# Code

December 16, 2020

[13]:
```
%%capture null
!pip3 install arviz
import tensorflow as tf
import tensorflow_probability as tfp
import numpy as np
%matplotlib inline
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
plt.style.use('ggplot')

tf.compat.v1.reset_default_graph()

plt.rcParams['figure.figsize'] = [18, 10]
plt.rcParams['font.size'] = 16
plt.rcParams['lines.linewidth'] = 3

print("Num GPUs Available: ", len(tf.config.experimental.
 ↪list_physical_devices('GPU')))

# Distributions
tfd = tfp.distributions
tfb = tfp.bijectors

PI = tf.cast(tf.constant(np.pi), tf.float32)

intentions = dict(enumerate(["keep course", "starboard", "port"]))
ship_type = dict(enumerate(["overtaking", "head_on", "crossing"]))
```

[14]:
```
```

[15]:
```
def make_model(n=(), a=[1.0, 1.0, 1.0], no_rules=(2.0, 3.0), mix_std=(3.0, 1.
 ↪0), mix_loc=(PI/2, PI/6)):
  def mixture(loc, std, p_D, alpha):
    theta_dist_no_rules = tfd.Uniform(tf.broadcast_to(-PI, std.shape), PI)
    theta_dist_rules = tfd.Normal(
        tf.stack([tf.broadcast_to(tf.constant(0.0), loc.shape), -loc, loc], -1),
        tf.expand_dims(std, -1))
    theta_rules = tfd.MixtureSameFamily(
```

```
        tfd.Categorical(
            probs=alpha
        ),
        theta_dist_rules
    )
    probs = tf.stack([p_D, 1-p_D], axis=-1)
    return tfd.Sample(
        tfd.Mixture(
          tfd.Categorical(
              probs=probs
            ),
            [theta_dist_no_rules, theta_rules],
        ),
        sample_shape=n,
        name="theta"
      )
  return tfd.JointDistributionSequential([
  tfd.Dirichlet(a, name="alpha"),
  tfd.Beta(1.0, 1.0, name="p_D"),
  tfd.InverseGamma(*mix_std, name="std"),
  tfb.Scale(PI)(tfd.Beta(6.0, 6.0), name="loc"),
  mixture
  ])
```

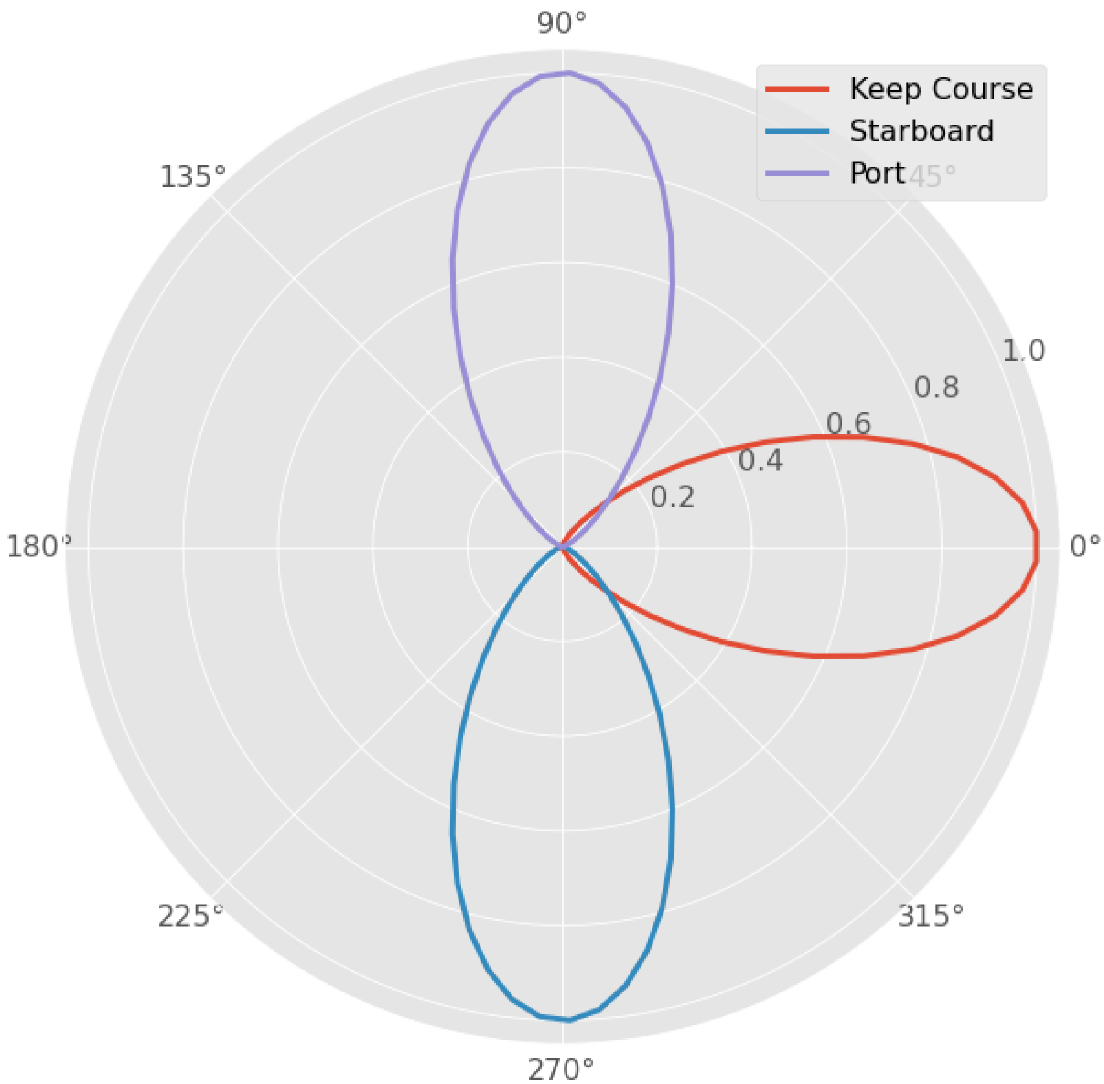
[15]:

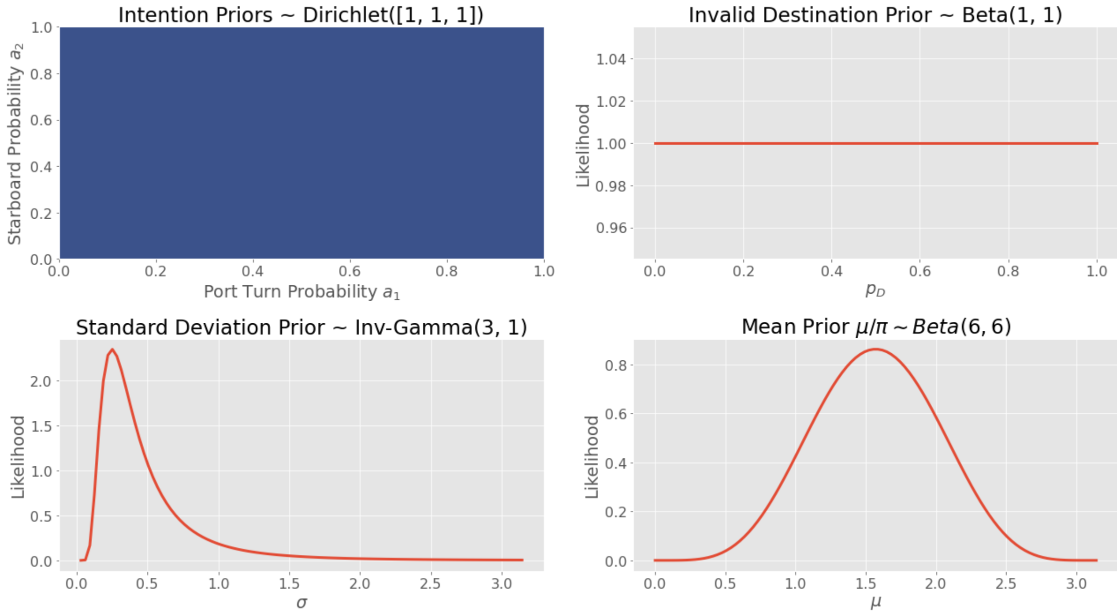
[16]:
```
def make_data(loc, std, p_no_rules, alpha, N=100, seed=1234):
  def mixture(std, p_no_rules, alpha):
    theta_dist_no_rules = tfd.Uniform(tf.broadcast_to(-PI, std.shape), PI)
    theta_dist_rules = tfd.Normal([0.0, -loc, loc], tf.expand_dims(std, -1))
    theta_rules = tfd.MixtureSameFamily(
        tfd.Categorical(
            probs=alpha
        ),
        theta_dist_rules
    )
    probs = tf.stack([p_no_rules, 1-p_no_rules], axis=-1)
    return tfd.Mixture(
        tfd.Categorical(
            probs=probs
          ),
          [theta_dist_no_rules, theta_rules],
          name="theta"
        )
  return mixture(std, p_no_rules, alpha).sample(N, seed=seed)
```

