Supporting information for: Model Specific to Model General Uncertainty for Physical Properties

Ni Zhan and John R. Kitchin*

Department of Chemical Engineering, Carnegie Mellon University, 5000 Forbes, Ave, Pittsburgh, PA 15213

E-mail: jkitchin@andrew.cmu.edu

Contents

1 Data

Here we get the data from ase databases, and make a plot.

master.db

data.db

```
from ase.db import connect
from ase.eos import EquationOfState
import numpy as np
import matplotlib.pyplot as plt
import matplotlib as mpl

#get Pd data
db = connect('master.db')
E = [d.energy for d in db.select(['bulk', 'strain=xyz'])]
V = [d.volume for d in db.select(['bulk', 'strain=xyz'])]

sel = V[E.index(min(E))]

ind = (np.array(V) > sel - 15) & (np.array(V) < sel + 15)
indzoom = (np.array(V) > sel - 2.2) & (np.array(V) < sel + 3.2)

Vall = np.array(V)[ind]
nrgall = np.array(E)[ind]

V = np.array(V)[indzoom]
nrg = np.array(E)[indzoom]
print(f'Number of Pd data points: {V.shape[0]}')
np.save('v-pd.npy',V)
np.save('nrg-pd.npy',nrg)
```

```
25
26  #get Au data
27  db = connect('data.db')
28
29  Vau, Qe = [], []
30  for d in db.select(['bulk=fcc', 'factor']):
31      Vau += [d.volume/d.natoms]
32      Qe += [d.energy/d.natoms]
33
34  Vauall = np.array(Vau)
35  nrgauall = np.array(Qe)
36
37  sortind = np.argsort(Vauall)
38  Vauall = Vauall[sortind]
39  nrgauall = nrgauall[sortind]
40
41  sel = Vauall[np.argmin(nrgauall)]
42
43  ind = (Vauall > sel - 15) & (Vauall < sel + 15)
44  indzoom = (Vauall > sel - 3.5) & (Vauall < sel + 3.5)
45
46  Vau = Vauall[indzoom]
47  nrgau = nrgauall[indzoom]
48
49  Vauall = Vauall[ind]
50  nrgauall = nrgauall[ind]
51  print(f'Number of Au data points: {Vau.shape[0]}')
52  np.save('v-au.npy',Vau)
53  np.save('nrg-au.npy',nrgau)
54
55  #make plot
56  plt.rcParams.update({'font.size': 12})
57  mpl.rcParams['mathtext.default'] = 'regular'
58
59  fig, ax = plt.subplots(ncols=2, nrows=1, sharex=False, sharey='row')
60  ax[0].plot(Vall,nrgall, '.')
61  ax[0].set_xlabel('Volume/Atom ($\AA^3$)')
62  ax[0].set_ylabel('Energy (eV)')
63  ax[0].axvspan(V[0], V[-1], alpha=0.5)
64
65  ax[1].plot(Vauall, nrgauall, '.')
```

```
ax[1].set_xlabel('Volume/Atom ($\AA^3$)')
ax[1].axvspan(Vau[0], Vau[-1], alpha=0.5)
plt.tight_layout()
plt.subplots_adjust(wspace=0)
plt.savefig('data-pd-au.png', dpi=200)
```

```
Number of Pd data points: 24
Number of Au data points: 29
```

2 Delta method

In this section, we first do nonlinear regression using ASE EquationOfState, and find the max likelihood estimate for equilibrium volume, energy, and bulk modulus. We also calculated the root mean squared error (RMSE) and mean absolute error (MAE). Then we used the delta method to obtain standard errors and confidence intervals for the physical properties. There are all models for Pd and Au. Stabilized jellium (SJ), AntonSchmidt, and Polynomial3 have special calculations of the physical properties because they do not have E0, V0, and B0 as model parameters. The remaining models (Murnaghan, Birch, PoirierTarantola, and Vinet) have the uncertainties for E0, V0, and B0 calculated in the same way across models. Many of the functions are in the deltam.py file  .

2.1 Pd

2.1.1 SJ

```
from autograd import hessian
import autograd
import autograd.numpy as np
from ase.eos import EquationOfState
from scipy.stats.distributions import t
import deltam
from deltam import *

```

```
def sj(params, V):
    p = np.poly1d(params)
    E = p(V**(-1/3))
    return E

def get_delta_u_sj(eosname, funcname, V, nrg):
    eos = EquationOfState(V, nrg, eosname)
    v0, e0, B = eos.fit()
    print(f'V0: {v0:.4f}\nE0: {e0:.4f}\nB: {B*160.2:.1f}')
    params = eos.fit0.c
    h1 = hessian(pred_sse)(params, funcname, V, nrg)
    alpha = pred_sse(params, funcname, V, nrg)
    print(f'RMSE: {np.sqrt(alpha/V.shape[0]):.6f}')
    print(f'MAE: {eos_pred_mse(eos, V, nrg)[1]:.6f}\n')
    pcov = get_pcov(h1, alpha)
    tval = t.ppf(0.975, V.shape[0]-params.shape[0])
    seconf = np.sqrt(np.diagonal(pcov))

    #delta method for V0.
    def get_v0(params):
        a = params[0]
        b = params[1]
        c = params[2]
        root = (-b + np.sqrt(b**2 - 3 * a * c)) / (3 * a)
        return root**(-3)
    gprime = autograd.elementwise_grad(get_v0,0)(params)
    sesq = gprime @ pcov @ gprime
    seconfv0 = np.sqrt(sesq)

    #delta method for E0.
    def get_e0(params):
        a = params[0]
        b = params[1]
        c = params[2]
        root = (-b + np.sqrt(b**2 - 3 * a * c)) / (3 * a)
        return a*root**3 + b*root**2 + c*root + params[3]
    gprime = autograd.elementwise_grad(get_e0, 0)(params)
    #gprime = autograd.elementwise_grad(funcname, 0)(params, v0)
    sesq = gprime @ pcov @ gprime
    seconfe0 = np.sqrt(sesq)
    seprede0 = np.sqrt(sesq + alpha/V.shape[0])
```

```
50
51      #delta method for B.
52      def get_b(params):
53          a = params[0]
54          b = params[1]
55          c = params[2]
56          root = (-b + np.sqrt(b**2 - 3 * a * c)) / (3 * a)
57          return root**5*2*(3*a*root + b)/9
58
59      gprime = autograd.elementwise_grad(get_b, 0)(params)
60      sesq = gprime @ pcov @ gprime
61      seconfb = np.sqrt(sesq)
62
63
64      print('Standard Error Confidences:')
65      print('---------------------------')
66      print(f'V0: {seconfv0:.5f} \nE0: {seconfe0:.5f} \nB: {seconfb*160.2:.3f}\n')
67      print('95% Confidence Intervals:')
68      print('-------------------------')
69      print(f'V0: [{v0-tval*seconfv0:.4f}, {v0+tval*seconfv0:.4f}]')
70      print(f'E0: [{e0-tval*seconfe0:.4f}, {e0+tval*seconfe0:.4f}]')
71      print(f'B: [{(B-tval*seconfb)*160.2:.3f}, {(B+tval*seconfb)*160.2:.3f}]\n')
72
73  V = np.load('v-pd.npy')
74  nrg = np.load('nrg-pd.npy')
75  get_delta_u_sj('sj', sj, V, nrg)
```

```
V0: 15.3041

E0: -5.2146

B: 168.8

RMSE: 0.000103

MAE: 0.000085


Standard Error Confidences:

---------------------------

V0: 0.00068
```

```
E0: 0.00008

B: 0.069


95% Confidence Intervals:

-------------------------

V0: [15.3027, 15.3055]

E0: [-5.2148, -5.2145]

B: [168.643, 168.929]
```

2.1.2 AntonSchmidt

```python
from autograd import hessian
import autograd
import autograd.numpy as np
from ase.eos import EquationOfState
from scipy.stats.distributions import t
import deltam
from deltam import *

def antonschmidt(params, V):
    """From Intermetallics 11, 23-32 (2003)

    Einf should be E_infinity, i.e. infinite separation, but
    according to the paper it does not provide a good estimate
    of the cohesive energy. They derive this equation from an
    empirical formula for the volume dependence of pressure,

    E(vol) = E_inf + int(P dV) from V=vol to V=infinity

    but the equation breaks down at large volumes, so E_inf
    is not that meaningful

    n should be about -2 according to the paper.

    """
```

```
    Einf = params[0]
    B = params[1]
    n = params[2]
    V0 = params[3]

    E = B * V0 / (n + 1) * (V / V0)**(n + 1) * (np.log(V / V0) -
                                                (1 / (n + 1))) + Einf
    return E

def get_delta_u_as(eosname, funcname, V, nrg):
    eos = EquationOfState(V, nrg, eosname)
    v0, e0, B = eos.fit()
    print(f'V0: {v0:.4f}\nE0: {e0:.4f}\nB: {B*160.2:.1f}')
    params = eos.eos_parameters
    h1 = hessian(pred_sse)(params, funcname, V, nrg)
    alpha = pred_sse(params, funcname, V, nrg)
    print(f'RMSE: {np.sqrt(alpha/V.shape[0]):.6f}')
    print(f'MAE: {eos_pred_mse(eos, V, nrg)[1]:.6f}\n')
    pcov = get_pcov(h1, alpha)
    tval = t.ppf(0.975, V.shape[0]-params.shape[0])
    seconf = np.sqrt(np.diagonal(pcov))

    e0 = funcname(params, params[3])
    #delta method for E0. bc E0 is not a parameter in A-S model.
    gprime = autograd.elementwise_grad(funcname, 0)(params, params[3])
    sesq = gprime @ pcov @ gprime
    seconfe0 = np.sqrt(sesq)
    seprede0 = np.sqrt(sesq + alpha/V.shape[0])

    print('Standard Error Confidences:')
    print('---------------------------')
    print(f'V0: {seconf[3]:.5f} \nE0: {seconfe0:.5f} \nB: {seconf[1]*160.2:.3f}\n')
    print('95% Confidence Intervals:')
    print('-------------------------')
    print(f'V0: [{v0-tval*seconf[3]:.4f}, {v0+tval*seconf[3]:.4f}]')
    print(f'E0: [{e0-tval*seconfe0:.4f}, {e0+tval*seconfe0:.4f}]')
    print(f'B: [{(B-tval*seconf[1])*160.2:.3f}, {(B+tval*seconf[1])*160.2:.3f}]\n')


V = np.load('v-pd.npy')
nrg = np.load('nrg-pd.npy')
```

```
66 get_delta_u_as('antonschmidt', antonschmidt, V, nrg)
```

```
V0: 15.3029

E0: -0.0863

B: 167.6

RMSE: 0.000074

MAE: 0.000066


Standard Error Confidences:

---------------------------

V0: 0.00108

E0: 0.00008

B: 0.130


95% Confidence Intervals:

-------------------------

V0: [15.3006, 15.3051]

E0: [-5.2146, -5.2143]

B: [167.286, 167.827]
```

2.1.3 Polynomial3

```
1 from autograd import hessian
2 import autograd
3 import autograd.numpy as np
4 from ase.eos import EquationOfState
5 from scipy.stats.distributions import t
6 import deltam
7 from deltam import *
8
```

```
9
10  def p3(params, V):
11      'polynomial fit'
12      c0 = params[0]
13      c1 = params[1]
14      c2 = params[2]
15      c3 = params[3]
16
17      E = c0 + c1 * V + c2 * V**2 + c3 * V**3
18      return E
19
20  def get_delta_u_p3(eosname, funcname, V, nrg):
21      eos = EquationOfState(V, nrg, eosname)
22      v0, e0, B = eos.fit()
23      print(f'V0: {v0:.4f}\nE0: {e0:.4f}\nB: {B*160.2:.1f}')
24      params = eos.eos_parameters
25      h1 = hessian(pred_sse)(params, funcname, V, nrg)
26      alpha = pred_sse(params, funcname, V, nrg)
27      print(f'RMSE: {np.sqrt(alpha/V.shape[0]):.6f}')
28      print(f'MAE: {eos_pred_mse(eos, V, nrg)[1]:.6f}\n')
29      pcov = get_pcov(h1, alpha)
30      tval = t.ppf(0.975, V.shape[0]-params.shape[0])
31      seconf = np.sqrt(np.diagonal(pcov))
32
33      #delta method for V0.
34      def get_v0(params):
35          a = 3 * params[3]
36          b = 2 * params[2]
37          c = params[1]
38          return (-b + np.sqrt(b**2 - 4 * a * c)) / (2 * a)
39      gprime = autograd.elementwise_grad(get_v0,0)(params)
40      sesq = gprime @ pcov @ gprime
41      seconfv0 = np.sqrt(sesq)
42
43      #delta method for E0.
44      def get_e0(params):
45          a = 3 * params[3]
46          b = 2 * params[2]
47          c = params[1]
48          V = (-b + np.sqrt(b**2 - 4 * a * c)) / (2 * a)
49          return params[0] + params[1] * V + params[2] * V**2 + params[3] * V**3
```

```
    gprime = autograd.elementwise_grad(get_e0, 0)(params)
    sesq = gprime @ pcov @ gprime
    seconfe0 = np.sqrt(sesq)
    seprede0 = np.sqrt(sesq + alpha/V.shape[0])

    #delta method for B.
    def get_b(params):
        a = 3 * params[3]
        b = 2 * params[2]
        c = params[1]
        V = (-b + np.sqrt(b**2 - 4 * a * c)) / (2 * a)
        return (2*params[2] + 6*params[3]*V)*V
    gprime = autograd.elementwise_grad(get_b, 0)(params)
    sesq = gprime @ pcov @ gprime
    seconfb = np.sqrt(sesq)


    print('Standard Error Confidences:')
    print('---------------------------')
    print(f'V0: {seconfv0:.5f} \nE0: {seconfe0:.5f} \nB: {seconfb*160.2:.3f}\n')
    print('95% Confidence Intervals:')
    print('-------------------------')
    print(f'V0: [{v0-tval*seconfv0:.4f}, {v0+tval*seconfv0:.4f}]')
    print(f'E0: [{e0-tval*seconfe0:.4f}, {e0+tval*seconfe0:.4f}]')
    print(f'B: [{(B-tval*seconfb)*160.2:.3f}, {(B+tval*seconfb)*160.2:.3f}]\n')


V = np.load('v-pd.npy')
nrg = np.load('nrg-pd.npy')
get_delta_u_p3('p3', p3, V, nrg)
```

V0: 15.3311

E0: -5.2164

B: 179.6

RMSE: 0.001949

MAE: 0.001686

Standard Error Confidences:

```
--------------------------

V0: 0.02503

E0: 0.00213

B: 4.294


95% Confidence Intervals:

-------------------------

V0: [15.2789, 15.3833]

E0: [-5.2208, -5.2119]

B: [170.593, 188.507]
```

2.1.4 Murnaghan

```
from autograd import hessian
import autograd.numpy as np
from ase.eos import EquationOfState
from scipy.stats.distributions import t
import deltam
from deltam import *

def murnaghan(params, V):
    'From PRB 28,5480 (1983'

    E0 = params[0]
    B0 = params[1]
    BP = params[2]
    V0 = params[3]

    E = E0 + B0 * V / BP * (((V0 / V)**BP) / (BP - 1) + 1) - V0 * B0 / (BP - 1)
    return E

V = np.load('v-pd.npy')
nrg = np.load('nrg-pd.npy')
get_delta_u('murnaghan', murnaghan, V, nrg)
```

```
V0: 15.3017

E0: -5.2140

B: 164.7

RMSE: 0.000456

MAE: 0.000399


Standard Error Confidences:

--------------------------

V0: 0.00758

E0: 0.00049

B: 0.752


95% Confidence Intervals:

-------------------------

V0: [15.2859, 15.3175]

E0: [-5.2150, -5.2130]

B: [163.131, 166.267]
```

2.1.5 Birch

```python
from autograd import hessian
import autograd.numpy as np
from ase.eos import EquationOfState
from scipy.stats.distributions import t

def pred_sse(params, myfunc):
    pred = myfunc(params, V)
    sse = np.sum((pred-nrg)**2)
    return sse

```

```
def eos_pred_mse(eos):
    if eos.eos_string == 'sj':
        y = eos.fit0(V**-(1 / 3))
    else:
        y = eos.func(V, *eos.eos_parameters)
    mae = np.mean(np.absolute(y - nrg))
    mse = np.mean((y-nrg)**2)
    return np.sqrt(mse), mae

#get inverse fisher information
def get_pcov(h, alpha):
    eigs0 = np.linalg.eigvalsh(h)[0]
    if (eigs0 <0):
        eps = max(1e-5, eigs0*-1.05)
    else:
        eps = 1e-5
    j = np.linalg.pinv(h + eps*np.identity(h.shape[0]))
    pcov1 = j*alpha
    u, v = np.linalg.eigh(pcov1)
    return v @ np.diag(np.maximum(u,0)) @ v.T

def birch(params, V):
    """
    From Intermetallic compounds: Principles and Practice, Vol. I: Principles
    Chapter 9 pages 195-210 by M. Mehl. B. Klein, D. Papaconstantopoulos
    paper downloaded from Web

    case where n=0
    """
    E0 = params[0]
    B0 = params[1]
    BP = params[2]
    V0 = params[3]

    E = (E0 +
         9 / 8 * B0 * V0 * ((V0 / V)**(2 / 3) - 1)**2 +
         9 / 16 * B0 * V0 * (BP - 4) * ((V0 / V)**(2 / 3) - 1)**3)
    return E

def get_delta_u(eosname, funcname):
    eos = EquationOfState(V, nrg, eosname)
```

```
    v0, e0, B = eos.fit()
    print(f'V0: {v0:.4f}\nE0: {e0:.4f}\nB: {B*160.2:.1f}')
    params = eos.eos_parameters
    h1 = hessian(pred_sse)(params, funcname)
    alpha = pred_sse(params, funcname)
    print(f'RMSE: {np.sqrt(alpha/V.shape[0]):.6f}')
    print(f'MAE: {eos_pred_mse(eos)[1]:.6f}\n')
    pcov = get_pcov(h1, alpha)
    tval = t.ppf(0.975, V.shape[0]-params.shape[0])
    seconf = np.sqrt(np.diagonal(pcov))
    print('Standard Error Confidences:')
    print('---------------------------')
    print(f'V0: {seconf[3]:.5f} \nE0: {seconf[0]:.5f} \nB: {seconf[1]*160.2:.3f}\n')
    print('95% Confidence Intervals:')
    print('-------------------------')
    print(f'V0: [{v0-tval*seconf[3]:.4f}, {v0+tval*seconf[3]:.4f}]')
    print(f'E0: [{e0-tval*seconf[0]:.4f}, {e0+tval*seconf[0]:.4f}]')
    print(f'B: [{(B-tval*seconf[1])*160.2:.3f}, {(B+tval*seconf[1])*160.2:.3f}]\n')

V = np.load('v-pd.npy')
nrg = np.load('nrg-pd.npy')
get_delta_u('birch', birch)
```

```
V0: 15.3029
E0: -5.2145
B: 167.6
RMSE: 0.000068
MAE: 0.000061

Standard Error Confidences:
---------------------------
V0: 0.00110
E0: 0.00007
B: 0.127
```

95% Confidence Intervals:

V0: [15.3006, 15.3052]

E0: [-5.2146, -5.2143]

B: [167.338, 167.866]

2.1.6 PoirierTarantola

```
from autograd import hessian
import autograd.numpy as np
from ase.eos import EquationOfState
from scipy.stats.distributions import t
import deltam
from deltam import *

def poiriertarantola(params, V):
    'Poirier-Tarantola equation from PRB 70, 224107'

    E0 = params[0]
    B0 = params[1]
    BP = params[2]
    V0 = params[3]

    eta = (V / V0)**(1 / 3)
    squiggle = -3 * np.log(eta)

    E = E0 + B0 * V0 * squiggle**2 / 6 * (3 + squiggle * (BP - 2))
    return E

V = np.load('v-pd.npy')
nrg = np.load('nrg-pd.npy')
get_delta_u('pouriertarantola', poiriertarantola, V, nrg)
```

V0: 15.3072

E0: -5.2149

```
B: 170.7

RMSE: 0.000389

MAE: 0.000334


Standard Error Confidences:

---------------------------

V0: 0.00607

E0: 0.00042

B: 0.824


95% Confidence Intervals:

-------------------------

V0: [15.2945, 15.3198]

E0: [-5.2158, -5.2141]

B: [168.979, 172.418]
```

2.1.7 Vinet

```
from autograd import hessian
import autograd.numpy as np
from ase.eos import EquationOfState
from scipy.stats.distributions import t
import deltam
from deltam import *

def vinet(params, V):
    'Vinet equation from PRB 70, 224107'

    E0 = params[0]
    B0 = params[1]
    BP = params[2]
```

```
    V0 = params[3]

    eta = (V / V0)**(1 / 3)

    E = (E0 + 2 * B0 * V0 / (BP - 1)**2 *
         (2 - (5 + 3 * BP * (eta - 1) - 3 * eta) *
          np.exp(-3 * (BP - 1) * (eta - 1) / 2)))
    return E

V = np.load('v-pd.npy')
nrg = np.load('nrg-pd.npy')
get_delta_u('vinet', vinet, V, nrg)
```

```
V0: 15.3034

E0: -5.2146

B: 168.4

RMSE: 0.000041

MAE: 0.000033


Standard Error Confidences:

---------------------------

V0: 0.00067

E0: 0.00004

B: 0.075


95% Confidence Intervals:

-------------------------

V0: [15.3020, 15.3048]

E0: [-5.2147, -5.2145]

B: [168.214, 168.526]
```

2.2 Au

2.2.1 SJ

```
from autograd import hessian
import autograd
import autograd.numpy as np
from ase.eos import EquationOfState
from scipy.stats.distributions import t
import deltam
from deltam import *

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')
get_delta_u_sj('sj', sj, V, nrg)
```

```
V0: 17.9596

E0: -3.2217

B: 141.2

RMSE: 0.000209

MAE: 0.000174


Standard Error Confidences:

---------------------------

V0: 0.00152

E0: 0.00017

B: 0.130


95% Confidence Intervals:

-------------------------

V0: [17.9565, 17.9628]

E0: [-3.2220, -3.2214]
```

```
B: [140.969, 141.506]
```

2.2.2 AntonSchmidt

```
from autograd import hessian
import autograd
import autograd.numpy as np
from ase.eos import EquationOfState
from scipy.stats.distributions import t
import deltam
from deltam import *

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')
get_delta_u_as('antonschmidt', antonschmidt, V, nrg)
```

```
V0: 17.9680

E0: 1.0664

B: 139.5

RMSE: 0.000323

MAE: 0.000259


Standard Error Confidences:

--------------------------

V0: 0.00481

E0: 0.00034

B: 0.440


95% Confidence Intervals:

------------------------

V0: [17.9580, 17.9779]
```

E0: [-3.2221, -3.2207]

B: [138.549, 140.360]

2.2.3 Polynomial3

```
from autograd import hessian
import autograd
import autograd.numpy as np
from ase.eos import EquationOfState
from scipy.stats.distributions import t
import deltam
from deltam import *


V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')
get_delta_u_p3('p3', p3, V, nrg)
```

V0: 17.9269

E0: -3.2251

B: 160.0

RMSE: 0.003877

MAE: 0.003361

Standard Error Confidences:

V0: 0.05889

E0: 0.00400

B: 4.923

95% Confidence Intervals:

```
-------------------------
V0: [17.8056, 18.0482]
E0: [-3.2333, -3.2169]
B: [149.828, 170.106]
```

2.2.4 Murnaghan

```
from autograd import hessian
import autograd.numpy as np
from ase.eos import EquationOfState
from scipy.stats.distributions import t
import deltam
from deltam import *

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')
get_delta_u('murnaghan', murnaghan, V, nrg)
```

```
V0: 17.9841
E0: -3.2211
B: 136.9
RMSE: 0.000952
MAE: 0.000822

Standard Error Confidences:
---------------------------
V0: 0.01379
E0: 0.00103
B: 1.399

95% Confidence Intervals:
```

```
-------------------------
V0: [17.9557, 18.0125]
E0: [-3.2232, -3.2189]
B: [134.008, 139.769]
```

2.2.5 Birch

```
from autograd import hessian
import autograd.numpy as np
from ase.eos import EquationOfState
from scipy.stats.distributions import t
import deltam
from deltam import *



V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')
get_delta_u('birch', birch, V, nrg)
```

```
V0: 17.9667
E0: -3.2214
B: 139.6
RMSE: 0.000282
MAE: 0.000218

Standard Error Confidences:
---------------------------
V0: 0.00445
E0: 0.00030
B: 0.369
```

95% Confidence Intervals:

V0: [17.9575, 17.9759]

E0: [-3.2221, -3.2208]

B: [138.887, 140.408]

2.2.6 PoirierTarantola

```
from autograd import hessian
import autograd.numpy as np
from ase.eos import EquationOfState
from scipy.stats.distributions import t
import deltam
from deltam import *


V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')
get_delta_u('pouriertarantola', poiriertarantola, V, nrg)
```

V0: 17.9503

E0: -3.2222

B: 144.0

RMSE: 0.000735

MAE: 0.000632

Standard Error Confidences:

V0: 0.01212

E0: 0.00077

```
B: 0.878


95% Confidence Intervals:
-------------------------
V0: [17.9253, 17.9752]
E0: [-3.2238, -3.2206]
B: [142.228, 145.843]
```

2.2.7 Vinet

```
from autograd import hessian
import autograd.numpy as np
from ase.eos import EquationOfState
from scipy.stats.distributions import t
import deltam
from deltam import *

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')
get_delta_u('vinet', vinet, V, nrg)
```

```
V0: 17.9632
E0: -3.2215
B: 140.3
RMSE: 0.000178
MAE: 0.000096


Standard Error Confidences:
---------------------------
V0: 0.00274
E0: 0.00019
```

```
B: 0.244


95% Confidence Intervals:
-------------------------
V0: [17.9576, 17.9689]
E0: [-3.2219, -3.2212]
B: [139.800, 140.806]
```

3 Bayesian regression

In this section, we perform Bayesian nonlinear regression using stochastic variational inference (SVI) and Hamiltonian Monte Carlo (HMC), for the same set of models we used for the delta method. The SVI optimization for each model requires around 5 minutes on 2017 Lenovo laptop. The HMC for each model requires around 5-20 minutes on 2017 Lenovo laptop.

