GATE 10: A new versatile Python-driven Geant4 application for medical physics

N Krah, L Maigne, M Favaretto, A F Resch, C Trigila, G Mummaneni, E Roncali, M Jacquet, T Baudier, A Pereda, et al.

GATE 10: A new versatile Python-driven Geant4 application for medical physics

N. Krah[1], L. Maigne[2], M. Favaretto[3], A.F. Resch[3], C. Trigila[4], G. Mummaneni[4], E. Roncali[4], M. Jacquet[1], T. Baudier[1], A. Pereda[2], H.Fuchs[5], A.Coussat[6], and D. Sarrut[1]

[1]CREATIS, CNRS, Centre Léon Bérard, INSA Lyon, France
[2]LPC, CNRS, University of Clermont Auvergne, France
[3]MedAustron Ion Therapy Centre, Wiener Neustadt, Austria
[4]University of California at Davis, USA
[5]Medical, University of Vienna, Vienna, Austria
[6]Jagiellonian University in Kraków, Poland

Abstract In this contribution, we present GATE 10, the new version of the long-standing Geant4 application GATE, dedicated to medical physics, e.g. radiation imaging and radiation therapy systems. It features a Python-based user interface and integration of artificial intelligence methods. Moreover, being a Python package, GATE 10 can be easily integrated into external software, e.g. as a Monte Carlo dose engine. The user interface is easy to use, yet provides access to the rich functionality of Geant4 and preconfigured components of GATE (e.g. actors, sources).
GATE 10 is currently under rapid development and a first non-beta release is planned for May 2024. The software is open-source and available via the GitHub platform. Installation is performed via the Python package manager (`pip`). In this contribution, we present an overview of GATE 10 and a selection of new features and improvements.

1 Introduction

Since the beginning of GATE almost 20 years ago, users configure a simulation via macro commands grouped in text files [1, 2] similar to those in Geant4 [3]. Since then, the rise of high-level programming languages such as Python and new machine learning approaches have revolutionized scientific computing. The new version GATE 10 features a Python-based user interface and integration of artificial intelligence methods. Moreover, GATE 10 can be easily integrated as Python package into external software, e.g. as a Monte Carlo dose engine.

We have designed and implemented GATE 10 completely from scratch building onto the legacy of previous versions. On the one hand, it retains most of the features contributed to the code over time. This includes actors as well as preconfigured particle sources. Actors are collections of user actions to hook into Geant4's particle tracking to extract physics quantities, e.g. particle distributions, energy deposit, etc. GATE's particles sources are preconfigured primary particle generators including spatially distributed sources inside a patient for the use in nuclear medicine and imaging. On top of that, GATE 10 includes numerous new features and improvements, of which we will present a selection in this contribution.

2 Design principle of GATE 10

We have been developing GATE 10 with the following goals in mind:

1. Ease of use: The user should be able to concentrate on configuring the simulation rather than worrying about technical aspects of the software.
2. The Python script through which the simulation is set up should appear like an input file with the advantage of having access to all the computational tools available in Python and Python libraries (e.g. `numpy` [4]).
3. Give the user access to the rich Geant4-functionality, with a focus on applications in medical physics and medical imaging.
4. Provide an inner code structure with interfaces within which contributors can implement new features.
5. Keep a reasonable resemblance to the logic of older versions of GATE, i.e. version 9 and older.
6. Support a multitude of platforms including Linux, MacOS, and Windows.
7. Provide functionality to facilitate the logistics of running GATE simulations.
8. Do not compromise the simulation speed or even improve simulation speed compared to older versions of GATE.

3 Code structure

GATE 10 is composed of a part implemented in C++ and a part implemented in Python. The C++ library includes functionality that has a relevant impact on computational speed, i.e. mainly functions frequently called during particle tracking, e.g. after each step or at the beginning of each new track. This mainly concerns scoring (actors) and primary particle generation (sources). By implementing this part of GATE 10 in C++, Geant4 does not need to execute callbacks into Python code which would be associated with a computational overhead. Together with Geant4 classes and

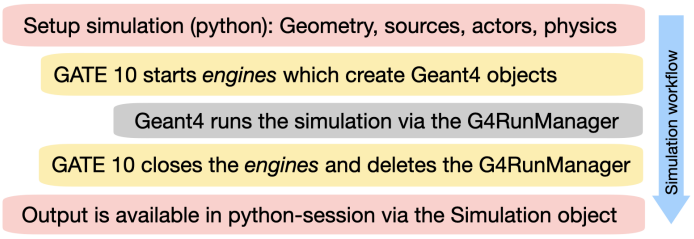

Figure 1: Workflow of a GATE 10 simulation from set-up to simulation run.

functions, the C++-portion of GATE 10 is exposed to Python via *pybind11* [5].

