BNL-230063-2026-TECH

Adiabatic Longitudinal Island Formation Due to Phase Modulation Using BLonD

H. Lovelace III

August 2026

Collider Accelerator Department

Brookhaven National Laboratory

U.S. Department of Energy

USDOE Office of Science (SC), Nuclear Physics (NP)

DISCLAIMER

Adiabatic Longitudinal Island Formation Due to Phase Modulation Using BLonD

H. Lovelace III, C. Chen

August 5, 2026

Abstract The studies of adiabatic island generation through tracking near the quarter integer resonance and separately, through phase modulation was conducted using the accelerator code BLonD [1]. Parameters from the Hadron Storage Ring (HSR) were used in the model. The demonstration of resonance island formation through phase modulation will help to understand this possible source of beam dilution during the store and further control of the bunch shaping. In this document, an algorithm to calculate the 5^{th}, 7^{th}, and N^{th} order resonances is given, tracking results are presented, and the stable fixed point calculations with comparison to Poincaré contours are given.

1 Introduction

The motion in a 2-D Hamiltonian system remains stable within the boundary of the rotation and oscillation, the separatrix. In the longitudinal case, this boundary follows the curve

$$\delta_{sep} \;=\; \pm\sqrt{\frac{2ZVE_0\beta^2}{\pi h|\eta|}}\cdot\sqrt{\cos(\phi_s)-\cos(\phi)+(\phi-\phi_s)\cdot\sin(\phi_s)} \tag{1}$$

where δ_{sep} is the difference of the off-momentum particle and on-momentum particle normalized to the momentum, the slippage η is given by the distance of the γ-transition from the Lorentz boost γ, and Z is the atomic number. The Hamiltonian from which the curve of the separatrix is derived is

$$\mathbf{H}(s,\delta) = \frac{h\,\eta\,\omega_0}{2\,\beta^2}\delta^2 \;+\; \frac{ZV}{2\pi h\,E_0\beta^2}\left[\cos\left(\phi_s+\frac{h\omega_0}{\beta c}\,s\right)-\cos(\phi_s)-\frac{h\omega_0}{\beta c}\,s\cdot\sin(\phi_s)\right] \tag{2}$$

where h is the harmonic number, ω_0 is the revolution angular frequency, E_0 is the nominal total energy of a particle, V is the RF voltage, and since the change in energy per turn is zero, the synchronous phase ϕ_s is zero. The separatrix for the constant energy system is

$$\delta_{sep} \;=\; \pm\sqrt{\frac{2ZVE_0\beta^2}{\pi h|\eta|}}\cdot\sqrt{1-\cos\phi} \tag{3}$$

Understanding longitudinal resonance island can help in mitigating beam dilution at store or injection due to coherent phase noise driven by the RF [2, 3]. Tune modulation [4] occurs when the synchrotron tune is an integer factor of the modulation tune

$$m\,Q_s = Q_m. \tag{4}$$

Typically, these resonant frequencies are avoided through tune mechanism. For the study presented in this tech note, the resonances are excited to generate resonance islands through phase modulation. To formulate the Hamiltonian with this external modulation, a perturbative term must be added. The perturbed Hamiltonian is written as:

$$\mathbf{H} = H_0(J) + H_m(J,\Psi) \tag{5}$$

where the we have that the linear, $H_0(J)$, part of the Hamiltonian is independent of the phase, Ψ for a given system. From Ref. [5] we have the perturbed Hamiltonian:

$$\mathbf{H} = \frac{1}{2}Q_s\delta^2 + Q_s[1-\cos\phi] + Q_m a\delta_n \cos Q_m\theta \tag{6}$$

where a is the amplitude of the phase shift, Q_m is the modulation tune, and θ is the orbital angle.

2 Theory

From Equ. 6, and the generating functions given in Ref. [5] for the small action limit $J \lesssim 2$ and to remove the momentum dependence,

$$\begin{aligned} F_1(\tilde{\phi},\psi) &= \frac{-\tilde{\phi}^2}{2}\tan\psi \\ F_2(\phi,\tilde{\delta}) &= (\phi - a\sin Q_m\theta)/\tilde{\delta} \end{aligned} \tag{7}$$

we are able to write the Hamiltonian as

$$H = \frac{\delta_n^2}{2Q_s} + Q_s\left[1-\cos\left(\tilde{\phi} + a\sin Q_m\theta\right)\right]. \tag{8}$$

The corresponding action, J, is

$$J = \frac{1}{2\pi}\oint \tilde{\delta}\, d\tilde{\phi} = \frac{8}{\pi}\left[E(k) - \left(1-k^2\right)K(k)\right], \tag{9}$$

where $E(k)$ and $K(k)$ are the complete elliptic integrals of the second and first kinds, respectively,

$$E(k) = \int_0^{\pi/2}\sqrt{1-k^2\sin^2 w}\, dw, \tag{10}$$

$$K(k) = \int_0^{\pi/2}\frac{dw}{\sqrt{1-k^2\sin^2 w}}. \tag{11}$$

Substituting the action-angle definitions back into the unperturbed Hamiltonian gives

$$H_0 = Q_s J\sin^2\psi + Q_s\left[1-\cos\left(\sqrt{2J}\cos\psi + a\sin Q_m\theta\right)\right]. \tag{12}$$

where at small-amplitude perturbation,

$$\Delta H_0 = Q_s\left[-\frac{J}{2}\cos 2\psi - 2\sum_{k=1}^{\infty}(-1)^k J_{2k}\left(\sqrt{2J}\right)\cos(2k\psi)\right]. \tag{13}$$

The Jacobi elliptic functions satisfy

$$\sin w = \mathrm{sn}(u|k), \qquad \cos w = \mathrm{cn}(u|k). \tag{14}$$

where

$$u = \int_0^w \frac{dw}{\sqrt{1-k^2\sin^2 w}} \qquad\qquad u_0 = \int_0^{w_0}\frac{dw}{\sqrt{1-k^2\sin^2 w}} \tag{15}$$

The expansion fo δ in Fourier harmonics of ψ is $\mathrm{cn}(u|k)$ in the angle variable

$$\psi = \pi u/(2K(k) \tag{16}$$

is

$$\mathrm{cn}(u|k) = \frac{2\pi}{kK(k)}\sum_{n=0}^{\infty}\frac{q^{(n+\frac{1}{2})}}{1+q^{2n+1}}\cos[(2n+1)\psi], \tag{17}$$

where the q is given by

$$q = e^{-\pi K'(k)/K(k)} = \frac{k^2}{16} + 8\left(\frac{k^2}{16}\right)^2 + 84\left(\frac{k^2}{16}\right)^3 + \cdots . \tag{18}$$

Here $K'(k) \equiv K(k')$, where the complementary modulus is

$$k' = \sqrt{1-k^2}. \tag{19}$$

which is valid for

$$J \lesssim 2. \tag{20}$$

The action, J, and modulus, k, are related by

$$J = 2k^2\left(1 + \frac{1}{8}k^2 + \frac{3}{64}k^4 + \cdots\right), \tag{21}$$

which may be inverted to obtain

$$k^2 = \frac{J}{2}\left(1 - \frac{J}{16} - \frac{J^2}{256} + \cdots\right). \tag{22}$$

Therefore,

$$2k\ \mathrm{cn}(u|k) = (2J)^{1/2}\cos\psi + \frac{(2J)^{3/2}}{64}\cos 3\psi + \frac{(2J)^{5/2}}{4096}\cos 5\psi + \cdots . \tag{23}$$