[18]:
```
joint = make_model()
n = 100
```

```
theta = np.linspace(-PI, PI, n)

theta_tf = tf.cast(theta, tf.float32)
std_tf = PI/8
loc_tf = PI/2
p_D_tf = 1e-8


fig = plt.figure(figsize=(10, 10))
fig.gca(polar=True)
probs1 = joint.prob(theta=theta_tf, alpha=[0.999, 0.0005, 0.0005], std=std_tf,␣
 ↪loc=PI/2, p_D=p_D_tf)
probs1 /= tf.reduce_max(probs1)
probs2 = joint.prob(theta=theta_tf, alpha=[0.0005, 0.999, 0.0005], std=std_tf,␣
 ↪loc=PI/2, p_D=p_D_tf)
probs2 /= tf.reduce_max(probs2)
probs3 = joint.prob(theta=theta_tf, alpha=[0.0005, 0.0005, 0.999], std=std_tf,␣
 ↪loc=PI/2, p_D=p_D_tf)
probs3 /= tf.reduce_max(probs3)
plt.plot(theta, probs1)
plt.plot(theta, probs2)
plt.plot(theta, probs3)
#plt.scatter(data, np.ones(len(data)))
plt.legend(["Keep Course", "Starboard", "Port"])
plt.savefig("intention_probs_by_angle.png")

fig, axs = plt.subplots(2, 2, figsize=(18, 10))
(alpha, p_no_rules, std, loc, _), _ = joint.sample_distributions()
X, Y = np.meshgrid(np.linspace(0, 1, n), np.linspace(0, 1, n))
a = tf.cast(tf.stack([1-X-Y, X, Y], axis=-1), tf.float32)
Z = alpha.prob(a)
axs[0][0].contourf(X, Y, Z, levels=100, )
axs[0][0].set_title("Intention Priors ~ Dirichlet([1, 1, 1])")
axs[0][0].set_xlabel("Port Turn Probability $a_1$")
axs[0][0].set_ylabel("Starboard Probability $a_2$")
x = np.linspace(0, 1, n)
axs[0][1].plot(x, p_no_rules.prob(x))
axs[0][1].set_title("Invalid Destination Prior ~ Beta(1, 1)")
axs[0][1].set_xlabel("$p_D$")
axs[0][1].set_ylabel("Likelihood")
x = np.linspace(0, PI, n)
axs[1][0].plot(x, std.prob(x))
axs[1][0].set_title("Standard Deviation Prior ~ Inv-Gamma(3, 1)")
axs[1][0].set_xlabel("$\sigma$")
axs[1][0].set_ylabel("Likelihood")
x = np.linspace(0, PI, n)
axs[1][1].plot(x, loc.prob(x))
```

```
axs[1][1].set_title("Mean Prior $\mu/ \pi \sim Beta(6, 6)$")
axs[1][1].set_xlabel("$\mu$")
axs[1][1].set_ylabel("Likelihood")
fig.tight_layout()
fig.savefig("priors.png")
```






[ ]:

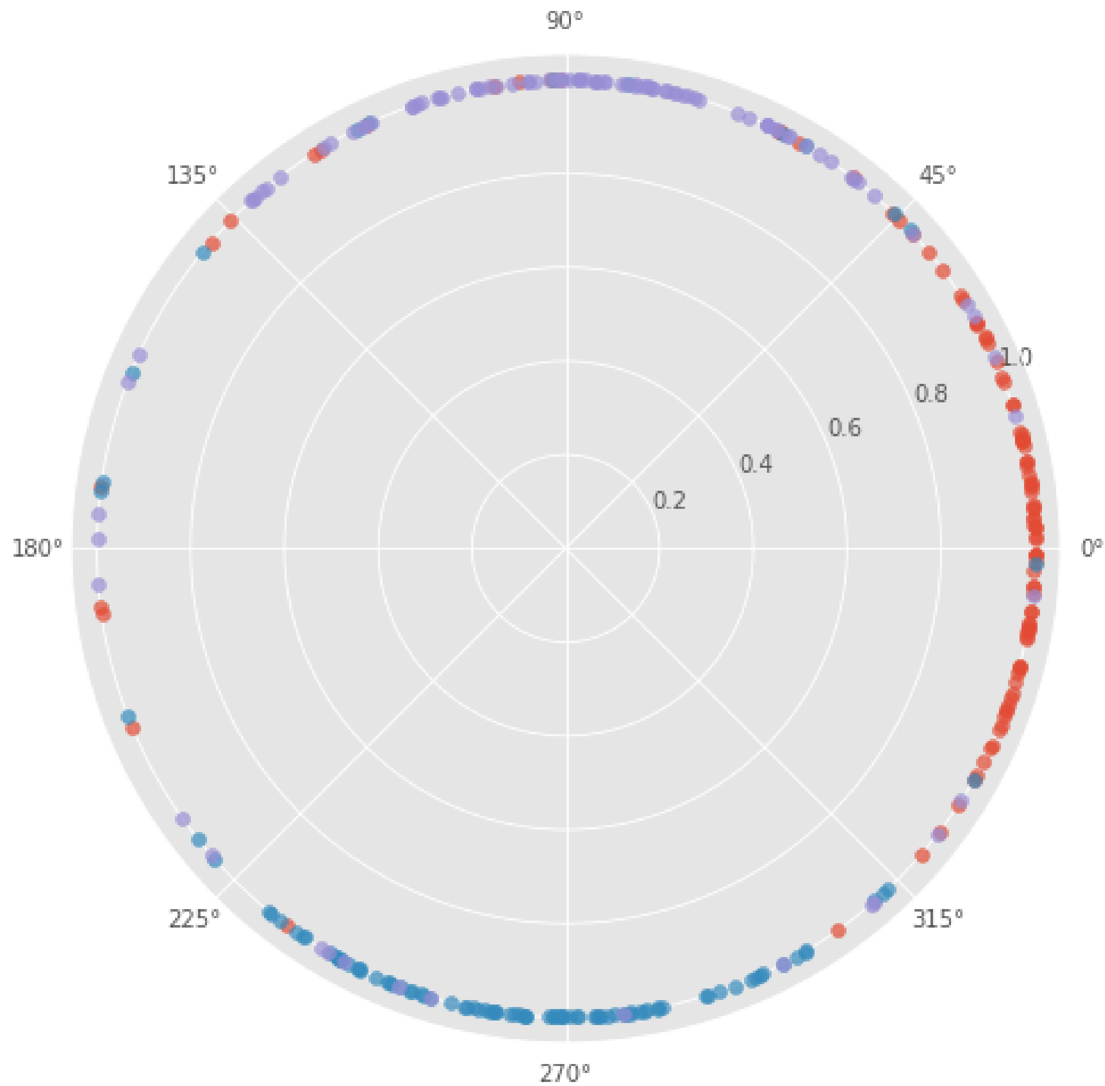
```
[ ]:
```

[ ]:

```python
data1 = make_data(PI/2, PI/8, 0.2, [0.9999, 0.00005, 0.00005])
data2 = make_data(PI/2, PI/8, 0.2, [0.00005, 0.9999, 0.00005])
data3 = make_data(PI/2, PI/8, 0.2, [0.00005, 0.00005, 0.9999])

fig = plt.figure(figsize=(14, 8))
fig.gca(polar=True)

plt.scatter(data1, np.ones(len(data1)), alpha=0.7)
plt.scatter(data2, np.ones(len(data2)), alpha=0.7)
plt.scatter(data3, np.ones(len(data3)), alpha=0.7)
```