3.1 Pd

3.1.1 SJ

To get SVI samples, run the following shell command. It will run SVI and save pickle file of the samples. –guide flag mf is "mean field". It seemed that mean field worked better than multivariate normal for this problem. –element flag is pd or au. –model flag is sj.

```
python bayesreg.py --element pd --guide mf --model sj --run svi
```

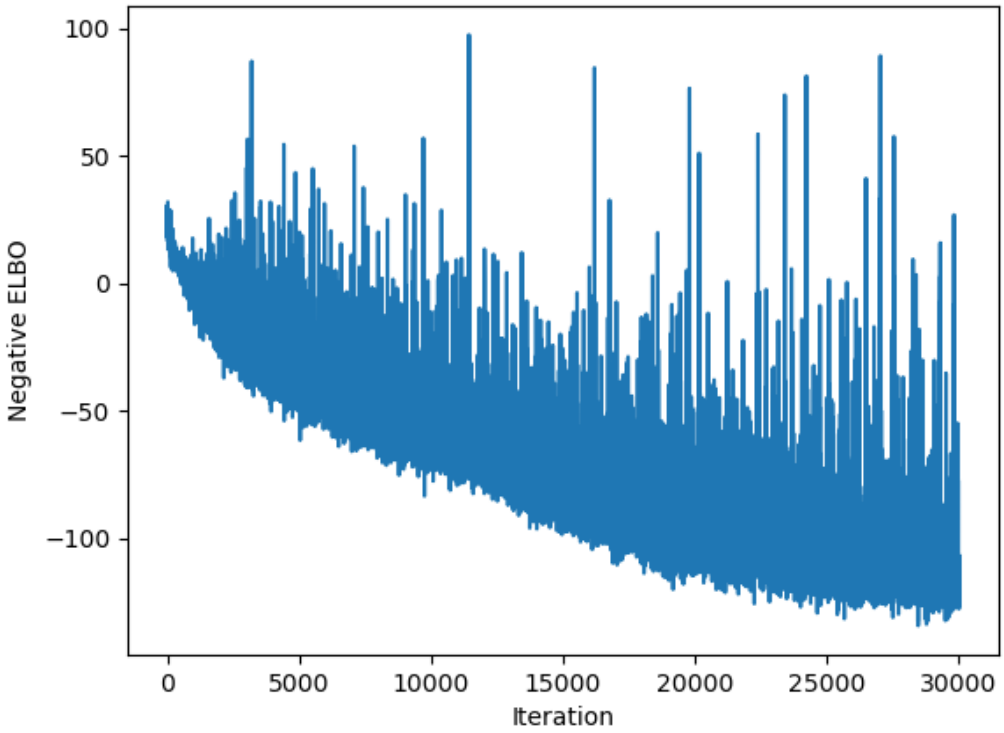

Figure S1: ELBO Pd stabilized jellium

To get HMC samples, run the following shell command. It will run HMC and save pickle file of the HMC samples. I used the following initial guess for SJ, warmup steps of 60, and init_to_mean as init_strategy.

p0 = pyro.sample("p0", dist.Normal(2755, 1.))

p1 = pyro.sample("p1", dist.Normal(-2881.5, 1.))

p2 = pyro.sample("p2", dist.Normal(980.5, 1.))

p3 = pyro.sample("p3", dist.Normal(-113., 1.))

sigma = pyro.sample("sigma", dist.Uniform(0., 1.0))

```
python bayesreg.py --element pd --model sj --run hmc --warmup_steps 60 --num_samples 1100 --init_mean_hmc True
```

To get following plot, run shell command.

```
python bayesreg.py --element pd --guide mf --model sj --run plot --ticks 4-4-4-3
```

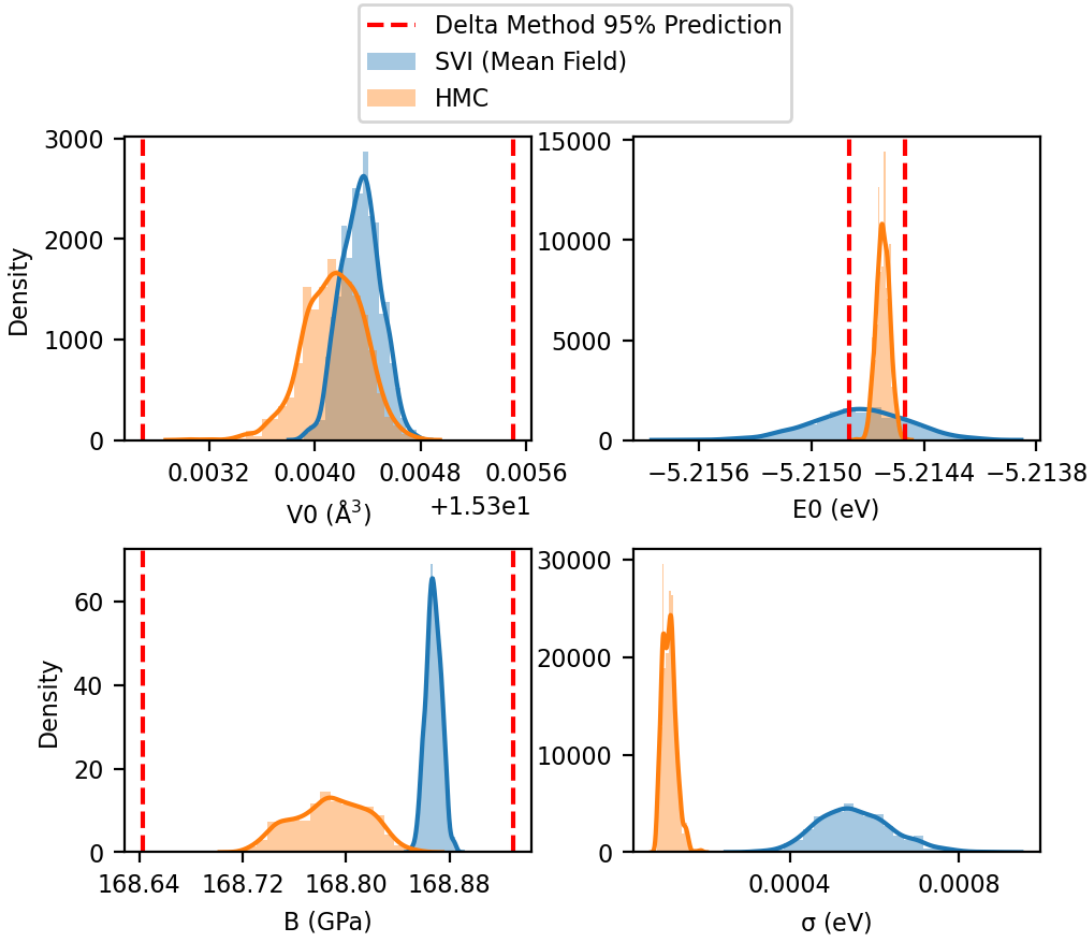

Figure S2: Pd stabilized jellium SVI HMC Delta

3.1.2 AntonSchmidt

Initial guess

einf = pyro.sample("einf", dist.Normal(-0.1, 0.05))

b = pyro.sample("b", dist.Normal(1.0, 0.1))

n = pyro.sample("n", dist.Normal(-2.8, 0.1))

v0 = pyro.sample("v0", dist.Normal(15., 0.2))

sigma = pyro.sample("sigma", dist.Uniform(0., 1.))

```
python bayesreg.py --element pd --guide mf --model as --run svi --num_iters 50000
```

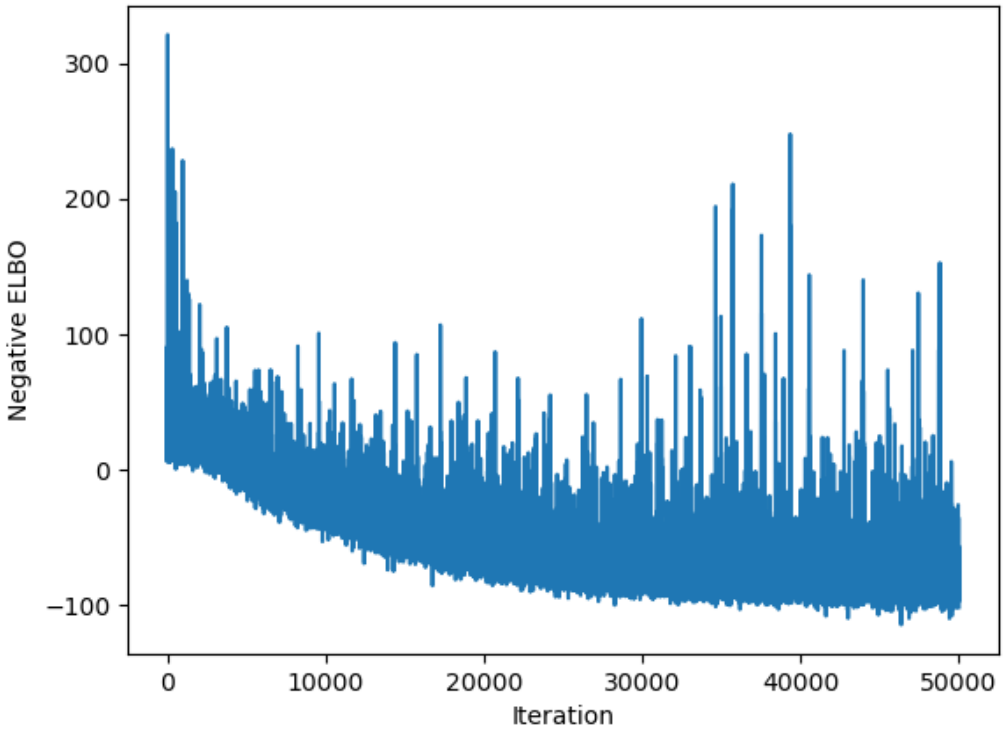

Figure S3: ELBO Pd antonschmidt

Run HMC:

```
python bayesreg.py --element pd --model as --run hmc --warmup_steps 80 --num_samples 1100 --init_mean_hmc True
```

Make plot:

```
python bayesreg.py --element pd --model as --run plot --guide mf
```

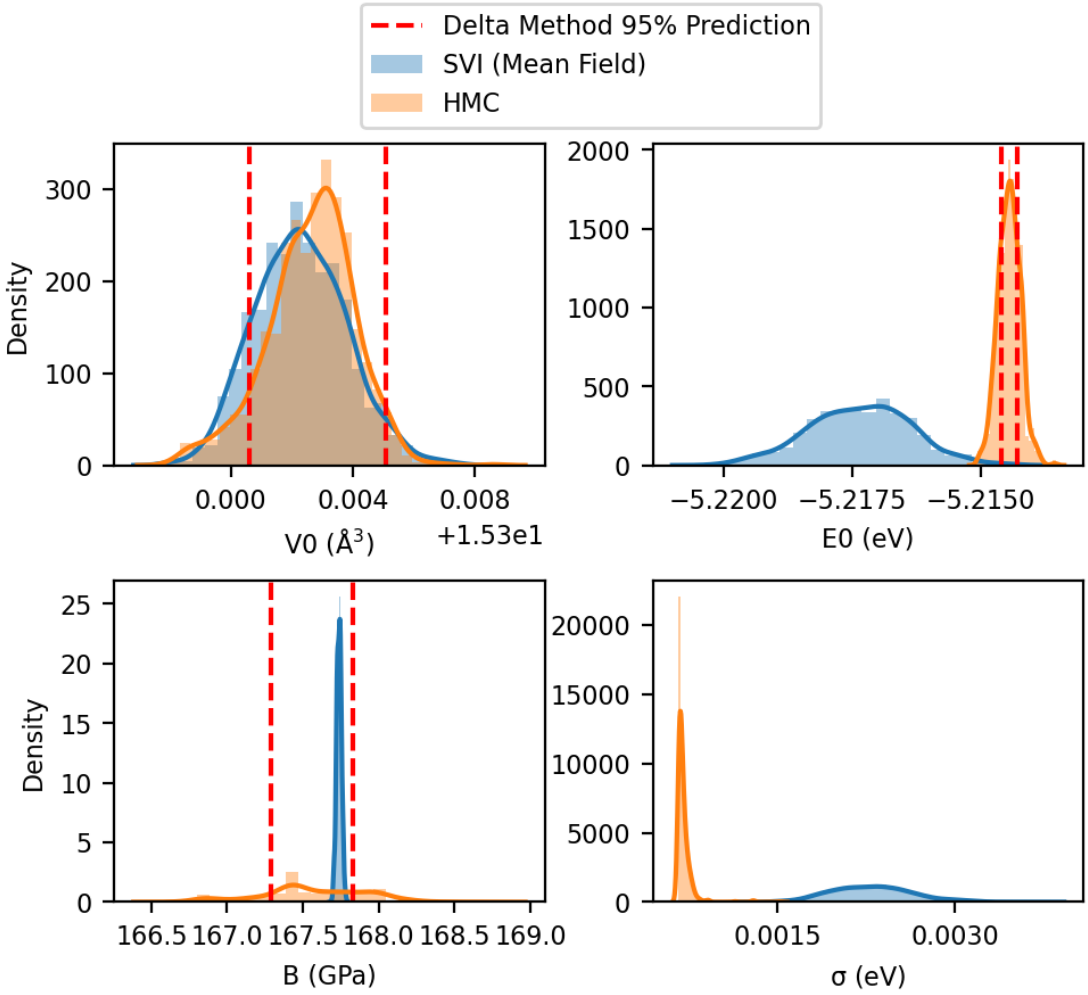

Figure S4: Pd anton-schmidt SVI HMC Delta

3.1.3 Polynomial3

We had to setup the problem specially because the optimal parameters are orders of magnitude different from each other.

Initial guess

c0 = pyro.sample("c0", dist.Normal(0., 1.))

c1 = pyro.sample("c1", dist.Normal(0., 1.))

c2 = pyro.sample("c2", dist.Normal(0., 1.))

c3 = pyro.sample("c3", dist.Normal(0., 1.))

sigma = pyro.sample("sigma", dist.Uniform(0., 1.))

mean = func(V, 19+c0, -4.19+0.1*c1, 0.236 + 1e-2*c2, -0.0043+1e-4*c3)

```
python bayesreg.py --element pd --guide mf --model p3 --run svi
```

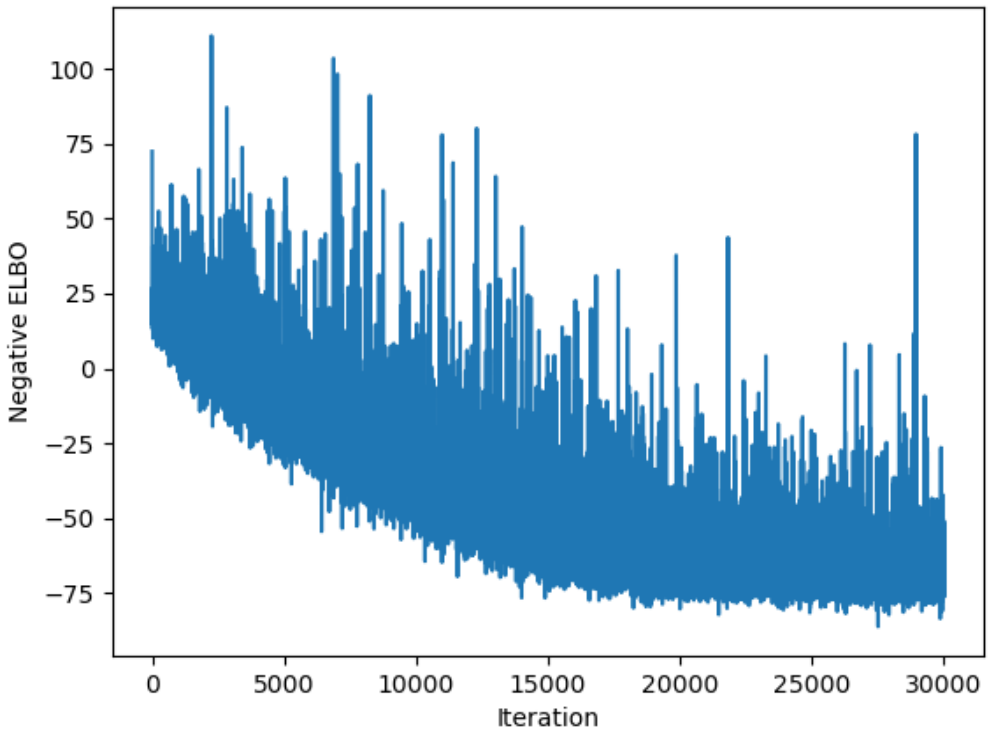

Figure S5: ELBO Pd polynomial3

Run HMC:

```
python bayesreg.py --element pd --model p3 --run hmc --warmup_steps 40 --num_samples 1100 --init_mean_hmc True
```

Make plot:

```
python bayesreg.py --element pd --model p3 --run plot --guide mf
```

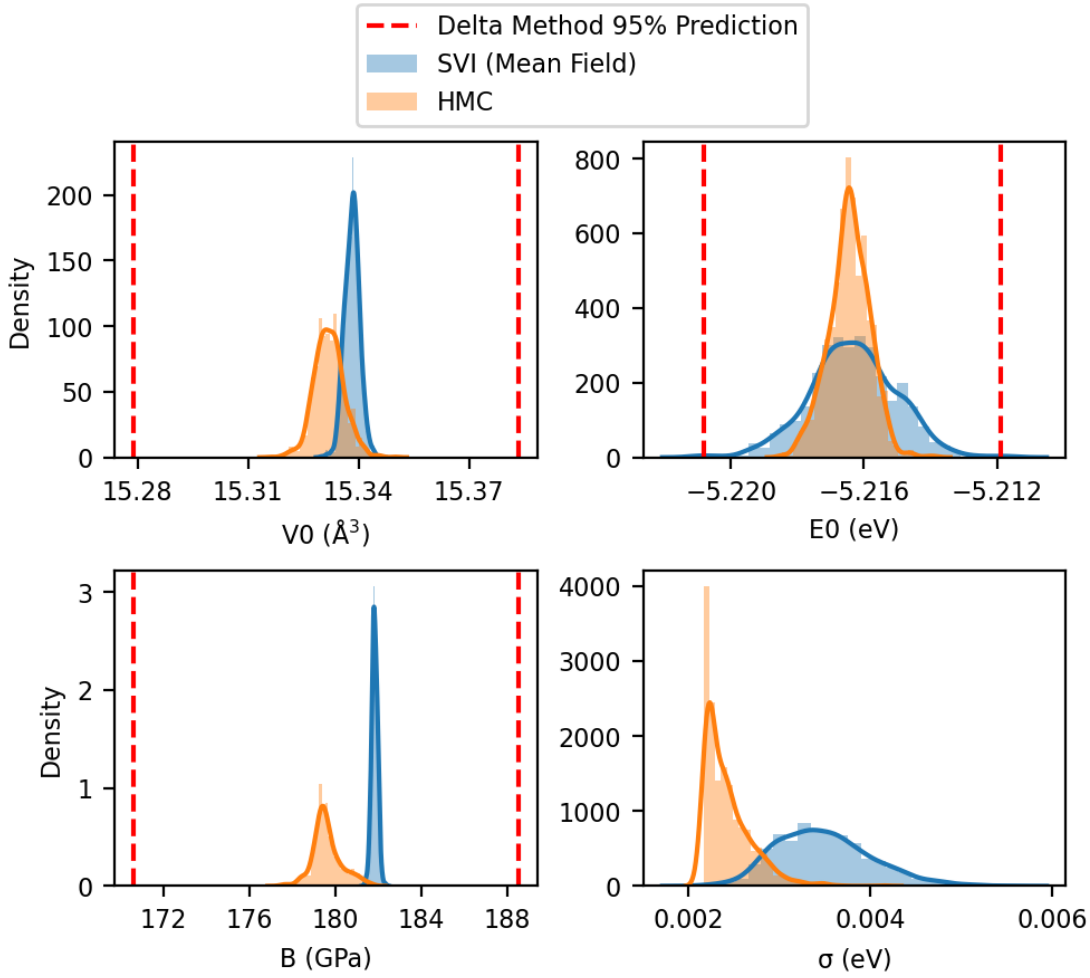

Figure S6: Pd polynomial SVI HMC Delta

3.1.4 Murnaghan

Init Guess

e0 = pyro.sample("e0", dist.Normal(-5., 1.))

b = pyro.sample("b", dist.Normal(1.0, 0.2))

bp = pyro.sample("bp", dist.Normal(5., 1.))

v0 = pyro.sample("v0", dist.Normal(15., 5.))

sigma = pyro.sample("sigma", dist.Uniform(0., 1.))

```
python bayesreg.py --element pd --model murn --run svi
```

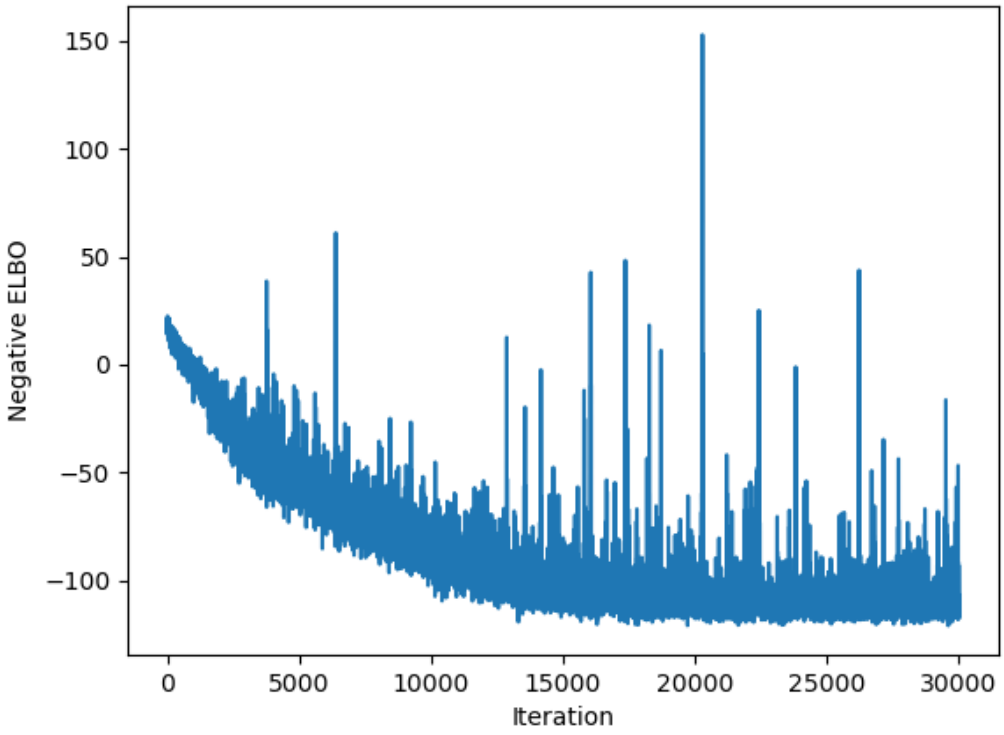

Figure S7: ELBO Pd murnaghan

```
python bayesreg.py --element pd --model murn --run hmc
```

```
python bayesreg.py --element pd --model murn --run plot --ticks 4-4-5-3
```

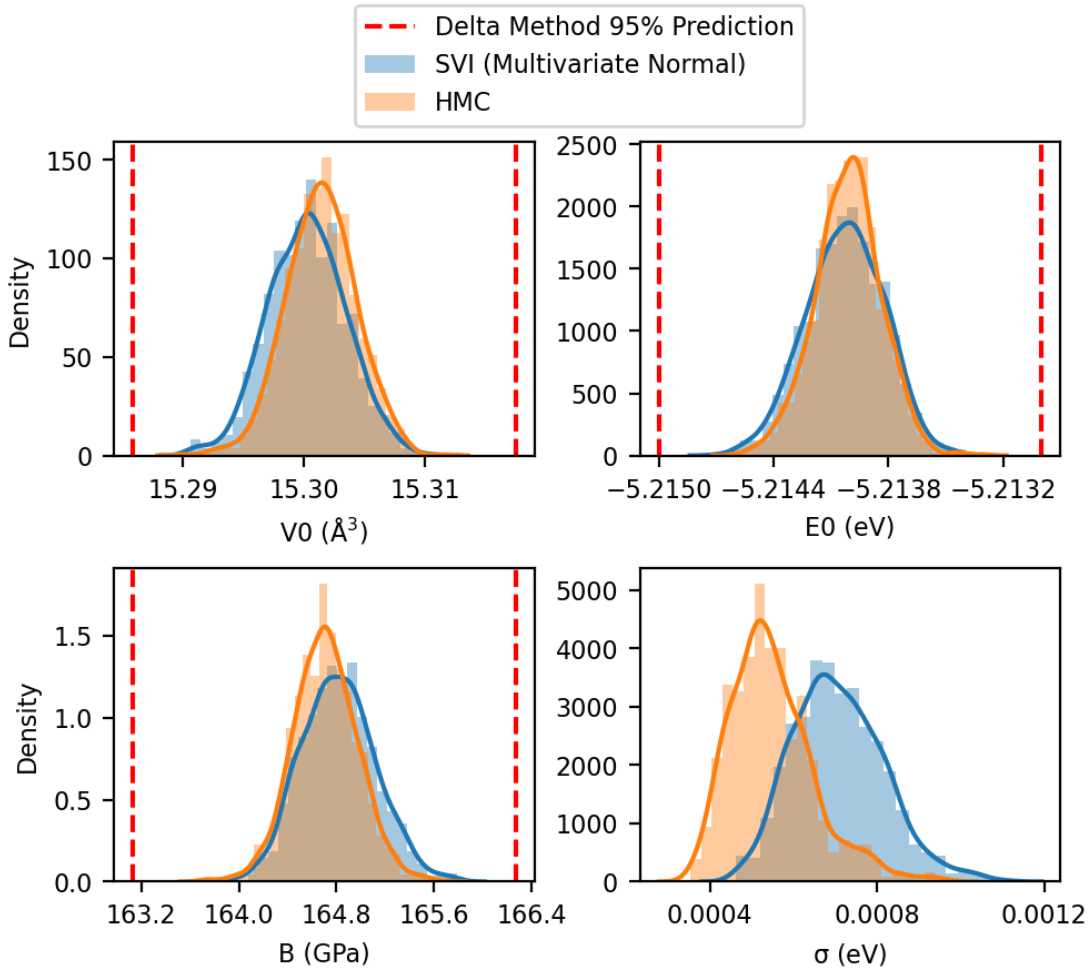

Figure S8: Pd murnaghan SVI HMC Delta

3.1.5 Birch

In the code below, we specify the probabilistic model using `pyro`, specify the variational distribution as the multivariate normal, and train using SVI. We then save 1,000 samples from the optimized variational distribution. The SVI optimization below requires about 5 minutes on 2017 Lenovo laptop.

```
import torch
import pyro
from pyro.infer.autoguide import AutoMultivariateNormal, init_to_mean
from pyro.infer import SVI, Trace_ELBO
import pyro.distributions as dist
```

```
import pyro.optim as optim
import numpy as np
import matplotlib.pyplot as plt
from pyro.infer import Predictive
import pickle

def birch(V, E0, B0, BP, V0):
    """
    From Intermetallic compounds: Principles and Practice, Vol. I: Principles
    Chapter 9 pages 195-210 by M. Mehl. B. Klein, D. Papaconstantopoulos
    case where n=0
    """
    E = (E0 +
         9 / 8 * B0 * V0 * ((V0 / V)**(2 / 3) - 1)**2 +
         9 / 16 * B0 * V0 * (BP - 4) * ((V0 / V)**(2 / 3) - 1)**3)
    return E

def model(V, nrg, func):
    e0 = pyro.sample("e0", dist.Normal(-5., 1.))
    b = pyro.sample("b", dist.Normal(1., 0.2))
    bp = pyro.sample("bp", dist.Normal(5., 1.))
    v0 = pyro.sample("v0", dist.Normal(15., 5.))
    sigma = pyro.sample("sigma", dist.Uniform(0., 1.))
    mean = func(V, e0, b, bp, v0)
    with pyro.plate("data", len(V)):
        pyro.sample("obs", dist.Normal(mean, sigma), obs=nrg)

guide = AutoMultivariateNormal(model, init_loc_fn=init_to_mean)

num_iters = 40000
optimizer = torch.optim.Adam
gamma = 0.5**(1/num_iters)
scheduler = optim.ExponentialLR({'optimizer': optimizer,
                                 'optim_args': {'lr': 0.001},
                                 'gamma': gamma})

svi = SVI(model,
          guide,
          scheduler,
          loss=Trace_ELBO(),
num_samples=16)
```

```
47
48  V = np.load('v-pd.npy')
49  nrg = np.load('nrg-pd.npy')
50  V = torch.Tensor(V)
51  nrg = torch.Tensor(nrg)
52
53  pyro.clear_param_store()
54
55  elbos = []
56  for i in range(num_iters):
57      elbo = svi.step(V, nrg, birch)
58      elbos += [elbo]
59      scheduler.step()
60
61  plt.clf()
62  plt.plot(np.array(elbos))
63  plt.xlabel('Iteration')
64  plt.ylabel('Negative ELBO')
65  plt.savefig('elbo-pd-birch.png')
66  print(f'''#+attr_org: :width 600
67  #+caption: ELBO Pd birch
68  [[./elbo-pd-birch.png]]''')
69
70  num_samples=1000
71  predictive = Predictive(model, guide=guide, num_samples=num_samples)
72
73  svi_mvn_samples = {k: v.reshape(num_samples).detach().cpu().numpy()
74                     for k, v in predictive(V, nrg, birch).items()
75                     if k != "obs"}
76
77  with open('pd-birch-svi-mvn-samples.pickle', 'wb') as handle:
78      pickle.dump(svi_mvn_samples, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

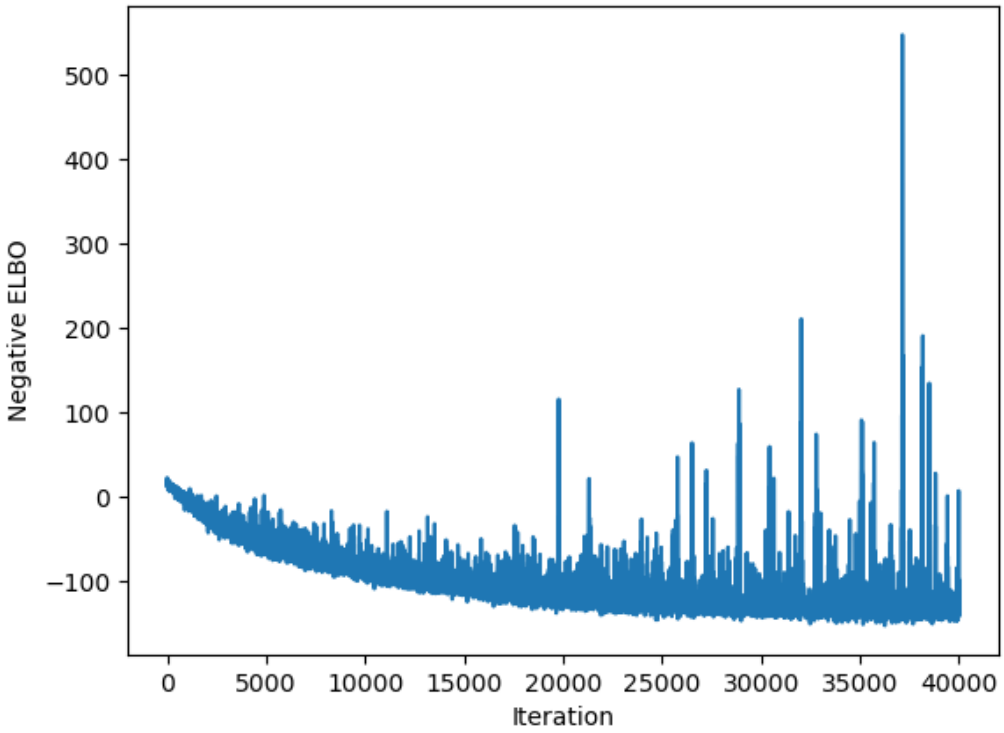

Figure S9: ELBO Pd birch

The code below runs the HMC for the same probabilistic model, and saves 1,000 samples. The HMC below takes around 20 minutes on 2017 Lenovo laptop.