For the m^{th}-order resonance ($m = 2k+1$), the driving amplitude is proportional to $Q_s a J_m(\sqrt{2J})$. Using the small-argument Taylor expansion of the Bessel function, J_m,

$$J_m(x) \approx \frac{x^m}{2^m m!}, \tag{24}$$

for $x = \sqrt{2J}$,

$$J_m(\sqrt{2J}) = \frac{(2J)^{m/2}}{2^m m!}. \tag{25}$$

The linear contribution to the Hamiltonian,

$$\mathbf{H_0} = \left(Q_s - \frac{Q_m}{N}\right)J - \frac{Q_s}{16}J^2, \tag{26}$$

is obtained by transforming to the rotating frame with phase

$$\tilde{\psi} = \psi - Q_m\theta/N. \tag{27}$$

Under this canonical transformation, the Hamiltonian acquires the additional term $-(Q_m/N)J$, while the expansion of the unperturbed pendulum Hamiltonian to second order in the action contributes the nonlinear detuning term $-Q_s J^2/16$. For odd orders, $m = 2k+1 = N$ and the subsequent Hamiltonian becomes:

$$\mathbf{H} = \left(Q_s - \frac{Q_m}{N}\right)J - \frac{Q_s}{16}J^2 - \frac{Q_s a}{2^N N!}(2J)^{N/2}\cos(N\tilde{\psi}). \tag{28}$$

We obtain the 3^{rd}, 5^{th}, and 7^{rd} order Hamiltonians:

$$\begin{aligned}
&\text{For N} = 3,\\
&\quad H = \left(Q_s - \frac{Q_m}{3}\right)J - \frac{Q_s}{16}J^2 - \frac{Q_s a}{64}(2J)^{3/2}\cos(3\psi),\\
&\text{for N} = 5,\\
&\quad H = \left(Q_s - \frac{Q_m}{5}\right)J - \frac{Q_s}{16}J^2 - \frac{Q_s a}{4096}(2J)^{5/2}\cos(5\psi),\\
&\text{and for N} = 7,\\
&\quad H = \left(Q_s - \frac{Q_m}{7}\right)J - \frac{Q_s}{16}J^2 - \frac{Q_s a}{262144}(2J)^{7/2}\cos(7\psi).
\end{aligned} \tag{29}$$

If the system undergoes voltage modulation instead phase, the even terms remain in the perturbed Hamiltonian [6, 7].

The fixed points of the Hamiltonian can be solved by finding the roots of of the equations of motion:

$$\frac{\partial H}{\partial J} = 0, \qquad \frac{\partial H}{\partial \psi} = 0 \tag{30}$$

The stability of the fixed points are evaluated using the Hessian matrix,

$$\boldsymbol{H} = \begin{pmatrix} \partial^2 H/\partial J^2 & \partial^2 H/(\partial J \partial \psi) \\ \partial^2 H/(\partial \psi \partial J) & \partial^2 H/\partial \psi^2 \end{pmatrix} \tag{31}$$

$$\det(\boldsymbol{H}) = \frac{\partial^2 H}{\partial J^2}\frac{\partial^2 H}{\partial \psi^2} - (\frac{\partial^2 H}{\partial J \partial \psi})^2 \tag{32}$$

From the determinant of the Hessian matrix, the fixed point is determined to be elliptic (stable) if $\det(\boldsymbol{H}) > 0$ or hyperbolic (unstable) if $\det(\boldsymbol{H}) < 0$.

The time averaged Hamiltonian is written as:

$$< H >= (Q_s - Q_m)\tilde{J} - \frac{Q_s}{16}J^2 + \frac{Q_s a}{2}(2J)^{1/2}\cos(\tilde{\psi}). \tag{33}$$

for $N = 1$. We can write the Hamiltonian equations of motion as:

$$\begin{aligned} \dot{J} &= -\frac{1}{2}Q_s a\sqrt{2J}\sin\psi \\ \dot{\psi} &= (Q_s - Q_m) - \frac{Q_s J}{8} - \frac{Q_s a}{2\sqrt{2J}}\cos\psi \end{aligned} \tag{34}$$

If we let

$$g = \sqrt{2J}\cos\psi \tag{35}$$

the fixed points of the Hamiltonian are

$$g^3 - 16(1 - \frac{Q_m}{Q_s})g + 8a = 0. \tag{36}$$

The roots obtained for the N = 1 case are shown in Fig. 1. The bifurcation tune, Q_{bif}, is directly proportional to the synchrotron tune through

$$Q_{bif} = Q_s\left(1 - \left(\frac{3}{16}\right)(4a)^{2/3}\right). \tag{37}$$

Using the methodology provided in [5], we find the 3^{rd}, 5^{th}, and 7^{th} order bifurcation tunes;

$$\begin{aligned} Q_{bif}^{(3)} &= 3Q_s\left(1 + \frac{9a^2}{1024}\right) \\ Q_{bif}^{(5)} &= 5Q_s\left(1 - \frac{16384}{675a^2}\right) \\ Q_{bif}^{(7)} &= 7Q_s\left(1 - \frac{192}{5(35a)^{2/3}}\right). \end{aligned} \tag{38}$$

It is important to note that from the

$$g_{bif}^{4-N} = 8aN(N-2)\frac{1}{8^{N-1}} \tag{39}$$

if $N \geq 5$ then the amplitude, a, moves to the denominator which means as the modulation becomes small, the bifurcation tune moves away from the $5Q_s$ which explains why the resonance islands collapse at higher orders and low perturbation strength.

3 Simulation Results

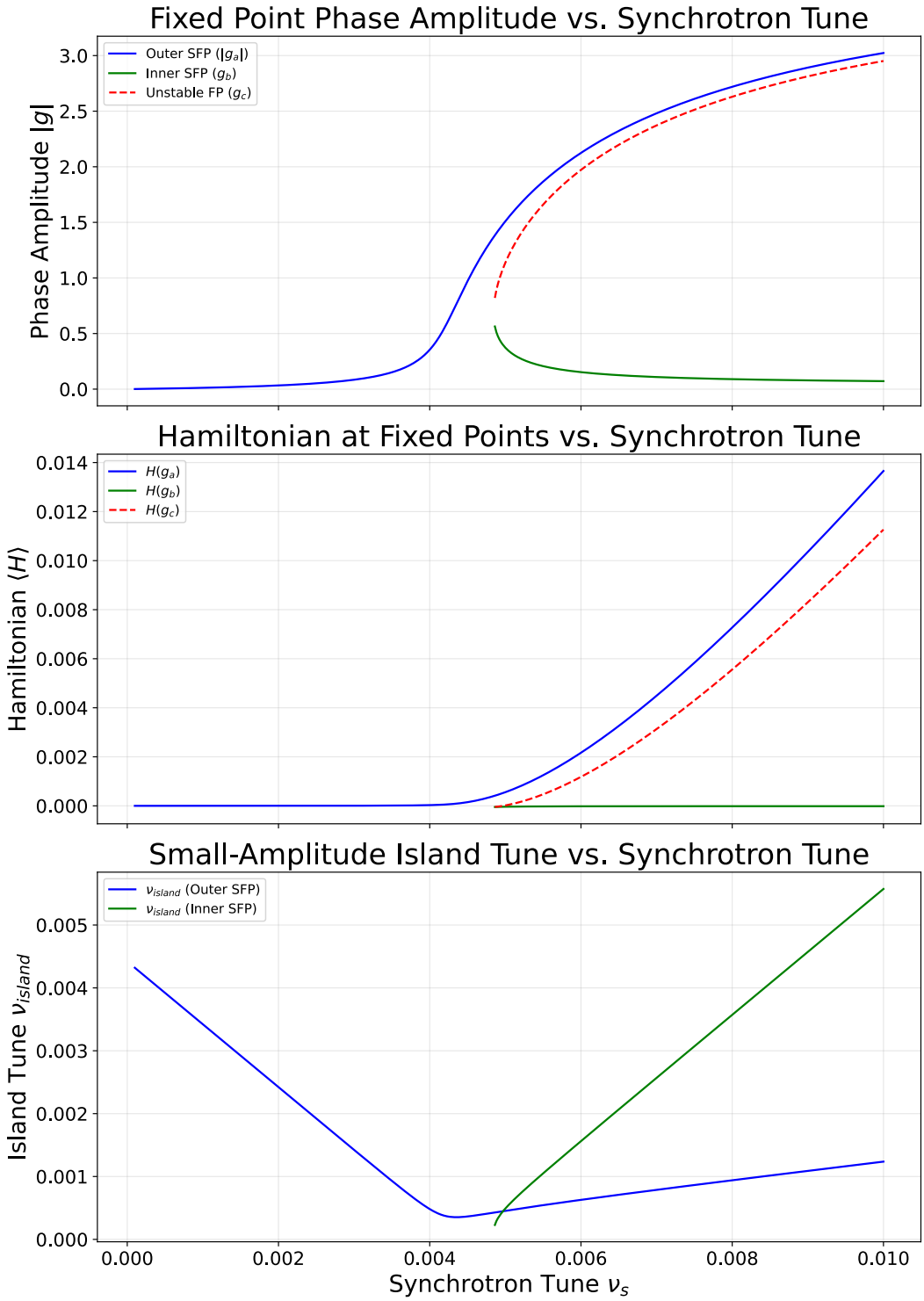

Figure 1: Top: observe the distance of the island formation from the centroid. Middle: The value of the time averaged Hamiltonian (Eq. 28). For values past the bifurcation tune marked in red, the time averaged Hamiltonian becomes non-zero, representing fixed point formation outside the centroid. Bottom: refers to the island tune, defined as the rate of oscillations of the particle bunch around a certain island.

