values given in the example below may be a good starting configuration for your local cluster, but may need tweaking depending on the available RAM and other factors.

Example configuration:: # how much memory should condor allocate per job? # Condor uses the unit "MB", which might mean either 1024*1024 bytes or 1000*1000 bytes. condor memory request minimum [MB] = 7000 condor memory request default [MB] = 8000 condor memory request maximum [MB] = 50000 condor memory fit proton ct = offset 1200 dosegrid 2.5e-05 ct 1.8e-06 condor memory fit proton phantom = offset 500.0 dosegrid 2.0e-05 nspots 0.0060 condor memory fit carbon ct = offset 1800 dosegrid 5e-05 condor memory fit carbon phantom = offset 1000.0 dosegrid 8.0e-06 # if e.g. a proton plan gets a dose grid of 200*200*200=8e6 voxels and a ct with 16e6 voxels # then the memory fit gives 1200 + 8e6*2.5e-5 + 16e6 * 1.8e-6 = 1428.8 MB estimated max RAM usage

[materials]

Everything about materials and material overrides.

`materials database` With this setting the basename of the material database should be specified. IDEAL then expects to find a file with this name in the `material` subdirectory of the *Commissioning Data Directory*. This file *can* just be the standard database file GateMaterials.db that is included in the source code package for Gate and defines a large number of materials that are important typical Gate applications but are not included in the standard set of Geant4 materials. The Gate material database is a simple text file, which you can extend with any additional materials that are relevant in your clinic. It is recommended to give such an extended database file a name that makes it clear that this database is different from the standard material database file. E.g.:

```
materials database = MyClinicalMaterials2020.db
```

`hu density tolerance [g/cm3]` The Schneider 2000 method is used to convert Hounsfield Unit (HU) values to materials, based on a density curve and a material composition table (see section *CT protocol Hounsfield Lookup tables*). For each new combination of density and composition Geant4 needs to define a new material. The density tolerance value defines the maximum difference between two "equivalent" density values. The full range of HU values is segmented in intervals such that the densities within each interval are equivalent to each other, and only one material defintion is associated with each interval.

Override materials You may sometimes want to override the material in a region of interest (ROI) in a CT with a particular material from the database, to be used by Gate/Geant4 instead of the materials given by the Schneider tables. A typical use case for this is that in a CT of a water phantom, the volume of the phantom is overridden with `G4_WATER`, the standard Geant4 definition of water. For all materials that you expect to use for such overrides (Geant4 materials or materials that you provide in the material database file, see above), you should add a line in this section of the system configuration file that associates the name of the material with a density (in units of `g/cm3`). For instance:

```
G4_WATER = 1.0
G4_AIR = 0.00120479
G4_GRAPHITE = 2.21
G4_STAINLESS-STEEL = 8.0
G4_ALANINE = 1.42
PMMA = 1.195
Tungsten = 19.3
G4_Ti = 4.54
```

It would be nice if you only needed to give the list of the names of the allowed materials and that IDEAL would figure out the densities somehow from Geant4's and Gate's databases. This may be implemented in a future release, hopefully. For now, you just need to make sure that the density values that you give here are consistent with what the Gate and Geant4 databases are using.

[user roles]

This is the list of users that are expected/allowed to use this installation of IDEAL in a particular role. For users with more than role you need to add one line per role. Each line is of the form `ACRONYM, username = role`. Here `username` is a name without whitespaces that contains part of the name of a user plus a word that indicates the role. ACRONYM is a short version of the user name, for instance just the user's initials. The role can be `clinical`, `commissioning` or `admin` (see *Intended Users*). Each username and acronym should be unique.

For instance:

```
OKE, obiwan = clinical
OKE_A, obiwan_admin = admin
OKE_C, obiwan_commissioning = commissioning
LSK, luke = clinical
LSK_C, luke_commissioning = commissioning
LES, leiha = clinical
HSO, han = clinical
DVA, darth = commissioning
YOD_A, yoda_admin = admin
YOD_C, yoda_commissioning = commissioning
```

1.3.2 Beam line modelling

The beam modelling information stored under the `beamlines` subdirectories of the *Commissioning Data Directory*. For each treatment machine in the clinic, a subdirectory to the `beamlines` directory should be created, with the name equal to the same treatment machine name that is also used in DICOM plan files.

One special beamline directory is `common`, which can contain specifications (of nozzle components or generic passive elements) that can be used in all beamlines. Subfolders can be created under `common` to organize the resources.

A beam model consists of two text files, namely a "source properties" text file and a beamline details macro file.

Source properties file

The source properties file is a one of the main inputs for the Gate TPS pencil beam actor. The name of the source properties file should be of the form `<TreatmentMachineName>_<RadiationType>_source_properties.txt`. For instance, for a beamline named `IR2HBL` that can be used both for protons and carbon ions, two source properties files should be provided, named `IR2HBL_PROTON_source_properties.txt` and `IR2HBL_ION_6_12_6_source_properties.txt`, respectively. The source properties file defines:

- in which plane the source particles will be generated
- source axis distance (X and Y)
- optical properties (beam size, angular spread, emittance; polynomials in energy)
- range properties (energy offset between generation and isocenter, energy spread; polynomial in energy)
- monitor unit calibration (polynomial in energy)

An example of a source properties files can be found in the GateContrib project on GitHub. The procedure for fitting a pencil beam model for Gate is described in [FuchsBeamlineModels2020]. The scripts that were used for that publication are (at the time of writing these docs) not yet open source, but we hope that they soon will be.

Todo: IDEAL currently supports only one single beam model per beam line. If your clinic has e.g. two different spot tunes, e.g. one spot tune with smaller and one with larger spot sizes, then this distinction is not supported by IDEAL. A simple solution (in a future release of IDEAL) might be to include the spot tune label (often this is a version number, like "3.0") in the names of the source properties files.

Beamline details macro file

This macro file should be named `<TreatmentMachineName>_beamline_details.mac`. For the example of a beamline named `IR2HBL`, this would be `IR2HBL_beamline_details.mac`. The information in it should be formatted using the Gate/Geant4 macro language

The minimal content for this file is to specify for each supported radiation type and for each of the two lateral directions whether the beam is convergent (or not) in the source generation plane. This is done by defining an alias that will be used by IDEAL in the configuration of the TPS PencilBeam source actor[1]. The names of the two aliases should be `<RADIATION_TYPE>_XTHETA` and `<RADIATION_TYPE>_YPHI`; for instance, for a beamline which supports two radiation types, namely a convergent proton beam and a divergent carbon beam, the definitions would be:

```
/control/alias PROTON_CONVERGENCE_XTHETA     true
/control/alias PROTON_CONVERGENCE_YPHI       true
/control/alias ION_6_12_6_CONVERGENCE_XTHETA false
/control/alias ION_6_12_6_CONVERGENCE_YPHI   false
```

In addition to the convergence information, the beamline details may contain the definition of the geometry of the parts of the nozzle that are relevant for particle propagation between the generation plane and the isocenter, for example the ionization chambers and collimators (if any).

For elaborate nozzle models, or for modeling several nozzles with common design elements, it may be preferable to split up the code in multiple source files, using the standard Geant4 `/control/execute data/macro.mac` directive. Those extra files can reside either in the common beamline folder, or in the treatment machine specific beamline subfolder. In the latter case, the names of the extra files and folders should all start with the treatment machine name. Relative paths should be used to refer to the executable macros and other input data. All files and folders in the `common` folder and in the treatment machine specific folder will be copied into the `data` subfolder of the Gate working directory for each IDEAL job; hence all paths to input data and to external macros run with `/control/execute` should be referred to with a path that starts with `./data`.

Optional beamline description text file

Finally, each beamline directory optionally can contain a text file `<TreatmentMachineName>_description.txt` with a short description of the beamline that can be displayed by the IDEAL user interaces to the user.

