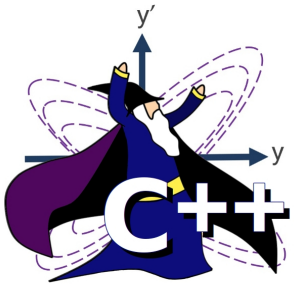
Merlin++
A Quick Start Guide



R. Barlow, S. Rowan, S. Tygier
on behalf of the
Merlin Collaboration

Updated: March 2019

Welcome to Merlin++!

Merlin++ is a particle accelerator simulation software package. Its functionality is similar to that of Methodical Accelerator Design (MAD), but with many additional features. Merlin++ is written in C++ and comprises a library of flexible and modular C++ classes, taking full advantage of the benefits of object orientated techniques and methodologies in the process. To run Merlin++, a user therefore constructs their own script or simulation program and compiles it against the class library.

The following document provides information on how to get up and running with Merlin++, going over the basics as well as walking-through a few choice use-case examples.

Contents

Chapter 1

A Brief History

Merlin++, formerly just Merlin, is a C++ accelerator simulation and particle tracking software package, originally developed by Nick Walker *et al.* at DESY/LBNL in the late 1990's for International Linear Collider (ILC) simulations, *i.e.* electron linac beamlines. In the mid 2000's, Merlin was further adapted to allow for proton tracking as well as circular accelerator simulations and has been used in High Luminosity Large Hadron Collider (HL-LHC) studies ever since. In recent years, Merlin++ has been optimized for high-throughput tracking and loss simulations through the addition of conventional and Hollow-Electron Lens (HEL) collimation capabilities as well as the appropriate pomeron scattering physics processes. Furthermore, recent developments have also focused on updating the code base to adhere to modern open source and software sustainability practices. In this regard, we find Merlin++ to be one of the most powerful, versatile and accessible software packages available in particle accelerator design.

Chapter 2

Getting Started

2.1 Information and Support

Information on Merlin++, including general documentation, class library and developers guide can be found either in the 'MerlinDocumentation' folder included with the software package or via our website:

url: http://www.accelerators.manchester.ac.uk/merlin/

Support for both users and developers can be sought by contacting any of the current developers through github at:

url: https://github.com/Merlin-Collaboration

2.2 Downloading and Installation

Prior to downloading Merlin++, it is important that you have all the required prerequisites.

Below is a step-by-step guide to properly downloading, configuring and installing all prerequisites as well as Merlin++ itself. The following only provides install command examples specific to Ubuntu, for other linux distros, please use as close to the equivalent commands as possible.

NOTE: If necessary, Merlin++ can be downloaded and run without the following prerequisites. However, it is advised that user/developer installs Merlin+ in accordance with the following to prevent compatibility issues.

Prerequisites

- Update system

```
sudo apt install update
```

- C++ compiler, such as g++

```
sudo apt install build-essential
```

- CMake, including CCMake

```
sudo apt install cmake cmake-curses-gui
```

- Python, including numpy & scipy

```
sudo apt install python python-numpy python-scipy
```

- Git version control system

```
sudo apt install git
```

- Java Runtime Environment (Only required for Eclipse CDT IDE)

```
sudo apt install openjdk-8-jre
```

- OPTIONAL: Doxygen

```
sudo apt install doxygen graphviz -y
```

- OPTIONAL: ROOT

```
Download from url: root.cern.ch
```

Downloading and Installing

For the purposes of this quick start guide, we only discuss how to download and install Merlin++ for a Linux OS via git.

NOTE: It is, otherwise, also possible to download the Merlin++ source code directly via either our online public github repository:

url: https://github.com/Merlin-Collaboration/Merlin

- Firstly, call your desired directory, we will assume home as default

```
cd ~
```

- Subsequently, create and enter a suitable git repository directory

```
mkdir git
cd git
```

- Initialize this directory as an empty git repository

```
git init
```

- Download the latest release of Merlin++ from our public github repository (check github link for most recent release version)

```
git clone -b Release-X.XX \
https://github.com/MERLIN-Collaboration/Merlin ./Merlin
```

- Now that you have a clone of the source code, you are ready to install and run Merlin++! First, enter the cloned directory

```
cd Merlin
```

- Create and enter a suitable build directory
  NOTE: Merlin++ **does not** allow in-source builds

```
mkdir build
cd build
```

- Create the release version makefile with the cmake command
  NOTE: Interactive configuration is available using ccmake instead

```
cmake -DCMAKE\_BUILD\_TYPE=Release ..
// OR
ccmake ..
```

- Start building Merlin++!

```
make -jN // where N is the number of available CPUs, e.g. -j8
```

- To confirm the installation has been successful, run the make test suite

```
make test
```

- Pending successful installation you should see something similar to the following (more tests may have been developed):