[ ]:

```
<matplotlib.collections.PathCollection at 0x7f9adc3a65f8>
```




[ ]:
```
def target_log_prob(data, joint):
  def f(alpha, p_D, std, loc):
    return joint.log_prob(alpha=alpha, std=std, loc=loc, p_D=p_D, theta=data)
  return f

def vi(target_log_prob, joint):
  surrogate_posterior = tfp.experimental.vi.build_factored_surrogate_posterior(
    event_shape=joint.event_shape_tensor()[:-1],
    constraining_bijectors=[
                               tfb.SoftmaxCentered(),
                               tfb.Sigmoid(),
                               tfb.Softplus(),
                               tfb.SoftClip(low=0.0, high=PI)
                               ]
```

```
  )
  optimizer = tf.optimizers.Adam(learning_rate=1e-2)
  losses = tfp.vi.fit_surrogate_posterior(
    target_log_prob,
    surrogate_posterior,
    optimizer=optimizer,
    num_steps=1000,
    seed=42,
    sample_size=2
  )
  (alpha,
  p_D,
  std,
  loc), _ = surrogate_posterior.sample_distributions()
  return (alpha, p_D, std, loc), losses

@tf.function(autograph=False)
def mcmc_sample(target_log_prob, initial):
  burnin = 10000
  return tfp.mcmc.sample_chain(
    num_results=10000,
    num_burnin_steps=burnin,
    current_state=list(initial),
    num_steps_between_results=1,
    kernel=tfp.mcmc.DualAveragingStepSizeAdaptation(
        tfp.mcmc.TransformedTransitionKernel(
            inner_kernel=tfp.mcmc.HamiltonianMonteCarlo(
                target_log_prob_fn=target_log_prob,
                num_leapfrog_steps=2,
                step_size=0.5),
            bijector=[tfb.SoftmaxCentered(), tfb.Sigmoid(), tfb.Softplus(), tfb.
 ↪SoftClip(low=0.0, high=PI)]),
         num_adaptation_steps=int(0.8*burnin)),
    trace_fn=lambda _, pkr: pkr.inner_results.inner_results.is_accepted)
```

[ ]:
```
def mcmc_plot_chains(res, xlim=None):

  fig, axes = plt.subplots(3, 2)

  [alpha, p_D, std, loc] = res[0]
  colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
  axes[0][0].plot(alpha[:, :, 0])
  axes[1][0].plot(alpha[:, :, 1])
  axes[2][0].plot(alpha[:, :, 2])
  axes[0][0].set_title(f"Intention Probabilitity: {intentions[0]}")
  axes[0][0].set_xlabel("Step")
  axes[1][0].set_title(f"Intention Probabilitity: {intentions[1]}")
```

```python
  axes[1][0].set_xlabel("Step")
  axes[2][0].set_title(f"Intention Probabilitity: {intentions[2]}")
  axes[2][0].set_xlabel("Step")

  axes[0][1].plot(p_D)
  axes[0][1].set_title("Invalid Destination Probability")
  axes[0][1].set_xlabel("Step")
  axes[1][1].plot(std)
  axes[1][1].set_title("Standard Deviation")
  axes[1][1].set_xlabel("Step")
  axes[2][1].plot(loc)
  axes[2][1].set_title("Mean $\mu$")
  axes[2][1].set_xlabel("Step")
  if xlim is not None:
    for aa in axes:
      for a in aa:
        a.set_xlim(xlim)

def mcmc_plot_alpha(res, sim, ax, i, with_gt=True, alpha=1, ymax=None):
  alpha = res[0][0]
  alpha_samples = tf.reshape(alpha, (-1, 3))

  h, b = np.histogram(alpha_samples[:, i], bins=50, density=True)
  b = np.concatenate([[0], b[:-1], [1]])
  h = np.concatenate([[0], h, [0]])
  ax.plot(b, h)
  ax.fill_between(b, h, alpha=0.4)
  ax.set_title(f"Intention Probability: {intentions[i]}")
  ax.set_xlim([0, 1])
  if ymax is not None:
    ax.set_ylim([0, ymax])
  ax.set_ylabel("Likelihood")
  ax.set_xlabel("$a$")
  if with_gt:
    lim = ax.get_ylim()
    ax.plot([sim.alpha[i]]*2, [-20, 1000], color="black", scaley=False,␣
 ↪scalex=False)
    ax.set_ylim(lim)
    ax.legend(["Estimate", "True value"])

def mcmc_plot_p_D(res, sim, ax, with_gt=True, alpha=1, ymax=None):
  p_D = res[0][1]
  p_D = tf.reshape(p_D, -1)
  h, b = np.histogram(p_D, bins=50, density=True)
  b = np.concatenate([[0], b[:-1], [1]])
  h = np.concatenate([[0], h, [0]])
  ax.plot(b, h)
```

```
  ax.fill_between(b, h, alpha=0.4)
  if with_gt:
    lim = ax.get_ylim()
    ax.plot([sim.p_D]*2, [-20, 1000], color="black", scaley=False, scalex=False)
    ax.set_ylim(lim)
    ax.legend(["Estimate", "True value"])
  ax.set_xlim([0, 1])
  if ymax is not None:
    ax.set_ylim([0, ymax])
  ax.set_title("Invalid Destination Probability")
  ax.set_ylabel("Likelihood")

def mcmc_plot_std(res, sim, ax, with_gt=True, xr=2, alpha=1, ymax=None):
  std = res[0][2]
  std = tf.reshape(std, -1)
  h, b = np.histogram(std, bins=50, density=True)
  b = np.concatenate([[0], b[:-1], [xr*sim.std]])
  h = np.concatenate([[0], h, [0]])
  ax.plot(b, h)
  ax.fill_between(b, h, alpha=0.4)
  ax.set_xlim([0, 1])
  ax.set_xlabel("$\sigma$")
  ax.set_title("Standard Deviation")
  if with_gt:
    lim = ax.get_ylim()
    ax.plot([sim.std]*2, [-20, 1000], color="black", scalex=False, scaley=False)
    ax.set_ylim(lim)
    ax.legend(["Estimate", "True value"])
  ax.set_xlim([0, xr*sim.std])
  if ymax is not None:
    ax.set_ylim([0, ymax])
  ax.set_ylabel("Likelihood")

def mcmc_plot_loc(res, sim, ax, with_gt=True, xr=PI/4, alpha=1, ymax=None):
    loc = res[0][3]
    loc=tf.reshape(loc, -1)
    h, b = np.histogram(loc, bins=50, density=True)
    b = np.concatenate([[0], b[:-1], [sim.loc+xr]])
    h = np.concatenate([[0], h, [0]])
    ax.plot(b, h)
    ax.fill_between(b, h, alpha=0.4)
    ax.set_title("Mean $\mu$")
    if with_gt:
      lim = ax.get_ylim()
      ax.plot([sim.loc]*2, [-20, 1000], color="black", scaley=False)
      ax.set_ylim(lim)
      ax.legend(["Estimate", "True value"])
```

```
    ax.set_xlim([sim.loc - xr, sim.loc + xr])
    if ymax is not None:
      ax.set_ylim([0, ymax])
    ax.set_xlabel("$\mu$")
    ax.set_ylabel("Likelihood")

def vi_plot_alpha(res, sim, ax, i, with_gt=True):
  alpha = res[0][0]
  alpha_samples = alpha.sample(100000)
  h, b = np.histogram(alpha_samples[:, i], bins=50, density=True)
  b = np.concatenate([[0], b[:-1], [1]])
  h = np.concatenate([[0], h, [0]])
  ax.plot(b, h)
  ax.fill_between(b, h, alpha=0.4)
  ax.set_xlim([0, 1])
  if with_gt:
    lim = ax.get_ylim()
    ax.plot([sim.alpha[i]]*2, [-20, 1000], color="black", scalex=False,␣
 ↪scaley=False)
    ax.set_ylim(lim)
    ax.legend(["Estimate", "True value"])
  ax.set_title(f"Intention Probability: {intentions[i]}")
  ax.set_xlabel("$a$")
  ax.set_ylabel("Likelihood")


def vi_plot_p_D(res, sim, ax, with_gt=True):
  [_, p_D, _, _], _ = res
  x = np.linspace(0, 1, 1000)
  ax.plot(x, p_D.prob(x))
  ax.fill_between(x, p_D.prob(x), alpha=0.4)
  ax.set_xlim([0, 1])
  if with_gt:
    lim = ax.get_ylim()
    ax.plot([sim.p_D]*2, [-20, 1000], color="black", scalex=False, scaley=False)
    ax.set_ylim(lim)
    ax.legend(["Estimate", "True value"])
  ax.set_title("Invalid Destination Probability")
  ax.set_xlabel("$p_D$")
  ax.set_ylabel("Likelihood")


def vi_plot_std(res, sim, ax, with_gt=True):
  [_, _, std, _], _ = res
  x = np.linspace(0, PI, 10000)
  ax.plot(x, std.prob(x))
  ax.fill_between(x, std.prob(x), alpha=0.4)
```

```
  ax.set_xlim([0, 2*sim.std])
  if with_gt:
    lim = ax.get_ylim()
    ax.plot([sim.std]*2, [-20, 1000], color="black", scalex=False, scaley=False)
    ax.set_ylim(lim)
    ax.legend(["Estimate", "True value"])
  ax.set_title("Standard Deviation")
  ax.set_xlabel("$\sigma$")
  ax.set_ylabel("Likelihood")


def vi_plot_loc(res, sim, ax, with_gt=True):
  [_, _, _, loc], _ = res
  x = np.linspace(0, PI, 1000)
  ax.plot(x, loc.prob(x))
  ax.fill_between(x, loc.prob(x), alpha=0.4)
  ax.set_xlim([sim.loc - PI/4, sim.loc + PI/4])
  ax.set_xlabel("$\mu$")
  ax.set_ylabel("Likelihood")
  if with_gt:
    lim = ax.get_ylim()
    ax.plot([sim.loc]*2, [-20, 1000], color="black", scaley=False, scalex=False)
    ax.set_ylim(lim)
    ax.legend(["Estimate", "True value"])
  ax.set_title("Mean $\mu$")

class MCMCResult():
  def __init__(self, res, sim):
    self.res = res
    self.sim = sim

  def plot_chains(self, *args, **kwargs):
    mcmc_plot_chains(self.res, *args, **kwargs)

  def plot_alpha(self, *args, **kwargs):
    mcmc_plot_alpha(self.res, self.sim, *args, **kwargs)

  def plot_p_D(self, *args, **kwargs):
    mcmc_plot_p_D(self.res, self.sim, *args, **kwargs)

  def plot_std(self, *args, **kwargs):
    mcmc_plot_std(self.res, self.sim, *args, **kwargs)

  def plot_loc(self, *args, **kwargs):
    mcmc_plot_loc(self.res, self.sim, *args, **kwargs)

  def plot(self, figname=None):
```

```
    fig, axs = plt.subplots(3,2)
    self.plot_alpha(axs[0][0], 0)
    self.plot_alpha(axs[1][0], 1)
    self.plot_alpha(axs[2][0], 2)
    self.plot_p_D(axs[0][1])
    self.plot_std(axs[1][1])
    self.plot_loc(axs[2][1])
    fig.tight_layout()
    if figname is not None:
      fig.savefig(figname)

class VIResult():
  def __init__(self, res, sim):
    self.res = res
    self.sim = sim

  def plot_losses(self, figname=None):
    losses = self.res[1]
    fig, ax = plt.subplots(1, 1)
    ax.plot(losses)
    ax.set_xlabel("Iteration")
    ax.set_ylabel("Negative ELBO")

  def plot_alpha(self, *args, **kwargs):
    vi_plot_alpha(self.res, self.sim, *args, **kwargs)

  def plot_p_D(self, *args, **kwargs):
    vi_plot_p_D(self.res, self.sim, *args, **kwargs)


  def plot_std(self, *args, **kwargs):
    vi_plot_std(self.res, self.sim, *args, **kwargs)

  def plot_loc(self, *args, **kwargs):
    vi_plot_loc(self.res, self.sim, *args, **kwargs)

  def plot(self, figname=None):
    fig, axs = plt.subplots(3, 2)
    self.plot_alpha(axs[0][0], 0)
    self.plot_alpha(axs[1][0], 1)
    self.plot_alpha(axs[2][0], 2)
    self.plot_p_D(axs[0][1])
    self.plot_std(axs[1][1])
    self.plot_loc(axs[2][1])
    fig.tight_layout()
    if figname is not None:
      fig.savefig(figname)
```

```python
class Simulation():
  def __init__(self, alpha, p_D, std, loc, N, seed=1000):
    self.loc = tf.constant(loc)
    self.std = tf.constant(std)
    self.p_D = tf.constant(p_D)
    self.alpha = tf.constant(alpha)
    self.N = tf.constant(N)
    self.data_ = None
    self.mcmc_cache_ = None
    self.vi_cache_ = None
    self.joint = make_model(n=N)
    self.seed=seed

  @property
  def data(self):
    if self.data_ is None:
      self.data_ = make_data(self.loc, self.std, self.p_D, self.alpha, self.N,␣
↪seed=self.seed)
    return self.data_

  @property
  def target_log_prob(self):
    return target_log_prob(self.data, self.joint)

  def visualize_data(self):
    fig = plt.figure()
    fig.gca(polar=True)
    plt.scatter(self.data, np.ones(len(self.data)))

  def mcmc(self):
    if self.mcmc_cache_ is None:
      initial = self.joint.sample(6)[:-1]
      self.mcmc_cache_ = MCMCResult(mcmc_sample(self.target_log_prob, initial),␣
↪self)
    return self.mcmc_cache_

  def vi(self):
    if self.vi_cache_ is None:
      self.vi_cache_ = VIResult(vi(self.target_log_prob, self.joint), self)
    return self.vi_cache_
```

