tokubetsukaiseki

2026 年 6 月 9 日

1　物理モデルを必要としない逆問題推定の方法についての試行

1.1　概要

惑星の初期半径分布を $\{(R_i, P_i)\}_{i=1}^{n}$ とし、時間発展写像

$$R^f = R^f(R_i, P_i)\ ,\ P^f = P^f(R_i, P_i)$$

によって進化するようなモデルを考える。この時、最終的に得られる観測情報（「半径の谷」）を使って逆問題的に初期惑星分布の推定を行いたい。ただし、物理モデル R^f, P^f は不明であるとする。

そこで、物理モデルをニューラルネットワークで置き換えて推定を行うことにする。ただし、ただニューラルネットワークを入れただけだと柔軟すぎていくらでも fit できてしまうので、今回は定性的な情報を正則化項として導入する。

1.1.1　import

[ ]:
```
# 機械学習系
import torch
import torch.nn as nn
import torch.nn.functional as F
from itertools import product

# その他
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy.ndimage import gaussian_filter
import os
from IPython.display import display, Image

# 並列化
from joblib import Parallel, delayed
import gc
```

1.1.2 パラメタ

[ ]:
```
# R-Pプロット関係
nummesh_for_histogram = 200
histogram_Pmin = 0.8
histogram_Pmax = 100
histogram_Rmin = 1
histogram_Rmax = 10
sigma_hist_par = 0.03 #ガウシアンフィルタで平滑化する時の分散（大きいほど平滑化強い）

# Rプロット関係
num_epochs_par=3000
nummesh_for_histogram_Ronly = 30
histogram_Ronly_min = 0.7
histogram_Ronly_max = 20.0

# 学習関係
n_samples = 1000 # 初期惑星数
depth = [1,2,3]
hidden_dims = [4,8,16,32]
device = torch.device(
    "cuda" if torch.cuda.is_available()
    else "mps" if hasattr(torch.backends, "mps") and torch.backends.mps.is_available()
    else "cpu"
)
device=torch.device("cpu")
print("device =", device)
eps_cholesky = 1e-4 #初期分布生成を与える分散共分散行列をコレスキー分解した時の対角成分が >0 となるためのバッファ
```

```
device = cpu
```

[ ]:
```
if device.type == "cuda":
    from google.colab import drive
    drive.mount("/content/drive")

    base_dir = "/content/drive/MyDrive/evap_valley_inverse"
    os.makedirs(base_dir, exist_ok=True)
    os.makedirs(f"{base_dir}/data", exist_ok=True)
    os.makedirs(f"{base_dir}/fig", exist_ok=True)
    os.makedirs(f"{base_dir}/result", exist_ok=True)
else:
    os.makedirs("./data", exist_ok=True)
    os.makedirs("./fig", exist_ok=True)
    os.makedirs("./result", exist_ok=True)


from astroquery.vizier import Vizier
```

[ ]:
```
os.environ["OMP_NUM_THREADS"] = "1"
os.environ["MKL_NUM_THREADS"] = "1"
os.environ["VECLIB_MAXIMUM_THREADS"] = "1"
os.environ["NUMEXPR_NUM_THREADS"] = "1"
torch.set_num_threads(1)
try:
    torch.set_num_interop_threads(1)
except RuntimeError:
    pass
```

1.1.3 データの取得

astroquery の API を使うと CKS を持って来れるらしい。id は J/AJ/156/264 って持って来れる。KOI が惑星の識別で、実質的に欲しいのは per（周期）と、Rp の情報のみ。単位はそれぞれ日、地球半径。

[ ]:
```
#おまじない
Vizier.ROW_LIMIT=-1

#CKS-VIIを持ってくる。table1,table2,table3が入る。
#table2:steller properties
#table3:planetary parameters ←欲しいやつ！
#table4:detection statistics
tables = Vizier.get_catalogs("J/AJ/156/264")

t3 = tables["J/AJ/156/264/table3"].to_pandas()
t4 = tables["J/AJ/156/264/table4"].to_pandas()


#データの中身を見てみる。
with open("./data/table3.dat","w") as f:
    f.write(t3.to_string())
with open("./data/table4.dat","w") as f:
    f.write(t4.to_string())

#列を持ってくる
R_data = t3["Rp"]
P_data = t3["Per"]

# print(P.max(),P.min(),R.max(),R.min()) #負はない！

#NaNとか、変なデータを処理
mask = np.isfinite(R_data) & np.isfinite(P_data)
R_data=R_data[mask]
P_data=P_data[mask]
```

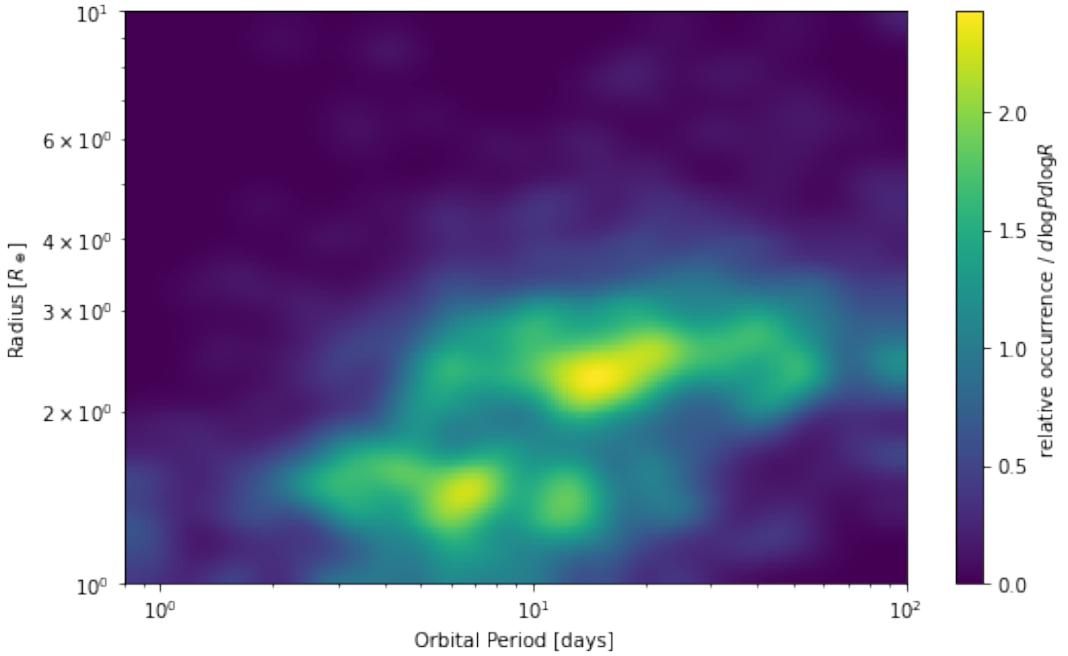
```
WARNING: UnitsWarning: Unit 'Sun' not supported by the VOUnit standard. Did you
mean uN? [astropy.units.format.vounit]
WARNING: UnitsWarning: Unit 'x' not supported by the VOUnit standard.
[astropy.units.format.vounit]
```

1.1.4 可視化

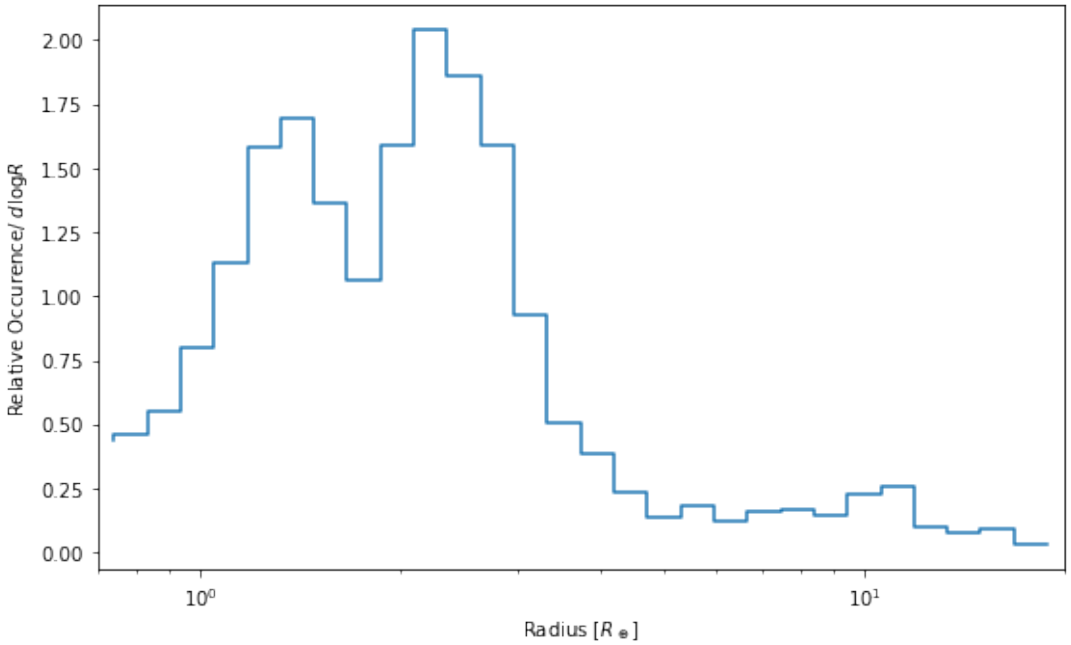
半径の谷の可視化

[ ]:
```
#histogram
#log空間の相対出現頻度ヒストグラム（個数/bin面積）
def draw_histogram_PR(R,P,filename):

    global histogram_Rmin,histogram_Rmax,histogram_Pmin,histogram_Pmax,nummesh_for_histogram,sigma_hist_par

    logR_edges=np.linspace(np.log10(histogram_Rmin),np.log10(histogram_Rmax),nummesh_for_histogram)
    logP_edges=np.linspace(np.log10(histogram_Pmin),np.log10(histogram_Pmax),nummesh_for_histogram)

    histogram,P_edges,R_edges= np.histogram2d(
        np.log10(P),np.log10(R),bins=[logP_edges,logR_edges]
        )
    area = np.diff(P_edges)[:,None]*np.diff(R_edges)[None,:]
    #print(histogram.shape) #(nummeshforhistogram-1)^2
    #print(area.shape)
    density = histogram/area
    density = density / np.sum(histogram)
    sigma_hist = sigma_hist_par * nummesh_for_histogram
    density = gaussian_filter(density, sigma=sigma_hist)


    plt.figure(figsize=(9.0, 5.4))
    plt.pcolormesh(10**P_edges,10**R_edges,density.T) #pcolormeshはなぜか(y,x)で読むらしい。
    #plt.scatter(P,R,color='red',s=0.5)
    plt.xscale("log")
    plt.yscale("log")
    plt.xlim(histogram_Pmin,histogram_Pmax)
    plt.ylim(histogram_Rmin,histogram_Rmax)
    plt.xlabel("Orbital Period [days]")
    plt.ylabel("Radius [$R_\oplus$]")
    plt.colorbar(label="relative occurrence / $d \log P d \log R$")
    plt.savefig(filename,dpi=200, bbox_inches="tight")
    plt.show()
    plt.close()
```

```
draw_histogram_PR(R_data,P_data,"./fig/CKS_evapvalley.png")
```



[ ]:
```
# R方向のヒストグラムを描く
def draw_histogram_Ronly(R,filename):

    global histogram_Ronly_min,histogram_Ronly_max,nummesh_for_histogram_Ronly

    logR_edges=np.linspace(np.log10(histogram_Ronly_min),np.log10(histogram_Ronly_max),nummesh_for_histogram_Ronly)
    histogram , R_edges = np.histogram(
        np.log10(R),bins=logR_edges
    )
    #観測個数密度に直すことを考えると、データ同化的な考え方をしなくちゃいけない。(幅がでかい bin では実際に存在するより多く見つかっていると考えるべき。)
    width = np.diff(R_edges)
    density = histogram/width
    density = density/np.sum(histogram)

    centers = 10**(0.5*(R_edges[:-1]+R_edges[1:]))
    plt.figure(figsize=(9.0,5.4))
    plt.step(centers,density)
    plt.xscale("log")
    plt.xlim(histogram_Ronly_min,histogram_Ronly_max)
    plt.xlabel("Radius [$R_\oplus$]")
    plt.ylabel("Relative Occurence/ $d\log R$")
    plt.savefig(filename,dpi=200, bbox_inches="tight")
    plt.show()
    plt.close()

draw_histogram_Ronly(R_data,"./fig/CKS_evapvalley_Ronly.png")
```



