`gm2fr`: Analysis & Simulation Toolkit for Fast Rotation

Tyler Barrett
`tjb269@cornell.edu`

January 29, 2021

1 Introduction

The *fast rotation signal* represents the cyclotron motion of the muon beam through a fixed detector plane. The beam is injected as a bunch, somewhat localized in space, which yields an oscillating signal as the bunch passes through the detector on each turn around the ring. Over time, the spread in muon momenta causes the beam to debunch: those with lower momenta have tighter curvature and complete their revolutions more quickly, pulling ahead of those with higher momenta. The peak amplitude of the oscillation decreases as the bunch spreads out, eventually leveling off to a constant uniform current around the ring.

To simulate the dynamics of fast rotation, the initial conditions of the muon distribution must be specified. For each muon, two initial conditions are required: a fast rotation phase and frequency, analogous to initial position and velocity, as described below.

- *Injection time*, τ. This sets the phase of each muon's fast rotation. Conceptually, it is the time when a muon passes through the end of the inflector and enters the storage ring. It is expressed as an offset relative to the mean arrival time (the centroid of the bunch), typically with an RMS spread of about ± 25 ns. Positive time offsets indicate late arrival (the tail of the bunch), and negative time offsets indicate early arrival (the head of the bunch).
- *Cyclotron frequency*, f. This sets the speed of the muon's fast rotation, which is taken as constant throughout the fill. It is interchangeable with other kinematics variables, such as energy, momentum, cyclotron radius, or cyclotron period. This toolkit tracks muon kinematics internally in units of cyclotron frequency, but other kinematics variables may be used as input.

Together, these initial conditions set the ith muon's fast rotation signal as

$$\mathcal{S}_i(t) = \sum_n \delta\left[t - \left(\frac{n+n_0}{f_i} + \tau_i\right)\right], \tag{1}$$

where δ is the Dirac delta function, n is the turn number, and n_0 is the location of the detector plane as a fraction of a turn. Therefore, the first detection on turn $n = 0$ takes place at time $n_0/f_i + \tau_i$. This is the initial time offset (τ_i) plus the time it takes to get to the detector plane (n_0/f_i). The full fast rotation signal is then the sum of Eq. (1) over all muons.

This is a useful model for fast rotation, but it cannot be realized directly in data, because interactions with a detector plane would scatter the muon beam. Instead, a reference calorimeter C_{ref} is chosen as a proxy for the detector plane, with the decay positron counts a proxy for the number of muons. Then the decay positron signal from each calorimeter C_i is aligned in time with C_{ref}, assuming a delay of $[(C_i - C_{\text{ref}})/24]\, T_{\text{magic}}$, where T_{magic} is the magic cyclotron period. Other contributions to the combined positron signal (i.e. decay envelope, spin precession, beam dynamics, etc.) are removed by dividing out a fit function or taking a ratio of two time-shifted signals to cancel slow effects. This leaves only the fast cyclotron motion, as the beam periodically approaches and departs from the acceptance region of the reference calorimeter.

2 Prerequisites

First, be sure you can run `python3` and that all of the following imports work correctly.

```
import ROOT
import numpy
import scipy
import matplotlib.pyplot
```

Next, you should add the top-level `gm2fr` directory to your `PYTHONPATH` environment variable, so that Python can find where the package is located in your filesystem. Using `bash`, add the following line to your start-up script.

```
export PYTHONPATH=${PYTHONPATH}:/path/to/gm2fr
```

3 Usage

The main simulation code is encapsulated within the class `gm2fr.simulation.simulator.Simulator`. When the `Simulator` object is instantiated, the constructor has a single required argument: a string which indicates the name of the desired output directory. By default, the program quits if the output directory already exists; this behavior can be overridden by setting the boolean keyword argument `overwrite`.

```
from gm2fr.simulation.simulator import Simulator

simulation = Simulator("outputFolder", overwrite = True)
```

3.1 Input

After creating the `Simulator` instance, the user must specify distributions for the muons' injection times and kinematics variables (e.g. cyclotron frequency). These distributions will be used to generate random samples for each muon in the simulation.

There are three supported input formats, described below: Gaussian mixtures, `ROOT TH1` histograms, or a `ROOT TH2` joint histogram. Using any format, the injection time units may be nanoseconds, microseconds, or seconds, and the kinematics variable may be cyclotron frequency (in kHz), momentum (in GeV), or fractional offset from the magic momentum.