Having confirmed the formation of resonance islands can be simulated in BLonD for one of the island formation conditions, phase modulation was introduced to test the second mechanism. Given a certain modulation amplitude, frequency, and the properties of the RF Station, periodic sinusoidal modulation is introduced to the RF Station Object. The 3rd Order Hamiltonian can be modeled by Equ. 28 for N=3. Fig. 1 models the bifurcation tune for this Hamiltonian, which defined the tune regime at which island formation exists. Prior to island formation, the bunch oscillates about the central fixed point. Characterizing the bifurcation establishes a baseline for the phase modulation studies and enables the relationship between synchrotron tune, modulation frequency, and modulation amplitude required for island formation to be investigated.

Given a certain modulation amplitude, frequency, and the properties of the RF Station, periodic sinusoidal modulation is introduced. The station is first tested for a $Q_s = 4.42 \times 10^{-3}$ and $V = 2,838$ kV.

Parameter	**Units**	**Value**
Ring Circumference, C	[m]	3833.578
Particle Type	[#]	Proton
Harmonic Number, h	[#]	315
Design Energy, p_i	[GeV]	23.5
Voltage, V	[kV]	2838
Initial RMS Momentum Spread, σ_δ	[#]	1×10^{-3}
Initial RMS Bunch Length, σ_s	[m]	0.50
Modulation Amplitude, a	[#]	0.08*
Synchrotron Tune, Q_s	[#]	4.42×10^{-3}*

Table 1: Machine parameters for initial modulation study. Asterisk indicates unless noted differently in text.

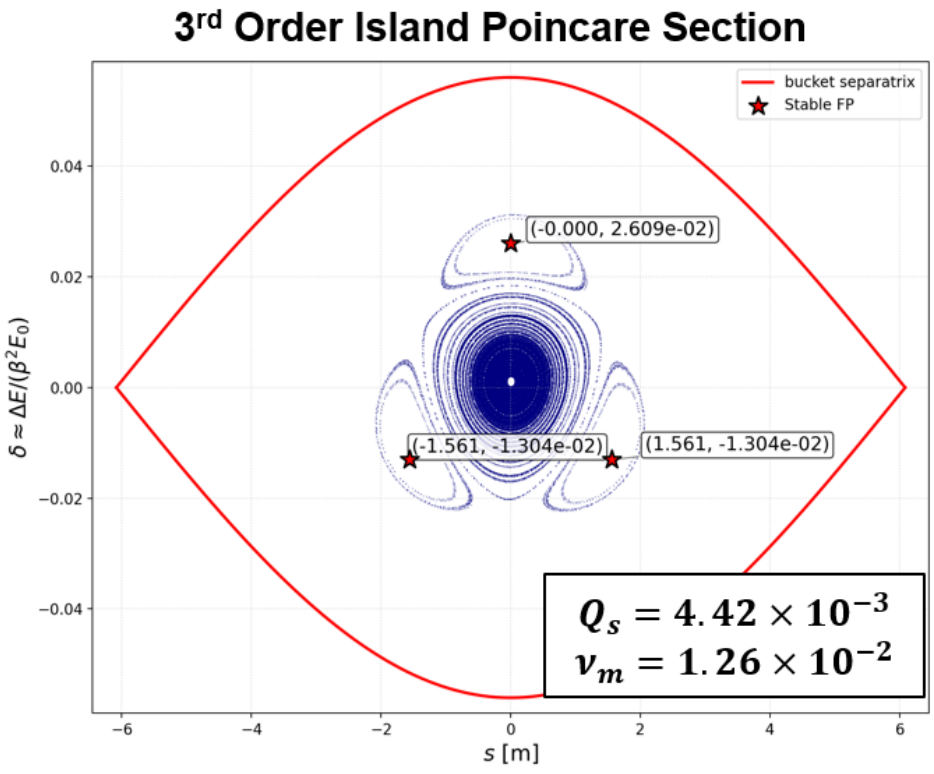

Figure 2: 3rd order resonance with analytic fixed points labeled.

The 3rd order resonance Poincaré contour is shown in Fig. 2 where the modulation tune, $\nu_m = 1.26 \times 10^{-2}$, is

$2.85 \times Q_s$. The red curve indicates the separatrix calculated using Equ. 3. The stable fixed points are plotted by evaluating Equ. 28 using the Hessian matrix to validate stability and the roots of the Hamiltonian for the conjugate pair of coordinates.

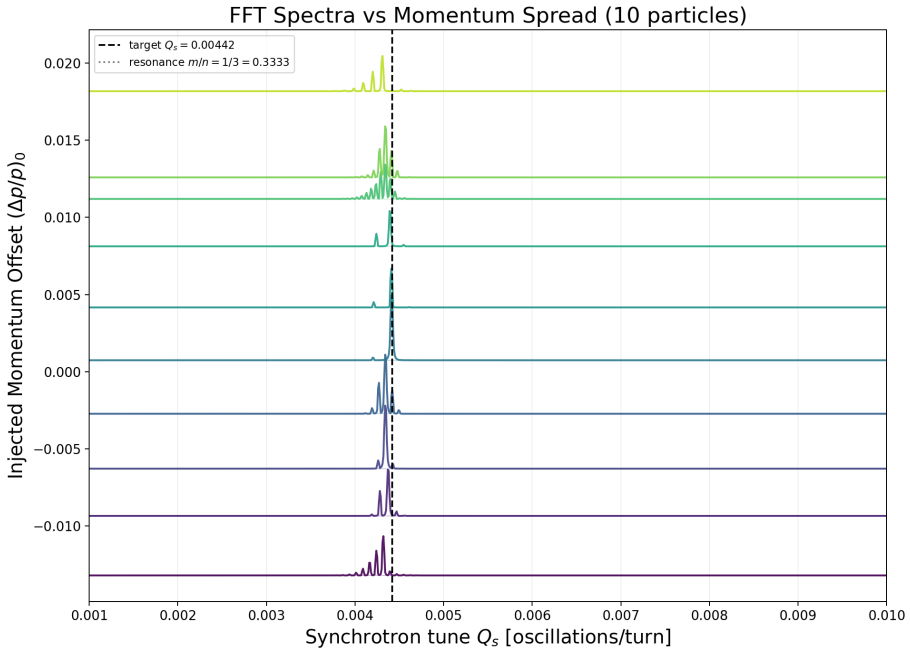

Figure 3: DFT Spectral Plot for the 3rd order resonance island, displaying a synchrotron tune close to $Q_s = 4.42 \times 10^{-3}$

Figure 3 depicts the Q_s signal for the modulation study, where the primary bucket is set to a synchrotron tune of 4.42×10^{-2}. This is the DFT scan for the 3rd order resonance. Not pictured is the modulation tune, which will also produce a signal in the DFT around the modulation amplitude.