The pure Python part of GATE 10 provides a user interface to build simulations and an internal interface to Geant4. In particular, GATE 10 has two distinct kinds of classes which handle a simulation. *Managers* provide an interface to the user to set-up and configure a simulation and collect and organize user parameters in a structured way. *Engines* provide the interface to Geant4 and are responsible for creating all Geant4 objects. Managers and engines are divided into sub-managers and sub-engines responsible for certain logical parts of a simulation, such as geometry, physics, primary particle generation (sources), and scoring (actions, actors). Additionally, many components in GATE are implemented as classes which provide interfaces to the managers and engines. Figure 1 shows a scheme of the simulation workflow. The user sets up a simulation starting with the `Simulation` class, the main manager. It collects general parameters, e.g. about verbosity and visualization, and it manages the way the simulation is run (e.g., in a subprocess or not). Sub-managers are: `VolumeManager`, `PhysicsManager`, `ActorManager`, `SourceManager`. These managers can be thought of as bookkeepers. For example, the VolumeManager keeps a dictionary of all the volumes added to a simulation, a dictionary of all the parallel world volumes, etc. Furthermore, it also provides the user with methods to perform certain tasks, e.g. `VolumeManager.add_parallel_world()`.

The `SimulationEngine` is the main driver of the Geant4 simulation. Given a simulation object `sim`, it creates a new `SimulationEngine` every time the user calls `sim.run()`, which in turn creates all the sub-engines: `VolumeEngine`, `PhysicsEngine`, `SourceEngine`, `ActorEngine`, `ActionEngine`. The method `SimulationEngine.run_engine()` actually triggers the construction and run of the Geant4 simulation. It takes the role of the `main.cc` file in a pure Geant4 simulation. When the Geant4 simulation has terminated, `SimulationEngine.run_engine()` returns the simulation output which is then collected by the `Simulation` object.

GATE classes representing a Geant4 object (or multiple Geant4 objects combined as, e.g., in the Volume classes) are designed to do multiple things:

- Store user parameters. Example: the `Region` class holds the user parameters `user_limits`, `production_cuts`, and `em_switches`.
- Provide interface functions to the user and to manager classes to configure the object or inquire about it. Examples: `Region.associate_volume()`, `Region.need_step_limiter()`
- Provide interface functions to the engines to handle the Geant4 objects, such as `initialize()` and `close()`.
- Provide convenience functionality such as yielding the object's configuration as a dictionary (`to_dictionary()`), reloading it from a dictionary (`from_dictionary()`), dumping info about the object (e.g. `Region.dump_production_cuts()`), and copying the configuration from another object (`volume2.copy_user_info(volume1)`).
- Handle technical aspects such as serialization via pickling (e.g. for subprocesses dispatching) in a unified way.

4 Example: Dose calculation

In this section, we present a simple example to illustrate the use of GATE 10. The simulation consists of a patient parametrised via a CT image and a conversion table from CT numbers (in Hounsfield units) to material, a proton beam source, and a dose actor which scores dose on a voxel grid attached to the patient geometry. Secondary electrons are only produced inside the patient if their range is greater than 3 mm. The option `start_new_process=True` lets GATE 10 run the Geant4 simulation in a separate process. In this way, the simulation can be run multiple times in a loop, each time with a different beam energy. Dispatching to a subprocess also allows the user to run the same simulation multiple times in an interactive Python session, such as *ipython* or *jupyter*.

```
import opengate as gate
sim = gate.Simulation()
mm = gate.g4_units.mm
MeV = gate.g4_units.MeV
patient = sim.add_volume("Image", name="patient")
patient.image = "patient-4mm.mhd"
patient.mother = "world"
patient.voxel_materials = [
    [-2000, -900, "G4_AIR"],
    [-900, -100, "Lung"],
    [-100, 0, "G4_ADIPOSE_TISSUE_ICRP"],
    [0, 300, "G4_TISSUE_SOFT_ICRP"],
    [300, 800, "G4_B-100_BONE"],
    [800, 6000, "G4_BONE_COMPACT_ICRU"]]
patient.set_production_cut(
    particle_name="electron",
    value=3 * mm)
src = sim.add_source("GenericSource", name="p_src")
src.particle = "proton"
src.position.type = "sphere"
src.position.radius = 10 * mm
src.position.translation = [0, 0, -140 * mm]
src.n = 1e4 # number of primaries
src.direction.type = "momentum"
src.direction.momentum = [0, 0, 1]
dose = sim.add_actor("DoseActor", "dose")
```

```
dose.mother = "patient"
dose.size = [99, 99, 99]
dose.spacing = [2 * mm, 2 * mm, 2 * mm]
dose.img_coord_system = True
dose.translation = [2 * mm, 3 * mm, -2 * mm]
for en in [130, 140, 150, 160]:
    src.energy.mono = en * MeV
    dose.output = f"patient_dose_energy_{en}.mhd"
    sim.run(start_new_process=True)
```