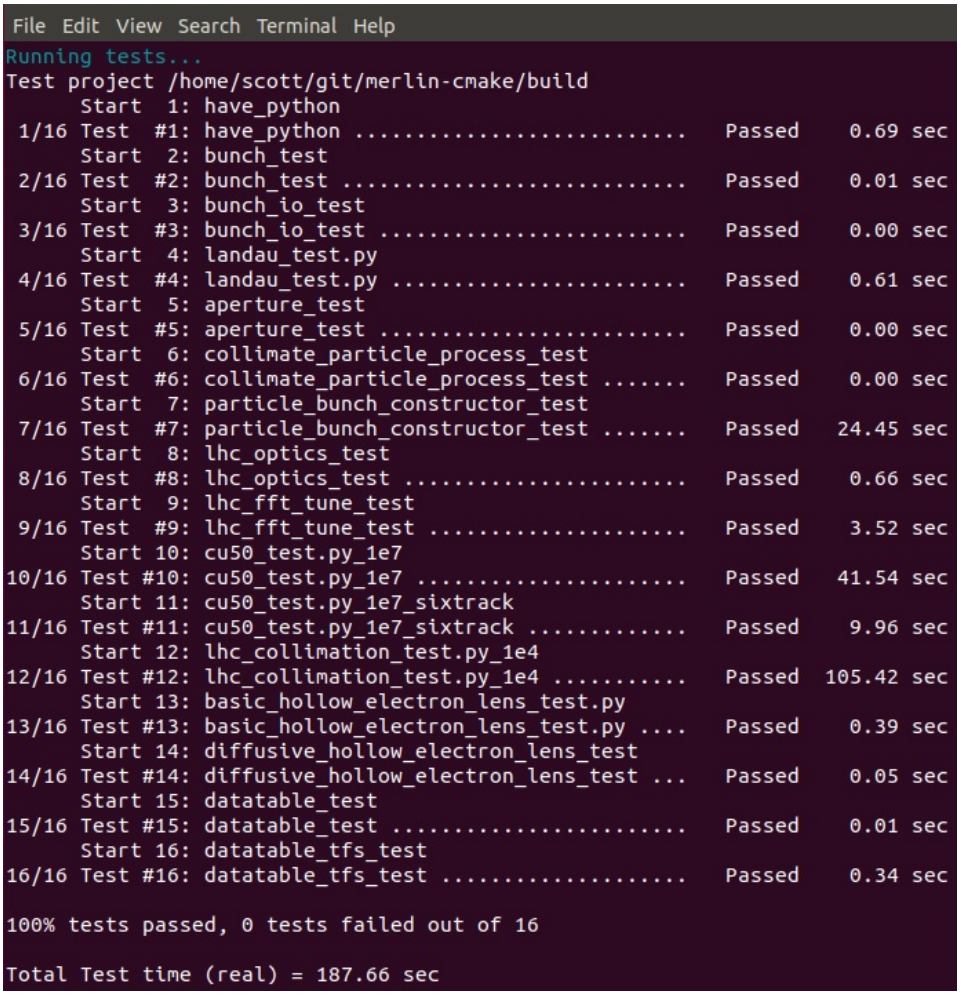


Figure 1: Successful installation test results.

2.3 Doxygen Class Documentation

Doxygen is a common tool for generating class documentation from annotated C++ source. Merlin++ has descriptive class comments in the header files of all major classes formatted such that they are picked-up by doxygen. To generate the class library for Merlin++, simply enter the following command following successful installation. If using Eclipse CDT, use Build Targets in a similar manner to how 'make test' is run.

- Generate class documentation via doxygen

```
make doxygen
```

The above command generates a 'Doxygen' folder and a full interactive .html site in the build directory. To load, simply open the index.html file.

Within the html site, one can find class and class member function descriptions as well as inheritence information, namespace list and information and file structure. Furthermore, developers have begun using this location to document information on any recent API changes. A screenshot of the interactive html class library is shown.

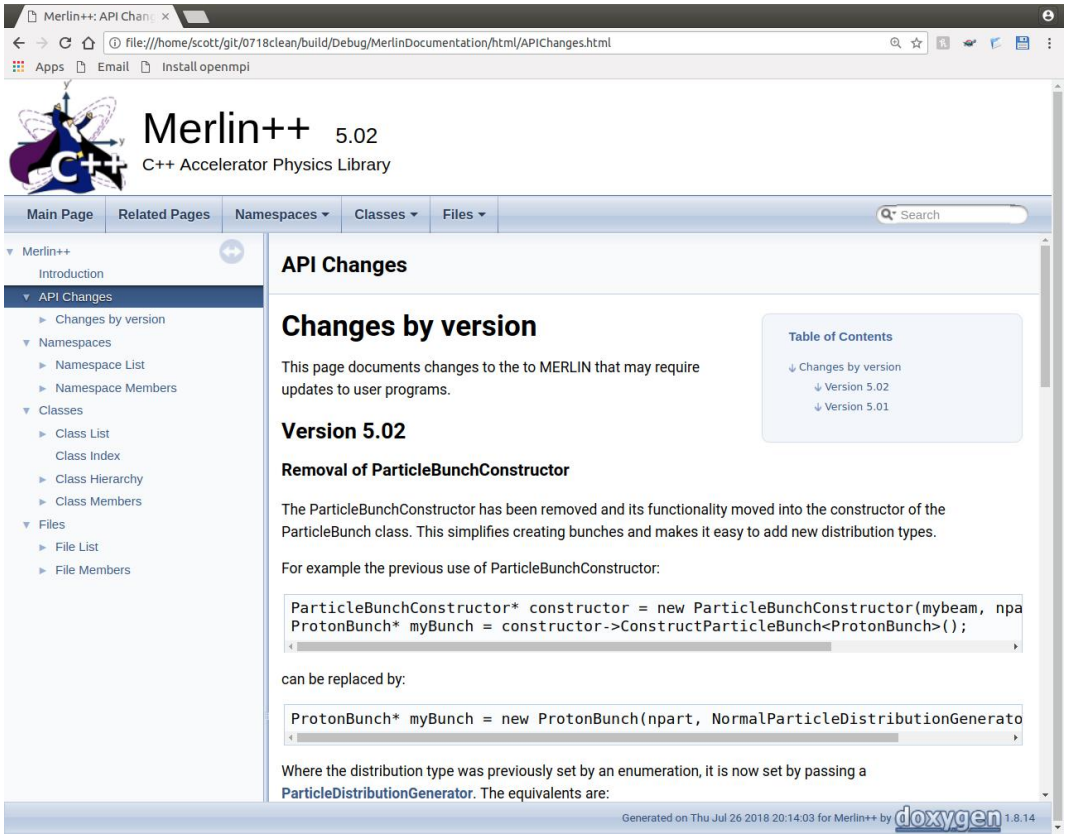


Figure 2: Interactive html Merlin++ class library procduced by doxygen.

2.4 Running Merlin++ in Eclipse CDT

As many developers and script writers alike prefer using an IDE, the following details how to get Merlin++ up and running within the Eclipse CDT IDE. If you are happy running Merlin++ from a command window, feel free to skip this section.

NOTE: The following assumes you have followed through the guide thus far, *i.e.* have installed all prerequisites and have configured Merlin++ successfully.

- Download the latest .tar.gz Eclipse IDE for C/C++ Developers

```
https://www.eclipse.org/downloads/packages/
```

- Uncompress then remove the .tar.gz file

```
sudo tar -xvf eclipse.cpp...tar.gz
sudo rm eclipse.cpp...tar.gz
```

- Move the eclipse file to your prefered install location, e.g. /opt/

```
sudo mv -r ./eclipse /opt
```

- Configure the eclipse launch command

```
sudo cd /usr/local/bin
sudo ln -s /opt/eclipse/eclipse
```

- Launch eclipse and provide an appropriate workspace name

```
eclipse
```

- Install cmake4eclipse plugin through the Eclipse Marketplace

```
> Help > Eclipse Marketplace
- search for and install cmake4eclipse
- agree to licence and select 'Install Anyway' if an error appears
```

- Configure Eclipse Console

```
- While on 'Console' tab, click on the drop down menu at far right
    and select '1 C\C++ Build Console' to display all make outputs
```