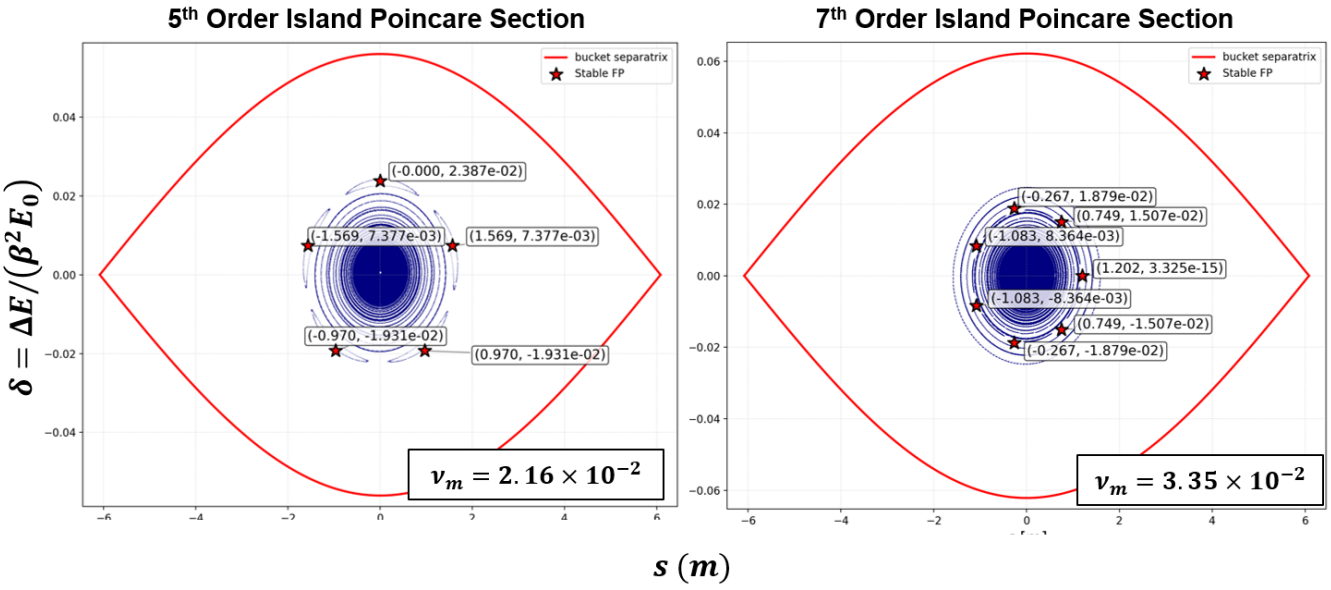

Figure 4: 5th and 7th order island formation for $Q_s = 4.42 \times 10^{-2}$

Figure 4 depicts the Poincaré sections of the 5^{th} and 7^{th} order islands. It is notable that as the simulation reached higher order resonances, the amplitude of islands in phase space decreased despite no change in the modulation amplitude throughout the simulation. At higher orders, the island center prediction given by Equ. 28 is seen to become less fit at higher order resonances.

Following the higher tune study, the Q_s was lowered to $Q_s = 2 \times 10^{-4}$, with a voltage of $V = 41kV$ to model conditions that could be used in an experiment.

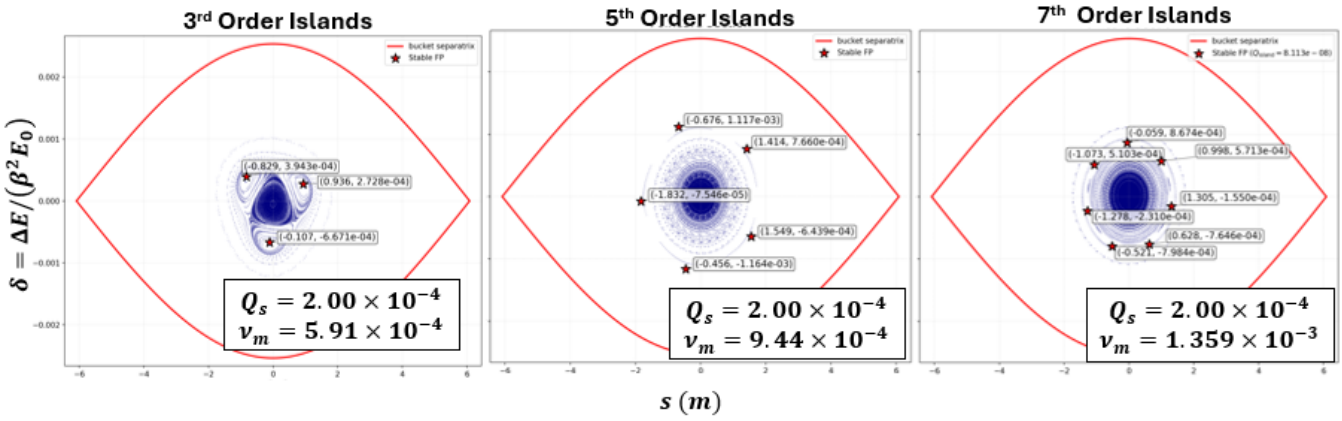

Figure 5: From left to right: 3^{rd}, 5^{th}, and 7^{th} Order Island Formation at $Q_s = 2 \times 10^{-4}$

.

To lower the required voltage for the synchrotron tune, the requested tune is reduced to 4.5%, or $Q_s = 2 \times 10^{-4}$. At this tune, the modulation strength was also reduced. This meant that the modulation amplitude was increased from $a = 0.08$ to $a = 0.1$, the particle count was increased from 200 to 300, and the turn count was doubled from 100,000 to 200,000. This allowed for island formation to occur under a weaker modulation, as a lower synchrotron tune would require more synchrotron periods to develop islands. The time taken for a full simulation without monitoring the turn by turn snapshot doubled from roughly three minutes to seven minutes on average.

4 Conclusion

The generation of resonance islands from phase modulation using the BLonD accelerator code was presented. The theoretical framework was proven correct by the generation of resonance islands production near NQ_s where N is the resonance order in all cases. As the resonance order increased to 5 and above, the islands started to flattened and the calculation of the fixed points started to deviate from roots of the Hamiltonian which was used for the lower resonance orders.

5 Acknowledgments

We thank the BNL collider accelerator department for their patience in reviewing this project. We thank the BLonD development community. The Gemini LLM was used to proofread the document.