1.1.5 コード本体

やることは、$(\log R, \log P) \sim \mathcal{N}(\mu_R, \mu_P, \Sigma_{R,P})$ なる二次元正規分布を初期惑星分布として仮定し、初期分布の変数 $\vec{\mu}, \Sigma$ 及びニューラルネットの重みについて学習を行う。ただし、ニューラルネットの複雑性はあらかじめ固定し、複雑性に依存して最終結果がどう変化するか観察する。

なので大まかには次のようになる。

```
for (NN compexity) in [depth,hidden_dimention]:

    (R_i,P_i) = give_initial_distribution(mu,Sigma)
    (R_f,P_f) = R_i+evolution_func_R(W_R),P_i+evolution_func_P(W_P)

    for (lambda) in hyper_parameters
        loss = |(R_f,P_f)-(R_obs,P_obs)| + lambda * regularization_term

        initialize NN parameters
        for (epoch) in epochs:
            Machine_Learn_and_update(mu,Sigma,W_R,W_P)

            evaluate information_criterion

        if (lambda gave best information criterion):
            save lambda as lambda_best

    lambda = lambda_best
    loss = |(R_f,P_f)-(R_obs,P_obs)| + lambda * regularization_term

    initialize NN parameters
    for (epoch) in epochs:
        Machine_Learn_and_update(mu,Sigma,W_R,W_P)

        if (every 10 or so epochs):
            print (intial distribution & estimate distribution & Loss) as (video & video & graph)

    print (NN compexity & lambda_best & final initial distribution & final estimate distribution & final Loss) as (data
    ↪  & data & png & png & data)
```

また、実際に学習する上では、生成した初期分布を正規化したものを進化させる。そのため、観測データも正規化する。すなわち今、$x_i = (\log R_i, \log P_i)$ として与えられた初期条件および観測データは、いずれも

$$z_i = \frac{x_i - \overline{x}}{\sigma(x)}$$

として正規化することになる。ただし平均と分散は観測データについての平均と分散である。

■データ整形

[ ]:
```
#観測データを torch で使える形にする。

R_np= R_data.astype(float).to_numpy()
P_np= P_data.astype(float).to_numpy()
#外れ値を学習して欲しくない
mask = (
    np.isfinite(R_np)
    & np.isfinite(P_np)
    & (R_np > 0)
    & (P_np > 0)
    & (histogram_Rmin <= R_np)
    & (R_np <= histogram_Rmax)
    & (histogram_Pmin <= P_np)
    & (P_np <= histogram_Pmax)
)
R_np = R_np[mask]
P_np = P_np[mask]

#(R,P) に
x_obs_np = np.stack([
    np.log(R_np),
    np.log(P_np)
], axis=1)
x_mean_np = x_obs_np.mean(axis=0)
x_std_np = x_obs_np.std(axis=0)
z_obs_np = (x_obs_np - x_mean_np)/x_std_np
z_obs = torch.tensor(z_obs_np , dtype=torch.float32, device=device)
```

■NN complexity　基本的に、ニューラルネットを使うのは evolution_func_R,evolution_func_P の具体的な表現の部分。ここの隠れ次元数と、層の数が NN compexity ということになる。実際には、

```
depth = [1,2,3]
hidden_dims = [4,8,16,32]
```

くらいで試そうかなという感じではある。

■初期分布　初期分布布は、中心 $\vec{\mu}$, 分散共分散行列 Σ で決まる。よって、学習の仕組みとしては、

```
random(n_samples) 入力
↓
----------------------------------------
↓
mu(2),Cov(2,2) を元に初期惑星分布を決定する
↓
evolution_func に従って最終惑星分布を決定する
↓
----------------------------------------
↓
最終分布 (Rf,Pf)_i 出力
↓
Loss を評価
```

のようにするべきで、よって——に挟まれた二つの module を `nn.module` として導入してやる必要がある。

よって初期分布を決定する module は、惑星数 `n_samples` を受け入れ、$\vec{mu}, \Sigma$ を参照して初期分布 $(R_i, P_i)_{i=1}^n$ を出力しなければならない。（ただし、$\mathcal{N}(\vec{mu}, \Sigma)$ に従うのは $(\log R, \log P)$ ）ただし、Σ を学習パラメタとしてしまうと、分散共分散行列が満たすべき正定値性を満たすことが難しい。そこで次のコレスキー分解を考える。

$$\Sigma = LL^T \ , \ L = \begin{pmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{pmatrix} \ , \ l_{11}, l_{22} > 0$$

すなわち、`nn.parameter` を $l_{11}^0, l_{22}^0, 21$ などと置いて、Σ を

$$L = \begin{pmatrix} \mathrm{softplus}(l_{11}^0) & 0 \\ l_{21} & \mathrm{softplus}(l_{22}^0) \end{pmatrix} \ , \ \Sigma = LL^T$$

のようにして Σ を計算することにすれば良い。ただし実際に初期分布を与えるにあたっては Σ を計算する必要もなく、入力の `n_samples` に対して

$$\vec{\varepsilon}_i \in \mathbb{R}^2 \ , \ \vec{\varepsilon}_i \sim \mathcal{N}\left(\vec{0}, I_2\right)$$

として $\mathcal{N}(\vec{0}, I_2)$ に従う random 変数 $\vec{\varepsilon_1}, \vec{\varepsilon_2}, ...$ を生成して、

$$\begin{pmatrix} \log R_i \\ \log P_i \end{pmatrix} = \vec{x} = \vec{\mu} + L\vec{\varepsilon}_i$$

のように定めれば、

$$E(\vec{x}) = \vec{\mu} \ , \ \Sigma_{\vec{x}} = LL^T = \Sigma$$

となっていることがわかる。（より厳密には、ちゃんと $\vec{x} \sim \mathcal{N}(\vec{\mu}, \Sigma)$ の多変量正規分布になっている。）ただし、実際の学習は観測データを用いて正規化したものに対して行うので、得られた惑星分布 $x_i = \log R_i, \log P_i$ に対して

$$z_i = \frac{x_i - \overline{x_{\mathrm{obs}}}}{\sigma(x_{\mathrm{obs}})}$$

を計算し、それを進化させて、観測データと比較する。（観測データもちゃんと log をとって正規化すれば、x は $\log R, \log P$ の意味をちゃんと持ってくれる。）

と、いうわけで、`class Initial_distribution(nn.module)` を定義し、`nn.parameters` として `mu(2)`,`l11_raw`,`l21`,`l22_raw` を保持し、`foward` により、入力 `n_samples` を受け入れて惑星初期分布

```
x(n_samples,2) = mu(n_samples,2) + epsilon(n_samples,2)*L^T(2,2)
```

を返せばいい。

[ ]:
```
class Initial_distribution(nn.Module):
    def __init__(self,x_mean_np,x_std_np,eps_cholesky=1e-4):
        super().__init__()
        self.eps_cholesky = eps_cholesky
        self.register_buffer("x_mean_obs",torch.tensor(x_mean_np, dtype=torch.float32))
        self.register_buffer("x_std_obs",torch.tensor(x_std_np, dtype=torch.float32))

        self.mu = nn.Parameter(torch.zeros(2))
        self.l11_raw = nn.Parameter(torch.tensor(0.0))
        self.l22_raw = nn.Parameter(torch.tensor(0.0))
        self.l21 = nn.Parameter(torch.tensor(0.0))

    def get_L(self):
        l11 = F.softplus(self.l11_raw) + self.eps_cholesky # 正定値なので、0になるのはまずい！
        l22 = F.softplus(self.l22_raw) + self.eps_cholesky
        L = torch.zeros(2,2,device=self.mu.device,dtype=self.mu.dtype)
        L[0,0] = l11
        L[1,1] = l22
        L[1,0] = self.l21
        return L

    def forward(self,n_samples):
        eps = torch.randn(n_samples,2,device=self.mu.device,dtype=self.mu.dtype)
        L=self.get_L()
        x = self.mu[None,:] + eps @ L.T
        z = (x-self.x_mean_obs[None,:])/self.x_std_obs[None,:]
        # x =  (logRi,logPi)_i~{n_samples} with shape (n_sampes,2)
        # z = standarized(x), (n_sampes,2)
        return z,x
```

■進化関数　今、正規化された変数

$$z_i = (z_i^R\ ,\ z_i^P)^T = \left(\frac{\log R_i - \overline{\log R_{\mathrm{obs}}}}{\sigma\left(\log R_{\mathrm{obs}}\right)}\ ,\ \frac{\log P_i - \overline{\log P_{\mathrm{obs}}}}{\sigma\left(\log P_{\mathrm{obs}}\right)} \right)^T$$

の進化を与える写像を次のように定義する。

$$z_i'^R = z_i^R + f^R(z_i^R, z_i^P)$$
$$z_i'^P = z_i^P + f^P(z_i^R, z_i^P)$$

よって f^R, f^P はともに二変数入力一変数出力の関数である。特に、R の変化については、「半径は減少しかしない」という制約を入れたいので、最終出力層を $f^R = -\mathrm{softplus}$ という形にする。（ただし、$-\alpha\,\mathrm{softplus}$ として α を適宜調整しないと、「動きすぎる」可能性がある。）また、周期は変化の度合いが小さいとしたいので、最終出力層を $f^P = \varepsilon \tanh$ という形にし、ε が小さいという制約を入れることにする。

なので、二次元入力を受け取ったら、それを MLP に通して、最終層で再び二次元にして、softplus と tanh を通すことになる。

[ ]:
```
class EvolutionNet(nn.Module):
    def __init__(self,hidden_dim,depth,eps_P=0.01,alpha_R=0.1):
        super().__init__()

        self.alpha_R = alpha_R
        self.eps_P = eps_P

        layers = []
```

```
        in_dim = 2

        for _ in range(depth):
            layers.append(nn.Linear(in_dim,hidden_dim))
            layers.append(nn.Tanh())
            in_dim = hidden_dim
        self.body_network = nn.Sequential(*layers)
        self.outlayer = nn.Linear(hidden_dim,2)

        nn.init.zeros_(self.outlayer.weight)
        nn.init.zeros_(self.outlayer.bias)

    def forward(self,z):
        h = self.body_network(z)
        out = self.outlayer(h)

        out_R_bef = out[:,0]
        out_P_bef = out[:,1]

        dz_R = -self.alpha_R * F.softplus(out_R_bef)
        dz_P = self.eps_P * torch.tanh(out_P_bef)

        dz = torch.stack([dz_R,dz_P],dim=1)
        z_val = z + dz

        return z_val,dz
```

■Lossfunction　先に述べたように、損失関数は モデル出力分布と観測分布の距離＋正則化項 で記述する。今回の場合は、

$$\mathcal{L} = |z_{\text{model}} - z_{\text{obs}}| + \lambda\,(\text{正則化})$$

である。よってまずは $|z_{\text{model}} - z_{\text{obs}}|$ について考える。

今、二つのサンプル X, Y 集合の距離が小さくなるように学習したい時、Loss function としては Maximum Mean Discrepancy などを用いることができる。（参考：https://arxiv.org/pdf/2111.10344 ）

$$\mathcal{L}_{\text{MMD}} = \frac{1}{N^2}\sum_{i}^{N}\sum_{j}^{N} K\left(x_i, x_j\right) + \frac{1}{M^2}\sum_{i}^{M}\sum_{j}^{M} K\left(y_i, y_j\right) - \frac{2}{NM}\sum_{i}^{N}\sum_{j}^{M} K\left(x_i, y_j\right)$$

ただし、サンプル上の二つの点の近さに対応した量であるカーネル $K(x, y)$ の取り方はさまざま考えられるが、ここでは Gaussian RBF Kernel を採用する。

$$K\left(\vec{x}_i, \vec{y}_j\right) = \exp\left(-\frac{||\vec{x}_i - \vec{y}_j||^2}{2\sigma^2}\right)$$

ただし σ はハイパーパラメタであり、σ が大きいと、「ある程度距離が離れてても近いとみなす」という感じになり、σ が小さいと、「よっぽど近くないと近いと見做さない」ということになる。なので、大きくても小さくてもダメで、ここはうまく調整してあげなきゃいけない。

また実際に計算する上では、`torch.cdist(x,y)` が $\left(||x_i, y_j||\right)_{i,j}$ という行列を保存してくれるので非常に便利！

[ ]:
```
def MMD_loss(x,y,sigmas=(0.1, 0.2, 0.5, 1.0, 2.0)):
    dxx = torch.cdist(x,x).pow(2)
    dyy = torch.cdist(y,y).pow(2)
    dxy = torch.cdist(x,y).pow(2)

    kxx = 0.0
    kyy = 0.0
    kxy = 0.0

    for sigma in sigmas:
        gamma = 1.0 / (2.0 * sigma * sigma)
        kxx = kxx + torch.exp(-gamma * dxx)
        kyy = kyy + torch.exp(-gamma * dyy)
        kxy = kxy + torch.exp(-gamma * dxy)

    return kxx.mean() + kyy.mean() - 2.0*kxy.mean()
```