- Configure the Merlin++ Project
  NOTE: At this stage it is advised that you delete any build directory created thus far (assuming you followed the command line instructions).

```
Within Eclipse CDT:

> File > New > C/C++ project
- Select 'All' > 'C++ Managed Build'
- Enter a suitable Project name, such as 'Merlin++'
- Uncheck 'Use default location' and enter Merlin++ source directory
    > /home/username/git/Merlin
- Select 'Executable > Empty Project' and 'Linux GCC'
```

```
- Click ‘Finish’
- Your should now have a project ‘Merlin [Merlin Release-X.XX] within
    the ‘Project Explorer’ window’
- Right-click project, select Properties
IMPORTANT: For all of the following you must do for both Debug and
    Release configurations seperately (drop down menu at the top) -
    the ‘All configurations’ option is not supported by Merlin++.
- Go to ‘C/C++ Build’ and click on the ‘Behaviour’ tab. Check ‘Enable
    parallel build’
- Go to ‘C/C++ Build > CMake’ and click on the ‘General’ tab. Check
    ‘Force cmake to run with each build’ under ‘Build Behavior’
- Go to ‘C/C++ Build > Tool Chain Editor’ and set ‘Current builder:’
    to ‘CMake Builder (portable)’ via the drop down menu
- Click ‘Apply’
- Once you have done each of the above for both Debug and Release
    Configurations, click ‘Apply and Close’
- Build Merlin++ using the hammer icon on the main workspace toolbar
```

- Confirm configuration with the Merlin++ test suite within Eclipse CDT

```
- Go back to the ‘Project Explorer’ window
- Within the newly created ‘build’ directory, right-click ‘Debug’ or
    ‘Release’ folders and select ‘Build Targets > Create...’
- Enter the ‘Target Name:’ ‘test’ and click OK
- A ‘Build Targets’ directory should have appeared within your build
    sub-directory
- To launch the test suite simply double-click ‘Build Targets > test’
NOTE: This is same as running ‘make test’ from the command line.
    Similarly, to run ‘make doxygen’ or equivalent, simply create a
    new build target with the corresponding name. Eclipse can provide
    some additional benefits in this regard as it provides a built-in
    browser, useful for navigating the Merlin++ doxygen class library
- If configuration was successful, all tests should pass as before
```

- Launching Merlin++ UserSim or other individual applications

```
- With the ‘Project Explorer’, navigate to ‘build > Debug/Release >
    UserSim > UserSim
- Right-click and select ‘Run As > (2) Local C++ Application’
- You should now be greeted with the Merlin++ welcome message
```

- Congratulations! Merlin++ is successfully configured in Eclipse CDT and you are now ready to start constructing your own user script.

Chapter 3

Code Base & Design Philosophy

3.1 Merlin++ Design Philosophy

Merlin++ is a result of the requirement for object-orientated design (OOD) methodologies in particle accelerator design software. This is mostly due to the extensive lifetime accelerator design software packages tend to have (decades), giving rise to a need for code maintainability and sustainability. C++ was chosen as it excels in this regard, while also allowing for low-level processes to written where necessary to maintain optimal performance. There are two key design philosophies behind Merlin++ (not mutually exclusive):

1. To be able to do the job required in the simplest, but most flexible form
2. To produce a set of loosely coupled software components which are easily maintainable, extendible, and re-usable

The Merlin++ developers encourage that all users and future developers adhere to this design philosophy, too.

3.2 Understand the Design Fundamentals

Merlin++ is a C++ class library for performing charged particle tracking and particle accelerator simulations. The library can be loosely divided into two parts:

1. Accelerator Model construction
2. Beam dynamics and tracking simulation

The model of the accelerator system being studied contains only a description of the physical components *e.g.* magnets, diagnostics, vacuum chambers,

cavities, power supplies, support structures. The classes responsible for performing the required beam dynamics simulations use this information to construct the more abstract algorithm related representation of the model (e.g. construction of a map for particle tracking.) In this respect, the accelerator model can be thought of as a form of database.

This separation allows (in principle) the same accelerator model to be used repeatedly for different forms of beam dynamics simulations, without the need to modify the accelerator model code directly. Currently, the only beam dynamics package supported is particle tracking (*i.e.* ray tracing.)

3.2.1 Accelerator Model

The accelerator model (class AcceleratorModel) is constructed from model elements (class ModelElement). All accelerator components types have a common base class, AcceleratorComponent, and have specific list of associated attributes, see Table 1. The majority of these model elements represent the more tradition beamline components found in other optics codes, see Table 2.

Table 1: Accelerator component attributes and their corresponding classes.

Component Attribute	Merlin++ class
Aperture	class Aperture
Electro-Magnetic Field	class EMField
Geometry	class AcceleratorGeometry

Table 2: Accelerator component types and their corresponding classes.

Component Type	Merlin++ class
Drift Section	class Drift
Sector Bend	class SectorBend
Quadrupole	class Quadrupole
Sextupole	class Sextupole
Octupole	class Octupole
Skew-Quadrupole	class SkewQuadrupole
Skew-Sextupole	class SkewSextupole
Standing-wave RF Cavity	class SWRFStructure
Traveling-wave RF Cavity	class TWRFStructure
Horizontal Correctors (xcor)	class XCor
Vertical Correctors (ycor)	class YCor
Beam Position Monitors (BPM)	class BPM
Profile Monitors	class RMSProfileMonitor

An Aperture defines the physical aperture of the component. An EMField represents the electro-magnetic field of a component; it can be time-dependent. Both the Aperture and EMField are defined in the local coordinate frame of the component, which is a property of its AcceleratorGeometry. In Merlin, an AcceleratorGeometry is an important concept and serves several functions. Primarily it is responsible for calculating coordinate transformations to and from the local component coordinate frame during tracking operations.

3.2.2 Particle Tracking

By 'tracking', we generally mean either tracking some representation of the beam through the accelerator beamline, or propagating some map. The Merlin class library has been designed to allow addition of different forms of tracking without the need to modify existing code (in particular the accelerator model classes.) Currently, the Merlin library only supports particle tracking (*i.e.* ray tracing.)

Merlin separates the responsibilities for tracking (iterating) over a beamline and performing the necessary coordinate transformations between component (frames), and actually tracking, or integrating through a component (e.g. a quadrupole.) The latter is the job of the ComponentTracker class, which supplies the primary tracking interface to the accelerator components.

A ComponentTracker contains a set of Integrators which perform the physical tracking through a component. Typically, there is one integrator object per component class in the accelerator model, although there are often less integrators than component types, *e.g.* the particle tracker uses a single integrator to deal with all rectangular multipoles magnets. The job of the ComponentTracker class, therefore is to simple select the correct integrator for the current component.

To add a new tracking module, we must perform the following to steps:

- Derive a new class from ComponentTracker
- Construct a set of integrators specific to that tracker for each relevant component in the accelerator model.

Once the ComponentTracker object has been initialised with an accelerator component, it can be told to track either the entire component in one go, or take a specific step through the component. The integration step is therefore under the control of the application, avoiding having to split magnets in the model description (input deck), which is so often necessary in existing optics codes.