```
from pyro.infer import MCMC, NUTS
import pyro.distributions as dist
import bayesreg
from bayesreg import *
import numpy as np
import torch
import pickle

V = np.load('v-pd.npy')
nrg = np.load('nrg-pd.npy')
V = torch.Tensor(V)
nrg = torch.Tensor(nrg)

nuts_kernel = NUTS(model_others)
```

```
mcmc = MCMC(nuts_kernel, num_samples=1000, warmup_steps=200)
mcmc.run(V, nrg, birch)

hmc_samples = {k: v.detach().cpu().numpy() for k, v in mcmc.get_samples().items()}

with open('pd-birch-hmc-samples-test.pickle', 'wb') as handle:
    pickle.dump(hmc_samples, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

The code below makes the plot to compare SVI, HMC posteriors and the delta method interval.

```
import pickle
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib as mpl
import numpy as np
from matplotlib.ticker import MaxNLocator

with open('pd-birch-svi-mvn-samples.pickle', 'rb') as handle:
    svi_mvn_samples = pickle.load(handle)
with open('pd-birch-hmc-samples.pickle', 'rb') as handle:
    hmc_samples = pickle.load(handle)

sites = ["v0", "e0", "b", "sigma"]
xlabels = ['V0 ($\AA^3$)', 'E0 (eV)', 'B (GPa)', '$\sigma$ (eV)']
nbinsi = [4,4,4,4]
ylabels = ['Density', '', 'Density', '']

deltam = np.array([[15.3006, 15.3052],[-5.2146,-5.2143], [167.338,167.866]])

plt.rcParams.update({'font.size': 8})
mpl.rcParams['mathtext.default'] = 'regular'
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(5.5, 4.5))

for i, ax in enumerate(axs.reshape(-1)):
    site = sites[i]
    if site == 'b':
        sns.distplot(160.2*svi_mvn_samples[site], ax=ax,
                     label="SVI (Multivariate Normal)")
```

```
        sns.distplot(160.2*hmc_samples[site], ax=ax, label="HMC")
    else:
        sns.distplot(svi_mvn_samples[site], ax=ax,
                     label="SVI (Multivariate Normal)")
        sns.distplot(hmc_samples[site], ax=ax, label="HMC")
    if i != 3:
        ax.axvline(x=deltam[i,0], ls='--', c='r',
                   label='Delta Method 95% Prediction')
        ax.axvline(x=deltam[i,1], ls='--', c='r')
    ax.xaxis.set_major_locator(MaxNLocator(nbins=nbinsi[i]))
    ax.set_xlabel(xlabels[i])
    ax.set_ylabel(ylabels[i])
handles, labels = axs[0,0].get_legend_handles_labels()
legend=fig.legend(handles, labels, loc='upper right',bbox_to_anchor=(0.7, 1.0))
plt.subplots_adjust(wspace = 0.28, hspace=0.36, top=0.85)
plt.savefig('pd-birch-svi-hmc-delta.png',bbox_extra_artists=(legend,),
            bbox_inches='tight', dpi=200)

print(f'''#+attr_org: :width 600
#+caption: Pd birch SVI HMC Delta
[[./pd-birch-svi-hmc-delta.png]]''')
```

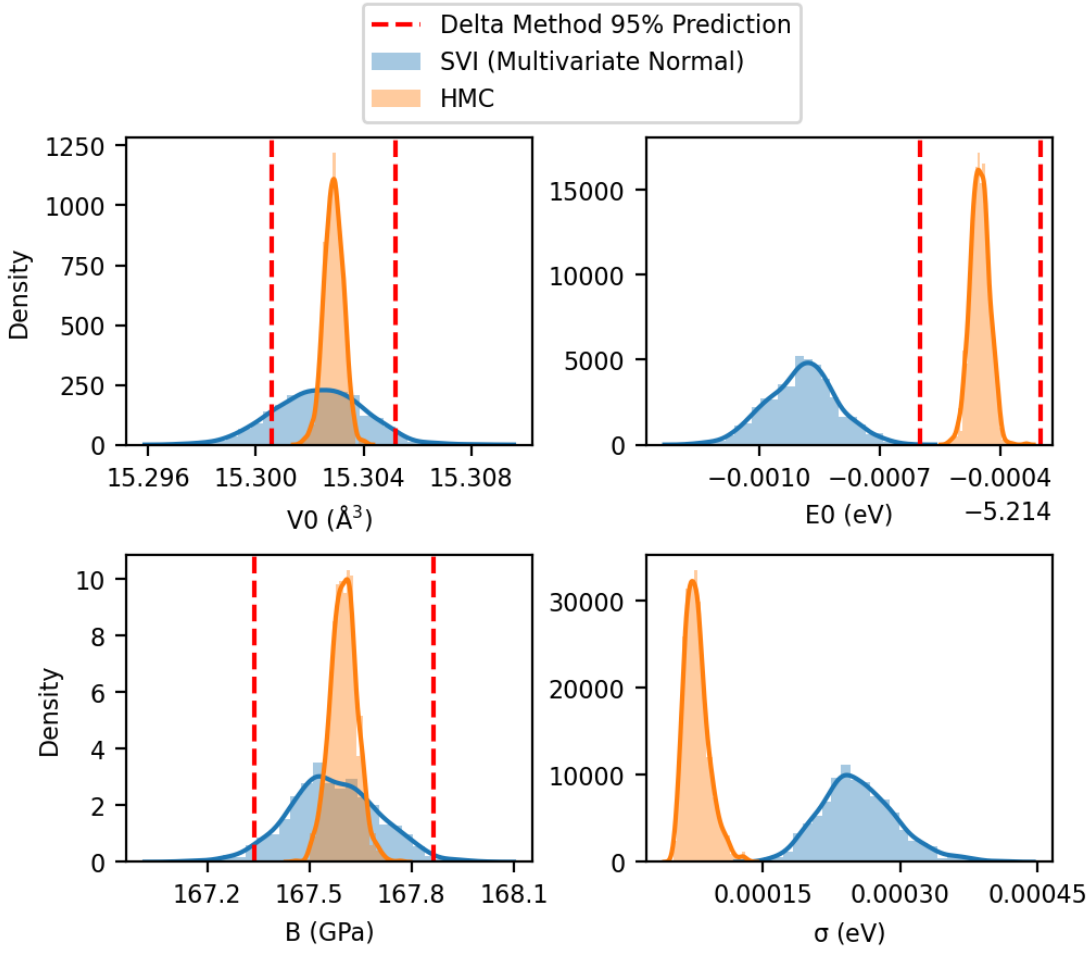

Figure S10: Pd birch SVI HMC Delta

3.1.6 PoirierTarantola

```
import numpy as np
import torch
import bayesreg
from bayesreg import *

V = np.load('v-pd.npy')
nrg = np.load('nrg-pd.npy')
V = torch.Tensor(V)
nrg = torch.Tensor(nrg)

optim_vi(model_others, V, nrg, poiriertarantola, 30000, 'pd-pt')

```

```
print(f'''#+attr_org: :width 600
#+caption: ELBO Pd poiriertarantola
[[./elbo-pd-pt.png]]''')
```

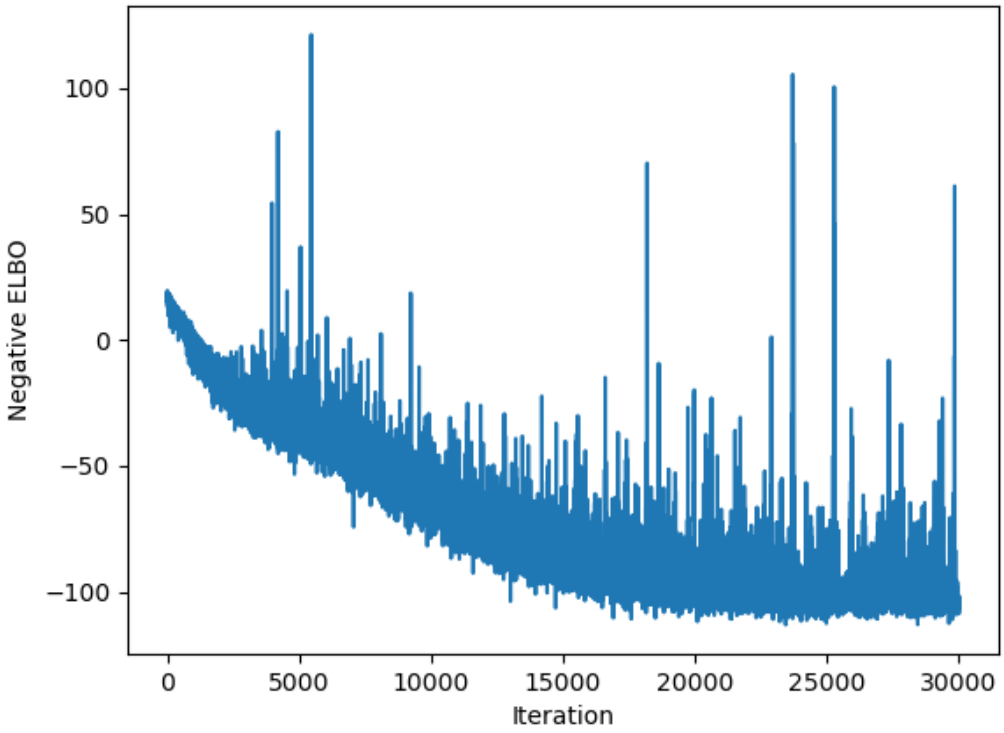

Figure S11: ELBO Pd poiriertarantola

```
from pyro.infer import MCMC, NUTS
import pyro.distributions as dist
import bayesreg
from bayesreg import *
import numpy as np
import torch
import pickle

V = np.load('v-pd.npy')
nrg = np.load('nrg-pd.npy')
V = torch.Tensor(V)
nrg = torch.Tensor(nrg)

```

```
nuts_kernel = NUTS(model_others)

mcmc = MCMC(nuts_kernel, num_samples=1000, warmup_steps=200)
mcmc.run(V, nrg, poiriertarantola)

hmc_samples = {k: v.detach().cpu().numpy() for k, v in mcmc.get_samples().items()}

with open('pd-pt-hmc-samples.pickle', 'wb') as handle:
    pickle.dump(hmc_samples, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

```
import pickle
import numpy as np
import bayesreg
from bayesreg import make_plot

with open('pd-pt-svi-mvn-samples.pickle', 'rb') as handle:
    svi_mvn_samples = pickle.load(handle)
with open('pd-pt-hmc-samples.pickle', 'rb') as handle:
    hmc_samples = pickle.load(handle)

deltam = np.array([[15.2945, 15.3198],[-5.2158,-5.2141], [168.979,172.418]])

make_plot(svi_mvn_samples, hmc_samples, deltam,
          'pd-pt-svi-hmc-delta', nbinsi = [4,4,6,3])

print(f'''#+attr_org: :width 600
#+caption: Pd poirier-tarantola SVI HMC Delta
[[./pd-pt-svi-hmc-delta.png]]''')
```

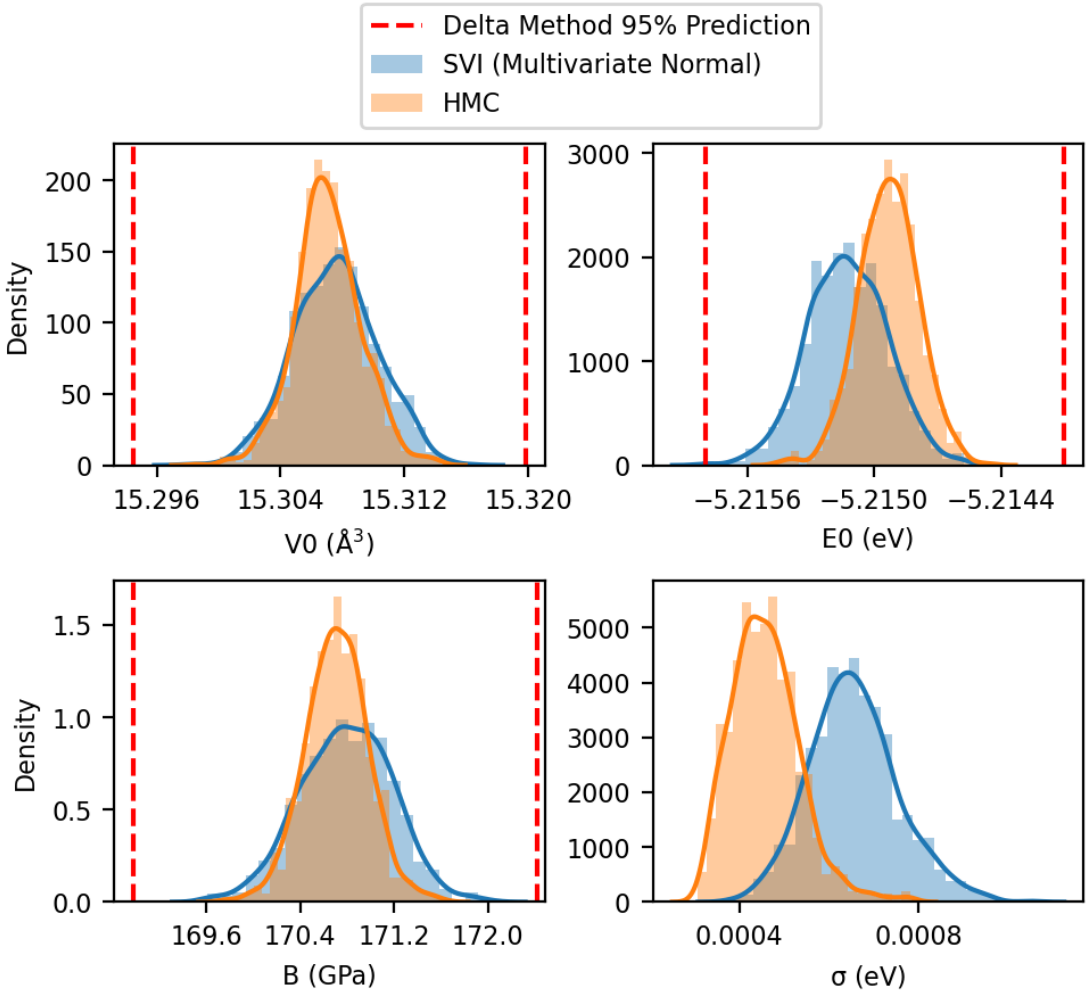

Figure S12: Pd poirier-tarantola SVI HMC Delta

3.1.7 Vinet

Init Guess

e0 = pyro.sample("e0", dist.Normal(-5., 0.5))

b = pyro.sample("b", dist.Normal(1.0, 0.2))

bp = pyro.sample("bp", dist.Normal(5.5, 0.5))

v0 = pyro.sample("v0", dist.Normal(15., 1.))

sigma = pyro.sample("sigma", dist.Uniform(0., 1.))

–gamma is a parameter for the optimizer learning rate schedule. The default gamma is

1.0, here we use 0.5 which means the optimizer learn-rate is 0.5 times its initial learn-rate after 30000 steps. (gradually decreasing learn-rate)

```
python bayesreg.py --element pd --model vinet --run svi --gamma 0.5
```

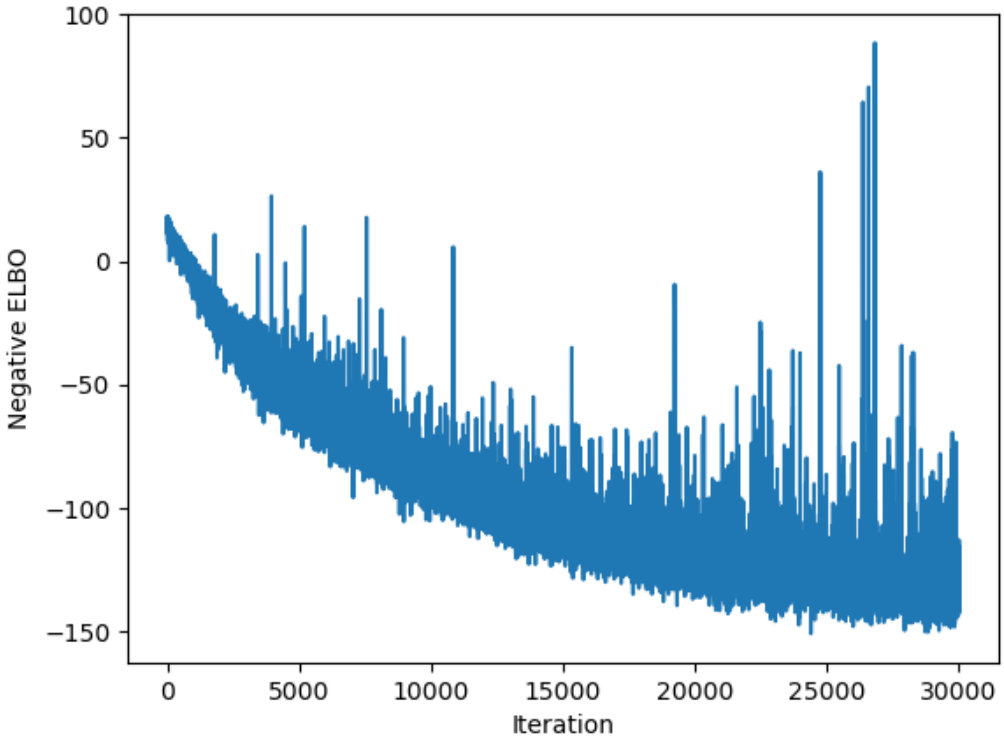

Figure S13: ELBO Pd vinet

```
python bayesreg.py --element pd --model vinet --run hmc
```

```
python bayesreg.py --element pd --model vinet --run plot --ticks 4-3-4-3
```

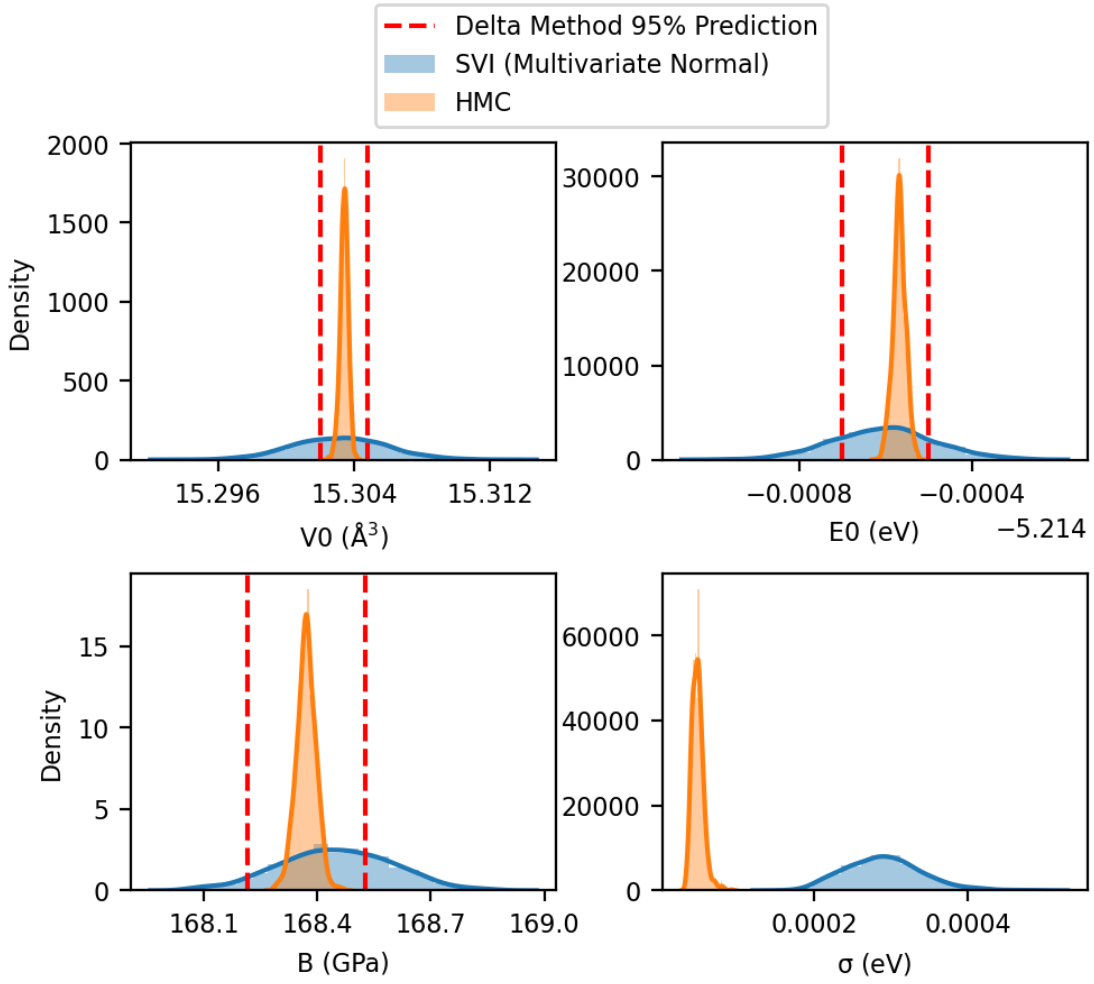

Figure S14: Pd vinet SVI HMC Delta

3.2 Au

3.2.1 SJ

Init Guess:

p0 = pyro.sample("p0", dist.Normal(3784., 1.))

p1 = pyro.sample("p1", dist.Normal(-3846., 1.))

p2 = pyro.sample("p2", dist.Normal(1282., 1.))

p3 = pyro.sample("p3", dist.Normal(-143., 1.))

sigma = pyro.sample("sigma", dist.Uniform(0., 1.0))

```
python bayesreg.py --element au --guide mf --model sj --run svi
```

From the plot, running for more than 30,000 iterations should help reach a better optimal point.

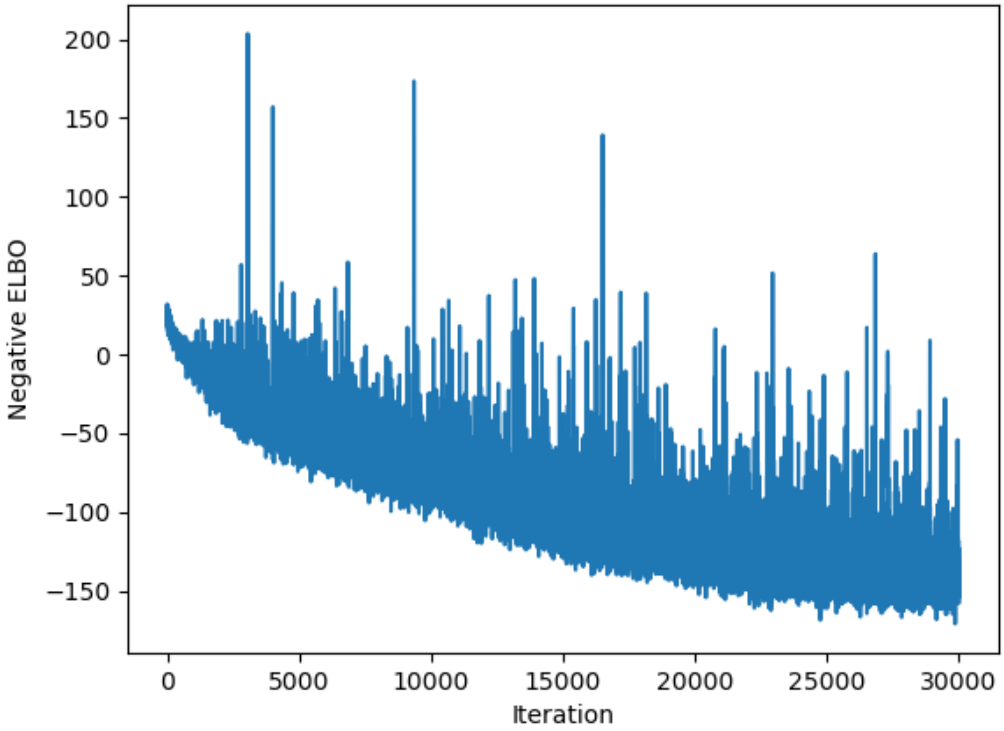

Figure S15: ELBO Au stabilized jellium

```
python bayesreg.py --element au --model sj --run hmc --warmup_steps 60 --num_samples 1100 --init_mean_hmc True
```

```
python bayesreg.py --element au --guide mf --model sj --run plot --ticks 4-4-5-3
```

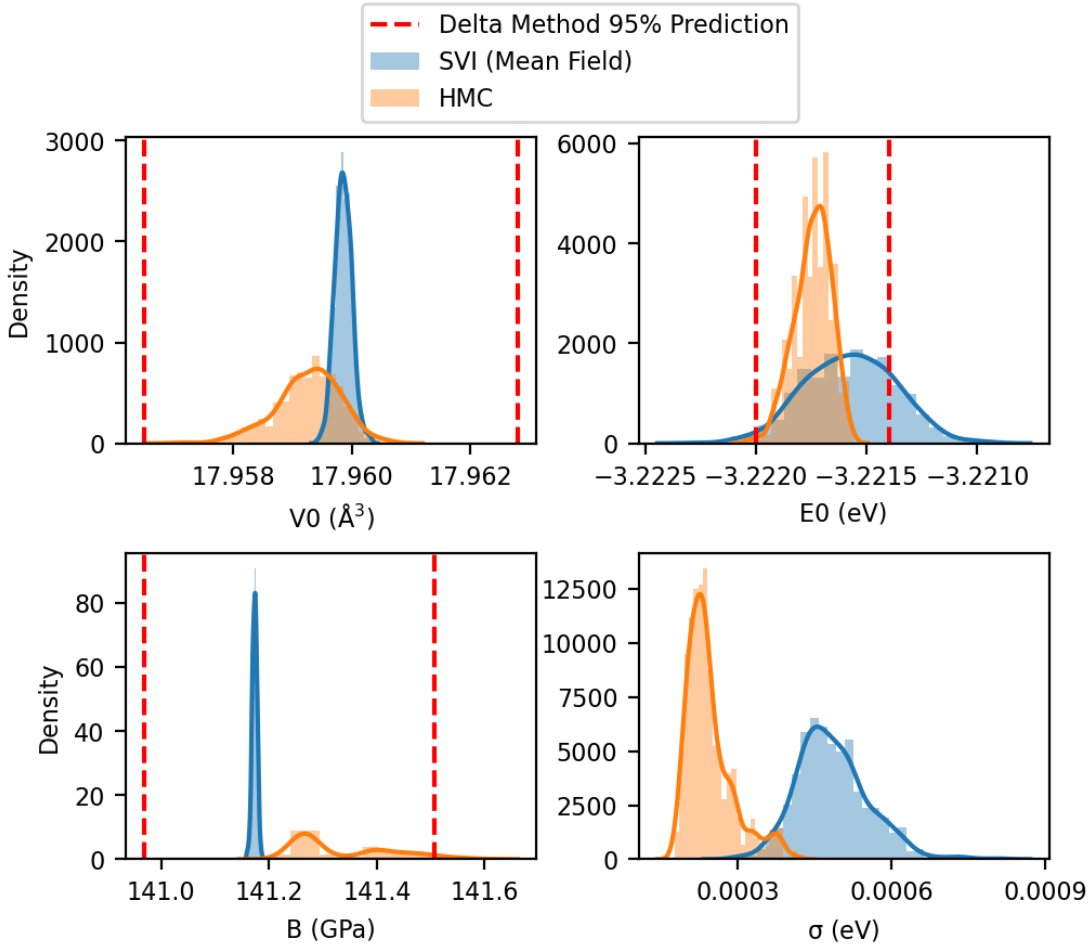

Figure S16: Au stabilized jellium SVI HMC Delta

3.2.2 AntonSchmidt

Init Guess

einf = pyro.sample("einf", dist.Normal(1., 0.1))

b = pyro.sample("b", dist.Normal(0.9, 0.05))

n = pyro.sample("n", dist.Normal(-2.9, 0.1))

v0 = pyro.sample("v0", dist.Normal(18., 0.2))

sigma = pyro.sample("sigma", dist.Uniform(0., 1.))