Alternative beamline models for commissioning/research

A commissioning/research user may create a beamline model directory with a name that is different from all actual "treatment machine names" in the clinic or data set[2]. The beamline model in this directory will only be used if the user explicitly overrides the beamline model to use for IDEAL, instead of the model(s) that have the same name as the treatment machines. This can be desirable for commissioning and research activities (and should be avoided by users in a 'clinical' role). For instance, it might be instructive to compare a simple beam model in which primary particles are generated at the exit of the nozzle with a more elaborate model that starts at the entrance of the nozzle and takes all nozzle details into account.

[1] This convergence information should really be part of the source properties file. Also, for some beamlines the convergence may depend on energy, e.g. for low energies the beam is divergent and for higher energies it is convergent. In a future releases of Gate (and GateRTion, and IDEAL) these issues will hopefully be addressed; the convergence information should be part of the source properties file and the parametrization should allow this to be energy dependent.

[2] It is advisable to choose beamline model names with only alphanumerical characters, so without e.g. spaces, commas and parentheses.

Note that the name of the alternative beamline model should be used everywhere where normally the "treatment machine name" would be used. E.g. if you have a beamline named IR5 and you create an alternative beamline model named MyToyModelForIR5, then the source properties file for protons should be named MyToyModelForIR5_PROTON_source_properties.txt.

1.3.3 CT protocol Hounsfield Lookup tables

The *Commissioning Data Directory* should have a CT subdirectory. This directory contains the hlut.conf configuration file and three subdirectories, density, composition and cache.

The hlut.conf configuration file describes for each CT protocol which DICOM tags and values should be used to recognize it, as well as how to convert the HU values to materials. This text file should be edited by the commissioning physicist to provide the data for each of the relevant CT protocols in the clinic where IDEAL is installed.

There are two different ways to define the conversion from HU to materials.

1. For [Schneider2000] type conversion, provide the names of density and composition text files, which are expected to be found in the respective subfolders.
2. Provide a direct HU-to-material conversion table. A list of HU intervals can be given, and for each interval the name of the material is specified. This can be either a Geant4 material (e.g. G4_WATER) or a material that is defined in the GateMaterials.db file (e.g. PMMA).

Examples of the density and composition files for Schneider-type material conversion can be found in the GateContrib project on GitHub: * density * composition

IDEAL will not directly use these tables. Rather, for every new combination of density, composition and density tolerance value (usually set in the system configuration file) it will generate a new material database and an interpolated HU-to-material table, in which associates HU intervals directly with a material from the the new database. The HU intervals are chosen just small enough such that the steps in density between the subsequent materials are less than the density tolerance value. These generated tables are stored in a cache folder CT/cache and will be reused in following IDEAL jobs with the same combination of CT protocol and density tolerance.

1.3.4 Phantoms

In the phantoms subdirectory of the *Commissioning* directory you can define the phantoms that may be used in IDEAL simulations, instead of the CT image, for instance to simulate patience specific quality assurance (PSQA) measurements.

1.3.5 Simulation parameters

The so-called "cut parameters" in Geant4/Gate simulations define the level of detail into which particles need to be tracked. This implies a trade-off between simulation accuracy and speed. Typically, for the volume outside of the patient volume speed is more important than accuracy, while inside the patient volume the priorities are reversed: accuracy is more important than speed.

The three kinds of parameters that need to be configured, per volume and per particle, are the "cut in region", "minimum remaining" range and the "maximum step size". Recommended values for clinical applications were reported in [Winterhalter2020].

The settings of the "cut parameters" can be different for computing the dose in the CT or in a custom phantom. IDEAL expects to find these settings settings in Gate macro files named ct-parameters.mac and phantom-parameters.mac, respectively, stored in the material subdirectory of the *Commissioning Data Directory*.

For CT simulations, the names of the relevant volumes are world, patient_box and patient. The patient volume is the the CT volume, the patient_box is a slightly larger volume that includes a layer of air around the patient, and the world volume encloses everything, including the patient, the nozzle and the particle generation plane.

For phantom simulations, the names of the non-`world` volumes are given in the phantom specifications. The name of the phantom selected by the user is available via the Geant4 alias `{phantom_name}`.

An example of the parameter settings for CT is given below.

```
###########################################################
# cut
###########################################################

# TODO: would be nice to use the Winterhalter values here.

# in air
/gate/physics/Gamma/SetCutInRegion      world 1 m
/gate/physics/Electron/SetCutInRegion   world 1 m
/gate/physics/Positron/SetCutInRegion   world 1 m

# in air close to patient
/gate/physics/Gamma/SetCutInRegion      patient_box 1 mm
/gate/physics/Electron/SetCutInRegion   patient_box 1 mm
/gate/physics/Positron/SetCutInRegion   patient_box 1 mm

# in patient
/gate/physics/Gamma/SetCutInRegion      patient 0.5 mm
/gate/physics/Electron/SetCutInRegion   patient 0.5 mm
/gate/physics/Positron/SetCutInRegion   patient 0.5 mm

###########################################################
# tracking cut
###########################################################

#Tracking cut in air
/gate/physics/SetMinRemainingRangeInRegion world 10 mm
/gate/physics/ActivateSpecialCuts e-
/gate/physics/ActivateSpecialCuts e+

#Tracking cut in air close to patient
/gate/physics/SetMinRemainingRangeInRegion patient_box 1 mm
/gate/physics/ActivateSpecialCuts e-
/gate/physics/ActivateSpecialCuts e+

#Tracking cut in patient
/gate/physics/SetMinRemainingRangeInRegion patient 0.5 mm
/gate/physics/ActivateSpecialCuts e-
/gate/physics/ActivateSpecialCuts e+

###########################################################
# step limiter
###########################################################

#stepLimiter in air
/gate/physics/SetMaxStepSizeInRegion world 1 m
/gate/physics/ActivateStepLimiter proton
/gate/physics/ActivateStepLimiter deuteron
/gate/physics/ActivateStepLimiter triton
/gate/physics/ActivateStepLimiter alpha
/gate/physics/ActivateStepLimiter GenericIon

#stepLimiter in air close to patient
/gate/physics/SetMaxStepSizeInRegion patient_box 1 mm
/gate/physics/ActivateStepLimiter proton
/gate/physics/ActivateStepLimiter deuteron
/gate/physics/ActivateStepLimiter triton
/gate/physics/ActivateStepLimiter alpha
```

(continues on next page)

(continued from previous page)

```
/gate/physics/ActivateStepLimiter GenericIon

#stepLimiter in patient
/gate/physics/SetMaxStepSizeInRegion patient 0.5 mm
/gate/physics/ActivateStepLimiter proton
/gate/physics/ActivateStepLimiter deuteron
/gate/physics/ActivateStepLimiter triton
/gate/physics/ActivateStepLimiter alpha
/gate/physics/ActivateStepLimiter GenericIon
```