3.2.3 The 'Process' Concept

The ComponentTracker concept dealt with in the last section is responsible for performing some concrete tracking algorithm though individual components. Generally, we perform such tracking on a sequence of such components that represents some lattice or beamline. The applications programmer could simple write a do-loop that iterated over the components, but Merlin supplies a much higher-level concept to do this, under the control of one or more so-called Processes.

The TrackingSimulation class is a top-level abstraction for performing some accelerator simulation. Exactly what form that simulation takes is defined by one or more processes (class BunchProcess). Generally, processes fall into two categories:

- A process which increments the independent variable (usual s), *i.e.* actually tracks through a step ds; these processes are know as Transport Processes
- A process that does not actual increment s, but applies some impulse to the bunch; these are know as Impulse Processes

Processes normally represent some physics process (*e.g.* particle scattering, space charge, synchrotron radiation), but one could easily have a process that formed part of the application (*e.g.* output at certain point during tracking could be under control of a process.) A TrackingSimulation generally has at least one Transport Process that does the 'tracking. As an example, the class ParticleTracker provides the primary interface to the particle tracking module. ParticleTracker inherits from TrackingSimulation, and on construction, automatically adds a ParticleTransportProcess to its list of processes. Additional processes (*e.g.* SynchRadParticleProcess) can be added if required. ParticleTransportProcess uses a ComponentTracker to track through each component.

The real power behind a process is that taken together, the list of processes defines the finite steps taken through each component. As an example, you can tell the SynchRadParticleProcess to take twenty equidistant steps through each magnet. TrackingSimulation is responsible for managing the step length. It determines the next step to take (ds) after interrogating each of its processes in turn, after which it tells each process to take that step.

Note that processes can be turned on or off, or made active for only certain types of/specific components.

Chapter 4

Writing A User Script

4.1 Merlin++ user script fundamentals

Merlin++ follows a relatively simple overarching design in that a user script can be thought of in just five parts.

1. Defining the lattice
2. Defining the beam
3. Defining the tracker
4. Running the simulation
5. Post-processing & output of results

The following provides a code snippet corresponding to each part. Note that this is simply an example of the most basic implementation and more advanced methods exist as can be found in the tutorials.

1. Defining the Lattice (A ***OR*** B)

   A Define lattice apertures manually within Merlin++

```
AcceleratorModelConstructor newModel();
newModel.NewModel();
newModel.AppendComponent(new Quadrupole( QF ,1.0,5.5));
newModel.AppendComponent(new Drift( D1 ,12.55));
newModel.AppendComponent(new SectorBend( BFD ,5.0,0.0255,11.23));
newModel.AppendComponent(new Drift( D1 ,12.55));
newModel.AppendComponent(new Quadrupole( QD ,1.0,-5.5));
// etc..
AcceleratorModel* theModel = newModel.GetModel();
```

B Importing MAD optices/lattice apertures file

```
MADInterface madInterface("/path/to/lattice.tfs", beamenergy);
AcceleratorModel* theModel = madInterface.ConstructModel();
```

2. Defining the Beam

```
ParticleBunch* theBunch = new ParticleBunch(beamenergy);
particleBunch->AddParticle(co);
```

3. Defining the Tracker

```
ParticleTracker tracker(theModel->GetBeamline(), theBunch);
```

4. Running the Simulation

```
tracker.Run();
```

5. Processing & Output of Results

```
// Prior to running the tracker:
ofstream trackingLog("build/trackinginfo.out");

// Post running the tracker:
ParticleBunch& tracked = tracker.GetTrackedBunch();
tracked.Output(trackingLog);

// Tracking data can be imported and plotted in python/octave etc
```

For a more in-depth look at how to write user scripts for varying use-cases, please continue on to the tutorials in Chapter 5.

Chapter 5

Tutorials

In the chapter, we walk-through a number of short tutorials to demonstrate some of the key features of the Merlin++ framework. Each section corresponds to an tutorial as provided with the Merlin++ source package as default. Note that the provided tutorials focus on simulating with a circular accelerator - linac tutorials will be added in future. Includes, namespaces, defines and typedefs will not be discussed as basic C++ knowledge is assumed. Furthermore, the tutorials are designed to done methodically in order, *i.e.* only new code will be discussed in each subsequent tutorial section.

5.1 Tutorial 1 - LatticeConfiguration

The provided ManualConstruction tutorial demonstrates how to build an accelerator model manually using the Merlin++ API. The script constructs a FODO lattice storage ring made solely of quadrupole and dipoles.

LatticeConfiguration

- The following snippet shows how to construct a manually defined simple FODO lattice within Merlin++. This tutorial goes on to show how to calculate and plot the lattice horizontal beta function.

  First, an instance of a AcceleratorModelConstructor is opened and a AcceleratorModelConstructor::NewModel() is defined. Subsequently, a loop (only for periodic lattices) appends individual components with defined length, field and distance from the previous component. A final drift is appended at the end of the loop to complete the circle. Finally, and instance of AcceleratorModel is defined by the member function AcceleratorModelConstructor::GetModel().

```
AcceleratorModelConstructor latticeConstructor;
latticeConstructor.NewModel();

double rigidity = beamenergy/eV/SpeedOfLight;
double curv = (2*pi/(4*ncell));

//FODO lattice periodic
for (int n=1;n<(ncell+1);++n) {
latticeConstructor.AppendComponent(new
    Quadrupole("QF",lquad,0.0098*rigidity), n==1 ? 0 :
    0.15*lcell-ldipole);
latticeConstructor.AppendComponent(new
    SectorBend("MB",ldipole,curv,rigidity*curv), 0.15*lcell-lquad);
latticeConstructor.AppendComponent(new
    SectorBend("MB",ldipole,curv,rigidity*curv), 0.2*lcell-ldipole);
latticeConstructor.AppendComponent(new
    Quadrupole("QD",lquad,-0.0098*rigidity), 0.15*lcell-ldipole);
latticeConstructor.AppendComponent(new
    SectorBend("MB",ldipole,curv,rigidity*curv), 0.15*lcell-lquad);
latticeConstructor.AppendComponent(new
    SectorBend("MB",ldipole,curv,rigidity*curv), 0.2*lcell-ldipole);
}
latticeConstructor.AppendDrift(0.15*lcell-ldipole);
AcceleratorModel* lattice = latticeConstructor.GetModel();
```

- To confirm the lattice is viable for tracking it we calculate the closed orbit of a single reference particle following:

```
ClosedOrbit theClosedOrbit(lattice,beamenergy);
Particle particle(0);
theClosedOrbit.FindClosedOrbit(particle);
```