A Truncated Program

```
import os, sys, csv, math
from dataclasses import dataclass, asdict, replace
from typing import Optional
from pathlib import Path

# BLonD and Math imports
import numpy as np
from scipy.constants import c
from scipy.optimize import fsolve
import matplotlib; matplotlib.use("Agg")
import matplotlib.pyplot as plt

# disables GPU manually in case GPU tries to run
for mod in ["cupy", "blond.gpu", "blond.gpu.__init__"]:
    sys.modules.setdefault(mod, None)

from blond.beam.beam import Beam, Proton
from blond.input_parameters.rf_parameters import RFStation
from blond.input_parameters.ring import Ring
from blond.trackers.tracker import RingAndRFTracker
from blond.llrf.rf_modulation import PhaseModulation

# config--------------------------------------------------------------
@dataclass
class Config:
    OUTPUT_DIR: str = "output" #output path, creates a new folder in the current directory
    CIRCUMFERENCE: float = 3833.587 #ring circumference
    PROTON_MASS_EV: float = 0.9382720813e9 #proton mass in eV
    KINETIC_ENERGY_EV: float = 23.5e9 #Kinetic energy gained each rev
    TOTAL_ENERGY_EV: float = KINETIC_ENERGY_EV + PROTON_MASS_EV #total energy used for momentum
    ↪  calculation
    MOMENTUM_EV: float = np.sqrt(TOTAL_ENERGY_EV**2 - PROTON_MASS_EV**2) #momentum gained
    GAMMA_L: float = np.sqrt(1.0 + (MOMENTUM_EV / PROTON_MASS_EV)**2) #relativistic gamma factor
    HARMONIC: int = 315 #harmonic of the bucket
    GAMMA_TR: float = 23.511 #transition gamma
    ETA: float = (1 / GAMMA_TR**2) - (1 / GAMMA_L**2) #slippage factor
    TARGET_QS: float = 0.0002 #Synchronous tune

    N_TURNS: int = 200000 #number of turns
    N_P: int = 201 #number of particles


    BEAM_DISTRIBUTION: str = "twiss" #distribution, set to "twiss or momentum_scan"
    BEAM_SEED: int = 1 #random generation seed
    S0_M: float = 0 # Beam center offset in s
    BEAM_CENTROID_DP: float = 0 #beam center offset in dp
    CENTER_ON_RF_FIXED_POINT: bool = False #force centered beam
    TWISS_SIGMA_S_M: float = 0.50 #spread of beam in s
    TWISS_SIGMA_DP: float = 0.1e-3 #spread of beam in delta
    TWISS_ALPHA: float = 0.0 #tilt

    #momentum scan (particles initialized individually with momentum offsets to span the height
    ↪  of the bucket)
```

```
    BUCKET_EDGE_FRACTION: float = 0.985 # span is a certain fraction of the full bucket height,
    ↪  this is how close it will get to the edge
    DELTA_MIN: float = -0.020 #set custom delta min/max
    DELTA_MAX: float = 0.020

    RESONANCE_NUMERATOR_M: int = 1 #reccomended to use 1
    RESONANCE_ORDER_N: int = 3 #3, 5, 7

    POINCARE_START_TURN: int = 100000 #plotting start turn, reccomended to use 1/2 of total
    ↪  turns
    MASK_OUTSIDE_IN_PLOTS: bool = True #disables/enables particle removal at separatrix

    ENABLE_PHASE_MODULATION: bool = True #phase modulation
    MODULATION_TUNE: float = 0.000591 #modulation tune
    MODULATION_AMPLITUDE_RAD: float = 0.1 #modulation amplitude
    MODULATION_OFFSET_RAD: float = 0.0 #
    MODULATION_MULTIPLIER: float = 1.0 #
    MODULATE_RF_FREQUENCY: bool = True #set to true
    FP_SCAN: bool = True # compares island formation to the analytic model.
    #Can break and plot unrealistic points if turned on when no island formaiton is observed

    RUN_MODULATION_TUNE_SCAN: bool = False # Runs through a variety of modulation tunes to find
    ↪  island formation
    MODULATION_TUNE_SCAN_MIN: float = 0.0005 #lowest modulation tune
    MODULATION_TUNE_SCAN_MAX: float = 0.0015 #highest modulation tune
    MODULATION_TUNE_SCAN_POINTS: int = 100 #number of modulation tunes sampled
    MODULATION_SCAN_TURNS: int = N_TURNS #number of turns sampled
    MODULATION_SCAN_START_TURN: int = 1 #for quicker scans if needed
    MODULATION_SCAN_SUBDIR: str = "modulation_tune_scan" #name of mod_scan subfolder

    PLOT_DPI: int = 180 #figure resolution
    @property
    def target_resonance_tune(self):
        return self.RESONANCE_NUMERATOR_M / self.RESONANCE_ORDER_N
    @property
    def output_dir(self): return self.OUTPUT_DIR

# utilities-------------------------------------------------------------------------

def to_1d(x, dtype=float) -> np.ndarray:
    #ravels arrays
    return np.asarray(x.get() if hasattr(x, "get") else x, dtype=dtype).ravel()

def scalar_at(x, i: int) -> float:
    #converts arrays to scalars
    a = to_1d(x)
    if a.size == 0: raise ValueError("Empty array for scalar_at.")
    return float(a[min(max(int(i), 0), a.size - 1)])

def wrap_to_pi(x):
    return (np.asarray(x, dtype=float) + np.pi) % (2.0 * np.pi) - np.pi

def alloc_hist(n_turns: int, n_particles: int, dtype=float, fill=np.nan) -> np.ndarray:
    return np.full((n_turns + 1, n_particles), fill, dtype=dtype)
```

```
def check_finite(arr: np.ndarray) -> np.ndarray:
    return np.isfinite(to_1d(arr))

def validate_history(name: str, arr: np.ndarray, n_particles: Optional[int] = None) ->
↪  np.ndarray:
    a = np.asarray(arr, dtype=float)
    if a.ndim != 2: raise ValueError(f"{name} must have shape [turns, particles].")
    if n_particles and a.shape[1] != n_particles: raise ValueError(f"{name} shape mismatch.")
    return a

def write_csv(path: str, rows: list, fieldnames: list = None):
    Path(path).parent.mkdir(parents=True, exist_ok=True)
    with open(path, "w", newline="", encoding="utf-8") as f:
        if isinstance(rows[0], dict):
            writer = csv.DictWriter(f, fieldnames=fieldnames or rows[0].keys())
            writer.writeheader(); writer.writerows(rows)
        else:
            writer = csv.writer(f)
            if fieldnames: writer.writerow(fieldnames)
            writer.writerows(rows)

def setup_plot(figsize, xlabel, ylabel, title=None, dpi=None):
    plt.rcParams.update({"font.size": 14, "axes.labelsize": 16, "axes.titlesize": 18,
    ↪  "legend.fontsize": 12})
    fig, ax = plt.subplots(figsize=figsize, dpi=dpi)
    ax.set_xlabel(xlabel); ax.set_ylabel(ylabel)
    if title: ax.set_title(title)
    return fig, ax

@dataclass
class DistributionResult:
    mode: str; n_particles: int; centroid_dt: float; centroid_dE: float
    sigma_dt: float; sigma_dE: float; injected_delta: np.ndarray = None; inside0: np.ndarray =
    ↪  None

@dataclass(frozen=True)
class FFTScanResult:
    injected_delta: np.ndarray; tune: np.ndarray; peak_power: np.ndarray; signal_rms: np.ndarray
    valid: np.ndarray; tune_axis: np.ndarray; spectra: np.ndarray
    plot_path: Path; csv_path: Path; npz_path: Path

# machine and bucket ------------------------------------------------------------
class Machine:
    #Machine class handling the general set up of the ring, rf station, beam, and tracker
    def __init__(self, cfg):
        self.cfg = cfg
        self.proton = Proton()
        self.mass_ev = float(self.proton.mass)
        self.momentum_ev = float(cfg.MOMENTUM_EV)
        self.energy_ev = math.sqrt(self.momentum_ev**2 + self.mass_ev**2)
        self.gamma = self.energy_ev / self.mass_ev
        self.beta = self.momentum_ev / self.energy_ev
        self.c = c
        self.alpha_0 = 1.0 / float(cfg.GAMMA_TR)**2
        self.eta_0 = self.alpha_0 - 1.0 / self.gamma**2
```

```
        self.rf_phase_base = 0.0 if self.eta_0 < 0.0 else math.pi
        focusing_term = abs(self.eta_0 * math.cos(self.rf_phase_base))
        if focusing_term <= 0.0: raise ValueError("No longitudinal focusing.")

        beta2_energy = self.beta**2 * self.energy_ev
        self.voltage = (2.0 * math.pi * beta2_energy * float(cfg.TARGET_QS)**2) /
        ↪  (float(cfg.HARMONIC) * focusing_term)

    def print_summary(self):
        print(f"\nMACHINE SUMMARY\nmomentum [GeV/c]: {self.momentum_ev / 1.0e9:.8f}")
        print(f"voltage [MV]: {self.voltage / 1.0e6:.8f}\nTARGET_QS: {self.cfg.TARGET_QS:.8f}")

    def build(self, n_macroparticles=None, n_turns=None):
        cfg = self.cfg
        np_turns = int(n_turns or cfg.N_TURNS)
        ring = Ring(cfg.CIRCUMFERENCE, self.alpha_0, np.full(np_turns+1, self.momentum_ev),
        ↪  self.proton, np_turns)
        rf = RFStation(ring, [cfg.HARMONIC], [np.full(np_turns+1, self.voltage)],
        ↪  [np.full(np_turns+1, self.rf_phase_base)],
                       phi_modulation=build_phase_modulation(cfg, ring))
        beam = Beam(ring, int(n_macroparticles or cfg.N_P), float(getattr(cfg, "INTENSITY",
        ↪  1.0)))
        return ring, rf, beam, RingAndRFTracker(rf, beam)

class Bucket:
    #bucket class holding the separatrix calculations
    def __init__(self, machine):