```
python bayesreg.py --element au --guide mf --model as --run svi --num_iters 50000
```

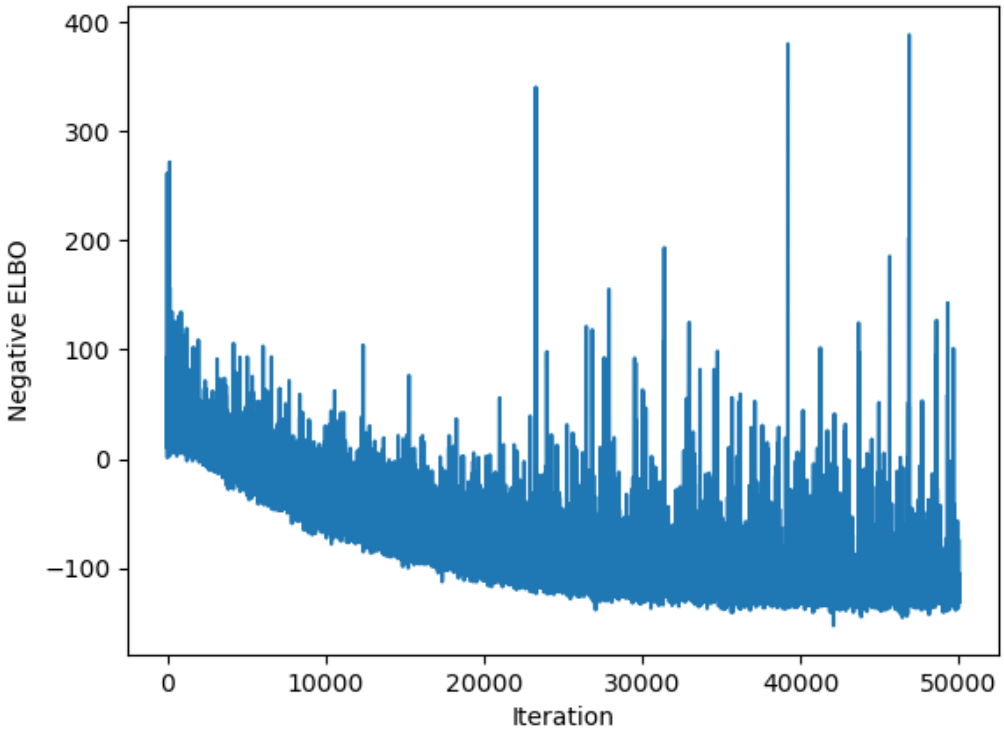

Figure S17: ELBO Au antonschmidt

Run HMC:

```
python bayesreg.py --element au --model as --run hmc --warmup_steps 200 --num_samples 1000 --init_mean_hmc False
```

Make plot:

```
python bayesreg.py --element au --model as --run plot --guide mf
```

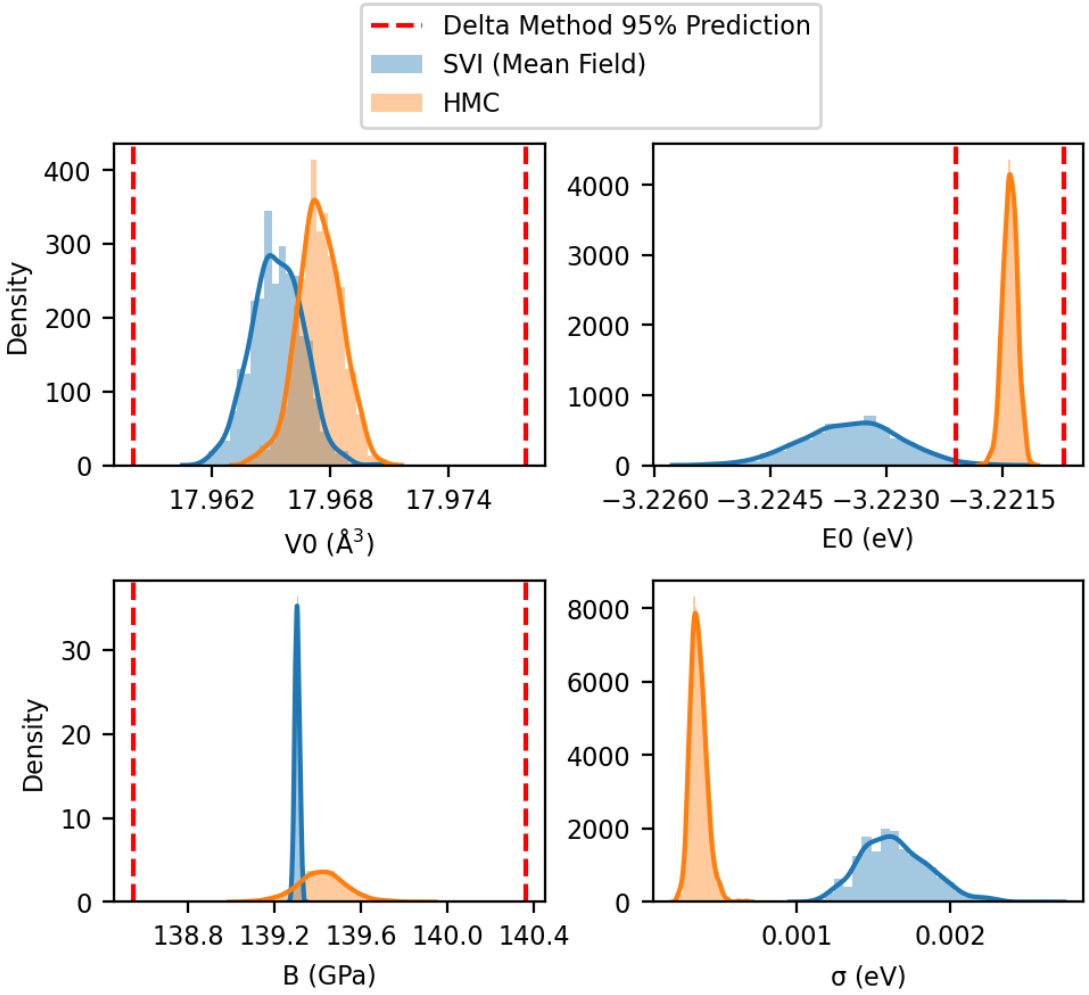

Figure S18: Au anton-schmidt SVI HMC Delta

3.2.3 Polynomial3

Initial guess

c0 = pyro.sample("c0", dist.Normal(0., 1.))

c1 = pyro.sample("c1", dist.Normal(0., 1.))

c2 = pyro.sample("c2", dist.Normal(0., 1.))

c3 = pyro.sample("c3", dist.Normal(0., 1.))

sigma = pyro.sample("sigma", dist.Uniform(0., 1.))

mean = func(V, 27+c0, -4.5+0.1*c1, 0.23 + 1e-2*c2, -0.0036+1e-4*c3)

```
python bayesreg.py --element au --guide mf --model p3 --run svi
```

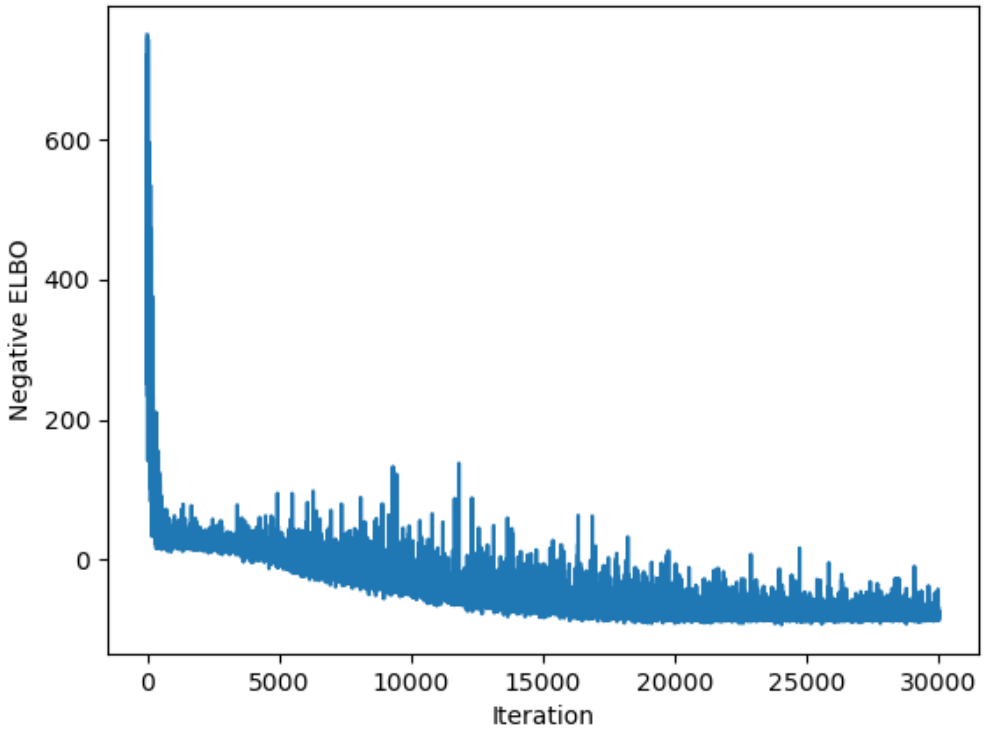

Figure S19: ELBO Au polynomial3

Run HMC:

```
python bayesreg.py --element au --model p3 --run hmc --warmup_steps 60 --num_samples 1100 --init_mean_hmc True
```

Make plot:

```
python bayesreg.py --element au --model p3 --run plot --guide mf
```

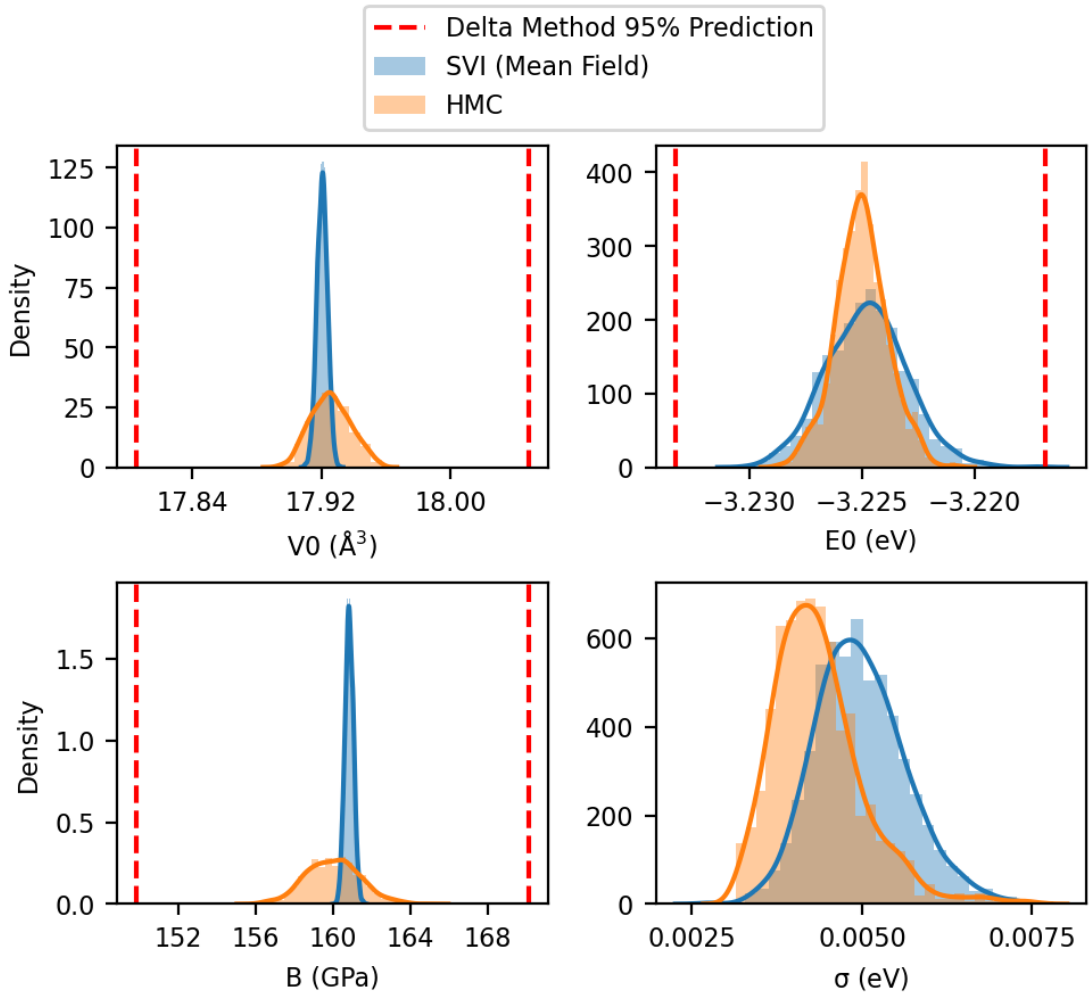

Figure S20: Au polynomial SVI HMC Delta

3.2.4 Murnaghan

```
import numpy as np
import torch
import bayesreg
from bayesreg import *

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')
V = torch.Tensor(V)
nrg = torch.Tensor(nrg)

def model(V, nrg, func):
```

```
    e0 = pyro.sample("e0", dist.Normal(-3., 1.))
    b = pyro.sample("b", dist.Normal(0.8, 0.2))
    bp = pyro.sample("bp", dist.Normal(5., 1.))
    v0 = pyro.sample("v0", dist.Normal(18., 7.5))
    sigma = pyro.sample("sigma", dist.Uniform(0., 1.))
    mean = func(V, e0, b, bp, v0)
    with pyro.plate("data", len(V)):
        pyro.sample("obs", dist.Normal(mean, sigma), obs=nrg)

optim_vi(model, V, nrg, murnaghan, 30000, 'au-murn')

print(f'''#+attr_org: :width 600
#+caption: ELBO Au murnaghan
[[./elbo-au-murn.png]]''')
```

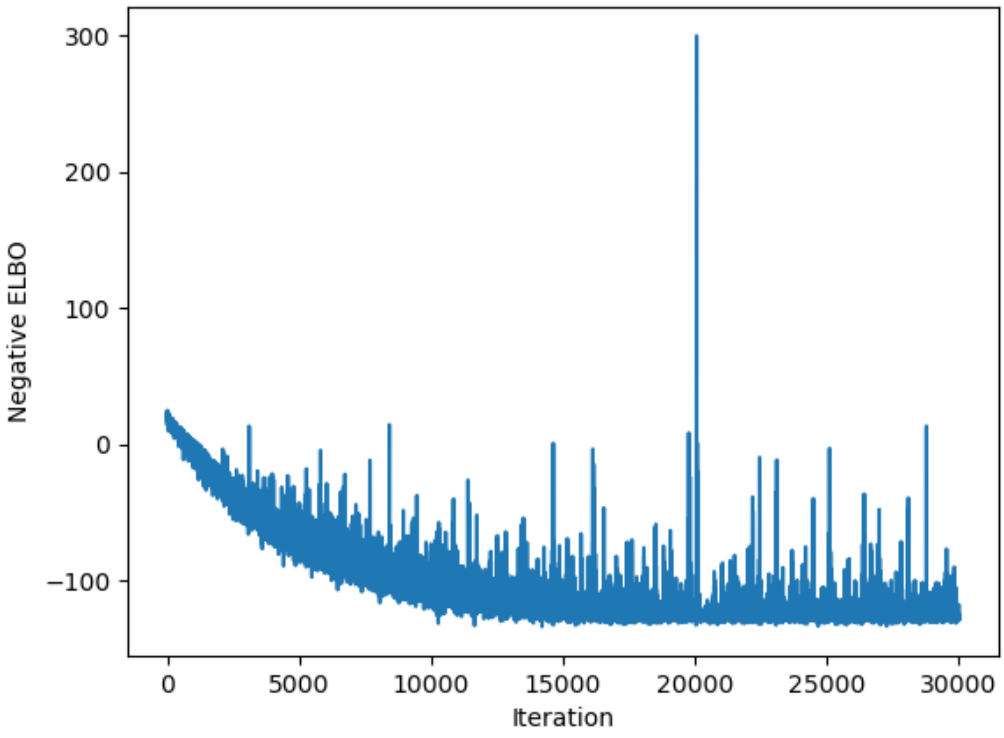

Figure S21: ELBO Au murnaghan

```
from pyro.infer import MCMC, NUTS
import pyro.distributions as dist
```

```
import bayesreg
from bayesreg import *
import numpy as np
import torch
import pickle

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')
V = torch.Tensor(V)
nrg = torch.Tensor(nrg)

nuts_kernel = NUTS(model_others)

mcmc = MCMC(nuts_kernel, num_samples=1000, warmup_steps=200)
mcmc.run(V, nrg, murnaghan)

hmc_samples = {k: v.detach().cpu().numpy() for k, v in mcmc.get_samples().items()}

with open('au-murn-hmc-samples.pickle', 'wb') as handle:
    pickle.dump(hmc_samples, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

```
import pickle
import numpy as np
import bayesreg
from bayesreg import make_plot

with open('au-murn-svi-mvn-samples.pickle', 'rb') as handle:
    svi_mvn_samples = pickle.load(handle)
with open('au-murn-hmc-samples.pickle', 'rb') as handle:
    hmc_samples = pickle.load(handle)

deltam = np.array([[17.9557, 18.0125],[-3.2232,-3.2189], [135.008,139.769]])

make_plot(svi_mvn_samples, hmc_samples, deltam,
          'au-murn-svi-hmc-delta', nbinsi = [4,4,6,3])

print(f'''#+attr_org: :width 600
#+caption: Au murnaghan SVI HMC Delta
[[./au-murn-svi-hmc-delta.png]]''')
```

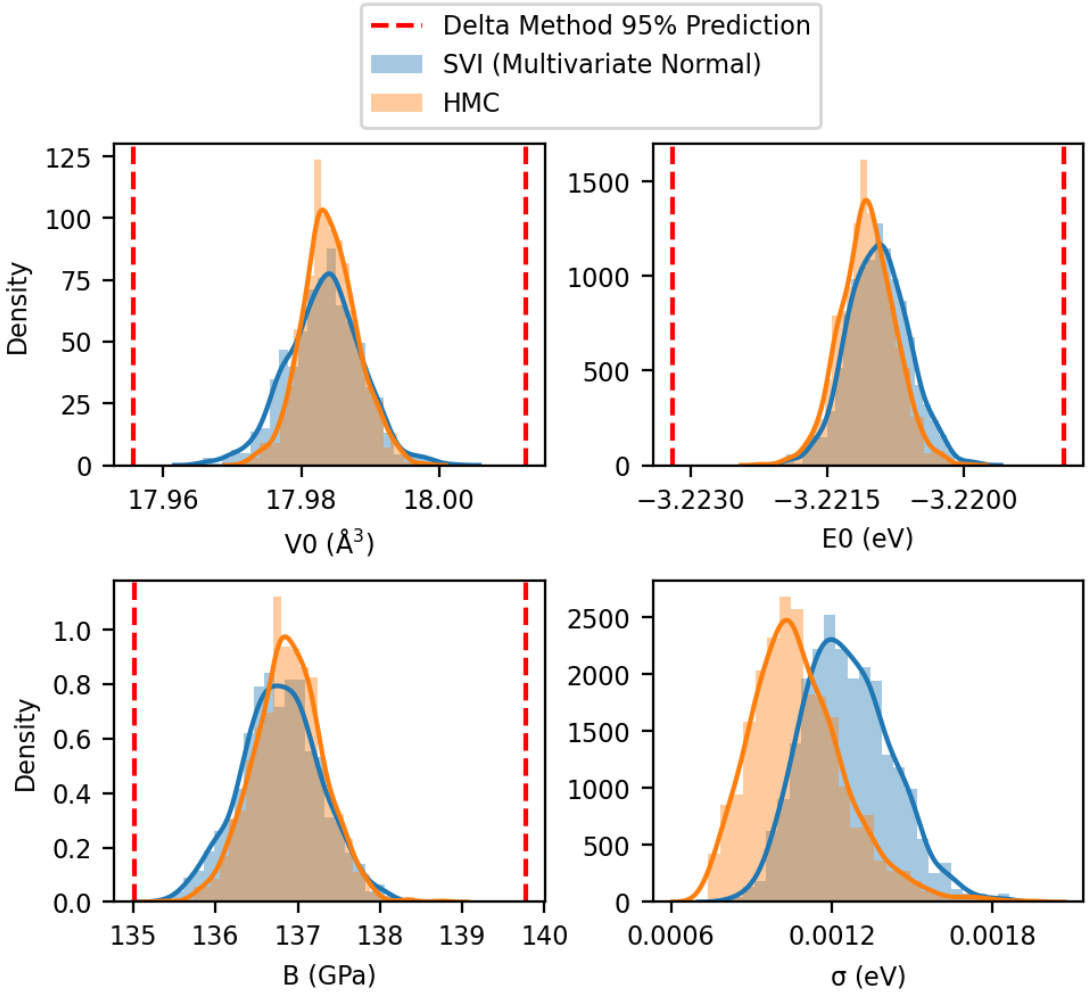

Figure S22: Au murnaghan SVI HMC Delta

3.2.5 Birch

Init Guess

e0 = pyro.sample("e0", dist.Normal(-3., 1.))

b = pyro.sample("b", dist.Normal(0.9, 0.2))

bp = pyro.sample("bp", dist.Normal(6., 1.))

v0 = pyro.sample("v0", dist.Normal(18., 7.5))

sigma = pyro.sample("sigma", dist.Uniform(0., 1.))

```
python bayesreg.py --element au --model birch --run svi --gamma 0.6
```

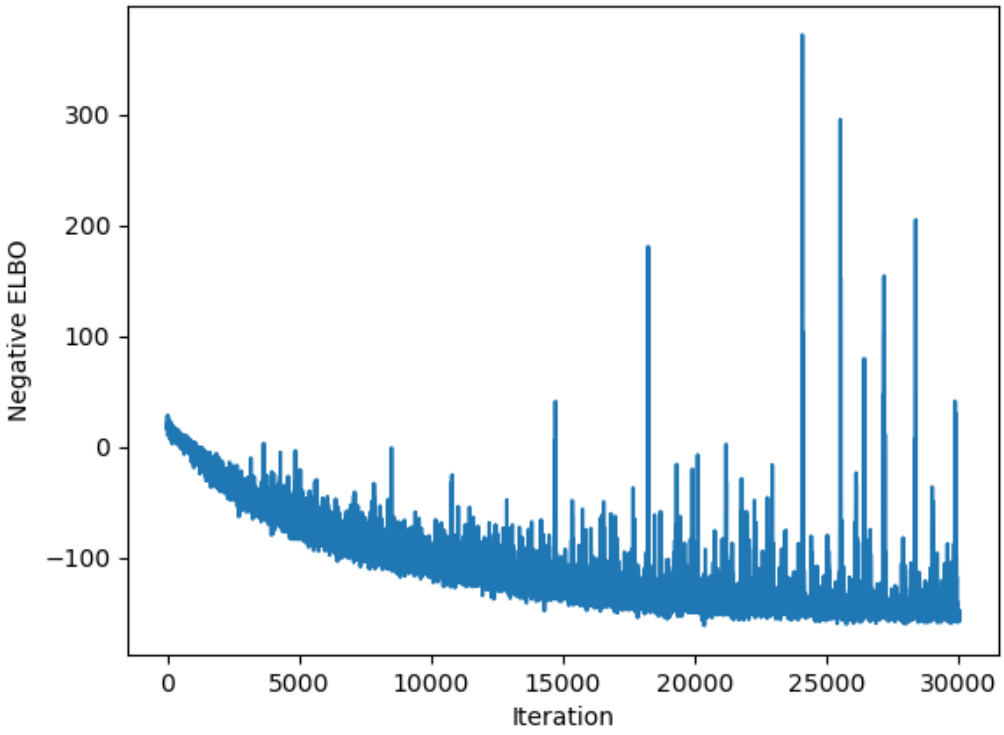

Figure S23: ELBO Au birch

```
python bayesreg.py --element au --model birch --run hmc
```

```
python bayesreg.py --element au --model birch --run plot --ticks 4-4-5-3
```

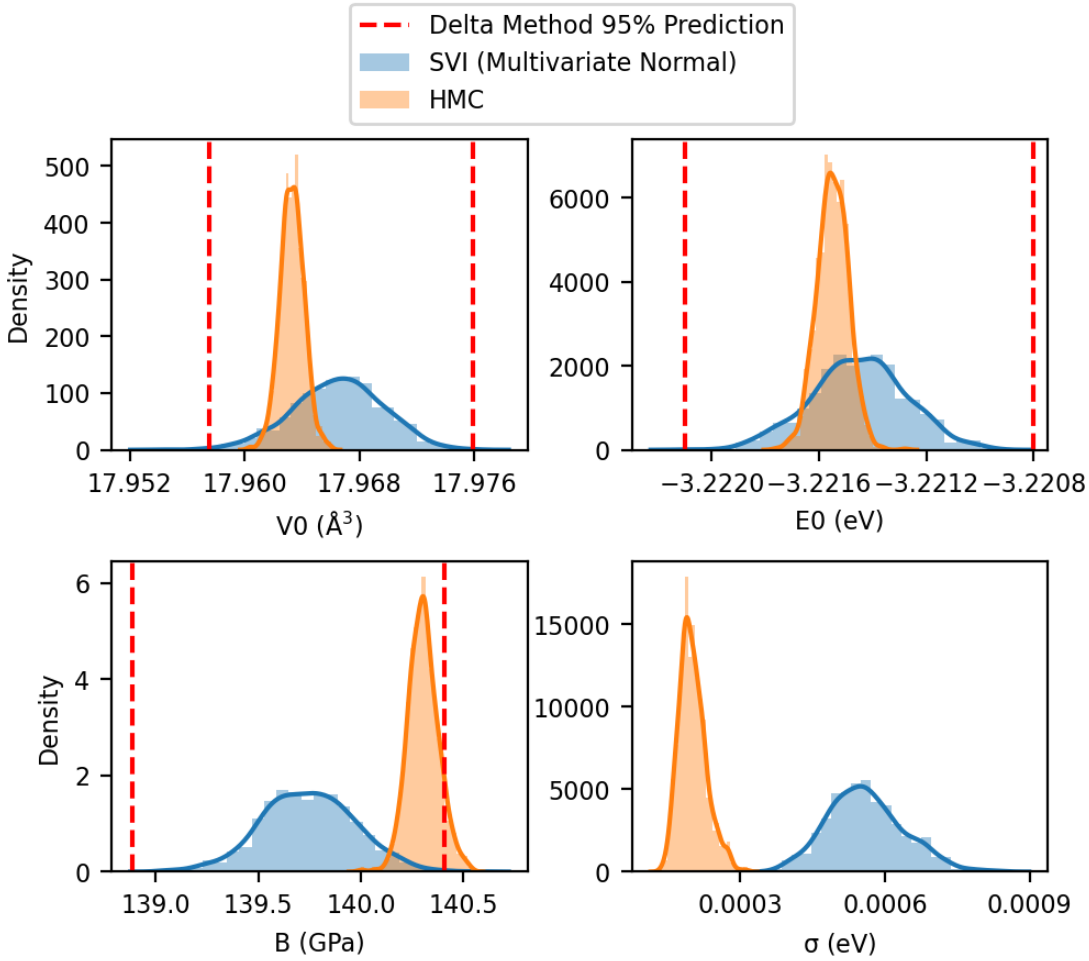

Figure S24: Au birch SVI HMC Delta

3.2.6 PoirierTarantola

```
python bayesreg.py --element au --model pt --run svi
```

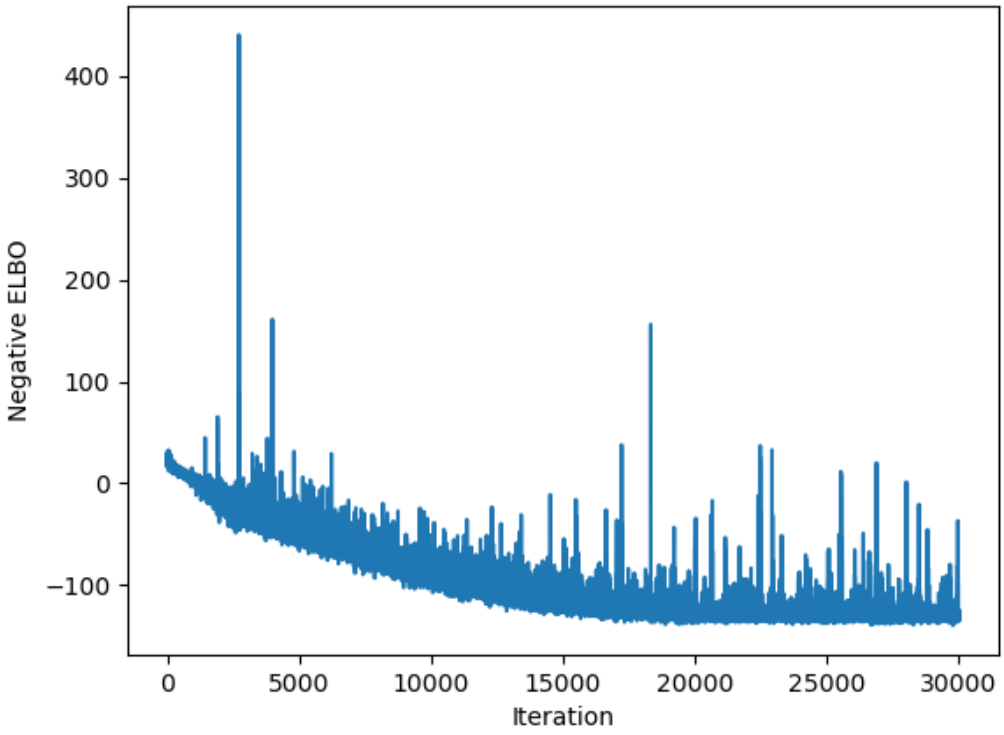

Figure S25: ELBO Pd poiriertarantola

```
python bayesreg.py --element au --model pt --run hmc
```

```
python bayesreg.py --element au --model pt --run plot
```