[ ]:
```python
sim1 = Simulation([0.5, 0.3, 0.2], 0.3, 0.6, PI/3, 1000)
```

[ ]:
```python
%%time
sim1.vi()
```

```
CPU times: user 15.3 s, sys: 2.52 s, total: 17.8 s
Wall time: 13.3 s
```

[ ]: <__main__.VIResult at 0x7f9adc36cdd8>

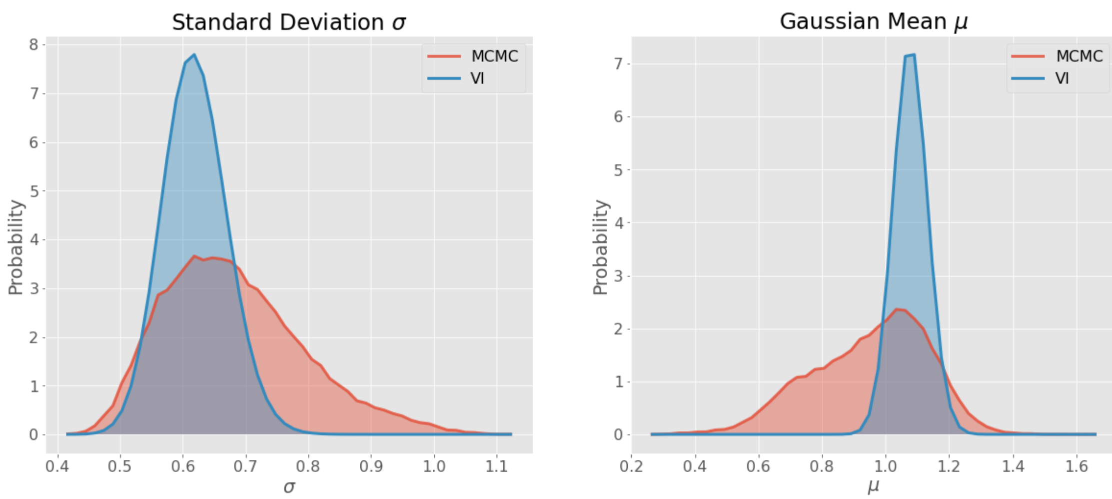
[ ]:
```
%%time
sim1.mcmc()
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/tensorflow_probability/python/mcmc/kernel.py:104: calling
HamiltonianMonteCarlo.__init__ (from tensorflow_probability.python.mcmc.hmc)
with step_size_update_fn is deprecated and will be removed after 2019-05-22.
Instructions for updating:
The `step_size_update_fn` argument is deprecated. Use
`tfp.mcmc.SimpleStepSizeAdaptation` instead.
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-
packages/tensorflow/python/util/deprecation.py:507: calling
HamiltonianMonteCarlo.__init__ (from tensorflow_probability.python.mcmc.hmc)
with seed is deprecated and will be removed after 2020-09-20.
Instructions for updating:
The `seed` argument is deprecated (but will work until removed). Pass seed to
`tfp.mcmc.sample_chain` instead.
CPU times: user 10min 54s, sys: 2min 7s, total: 13min 1s
Wall time: 8min 5s
```