1.4 User Manual

1.4.1 The IDEAL workflow

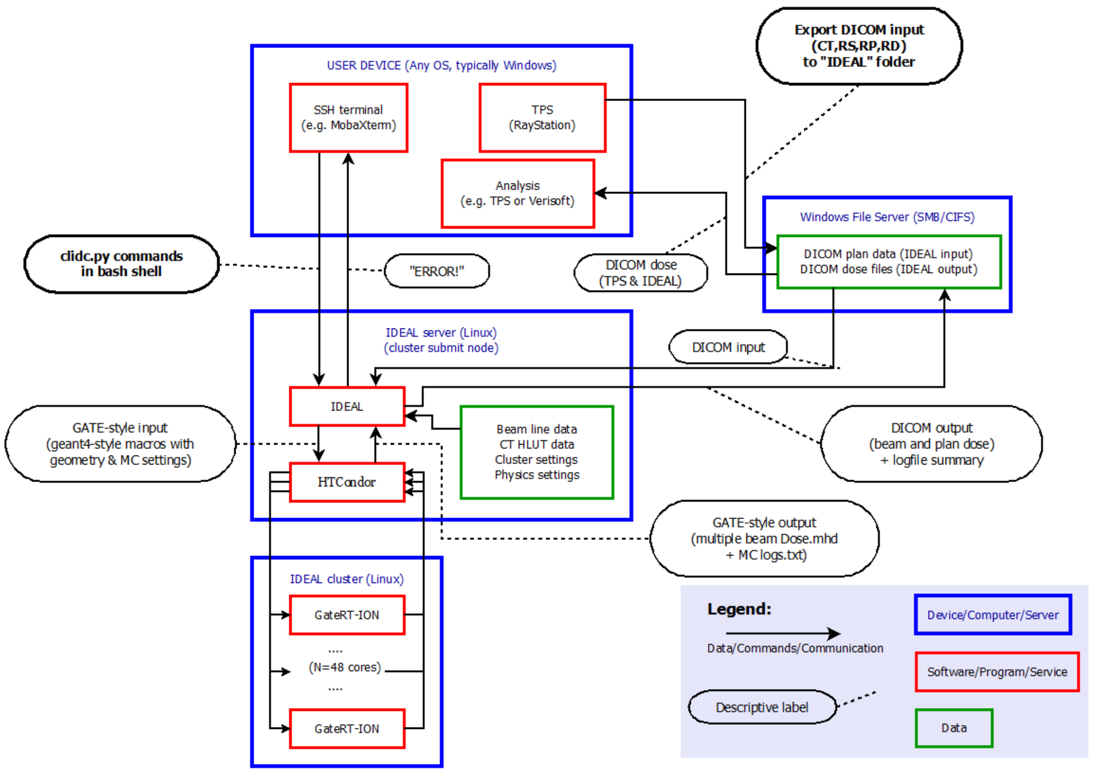
An independent dose calculation with IDEAL

There are different user interfaces for IDEAL, but the basic workflow is always as follows:

- User starts user interface with username and role
- User provides DICOM input data (CT, RS, RP, RD) * For instance by exporting them from the TPS as DICOM files on a shared file system.
- User chooses MC settings:
  - required: statistical goal (number of primaries, uncertainty, time out, or a combination of these)
  - optional: CT protocol[1] (required if CT protocol is not automatically recognized).
  - optional: material overrides (default: only `G4_AIR` for volume outside of external)
  - optional: spatial resolution overrides (default: same as TPS dose)
  - optional: which beams to simulate (default: all of them)
- GATE-RTion Simulation runs on the cluster (typically for a couple of hours, the actual runtime depends on many factors)
- Beam dose distributions are exported by IDEAL as DICOM files (and MHD format, if configured that way)
- User analyzes and evaluates dose distributions using third party tools (e.g. [VeriSoft] or a TPS like [RayStation])

For a user in a light ion beam therapy clinic, this workflow is illustrated in figure below.

[1] The CT protocol selection may be automated in later release, but the users will still be able to override the protocol.


Export DICOM input (CT,RS,RP,RD) to "IDEAL" folder
USER DEVICE (Any OS, typically Windows)
SSH terminal (e.g. MobaXterm)
TPS (RayStation)
Analysis (e.g. TPS or Verisoft)
Windows File Server (SMB/CIFS)
DICOM plan data (IDEAL input)
DICOM dose files (IDEAL output)
clidc.py commands in bash shell
"ERROR!"
DICOM dose (TPS & IDEAL)
IDEAL server (Linux) (cluster submit node)
DICOM input
IDEAL
Beam line data
CT HLUT data
Cluster settings
Physics settings
GATE-style input (geant4-style macros with geometry & MC settings)
HTCondor
DICOM output (beam and plan dose) + logfile summary
GATE-style output (multiple beam Dose.mhd + MC logs.txt)
IDEAL cluster (Linux)
GateRT-ION
....
(N=48 cores)
....
GateRT-ION
Legend:
Data/Commands/Communication
Descriptive label
Device/Computer/Server
Software/Program/Service
Data

Longer term work flow

1. You work in an proton/ion beam therapy clinic and decide that Geant4-based independent dose calculations may be useful in your clinic.
2. A sufficiently powerful and extendible computer cluster is acquired, with resources as described in *System requirements*.
3. IDEAL is installed on this cluster as described in *IDEAL installation* by an employee with an admin role.
4. The medical physicist(s) with a "commissioning" role develops the beam model(s), installs the CT Hounsfield lookup tables and edits the *system configuration file* with correct values.
5. The medical physics team runs a series of tests to make sure that the installation went well. This should probably include:

- Integrated depth dose curves for several energies in water (using a CT of a water phantom)
  - compare with measurements (both the shape of the curve and the absolute dose)
  - determine/adjust correction factors
- Beam profiles in air, compare with measurements
- Example treatment plans from recent clinical practice:
  - Check level of agreement, find any systematic offsets in dose or in geometry
  - Determine cluster performance:
    * how does the simulation runtime vary with statistical goals and with complexity of the plan (number of beams, number of spots, radiation type, etc)
    * are there any crashes (e.g. due to insufficient memory)?
  - Check all combinations of beamline, radiation type, passive elements and CT protocol that you expect to use

- **You are yourself responsible for the correct use of IDEAL and any consequences of using its results. If you intend to use IDEAL clinically, then you need follow all procedures for using "Software Of Unknown Provenance" (SOUP). IDEAL is NOT a medically certified product.**

6. Use IDEAL to perform independent dose calculations in your clinic or for research.
   - The commissioning physicist should make sure that the commissioning data are correct and up to date. Whenever new beamlines or CT protocols are introduced or the existing ones are changing, the commissioning data should be updated.
   - The admin should make sure that the disks do not fill up, by regularly purging data from older simulations.
   - Periodically review the performance of your IDEAL installation:
     - with experience you may improve some choices made during installation and commissioning
     - buy more cluster nodes, more RAM (and update the *System configuration file*)
     - share your observations, experiences and ideas for improvement with the IDEAL development team

1.4.2 Requirements and limitations on DICOM input

What kind of treatment plans can be studied using IDEAL?

- The input treatment plan should be a plan for scanning ion beam treatment. IDEAL is **not** designed to work for e.g. passive scattering proton therapy and conventional radiation therapy with photons.
- IDEAL takes the DICOM plan file as input, and expects to find the following DICOM files in the same directory (or in subfolders): * Structure set file, with the `SOP Instance UID` that is referred to by the treatment plan, * CT files, with the `Series Instance UID` that is referred to by structure set, * TPS dose file(s), which refer to the `SOP Instance UID` of the treatment plan.
- Hounsfield lookup tables for the protocol that was used for the input CT should be available under *CT protocol Hounsfield Lookup tables* in the *Commissioning Data Directory*.
- Beam line model for the 'Treatment Machine' should be made available under *Beam line modelling* in the *Commissioning Data Directory*.
- In case multiple TPS dose files are found (physical and/or effective dose, dose per beam and/or dose for the whole beamset) it is important that all have the same geometry (origin coordinates, voxel size a.k.a. "spacing", number of voxels in each dimension a.k.a. "image size").

Full details of how IDEAL uses DICOM can be found in the *DICOM specification*.

1.4.3 Details per user interface

Command line interface: `clidc.py`

The name `clidc` stands for: **C**ommand **L**ine interface for **I**ndependent **D**ose **C**alculation with "IDEAL".

The command is run in a bash shell environment (or equivalent, a Z-shell probably also works). In order to set up the environment to make it work, the user should first run `source /path/to/IDEAL/install/bin/IDEAL_env.sh`, where `/path/to/IDEAL/install` should obviously be replaced with the actual directory path where IDEAL was installed[2].

Ways to call this script:

- Get help, remind yourself of all the possible options:

```
clidc.py --help
```

[2] Once IDEAL has been fully "pip"-ified, this sourcing step might not be necessary any more.