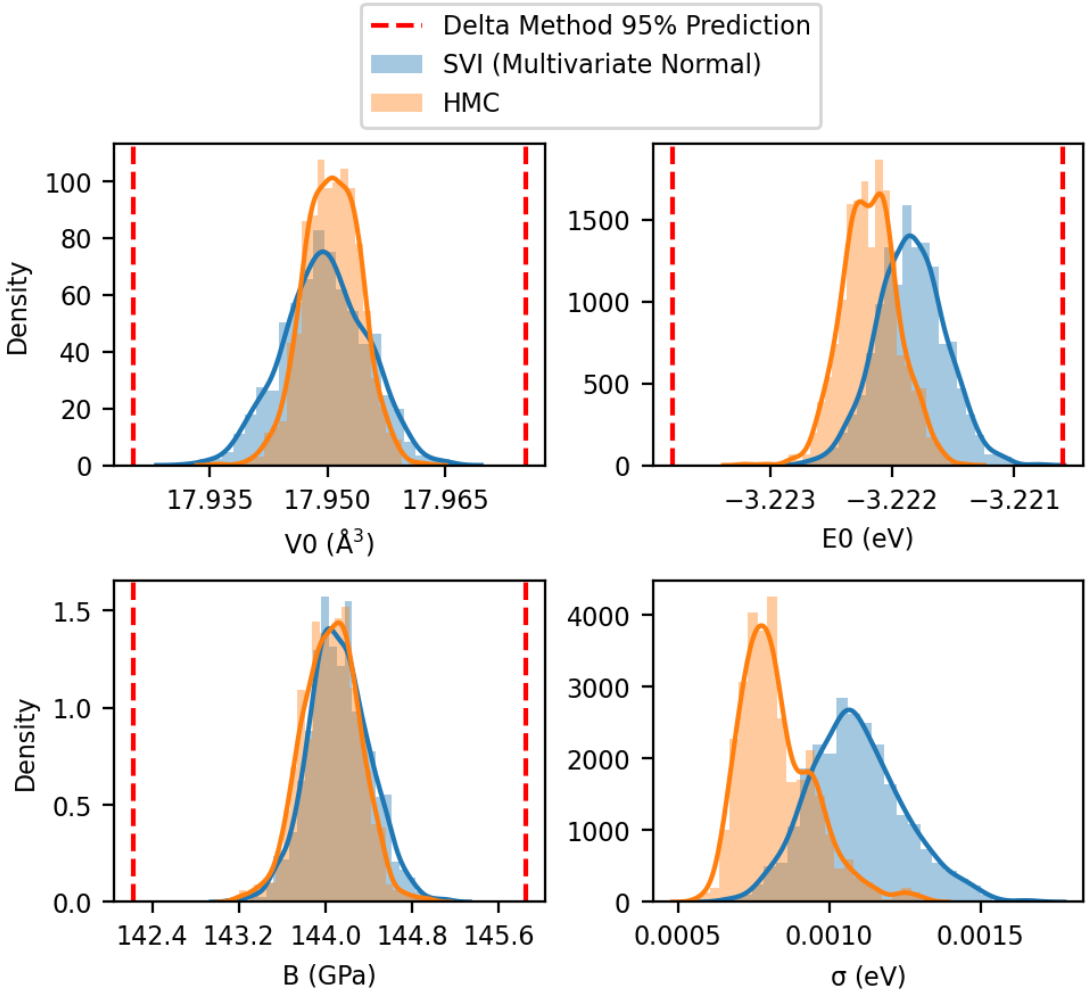

Figure S26: Au poirier-tarantola SVI HMC Delta

3.2.7 Vinet

```
import numpy as np
import torch
import bayesreg
from bayesreg import *

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')
V = torch.Tensor(V)
nrg = torch.Tensor(nrg)

def model(V, nrg, func):
```

```
    e0 = pyro.sample("e0", dist.Normal(-3., 1.))
    b = pyro.sample("b", dist.Normal(0.9, 0.2))
    bp = pyro.sample("bp", dist.Normal(6., 1.))
    v0 = pyro.sample("v0", dist.Normal(18., 7.5))
    sigma = pyro.sample("sigma", dist.Uniform(0., 1.))
    mean = func(V, e0, b, bp, v0)
    with pyro.plate("data", len(V)):
        pyro.sample("obs", dist.Normal(mean, sigma), obs=nrg)

num_iters = 30000
gamma = 0.6**(1/num_iters)
optim_vi(model, V, nrg, vinet, num_iters, 'au-vinet', gamma=gamma)

print(f'''#+attr_org: :width 600
#+caption: ELBO Au vinet
[[./elbo-au-vinet.png]]''')
```

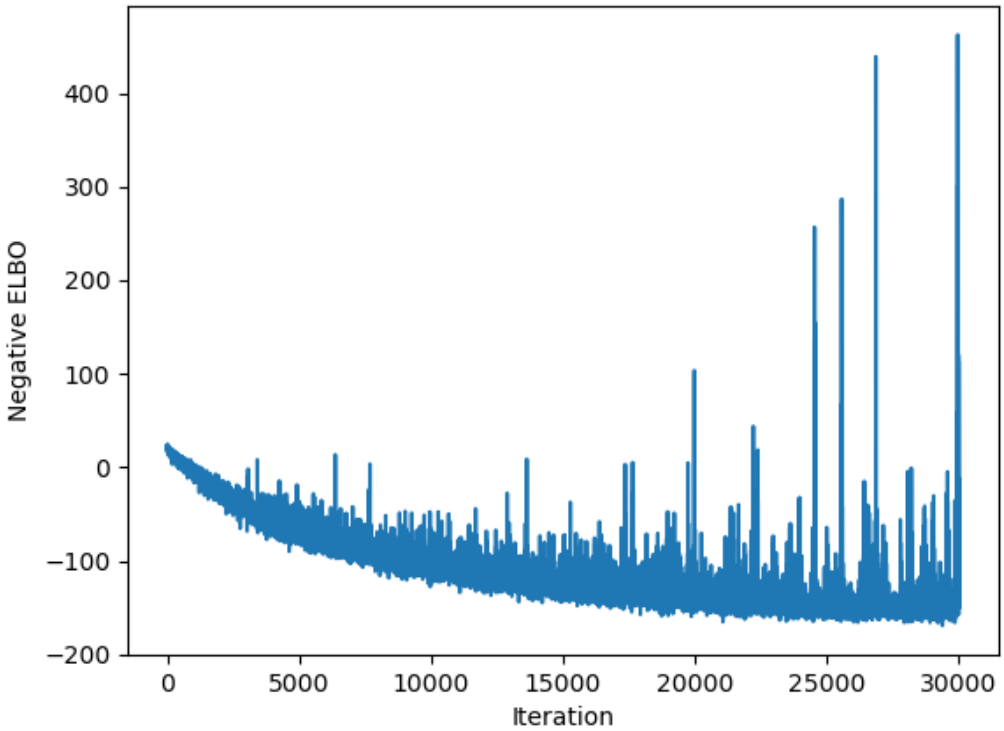

Figure S27: ELBO Au vinet

```
from pyro.infer import MCMC, NUTS
```

```
import pyro.distributions as dist
import bayesreg
from bayesreg import *
import numpy as np
import torch
import pickle

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')
V = torch.Tensor(V)
nrg = torch.Tensor(nrg)

nuts_kernel = NUTS(model_others)

mcmc = MCMC(nuts_kernel, num_samples=1000, warmup_steps=200)
mcmc.run(V, nrg, vinet)

hmc_samples = {k: v.detach().cpu().numpy() for k, v in mcmc.get_samples().items()}

with open('au-vinet-hmc-samples.pickle', 'wb') as handle:
    pickle.dump(hmc_samples, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

```
import pickle
import numpy as np
import bayesreg
from bayesreg import make_plot

with open('au-vinet-svi-mvn-samples.pickle', 'rb') as handle:
    svi_mvn_samples = pickle.load(handle)
with open('au-vinet-hmc-samples.pickle', 'rb') as handle:
    hmc_samples = pickle.load(handle)

deltam = np.array([[17.9576, 17.9689],[-3.2219,-3.2212], [139.8,140.806]])

make_plot(svi_mvn_samples, hmc_samples, deltam,
          'au-vinet-svi-hmc-delta', nbinsi = [4,4,5,3],
          wspace=0.28)
print(f'''#+attr_org: :width 600
#+caption: Au vinet SVI HMC Delta
[[./au-vinet-svi-hmc-delta.png]]''')
```

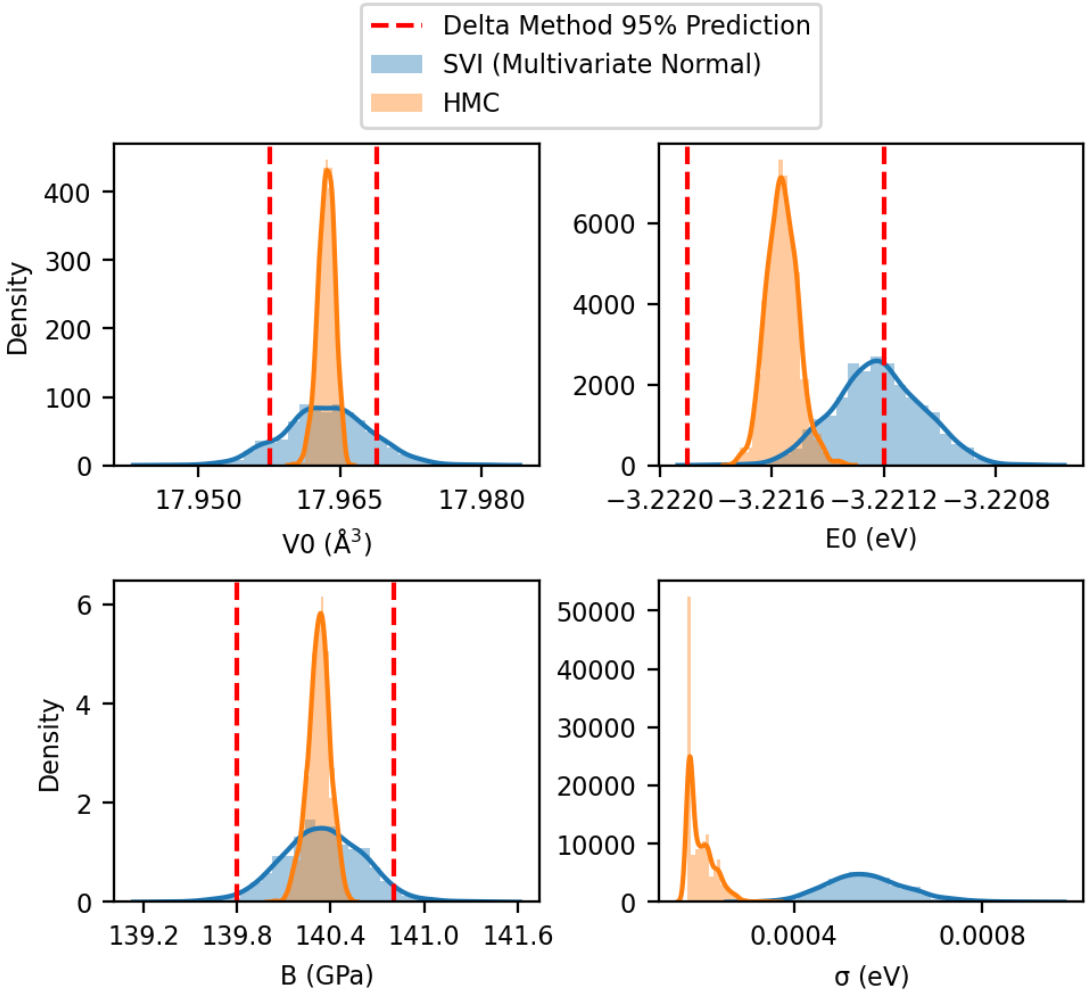

Figure S28: Au vinet SVI HMC Delta

4 GP

In this section, we have Gaussian process results for Pd and Au.

4.1 Pd

The code below defines the GP using `gpytorch` and performs the training. We make plots to check the loss over training, and that the hyperparameters converged. We take note of the transformed output-scale and lengthscale.

```
import gpytorch
import torch
import matplotlib.pyplot as plt
import numpy as np

class ExactGPModel(gpytorch.models.ExactGP):
    def __init__(self, train_x, train_y, likelihood):
        super(ExactGPModel, self).__init__(train_x, train_y, likelihood)
        self.mean_module = gpytorch.means.ConstantMean()
        self.covar_module = gpytorch.kernels.ScaleKernel(gpytorch.kernels.RBFKernel())

    def forward(self, x):
        mean_x = self.mean_module(x)
        covar_x = self.covar_module(x)
        return gpytorch.distributions.MultivariateNormal(mean_x, covar_x)

# initialize likelihood and model
likelihood = gpytorch.likelihoods.GaussianLikelihood(require_grad=True)

V = np.load('v-pd.npy')
nrg = np.load('nrg-pd.npy')
V = torch.Tensor(V)
nrg = torch.Tensor(nrg)

model = ExactGPModel(V, nrg, likelihood)

# Find optimal model hyperparameters
model.train()
likelihood.train()

# Use the adam optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.1)

# "Loss" for GPs - the marginal log likelihood
mll = gpytorch.mlls.ExactMarginalLogLikelihood(likelihood, model)

training_iter = 2000

losses = []
ls = []
```

```
outputscales = []
for i in range(training_iter):
    # Zero gradients from previous iteration
    optimizer.zero_grad()
    # Output from model
    output = model(V)
    # Calc loss and backprop gradients
    loss = -mll(output, nrg)
    loss.backward()
    losses += [loss.item()]
    ls += [model.covar_module.base_kernel.lengthscale.item()]
    outputscales += [model.covar_module.raw_outputscale.item()]
    optimizer.step()

raw_noise =  model.likelihood.noise_covar.raw_noise
constraint =  model.likelihood.noise_covar.raw_noise_constraint

print('Transformed noise:', f'{constraint.transform(raw_noise).item():.6f}')

raw_lengthscale = model.covar_module.base_kernel.raw_lengthscale
constraint = model.covar_module.base_kernel.raw_lengthscale_constraint

print('Transformed lengthscale:', f'{constraint.transform(raw_lengthscale).item():.4f}')

raw_outputscale = model.covar_module.raw_outputscale
constraint = model.covar_module.raw_outputscale_constraint

print('Transformed outputscale:', f'{constraint.transform(raw_outputscale).item():.4f}')

#make plot
plt.clf()
plt.plot(losses)
plt.xlabel('Iteration')
plt.ylabel('- Marginal Log Likelihood')
plt.savefig('mll-pd-gp.png')
print(f'''#+attr_org: :width 600
#+caption: MLL Pd Gaussian process
[[./mll-pd-gp.png]]''')

plt.clf()
plt.plot(ls)
```

```
plt.xlabel('Iteration')
plt.ylabel('Raw Lengthscale')
plt.savefig('pd-gp-ls.png')
print(f'''#+attr_org: :width 600
#+caption: Pd Gaussian process raw lengthscale
[[./pd-gp-ls.png]]''')

plt.clf()
plt.plot(outputscales)
plt.xlabel('Iteration')
plt.ylabel('Raw Outputscale')
plt.savefig('pd-gp-opscale.png')
print(f'''#+attr_org: :width 600
#+caption: Pd Gaussian process raw outputscale
[[./pd-gp-opscale.png]]''')
```

Transformed noise: 0.000100 Transformed lengthscale: 2.5931 Transformed outputscale: 0.0796

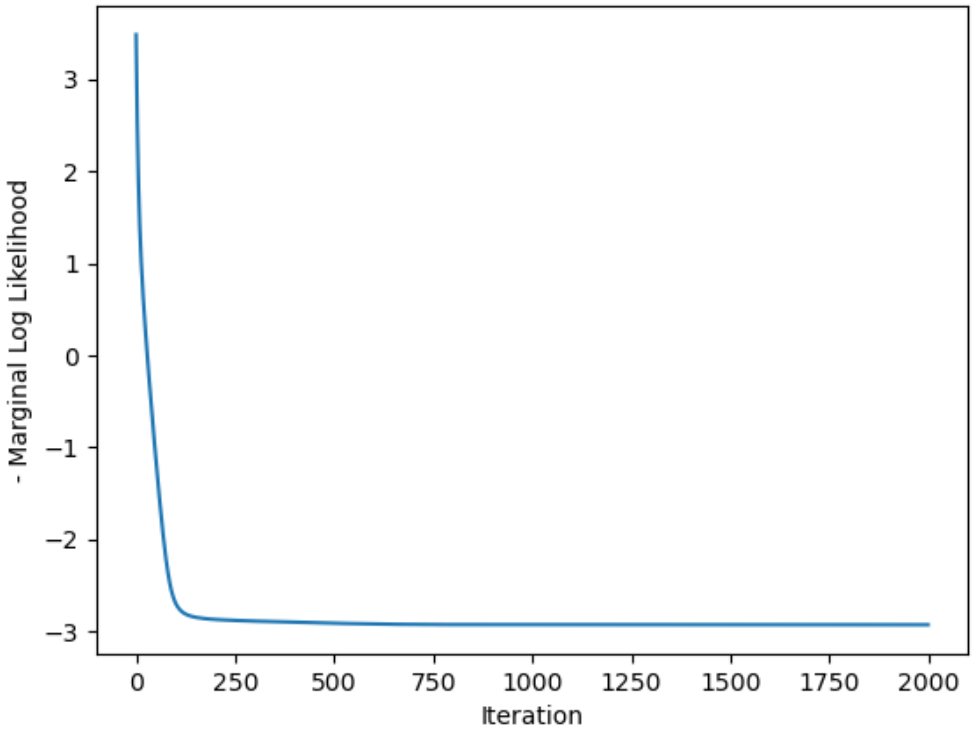

Figure S29: MLL Pd Gaussian process

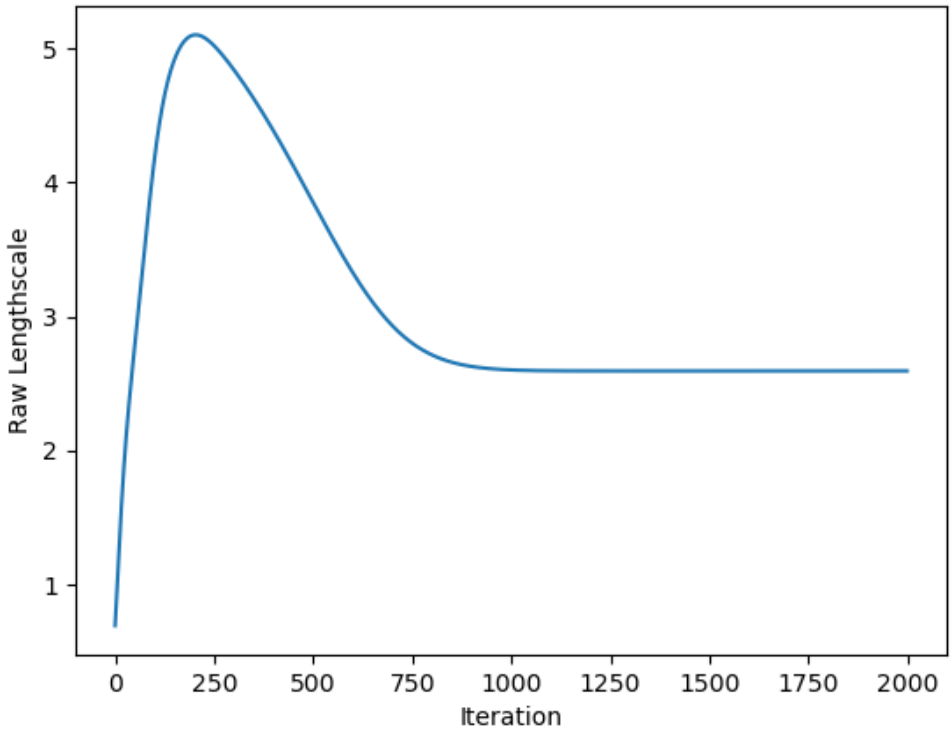

Figure S30: Pd Gaussian process raw lengthscale

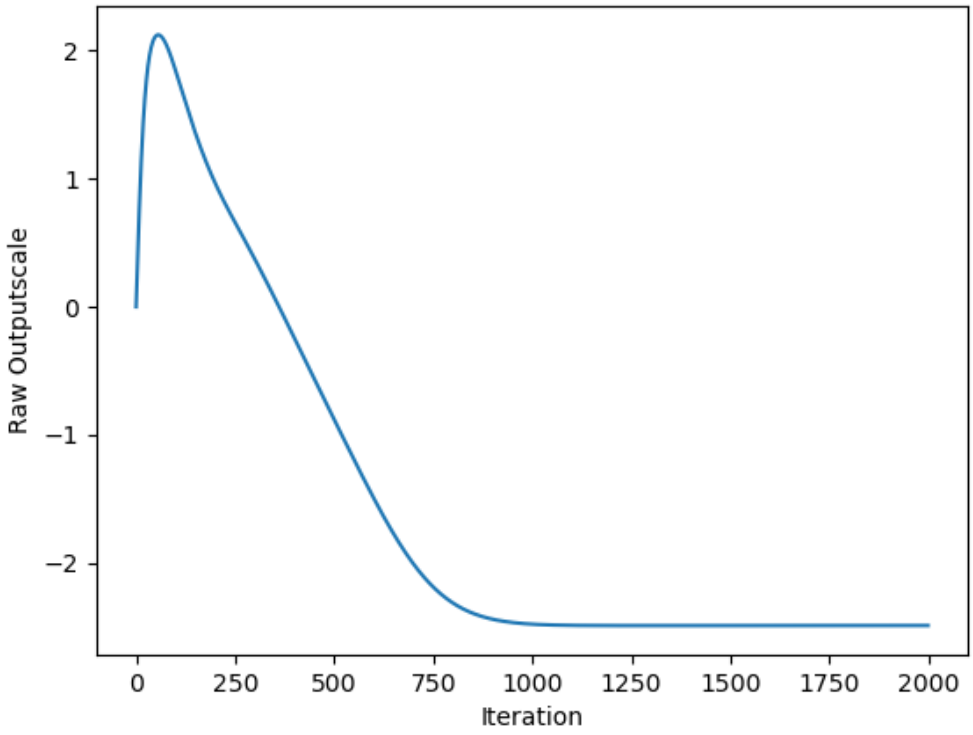

Figure S31: Pd Gaussian process raw outputscale

To find GP mean and covariance, we use the Colab notebook because it has jax capability. The code in the Colab notebook has the calculation of GP joint over function, first and second derivative. We use the transformed outputscale and lengthscale from gpytorch as hyperparameters in the kernel. https://colab.research.google.com/drive/1yZW7I-TFTo72PUGkbdRAOkkq1DkAEtCB?usp=sharing

The code below plots the mean, ± 2 std.dev., and samples from the GP posterior.

```
import matplotlib.pyplot as plt
import matplotlib as mpl
import numpy as np

V = np.load('v-pd.npy')
nrg = np.load('nrg-pd.npy')

mu1 = np.load('mu1.npy')
```

```
var1 = np.load('var1.npy')
samples = np.load('gp-samples-pd.npy')

plt.rcParams.update({'font.size': 12})
mpl.rcParams['mathtext.default'] = 'regular'

fig, ax = plt.subplots(ncols=1, nrows=3,
                       sharex=True, sharey=False,
                       figsize=(5.5, 7.4))
v_test1 = np.linspace(V[0], V[-1], 1000)
slices = [np.s_[:1000],np.s_[1000:2000], np.s_[2000:]]

for i in range(3):
    for j in range(1000):
        ax[i].plot(v_test1, samples[j,slices[i]],
                   c='gray', alpha=0.3,
                   label='1000 samples', linewidth=0.7)
    y_std = np.sqrt(np.diagonal(var1[slices[i], slices[i]]))
    y_mean = mu1[slices[i]]
    ax[i].plot(v_test1, y_mean, 'k', label='Mean', linewidth=0.7)
    ax[i].plot(v_test1, y_mean-2*y_std, 'r', ls='--', linewidth=0.9)
    ax[i].plot(v_test1, y_mean+2*y_std, 'r',
               ls='--', linewidth=0.7, label=r"$\pm$ 2 std. dev.")

ax[2].set_xlabel('V ($\AA^3$)')
ax[0].set_ylabel('E (eV)')
ax[1].set_ylabel(r'$\frac{d E}{d V}$ (eV/$\AA^3$)')
ax[2].set_ylabel(r'$\frac{d^2 E}{d V^2}$ (eV/$\AA^6$)')
handles, labels = plt.gca().get_legend_handles_labels()
ax[2].legend([handles[-2], handles[-1], handles[0]],
             [labels[-2], labels[-1], labels[0]])
ax[0].annotate("a)", xy=(-0.2, 0.93), xycoords="axes fraction")
ax[1].annotate("b)", xy=(-0.2, 0.93), xycoords="axes fraction")
ax[2].annotate("c)", xy=(-0.2, 0.93), xycoords="axes fraction")


plt.tight_layout()
plt.subplots_adjust(hspace=0)
plt.savefig('gp-posterior-pd.png', dpi=300)

print(f'''#+attr_org: :width 600
```

```
50  #+caption: Pd Gaussian process posterior
51  [[./gp-posterior-pd.png]]''')
```

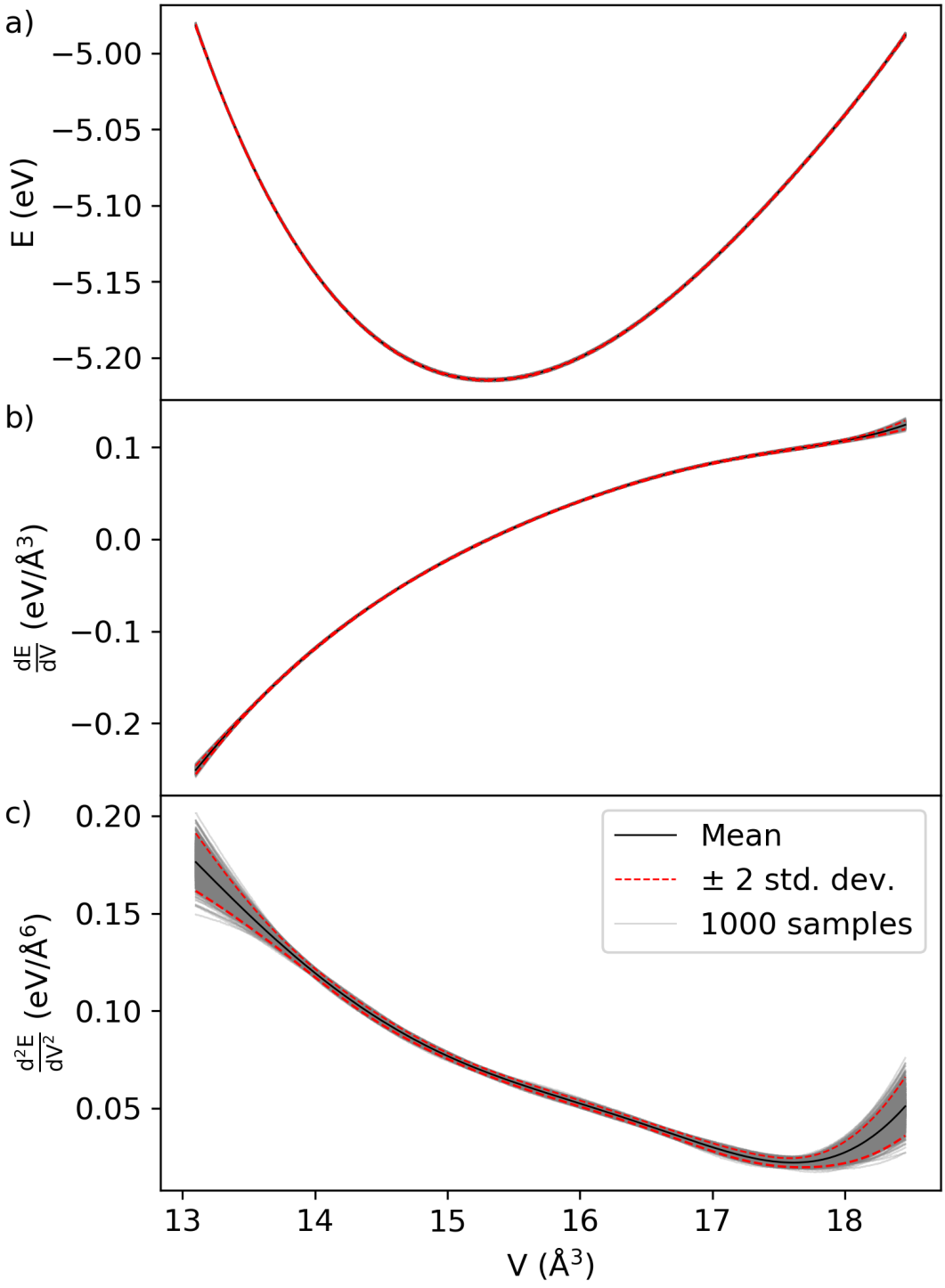

Figure S32: Pd Gaussian process posterior

The code below makes the plot with HMC posterior, delta method interval, and Gaussian process posterior. We also include the MLE estimates from all of the nonlinear models.