[ ]: <__main__.MCMCResult at 0x7f9adc363a90>

[ ]:
```
fig = plt.figure(figsize=(20, 8))
plt.subplot(1, 2, 1)
mcmc_std = sim1.mcmc().res[0][2]
h0, b0 = np.histogram(mcmc_std, bins=50, density=True)
vi_std = sim1.vi().res[0][2]
plt.plot(b0[:-1], h0, alpha=0.8)
plt.plot(b0[:-1], vi_std.prob(b0[:-1]))
plt.fill_between(b0[:-1], h0, alpha=0.4)
plt.fill_between(b0[:-1], vi_std.prob(b0[:-1]), alpha=0.4)
plt.title("Standard Deviation $\sigma$")
plt.xlabel("$\sigma$")
plt.ylabel("Probability")
plt.legend(["MCMC", "VI"])

plt.subplot(1, 2, 2)
mcmc_loc = sim1.mcmc().res[0][3]
h0, b0 = np.histogram(mcmc_loc, bins=50, density=True)
vi_loc = sim1.vi().res[0][3]
plt.plot(b0[:-1], h0, alpha=0.8)
plt.plot(b0[:-1], vi_loc.prob(b0[:-1]))
plt.fill_between(b0[:-1], h0, alpha=0.4)
```

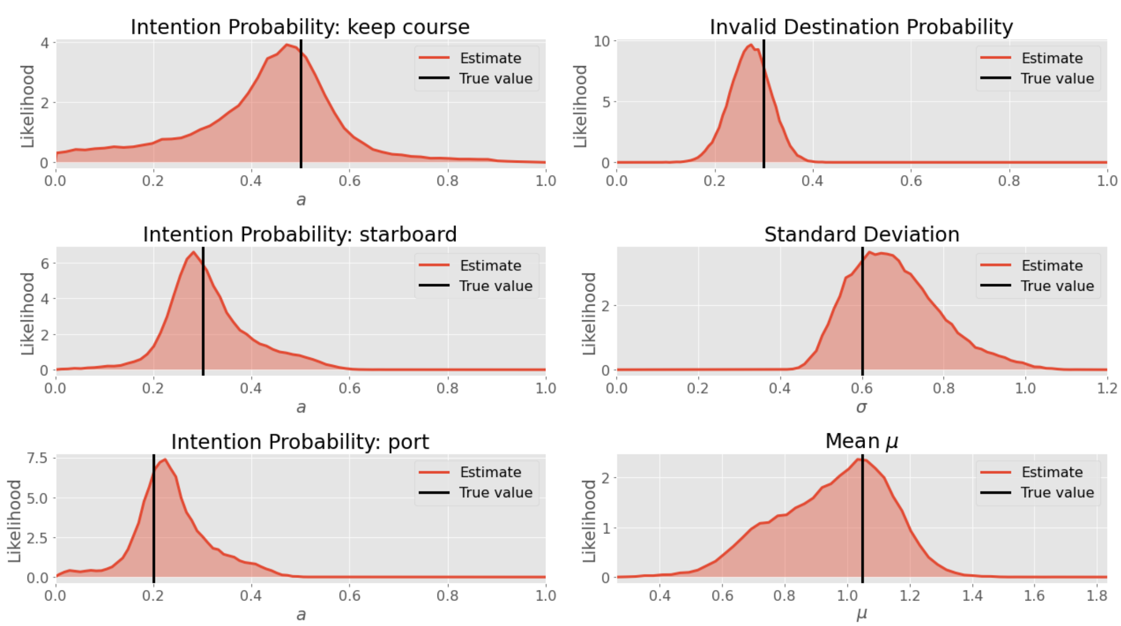
```
plt.fill_between(b0[:-1], vi_loc.prob(b0[:-1]), alpha=0.4)
plt.title("Gaussian Mean $\mu$")
plt.xlabel("$\mu$")
plt.ylabel("Probability")
plt.legend(["MCMC", "VI"])

plt.savefig("example_vi_mcmc_comparison.png")
```



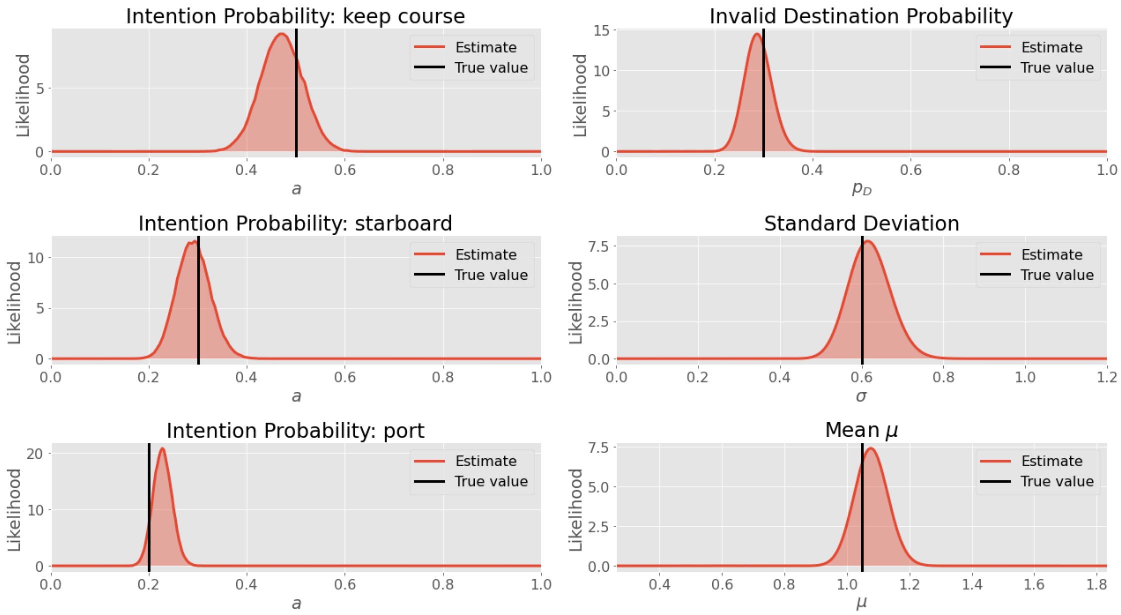

[ ]:
```
tf.reduce_mean(tf.cast(sim1.mcmc().res[1], tf.float32))
```

[ ]: <tf.Tensor: shape=(), dtype=float32, numpy=0.7475>

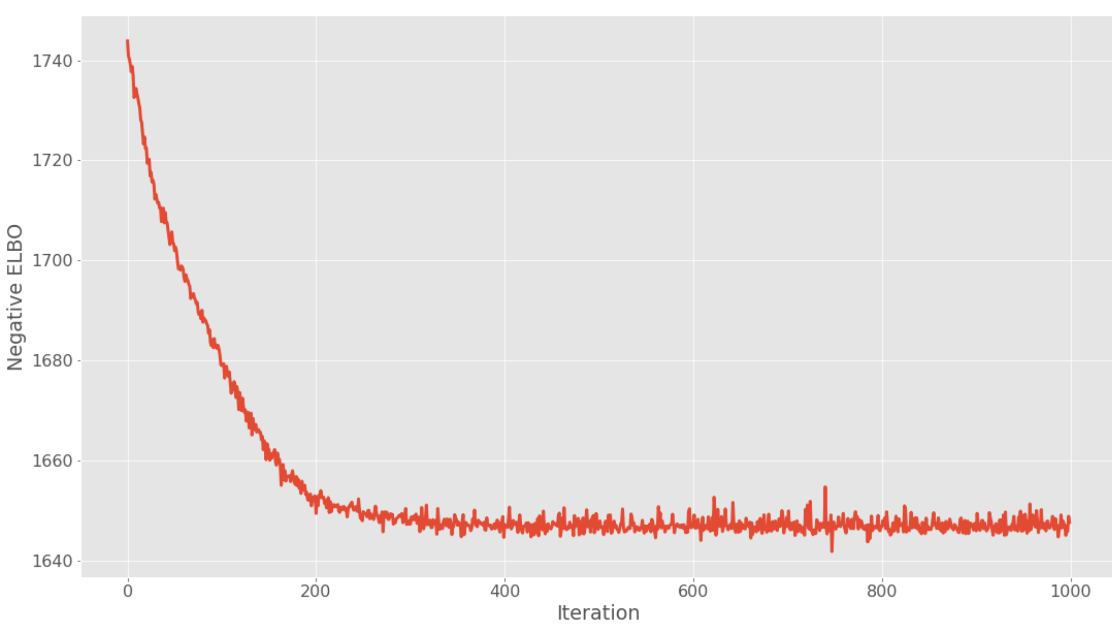
[ ]:
```
sim1.mcmc().plot(figname="example_mcmc.png")
#
```