- Query which hardware is supported (by the commissioning physicist) in the current installation:

```
# print the list of supported CT protocols, modelled phantoms, override␣
↪materials and beamline names.
clidc.py -C -P -M -Z
```

- Query a plan:

```
# query a specific treatment plan, to print to stdout:
# * list all ROIs in (the structure set used by) a plan
# * the default size (number of voxels) of the dose distribution
# * the list of beam names in the plan
clidc.py -R -X -B /path/to/some/RP_treatment_plan.dcm
```

- Configure & submit a plan for independent dose calculation:

```
# you need to provide at least one statistics goal and a treatment plan␣
↪filepath
# you need to provide EITHER a CT protocol OR a phantom name
# tweaks are optional
clidc.py -l user_role \
     [statistics options] \
     [ -c CT_protocol | -p phantom ] \
     [ configure tweaks ] \
     /path/to/some/RP_treatment_plan.dcm
```

So there are basically four groups of options:

General options (`lvsjd`) These options can be combined with any of the other options.

- `-l`: "login" or "username and role". This option specifies the username and role under which you would like to run `clidc.py`. The username and role is part of the output dose files. Some configuration options can only be used with `commissioning` role (e.g. make the beamline model than the default model for the `TreatmentMachine`). The default value for `-l` is the Linux user name under which the `clidc.py` script is run. User names are roles are configured in the *System configuration file*.
- `-v`: This option causes the `debug` level output of the query or job submission to be printed to standard output. Default is to print only the `info` level (which is already quite verbose). The verbose logging output can always be read in the log file that is written for every invocation of `clidc.py` in the *logging* directory.
- `-s`: With this option you can run `clidc.py` with an alternative system configuration file. This can be useful for commissioning/research. Default system configuration file is the `cfg/system.cfg` file in the directory where IDEAL is installed.
- `-j`: Number of cores to use per beam. Default: the number of cores specified in the *System configuration file*.
- `-d`: This option is currently a no-op. The intention is that when set, the temporary data generated by the simulations will not be compressed/deleted, so that it remains available for diagnostics and debugging by the user/developer/researcher. Currently, the temporary data are always compressed, unless the postprocessing script crashes before it gets to perform that operation.

Query options (with a uppercase letter: `BCMPRXZ`) These cause `clidc.py` to perform a query instead of a dose calculation. These options **cannot** be combined with the following two groups of options (configuration tweaks and statistics goals).

- `-B`: beam names in a treatment plan
- `-C`: available CT protocols (Hounsfield lookup tables)
- `-M`: available override materials
- `-P`: available phantom models

- `-R`: ROIs in (the structure set that is used by) a treatment plan
- `-X`: default dose resolution (number of voxels in each direction)
- `-Z`: available beamline models (useful for commissioning and research)

Statistical goal options (with a lowercase letter: `tnu`) These cause `clidc.py` to perform a dose calculation and **cannot** be combined with query options.

- `-t`: maximum time to run (excluding queue waiting, pre- and post-processing time) in **minutes**.
- `-n`: minimum number of primary particles to simulate.
- `-u`: average uncertainty to reach (or better) in the voxels that have at least half the maximum dose.

More details below.

Configuration options (with a lowercase letter: `bcmpxz`) These options change the way how `clidc.py` to perform a dose calculation and **cannot** be combined with query options. Note that for each uppercase query option `BCMPXZ` that *queries* a quantity/property there is a corresponding lowercase configuration option `bcmpxz` that *changes* the value/setting of that quantity/property. (The information queried with `R` is used in the `m` material override option, so there is not separate `r` tweak option.)

- `-b`: beam names to simulate (instead of all beams in the treatment plan)
- `-c`: CT protocols (Hounsfield lookup tables) to use
- `-m`: followed by one or more values of the form `ROINAME:MATERIALNAME`, to specify material overrides.
- `-p`: specify phantom to use INSTEAD of the CT
- `-x`: changes the dose resolution (number of voxels in each direction; the origin and physical size remain unchanged).
- `-f`: score dose on full CT resolution, do not resample to the TPS planning dose
- `-z`: use this beamline model for all beams in the plan (only available with commissioning and research roles)

The `-c` and `-p` are mutually exclusive, as are `-x` and `-f`.

Examples of all 7 possible combinations of the three statistical constraints/goals:

```
-n 1000000: stop when >1000000 primaries were simulated.
-u 1.5: stop when average statistical uncertainty in the high dose region is <1.5%.
-t 18000: stop after running for >5 hours.
-n 1000000 -u 5: run >1000000 primaries, and then continue running until <5%␣
↪uncertainty is reached.
-t 18000 -u 4: try to reach <4% uncertainty, but stop if it takes >5 hours.
-t 18000 -n 1000000: try to simulate >1000000 primaries, but stop if it takes␣
↪longer than 5 hours.
-t 18000 -n 1000000 -u 2.0: stop after 5 hours, unless the goals of running >
↪1000000 primaries AND <2% average statistical accuracy have been reached earlier.
```

The constraints are not checked after every individually simulated primary. Rather, they are checked periodically (e.g. every two minutes, this can be configured). This means that the constraints will typically be exceeded by a bit: you may get a couple of thousand primaries more than you asked for, the simulation may keep running for a while after the timeout has expired and/or the average statistical uncertainty may be slightly better than the goal.

Todo: Describe better how the average dose uncertainty is computed. There is some explanation buried in the system configuration file docs, but that is in the install/commissioning part. It should actually be explained in the "usage" part of the manual.

PyQt interface

More details.

Research interface

More details.

1.5 DICOM specification

We will put a table here with a complete specification of all DICOM tags that are used by IDEAL: which ones are used in which way, with what kinds of assumptions.

Unfortunately we did not manage to compile this table before the 1.0 release. We'll try to include it in the docs 1.1 release, which should only change in the docs and in the organization of the code, not in the actual functionality.

1.6 Reference manual

1.6.1 Architectural overview

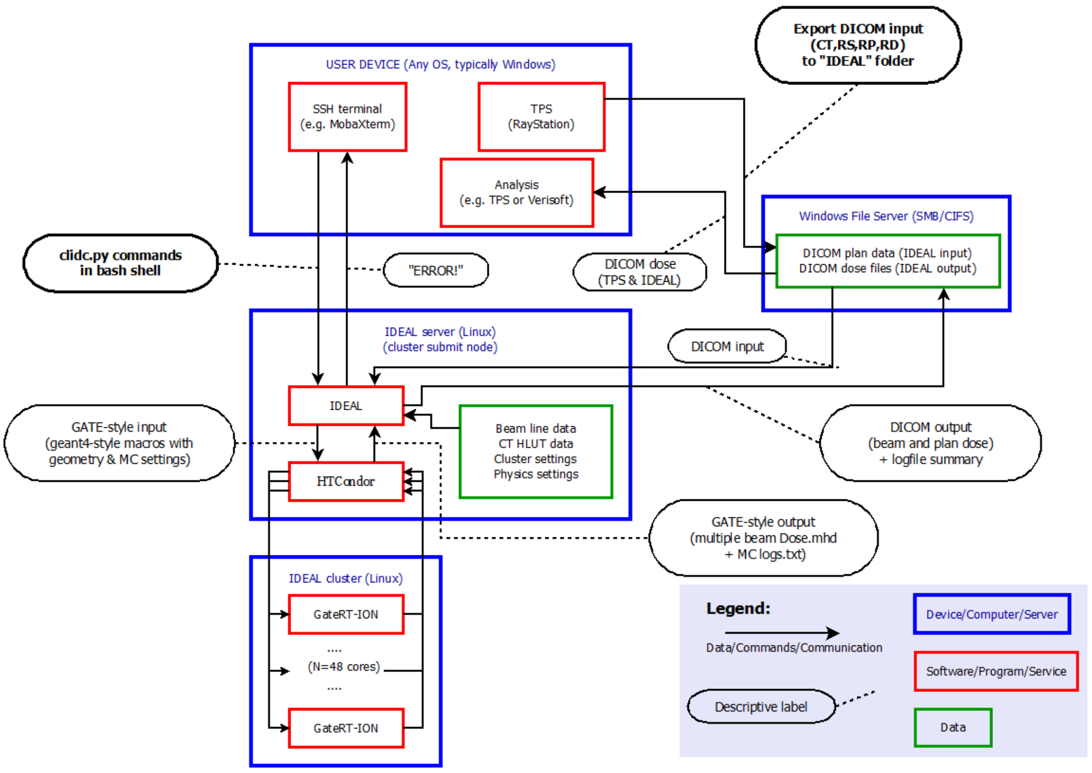
Synopsis

The core functionality of the IDEAL software is to compute the dose distribution for any PBS ion beam treatment plan, using Geant4/GATE as the dose engine. The primary purpose is to provide an independent dose calculation tool that may be useful in clinics, but the software is designed to be useful also for a variety of research topics related to dose calculations.