```
import matplotlib as mpl
import re
import matplotlib.pyplot as plt
mpl.rcParams['mathtext.default'] = 'regular'
from scipy.interpolate import interp1d
from scipy.optimize import brentq
import numpy as np
import pickle
import seaborn as sns

def beos_gp(sample):
    '''
    we are going to assume that the observations, deriv, and 2nd deriv
    are sampled at the same x points
    also going to assume that the x points are in order from smallest to largest.
    '''
    func = interp1d(v_test1, sample[:1000], fill_value='extrapolate')
    deriv = interp1d(v_test1, sample[1000:2000],fill_value='extrapolate')
    secderiv = interp1d(v_test1, sample[2000:],fill_value='extrapolate')
    vmin = brentq(deriv, 12.5, 17.5)
    emin = func(vmin)
    secderivmin = secderiv(vmin)
    return vmin, emin, vmin*secderivmin

def get_phys(samples):
    vmins = []
    emins = []
    bmods = []

    numsample = len(samples)

    if numsample > 1000:
        randinds = np.random.choice(np.arange(numsample), 1000, replace=False)

    else:
        randinds = np.arange(samples.shape[0])

    for i in randinds:
```

```
        a, b, c = beos_gp(samples[i])
        vmins += [a]
        emins += [b]
        bmods += [c]

    return np.array(vmins), np.array(emins), np.array(bmods)

V = np.load('v-pd.npy')
nrg = np.load('nrg-pd.npy')
v_test1 = np.linspace(V[0], V[-1], 1000)
samples = np.load('gp-samples-pd.npy')
gpvmins, gpemins, gpbmods = get_phys(samples)

with open('pd-pt-hmc-samples.pickle', 'rb') as handle:
    pt_hmc_samples = pickle.load(handle)

with open('pd-birch-hmc-samples.pickle', 'rb') as handle:
    birch_hmc_samples = pickle.load(handle)

birch_deltam = np.array([[15.3006, 15.3052],[-5.2146,-5.2143], [167.338,167.866]])

pt_deltam = np.array([[15.2945, 15.3198],[-5.2158,-5.2141], [168.979,172.418]])

nonlin_dict = {'e0': [-5.2146, -5.2144, -5.2164, -5.2140, -5.21457],
               'v0': [15.3041, 15.3029, 15.3311, 15.3017, 15.3034],
               'b': [1.0536, 1.0459, 1.1208, 1.0281, 1.0510],
               'nl_model': ['SJ', 'AS', 'P3', 'M', 'V']}

def make_plot_model(gpdist, hmc_samples, deltam,
                    modeln, site, xlabel, descr,
                    nonlin_dict = None):
    plt.clf()
    colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
    markers = ['o', 's', '^', '*', 'D']
    if site=='b':
        factor = 160.2
    else:
        factor = 1.0
    sns.histplot(factor*gpdist, label='GP', kde=True,  linewidth=0)
    for i,hmc_sample in enumerate(hmc_samples):
        sns.histplot(factor*hmc_sample[site], label=f'{modeln[i]} HMC',
```

```
                    kde=True, linewidth=0, color=colors[i+1])
    for i,delta in enumerate(deltam):
        plt.axvline(x=delta[0], color=colors[i+1], ls='--', label=f'{modeln[i]} Delta')
        plt.axvline(x=delta[1], color=colors[i+1], ls='--')
    if nonlin_dict is not None:
        for i, value in enumerate(nonlin_dict[site]):
            plt.plot(factor*value,5, marker=markers[i], color=colors[i+3], ls='None',
                    label=nonlin_dict['nl_model'][i])
    handles, labels = plt.gca().get_legend_handles_labels()
    order = [7,8,0,9,1,2,3,4,5,6]
    plt.legend([handles[idx] for idx in order],[labels[idx] for idx in order])
    plt.xlabel(xlabel)
    plt.tight_layout()
    plt.savefig(f'{descr}-gp-{"-".join(modeln)}-{site}.png', dpi=200,bbox_inches='tight')

make_plot_model(gpbmods,
                [birch_hmc_samples, pt_hmc_samples],
                [birch_deltam[2], pt_deltam[2]],
                ['Birch','PT'],
                'b', 'Bulk Modulus (GPa)','pd', nonlin_dict)

make_plot_model(gpvmins, [birch_hmc_samples, pt_hmc_samples],
                [birch_deltam[0], pt_deltam[0]], ['Birch','PT'],
                'v0', 'Volume ($\AA^3$)','pd', nonlin_dict)

make_plot_model(gpemins, [birch_hmc_samples, pt_hmc_samples],
                [birch_deltam[1], pt_deltam[1]], ['Birch','PT'],
                'e0', 'Energy (eV)','pd', nonlin_dict)

print(f'''#+attr_org: :width 600
#+caption: Pd model uncertainties bulk modulus
[[./pd-gp-birch-pt-b.png]]''')

print(f'''#+attr_org: :width 600
#+caption: Pd model uncertainties E0
[[./pd-gp-birch-pt-e0.png]]''')

print(f'''#+attr_org: :width 600
#+caption: Pd model uncertainties V0
[[./pd-gp-birch-pt-v0.png]]''')
```

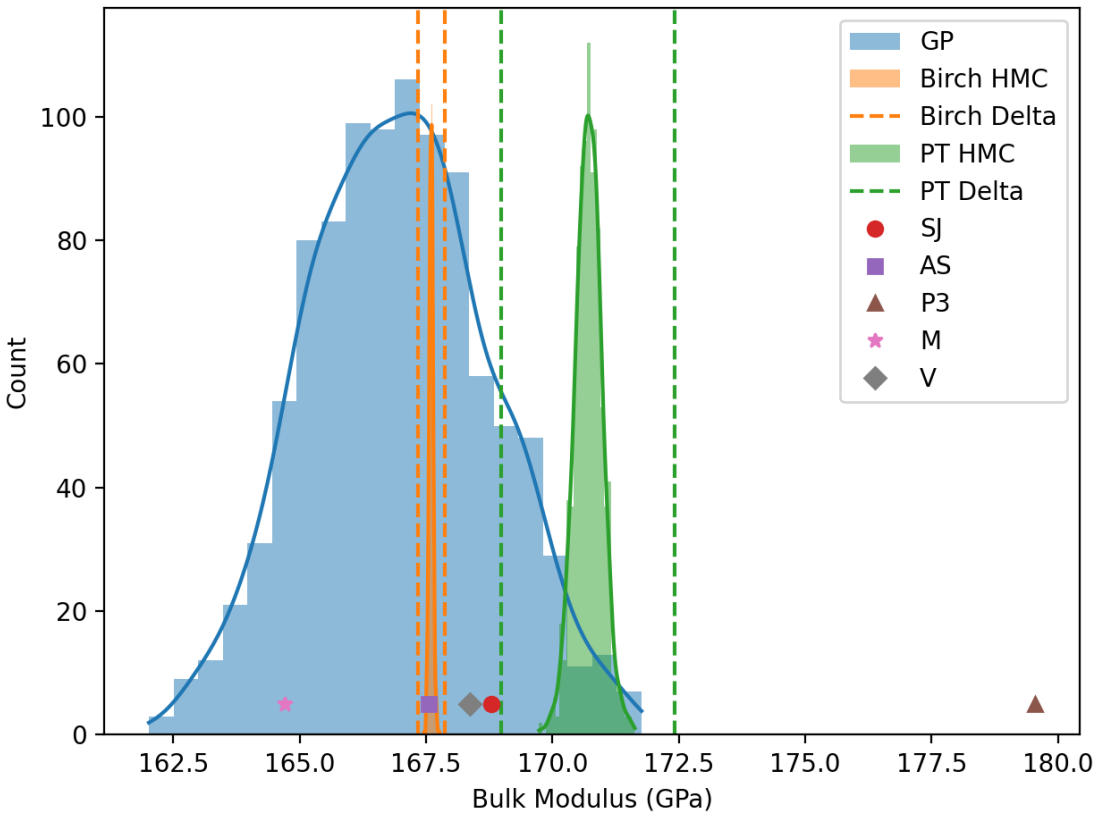

Figure S33: Pd model uncertainties bulk modulus

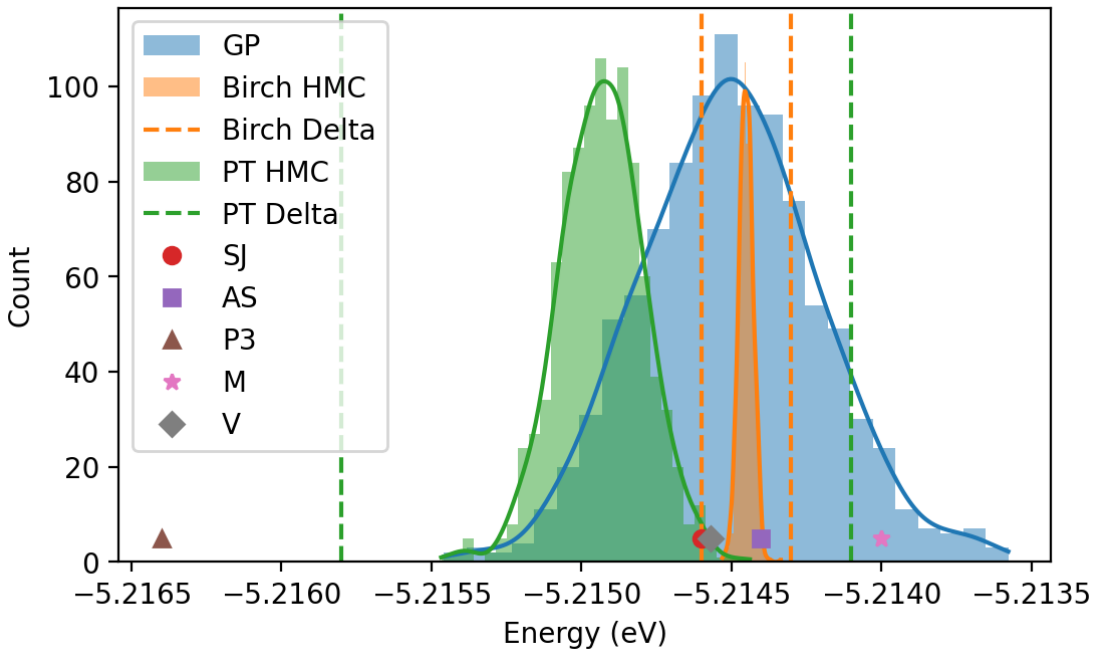

Figure S34: Pd model uncertainties E0

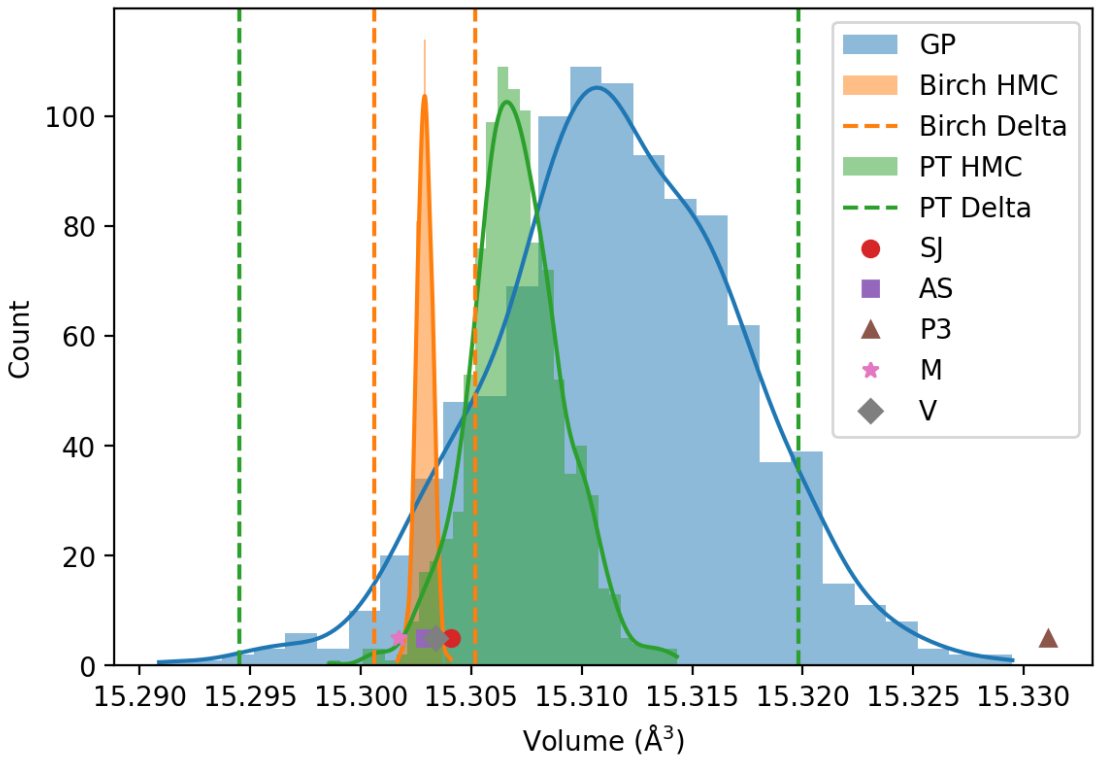

Figure S35: Pd model uncertainties V0

4.2 Au

```
1  import torch
2  import numpy as np
3  from gp import *
4
5  V = np.load('v-au.npy')
6  nrg = np.load('nrg-au.npy')
7  V = torch.Tensor(V)
8  nrg = torch.Tensor(nrg)
9
10 train_gp(V, nrg, 'Au')
```

Transformed noise: 0.000100 Transformed lengthscale: 4.2634 Transformed outputscale: 0.8007

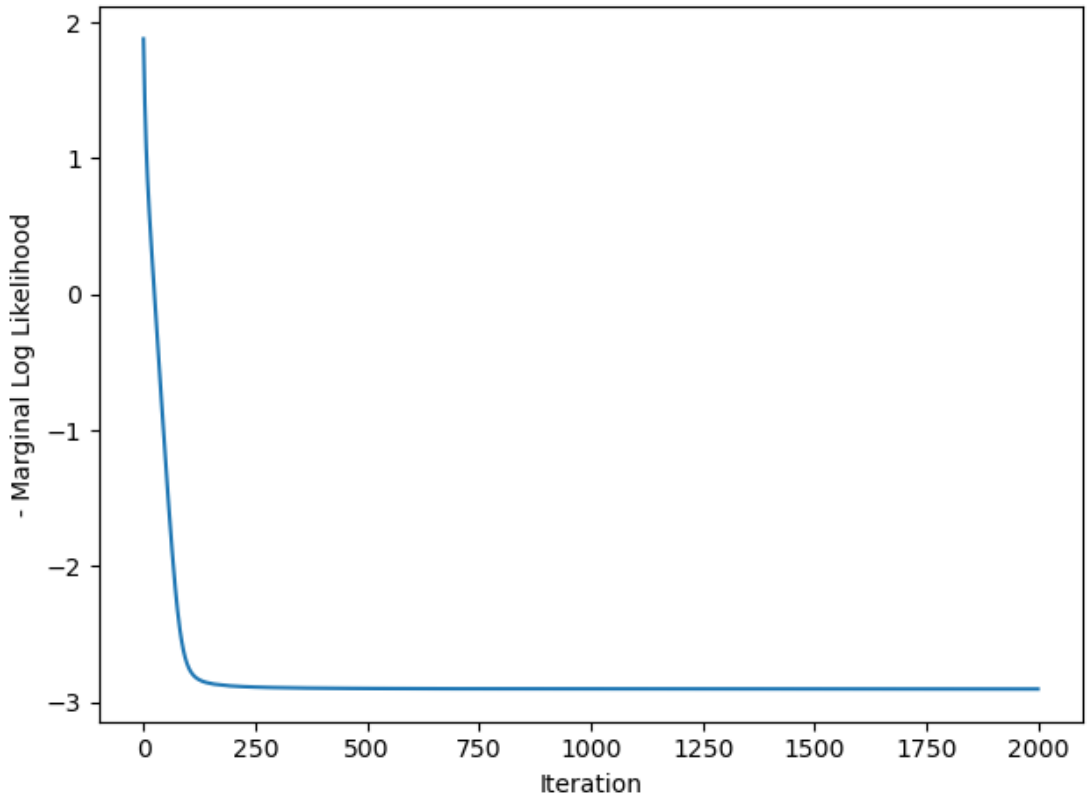

Figure S36: MLL Au Gaussian process

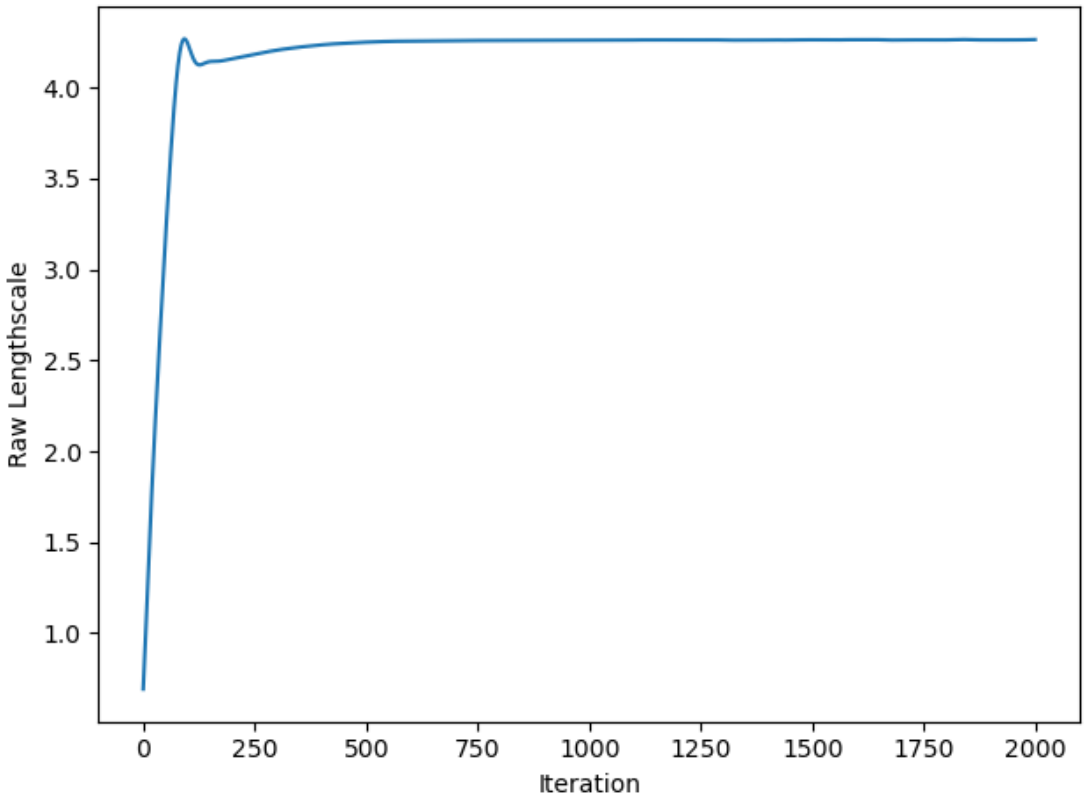

Figure S37: Au Gaussian process raw lengthscale

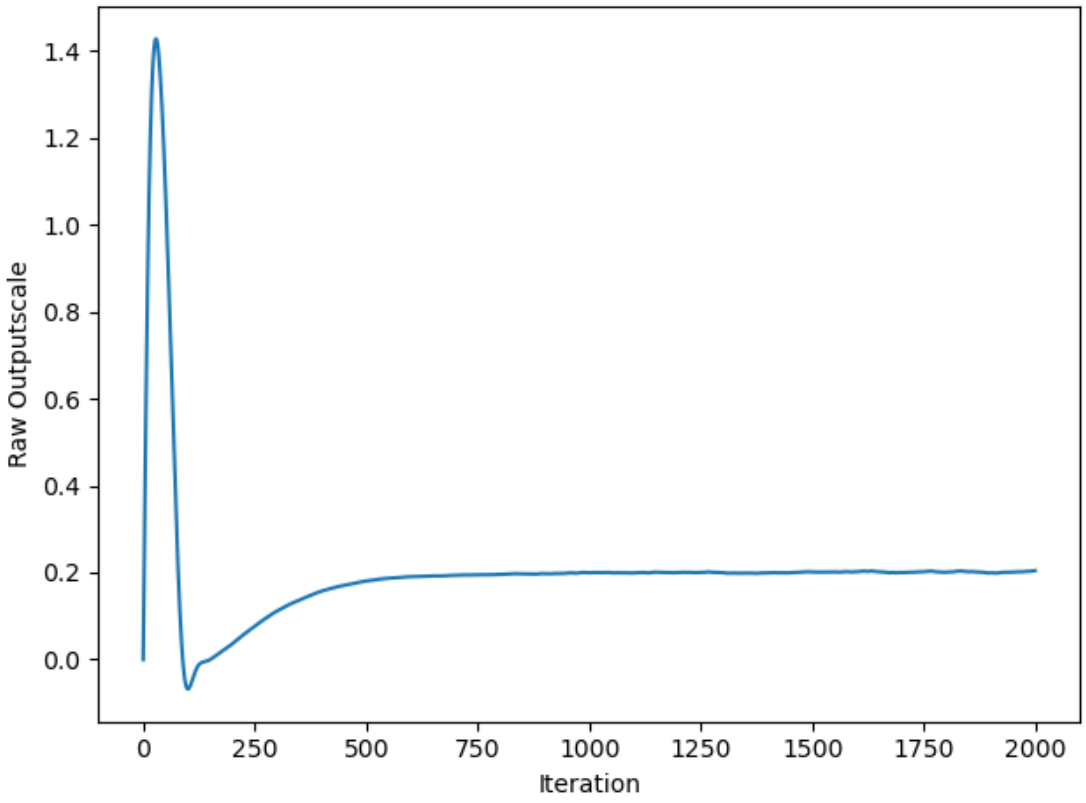

Figure S38: Au Gaussian process raw outputscale

```
import numpy as np
from gp import *

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')

mu1 = np.load('gp-mu-au.npy')
var1 = np.load('gp-var-au.npy')
samples = np.load('gp-samples-au.npy')
v_test1 = np.linspace(V[0], V[-1], 1000)

make_gp_posterior(mu1, var1, samples, v_test1, 'au')

print(f'''#+attr_org: :width 600
#+caption: Au Gaussian process posterior
[[./gp-posterior-au.png]]''')
```

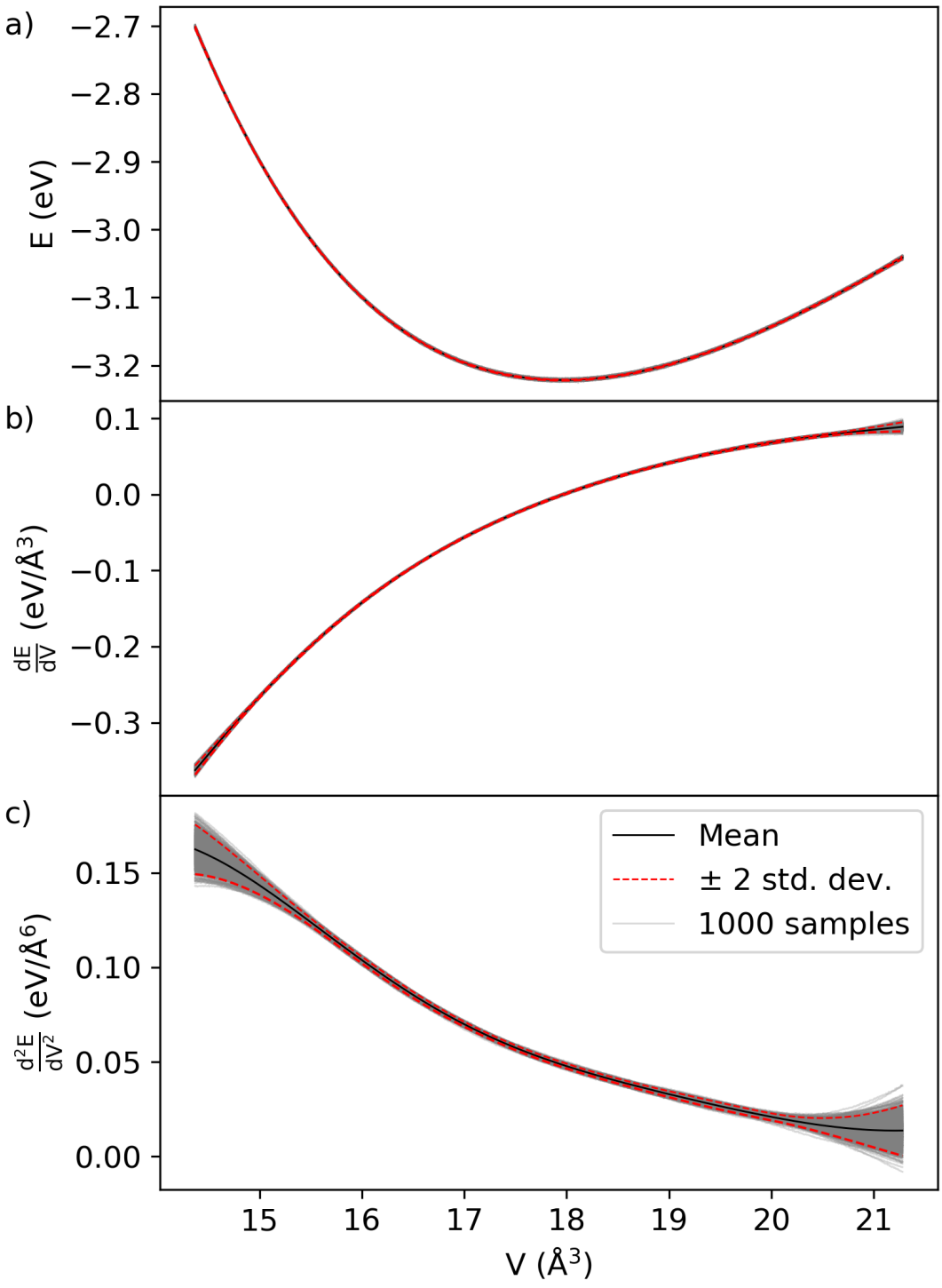

Figure S39: Au Gaussian process posterior