■正則化　正則化については、「半径方向の進化関数が複雑すぎない（せいぜいピーク一つ）」と、「周期方向の進化関数が複雑すぎず、特に周期方向に単調」というものを入れる。すなわち、

$$\frac{\partial^3 f^R}{\partial \left(z^R\right)^3} \sim 0\,,\ \frac{\partial^3 f^R}{\partial \left(z^P\right)^3} \sim 0$$

$$\frac{\partial^3 f^P}{\partial \left(z^R\right)^3} \sim 0\,,\ \frac{\partial^2 f^P}{\partial \left(z^P\right)^2} \sim 0$$

具体的には、

$$\int \left|\frac{\partial^n f^x}{\partial \left(z^y\right)^n}\right|^2 dz^y : \text{small} \ \rightsquigarrow\ \sum_i \sum_j \left|\frac{\partial^n f_i^x}{\partial \left(z_j^y\right)^n}\right|^2 : \text{small}$$

とする。また、全体として惑星の半径、周期変化はそこまで大きくならない、という制約を入れる。

[ ]:
```
def grad_component(y,z,component):
    #y の z による微分
    #component=0 -> zR 微分
    #component=1 -> zP 微分
    grad = torch.autograd.grad(y.sum(),z,create_graph=True,retain_graph=True)[0]
    return grad[:,component]

def nth_derivative(y,z,component,n):
    #n 階微分
    #component=0 -> zR 微分
    #component=1 -> zP 微分
    out = y
    for _ in range(n):
        out = grad_component(out,z,component)
    return out

def derivative_regularization(z0,dz):
    fR = dz[:,0]
    fP = dz[:,1]

    d3fR_dzR3 = nth_derivative(fR,z0,component=0,n=3)
    d3fR_dzP3 = nth_derivative(fR,z0,component=1,n=3)
    d3fP_dzR3 = nth_derivative(fP,z0,component=0,n=3)
    d3fP_dzP2 = nth_derivative(fP,z0,component=1,n=2)

    loss_fR_Rsmooth = torch.mean(d3fR_dzR3**2)
    loss_fR_Psmooth = torch.mean(d3fR_dzP3**2)
    loss_fP_Rsmooth = torch.mean(d3fP_dzR3**2)
```

```
    loss_fP_Psmooth = torch.mean(d3fP_dzP2**2)

    return loss_fR_Rsmooth,loss_fR_Psmooth,loss_fP_Rsmooth,loss_fP_Psmooth

def make_reg_grid(nR=24, nP=24, zmin=-3.0, zmax=3.0, device="cpu"):
    zR = torch.linspace(zmin, zmax, nR, device=device)
    zP = torch.linspace(zmin, zmax, nP, device=device)
    ZZ_R, ZZ_P = torch.meshgrid(zR, zP, indexing="ij")
    z_grid = torch.stack([ZZ_R.reshape(-1), ZZ_P.reshape(-1)], dim=1)
    z_grid.requires_grad_(True)
    return z_grid
```

■学習の準備

[ ]:
```
hidden_dim = 32 #本当は色々動かす
depth = 2 #本当はハイパーパラメタ
eps_P = 0.01 #本当はハイパーパラメタ
alpha_R = 0.1 #本当はハイパーパラメタ
sigma=1.0 #本当はハイパーパラメタ
LR = 1e-3

initial_dist = Initial_distribution(x_mean_np,x_std_np,eps_cholesky).to(device)
with torch.no_grad():
    initial_dist.mu.copy_(
        torch.tensor(x_mean_np, dtype=torch.float32, device=device)
    )
    # 初期半径は観測よりだいぶ大きめから始める。(ここは要確認...)
    initial_dist.mu[0] += 0.5

evolution_net = EvolutionNet(hidden_dim,depth,eps_P,alpha_R).to(device)
optimizer = torch.optim.Adam(
    list(initial_dist.parameters())+list(evolution_net.parameters()),
    lr = LR
    )
```

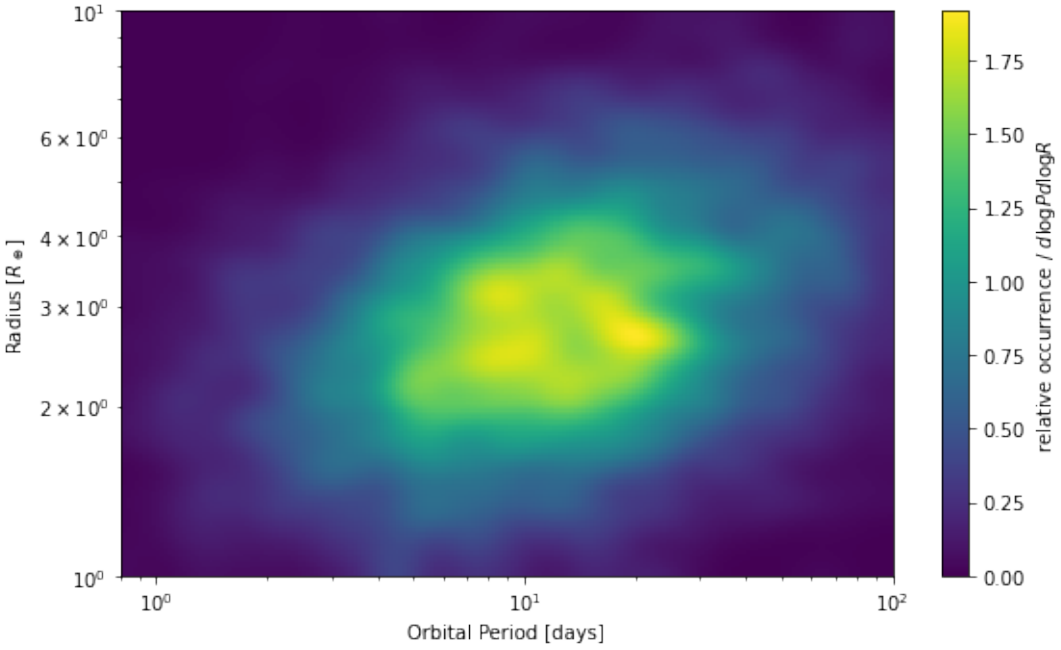
■学習

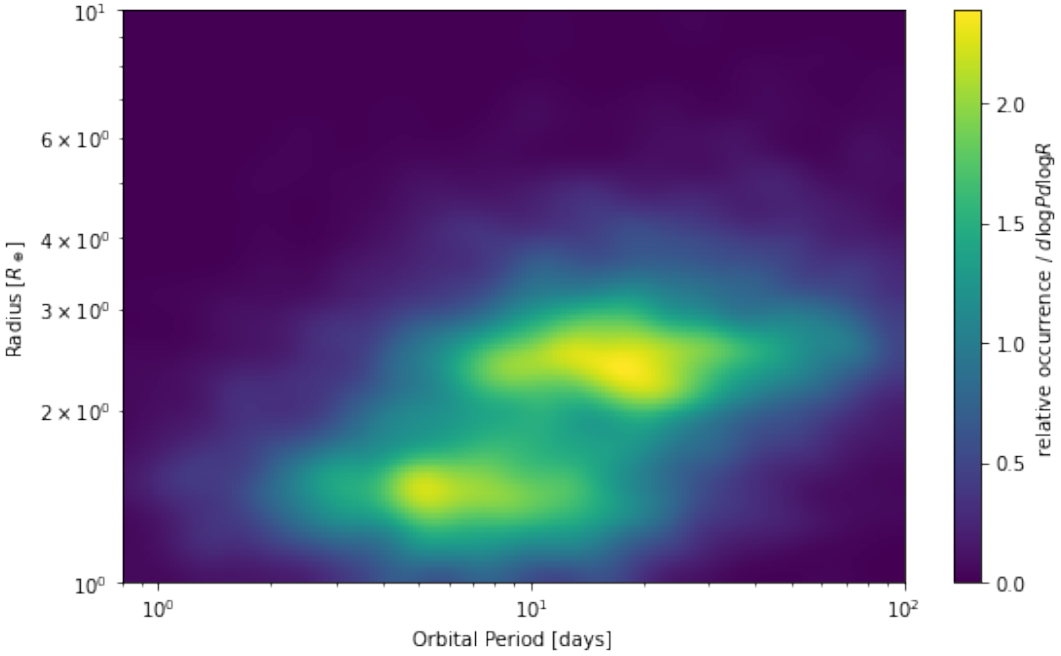
[ ]:
```
num_epochs = num_epochs_par

lambda_fR_Rsmooth=1e-4 #本当はハイパーパラメタ
lambda_fR_Psmooth=1e-4 #本当はハイパーパラメタ
lambda_fP_Rsmooth=1e-4 #本当はハイパーパラメタ
lambda_fP_Psmooth=1e-4 #本当はハイパーパラメタ
lambda_move=0.0 #いったん0にしておく

loss_history=[]
for epoch in range(num_epochs):
    optimizer.zero_grad()
    z0,x0 = initial_dist(n_samples)
    z0.requires_grad_(True)

    z_model,dz=evolution_net(z0)

    loss_MMD = MMD_loss(z_model,z_obs)

    z_reg = make_reg_grid(device=device)
    _, dz_reg = evolution_net(z_reg)
    loss_fR_Rsmooth,loss_fR_Psmooth,loss_fP_Rsmooth,loss_fP_Psmooth=derivative_regularization(
        z_reg,
        dz_reg
        )

    loss_move = torch.mean(dz**2)

    loss = (
        loss_MMD
        + lambda_fR_Rsmooth*loss_fR_Rsmooth
        + lambda_fR_Psmooth*loss_fR_Psmooth
        + lambda_fP_Rsmooth*loss_fP_Rsmooth
        + lambda_fP_Psmooth*loss_fP_Psmooth
        + lambda_move*loss_move
    )
```

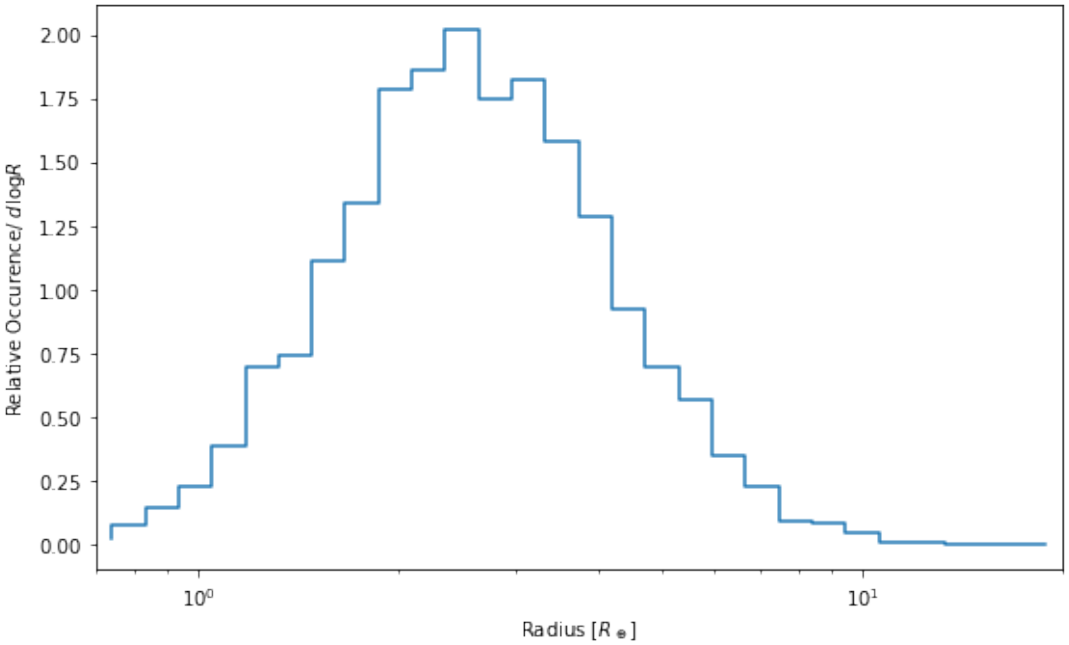
```
    loss.backward()
    optimizer.step()
    if epoch % 100 == 0:
        print(
            f"epoch={epoch:5d}",
            f"loss={loss.item():.6f}"
        )
```