5 Memory management between Python and Geant4

In this section, we describe a rather technical detail about the combination of Geant4 and Python which is, however, crucial to make simulations work correctly. GATE 10 creates certain kinds of Geant4 objects within code on the Python-side while other Geant4 objects are created on the C++-side, in particular by Geant4 itself. To avoid memory leakage, Geant4 objects need to be deleted at the end of a simulation, either by the C++-side, usually meaning Geant4, or by the garbage collector on the Python-side. This requires some understanding of how lifetime is managed in Geant4. GATE 10 uses the `G4RunManager` to initialize, run, and deconstruct a simulation. The `G4RunManager`'s destructor triggers a nested sequence of calls to the destructors of many objects handled by the run manager, e.g. geometry and physics. Volumes, for example, are created through code on the Python-side, but they should never be deleted on the Python-side because the `G4RunManager` is responsible for deletion. Segmentation faults occur if the objects no longer exist at the time of supposed destruction triggered by the `G4RunManager`. This is achieved by correct configuration of the wrapping in *pybind11*.
There is also an important aspect on the Python-side concerning Geant4 objects that are deleted by the `G4RunManager`: Python usually stores a reference to the Geant4 objects whose creation is implemented on the Python-side. This reference will not find the object anymore once the `G4RunManager` has deleted it upon its own destruction. Therefore, we have implemented a mechanism that makes sure that references to Geant4 objects are unset before the `G4RunManager` is garbage collected (and thus its destructor is called).
GATE 10 is now very robust in terms of memory handling and segmentation faults.

6 Novel features

GATE 10 includes several novel features compared to previous versions. We briefly describe a selection of them in this section.

6.1 Native Geant4 multithreading

As a new feature in GATE 10, native Geant4 multithreading capability is available (for most of the functionalty so far). The user simply sets the number of threads, e.g. `sim.number_of_threads=4` and the `SimulationEngine` automatically sets up the `G4MTRunManager` appropriately.

6.2 Export to structured file, e.g. JSON

A simulation can be exported as a structured plain-text file in JSON format. This JSON file can be reloaded at a later stage to re-create the simulation. The user can choose, via an option, whether such as JSON file is automatically created and stored alongside the simulation output:

```
import opengate as gate
sim = gate.Simulation()
sim.store_json_archive = True
sim.store_input_files = True
sim.json_archive_filename = "simu_iccr.json"
sim.output_dir = "output"
...
sim.run()
```

The above example creates a JSON file in a folder called *output*, and alongside with it also stores a copy of all potential input files (images, material descriptions, etc.). The JSON file is structured, human-readable, and compact in size. The simulation can be re-run later by:

```
import opengate as gate
sim = gate.Simulation()
sim.from_json_file("output/simu_iccr.json")
sim.run()
```

This feature helps a lot in data analysis and is particularly interesting in applications where simulation output needs to be archived, such as clinical dose calculation. It can also be helpful for dispatching simulations in a cluster environment and for debugging. The functionality could be easily extended to other file formats such as XML.

6.3 Generative Adversarial Networks for phase-space generation

GATE 10 gives the user the possibility to generate particles via a generative adversarial network (GAN) instead of using, e.g., a conventional phase space file. The idea behind this approach is that particles are not tracked from the sources on, but generated at a later stage, thereby skipping part of the simulation geometry [6–8]. The GAN is also more convenient to store than typically huge phase space files. GATE 10 relies on the `torch` library and provides functionalities to train and apply the neural network.

6.4 User-friendly repeated volumes

Many simulations in medical imaging require geometries which contain many repetitions of the same kind of volume in space. For example, a detector head could consist of a grid of crystal segments where all segments share the same shape, size, and physical properties. Geant4 allows volumes to be

present in multiple locations in space by associating multiple G4PhysicalVolumes to the same G4LogicalVolume. Previous versions of GATE already allowed the user to repeat volumes, but GATE 10 now exposes this feature in a simple manner to the user, as can be seen in the following example snippet:

```
crystal = sim.add_volume("Box", "crystal")
crystal.size = [1 * cm, 1 * cm, 1 * cm]
crystal.translation = [
    [1 * cm, 0 * cm, 0],
    [0.2 * cm, 2 * cm, 0],
    [-0.2 * cm, 4 * cm, 0],
    [0, 6 * cm, 0]]
crystal.material = "LYSO"
```

Instead of setting translation to a single vector to define the placement of the crystal volume, we provide a list of translation vectors. GATE 10 automatically creates the correct set of G4PhysicalVolumes, thereby effectively replicating the volume 4 times in space. Similarly, a list of rotation matrices can be set instead of a single matrix, to inform GATE 10 how each copy of the volume should be rotated.
Multiple utility functions are available to automatically generated sets of translation vectors and/or rotation matrices for common geometric dispositions, such as cartesian grids and circular arrangements, e.g. for PET simulation.