        self.m = machine
        self.beta2_energy = machine.beta**2 * machine.energy_ev
        self.beta_c = machine.beta * machine.c
        self.momentum_app = (self.beta2_energy * machine.voltage) / (2.0 * np.pi *
        ↪  float(self.m.cfg.HARMONIC) * abs(machine.eta_0))
        self.H_sep = 2.0 * self.momentum_app

    def delta_to_dE(self, delta): return self.beta2_energy * np.asarray(delta, dtype=float)
    def dE_to_delta(self, dE): return np.asarray(dE, dtype=float) / self.beta2_energy
    def dt_to_s(self, dt): return self.beta_c * np.asarray(dt, dtype=float)
    def s_to_dt(self, s): return np.asarray(s, dtype=float) / self.beta_c
    def s_to_q(self, s, omega_rf): return float(omega_rf) * np.asarray(s, dtype=float) /
    ↪  self.beta_c
    def q_to_s(self, q, omega_rf): return self.beta_c * np.asarray(q, dtype=float) /
    ↪  float(omega_rf)

    def instantaneous_q(self, dt, omega_rf, phi_rf):
        return wrap_to_pi(float(omega_rf) * np.asarray(dt, dtype=float) + float(phi_rf) -
        ↪  self.m.rf_phase_base)
    def hamiltonian(self, q, dE):
        return 0.5 * np.asarray(dE, dtype=float)**2 + self.momentum_app * (1.0 -
        ↪  np.cos(wrap_to_pi(q)))
    def inside_instantaneous(self, dt, dE, omega_rf, phi_rf, safety=None):
        safety = safety or (1.0e-8 * abs(self.H_sep))
        return self.hamiltonian(self.instantaneous_q(dt, omega_rf, phi_rf), dE) <= self.H_sep +
        ↪  safety
    def inside(self, s, dE, omega_rf, safety=None):
```

```
        return self.hamiltonian(self.s_to_q(s, omega_rf), dE) <= self.H_sep + (safety or 1.0e-8
        ↪  * abs(self.H_sep))
    def dE_limit_at_q(self, q):
        dE_sq = 2.0 * (self.H_sep - self.momentum_app * (1.0 - np.cos(wrap_to_pi(q))))
        return np.sqrt(dE_sq, where=dE_sq >= 0, out=np.full_like(dE_sq, np.nan, dtype=float))
    def dE_limit_at_s(self, s, omega_rf):
        q = self.s_to_q(s=s, omega_rf=omega_rf,)
        return self.dE_limit_at_q(q)
    def make_separatrix_s_delta(self, omega_rf, n_points=4000):
        q = np.linspace(-np.pi, np.pi, int(n_points))
        dE_plus = self.dE_limit_at_q(q)
        return self.q_to_s(q, omega_rf), self.dE_to_delta(dE_plus), self.dE_to_delta(-dE_plus)
    def read_coords(self, beam):
        dt, dE = to_1d(beam.dt), to_1d(beam.dE)
        return dt, dE, self.dt_to_s(dt), self.dE_to_delta(dE)

# beam distributions ------------------------------------------------------------

def beam_statistics(beam, mode: str) -> DistributionResult:
    #converts inputs into BLonD coordinates
    dt, dE = to_1d(beam.dt), to_1d(beam.dE)
    return DistributionResult(mode, dt.size, np.mean(dt) if dt.size else np.nan, np.mean(dE) if
    ↪  dE.size else np.nan,
                              np.std(dt) if dt.size else np.nan, np.std(dE) if dE.size else
                              ↪  np.nan)

def calculate_twiss(x, y):
    #twiss parameters used in twiss generation
    x, y = to_1d(x), to_1d(y)
    if x.size == 0 or x.shape != y.shape: return (np.nan,) * 7
    mx, my = np.mean(x), np.mean(y)
    xc, yc = x - mx, y - my
    s11, s22, s12 = np.mean(xc**2), np.mean(yc**2), np.mean(xc*yc)
    det = max(0.0, s11 * s22 - s12**2)
    eps = np.sqrt(det)
    return (eps, -s12/eps if eps else 0.0, s11/eps if eps else 0.0, mx, my, np.sqrt(s22),
    ↪  np.sqrt(s11))

def initialize_distribution(cfg, beam, ring, rf, machine, bucket):
    #creates beam
    mode = str(cfg.BEAM_DISTRIBUTION).strip().lower()
    if mode in ["momentum_scan", "scan"]:
        omega0 = scalar_at(rf.omega_rf[0], 0)
        dE_lim = bucket.dE_limit_at_s(cfg.S0_M, omega0)
        dE_values = bucket.delta_to_dE(np.linspace(cfg.DELTA_MIN, cfg.DELTA_MAX, cfg.N_P)) if
        ↪  cfg.MOMENTUM_SCAN_MODE == "manual_delta" else \
                    np.linspace(-cfg.BUCKET_EDGE_FRACTION * dE_lim, cfg.BUCKET_EDGE_FRACTION *
                    ↪  dE_lim, cfg.N_P)
        beam.dt = np.full(cfg.N_P, cfg.S0_M / (machine.beta * c))
        beam.dE = dE_values
    elif mode == "twiss":
        beta_rel = scalar_at(ring.beta, 0)
        energy_eV = scalar_at(ring.energy, 0)
        rng = np.random.default_rng(getattr(cfg, "BEAM_SEED", None))
```

```
        sigma_s = float(cfg.TWISS_SIGMA_S_M)
        sigma_delta = float(cfg.TWISS_SIGMA_DP)
        alpha_twiss = float(cfg.TWISS_ALPHA)

        emittance = sigma_s * sigma_delta
        beta_twiss = sigma_s / sigma_delta
        gamma_twiss = (1.0 + alpha_twiss**2) / beta_twiss

        target_covariance = emittance * np.array(
            [
                [beta_twiss, -alpha_twiss],
                [-alpha_twiss, gamma_twiss],
            ],
            dtype=float,
        )

        # Generate finite Gaussian sample
        normalized = rng.standard_normal((2, cfg.N_P))
        # Exact zero centroid
        normalized -= normalized.mean(axis=1, keepdims=True)

        # Whiten finite sample
        sample_covariance = normalized @ normalized.T / cfg.N_P
        eigvals, eigvecs = np.linalg.eigh(sample_covariance)

        if np.any(eigvals <= 0):
            raise RuntimeError("Generated Gaussian sample is singular.")

        whitening = (eigvecs @ np.diag(1.0 / np.sqrt(eigvals))  @ eigvecs.T)
        normalized = whitening @ normalized

        # Apply requested Twiss covariance
        twiss_transform = np.linalg.cholesky(target_covariance)
        coords = twiss_transform @ normalized

        s = coords[0] + float(cfg.BEAM_CENTROID_S_M)
        delta = coords[1] + float(cfg.BEAM_CENTROID_DP)

        dt_offset = 0.0
        beam.dt = s / (beta_rel * c) + dt_offset
        beam.dE = delta * energy_eV * beta_rel**2
        print( "[beam_distribution] initialized Twiss Gaussian\n"
            f"  sigma_s      = {np.std(s):.6e} m\n"
            f"  sigma_delta  = {np.std(delta):.6e}\n"
            f"  alpha        = {alpha_twiss:.6e}\n"
            f"  emittance    = {emittance:.6e}")
    else: raise ValueError(f"Unknown distribution: {mode}")

# modulation ------------------------------------------------------------------
def build_phase_modulation(cfg, ring):
    #building modulation fed into BLonD PhaseModulation
    if not getattr(cfg, "ENABLE_PHASE_MODULATION", False): return None
    t_rev = to_1d(ring.t_rev)
    f_mod = cfg.MODULATION_TUNE * (1.0 / t_rev[0])
```

```
    return PhaseModulation(timebase=to_1d(ring.cycle_time), frequency=f_mod,
    ↪  amplitude=cfg.MODULATION_AMPLITUDE_RAD,
                           offset=cfg.MODULATION_OFFSET_RAD, harmonic=cfg.HARMONIC,
                           ↪  multiplier=cfg.MODULATION_MULTIPLIER,
                           modulate_frequency=cfg.MODULATE_RF_FREQUENCY)

def run_modulation_tune_scan(cfg, run_single_case):
    #scanning through different modulation tunes
    scan_dir = os.path.join(cfg.output_dir, cfg.MODULATION_SCAN_SUBDIR,)
    os.makedirs(scan_dir, exist_ok=True)

    #what tunes to scan though
    tunes = np.linspace(cfg.MODULATION_TUNE_SCAN_MIN, cfg.MODULATION_TUNE_SCAN_MAX,
    ↪  cfg.MODULATION_TUNE_SCAN_POINTS,)
    rows = []
    #makes one instance and then loops through the tunes
    for index, tune in enumerate(tunes):
        case_dir = os.path.join(
            scan_dir,
            f"qm_{tune:.6f}",
        )

        case_cfg = replace(
            cfg,
            OUTPUT_DIR=case_dir,
            MODULATION_TUNE=float(tune),
            N_TURNS=int(cfg.MODULATION_SCAN_TURNS),
            SAVE_SNAPSHOTS=False,
        )

        os.makedirs(case_dir, exist_ok=True)

        print( f"[modulation scan {index + 1}/{len(tunes)}] "  f"Qm = {tune:.6f}")

        result = run_single_case(case_cfg)

        s_history = np.asarray(result["s_history"],dtype=float,)
        delta_history = np.asarray(result["delta_history"], dtype=float,)

        surviving_particles = int(np.sum(np.isfinite(s_history[-1]) &
        ↪  np.isfinite(delta_history[-1]) ) )

        row = {"modulation_tune": float(tune), "surviving_particles":
        ↪  surviving_particles,"case_dir": case_dir,}
        rows.append(row)

    csv_path = os.path.join(scan_dir, "modulation_tune_scan.csv",)

    with open(csv_path, "w", newline="") as output:
        writer = csv.DictWriter(output, fieldnames=rows[0].keys(),)
        writer.writeheader()
        writer.writerows(rows)

    return rows
```

```
# plotting ------------------------------------------------------------------

def _inside_limits(sep_s, sep_plus, sep_minus):
    #checks if a particle is inside the separatrix boundary