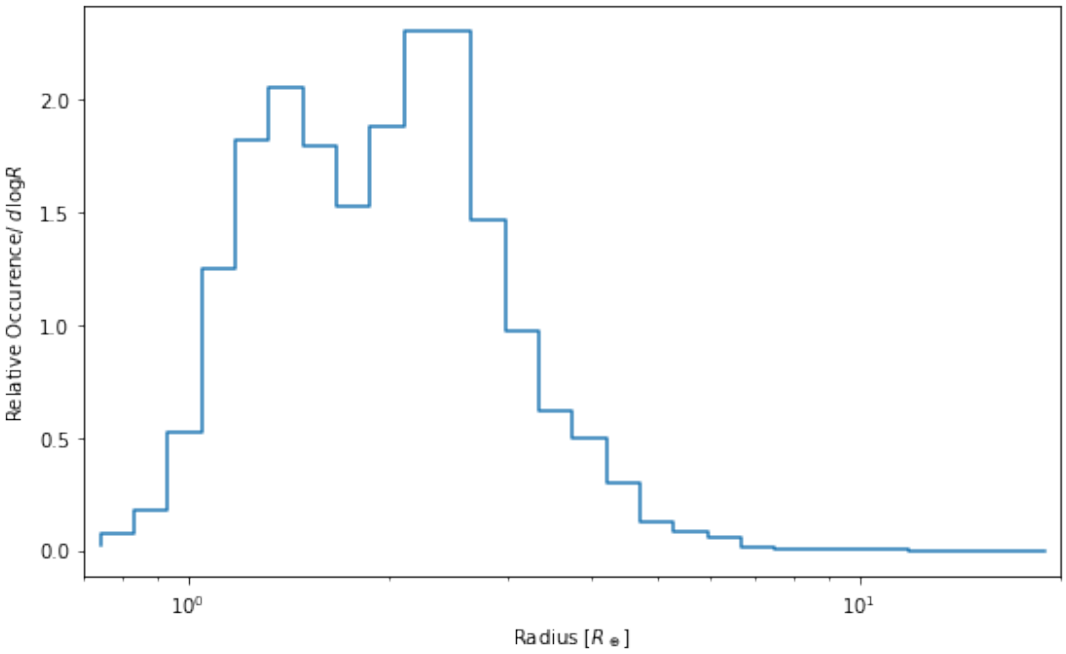
```
epoch=    0 loss=0.332436
epoch=  100 loss=0.138049
epoch=  200 loss=0.059500
epoch=  300 loss=0.025527
epoch=  400 loss=0.023606
epoch=  500 loss=0.013518
epoch=  600 loss=0.011387
epoch=  700 loss=0.009539
epoch=  800 loss=0.010047
epoch=  900 loss=0.008356
epoch= 1000 loss=0.008374
epoch= 1100 loss=0.007497
epoch= 1200 loss=0.007685
epoch= 1300 loss=0.007032
epoch= 1400 loss=0.007122
epoch= 1500 loss=0.006225
epoch= 1600 loss=0.006608
epoch= 1700 loss=0.007818
epoch= 1800 loss=0.005501
epoch= 1900 loss=0.005697
epoch= 2000 loss=0.006143
epoch= 2100 loss=0.007816
epoch= 2200 loss=0.005652
epoch= 2300 loss=0.005909
epoch= 2400 loss=0.006287
epoch= 2500 loss=0.007239
epoch= 2600 loss=0.005028
epoch= 2700 loss=0.005798
epoch= 2800 loss=0.004656
epoch= 2900 loss=0.007041
```

[ ]:
```
with torch.no_grad():
    z0, x0 = initial_dist(5000)
    z_model, dz = evolution_net(z0)

    # z_model -> x_model = (logR, logP)
    x_model = (
        z_model * initial_dist.x_std_obs[None, :]
        + initial_dist.x_mean_obs[None, :]
    )

    # 初期分布
    R0 = torch.exp(x0[:, 0]).cpu().numpy()
    P0 = torch.exp(x0[:, 1]).cpu().numpy()

    # 進化後分布
    R_model = torch.exp(x_model[:, 0]).cpu().numpy()
    P_model = torch.exp(x_model[:, 1]).cpu().numpy()

draw_histogram_PR(R0,P0,"./fig/model_initial_PR.png")
draw_histogram_PR(R_model,P_model,"./fig/model_estimate_PR.png")
draw_histogram_Ronly(R0,"./fig/model_initial_Ronly.png")
draw_histogram_Ronly(R_model,"./fig/model_estimate_Ronly.png")
```


Radius [$R_\oplus$]
Orbital Period [days]
relative occurrence / $d\log P d\log R$


Radius [$R_\oplus$]
Orbital Period [days]
relative occurrence / $d\log P d\log R$


2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Relative Occurence/ dlogR
10^0
10^1
Radius [$R_\oplus$]


2.0
1.5
1.0
0.5
0.0
Relative Occurence/ dlogR
10^0
10^1
Radius [$R_\oplus$]

■初期条件の初期値を色々変えて見る　→ 初期条件の初期値によらず一つの初期条件に収束してくれれば、ニューラルネットワークで複雑性を吸収せずに初期条件を推定できてるだろうと予測できる。

■information criterion　その際、正則化の強さを決めるハイパーパラメタを次のように決定する。(ここは、明確な基準がないので恣意的に決める必要がある。)

$$\text{Information criterion} = \text{MMDloss} + \text{推定初期分布のばらつき}$$

すなわち、「観測と一致する」「初期分布が学習初期条件によらず一つに収束する」という条件を満たしてほしい。(NN が複雑性を吸収してしまうと、観測とはよく一致するが推定される初期分布はバラバラになると考えられ、NN が単純すぎると、初期分布で合わせに行くので初期分布は一意になる NN は単純なので観測との一致度合いは悪くなる。) 定量的には、ハイパーパラメタ λ , 学習初期条件 α での推定初期分布を　q_λ^α などとして、

$$\mathrm{IC}(\lambda) = \frac{1}{N}\sum_{1\le i\le K} \mathcal{L}_{\mathrm{MMD}}(\lambda,\alpha_i) + \beta \times \frac{2}{K(K-1)}\sum_{1\le i\le K}\sum_{1\le j\le K} \mathrm{MMD}(q_\lambda^{\alpha_i}, q_\lambda^{\alpha_j})$$

($i=j$ の時 MMD は 0 になるので、$k(k-1)$ この平均を取れば良い。) ただし、β は二つの項の値のスケールがおおよそ同じくらいになるように調整する。(ここは諸説ある。)

[ ]:
```
# 推定初期分布のばらつき
# lambda ごとの結果 (alpha 様々) を z0_final_list=[] に入れておく。
def compute_variance_final_init(z0_final_list,device="cpu"):
    K = len(z0_final_list)
    pairs=[]
    for i in range(K):
        for j in range(i+1,K):
            pairs.append((i,j))

    values = []
    with torch.no_grad():
        for (i,j) in pairs:
            zi = z0_final_list[i].to(device)
            zj = z0_final_list[j].to(device)
            d = MMD_loss(zi,zj).detach().cpu().item()
            values.append(d)
    return float(np.mean(values))

# information criterion
# 全ての結果 (lambda,alpha 様々)
# を run_results=[dict,lambda,alpha,loss_MMD,...] に入れておく。

def information_criterion(run_results):

    #lambda の値ごとに results をまとめる
    grouped={}
    for r in run_results:
        key=(r["lambda_smooth"], r["lambda_move"])
        grouped.setdefault(key, []).append(r)

    rows=[]
    for (lambda_smooth, lambda_move), group in grouped.items():
        mmd_values = [g["mmd"] for g in group]
        z0_final_list = [g["z0_final"] for g in group]
        var_init_mean = compute_variance_final_init(z0_final_list,device=device)
        rows.append({
```

```
            "lambda_smooth": lambda_smooth,
            "lambda_move": lambda_move,
            "n_runs": len(group),
            "mmd_mean": float(np.mean(mmd_values)),
            "mmd_std": float(np.std(mmd_values)),
            "var_init_mean": var_init_mean,
        })
    summary = pd.DataFrame(rows)

    typical_mmd = summary["mmd_mean"].median()
    typical_var = summary["var_init_mean"].median()
    beta = typical_mmd / (typical_var + 1e-12)

    summary["beta"] = beta
    summary["IC"] = summary["mmd_mean"]+beta*summary["var_init_mean"]
    summary = summary.sort_values("IC").reset_index(drop=True)

    return summary
```

■可視化用

[ ]:
```
def draw_histogram_PR_multi(R0initial,P0initial,R0,P0,R,P,Robs,Pobs,filename):

    global histogram_Rmin,histogram_Rmax,histogram_Pmin,histogram_Pmax,nummesh_for_histogram,sigma_hist_par

    logR_edges=np.linspace(np.log10(histogram_Rmin),np.log10(histogram_Rmax),nummesh_for_histogram)
    logP_edges=np.linspace(np.log10(histogram_Pmin),np.log10(histogram_Pmax),nummesh_for_histogram)

    histogram_first_initial,P_edges,R_edges= np.histogram2d(
        np.log10(P0initial),np.log10(R0initial),bins=[logP_edges,logR_edges]
        )
    histogram_final_initial,P_edges,R_edges= np.histogram2d(
        np.log10(P0),np.log10(R0),bins=[logP_edges,logR_edges]
        )
    histogram_final_estimate,P_edges,R_edges= np.histogram2d(
        np.log10(P),np.log10(R),bins=[logP_edges,logR_edges]
        )
    histogram_obs,P_edges,R_edges= np.histogram2d(
        np.log10(Pobs),np.log10(Robs),bins=[logP_edges,logR_edges]
        )
    area = np.diff(P_edges)[:,None]*np.diff(R_edges)[None,:]
    #print(histogram.shape) #(nummeshforhistogram-1)^2
    #print(area.shape)

    sigma_hist = sigma_hist_par * nummesh_for_histogram

    density_first_initial = histogram_first_initial/area
    density_first_initial = density_first_initial / np.sum(histogram_first_initial)
    density_first_initial = gaussian_filter(density_first_initial, sigma=sigma_hist)

    density_final_initial = histogram_final_initial/area
    density_final_initial = density_final_initial / np.sum(histogram_final_initial)
    density_final_initial = gaussian_filter(density_final_initial, sigma=sigma_hist)

    density_final_estimate = histogram_final_estimate/area
    density_final_estimate = density_final_estimate / np.sum(histogram_final_estimate)
    density_final_estimate = gaussian_filter(density_final_estimate, sigma=sigma_hist)

    density_obs = histogram_obs/area
    density_obs = density_obs / np.sum(histogram_obs)
    density_obs = gaussian_filter(density_obs, sigma=sigma_hist)

    densities = [
        density_first_initial,
        density_final_initial,
        density_final_estimate,
        density_obs,
    ]

    titles = [
        "Initial distribution before training",
        "Initial distribution after training",
        "Model estimate distribution",
        "Observed distribution",
    ]
```

```
    fig, axes = plt.subplots(1,4,figsize=(20.0, 5.0),constrained_layout=True)
    im = None
    for ax, density, title in zip(axes, densities, titles):
        im = ax.pcolormesh(10**P_edges,10**R_edges,density.T)
        ax.set_xscale("log")
        ax.set_yscale("log")
        ax.set_xlim(histogram_Pmin, histogram_Pmax)
        ax.set_ylim(histogram_Rmin, histogram_Rmax)
        ax.set_title(title)
        ax.set_xlabel("Orbital Period [days]")
    axes[0].set_ylabel("Radius [$R_\oplus$]")
    fig.colorbar(im,ax=axes,label="relative density / $d\log P\,d\log R$")
    fig.savefig(filename, dpi=200, bbox_inches="tight", pad_inches=0.03)
    plt.close(fig)
```