- Finally, lattice functions are calculated and stored using the LatticeFunctionTable class. Note that due to the linearity of this lattice and no active RF component we must force longitudinal stability prior to calculation.

```
// Calculate beta and dispersion functions
LatticeFunctionTable latticeFunctions = new
    LatticeFunctionTable(lattice,beamenergy);
latticeFunctions->SetForceLongitudinalStability(true);
latticeFunctions->Calculate();
// Write lattice functions to output file
ofstream latticeFunctionLog("build/tutorial1.out");
latticeFunctions->PrintTable(latticeFunctionLog);
```

- A corresponding python script is provided to plot the lattice function output. The result should be as follows:

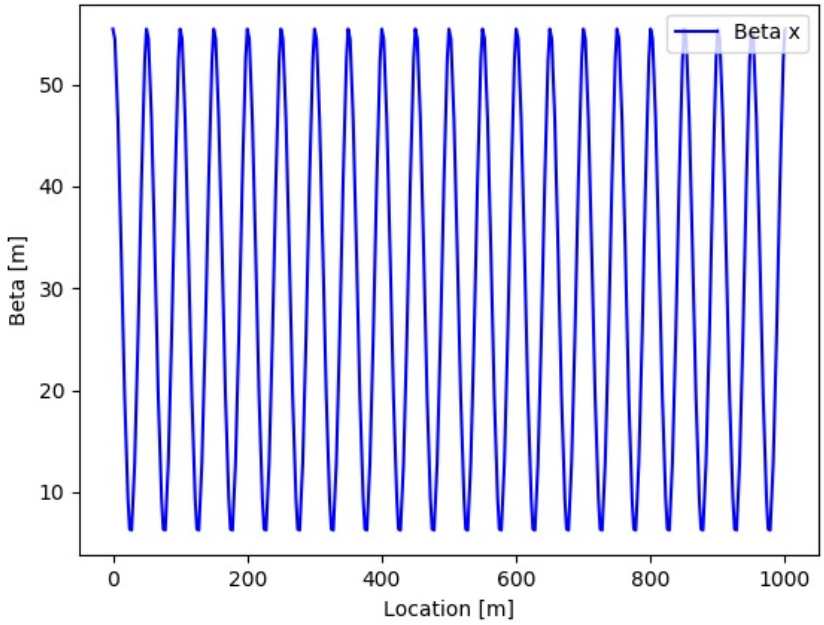


Figure 3: Horizontal Beta function of a basic manually defined FODO lattice.

5.2 Tutorial 2 - LatticeConfigurationMAD

The provided LatticeConfigurationMAD tutorial demonstrates how to build an accelerator model by importing a MAD twiss .tfs file with the Merlin++ API. The script constructs a slightly more complex FODO lattice storage consisting of dipoles, quadrupoles and sextupole corrector magnets. This tutorial also calculates and plots common lattice function calculations, including beta functions and horizontal dispersion.

LatticeConfigurationMAD

- The following shows how to import a lattice directly from a MAD twiss output .tfs file. The a MADInterface class is used imported the .tfs via C++ ifstream and the information is stored in a DataTable. The model is then constructed using the MADInterface::ConstructModel() function.

```
// Instantiate MADInterface with lattice file input
MADInterface MADinput(lattice_path, beamenergy);
// Construct Model
AcceleratorModel* theModel = MADinput.ConstructModel();
```

- A corresponding python script is provided to plot the lattice function output. The result should be as follows:

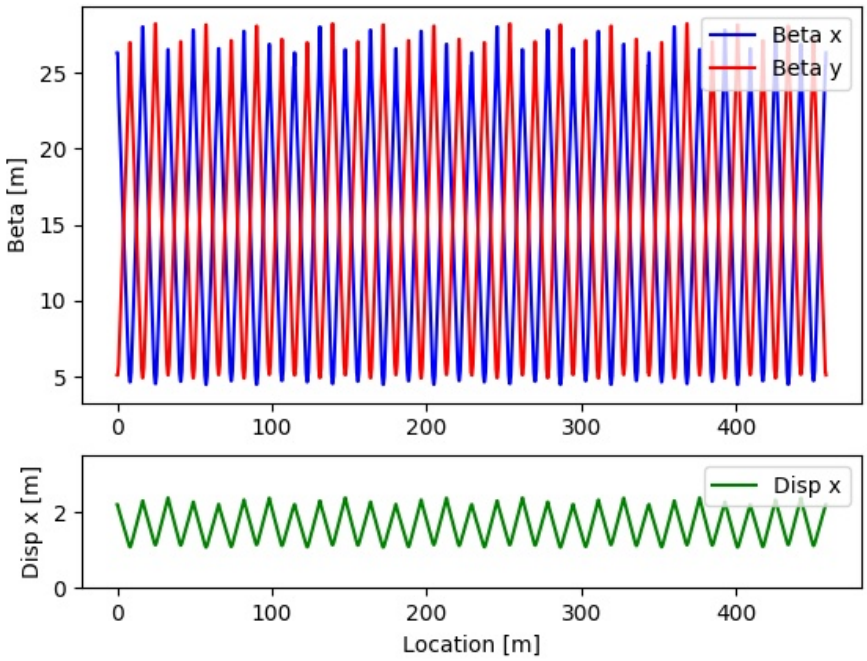


Figure 4: Beta and dispersion functions of a MAD-imported FODO lattice.

5.3 Tutorial 3 - LatticeManipulation

The LatticeManipulation tutorial shows how to alter individual component parameters within a user script directly to simulation alignment/field errors etc. This is done via the MagnetMover and MutipoleField classes.

LatticeManipulation

- The following snippet shows how to manually realignment a magnet components using the MagnetMover class. Two examples are shown. The first offsets the 20th magnet vertically along the y-axis by 100 μm. The second offsets the 40th magnet horizontally along the x-axis by 50 μm.

```
// Extract a list of magnet movers from the AcceleratorModel
MagnetMoverList magnetMovers;

theModel->ExtractTypedElements(magnetMovers);
// Offset 20th magnet 10 mm on y-axis
magnetMovers[20]->SetY(10.0e-3);
```

```
// Offset 40th magnet 50 um on x-axis
magnetMovers[40]->SetX(50.0e-6);
```

- The following snippet shows how to manually alter component fields. Two examples are shown. The first increases the 5th quadrupole field by 17%. The second increases the 21st quadrupole by 12%.

```
// Extract a list of the quadrupoles from the AcceleratorModel
vector<Quadrupole*> quadVec;
theModel->ExtractTypedElements(quadVec);

// Simulate a 17% gradient error on the 5th quadrupole
MultipoleField& field = quadVec[5]->GetField();
Complex b1a = field.GetComponent(1);
field.SetComponent(1, b1a.real() * 1.17, b1a.imag() * 1.17);

// Simulate a 12% gradient error on the 21st quadrupole
MultipoleField& field2 = quadVec[21]->GetField();
Complex b1b = field2.GetComponent(1);
field2.SetComponent(1, b1b.real() * 1.12, b1b.imag() * 1.12);
```