Since Geant4/GATE simulations are quite CPU time consuming, IDEAL is designed to run on a GNU/Linux cluster in order to achieve the statistical goal within a reasonable time. IDEAL is implemented as a set of python modules and scripts. These convert the information in the input DICOM plan data into a set of data files and scripts that can be used by Gate to simulate the delivery of each of the beams requested in the treatment plan, using beamline model details and CT calibration curves that are configured by the user during installation and commissioning of the IDEAL. The dose distribution for each beam is computed separately up to the statistical goal specified by the user: number of primaries, *average uncertainty* or a fixed calculation time. The dose grid is by default the same as used by the treatment planning system, but the user can choose to override the spatial resolution. The beam dose as well as the plan dose are exported as DICOM files.

Workflow

A device oriented example view of the workflow is illustrated in the figure below.


Export DICOM input (CT,RS,RP,RD) to "IDEAL" folder
USER DEVICE (Any OS, typically Windows)
SSH terminal (e.g. MobaXterm)
TPS (RayStation)
Analysis (e.g. TPS or Verisoft)
Windows File Server (SMB/CIFS)
DICOM plan data (IDEAL input)
DICOM dose files (IDEAL output)
clidc.py commands in bash shell
"ERROR!"
DICOM dose (TPS & IDEAL)
IDEAL server (Linux) (cluster submit node)
DICOM input
IDEAL
Beam line data
CT HLUT data
Cluster settings
Physics settings
GATE-style input (geant4-style macros with geometry & MC settings)
HTCondor
DICOM output (beam and plan dose) + logfile summary
GATE-style output (multiple beam Dose.mhd + MC logs.txt)
IDEAL cluster (Linux)
GateRT-ION
....
(N=48 cores)
....
GateRT-ION
Legend:
Data/Commands/Communication
Descriptive label
Device/Computer/Server
Software/Program/Service
Data

In this view, the user is logged in to the treatment planning workstation.

1. The user exports a treatment plan from the TPS to DICOM files on a shared file system. The exported data include the planning CT, the treatment plan (including the ion beam sequence), the structure set and physical and/or effective dose for the beams and/or the plan, in a folder that is also mounted on the submission node of the cluster.
2. The user logs in to the submission node of the cluster and runs one of the IDEAL user interface scripts (`clidc.py` or `socrates.py`) to start an independent dose calculation. The minimum input information is the treatment plan file and the statistical goal (number of primaries or average uncertainty). Optional overrides of the default settings are discussed in the following sections.
3. IDEAL collects and checks all necessary DICOM data referred to by the treatment plan file: the structure set (including exactly one ROI of type "External"), the planning CT and the TPS dose.
4. IDEAL finds the beamline model(s) corresponding to the treatment machine(s) specified in the plan file, as well as the conversion tables for the CT protocol of the planning CT.
5. A 'work directory' is set up with all scripts specific for the IDC of the treatment plan: * configuration file for *Preprocessing* (CT image cropping, material overrides) * shell scripts, macros and input files for `GateRTion` * configuration file for *Postprocessing* (dose accumulation, resampling and export to DICOM) * submit file for HTCondor (cluster job management system) "DAGman" job.
6. The IDC is started by submitting the condor "DAGman" job. The DAGman graph has just three nodes: * *Preprocessing* on the submit node, typically takes about a minute. * simulation on the calculation nodes (`Nbeams*Ncores` jobs, where `Nbeams` is the number of beams `Ncores` the number of physical cores in the cluster), runtime is typically in the order of hours, depending on many factors. * postprocessing on the submit node, takes a couple of minutes.
7. A "job control daemon" is spawned which will regularly (typically every 300 seconds) check whether the statistical goal (average uncertainty or number of primaries) has been reached for each successive beam. If the goal is reached, then a semaphore file "STOP-<beamname>" is created in the work directory. The scripts that are called by the Gate "StopOnScript" actor check the presence of that semaphore file to decide whether to stop the simulation or to continue.

Preprocessing

The job-dependent details for the preprocessing are computed and saved to a text file by `ideal.impl.idc_details.WritePreProcessingConfigFile()`. The preprocessing is performed by the `bin/preprocess_ct_image.py` script as part of the HTCondor DAGman corresponding to the IDC job.

- Cropping: the minimum bounding box is computed that encloses both the "padded" External ROI volume in the planning CT and the TPS dose distribution. The External ROI is padded with a fixed thickness of air on all six sides. The padding thickness can be configured with an entry for *air box margin [mm]* in the `[simulation]` section of the system configuration file.
- Hounsfield unit values are truncated to the maximum HU value specified in the density curve given for the relevant CT protocol.
- In all voxels whose central point is *not* within the External ROI volume, the material is overridden with air (`G4_AIR`).
- Each override specified by the IDEAL user is applied to all voxels whose central point lies within the ROI to which the override applies. The user should not specify different material overrides for two or more overlapping ROIs, since the order in which the overrides will be applied is random.
- The material overrides are implemented by extending a copy of the interpolated Schneider table corresponding to the relevant CT protocol. In the extension, HU values larger than the maximum HU value in the density curve tables are associated with the override materials, and in the preprocessed CT image those high HU values are used for the voxels that have material overrides.
- A mass file is created with the same geometry as the cropped CT image, with floating point voxel values representing the density (in grams per cubic centimeter). For voxels with material overrides (e.g. outside the external), the density value are taking from the overriding material; for all other voxels the density is obtained by a simple linear interpolation in the density curve.
- A dose mask file is created with the same geometry as the output dose files, with value 1 (0) for all voxels with the central point inside (outside) the External ROI.

Calculation of average uncertainty

The dose distributions of each beam in a beamset (or treatment plan) is computed separately. When computing the dose for one beam, on (almost) all physical cores of the cluster `GateRTion` is simulating protons or ions with kinematics that are randomly sampled from all spots in the beam, with probabilities proportional to the number of planned particles per spot, and Gaussian spreads given by *source properties file* for the beamline (treatment machine) by which the beam is planned to be delivered.

For all particles that are tracked through the TPS dose volume within the (cropped) CT geometry, the physical dose (in water, by convention) is computed by the `DoseActor` in `GateRTion` as the deposited energy divided by mass, multiplied by the stopping power ratio (water to material, where the material is either the Schneider material corresponding to the voxel Hounsfield value or override material, in case the user specified an override for a ROI including the voxel). The dose is saved periodically (by default every 300 seconds, *configurable* in the system configuration file).

The final dose distribution (typically the same as for the TPS) is typically not with the same spatial resolution as the CT. The job control daemon computes an estimate of the Type A ("statistical") dose uncertainty in each (resampled) voxel by resampling the (intermediate) dose distributions of from all condor jobs to the TPS dose grid, dividing the dose by the number of simulated primaries by each core, and computing the weighted average and weighted standard deviation of the dose-per-primary for each voxel. The number of primaries simulated on each core serve as weights. The relative uncertainty in each voxel is the the ratio of the (weighted) standard deviation and the (weighted) average.

A "mean maximum" value of dose-per-primary is estimated by computing the mean of the `Ntop` highest values in the distribution of the weighted average dose per voxel per primary. A threshold value is then defined as a fraction `P` (in percent) of this "mean maximum" dose-per-primary value. The "average uncertainty" is then computed as the simple unweighted average of the relative uncertainties in those voxels in which the dose-per-primary is higher than this threshold.

The values of Ntop *can be set* in the system configuration file (default `Ntop=100` and `P=50` percent).