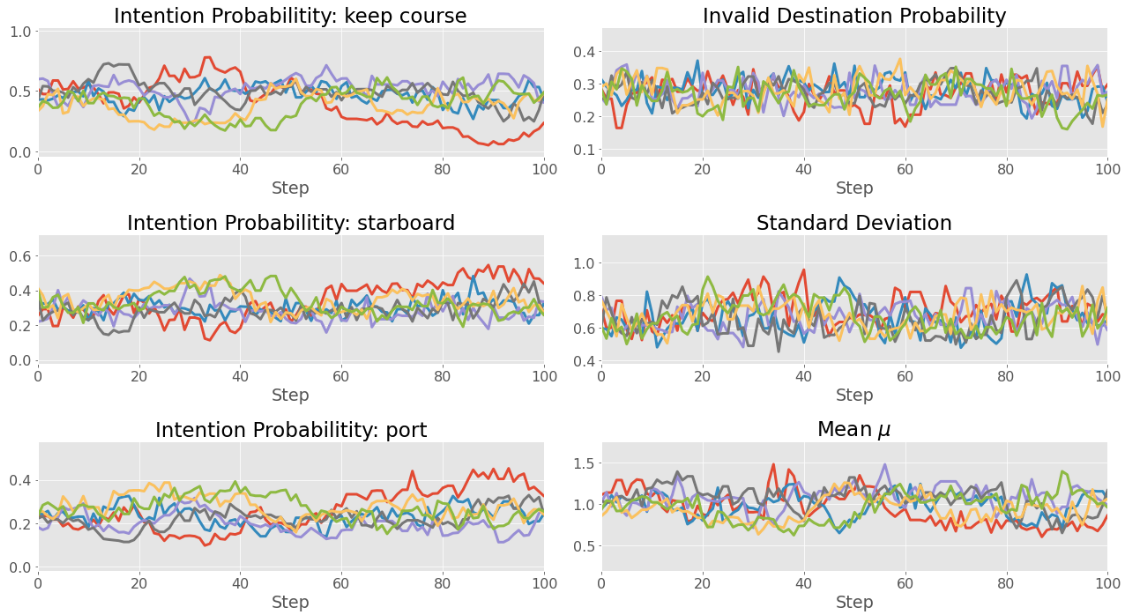
[ ]:
```
sim1.vi().plot(figname="example_vi.png")
```




[ ]:
```
sim1.vi().plot_losses(figname="example_vi_losses.png")
plt.savefig("example_vi_losses.png")
```



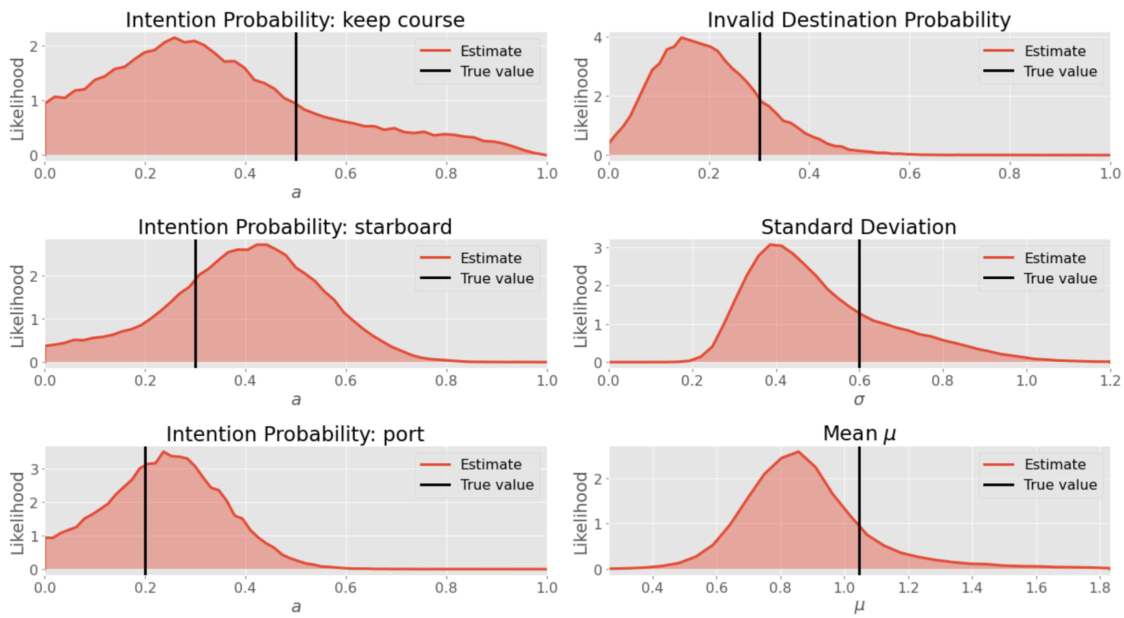
```
[ ]: mcmc_plot_chains(sim1.mcmc().res, xlim=[0, 100])
     plt.tight_layout()
     plt.savefig("example_mcmc_trace.png")
```



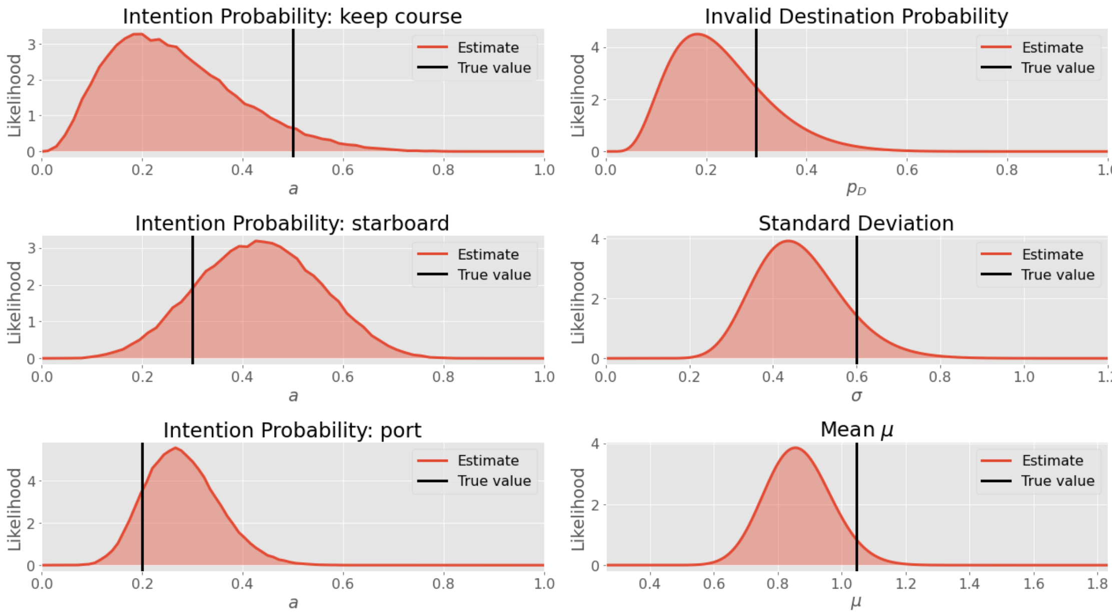

```
[ ]: sim2 = Simulation([0.5, 0.3, 0.2], 0.3, 0.6, PI/3, 50)
```

```
[ ]: m2 = sim2.mcmc()
     v2 = sim2.vi()
```

```
[ ]: m2.plot(figname="example_mcmc_low_N.png")
```