3.1.1 Gaussian Mixtures

The supplied class `gm2fr.simulation.mixture.GaussianMixture` can be used to represent a superposition of one or more Gaussians, whose relative weights, means, and widths are supplied as lists in the constructor arguments. This input mode is most useful for creating simple toy models from scratch. It also has the benefit of drawing from continuous probability distributions, rather than discretized `ROOT` histograms.

```
from gm2fr.simulation.mixture import GaussianMixture

# A single Gaussian.
oneGaussian = GaussianMixture(weights = [1], means = [6705], widths = [10])

# A superposition of two Gaussians. (Keyword names are optional.)
twoGaussians = GaussianMixture([1, 0.5], [6705, 6698], [10, 6])
```

After creating a `GaussianMixture` instance for the injection times and kinematics variables, the two distributions and their chosen units are specified using the `Simulator.useMixture(...)` method shown below.

```
from gm2fr.simulation.simulator import Simulator
from gm2fr.simulation.mixture import GaussianMixture

```

```
simulation = Simulator("outputFolder", overwrite = True)

frequency = GaussianMixture([1], [6705], [10])
injection = GaussianMixture([1], [0], [25])

simulation.useMixture(
  frequency,     # GaussianMixture object for kinematics variable
  "frequency",   # kinematics variable: "frequency" / "momentum" / "offset"
  injection,     # GaussianMixture object for injection time
  "nanoseconds"  # time units: "nanoseconds" / "microseconds" / "seconds"
)
```

3.1.2 `ROOT TH1` Histograms

In a similar manner, the injection time and kinematics distributions may be supplied as one-dimensional `ROOT TH1` histograms. This mode is most useful for importing distributions generated externally, such as output from a full-scale simulation like `gm2ringsim`. It may also be used to create custom distributions which cannot be easily approximated by the Gaussian mixture input mode. After instantiating the `Simulator` object, the input histograms and units are specified with the `Simulator.useHistogram1D(...)` method as shown.

```
from gm2fr.simulation.simulator import Simulator
import ROOT as root

simulation = Simulator("outputFolder", overwrite = True)

momentum = root.TFile("myFile.root").Get("myMomentumHistogram")
injection = root.TFile("myFile.root").Get("myInjectionHistogram")

simulation.useHistogram1D(
  momentum,      # TH1 object for kinematics variable
  "momentum",    # kinematics variable: "frequency" / "momentum" / "offset"
  injection,     # TH1 object for injection time
  "nanoseconds"  # time units: "nanoseconds" / "microseconds" / "seconds"
)
```

3.1.3 `ROOT TH2` Joint Histogram

The injection time and kinematics distributions may also be specified jointly as a two-dimensional `ROOT TH2` histogram. Like the `ROOT TH1` option, this mode is ideal for importing distributions generated externally, but this mode naturally incorporates any correlation between injection times and muon kinematics. This benefit makes the `ROOT TH2` option the best choice for Toy Monte Carlo based on full-scale realistic simulations from `gm2ringsim`, `COSY`, or `BMAD`. The input histogram and units are specified with the `Simulator.useHistogram2D(...)` method as shown.

```
from gm2fr.simulation.simulator import Simulator
import ROOT as root

simulation = Simulator("outputFolder", overwrite = True)

joint = root.TFile("myFile.root").Get("myJointHistogram")

simulation.useHistogram2D(
  joint,         # TH2 object for joint distribution
  "momentum",    # kinematics variable: "frequency" / "momentum" / "offset"
  "nanoseconds"  # time units: "nanoseconds" / "microseconds" / "seconds"
)
```

3.1.4 Toy Correlation

For the one-dimensional inputs (i.e. Gaussian mixture and `ROOT TH1`), there is an optional argument for a toy correlation polynomial which shifts the kinematics variable as a function of the injection time:

$$\Delta v(\tau) = c_n \tau^n + c_{n-1}\tau^{n-1} + \ldots + c_0, \tag{2}$$

where Δv is the shift in the kinematics variable relative to the specified distribution. The polynomial must be specified as a list of coefficients in decreasing order: `[c_n, ..., c_0]`. An example using a linear correlation polynomial with the Gaussian mixture input mode is shown below. The same applies for the `ROOT TH1` mode.

```python
from gm2fr.simulation.simulator import Simulator
from gm2fr.simulation.mixture import GaussianMixture

simulation = Simulator("outputFolder", overwrite = True)

frequency = GaussianMixture([1], [6705], [10])
injection = GaussianMixture([1], [0], [25])

# A linear correlation polynomial which shifts by +10 kHz over +50 ns.
correlation = [10 / 50, 0]

simulation.useMixture(
  frequency,     # GaussianMixture object for kinematics variable
  "frequency",   # string for kinematics variable: "frequency"/"momentum"/"offset"
  injection,     # GaussianMixture object for injection time
  "nanoseconds", # string for time units: "nanoseconds"/"microseconds"/"seconds",
  correlation    # (optional) correlation polynomial coefficients
)
```

3.2 Simulation

After instantiating the `Simulator` object and specifying the input distributions, the fast rotation signal is generated by calling the `Simulator.simulate(...)` method. The only required argument is the number of muons to simulate, but there are several optional arguments which can change the simulation mode.