When the user starts and IDC with an uncertainty value as statistical goal, then the job control daemon with apply this goal to the dose for every beam. The simulations for **each** beam will not be stopped until the above described average uncertainty estimate for the **beam** dose is below the target value. In other engines or treatment planning systems, the average uncertainty goal may refer to to the **plan** dose instead.

Postprocessing

Actions in post processing:

- Accumulate dose distributions and total number of primaries from simulations on all cluster cores.
- Scale the dose with the ratio of the planned and simulated number of primary particles.
- Scale the dose with an overall *correction factor*.
- Resample the dose to the output dose geometry (typically the same as the TPS dose geometry)
- For protons: compute a simple estimate of the "effective" dose by *multiplying with 1.1*.
- Save beam doses in the format(s) *configured* by the user.
- Compute the gamma index value for every voxel in the beam dose with dose above threshold, if gamma index parameters are *configured* and the corresponding TPS beam dose is available. Output only in MHD format (no DICOM).
- Compute plan doses (if plan dose output is *configured* by the user).
- Compute gamma index distributions for plan doses (if TPS plan dose is available and gamma index calculation is enabled in the system configuration).
- Update user log summary text file with settings and performance data.
- Copy final output (beam and plan dose files, user log summary) to another shared folder (typically a CIFS mount of a folder on Windows file share, to be *configured by the user* in the system configuration file).
- Clean up: compress (rather than delete: to allow analysis in case of trouble) the outputs from all `GateRTion` simulations, remove temporary copies. This is usually the most time consuming part of the post processing.

(To be continued.)

1.6.2 Developers' manual

How to hack stuff.

1.6.3 TODO list

- Make IDEAL/pyidc into a "pip" project.
- Support rotating gantry.
- Support range shifter on movable snout.
- The job_executor and idc_details classes are too big, lots of implementation details should be delegated into separate classes. + Specifically, the creation of the Gate work directory and the creation of all condor-related files should be wrapped in separate classes. And then it should be more straightforward to support other cluster job management systems, such as SLURM and OpenPBS.
- There are still many ways in which the user can provide wrong input and then get an error that is uninformative. For instance, using "G4_Water" instead of "G4_WATER" as override material for a ROI results in a KeyError that only says that the "key" is unknown.

1.6.4 Modules

The two central modules in IDEAL are the humbly named `ideal.impl.idc_details` and py:class:*ideal.impl.job_executor*. The `ideal.impl.idc_details` module is responsible for collecting and checking the input data and the user settings. The `ideal.impl.job_executor` is responsible for the execution of all steps of the calculation, which is organized in three phases: pre-processing, simulation and post-processing. Both modules should be used by a script (e.g. the command line user interface script `clidc.py`) that runs on the submission node of the cluster.

IDEAL currently has two scripts that provide user interfaces (UI): one "command line" user interface (`clidc.py`) and one "graphical" user interface (`sokrates.py`). Both these UIs are effectively just differently implemented wrappers around the `idc_details` and `job_executor` modules; with `clidc.py`, the user specifies the input plan file and simulation details via command line options while `sokrates.py`, these inputs are selected with traditional GUI-elements such as drop-down menus, radio buttons, etc. Research users are of course not limited to these UI scripts and can write their own scripts based on the IDEAL modules.

This module implements the "green" box in the "prepare" diagram included in the "device description" of IDEAL, the one that says "get beamline model".

class `impl.beamline_model.`**`Test_GetBeamLineModel`**(*methodName='runTest'*)

`impl.dicom_dose_template.`**`write_dicom_dose_template`**(*rtplan*, *beamnr*, *filename*, *phantom=False*)

Create a template DICOM file for storing a dose distribution corresponding to a given treatment plan. * rtplan: a pydicom Dataset object containing a PBS ion beam plan. * beamnr: a *string* containing the beam number to be used for referral. Should contain "PLAN" for plan dose files. * filename: file path of the desired output DICOM file * phantom: boolean flag to indicate whether this is for a CT (False) or phantom dose (True).

`impl.dual_logging.`**`get_dual_logging`**(*verbose=False*, *quiet=False*, *level=None*, *prefix='logfile'*, *daemon_file=None*)

Configure logging such that all log messages go both to a file and to stdout, filtered with different log levels.

This module defines the function which writes a Gate macro very similar to the one that Alessio composed for his PSQA simulations.

Some commands which were executed by Alessio from an external macro file are now copied here directly, in an effort to completely eliminate the use of aliases. Writing all beam numbers and beamset names explicitly hopefully facilitates later debugging.

`impl.gate_macro.`**`check`**(*ct=True*, ***kwargs*)

Function to check completeness and correctness of a long list of key word arguments for gate macro composition.

`impl.gate_macro.`**`roman_year`**()

Vanity function to print the year in Roman numerals.

`impl.gate_macro.`**`write_gate_macro_file`**(*ct=True*, ***kwargs*)

This function writes the main macro for PSQA calculations for a specific beam in a beamset. beamset: name of the beamset uid: DICOM identifier of the full plan/beamset user: string that identifies the person who is responsible for this calculation spotfile: text file

This module serves to define the available Hounsfield lookup tables (HLUT) as defined in the materials/HLUT/hlut.conf configuration file in the commissioning directory of an IDEAL installation.

In the 'hlut.conf' configuration file, each section represents a supported CT protocol and the data in the section should unambiguously describe for which kinds of CTs the protocol is relevant and how to use it to define the Gate/Geant4 material to use for each HU value in a given CT image with that protocol.

There are two kinds of protocols: Schneider protocols which are intended for both clinical and commissioning purposes.

For the Schneider type protocols the 'hlut.conf' file should specify two text files, providing the "density curve" and the "material composition", respectively. The density curve typically specifies the density for about a dozen HU values between -1024 and +3000. The composition file specifies for several HU intervals a mixture of elements.

Gate/Geant4 has a method for converting these two tables for a given density tolerance value into a "HU to material" table, along with a database file that has the same format as the "Gate material database" file (usually named "GateMaterials.db") which defines lots of "interpolated materials", each with slightly different density, but the composition exactly like one of the materials in the "composition" file.

The conversion of the Schneider density and composition tables into the HU-to-materials table and the associated new database is taken care of by the `gate_hlut_cache` module, which saves the new table and database in a cache directory such that they can be reused.

The protocols can either be directly selected through their name or through a match with the metadata included in the CT DICOM files.

A match with DICOM metadata allows the "automatic detection" of the CT protocol. In `the hlut.conf` file, the commissioning physicist lists one or more DICOM tags (by keyword or by hexadecimal tag) and which text strings the DICOM values are expected to have for that CT protocol. It is the responsibility of the commissioning physicist to define this in such a way that for all data that will be used with this particular IDEAL installation, the automatic selection will always result in one single match. In case multiple DICOM tags are given per CT protocol, the protocol with the most matches "wins".

The user may also provide a name or keyword in the IDEAL command invocation (e.g. the -c option from `clidc.py`) which is then directly matched with the CT protocol names. First a case sensitive perfect match is attempted, if that fails then a case insensitive partial match (the given keyword may be a substring of the full CT protocoll name). In case the partial match results in multiple results, an exception is thrown and the user should try again.

class `impl.hlut_conf.`**`hlut`**(*name*, *prsr_section*, *hutol=None*)

This class serves to provide the HU to material conversion tables, based on the information from the HLUT configuration file. This can be either a "Schneider" type conversion (Schneider tables, interpolated with a density tolerance value to a large list of interpolated materials) or a "commissioning" type conversion in which the HU to materials are directly coded by the commissioning physicist.

`match`(*ct*)

For now, check that all requirements match. :param ct: a DCM data set as read with pydicom.dcmread.

class `impl.hlut_conf.`**`hlut_conf`**(*s={}*)

This is a 'singleton' class, only one HLUT configuration object is supposed to exist. The singleton HLUT configuration object behaves like a Python dictionary with limited functionality: no comparisons (since there is only one such object), no additions or removals, but the elements are not strictly 'const'. The HLUT configuration is initialized the first time its instance is acquired with the static *getInstance()* method.

static getInstance(*fname=None*)

Get a reference to the one and only "HLUT configuration" instance, if it exists. The 'fname' argument should ONLY be used by the unit tests. In normal use, the HLUT conf file should be defined by the system configuration.