    s_vals, y_vals = sep_s[check_finite(sep_s)],
    ↪  np.concatenate([sep_plus[check_finite(sep_plus)], sep_minus[check_finite(sep_minus)]])
    if not s_vals.size: return (-1.0, 1.0), (-1.0, 1.0)
    sp, yp = 0.03 * max(s_vals.max() - s_vals.min(), 1e-15), 0.05 * max(np.max(np.abs(y_vals)),
    ↪  1e-15)
    return (s_vals.min() - sp, s_vals.max() + sp), (-y_vals.max() - yp, y_vals.max() + yp)

def plot_poincare(case_dir, p_s, p_delta, sep_s, sep_plus, sep_minus, cfg, fixed_points=None):
    #poincare plotting
    fig, ax = setup_plot((10, 8), r"$s$ [m]", r"$\delta \approx \Delta E/(\beta^2 E_0)$",
    ↪  "Poincare section", cfg.PLOT_DPI)
    ax.plot(sep_s, sep_plus, 'r', linewidth=2.2, label="separatrix"); ax.plot(sep_s, sep_minus,
    ↪  'r', linewidth=2.2)
    X = np.concatenate([s[check_finite(s)] for s in p_s]) if p_s else []
    Y = np.concatenate([d[check_finite(d)] for d in p_delta]) if p_delta else []
    if len(X): ax.scatter(X, Y, s=0.7, c="navy", alpha=0.6, linewidths=0, rasterized=True)
    stable_fps = [fp for fp in fixed_points if fp["kind"] == "sfp"]

    if stable_fps:
        ax.scatter([f["s"] for f in stable_fps], [f["dp"] for f in stable_fps],
                marker="*", s=150, c="red", edgecolors="k", label="Stable FP")
        for i, f in enumerate(stable_fps):
            ax.annotate(f"({f['s']:.3f}, {f['dp']:.4g})", (f["s"], f["dp"]),
                        xytext=(8, 12 + 16*i), textcoords="offset points",
                        fontsize=14, color="black",
                        bbox=dict(boxstyle="round,pad=0.2", facecolor="white",
                        ↪  edgecolor="black", alpha = 0.7))
    ax.set_xlim(*_inside_limits(sep_s, sep_plus, sep_minus)[0]);
    ↪  ax.set_ylim(*_inside_limits(sep_s, sep_plus, sep_minus)[1])
    ax.grid(True, linestyle=":", alpha=0.5); ax.legend(loc="best")
    fig.tight_layout(); fig.savefig(os.path.join(case_dir,
    ↪  f"poincare_n{cfg.RESONANCE_ORDER_N}.png")); plt.close(fig)

def phi_rf_vs_turn(turns, values, case_dir):
    #plots rf phase to check if modulation is working
    plt.figure(figsize=(8,4))
    plt.plot(turns, values, marker="o")
    plt.xlabel("Turn")
    plt.ylabel(r"$\phi_{\rm rf}$ [rad]")
    plt.grid(True)
    plt.tight_layout()
    plt.savefig(os.path.join(case_dir, "phi_rf_vs_turn.png"), dpi=200)
    plt.close()

# fixed points --------------------------------------------------------------
def coeff_from_elliptic(N: int) -> float:
    coeffs = {
        1: 1.0,
        3: 1/64,
        5: 1/4096,
        7: 1/262144,
```

```
    }

    if N in coeffs:
        return coeffs[N]

    if N % 2:
        raise ValueError(f"Unsupported resonance order N={N}")


def analytic_derivs(cfg, J: float, psi: float):
    nu_s = cfg.TARGET_QS
    nu_m = cfg.MODULATION_TUNE
    a = cfg.MODULATION_AMPLITUDE_RAD
    N = cfg.RESONANCE_ORDER_N

    K = coeff_from_elliptic(N)

    c = np.cos(N * psi)
    s = np.sin(N * psi)

    dHdJ = ((nu_s - nu_m / N)- (nu_s / 8.0) * J- nu_s * a * K * N * (2 * J) ** (N / 2 - 1) * c)
    dHdpsi = (nu_s * a * K * N * (2 * J) ** (N / 2) * s)
    d2HdJ2 = (-nu_s / 8.0 - nu_s * a * K * N * (N - 2) * (2 * J) ** (N / 2 - 2) * c)
    d2Hdpsi2 = (nu_s * a * K * N**2 * (2 * J) ** (N / 2) * c)
    d2HdJdpsi = (nu_s * a * K * N**2 * (2 * J) ** (N / 2 - 1) * s)

    return dHdJ, dHdpsi, d2HdJ2, d2Hdpsi2, d2HdJdpsi

def island_tune(HJJ, Hpp, HJp):
    disc = HJJ * Hpp - HJp**2
    if disc < 0:
        return None
    return np.sqrt(disc)
def _angle_diff(a, b):
    return np.arctan2(np.sin(a - b), np.cos(a - b))
def action_angle_to_s_dp(J, psi_res, cfg, machine):
    N = cfg.RESONANCE_ORDER_N

    theta = 2 * np.pi * cfg.POINCARE_START_TURN
    psi = psi_res + cfg.MODULATION_TUNE * theta / N - np.pi / (2 * N)

    A = np.sqrt(2 * J)

    phi = A * np.cos(psi)
    delta = -A * np.sin(psi)

    s = cfg.CIRCUMFERENCE * phi / (2 * np.pi * cfg.HARMONIC)
    dp = delta * cfg.TARGET_QS / (cfg.HARMONIC * machine.eta_0)

    return s, dp

def solve_fixed_points(cfg, machine, J_max=8.0, verbose=False):
    found = []

    guesses = [
```

```
    (u0, psi0)
    for u0 in np.linspace(np.sqrt(0.05), np.sqrt(J_max), 10)
    for psi0 in np.linspace(0.0, 2*np.pi, 36, endpoint=False)
]

def F(x):
    u, psi = x
    J = u*u
    return analytic_derivs(cfg, J, psi)[:2]

for guess in guesses:
    root, info, ier, msg = fsolve(F,x0=guess,full_output=True,xtol=1e-12,)
    if ier != 1:
        continue

    u, psi = root
    J = u*u
    psi = np.mod(psi, 2*np.pi)

    dHdJ, dHdpsi, HJJ, Hpp, HJp = analytic_derivs(cfg, J, psi)

    # Verify convergence
    if abs(dHdJ) > 1e-10 or abs(dHdpsi) > 1e-10:
        continue

    # Remove duplicates
    if any(abs(J - fp["J_tilde"]) < 1e-8 and abs(_angle_diff(psi, fp["psi_tilde"])) < 1e-8
    ↪  for fp in found):
        continue

    # Hamiltonian linearization
    A = np.array([[ HJp,  Hpp], [-HJJ, -HJp],])

    eig = np.linalg.eigvals(A)
    stable = np.all(np.abs(np.real(eig)) < 1e-12)

    kind = "sfp" if stable else "ufp"
    s, dp = action_angle_to_s_dp(J, psi, cfg, machine)
    if verbose:
        print(
            f"J={J:.6f}  "
            f"psi={np.degrees(psi):8.3f}°  "
            f"{kind}  "
            f"eig={eig}"
        )
    found.append({
        "J_tilde": J,
        "psi_tilde": psi,
        "s": s,
        "dp": dp,
        "kind": kind,
        "HJJ": HJJ,
        "Hpp": Hpp,
        "HJp": HJp,
        "eig": eig,
```

```
        })

    return found

# Run Loop ------------------------------------------------------------------

def run_scan(cfg, machine, bucket):
    c_name = f"qm_{cfg.MODULATION_TUNE:.6e}_s0_{cfg.S0_M:+.4e}_n{cfg.RESONANCE_ORDER_N}_m{cfg.R
    ↪  ESONANCE_NUMERATOR_M}_Qs{cfg.TARGET_QS:.6e}"
    c_dir = Path(cfg.output_dir) / c_name
    c_dir.mkdir(parents=True, exist_ok=True)

    ring, rf, beam, tracker = machine.build()
    initialize_distribution(cfg, beam, ring, rf, machine, bucket)

    _, _, init_s, init_delta = bucket.read_coords(beam)
    twiss_res = calculate_twiss(init_s, init_delta)
    sep_s, sep_plus, sep_minus = bucket.make_separatrix_s_delta(scalar_at(rf.omega_rf[0], 0))

    n_p = cfg.N_P
    h_s = alloc_hist(cfg.N_TURNS, n_p)
    h_d = alloc_hist(cfg.N_TURNS, n_p)
    h_in = alloc_hist(cfg.N_TURNS, n_p, bool, False)

    poincare_s, poincare_d = [], []
    phi_turns, phi_values = [], []

    # Poincaré sampling
    period = 1.0 / cfg.MODULATION_TUNE
    next_sample = cfg.POINCARE_START_TURN

    for turn in range(cfg.N_TURNS + 1):
        dt, dE, s, delta = bucket.read_coords(beam)
        omega = scalar_at(rf.omega_rf[0], turn)
        phi_rf = scalar_at(rf.phi_rf[0], turn)

        inside = bucket.inside_instantaneous(dt, dE, omega, phi_rf)

        h_s[turn], h_d[turn], h_in[turn] = s, delta, inside

        if turn >= round(next_sample):
            mask = inside if cfg.MASK_OUTSIDE_IN_PLOTS else np.ones(n_p, dtype=bool)
            poincare_s.append(s[mask].copy())
            poincare_d.append(delta[mask].copy())
            next_sample += period

        if turn < cfg.N_TURNS:
            tracker.track()

        #for checking modulation
        if turn % 1000 == 0:
            value = scalar_at(rf.phi_rf[0], turn)
            phi_turns.append(turn)
            phi_values.append(value)
            #comment out below to stop print out of phi_rf
```

```
            print(f"turn {turn:6d}  phi_rf = {value:.15f}")

    phi_rf_vs_turn(phi_turns, phi_values, c_dir)
    if poincare_s: plot_poincare(c_dir, poincare_s, poincare_d, sep_s, sep_plus, sep_minus, cfg,
    ↪  solve_fixed_points(cfg, machine) if cfg.FP_SCAN else None)
    with open(c_dir / "config.txt", "w") as f:
        f.write("\n".join(f"{k:35s} = {v}" for k, v in asdict(cfg).items()))
    return {"case_dir": c_dir, "N_PARTICLES": n_p, "sigma_s": twiss_res[6], "sigma_delta":
    ↪  twiss_res[5]}, h_s, h_d

def run_single_case(cfg):
    #running a single instance
    os.makedirs(cfg.output_dir, exist_ok=True)

    machine = Machine(cfg)
    bucket = Bucket(machine)

    machine.print_summary()
    result, s_history, delta_history = run_scan(cfg, machine, bucket)

    return {"result": result,"s_history": s_history,"delta_history": delta_history,}

def main():
    #runs scan or indiv
    cfg = Config()

    if cfg.RUN_MODULATION_TUNE_SCAN:
        run_modulation_tune_scan(cfg,run_single_case,)
    else:
        run_single_case(cfg)

    print("\nDone.")
    print(f"Output directory: {cfg.output_dir}")

if __name__ == "__main__":
    main()
```