- `muons`: the number of muons to simulate.
- `end`: the length of the simulation, in nanoseconds; defaults to 150×10^3.
- `detector`: the location of the detector plane, as a fraction of a turn (i.e. n_0); defaults to 0.74.
- `decay`: the muon decay mode, “`none`” / “`exponential`” / “`uniform`”; defaults to “`none`”.
  - “`none`” means each muon is counted repeatedly, once on every turn around the ring. This leads to fully correlated statistical fluctuations across turns in the fast rotation signal, but provides the smoothest signal using the fewest muons if bin uncertainties are not important. This choice is most appropriate for the Fourier analysis method.
  - “`exponential`” means each muon is counted only once, at the last detector crossing before decay. The decay time for each muon is chosen randomly from an exponential distribution using each muon’s boosted lifetime. After generating the signal, the exponential decay envelope is divided out using the ensemble average of the boosted lifetimes. This requires many more muons to generate a smooth signal, but provides more meaningful statistical uncertainties and more closely models the real data signals.

  - "`uniform`" means each muon is counted only once, on a turn number chosen uniformly at random. This option provides the most meaningful statistical uncertainties, because the bin contents are entirely uncorrelated and there is no decay envelope to remove. This choice is most appropriate for the χ^2 analysis method.

- `backward`: boolean switch for backward extension of the signal, `True` or `False`; defaults to `False`.
  - For decay mode "`none`", this simply extends the turn number n to the negative integers when evaluating each muon's detector crossings.
  - For decay mode "`exponential`", half of the muons get their decay times negated, which effectively yields a negative turn number n for those muons' detector crossings. Each muon is still counted only once.
  - For decay mode "`uniform`", each muon's randomly selected turn number is extended to include the negative integers. Each muon is still counted only once.

- `normalize`: boolean switch for normalization of the signal, `True` or `False`; defaults to `True`.
  - A normalized fast rotation signal converges to 1 after debunching (apart from noise effects). An unnormalized signal contains the raw number of counts in each time bin.
  - For decay mode "`none`", the signal is divided by the expected number of muons per time bin after debunching. In each turn, all N muons are evenly distributed over $\langle T\rangle/\Delta t$ bins, so the expected number per bin is $N \times \Delta t/\langle T\rangle$.
  - For decay mode "`exponential`", the signal is divided by the expected distribution $(N/\langle\tau\rangle)\, e^{-t/\langle\tau\rangle}$, where N is the total number of muons and in this context $\langle\tau\rangle$ is the average boosted lifetime. If `backward` is enabled, the scale is halved to $N/2$ and $|t|$ is used to divide a symmetric exponential in both directions. At late times, when raw counts become small and relative fluctuations become large, the normalized signal exhibits significantly growing noise which oscillates about 1.
  - For decay mode "`uniform`", like "`none`", the signal is divided by the expected number of muons per time bin after debunching. In this case, each turn sees an expected N/n_{total} muons evenly distributed over $\langle T\rangle/\Delta t$ bins, where N is the total number of muons and n_{total} is the total number of turns. Therefore, the expected number of muons per bin is $(N/n_{\text{total}}) \times \Delta t/\langle T\rangle$.

- `batchSize`: the maximum number of muons to process in memory at a time; defaults to 10^7.
  - The NumPy package performs fast numerical operations on arrays in Python by implementing loops under-the-hood in C. However, this requires allocating all muons' properties simultaneously as arrays in memory. If the desired number of muons is larger than what the system memory can accommodate, the total will be split into batches indicated by `batchSize`. A native Python loop then iterates over the set of batches, with each batch processed using NumPy as usual, and their fast rotation signals are added cumulatively.

    As a general guideline, the simulation uses up to about 10 arrays of 8-byte floating-point numbers, which amounts to $10 \times 8 \times N$ bytes in memory, where N is the number of muons. At the default batch size of 10^7, memory usage should therefore be limited to $\mathcal{O}(1\text{ GB})$ at any given time. The user may adjust the `batchSize` option as appropriate for their system.