```
import numpy as np
from gp import *
```

```
import pickle

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')
v_test1 = np.linspace(V[0], V[-1], 1000)
samples = np.load('gp-samples-au.npy')
gpvmins, gpemins, gpbmods = get_phys(samples, v_test1)

with open('au-murn-hmc-samples.pickle', 'rb') as handle:
    murn_hmc_samples = pickle.load(handle)

with open('au-vinet-hmc-samples.pickle', 'rb') as handle:
    vinet_hmc_samples = pickle.load(handle)

murn_deltam = np.array([[17.9557, 18.0125],[-3.2232,-3.2189], [134.008,139.769]])

vinet_deltam = np.array([[17.9576, 17.9689],[-3.2219,-3.2212], [139.800,140.806]])

nonlin_dict = {'e0': [-3.2217, -3.2214, -3.2251, -3.22145, -3.22155],
               'v0': [17.9596, 17.96795, 17.9269, 17.9667, 17.9632],
               'b': [0.8816, 0.8705, 0.9985, 0.8717, 0.8758],
               'nl_model': ['SJ', 'AS', 'P3', 'B', 'PT']}

make_plot_model(gpbmods, [murn_hmc_samples, vinet_hmc_samples],
                [murn_deltam[2], vinet_deltam[2]], ['Murnaghan','Vinet'],
                'b', 'Bulk Modulus (GPa)','au', nonlin_dict)

make_plot_model(gpvmins, [murn_hmc_samples, vinet_hmc_samples],
                [murn_deltam[0], vinet_deltam[0]], ['Murnaghan','Vinet'],
                'v0', 'Volume ($\AA^3$)','au', nonlin_dict)

make_plot_model(gpemins, [murn_hmc_samples, vinet_hmc_samples],
                [murn_deltam[1], vinet_deltam[1]], ['Murnaghan','Vinet'],
                'e0', 'Energy (eV)','au', nonlin_dict)

print(f'''#+attr_org: :width 600
#+caption: Au model uncertainties bulk modulus
[[./au-gp-murnaghan-vinet-b.png]]''')

print(f'''#+attr_org: :width 600
#+caption: Au model uncertainties E0
```

```
[[./au-gp-murnaghan-vinet-e0.png]]''')

print(f'''#+attr_org: :width 600
#+caption: Au model uncertainties V0
[[./au-gp-murnaghan-vinet-v0.png]]''')
```

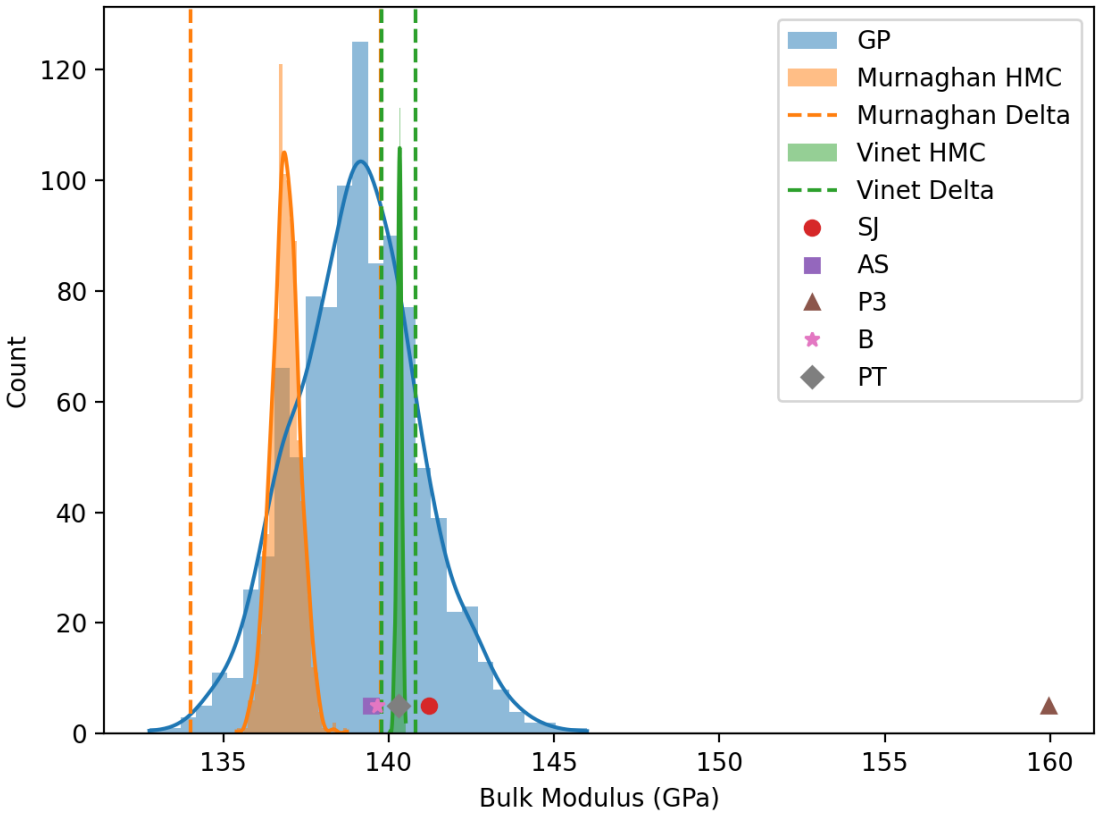

Figure S40: Au model uncertainties bulk modulus

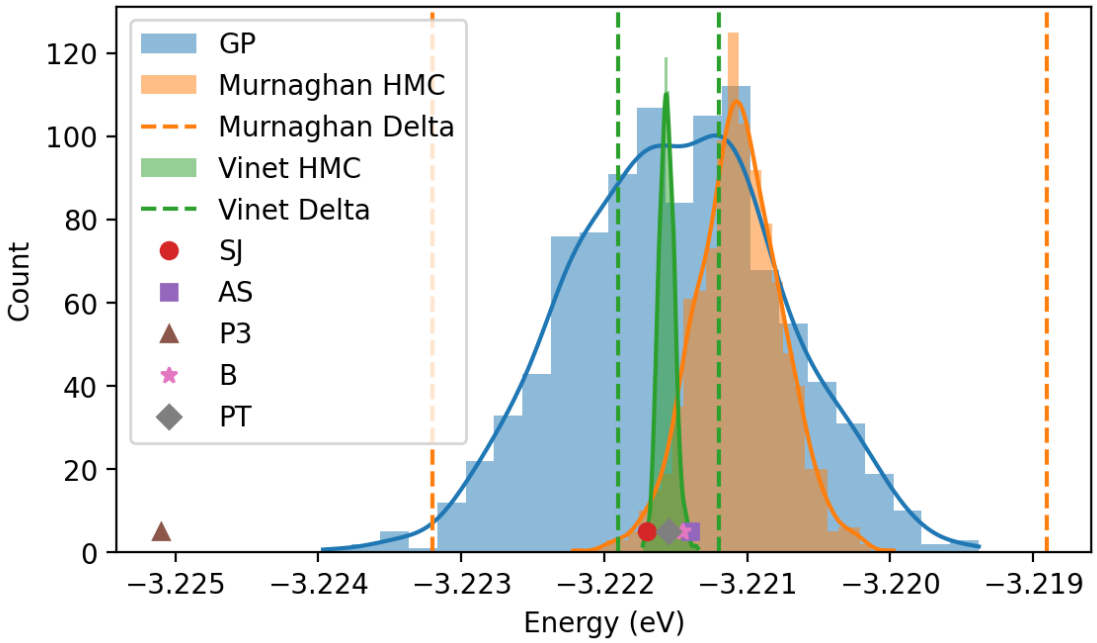

Figure S41: Au model uncertainties E0

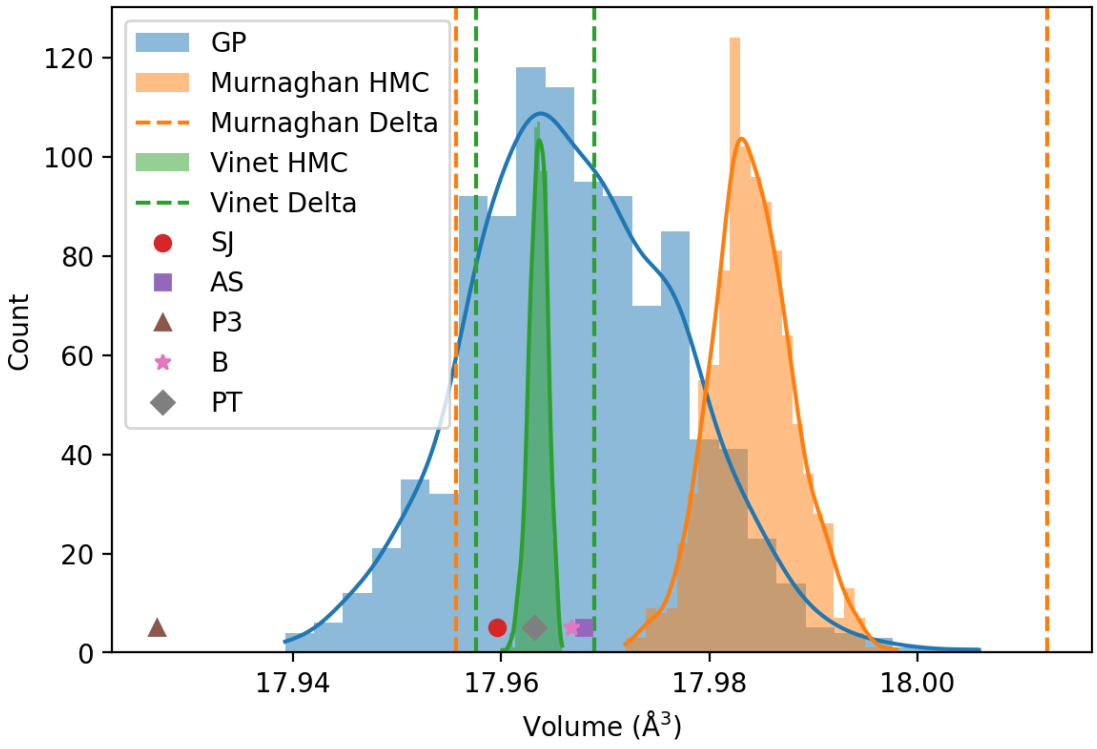

Figure S42: Au model uncertainties V0

5 Additional Plots

5.1 Stacked plots

Note that normalization for P3 in "calculate_samples" function has to be changed for Pd/Au.

For Pd:

```
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib as mpl
import pickle
import numpy as np
from matplotlib.ticker import MaxNLocator
import bayesreg
from bayesreg import calculate_samples
from scipy.interpolate import interp1d
from scipy.optimize import brentq

data_dict = {}
data_dict['category'] = ['S-J', 'A-S', 'P3', 'Murnaghan', 'Birch', 'P-T', 'Vinet']
data_dict['onese'] = [0.00068, 0.00108, 0.02503, 0.00758, 0.0023/2, 0.0127/2, 0.00067]
data_dict['mean'] = [15.3041, 15.3029, 15.3311, 15.3017, 15.3029, 15.3072, 15.3034]
dataset = pd.DataFrame(data_dict)

data_dicte = {}
data_dicte['category'] = ['S-J', 'A-S', 'P3', 'Murnaghan', 'Birch', 'P-T', 'Vinet' ]
data_dicte['onese'] = [0.00008, 0.00008, 0.00213, 0.00049, 0.0002/2, 0.0009/2, 0.00004]
data_dicte['mean'] = [-5.2146, -5.21445, -5.2164, -5.2140, -5.2145,-5.2149, -5.2146]
datasete = pd.DataFrame(data_dicte)

data_dictb = {}
data_dictb['category'] = ['S-J', 'A-S', 'P3', 'Murnaghan', 'Birch', 'P-T', 'Vinet' ]
data_dictb['onese'] = [0.069, 0.13, 4.294, 0.752, 0.262/2, 1.722/2, 0.075]
data_dictb['mean'] = [168.8, 167.6, 179.6, 164.7, 167.6, 170.7, 168.4]
datasetb = pd.DataFrame(data_dictb)

mpl.rcParams['mathtext.default'] = 'regular'
plt.clf()
plt.rcParams.update({'font.size': 13})
fig, ax = plt.subplots(1,3,sharex=False, sharey='row', figsize=(10,5.0))
```

```
34
35  def beos_gp(sample):
36      '''
37      we are going to assume that the observations, deriv, and 2nd deriv
38      are sampled at the same x points
39      also going to assume that the x points are in order from smallest to largest.
40      '''
41      func = interp1d(v_test1, sample[:1000], fill_value='extrapolate')
42      deriv = interp1d(v_test1, sample[1000:2000],fill_value='extrapolate')
43      secderiv = interp1d(v_test1, sample[2000:],fill_value='extrapolate')
44      vmin = brentq(deriv, 12.5, 17.5)
45      emin = func(vmin)
46      secderivmin = secderiv(vmin)
47      return vmin, emin, vmin*secderivmin
48
49  def get_phys(samples):
50      samplesd = {}
51      vmins = []
52      emins = []
53      bmods = []
54
55      numsample = len(samples)
56
57      if numsample > 1000:
58          randinds = np.random.choice(np.arange(numsample), 1000, replace=False)
59
60      else:
61          randinds = np.arange(samples.shape[0])
62
63      for i in randinds:
64          a, b, c = beos_gp(samples[i])
65          vmins += [a]
66          emins += [b]
67          bmods += [c]
68      samplesd['v0'] = np.array(vmins)
69      samplesd['e0'] = np.array(emins)
70      samplesd['b'] = np.array(bmods)
71      return samplesd
72
73
74  def plot_dist(samples, ymin, color='gray', alpha=0.85):
```

```
    myhist = np.histogram(samples['v0'])
    ax[0].bar(myhist[1][:-1], myhist[0]/500, width=(myhist[1][1] - myhist[1][0]), bottom=ymin,
              color=color, label='HMC', alpha=alpha)
    myhist = np.histogram(samples['e0'])
    ax[1].bar(myhist[1][:-1], myhist[0]/500, width=(myhist[1][1] - myhist[1][0]), bottom=ymin,
              color=color, alpha=alpha)
    myhist = np.histogram(samples['b'])
    ax[2].bar(myhist[1][:-1]*160.2, myhist[0]/500, width=(myhist[1][1] - myhist[1][0])*160.2,
              bottom=ymin, color=color, alpha=alpha)

V = np.load('v-pd.npy')
v_test1 = np.linspace(V[0], V[-1], 1000)
samples = np.load('gp-samples-pd.npy')
gpsamples = get_phys(samples)
for i in range(len(dataset),0,-1):
    plot_dist(gpsamples, i, 'tab:blue', 0.7)

for se,middle,y in zip(dataset['onese'],
                               dataset['mean'],
                               range(len(dataset),0,-1)):
    ax[0].plot((middle-2*se,middle,middle+2*se),(y,y,y),'ro-',color='orange',
               label='Delta 95% Confidence')


for se,middle,y in zip(datasete['onese'],
                               datasete['mean'],
                               range(len(datasete),0,-1)):
    ax[1].plot((middle-2*se,middle,middle+2*se),(y,y,y),'ro-',color='orange')

for se,middle,y in zip(datasetb['onese'],
                               datasetb['mean'],
                               range(len(datasetb),0,-1)):
    ax[2].plot((middle-2*se,middle,middle+2*se),(y,y,y),'ro-',color='orange')



for modeln, i in zip(['sj', 'as', 'p3', 'murn', 'birch', 'pt', 'vinet'], range(len(dataset),0,-1)):
    filen = f'pd-{modeln}-hmc-samples.pickle'
    with open(filen, 'rb') as handle:
        hmc_samples = pickle.load(handle)
    if modeln in ['sj', 'p3', 'as']:
```

```
        hmc_samples = calculate_samples(hmc_samples, modeln)
    plot_dist(hmc_samples, i)



ax[0].set_yticks(range(len(dataset),0,-1))
ax[0].set_yticklabels(list(dataset['category']))

ax[0].set_xlabel('$V_0$ ($\AA^3$)')
ax[1].set_xlabel('$E_0$ (eV/atom)')
ax[2].set_xlabel('B (GPa)')

ax[0].set_ylim((0.8, 7.8))

ax[1].xaxis.set_major_locator(MaxNLocator(nbins=3))

handles, _ = ax[0].get_legend_handles_labels()
legend=fig.legend([handles[0],handles[7],handles[-1]], ['Delta 95% Confidence', 'GP', 'HMC'],
                  loc='upper right',bbox_to_anchor=(0.71, 1.16))

plt.tight_layout()
plt.savefig('test-ci-delta-pd-all.png', dpi=200,bbox_inches='tight')

print(f'''#+attr_org: :width 600
#+caption: Pd all models comparison
[[./test-ci-delta-pd-all.png]]''')
```

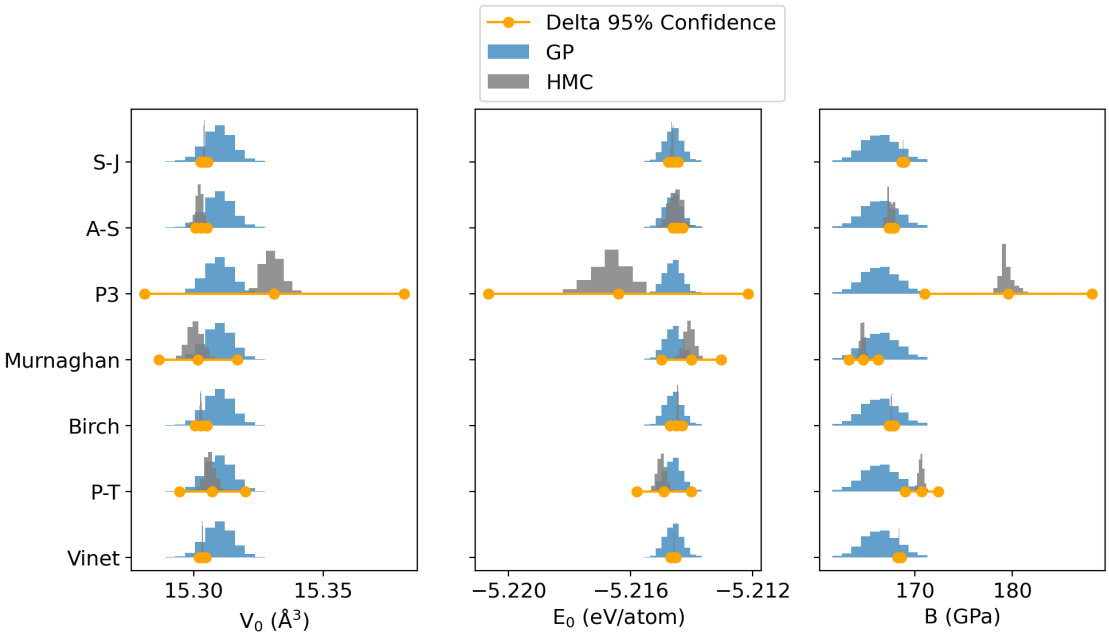

Figure S43: Pd all models comparison

For Au:

```
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib as mpl
import pickle
import numpy as np
from matplotlib.ticker import MaxNLocator
import bayesreg
from bayesreg import calculate_samples
from scipy.interpolate import interp1d
from scipy.optimize import brentq

data_dict = {}
data_dict['category'] = ['S-J', 'A-S', 'P3', 'Murnaghan', 'Birch', 'P-T', 'Vinet']
data_dict['onese'] = [0.00152, 0.00108, 0.05889, 0.01379, 0.00445, 0.01212, 0.00274]
data_dict['mean'] = [17.9596, 17.9680, 17.9269, 17.9841, 17.9667, 17.9503, 17.9632]
dataset = pd.DataFrame(data_dict)

data_dicte = {}
data_dicte['category'] = ['S-J', 'A-S', 'P3', 'Murnaghan', 'Birch', 'P-T', 'Vinet' ]
data_dicte['onese'] = [0.00017, 0.00034, 0.004, 0.00103, 0.0003, 0.00077, 0.00019]
data_dicte['mean'] = [-3.2217, -3.2214, -3.2251, -3.2211, -3.2214,-3.2222, -3.2215]
```

```
datasete = pd.DataFrame(data_dicte)

data_dictb = {}
data_dictb['category'] = ['S-J', 'A-S', 'P3', 'Murnaghan', 'Birch', 'P-T', 'Vinet' ]
data_dictb['onese'] = [0.13, 0.44, 4.923, 1.399, 0.369, 0.878, 0.244]
data_dictb['mean'] = [141.2, 139.5, 160.0, 136.9, 139.6, 144., 140.3]
datasetb = pd.DataFrame(data_dictb)

mpl.rcParams['mathtext.default'] = 'regular'
plt.clf()
plt.rcParams.update({'font.size': 13})
fig, ax = plt.subplots(1,3,sharex=False, sharey='row', figsize=(10,5.0))

def beos_gp(sample, v_test1):
    '''
    we are going to assume that the observations, deriv, and 2nd deriv are sampled at the same x points
    also going to assume that the x points are in order from smallest to largest.
    '''
    func = interp1d(v_test1, sample[:1000], fill_value='extrapolate')
    deriv = interp1d(v_test1, sample[1000:2000],fill_value='extrapolate')
    secderiv = interp1d(v_test1, sample[2000:],fill_value='extrapolate')
    vmin = brentq(deriv, v_test1[0], v_test1[-1])
    emin = func(vmin)
    secderivmin = secderiv(vmin)
    return vmin, emin, vmin*secderivmin

def get_phys(samples, v_test1):
    samplesd = {}
    vmins = []
    emins = []
    bmods = []

    numsample = len(samples)

    if numsample > 1000:
        randinds = np.random.choice(np.arange(numsample), 1000, replace=False)

    else:
        randinds = np.arange(samples.shape[0])

    for i in randinds:
```

```
        a, b, c = beos_gp(samples[i], v_test1)
        vmins += [a]
        emins += [b]
        bmods += [c]
    samplesd['v0'] = np.array(vmins)
    samplesd['e0'] = np.array(emins)
    samplesd['b'] = np.array(bmods)
    return samplesd


def plot_dist(samples, ymin, color='gray', alpha=0.85):
    myhist = np.histogram(samples['v0'])
    ax[0].bar(myhist[1][:-1], myhist[0]/500, width=(myhist[1][1] - myhist[1][0]), bottom=ymin,
              color=color, label='HMC', alpha=alpha)
    myhist = np.histogram(samples['e0'])
    ax[1].bar(myhist[1][:-1], myhist[0]/500, width=(myhist[1][1] - myhist[1][0]), bottom=ymin,
              color=color, alpha=alpha)
    myhist = np.histogram(samples['b'])
    ax[2].bar(myhist[1][:-1]*160.2, myhist[0]/500, width=(myhist[1][1] - myhist[1][0])*160.2,
              bottom=ymin, color=color, alpha=alpha)

V = np.load('v-au.npy')
v_test1 = np.linspace(V[0], V[-1], 1000)
samples = np.load('gp-samples-au.npy')
gpsamples = get_phys(samples, v_test1)
for i in range(len(dataset),0,-1):
    plot_dist(gpsamples, i, 'tab:blue', 0.7)

for se,middle,y in zip(dataset['onese'],
                       dataset['mean'],
                       range(len(dataset),0,-1)):
    ax[0].plot((middle-2*se,middle,middle+2*se),(y,y,y),'ro-',color='orange',
               label='Delta 95% Confidence')


for se,middle,y in zip(datasete['onese'],
                       datasete['mean'],
                       range(len(datasete),0,-1)):
    ax[1].plot((middle-2*se,middle,middle+2*se),(y,y,y),'ro-',color='orange')

for se,middle,y in zip(datasetb['onese'],
```

```
                                datasetb['mean'],
                                range(len(datasetb),0,-1)):
    ax[2].plot((middle-2*se,middle,middle+2*se),(y,y,y),'ro-',color='orange')



for modeln, i in zip(['sj', 'as', 'p3', 'murn', 'birch', 'pt', 'vinet'], range(len(dataset),0,-1)):
    filen = f'au-{modeln}-hmc-samples.pickle'
    with open(filen, 'rb') as handle:
        hmc_samples = pickle.load(handle)
    if modeln in ['sj', 'p3', 'as']:
        hmc_samples = calculate_samples(hmc_samples, modeln)
    plot_dist(hmc_samples, i)



ax[0].set_yticks(range(len(dataset),0,-1))
ax[0].set_yticklabels(list(dataset['category']))

ax[0].set_xlabel('$V_0$ ($\AA^3$)')
ax[1].set_xlabel('$E_0$ (eV/atom)')
ax[2].set_xlabel('B (GPa)')

ax[0].set_ylim((0.8, 7.8))

ax[1].xaxis.set_major_locator(MaxNLocator(nbins=3))

handles, _ = ax[0].get_legend_handles_labels()
legend=fig.legend([handles[0],handles[7],handles[-1]], ['Delta 95% Confidence', 'GP', 'HMC'],
                  loc='upper right',bbox_to_anchor=(0.71, 1.16))

plt.tight_layout()
plt.savefig('test-ci-delta-au-all.png', dpi=200,bbox_inches='tight')

print(f'''#+attr_org: :width 600
#+caption: Au all models comparison
[[./test-ci-delta-au-all.png]]''')
```

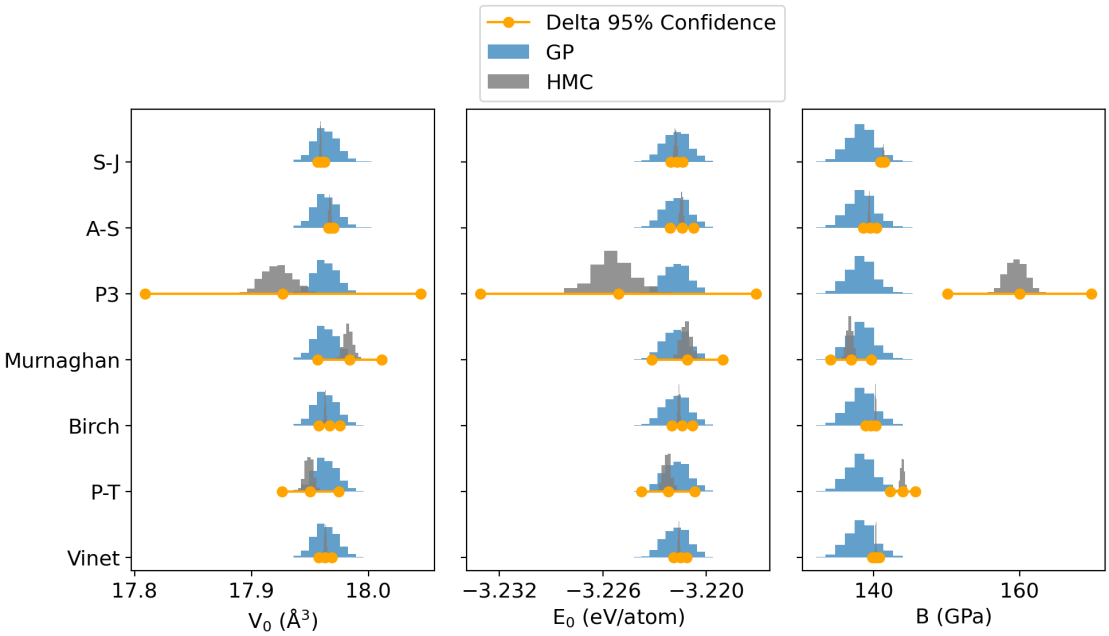

Figure S44: Au all models comparison

5.2 GP kernel parameters

For Au, we used the optimal lengthscale = 4.2634, outputscale = 0.8007. Suppose we did not find the optimal parameters and used lengthscale = 1.0, outputscale = 1.0. If we use different GP hyperparameters, then we will get different results.

```
import numpy as np
from gp import *

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')

mu1 = np.load('gp-mu-au-param2.npy')
var1 = np.load('gp-var-au-param2.npy')
samples = np.load('gp-samples-au-param2.npy')
v_test1 = np.linspace(V[0], V[-1], 1000)

make_gp_posterior(mu1, var1, samples, v_test1, 'au-param2')

print(f'''#+attr_org: :width 600
```

```
15  #+caption: Au Gaussian process posterior
16  [[./gp-posterior-au-param2.png]]''')
```

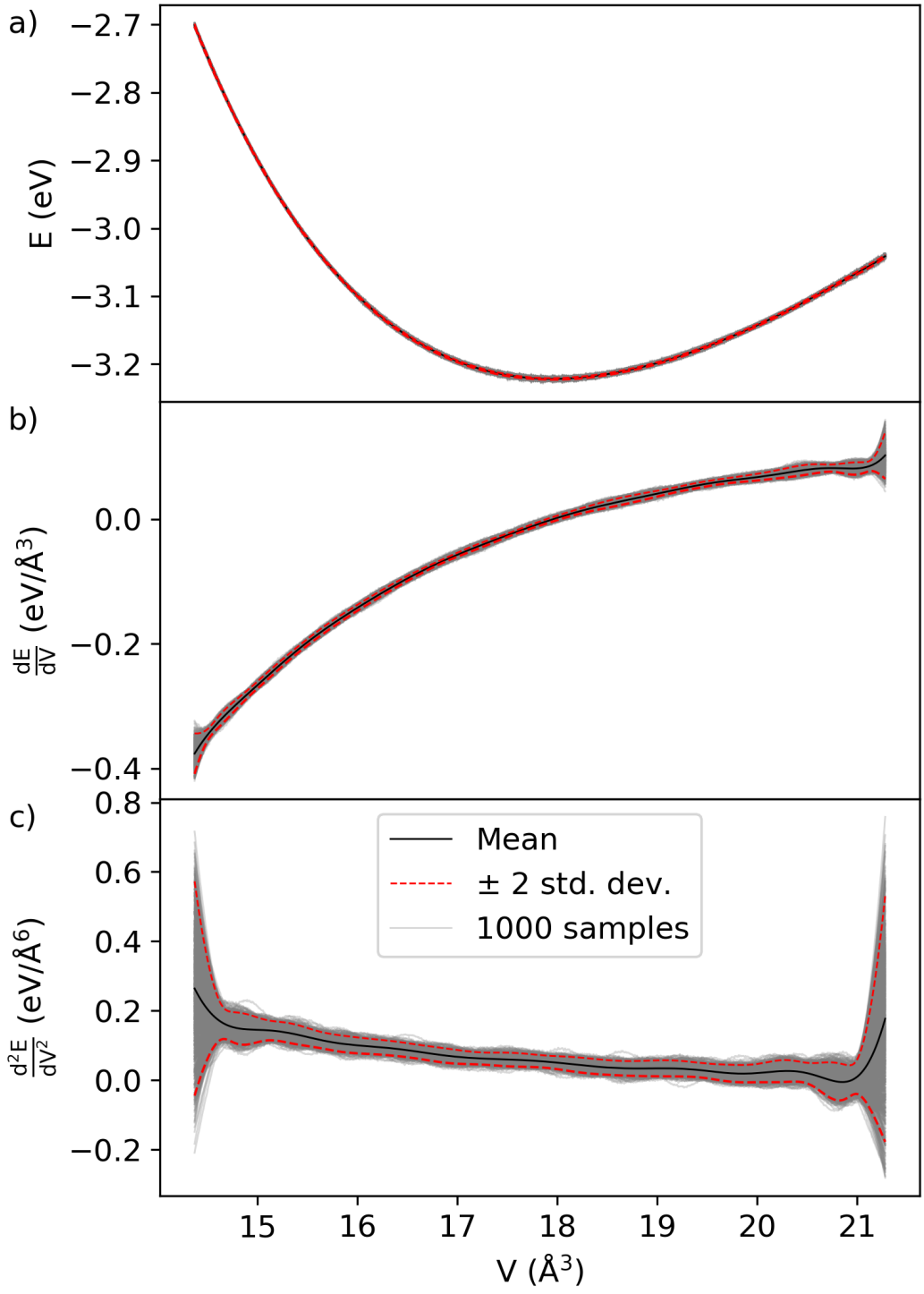

Figure S45: Au Gaussian process posterior

Below is using lengthscale = 10.0, outputscale = 1.0. This results in a tighter distribution compared with using the optimal parameters.