References

[1] Timko, H. and Albright, S. and Argyropoulos, T. and Damerau, H. and Iliakis, K. and Intelisano, L. and Karlsen-Baeck, B. E. and Karpov, I. and Lasheen, A. and Medina, L. and Quartullo, D. and Repond, J. and Vanel, A. L. and Müller, J. Esteban and Schwarz, M. and Tsapatsaris, P. and Typaldos, G., "Beam longitudinal dynamics simulation studies," *Phys. Rev. Accel. Beams*, vol. 26, p. 114602, Nov 2023.

[2] IEEE, *17th Particle Accelerator Conference*, (Piscataway, NJ), IEEE, 1998.

[3] S. Y. Lee and K. Y. Ng, "Application of a localized chaos generated by rf-phase modulations in phase-space dilution," 2011.

[4] S. Peggs, G. Robert-Demolaize, H. Lovelace III, and T. Satogata, "Tune domain behavior of single magnet lattices," *JACoW*, vol. IPAC2025, p. MOPS083, 2025.

[5] H. Huang, M. Ball, B. Brabson, J. Budnick, D. D. Caussyn, A. W. Chao, J. Collins, V. Derenchuk, S. Dutt, G. East, M. Ellison, D. Friesel, B. Hamilton, W. P. Jones, S. Y. Lee, D. Li, M. G. Minty, S. Nagaitsev, K. Y. Ng, X. Pei, A. Riabko, T. Sloan, M. Syphers, L. Teng, Y. Wang, Y. T. Yan, and P. L. Zhang, "Experimental

determination of the hamiltonian for synchrotron motion with rf phase modulation," *Phys. Rev. E*, vol. 48, pp. 4678–4688, Dec 1993.

[6] W. Gabella, J. Rosenzweig, R. Kick, and S. Peggs, "Rf voltage modulation at discrete frequencies with applications to crystal channeling extraction," tech. rep., Fermi National Accelerator Lab., Batavia, IL (United States), 05 1992.

[7] D. Li, M. Ball, B. Brabson, J. Budnick, D. Caussyn, A. Chao, V. Derenchuk, S. Dutt, G. East, M. Ellison, D. Friesel, B. Hamilton, H. Huang, W. Jones, S. Lee, J. Liu, M. Minty, K. Ng, X. Pei, A. Riabko, T. Sloan, M. Syphers, Y. Wang, Y. Yan, and P. Zhang, "Effects of rf voltage modulation on particle motion," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 364, no. 2, pp. 205–223, 1995.