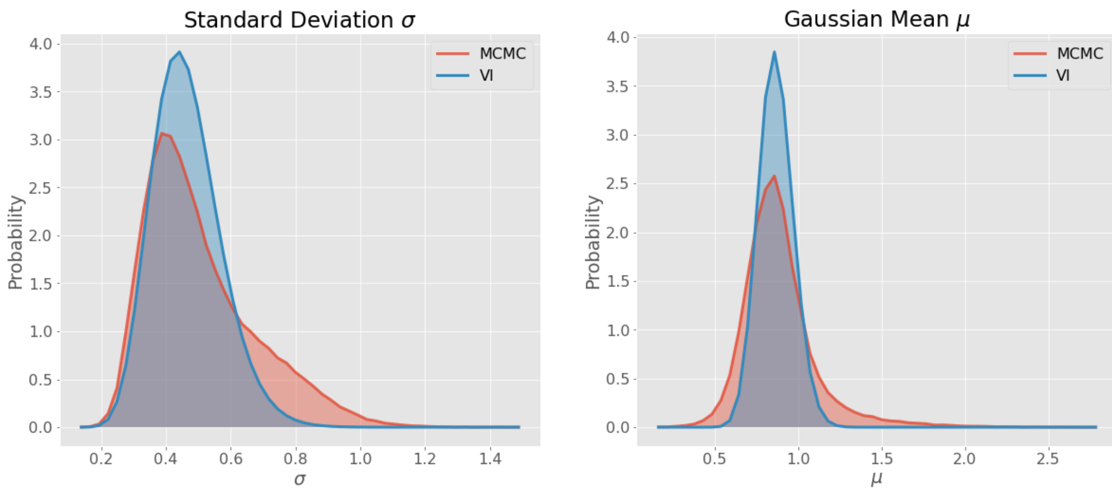


[ ]:
```
v2.plot(figname="example_vi_low_N.png")
```




[ ]:
```
fig = plt.figure(figsize=(20, 8))
plt.subplot(1, 2, 1)
mcmc_std = sim2.mcmc().res[0][2]
h0, b0 = np.histogram(mcmc_std, bins=50, density=True)
vi_std = sim2.vi().res[0][2]
plt.plot(b0[:-1], h0, alpha=0.8)
plt.plot(b0[:-1], vi_std.prob(b0[:-1]))
plt.fill_between(b0[:-1], h0, alpha=0.4)
plt.fill_between(b0[:-1], vi_std.prob(b0[:-1]), alpha=0.4)
plt.title("Standard Deviation $\sigma$")
plt.xlabel("$\sigma$")
plt.ylabel("Probability")
plt.legend(["MCMC", "VI"])

plt.subplot(1, 2, 2)
mcmc_loc = sim2.mcmc().res[0][3]
h0, b0 = np.histogram(mcmc_loc, bins=50, density=True)
vi_loc = sim2.vi().res[0][3]
plt.plot(b0[:-1], h0, alpha=0.8)
plt.plot(b0[:-1], vi_loc.prob(b0[:-1]))
plt.fill_between(b0[:-1], h0, alpha=0.4)
plt.fill_between(b0[:-1], vi_loc.prob(b0[:-1]), alpha=0.4)
plt.title("Gaussian Mean $\mu$")
plt.xlabel("$\mu$")
```

```
plt.ylabel("Probability")
plt.legend(["MCMC", "VI"])

plt.savefig("example_vi_mcmc_comparison_low_N.png")
```




[ ]:

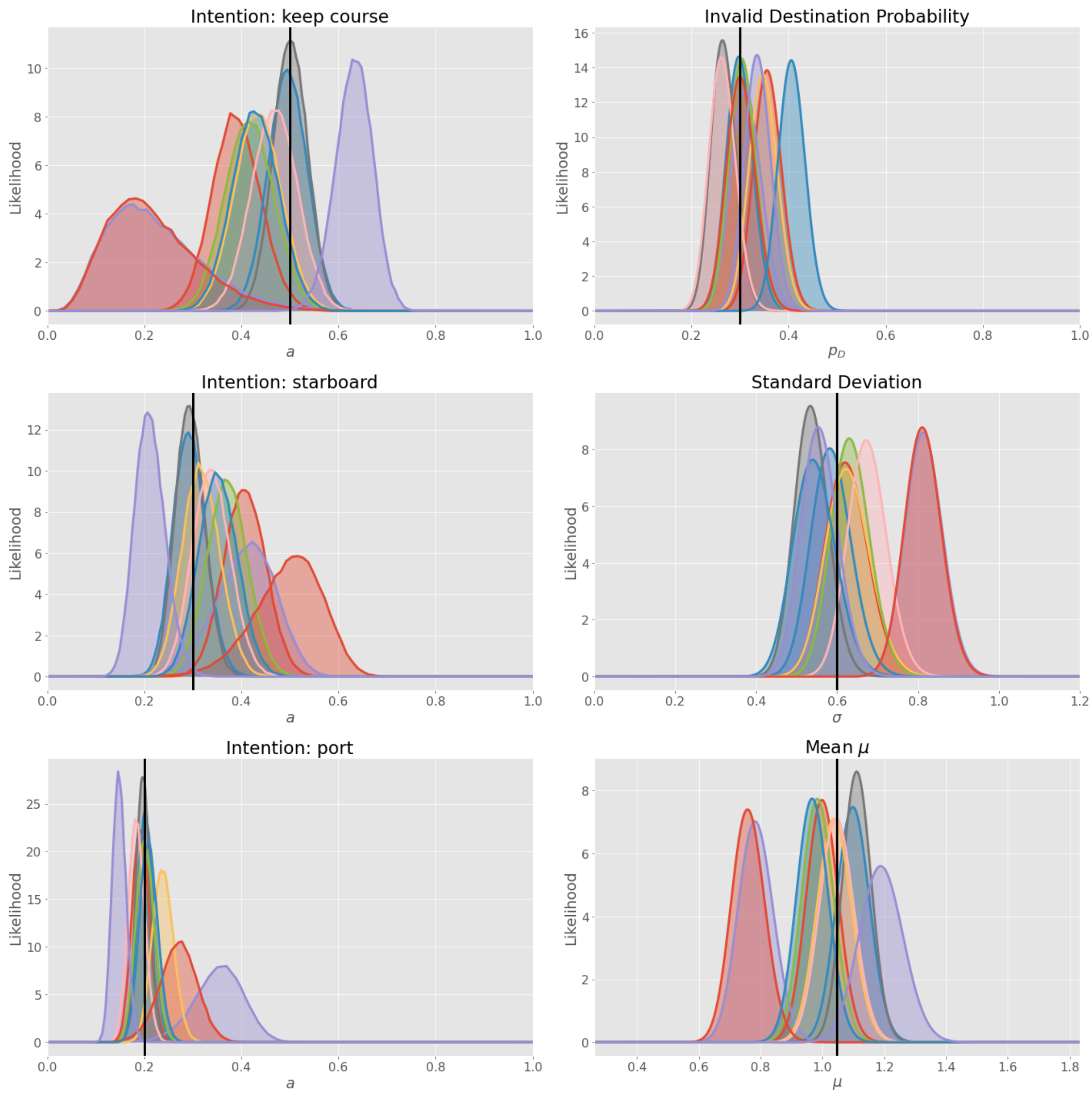
```
mc_trials = 10
sims = [Simulation([0.5, 0.3, 0.2], 0.3, 0.6, PI/3, 10000, seed=seed) for seed
 ↪in range(mc_trials)]

for i, sim in enumerate(sims):
  sim.vi()
  sim.mcmc()
  print(f"Sim {i+1} complete")
```

```
WARNING:tensorflow:11 out of the last 11 calls to <function mcmc_sample at
0x7f9adc381158> triggered tf.function retracing. Tracing is expensive and the
excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has experimental_relax_shapes=True
option that relaxes argument shapes that can avoid unnecessary retracing. For
(3), please refer to https://www.tensorflow.org/tutorials/customization/performa
nce#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
Sim 1 complete
WARNING:tensorflow:11 out of the last 11 calls to <function mcmc_sample at
0x7f9adc381158> triggered tf.function retracing. Tracing is expensive and the
excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
```

```
outside of the loop. For (2), @tf.function has experimental_relax_shapes=True
option that relaxes argument shapes that can avoid unnecessary retracing. For
(3), please refer to https://www.tensorflow.org/tutorials/customization/performa
nce#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
Sim 2 complete
WARNING:tensorflow:11 out of the last 11 calls to <function mcmc_sample at
0x7f9adc381158> triggered tf.function retracing. Tracing is expensive and the
excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has experimental_relax_shapes=True
option that relaxes argument shapes that can avoid unnecessary retracing. For
(3), please refer to https://www.tensorflow.org/tutorials/customization/performa
nce#python_or_tensor_args and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
Sim 3 complete
```