[ ]:
```
# R方向のヒストグラムを描く
def draw_histogram_Ronly_multi(R0initial,R0,R,Robs,filename):

    global histogram_Ronly_min,histogram_Ronly_max,nummesh_for_histogram_Ronly

    logR_edges=np.linspace(np.log10(histogram_Ronly_min),np.log10(histogram_Ronly_max),nummesh_for_histogram_Ronly)

    histogram_first_initial , R_edges = np.histogram(
        np.log10(R0initial),bins=logR_edges
    )
    histogram_final_initial , R_edges = np.histogram(
        np.log10(R0),bins=logR_edges
    )
    histogram_final_estimate , R_edges = np.histogram(
        np.log10(R),bins=logR_edges
    )
    histogram_obs , R_edges = np.histogram(
        np.log10(Robs),bins=logR_edges
    )
    #観測個数密度に直すことを考えると、データ同化的な考え方をしなくちゃいけない。(幅がでかい bin では実際に存在するより多く見つかっていると考えるべき。)
    width = np.diff(R_edges)

    density_first_initial = histogram_first_initial/width
    density_first_initial = density_first_initial/np.sum(histogram_first_initial)

    density_final_initial = histogram_final_initial/width
    density_final_initial = density_final_initial/np.sum(histogram_final_initial)

    density_final_estimate = histogram_final_estimate/width
    density_final_estimate = density_final_estimate/np.sum(histogram_final_estimate)

    density_obs = histogram_obs/width
    density_obs = density_obs/np.sum(histogram_obs)

    densities = [
        density_first_initial,
        density_final_initial,
        density_final_estimate,
        density_obs,
    ]

    titles = [
        "Initial distribution before training",
        "Initial distribution after training",
        "Model estimate distribution",
        "Observed distribution",
    ]

    centers = 10**(0.5*(R_edges[:-1]+R_edges[1:]))

    fig, axes = plt.subplots(1,4,figsize=(20.0, 5.0),constrained_layout=True)
    im = None
    for ax, density, title in zip(axes, densities, titles):
        im = ax.step(centers,density)
        ax.set_xscale("log")
        ax.set_xlim(histogram_Ronly_min,histogram_Ronly_max)
        ax.set_title(title)
        ax.set_xlabel("Radius [$R_\oplus$]")
    axes[0].set_ylabel("Relative Occurence/ $d\log R$")

    fig.savefig(filename,dpi=200, bbox_inches="tight",pad_inches=0.03)
```

```
    plt.close(fig)
```

■学習

[ ]:
```
# CPU並列化
N_JOBS = 4
torch.set_num_threads(1)
```

[ ]:
```
initial_distances = [0.1,0.5,1.0,1.2]
initial_periods = [-1.0,-0.5,0.1,0.5,1.0]

lambda_smooths=[1e-2,1e-3,1e-4,0.0]
lambda_moves=[1e-2,1e-4,0.0]

jobs = list(product(lambda_smooths, lambda_moves, initial_distances, initial_periods))
num_loops = len(jobs)
print("num_loops =", num_loops)
print("CPUs =", N_JOBS)
print("device =", device)

def run_one_job(job_id, job):
    lambda_smooth, lambda_move, initial_distance, initial_period = job
    torch.set_num_threads(1)

    hidden_dim = 32 #本当は色々動かす
    depth = 2 #本当はハイパーパラメタ
    eps_P = 0.01 #本当はハイパーパラメタ
    alpha_R = 0.1 #本当はハイパーパラメタ
    sigma=1.0 #本当はハイパーパラメタ
    LR = 1e-3

    local_device = torch.device("cpu")
    z_obs_local = z_obs.to(local_device)

    tagdir = (f"lsmooth_{lambda_smooth:.0e}_lmove_{lambda_move:.0e}".replace(".", "p").replace("-", "m").replace("+",␣
↪""))
    outdir = f"./fig/{tagdir}"
    os.makedirs(outdir, exist_ok=True)
    initial_dist = Initial_distribution(x_mean_np,x_std_np,eps_cholesky).to(local_device)

    with torch.no_grad():
        initial_dist.mu.copy_(
            torch.tensor(x_mean_np, dtype=torch.float32, device=local_device)
        )
        # 初期半径は観測よりだいぶ大きめから始める。(ここは要確認...)
        initial_dist.mu[0] += initial_distance
        initial_dist.mu[1] += initial_period


    with torch.no_grad():
        z0initial, x0initial = initial_dist(5000)
        R0initial = torch.exp(x0initial[:, 0]).cpu().numpy()
        P0initial = torch.exp(x0initial[:, 1]).cpu().numpy()

    evolution_net = EvolutionNet(hidden_dim,depth,eps_P,alpha_R).to(local_device)
    optimizer = torch.optim.Adam(
        list(initial_dist.parameters())+list(evolution_net.parameters()),
        lr = LR
        )

    num_epochs = num_epochs_par

    lambda_fR_Rsmooth=lambda_smooth
    lambda_fR_Psmooth=lambda_smooth
    lambda_fP_Rsmooth=lambda_smooth
    lambda_fP_Psmooth=lambda_smooth
    lambda_move=lambda_move

    loss_history=[]


    for epoch in range(num_epochs):
        optimizer.zero_grad()
```

```
        z0,x0 = initial_dist(n_samples)

        z0.requires_grad_(True)

        z_model,dz=evolution_net(z0)

        loss_MMD = MMD_loss(z_model,z_obs_local)

        z_reg = make_reg_grid(device=local_device)
        _, dz_reg = evolution_net(z_reg)
        loss_fR_Rsmooth,loss_fR_Psmooth,loss_fP_Rsmooth,loss_fP_Psmooth=derivative_regularization(
            z_reg,
            dz_reg
            )

        loss_move = torch.mean(dz**2)

        loss = (
            loss_MMD
            + lambda_fR_Rsmooth*loss_fR_Rsmooth
            + lambda_fR_Psmooth*loss_fR_Psmooth
            + lambda_fP_Rsmooth*loss_fP_Rsmooth
            + lambda_fP_Psmooth*loss_fP_Psmooth
            + lambda_move*loss_move
        )

        loss.backward()
        optimizer.step()
        if epoch % 1000 == 0:
            print(
                f"job={job_id+1:4d}/{num_loops:4d}",
                f"epoch={epoch:5d}/{num_epochs:5d}",
                f"loss={loss.item():.6f}",
                f"lambda_smooth={lambda_smooth:.0e}",
                f"lambda_move={lambda_move:.0e}",
                f"dR={initial_distance:.1f}",
                f"dP={initial_period:.1f}",
            )

        if epoch % 500 == 0 or epoch == num_epochs - 1:
            loss_history.append({
                "epoch": epoch,
                "loss": loss.detach().cpu().item(),
                "mmd": loss_MMD.detach().cpu().item(),
                "smooth_RR": loss_fR_Rsmooth.detach().cpu().item(),
                "smooth_RP": loss_fR_Psmooth.detach().cpu().item(),
                "smooth_PR": loss_fP_Rsmooth.detach().cpu().item(),
                "smooth_PP": loss_fP_Psmooth.detach().cpu().item(),
                "move": loss_move.detach().cpu().item(),
            })

    tag = (f"dR_{initial_distance:.1f}_dP_{initial_period:.1f}".replace(".", "p"))
    PR_path = f"{outdir}/PR{tag}.png"
    Ronly_path = f"{outdir}/Ronly{tag}.png"
    loss_path = f"{outdir}/loss{tag}.csv"

    with torch.no_grad():
        z0, x0 = initial_dist(5000)
        z_model, dz = evolution_net(z0)
        # z_model -> x_model = (logR, logP)
        x_model = (
            z_model * initial_dist.x_std_obs[None, :]
            + initial_dist.x_mean_obs[None, :]
        )
        # 初期分布
        R0 = torch.exp(x0[:, 0]).cpu().numpy()
        P0 = torch.exp(x0[:, 1]).cpu().numpy()
        # 進化後分布
        R_model = torch.exp(x_model[:, 0]).cpu().numpy()
        P_model = torch.exp(x_model[:, 1]).cpu().numpy()


    with torch.no_grad():
        z0_ic, x0_ic = initial_dist(1000)
        z_model_ic, dz_ic = evolution_net(z0_ic)

        mmd_ic = MMD_loss(z_model_ic, z_obs_local).detach().cpu().item()

    pd.DataFrame(loss_history).to_csv(loss_path, index=False)
```

```
    draw_histogram_PR_multi(R0initial,P0initial,R0,P0,R_model,P_model,R_data,P_data,PR_path)
    draw_histogram_Ronly_multi(R0initial,R0,R_model,R_data,Ronly_path)
    result={
            "lambda_smooth": lambda_smooth,
            "lambda_move": lambda_move,
            "initial_distance": initial_distance,
            "initial_period": initial_period,
            "mmd": mmd_ic,
            "z0_final": z0_ic.detach().cpu(),
            "PR_path": PR_path,
            "Ronly_path": Ronly_path,
            "loss_path": loss_path,
        }
    del initial_dist, evolution_net, optimizer
    gc.collect()

    return result
```

```
num_loops = 240
CPUs = 4
device = cpu
```

[ ]:
```
run_results = Parallel(
    n_jobs=N_JOBS,
    backend="loky",
    verbose=10
)(
    delayed(run_one_job)(job_id, job)
    for job_id, job in enumerate(jobs)
)
ic_summary = information_criterion(run_results)
os.makedirs("./result", exist_ok=True)

ic_summary.to_csv(
    "./result/information_criterion_summary.csv",
    index=False
)
display(ic_summary)
```

```
[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done   5 tasks      | elapsed: 20.8min
[Parallel(n_jobs=4)]: Done  10 tasks      | elapsed: 31.0min
[Parallel(n_jobs=4)]: Done  17 tasks      | elapsed: 49.8min
[Parallel(n_jobs=4)]: Done  24 tasks      | elapsed: 59.2min
[Parallel(n_jobs=4)]: Done  33 tasks      | elapsed: 87.5min
[Parallel(n_jobs=4)]: Done  42 tasks      | elapsed: 106.4min
[Parallel(n_jobs=4)]: Done  53 tasks      | elapsed: 134.5min
[Parallel(n_jobs=4)]: Done  64 tasks      | elapsed: 153.4min
[Parallel(n_jobs=4)]: Done  77 tasks      | elapsed: 191.0min
[Parallel(n_jobs=4)]: Done  90 tasks      | elapsed: 219.2min
[Parallel(n_jobs=4)]: Done 105 tasks      | elapsed: 256.9min
[Parallel(n_jobs=4)]: Done 120 tasks      | elapsed: 285.2min
[Parallel(n_jobs=4)]: Done 137 tasks      | elapsed: 332.2min
[Parallel(n_jobs=4)]: Done 154 tasks      | elapsed: 369.9min
[Parallel(n_jobs=4)]: Done 173 tasks      | elapsed: 416.9min
[Parallel(n_jobs=4)]: Done 192 tasks      | elapsed: 454.7min
[Parallel(n_jobs=4)]: Done 213 tasks      | elapsed: 511.4min
[Parallel(n_jobs=4)]: Done 240 out of 240 | elapsed: 568.0min finished
```

	lambda_smooth	lambda_move	n_runs	mmd_mean	mmd_std	var_init_mean	\
0	0.0010	0.0000	20	0.006440	0.001115	0.068157	
1	0.0001	0.0001	20	0.006590	0.001268	0.067750	
2	0.0010	0.0100	20	0.006864	0.001681	0.066194	
3	0.0100	0.0001	20	0.007473	0.001242	0.061310	
4	0.0010	0.0001	20	0.006741	0.001153	0.069941	
5	0.0100	0.0000	20	0.007521	0.001279	0.064912	
6	0.0001	0.0000	20	0.006650	0.001052	0.073734	
7	0.0001	0.0100	20	0.007041	0.001473	0.072149	
8	0.0100	0.0100	20	0.007519	0.001110	0.068055	
9	0.0000	0.0000	20	0.008608	0.002713	0.106823	
10	0.0000	0.0001	20	0.008805	0.002588	0.106179	
11	0.0000	0.0100	20	0.009751	0.003653	0.102339	

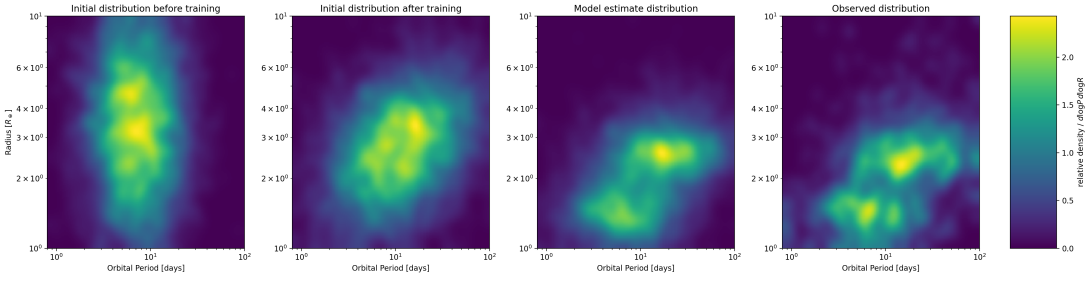
	beta	IC
0	0.105097	0.013603
1	0.105097	0.013710