`hlut_match_dicom`(*ctslice*)

This method tries to find the HLUT for which the DICOM metadata matching rules uniquely matches user-provided CT file object (a data set returned by pydicom.dcmread, e.g. first CT slice in the series coming with the DICOM treatment plan file).

In case of success, it returns the CT protocol name.

In case of failure, this function will throw a `KeyError` with some explaining text that should help the user understand what is going wrong.

`hlut_match_keyword`(*kw*)

This method compares a user-provided keyword (e.g. from the -c option for the `clidc.py` script) to the names of the CT protocols provided in the hlut.conf file.

In case of success, it returns the matching CT protocol name.

In case of failure, this function will throw a `KeyError` with some explaining text that should help the user understand what is going wrong.

class `impl.hlut_conf.`**`test_good_hlut_conf`**(*methodName='runTest'*)

setUp()
Hook method for setting up the test fixture before exercising it.

tearDown()
Hook method for deconstructing the test fixture after testing it.

This module implements the job executor, which uses a "system configuration" (system configuration object) and a "plan details" objbect to prepare the simulation subjobs, to run them, and to combine the partial dose distributions into a final DICOM dose and to report success or failure.

The job executor is normally created *after* a 'idc_details' object has been created and configured with the all the plan details and user wishes. The job executor then creates the Gate workspace and cluster submit files. The job is not immediately submitted to the cluster: if the user interacts with the "socrates" GUI, then the user can inspect the configuration by running a limited number of primaries and visualizing the result with "Gate –qt". After the OK by the user, the job is then submitted to the cluster and the control is taken over by the "job control daemon", which monitors the number of simulated primaries, the statistical uncertainty and the elapsed time since the start of the simulation and decides when to stop the simulations and accumulate the final results.

impl.system_configuration.**get_sysconfig**(*filepath=None*, *verbose=False*, *debug=False*, *username=None*, *want_logfile='default'*)
This function parses the "system configuration file", checks the contents against trivial mistakes and then creates the `system_configuration` singleton object. :param filepath: file path of the system configuration file, use `dirname(script)/../cfg/system.cfg` by default :param verbose: boolean, write DEBUG level log information to standard output if True, INFO level if False. :param debug: boolean, clean up the bulky temporary data if False, leave it for debugging if True. :param username: who is running this, with what role? :param want_logfile: possible values are empty string (no logfile), absolute file path (implying no logs to stdout) or 'default' (generated log file path, some logs will be streamed to standard output)

class impl.system_configuration.**system_configuration**(*s={}*)
This is a 'singleton' class, only one system configuration object is supposed to exist. The singleton system configuration object behaves like a dictionary, except that you get a (shallow) copy of an object from it, instead of a reference, when you "get" an "item".

The system configuration is initialized once at the start of whatever program is using it, through the function call *get_sysconfig* (see below). After initialization it should then not change anymore.

The implementation aims to avoid changing the system configuration *inadvertently*. This is not banking software.

static getInstance()
Get a reference to the one and only "system configuration" instance, if it exists.

This module was developed in as part of the effort in Uppsala (2015-2017) develop a framework to analyze the effect of patient motion on the dose distribution in proton PBS (primarily at Skandion, but it should work for other clinics as well). Authors: David Boersma and Pierre Granger

class utils.roi_utils.**contour_layer**(*points=None*, *ref=None*, *name='notset'*, *z=None*, *ignore_orientation=True*)
This is an auxiliary class for the *region_of_interest* class defined below. A *contour_layer* object describes the ROI at one particular z-value. It is basically a list of 2D contours. The ones with positive orientation (sum of angles between successive contour segments is +360 degrees) will be used to for inclusion, the ones with negative orientation (sum of angles is -360 degrees) will be used for exclusion. All points of an exclusion contour should be included by an inclusion contour.

utils.roi_utils.**list_roinames**(*ds*)
Return the names of the ROIs in a given dicom structure set as a list of strings.

utils.roi_utils.**list_roinumbers**(*ds*)
Return the names of the ROIs in a given dicom structure set as a list of strings.

utils.roi_utils.**scrutinize_contour**(*points*)
Test that *points* is a (n,3) array suitable for contour definition.

`utils.roi_utils.`**`sum_of_angles`**(*points*, *name='unspecified'*, *rounded=True*, *scrutinize=False*)

Code to determine left/right handedness of contour. We do this by computing the angles between each successive segment in the contour. By "segment" we mean the difference vector between successive contour points. The cross and dot products between two segments are proportional to sine and cosine of the angle between the segments, respectively.

This module provides a function for mass weighted resampling of an image file that contains a 3D dose distribution to match the geometry (origin, spacing, number of voxels per dimension) of a reference image.

The resampled dose R_j in voxel j of the new grid is computed as follows from the dose D_i of voxels i in the input dose distribution, the corresponding masses (or mass densities) M_i and the volumes V_i_j of each pair of input voxel i and output voxel j:

R_j = Sum_i w_i_j D_i / N_j w_i_j = V_i_j * M_i N_j = Sum_i w_i_j

If you choose to run the multithreaded version, then it is your own responsibility to make wise choice for the number of threads, based on (e.g.) the number of physical cores, the available RAM and the current workload on the machine.

`class` `utils.resample_dose.`**`dose_resampling_tests`**(*methodName='runTest'*)

`setUp`()

Hook method for setting up the test fixture before exercising it.

`utils.resample_dose.`**`enclosing_geometry`**(*img1*, *img2*)

Does img1 enclose img2?

This function could be used to test whether img2 can be used as a reference image for resampling img1.

`utils.resample_dose.`**`equal_geometry`**(*img1*, *img2*)

Do img1 and img2 have the same geometry (same voxels)?

This is an auxiliary function for *mass_weighted_resampling*.

`utils.resample_dose.`**`mass_weighted_resampling`**(*dose*, *mass*, *newgrid*)

This function computes a dose distribution using the geometry (origin, size, spacing) of the *newgrid* image, using the energy deposition and mass with some different geometry. A typical use case is that a Gate simulation first computes the dose w.r.t. a patient CT (exporting also the mass image), and then we want to resample this dose distribution to the geometry of the new grid, e.g. from the dose distribution computed by a TPS.

This implementation relies on a bit of numpy magic (np.tensordot, repeatedly). A intuitively more clear but in practice much slower implementation is given by *_mwr_wit_loops(dose,mass,newgrid)*; the unit tests are verifying that these two implementation indeed yield the same result.

`class` `utils.resample_dose.`**`overlaptests`**(*methodName='runTest'*)

This module provides a simple function that converts an ITK image with (short int) HU values into an float32 image with density values.

`utils.mass_image.`**`create_mass_image`**(*ct*, *hlut_path*, *overrides={}*)

This function creates a mass image based on the HU values in a ct image, a Hounsfield-to-density lookup table and (optionally) a dictionary of override densities for specific HU values.

If the HU-to-density lookup table has 2 columns, it is interpreted as a density curve that needs to be interpolated for the intermediate HU values. If the HU-to-density lookup table has 3 columns, then it is interpreted as a step-wise density table, with a constant density within each successive interval (no interpolation).

`class` `utils.mass_image.`**`mass_image_test`**(*methodName='runTest'*)

`class` `utils.gate_pbs_plan_file.`**`gate_pbs_plan`**(*planpath*, *bml=None*, *radtype='proton'*)

It might be more appropriate to call this a 'MCsquared' or 'REGGUI' plan. It can still be used for reading "normal" Gate-style PBS plan text files, which do not contain any information about passive elements.

class `utils.gate_pbs_plan_file.`**`gate_pbs_plan_file`**(*planpath*, *gangle=0.0*, *allow0=False*)

Class to write GATE plan descriptions from arbitrary spot specifications produced e.g. by a TPS such as RayStation.

class `utils.gate_pbs_plan_file.`**`test_gate_pbs_plan_reading`**(*methodName='runTest'*)

`setUp`()

Hook method for setting up the test fixture before exercising it.