- A corresponding python script is provided to plot the changes in lattice functions as result of the manipulations. The result should be as follows:

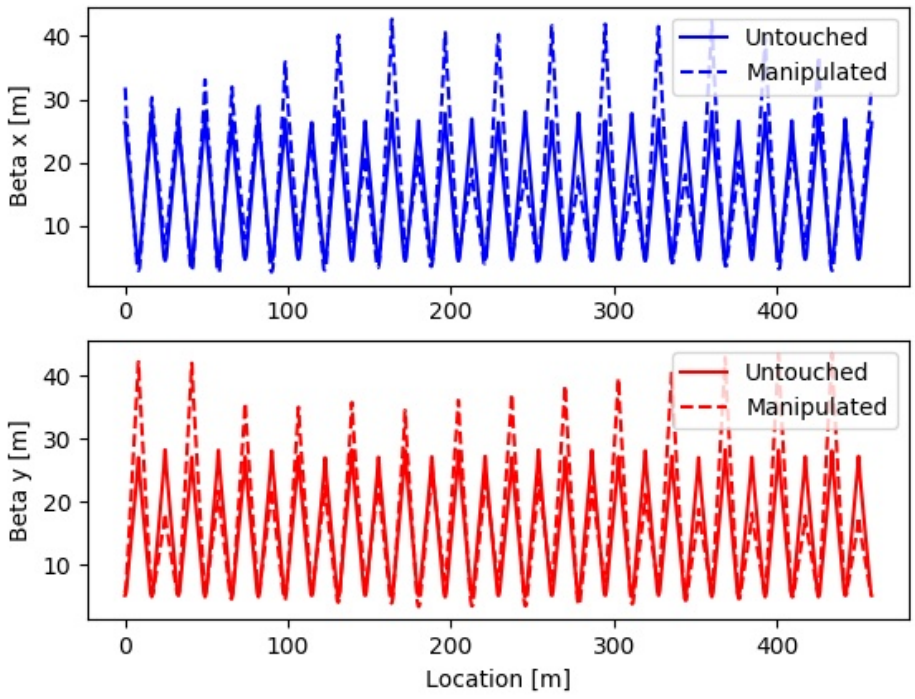


Figure 5: Variation in beta functions as a result of magnet manipulation.

5.4 Tutorial 4 - ParticleTracking

The ParticleTracking tutorial shows how to create and track a 'bunch' of 2 particles at different starting real space coordinates using the Particle, ParticleBunch and ParticleTracker classes.

ParticleTracking

- The follow shows how to use a loop to construct a particle bunch, adding each defined particle individually.

```
// Define particle bunch and add individual particles in a loop
Particle p(0);
ParticleBunch* theBunch = new ParticleBunch(beamenergy);
for(int xi = 1; xi <= 2; xi++)
{
  p.x() = xi * 0.001; // Each particle offset by 1mm on x-axis
  theParticles->AddParticle(p);
}
```

- The follow shows how to constructor a ParticleTracker using the constructed accelerator and particle bunch information.

```
// Construct a ParticleTracker to perform the tracking
ParticleTracker tracker(theModel->GetBeamline(), theParticles);
```

- The follow shows how to use a loop to track the constructed particles around the ring for multiple turns. Phase-space coordinates of each particle are also recorded after every turn using the ParticleTracker::∼GetTrackedBunch() and ParticleBunch::Output() member functions.

```
// Track and record phase-space coords after each turn for 100 turns
ofstream trackingLog("build/tutorial4.out");
for(int turn = 0; turn < 100; turn++)
{
  cout << "Tracking... turn: " << turn+1 << endl;
   if(turn == 0)
     tracker.Run();
   else
     tracker.Continue();
   ParticleBunch& tracked = tracker.GetTrackedBunch();
   tracked.Output(trackingLog);
}
```

- A corresponding python script is provided which plots the recorded particle evolution in phase-space coordinates. The result should be as follows:

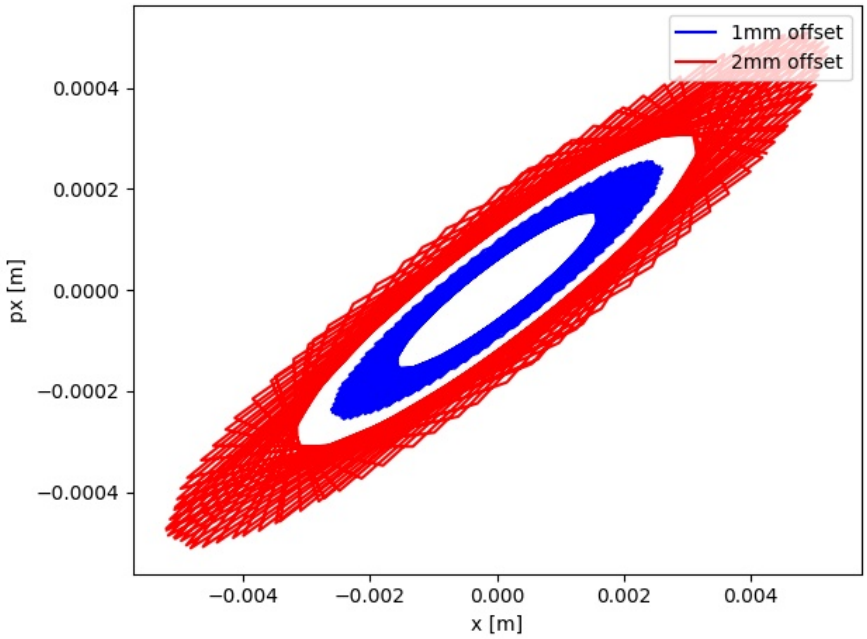


Figure 6: Horizontal phase-space evolution of 2 offset particles over 100 turns.

5.5 Tutorial 5 - ParticleBunchTracking

The ParticleTracking tutorial shows how to create and track a normally distributed particle bunch of 10,000 particles using the ParticleDistributionGeneration. When defining a bunch one must also define BeamData. The tutorial continues to show a different method of tracking, using beam position monitor (BPM) component buffers. This method records the bunch centroid in real space coordinates at every BPM component present in the constructed lattice.