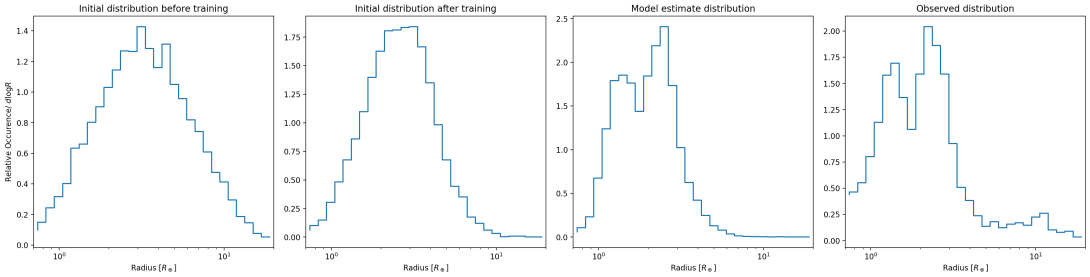
```
2   0.105097  0.013820
3   0.105097  0.013916
4   0.105097  0.014091
5   0.105097  0.014343
6   0.105097  0.014399
7   0.105097  0.014624
8   0.105097  0.014671
9   0.105097  0.019835
10  0.105097  0.019964
11  0.105097  0.020507
```

■best lambda の場合の可視化

[ ]:
```
best = ic_summary.iloc[0]

best_lambda_smooth = best["lambda_smooth"]
best_lambda_move = best["lambda_move"]

best_runs = [
    r for r in run_results if (r["lambda_smooth"] == best_lambda_smooth) and (r["lambda_move"] == best_lambda_move)
]
best_runs_df = pd.DataFrame([
    {
        "initial_distance": r["initial_distance"],
        "initial_period": r["initial_period"],
        "mmd": r["mmd"],
        "PR_path": r["PR_path"],
        "Ronly_path": r["Ronly_path"],
    }
    for r in best_runs
]).sort_values("mmd")
for _, row in best_runs_df.iterrows():
    print("=" * 80)
    print(
        f"initial_distance={row['initial_distance']}, "
        f"initial_period={row['initial_period']}, "
        f"mmd={row['mmd']:.6g}"
    )

    display(Image(filename=row["PR_path"]))
    display(Image(filename=row["Ronly_path"]))
```

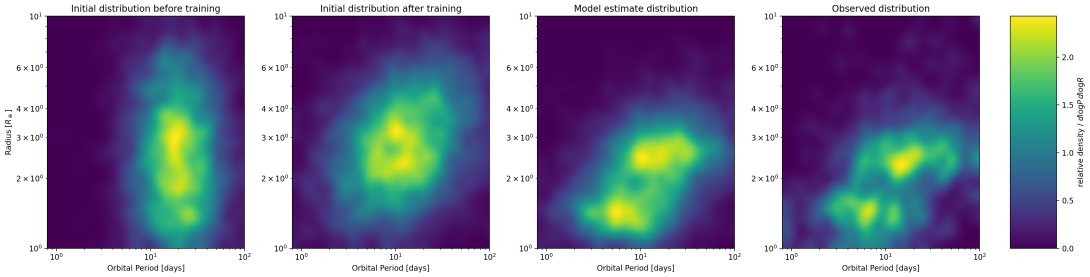
```
================================================================================
initial_distance=0.5, initial_period=-0.5, mmd=0.0047617
```

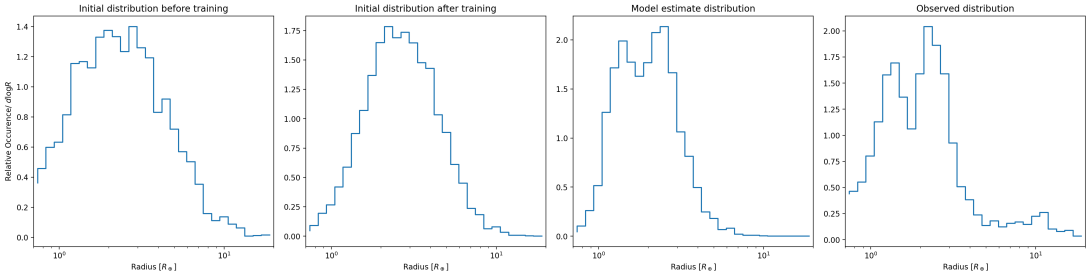




==========

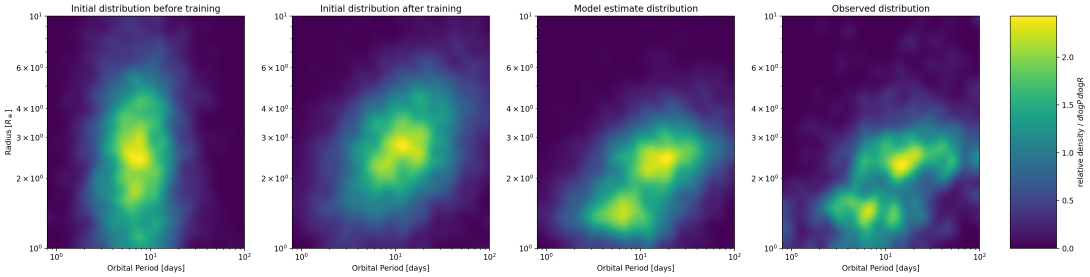
initial_distance=0.1, initial_period=0.5, mmd=0.00486326

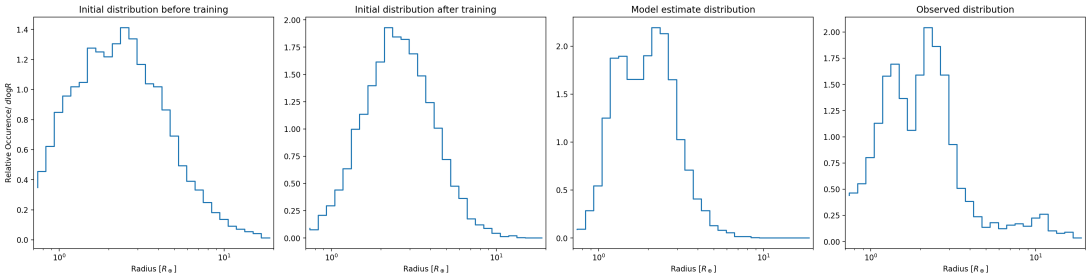




==========

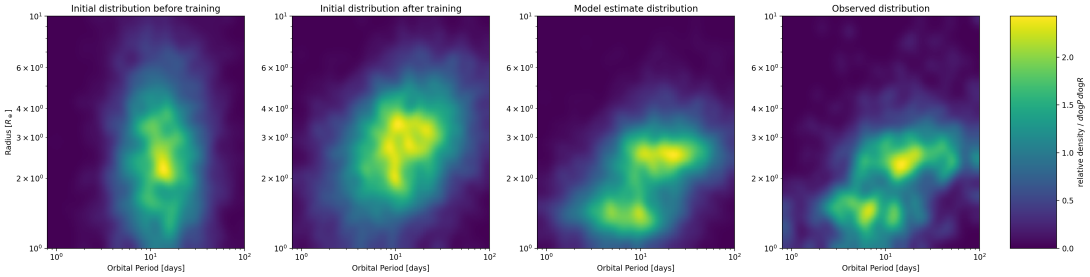
initial_distance=0.1, initial_period=-0.5, mmd=0.00539041

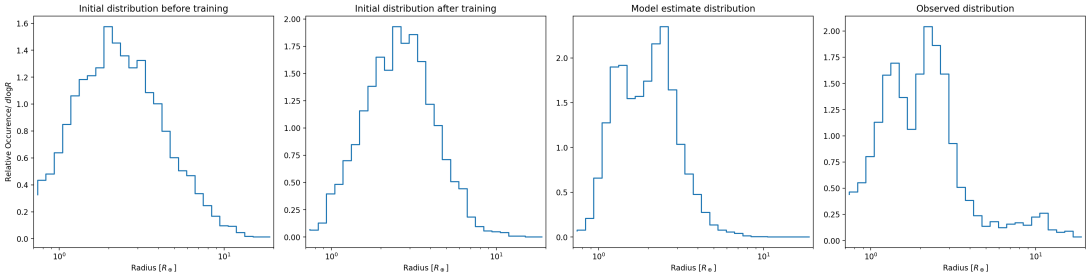




==========

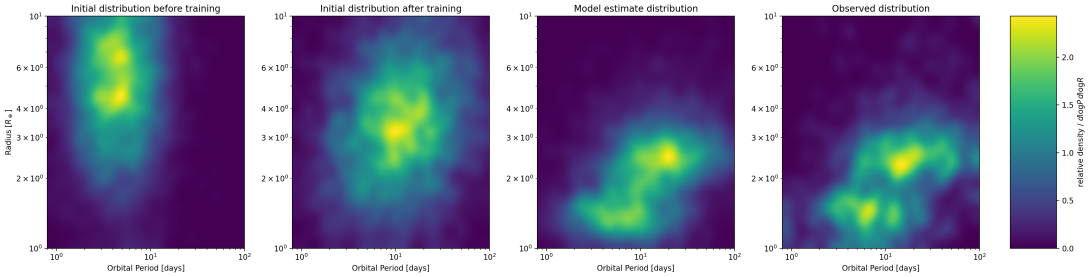
initial_distance=0.1, initial_period=0.1, mmd=0.00551534

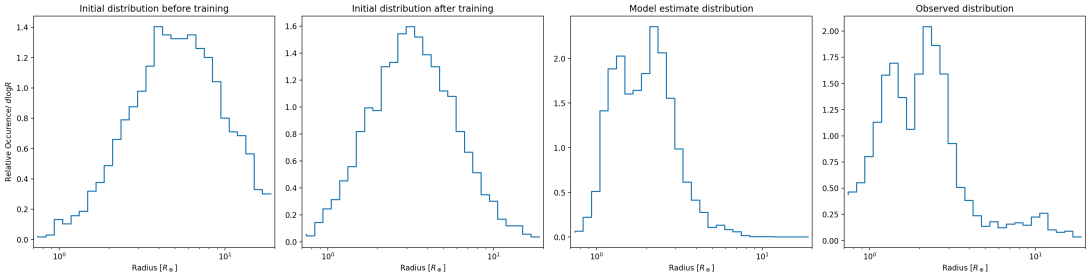




==========

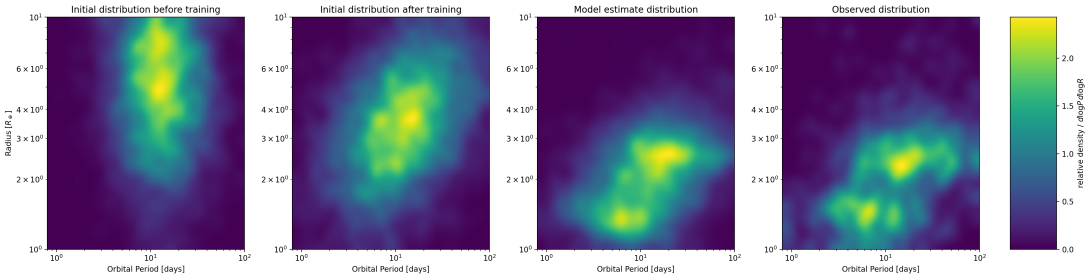
initial_distance=1.0, initial_period=-1.0, mmd=0.00567675

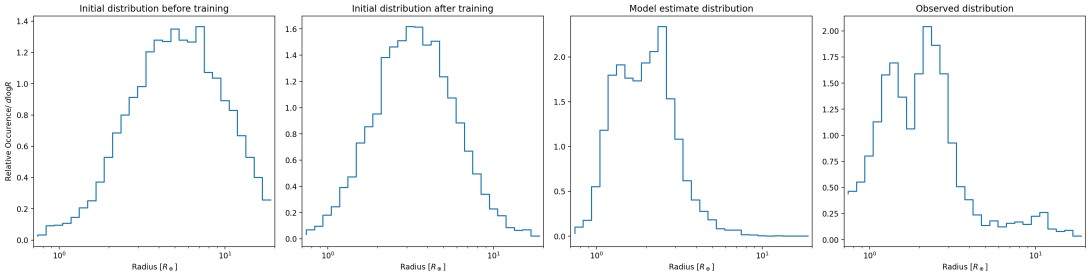




==========

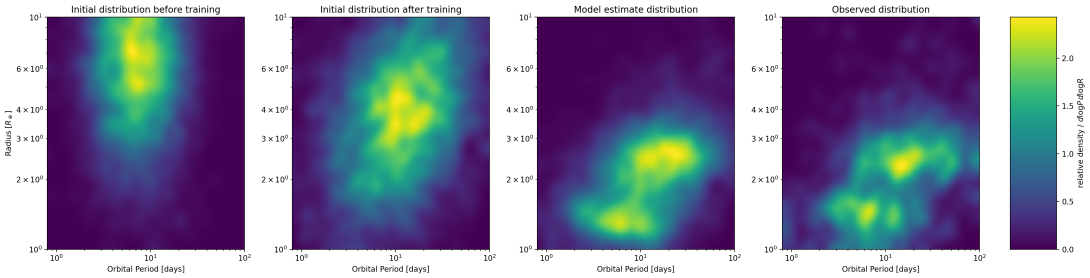
initial_distance=1.0, initial_period=0.1, mmd=0.00585675