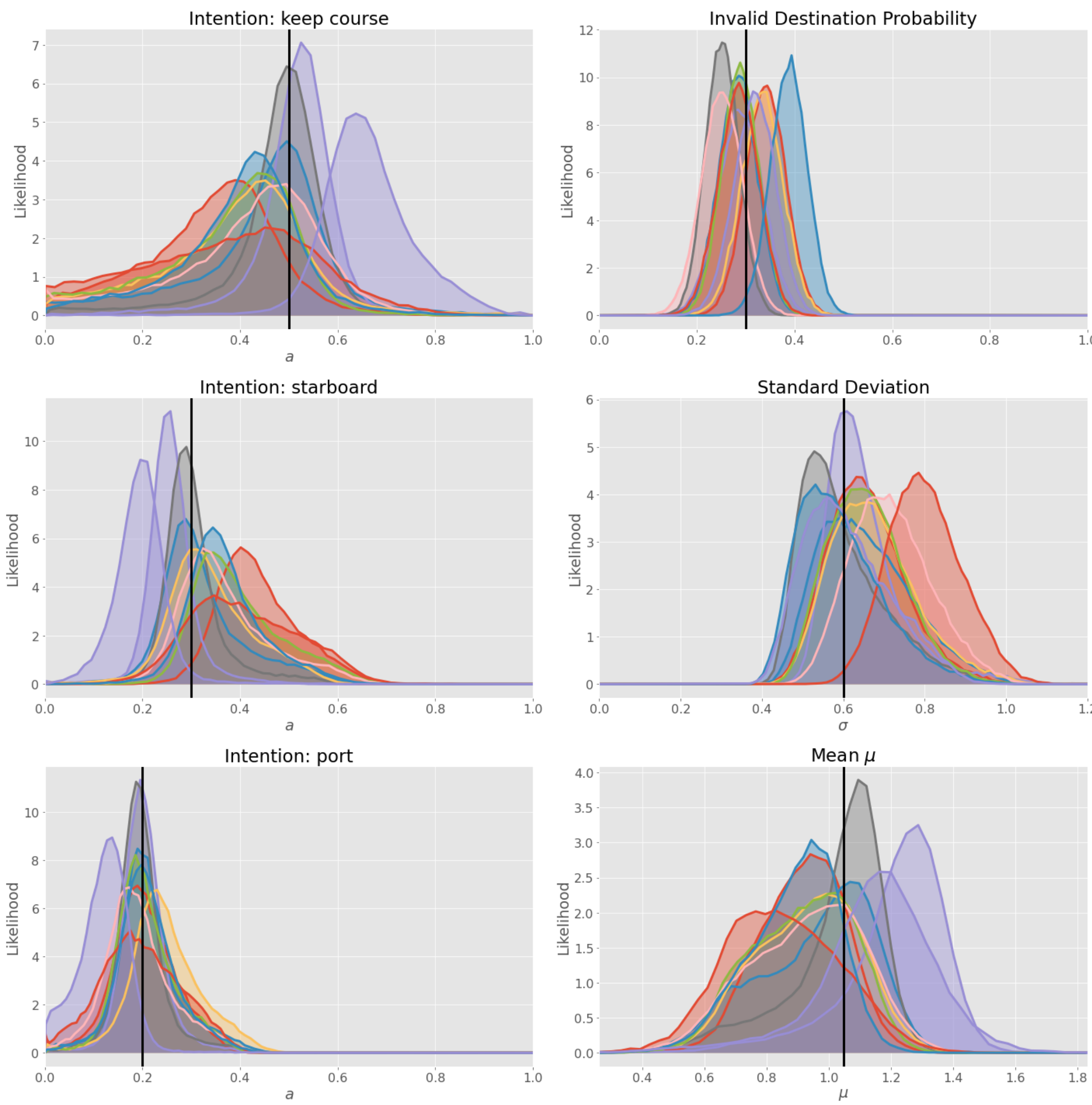
[ ]:
```
fig, ((aax1, ax1),(aax2, ax2),(aax3, ax3)) = plt.subplots(3,2, figsize=(20, 20))
for sim in sims:
  res = sim.vi()
  alpha = res.res[0][0]
  alpha_samples = alpha.sample(10000)
  res.plot_alpha(aax1, 0, with_gt=False)
  res.plot_alpha(aax2, 1, with_gt=False)
  res.plot_alpha(aax3, 2, with_gt=False)
  res.plot_p_D(ax1, with_gt=False)
  res.plot_std(ax2, with_gt=False)
  res.plot_loc(ax3, with_gt=False)
fig.tight_layout()

aax1.plot([0.5]*2, [-10, 100], scalex=False, scaley=False, color="black",␣
 ↪linewidth=3)
aax1.set_xlim([0, 1])
aax1.set_title(f"Intention: {intentions[0]}")
aax2.plot([0.3]*2, [-10, 100], scalex=False, scaley=False, color="black",␣
 ↪linewidth=3)
aax2.set_xlim([0, 1])
aax2.set_title(f"Intention: {intentions[1]}")
aax3.plot([0.2]*2, [-10, 100], scalex=False, scaley=False, color="black",␣
 ↪linewidth=3)
aax3.set_xlim([0, 1])
aax3.set_title(f"Intention: {intentions[2]}")
ax1.plot([0.3]*2, [-10, 100], scalex=False, scaley=False, color="black",␣
 ↪linewidth=3)
ax2.plot([0.6]*2, [-10, 100], scalex=False, scaley=False, color="black")
ax3.plot([PI/3]*2, [-10, 100], scalex=False, scaley=False, color="black")
```

```
fig.savefig("mc_sim_vi.png")
```




[ ]:
```
fig, ((aax1, ax1),(aax2, ax2),(aax3, ax3)) = plt.subplots(3,2, figsize=(20, 20))
for sim in sims:
  res = sim.mcmc()
  res.plot_alpha(aax1, 0, with_gt=False)
  res.plot_alpha(aax2, 1, with_gt=False)
  res.plot_alpha(aax3, 2, with_gt=False)
  res.plot_p_D(ax1, with_gt=False)
  res.plot_std(ax2, with_gt=False)
  res.plot_loc(ax3, with_gt=False)
fig.tight_layout()
```

```
aax1.plot([0.5]*2, [-10, 100], scalex=False, scaley=False, color="black", linewidth=3)
aax1.set_xlim([0, 1])
aax1.set_title(f"Intention: {intentions[0]}")
aax2.plot([0.3]*2, [-10, 100], scalex=False, scaley=False, color="black", linewidth=3)
aax2.set_xlim([0, 1])
aax2.set_title(f"Intention: {intentions[1]}")
aax3.plot([0.2]*2, [-10, 100], scalex=False, scaley=False, color="black", linewidth=3)
aax3.set_xlim([0, 1])
aax3.set_title(f"Intention: {intentions[2]}")
ax1.plot([0.3]*2, [-10, 100], scalex=False, scaley=False, color="black", linewidth=3)
ax2.plot([0.6]*2, [-10, 100], scalex=False, scaley=False, color="black")
ax3.plot([PI/3]*2, [-10, 100], scalex=False, scaley=False, color="black")
fig.savefig("mc_sim_mcmc.png")
```




[24]:

```
%%capture null
from google.colab import drive
drive.mount('/content/drive')
!apt-get install texlive texlive-xetex texlive-latex-extra pandoc
!pip install pypandoc
!cp "./drive/My Drive/Colab Notebooks/Code.ipynb" ./
!jupyter nbconvert --to PDF "Code.ipynb"
```