ParticleBunchTracking

- The following defines the number of particles to be generated as well as and initial beam parameters.

```
// Define number of particles in bunch
size_t npart = (size_t) 1e4;

// Define basic beam parameters (example)
BeamData beam;
beam.charge = 1e8;
beam.beta_x = 0.5500005011 * meter;
beam.beta_y = 0.5499999849 * meter;
```

```
beam.alpha_x = -7.115569055e-7 * meter;
beam.alpha_y = 1.797781918e-7 * meter;
beam.emit_x = 5.026457122e-10 * meter;
beam.emit_y = 5.026457122e-10 * meter;
beam.sig_z = 75.5 * millimeter;
beam.sig_dp = 0.000113;
beam.p0 = beamenergy;
```

- The following shows to generator a normally distributed bunch with the above beam parameters.

```
// Define distribution type
ParticleDistributionGenerator* bunchDist = new
    NormalParticleDistributionGenerator();

// Generate corresponding bunch
ParticleBunch* particleBunch = new ParticleBunch(npart, *bunchDist,
    beam);
```

- The following shows how to initialise the BPM buffers and track real space coordinate evolution of the bunch centroid.

```
// Construct a BPMBuffer to record the bunch centroid at each BPM
BPMVectorBuffer* bpmVecBuffer = new BPMVectorBuffer();
BPM::SetDefaultBuffer(bpmVecBuffer);


// While tracking: Write turn tracking results to a file
ofstream bpmLog("build/tutorial5.out");
vector<BPM::Data>& theBPMBuffer = bpmVecBuffer->BPMReading;
for(vector<BPM::Data>::iterator bpm_iter = theBPMBuffer.begin();
    bpm_iter != theBPMBuffer.end(); bpm_iter++)
{
  bpmLog << std::setw(14) << (bpm_iter->x).value;
  bpmLog << std::setw(14) << (bpm_iter->x).error;
  bpmLog << std::setw(14) << (bpm_iter->y).value;
  bpmLog << std::setw(14) << (bpm_iter->y).error;
  bpmLog << endl;
}
```

- A corresponding python script is provided which plots the recorded bunch centroid evolution in real space coordinates at each BPM over 2 turns. The result should be as follows:

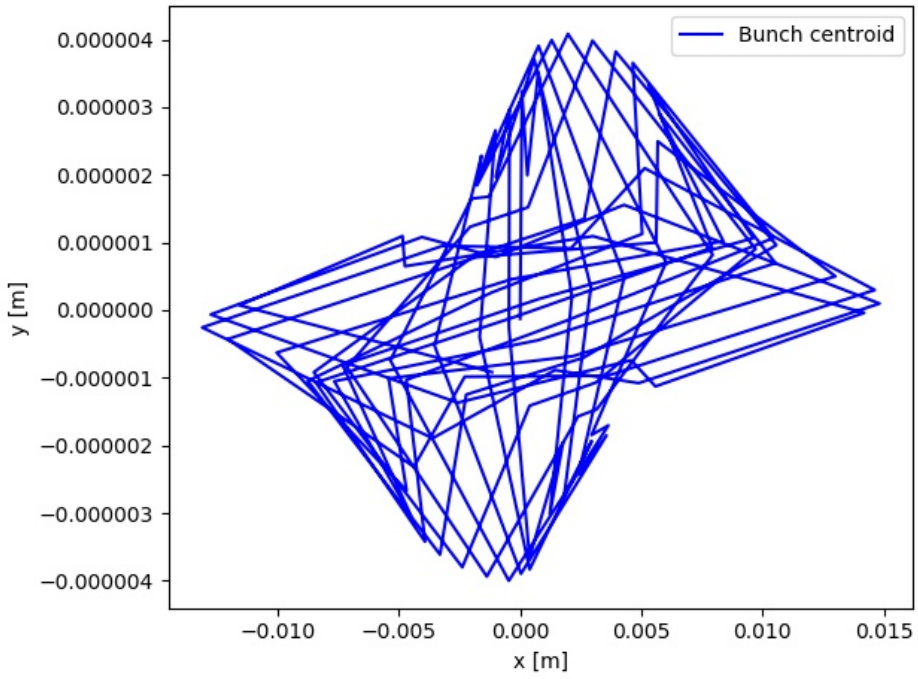


Figure 7: BPM recorded bunch centroid evolution over 2 turns.

5.6 Tutorial 6 - LHClattice

LHClattice tutorial shows how to import the LHC lattice, aperture and collimator information. A simple closed orbit calculation is performed to confirm stability and the lattice functions are subsequently calculated and recorded.

LHClattice

- The following snippet shows how to import and initialize the LHC lattice, aperture and collimator information.

```
// Import and construct LHC lattice
cout << "Loading MAD lattice file..." << endl;
MADInterface MADinput("Tutorials/input/LHC.tfs", beamenergy);
AcceleratorModel* theModel = MADinput.ConstructModel();

// Import and define component aperture information
cout << "Loading aperture information..." << endl;
ApertureConfiguration* apertures = new
    ApertureConfiguration("Tutorials/input/LHCbeam1apertureinfo.tfs");
apertures->ConfigureElementApertures(theModel);

// Define material database
cout << "Loading materials database..." << endl;
```

```
MaterialDatabase* material_db = new MaterialDatabase();

// Import and define collimator information
cout << "Loading collimators database..." << endl;
CollimatorDatabase* collimator_db = new
    CollimatorDatabase("Tutorials/input/LHCcollimatorinfo.dat",
    material_db, true);
```

- A corresponding python script is provided which plots the LHC lattice transverse beta and horizontal dispersion functions. The result should be as follows:

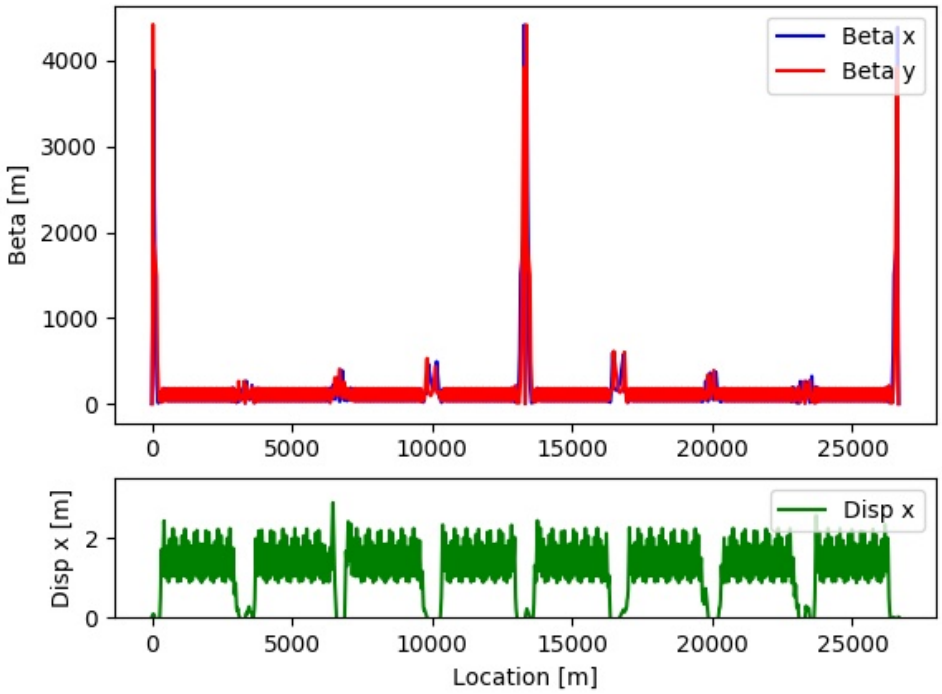


Figure 8: Tranverse beta and horizontal dispersion functions of the LHC.