The following example runs a simulation with 10^6 muons and all other options left to their defaults. This means the signal extends to 150 μs, the detector is placed at 0.74 turns, there is no muon decay, the signal goes only forward in time, and the debunched signal is normalized to 1. (Also, all 10^6 muons will be processed in memory as a single contiguous batch.)

```
simulation.simulate(1E6)
```

The following example runs a simulation with 10^9 muons and all keyword options set manually.

```
simulation.simulate(
  muons = 1E9,          # simulate one billion muons in total
  end = 200_000,        # signal extends to 200 microseconds
  detector = 0.13,      # detector plane offset of 0.13 turns
  decay = "uniform",    # each muon counted once on random turn
  backward = True,      # signal extended backward to -200 microseconds
  normalize = False,    # signal will include raw counts in each bin
  batchSize = 1E8       # more RAM available: allow 100 million muons per batch
)
```

Once completed, the resulting data is stored using `gm2fr.simulation.histogram.Histogram` objects. This is a simple wrapper class that encapsulates the NumPy arrays for bin edges/centers, heights, and errors, as listed below.

- `Histogram.xCenters`, the bin centers along the x axis.
- `Histogram.yCenters`, the bin centers along the y axis (if present).
- `Histogram.xEdges`, the bin edges along the x axis.
- `Histogram.yEdges`, the bin edges along the y axis (if present).
- `Histogram.heights`, the one- or two-dimensional array of bin heights.
  - If two-dimensional, the x bins are along the first dimension, and the y bins are along the second dimension. For example, `heights[i, j]` extracts the bin content at `xCenters[i]` and `yCenters[j]`. Similarly, `heights[i, :]` extracts a vertical profile at fixed `xCenters[i]`, and `heights[:, j]` extracts a horizontal profile at fixed `yCenters[j]`.
- `Histogram.errors`, the array of bin uncertainties, whose shape matches `Histogram.heights`.

The `Histogram` objects stored within the `Simulator` object may be accessed using the identifiers below.

- `Simulator.frequencies`, the truth-level cyclotron frequency histogram.
- `Simulator.profile`, the truth-level injection time histogram (or "beam profile").
- `Simulator.joint`, the truth-level joint histogram of injection time and cyclotron frequency.
- `Simulator.signal`, the fast rotation signal.

For example, the NumPy array for the fast rotation signal may be accessed using `Simulator.signal.heights`, with the time bin centers given by `Simulator.signal.xCenters`.

3.3 Output

The `Simulator` object can automatically save plots and data to the directory specified when the instance was created.

3.3.1 Plots

The method `Simulator.plot()` automatically generates the following plots and saves them in PDF format.

- `frequencies.pdf`, the truth-level frequency histogram.
- `profile.pdf`, the truth-level injection time histogram (or "beam profile").
- `joint.pdf`, the joint histogram of injection times and cyclotron frequencies.
- `signal.pdf`, the fast rotation signal.

– By default, the viewing window covers the full length of the signal, which typically exhibits the debunching envelope but not the turn-by-turn oscillations. The optional keyword argument `times` can be used to supply a list of times (in nanoseconds) that will be treated as the upper limits for a sequence of PDFs, in addition to the full viewing range. If `backward` is enabled, these will be treated as symmetric upper and lower time limits. The filenames will be formatted as `signal_i.pdf`, where `i` is the index of the time limit from the list. For example, the argument `times = [5_000, 10_000, 20_000]` plots the fast rotation signal up to 5 μs, 10 μs, and 20 μs, in addition to the full range.

3.3.2 NumPy Data

The plotted data are also saved in NumPy format within the `numpy` subdirectory of the simulation folder. Each `Histogram` object is consolidated into a `.npz` archive containing NumPy arrays for the bin ranges, heights, and errors. These archives are not intended to be accessed directly by the user, but rather the `Histogram` object can be easily recreated via the class method `Histogram.load(...)`, which takes the filename as its only argument. Then the data may be accessed directly from the `Histogram` object using the aforementioned instance variables, e.g. `Histogram.xCenters` and `Histogram.heights`.

```python
from gm2fr.simulation.histogram import Histogram

# Load the fast rotation signal histogram from disk.
hSignal = Histogram.load("mySimulation/numpy/signal.npz")

# Pull the NumPy arrays from the histogram object.
times = hSignal.xCenters
signal = hSignal.heights

# Load the true frequency distribution from disk.
hFrequency = Histogram.load("mySimulation/numpy/frequencies.npz")

# Pull the NumPy arrays from the histogram object.
frequencies = hFrequency.xCenters
distribution = hFrequency.heights
```

At this point, the NumPy arrays are available for analysis in the rest of your script.

3.3.3 ROOT Data