`tearDown`()

Hook method for deconstructing the test fixture after testing it.

Compare two 3D images using the gamma index formalism as introduced by Daniel Low (1998).

class `utils.gamma_index.`**`Test_GammaIndex3dIdenticalMesh`**(*methodName='runTest'*)

class `utils.gamma_index.`**`Test_GammaIndex3dUnequalMesh`**(*methodName='runTest'*)

`utils.gamma_index.`**`gamma_index_3d_equal_geometry`**(*imgref*, *imgtarget*, *dta=3.0*, *dd=3.0*, *ddpercent=True*, *threshold=0.0*, *defvalue=-1.0*, *verbose=False*, *threshold_percent=False*)

Compare two images with equal geometry, using the gamma index formalism as introduced by Daniel Low (1998). * ddpercent indicates “dose difference” scale as a relative value, in units percent (the dd value is this percentage of the max dose in the reference image) * ddabs indicates “dose difference” scale as an absolute value * dta indicates distance scale (“distance to agreement”) in millimeter (e.g. 3mm) * threshold indicates minimum dose value (exclusive) for calculating gamma values: target voxels with dose<=threshold are skipped and get assigned gamma=defvalue. * threshold_percent is a flag, True means that threshold is given in percent, False (default) means that the threshold is absolute. Returns an image with the same geometry as the target image. For all target voxels that have d>threshold, a gamma index value is given. For all other voxels the “defvalue” is given. If geometries of the input images are not equal, then a *ValueError* is raised.

`utils.gamma_index.`**`gamma_index_3d_unequal_geometry`**(*imgref*, *imgtarget*, *dta=3.0*, *dd=3.0*, *ddpercent=True*, *threshold=0.0*, *defvalue=-1.0*, *verbose=False*, *threshold_percent=False*)

Compare 3-dimensional arrays with possibly different spacing and different origin, using the gamma index formalism, popular in medical physics. We assume that the meshes are *NOT* rotated w.r.t. each other. * *dd* indicates by default the “dose difference” scale as a relative value, in units percent (the dd value is this percentage of the max dose in the reference image). * If *ddpercent* is False, then dd is taken as an absolute value. * *dta* indicates distance scale (“distance to agreement”) in millimeter (e.g. 3mm) * *threshold* indicates minimum dose value (exclusive) for calculating gamma values * *threshold_percent* is a flag, True means that threshold is given in percent, False (default) means that the threshold is absolute.

Returns an image with the same geometry as the target image. For all target voxels that are in the overlap region with the refernce image and that have d>threshold, a gamma index value is given. For all other voxels the “defvalue” is given.

`utils.gamma_index.`**`get_gamma_index`**(*ref*, *target*, ***kwargs*)

Compare two 3D images using the gamma index formalism as introduced by Daniel Low (1998). The positional arguments ‘ref’ and ‘target’ should behave like ITK image objects. Possible keyword arguments include: * *dd* indicates “dose difference” scale as a relative value, in units of percent (the dd value is this percentage of the max dose in the reference image) * *ddpercent* is a flag, True (default) means that dd is given in percent, False means that dd is absolute. * *dta* indicates distance scale (“distance to agreement”) in millimeter (e.g. 3mm) * *threshold* indicates minimum dose value (exclusive) for calculating gamma values * *verbose* is a flag, True will result in some chatter, False will keep the computation quiet. * *threshold_percent* is a flag, True means that threshold is given in percent, False (default) means that the threshold is absolute.

Returns an image with the same geometry as the target image. For all target voxels in the overlap between ref and target that have d>dmin, a gamma index value is given. For all other voxels the “defvalue” is given.

TODO: allow 2D images, by creating 3D images with a 1-bin Z dimension. Should be very easy. The 3D gamma image computed using these "fake 3D" images can then be collapsed back to a 2D image.

`utils.crop.`**`crop_and_pad_image`**(*input_img*, *from_index*, *to_index*, *hu_value_for_padding*)
We would like to use itk.RegionOfInterestFilter but that filter does recalculate the origin correctly. So now we do this manually, through numpy.

`utils.crop.`**`crop_image`**(*input_img*, *from_index*, *to_index*)
We would like to use itk.RegionOfInterestImageFilter but that filter does not recalculate the origin correctly. So now we do this manually, through numpy.

`class` `utils.crop.`**`test_crop`**(*methodName='runTest'*)

`setUp`()
Hook method for setting up the test fixture before exercising it.

`tearDown`()
Hook method for deconstructing the test fixture after testing it.

`class` `utils.crop.`**`test_crop_and_pad`**(*methodName='runTest'*)

`setUp`()
Hook method for setting up the test fixture before exercising it.

`tearDown`()
Hook method for deconstructing the test fixture after testing it.

`class` `utils.bounding_box.`**`bounding_box`**(***kwargs*)
Define ranges in which things are in 3D space. Maybe this should be more dimensionally flexible, to also handle 2D or N dimensional bounding boxes.

`indices_in_image`(*img*, *rtol=0.0*, *atol=0.001*)
Determine the voxel indices of the bounding box corners in a given image.

In case the corners are within rounding margin (given by *eps*) on a boundary between voxels in the image, then voxels that would only have an infinitesimal intersection with the bounding box are not included. The parameters *atol* and *rtol* are used to determine with *np.isclose* if BB corners are on a voxel boundary. In case the BB corners are outside of the image volume, the indices will be out of the range (negative, of larger than image size).

Returns two numpy arrays of size 3 and dtype int, representing the inclusive/exclusive image indices of the lower/upper corners, respectively.

`class` `utils.bounding_box.`**`test_bounding_box`**(*methodName='runTest'*)

`class` `utils.beamset_info.`**`beamset_info`**(*rpfp*)
This class reads a DICOM 'RT Ion Plan Storage' file and collects related information such as TPS dose files. It does NOT (yet) try to read a reffered structure set and/or CT images. This acts as a wrapper (all DICOM access on the plan file happens here). This has a few advantages over direct DICOM access in the other modules: * we can deal with different "DICOM dialects" here; some TPSs may store their plans in different ways. * if 'private tags' need to be taken into account then we can also do that here. * We can make a similar class, with the same attributes, for a treatment plan stored in a different format, e.g. for research, commissioning or QA purposes.

Then the rest of the code can work with that in the same way.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

References

BIBLIOGRAPHY

[GateGeant4] `Gate` is a Geant4 application for medical imaging and dosimetry studies, see the OpenGATE website.

[GateRTion] Gate/Geant4 release for clinical ion therapy applications, GATE-RTion, see the GATE-RTion website and *Technical Note: GATE-RTion: a GATE/Geant4 release for clinical applications in scanned ion beam therapy* by L. Grevillot et al., PMID: 32422684 DOI: 10.1002/mp.14242.

[HTCondor] *High Throughput Computing with Condor*, see HTCondor home page and the HTCondor docs.

[Schneider2000] Schneider W., Bortfeld T., Schlegel W.: *Correlation between CT numbers and tissue parameters needed for Monte Carlo simulations of clinical dose distributions.* Phys Med Biol. 2000; 45: 459-478.

[Winterhalter2020] Carla Winterhalter et al.: *Evaluation of GATE-RTion (GATE/Geant4) Monte Carlo simulation settings for proton pencil beam scanning quality assurance*, DOI: 10.1002/mp.14481.

[FuchsBeamlineModels2020] Hermann Fuchs et al., *Computer assisted beam modeling for particle therapy* DOI 10.1002/mp.14647

[Python3] Python 3, see the Python main website and the Python docs.

[Ubuntu] Ubuntu Linux.

[VeriSoft] VeriSoft by PTW.

[RayStation] RayStation by RaySearch.

[GPLv3] GNU General Public License v3.0, by the Free Software Foundation.

PYTHON MODULE INDEX

INDEX