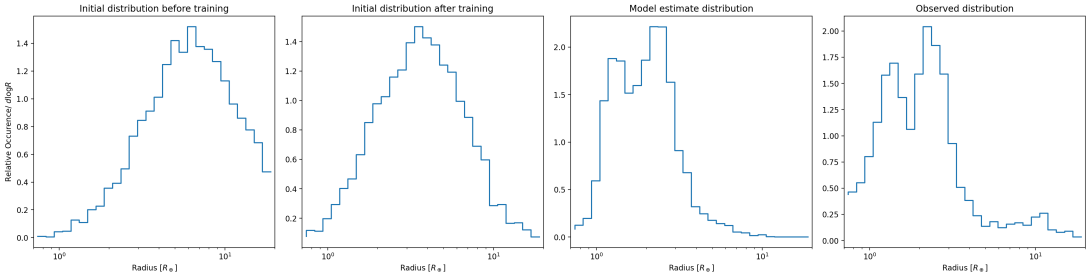




==========

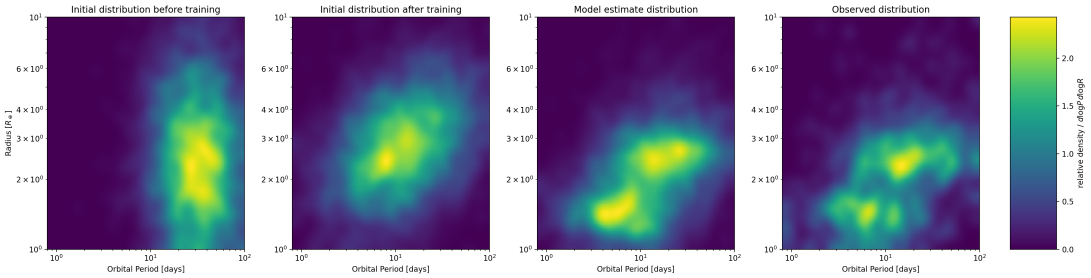
initial_distance=1.2, initial_period=-0.5, mmd=0.0059886

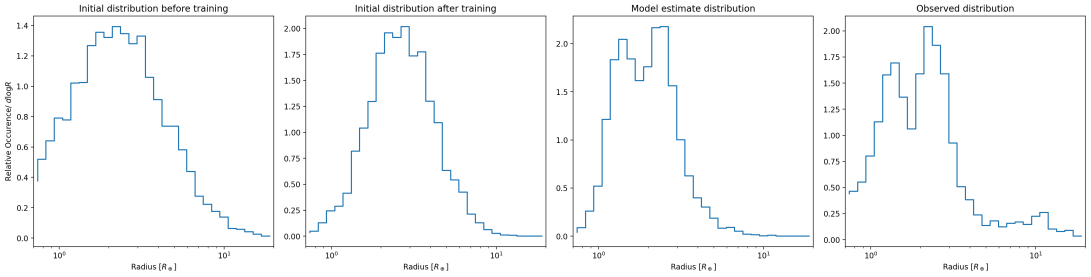




==========

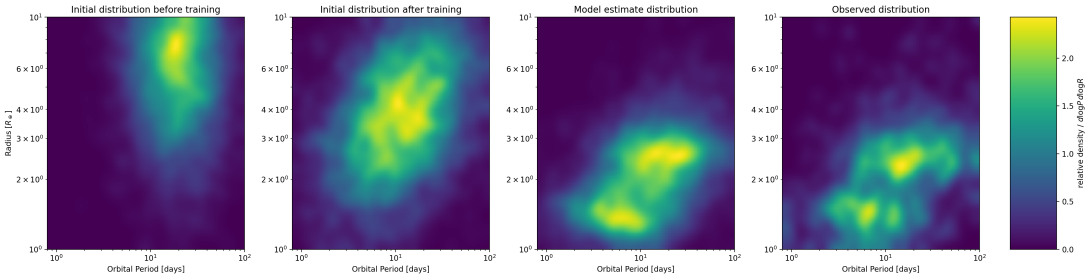
initial_distance=0.1, initial_period=1.0, mmd=0.00599742

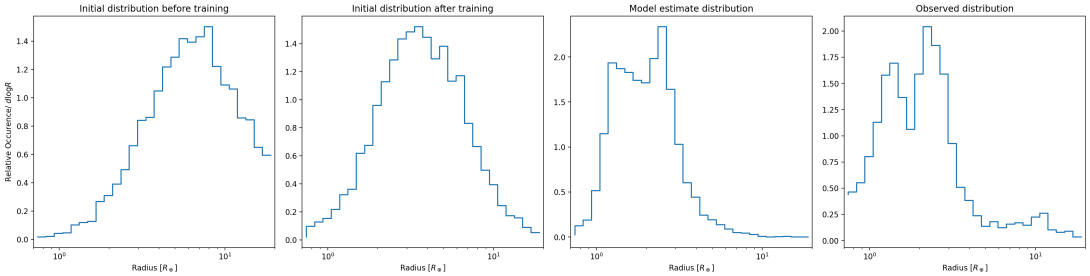




==========

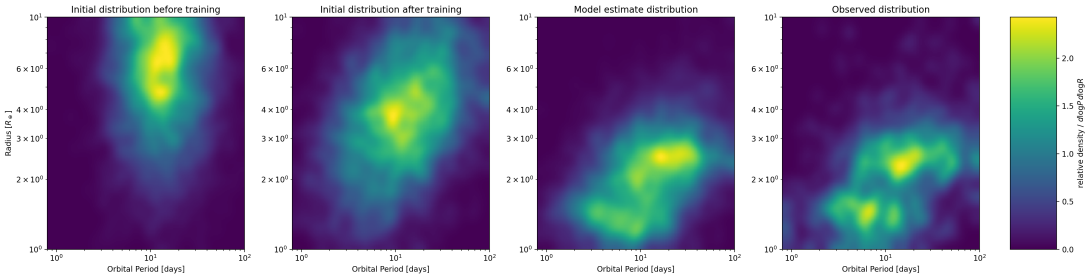
initial_distance=1.2, initial_period=0.5, mmd=0.00612187

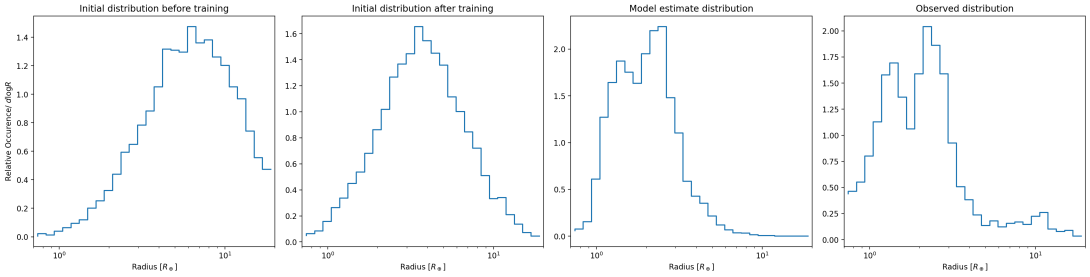




==========

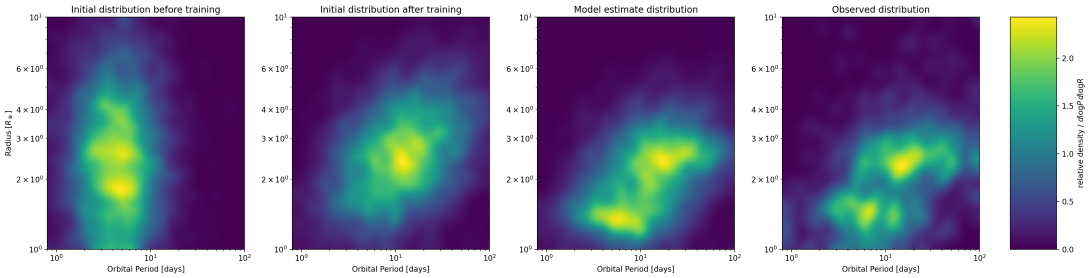
initial_distance=1.2, initial_period=0.1, mmd=0.00613427

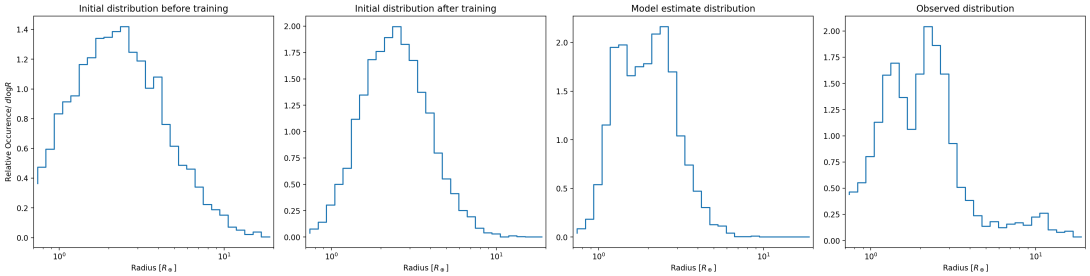




==========

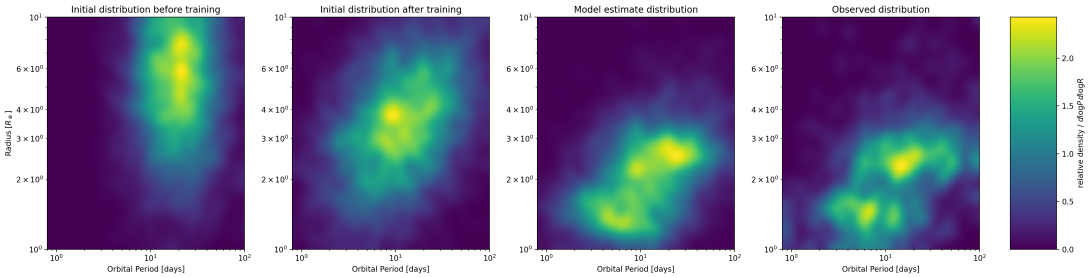
initial_distance=0.1, initial_period=-1.0, mmd=0.00625014

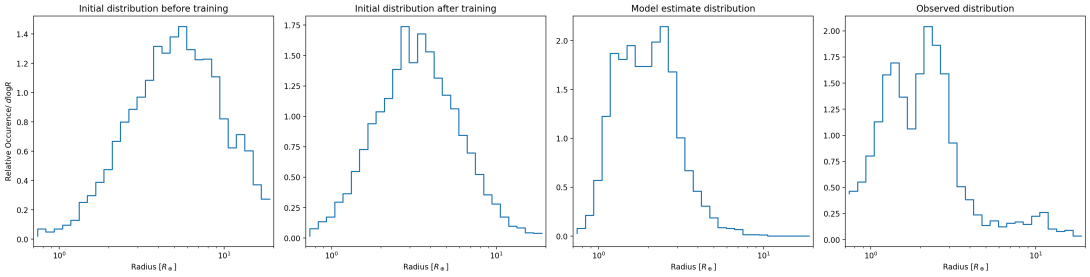




==========

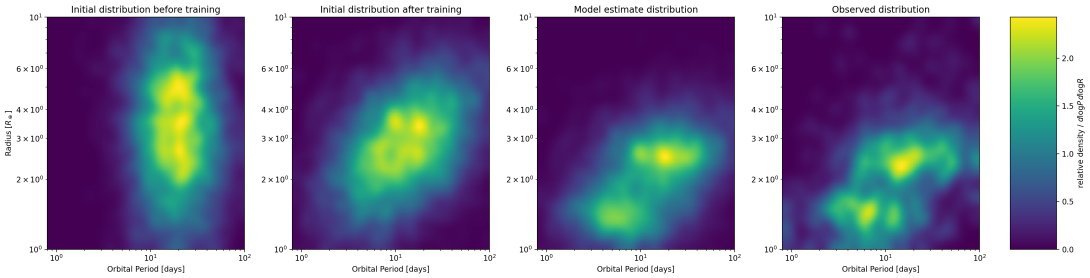
initial_distance=1.0, initial_period=0.5, mmd=0.00626731