6.5 Voxelized representation of the geometry

The user can generate a voxelized representation of the simulation geometry or a portion thereof. The output is a label image and a lookup table where each voxel contains a label which corresponds to a volume name in the lookup table. Slices of such an image can be easily visualised with common Python libraries such as matplotlib [9]. Such an image helps checking and verifying a geometry or can be used in post-processing simulation data. The user can specify the spatial binning and the extent of the geometry to be voxelized. For the latter, it is possible to provide a list of volumes and GATE 10 will automatically determine the suitable extent which encompasses all volumes.

7 Discussion

GATE 10 is currently under fast development, with new features being added at least at a monthly frequency. The current version as of writing this is beta7, which is stable, fully usable, and which includes more than 80% of functionality offered by previous GATE versions. A first non-beta release is planned for May 2024 and will be presented at the ICCR conference.
Until then, we will refactor the actors and sources to make the code more efficient and the user experience more intuitive. We are also working on improving computation speed of the dose actor (and derived variants) in multithreading mode. Special physics such as optical surfaces are currently being implemented as well as tesselated volumes. Several institutions are already using GATE 10 for their software projects. For example, it is currently being integrated into the IDEAL 1.2 platform for independent dose calculation in light ion beam therapy [10].
Part of our development strategy is that every feature in GATE 10 must have a test case to ensure code integrity over time. Currently, more than 140 test simulations are run automatically each time the code is modified. These tests also serve as examples for users.

8 Conclusion

GATE 10 is open-source and available via the GitHub platform [11]. Installation is performed via the Python package manager (pip), which allows for a simple installation process with one single command. All major platforms, i.e. Linux, MacOS, and Windows are supported. The user is not required to manually install Geant4 as it ships pre-compiled with GATE 10. GATE 10 can be easily used by other software as an external library.

References

[1] S. Jan, D. Benoit, E. Becheva, et al. "GATE V6: A major enhancement of the GATE simulation platform enabling modelling of CT and radiotherapy". *Physics in Medicine and Biology* 56.4 (2011), pp. 881–901. DOI: 10.1088/0031-9155/56/4/001.

[2] D. Sarrut, M. Bardiès, N. Boussion, et al. "A review of the use and potential of the GATE Monte Carlo simulation code for radiation therapy and dosimetry applications". *Medical Physics* 41.6Part1 (May 2014), p. 064301. DOI: 10.1118/1.4871617.

[3] S. Agostinelli, J. Allison, K. Amako, et al. "GEANT4 - A simulation toolkit". *Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 506.3 (2003), pp. 250–303. DOI: 10.1016/S0168-9002(03)01368-8.

[4] *Numpy python package*. URL: https://numpy.org.

[5] *Pybind11*. URL: https://github.com/pybind/pybind11.

[6] D. Sarrut, N. Krah, and J. M. Létang. "Generative adversarial networks (GAN) for compact beam source modelling in Monte Carlo simulations". *Physics in Medicine & Biology* 64.21 (Oct. 2019), p. 215004. DOI: 10.1088/1361-6560/ab3fc1.

[7] D. Sarrut, A. Etxebeste, N. Krah, et al. "Modeling complex particles phase space with GAN for Monte Carlo SPECT simulations: a proof of concept". *Physics in Medicine & Biology* 66.5 (Mar. 2021), p. 055014. DOI: 10.1088/1361-6560/abde9a.

[8] A. Saporta, A. Etxebeste, T. Kaprelian, et al. "Modeling families of particle distributions with conditional GAN for Monte Carlo SPECT simulations". *Physics in Medicine & Biology* 67.23 (Dec. 2022), p. 234001. DOI: 10.1088/1361-6560/aca068.

[9] *Matplotlib python package*. URL: https://matplotlib.org.

[10] L. Grevillot, D. J. Boersma, H. Fuchs, et al. "The GATE-RTion/IDEAL Independent Dose Calculation System for Light Ion Beam Therapy". *Frontiers in Physics* 9 (Aug. 2021). DOI: 10.3389/fphy.2021.704760.

[11] *OpenGATE github repository*. URL: https://github.com/OpenGATE/opengate.