5.7 Tutorial 7 - CollimationAndScattering

The CollimationAndScattering tutorial shows how to initialize a simulation such that aperture checks are carried out and particles are lost where they exceed aperture limits. Moreover, if particles are lost in a collimator component, they follow a collimation process and are scattered following a defined scatter process. The scattering process used is defined by R. Appleby *et al.* in [ref]. Each scattered and subsequently lost particle has its loss location record in a histogram. The following simulation uses the LHC accelerator lattice and

tracks 10,000 particles for 20 turns.

CollimationAndScattering

- The following initializes collimator settings and aperture information, using the CollimatorDatabase and CollimatorAperture member functions.

```
// Initialize collimator database
collimator_db->MatchBeamEnvelope(false);
collimator_db->EnableJawAlignmentErrors(false);
collimator_db->SetJawPositionError(0.0 * nanometer);
collimator_db->SetJawAngleError(0.0 * microradian);
collimator_db->SelectImpactFactor(start_element, 1.0e-6);
double impact = collimator_db->ConfigureCollimators(theModel,
    emittance, emittance, latticefunctions);

// Initialize collimator aperture info
vector<Collimator*> TCP;
int siz = theModel->ExtractTypedElements(TCP, start_element);
Aperture *ap = (TCP[0])->GetAperture();
CollimatorAperture* CollimatorJaw =
    dynamic_cast<CollimatorAperture*>(ap);
double h_offset = latticefunctions->Value(1, 0, 0,
    start_element_number);
double JawPosition = CollimatorJaw->GetFullWidth() / 2.0;
```

- For this simulation we show the use of a HorizontalHalo2ParticleDistribution, which is also filtered in accordance with the collimator jaw parameters. It is important to set the particle charge and enable scattering physics.

```
// Define horizontal bunch filter
HorizontalHaloParticleBunchFilter* hFilter = new
    HorizontalHaloParticleBunchFilter();
hFilter->SetHorizontalLimit(JawPosition);
hFilter->SetHorizontalOrbit(h_offset);

// Construct corresponding bunch
ProtonBunch* particleBunch = new ProtonBunch(npart,
    HorizontalHalo2ParticleDistributionGenerator(), beamData,
    hFilter);
particleBunch->SetMacroParticleCharge(beamData.charge);
```

- The following shows how to enable collimation and scattering. Particle loss locations are stored and loss maps can be produced using the LossMapCollimationOutput class.

```
// Enable scattering physics
particleBunch->EnableScatteringPhysics(ProtonBunch::Merlin);

LossMapCollimationOutput* lossOutput = new
    LossMapCollimationOutput(tencm);
ScatteringModel* scatterModel = new ScatteringModelMerlin;

CollimateProtonProcess* collimateProcess = new
    CollimateProtonProcess(2, 4);
collimateProcess->SetScatteringModel(scatterModel);

collimateProcess->ScatterAtCollimator(true);
collimateProcess->SetLossThreshold(200.0);
collimateProcess->SetOutputBinSize(0.1);
collimateProcess->SetCollimationOutput(lossOutput);
tracker->AddProcess(collimateProcess);
```

- A corresponding python script is provided which plots the LHC loss map for 10,000 over 20 turns with the above process properties. The result should be as follows:

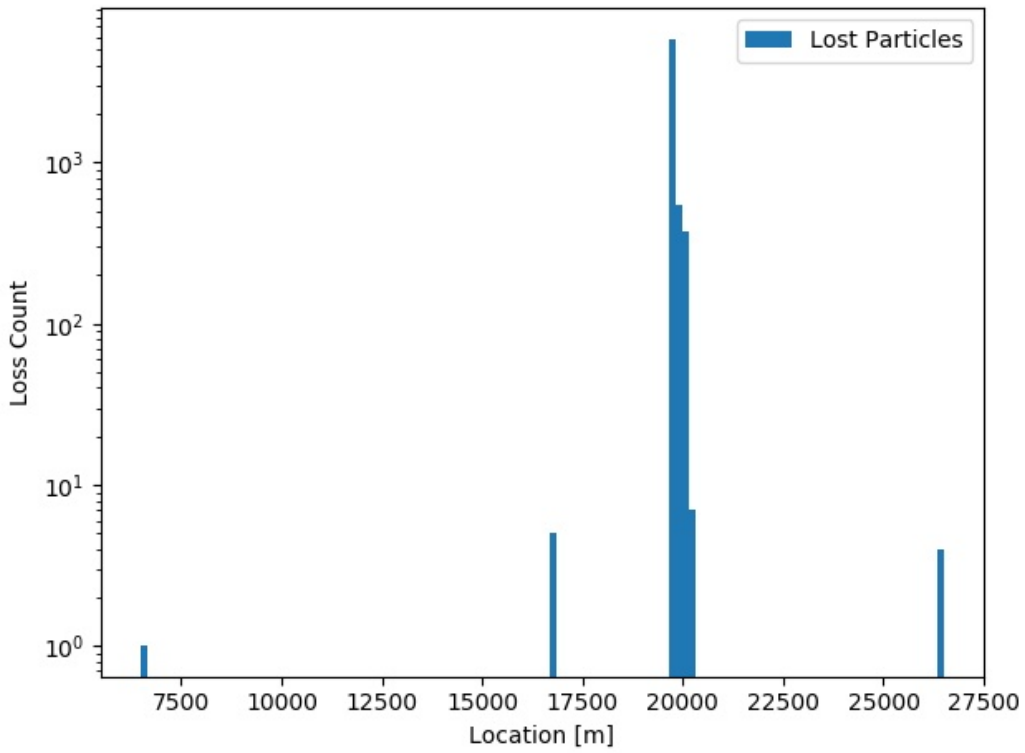


Figure 9: LHC collimation loss map following simulation of 10,000 particles.