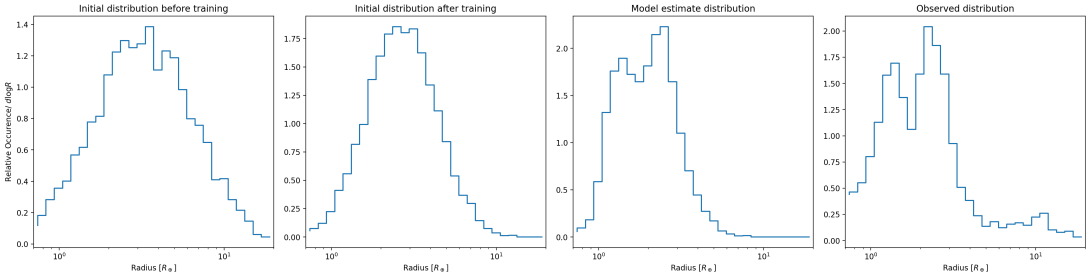




==========

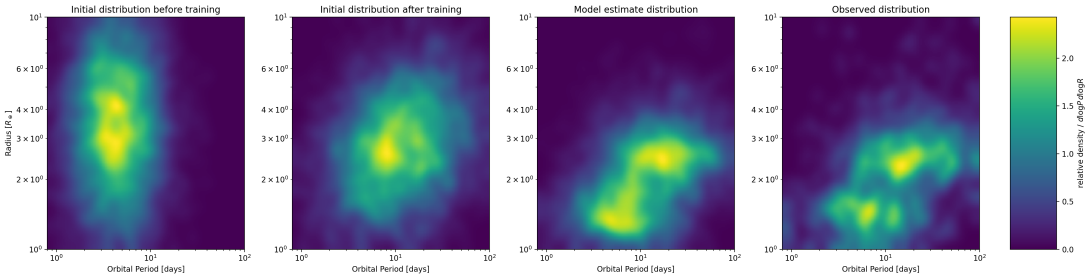
initial_distance=0.5, initial_period=0.5, mmd=0.00648832

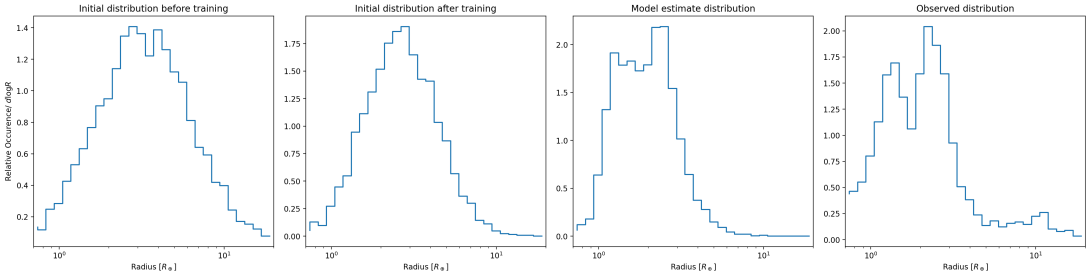




==========

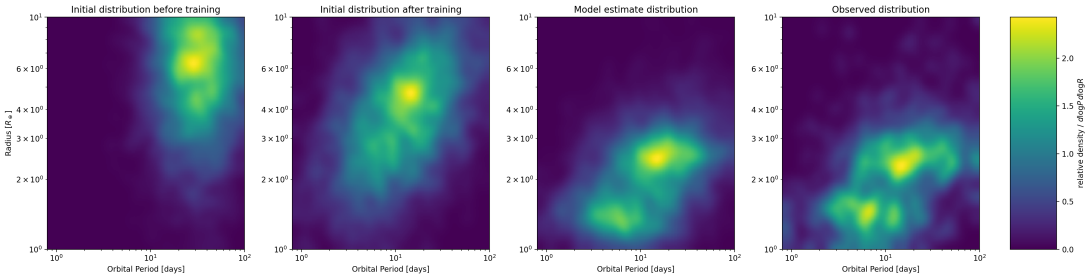
initial_distance=0.5, initial_period=-1.0, mmd=0.00659895

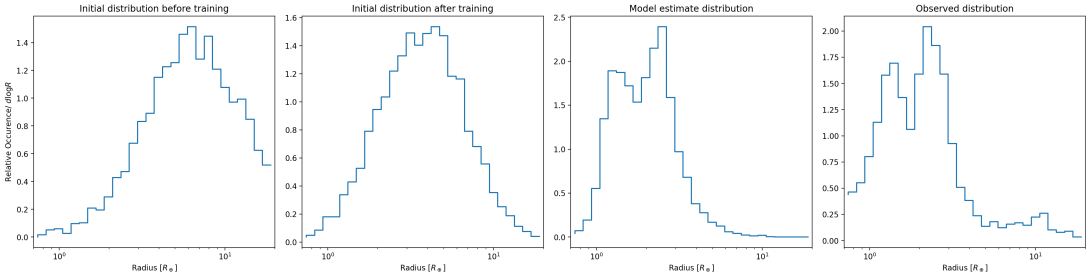




==========

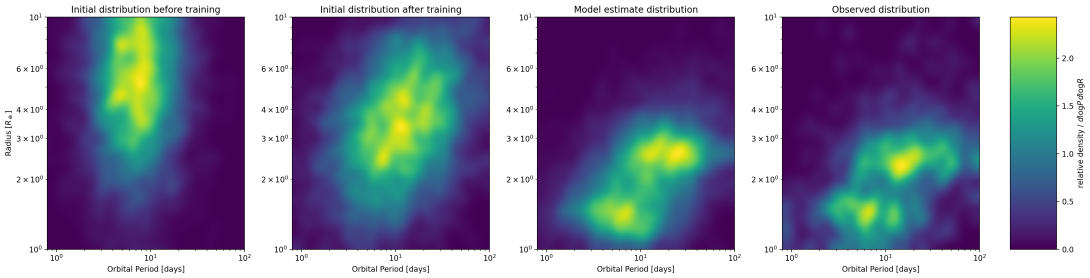
initial_distance=1.2, initial_period=1.0, mmd=0.00661993

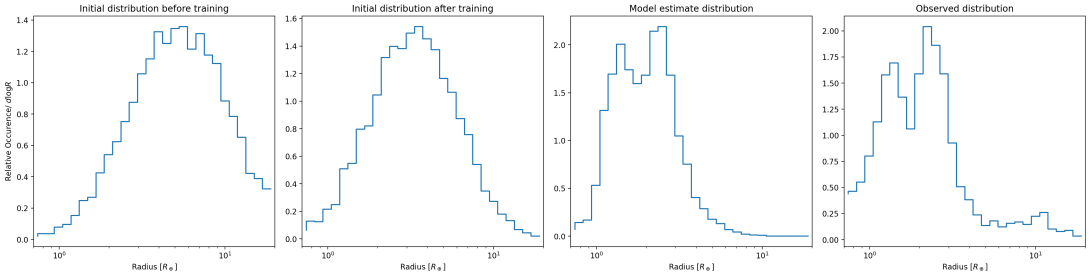




==========

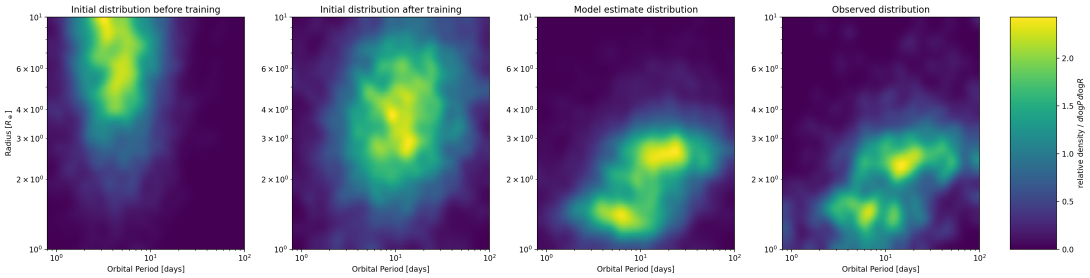
initial_distance=1.0, initial_period=-0.5, mmd=0.00731063

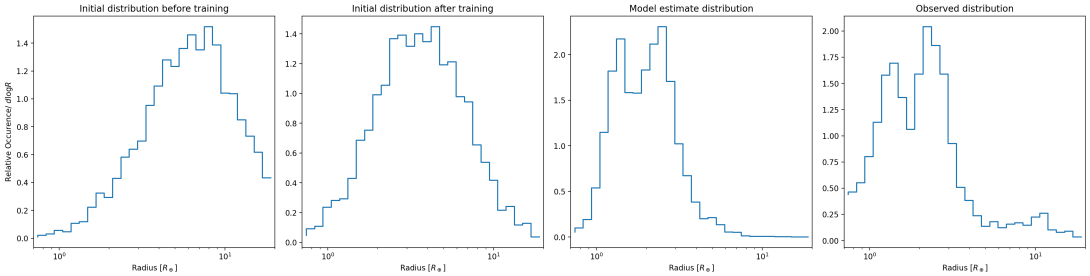




==========

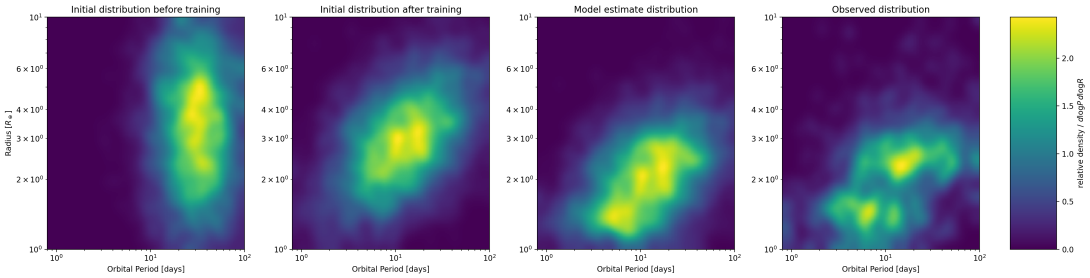
initial_distance=1.2, initial_period=-1.0, mmd=0.00748205

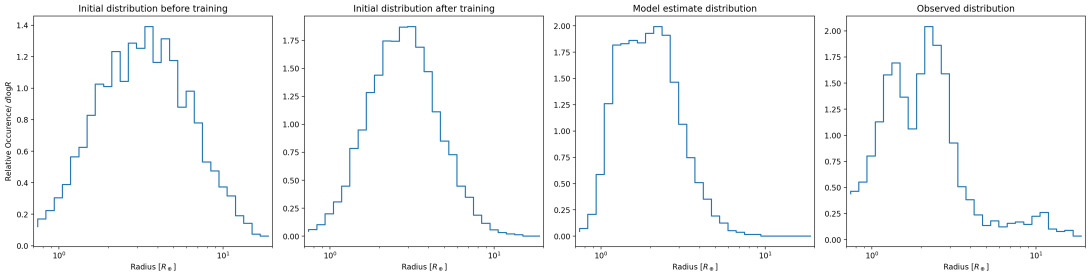




==========

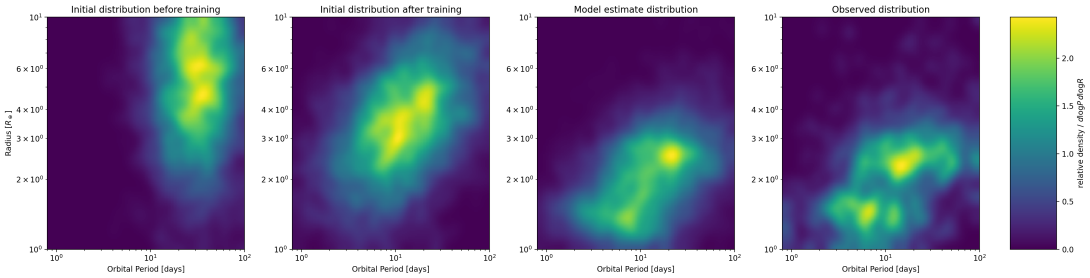
initial_distance=0.5, initial_period=1.0, mmd=0.00769472

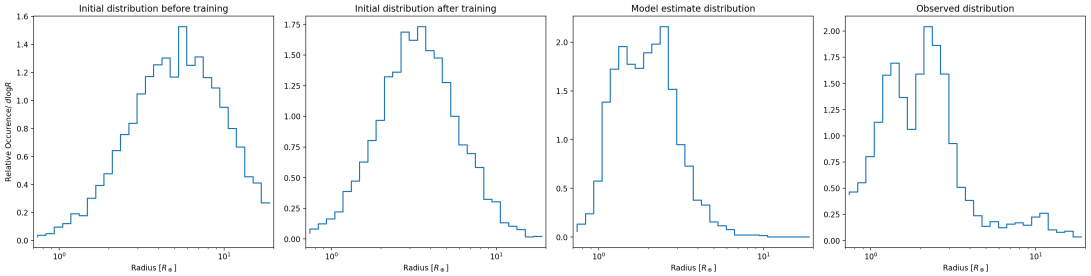




==========

initial_distance=1.0, initial_period=1.0, mmd=0.00839019





==========

initial_distance=0.5, initial_period=0.1, mmd=0.00939298