```
import numpy as np
from gp import *

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')

mu1 = np.load('gp-mu-au-param3.npy')
var1 = np.load('gp-var-au-param3.npy')
samples = np.load('gp-samples-au-param3.npy')
v_test1 = np.linspace(V[0], V[-1], 1000)

make_gp_posterior(mu1, var1, samples, v_test1, 'au-param3')

print(f'''#+attr_org: :width 600
#+caption: Au Gaussian process posterior
[[./gp-posterior-au-param3.png]]''')
```

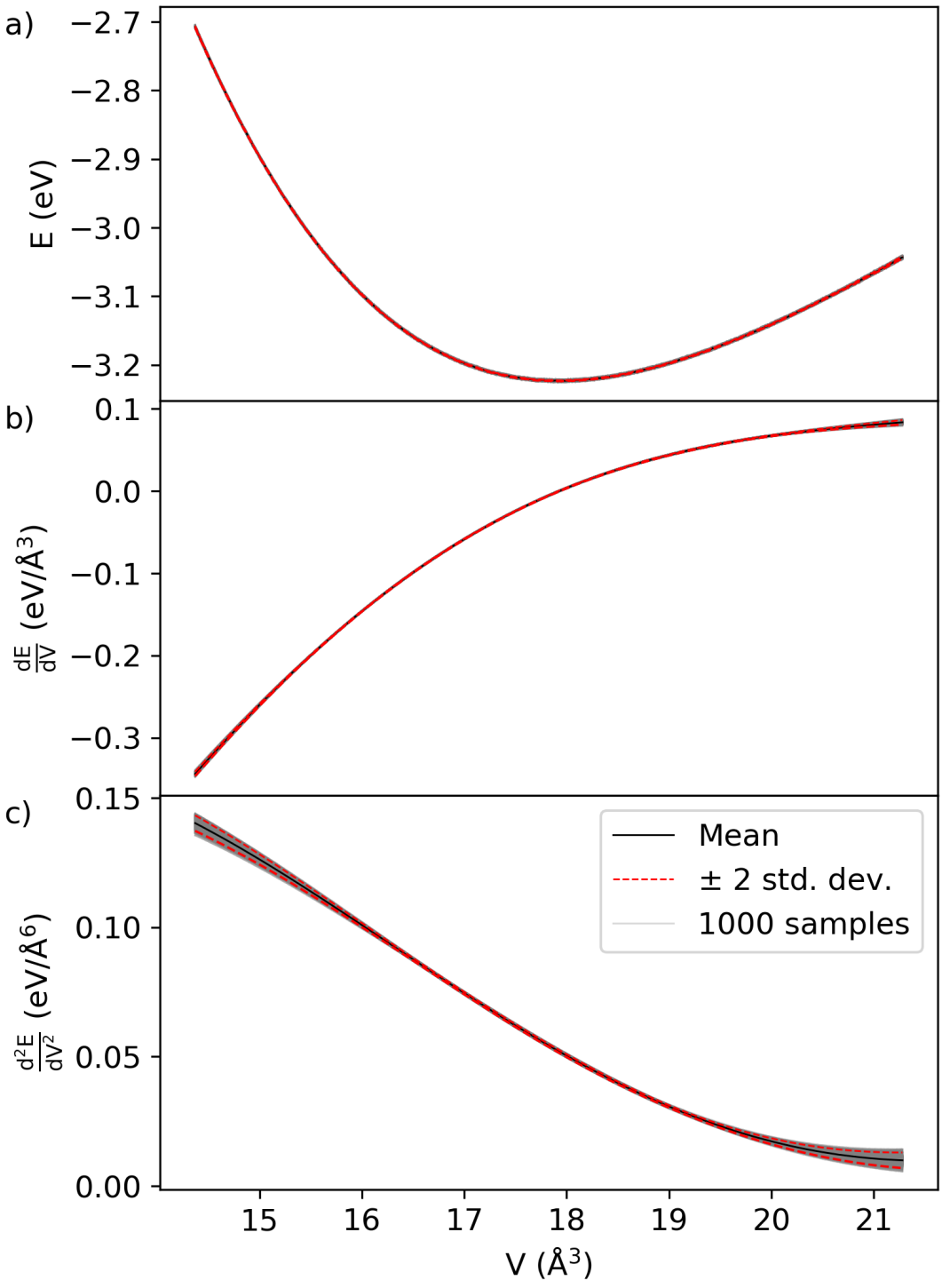

Figure S46: Au Gaussian process posterior

Below is using lengthscale = 5.0, outputscale = 10.0.

```
import numpy as np
from gp import *

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')

mu1 = np.load('gp-mu-au-param4.npy')
var1 = np.load('gp-var-au-param4.npy')
samples = np.load('gp-samples-au-param4.npy')
v_test1 = np.linspace(V[0], V[-1], 1000)

make_gp_posterior(mu1, var1, samples, v_test1, 'au-param4')

print(f'''#+attr_org: :width 600
#+caption: Au Gaussian process posterior
[[./gp-posterior-au-param4.png]]''')
```

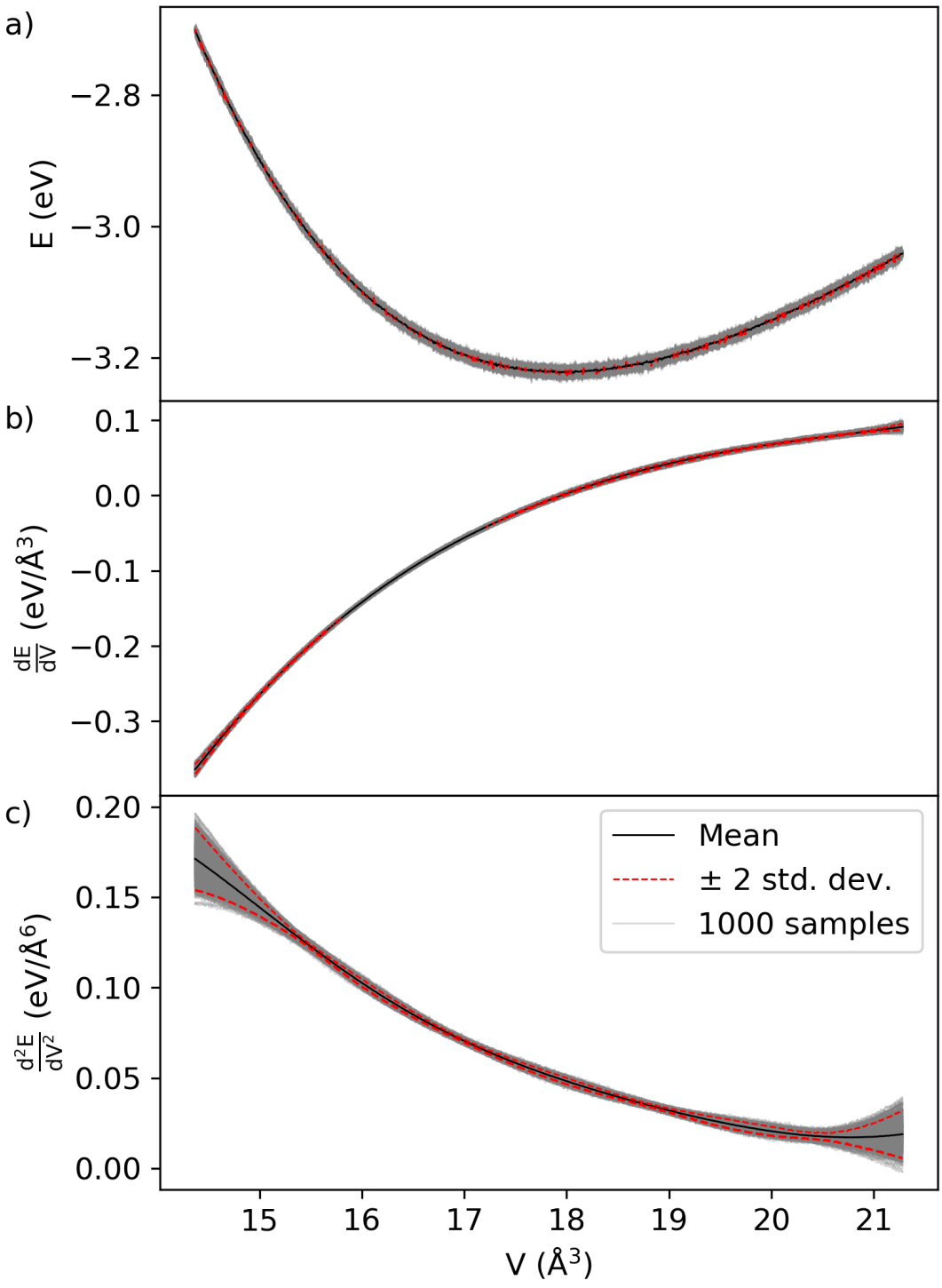

Figure S47: Au Gaussian process posterior

Lengthscale = 5.0, outputscale = 0.01. This one seems to overfit compared with the

optimal hyperparameters.

```
import numpy as np
from gp import *

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')

mu1 = np.load('gp-mu-au-param5.npy')
var1 = np.load('gp-var-au-param5.npy')
samples = np.load('gp-samples-au-param5.npy')
v_test1 = np.linspace(V[0], V[-1], 1000)

make_gp_posterior(mu1, var1, samples, v_test1, 'au-param5')

print(f'''#+attr_org: :width 600
#+caption: Au Gaussian process posterior
[[./gp-posterior-au-param5.png]]''')
```

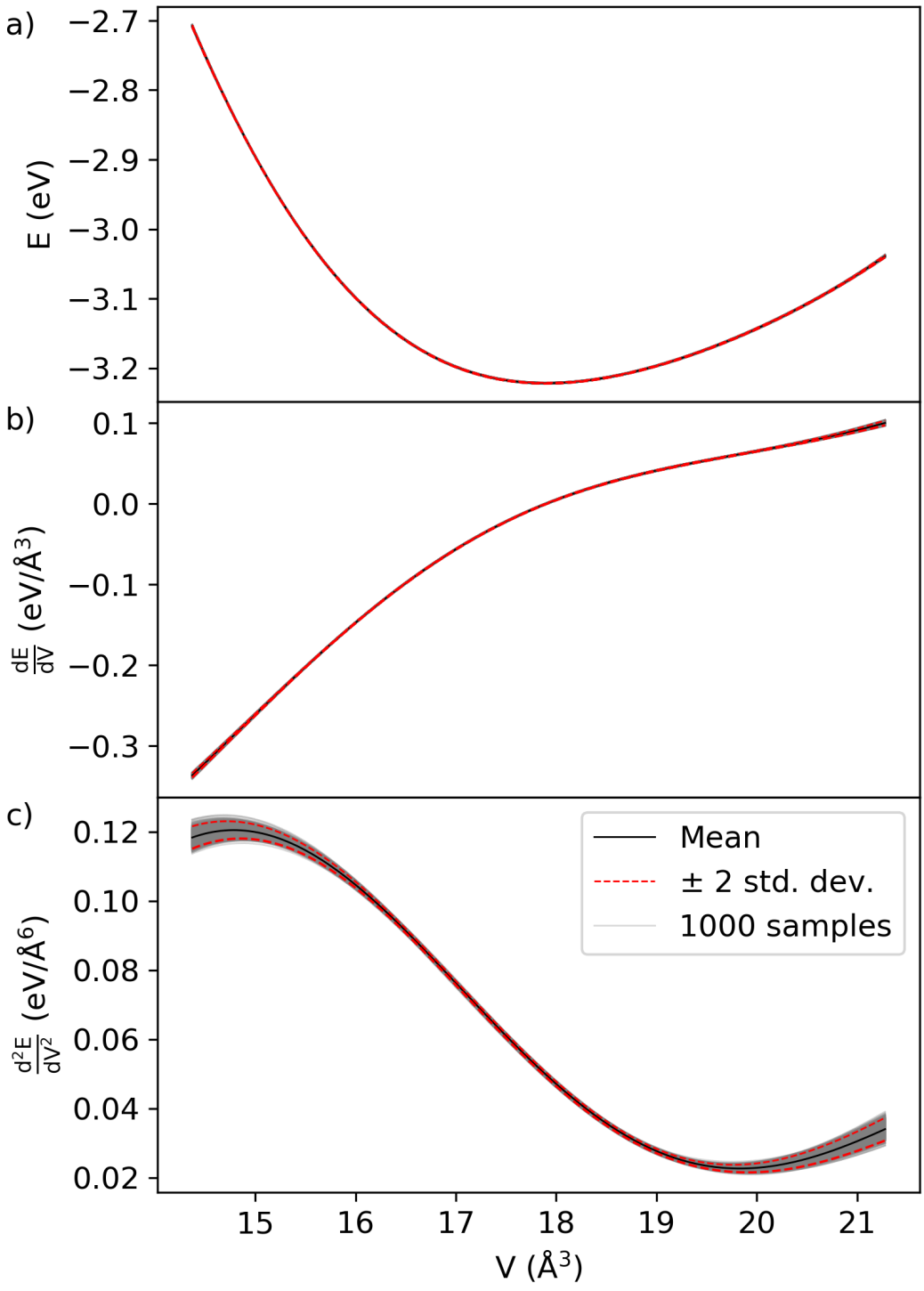

Figure S48: Au Gaussian process posterior

Below is Sanity check of linear kernel instead of RBF kernel. We do not expect the linear

kernel to be able to fit the function. Similarly, we do not think periodic kernel would work well since the function is not periodic. We also need a kernel that is twice differentiable (for example, that rules out some of the Matern kernels).

```
import numpy as np
from gp import *

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')

mu1 = np.load('gp-mu-au-param6.npy')
var1 = np.load('gp-var-au-param6.npy')
samples = np.load('gp-samples-au-param6.npy')
v_test1 = np.linspace(V[0], V[-1], 1000)

make_gp_posterior(mu1, var1, samples, v_test1, 'au-param6')

print(f'''#+attr_org: :width 600
#+caption: Au Gaussian process posterior
[[./gp-posterior-au-param6.png]]''')
```

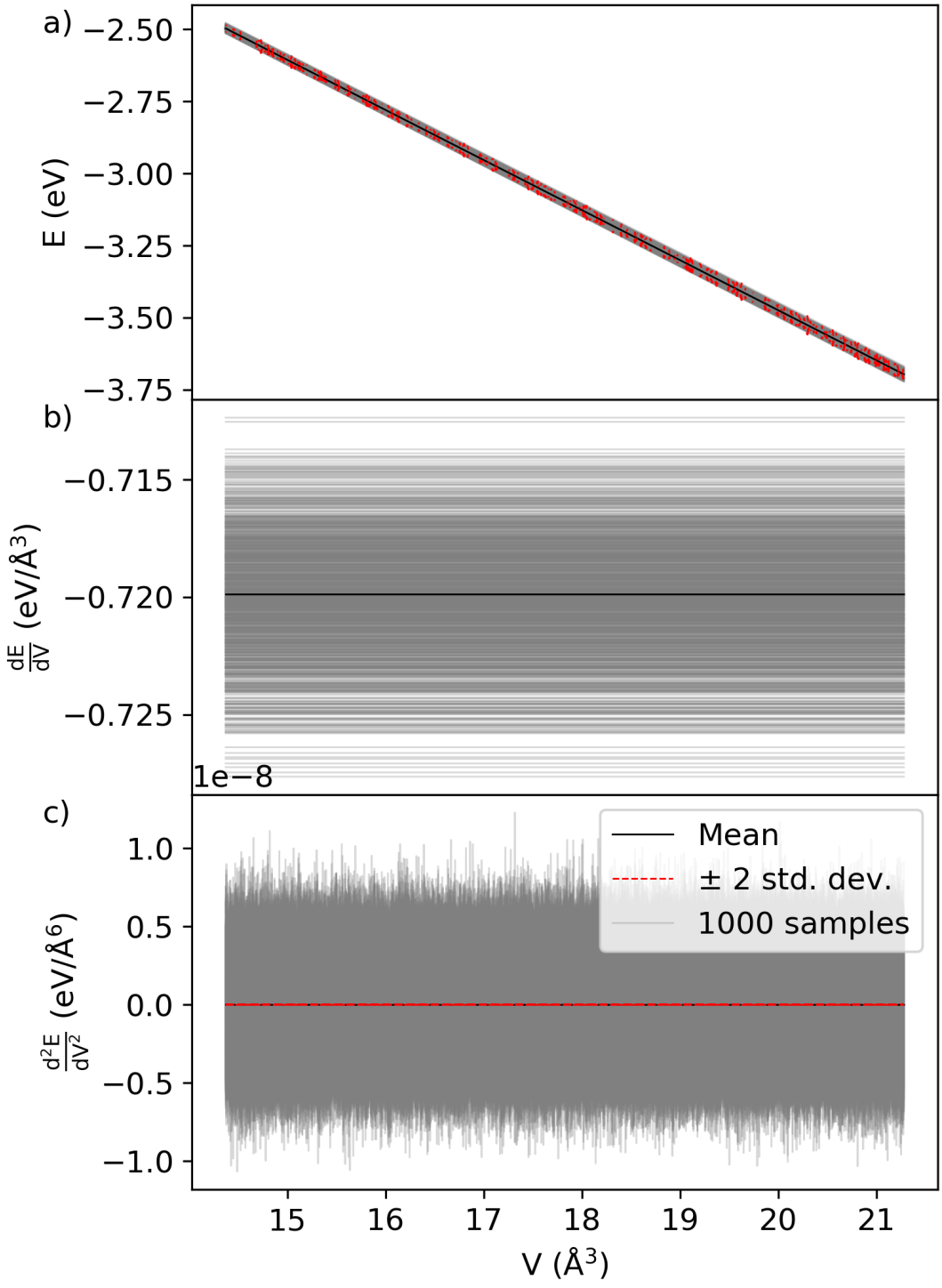

Figure S49: Au Gaussian process posterior

5.3 GP extrapolation plots

Here we check if the uncertainty from GP is able to capture the data outside of the training data range. We find that this is not the case. It is likely that the uncertainty estimate is biased by the RBF kernel which is very different from true EOS and not correct outside of the training data range.

```
import numpy as np
from gp import *
from ase.db import connect

#get Au data
db = connect('data.db')

Vau, Qe = [], []
for d in db.select(['bulk=fcc', 'factor']):
    Vau += [d.volume/d.natoms]
    Qe += [d.energy/d.natoms]

Vauall = np.array(Vau)
nrgauall = np.array(Qe)

sortind = np.argsort(Vauall)
Vauall = Vauall[sortind]
nrgauall = nrgauall[sortind]

sel = Vauall[np.argmin(nrgauall)]

ind = (Vauall > sel - 15) & (Vauall < sel + 15)


V = Vauall[ind]
nrg = nrgauall[ind]


mu1 = np.load('gp-mu-au-extrap.npy')
var1 = np.load('gp-var-au-extrap.npy')
samples = np.load('gp-samples-au-extrap.npy')
v_test1 = np.linspace(11, 31, 1000)
```

```
make_gp_posterior_extrap(mu1, var1, samples, v_test1, V, nrg, 'au-extrap')

print(f'''#+attr_org: :width 600
#+caption: Au Gaussian process posterior check extrapolation
[[./gp-posterior-au-extrap.png]]''')
```

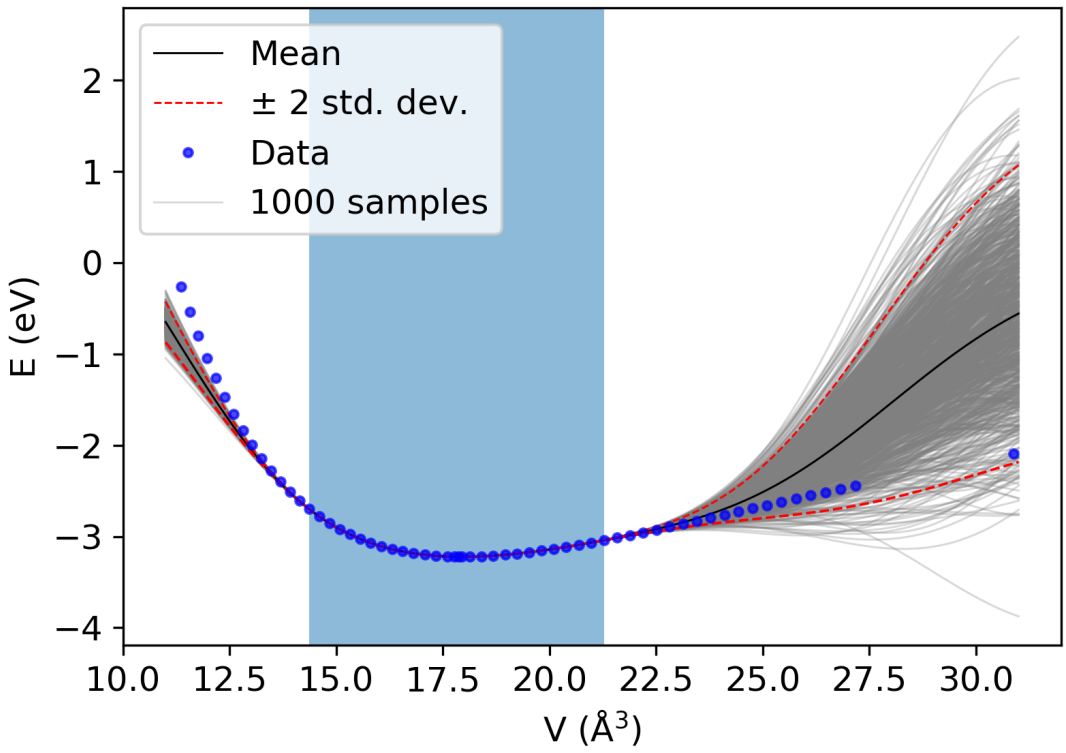

Figure S50: Au Gaussian process posterior check extrapolation

```
import numpy as np
from gp import *
from ase.db import connect

#get Pd data
db = connect('master.db')

E = [d.energy for d in db.select(['bulk', 'strain=xyz'])]
V = [d.volume for d in db.select(['bulk', 'strain=xyz'])]

sel = V[E.index(min(E))]
```

```
12
13  ind = (np.array(V) > sel - 15) & (np.array(V) < sel + 15)
14
15  V = np.array(V)[ind]
16  nrg = np.array(E)[ind]
17
18  mu1 = np.load('gp-mu-pd-extrap.npy')
19  var1 = np.load('gp-var-pd-extrap.npy')
20  samples = np.load('gp-samples-pd-extrap.npy')
21  v_test1 = np.linspace(9, 29, 1000)
22
23  make_gp_posterior(mu1, var1, samples, v_test1, 'pd-extrap2')
24
25  make_gp_posterior_extrap(mu1, var1, samples, v_test1, V, nrg, 'pd-extrap')
26
27  print(f'''#+attr_org: :width 600
28  #+caption: Pd Gaussian process posterior check extrapolation
29  [[./gp-posterior-pd-extrap.png]]''')
```

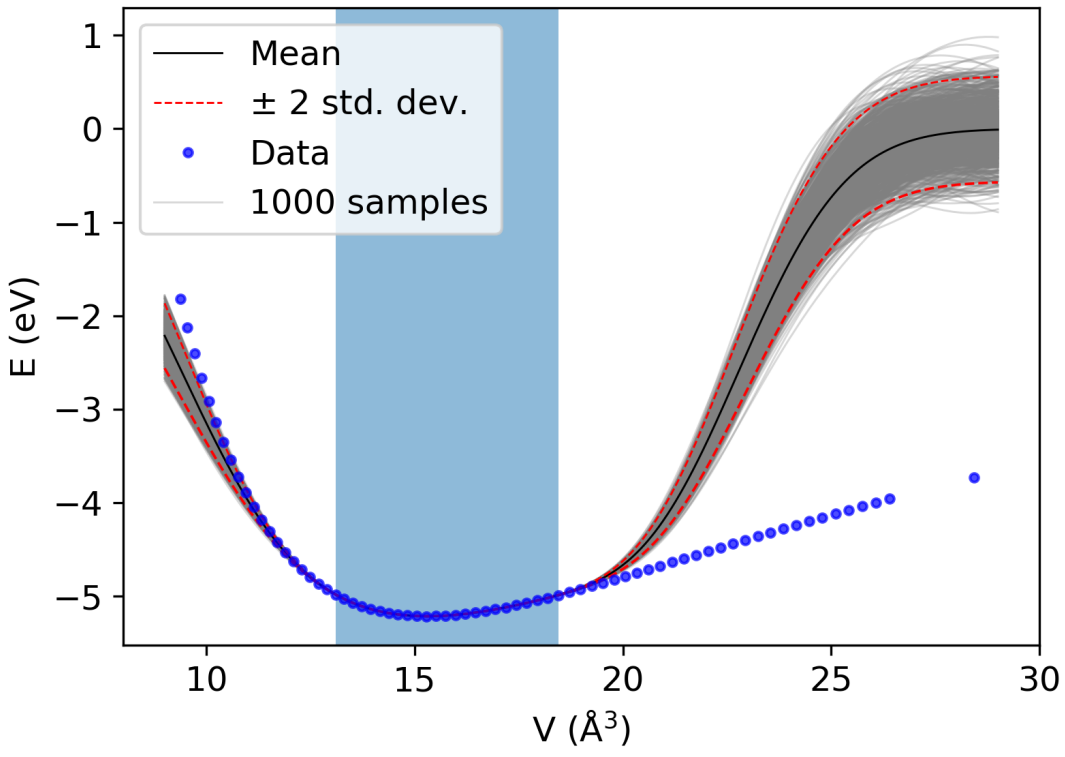

Figure S51: Pd Gaussian process posterior check extrapolation

5.4 Sparser data

Using Au Murnaghan as an example of using 1/5th of the data in the same range. For all methods (Delta, Bayesian Regression SVI/HMC, GP) the uncertainty becomes larger because the data is sparser. This is good behavior, and shows that the uncertainty includes uncertainty about the training data which is part of epistemic uncertainty.

```
from autograd import hessian
import autograd.numpy as np
from ase.eos import EquationOfState
from scipy.stats.distributions import t
import deltam
from deltam import *

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')
print(V.shape)
V = V[::5]
print(V.shape)
nrg = nrg[::5]
get_delta_u('murnaghan', murnaghan, V, nrg)
```

```
(29,)
(6,)
V0: 17.9856
E0: -3.2215
B: 138.3
RMSE: 0.000653
MAE: 0.000585


Standard Error Confidences:
---------------------------
V0: 0.00866
```

```
E0: 0.00077

B: 1.294


95% Confidence Intervals:

-------------------------

V0: [17.9484, 18.0229]

E0: [-3.2248, -3.2182]

B: [132.719, 143.852]
```

```
import numpy as np
import torch
import bayesreg
from bayesreg import *

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')
V = V[::5]
nrg = nrg[::5]
V = torch.Tensor(V)
nrg = torch.Tensor(nrg)

def model(V, nrg, func):
    e0 = pyro.sample("e0", dist.Normal(-3., 1.))
    b = pyro.sample("b", dist.Normal(0.8, 0.2))
    bp = pyro.sample("bp", dist.Normal(5., 1.))
    v0 = pyro.sample("v0", dist.Normal(18., 7.5))
    sigma = pyro.sample("sigma", dist.Uniform(0., 1.))
    mean = func(V, e0, b, bp, v0)
    with pyro.plate("data", len(V)):
        pyro.sample("obs", dist.Normal(mean, sigma), obs=nrg)

optim_vi(model, V, nrg, murnaghan, 30000, 'au-murn-sparse')

print(f'''#+attr_org: :width 600
#+caption: ELBO Au murnaghan
[[./elbo-au-murn-sparse.png]]''')
```

(29,) (6,)

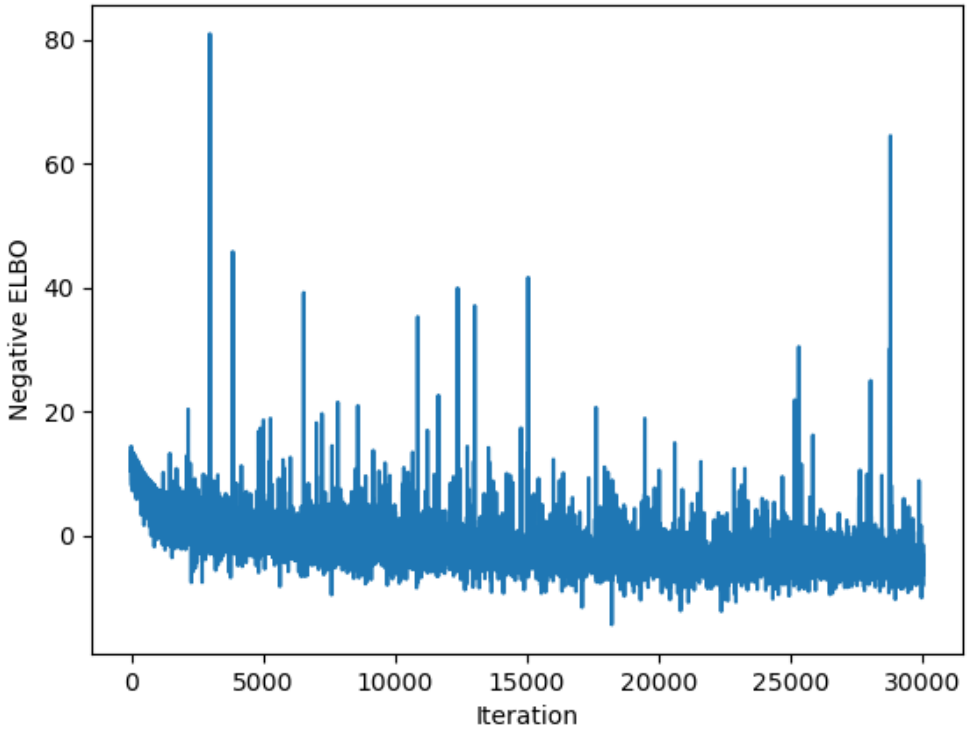

Figure S52: ELBO Au murnaghan

```
from pyro.infer import MCMC, NUTS
import pyro.distributions as dist
import bayesreg
from bayesreg import *
import numpy as np
import torch
import pickle

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')
V = V[::5]
nrg = nrg[::5]
V = torch.Tensor(V)
nrg = torch.Tensor(nrg)
```

```

nuts_kernel = NUTS(model_others)

mcmc = MCMC(nuts_kernel, num_samples=1000, warmup_steps=200)
mcmc.run(V, nrg, murnaghan)

hmc_samples = {k: v.detach().cpu().numpy() for k, v in mcmc.get_samples().items()}

with open('au-murn-sparse-hmc-samples.pickle', 'wb') as handle:
    pickle.dump(hmc_samples, handle, protocol=pickle.HIGHEST_PROTOCOL)
```

```
import pickle
import numpy as np
import bayesreg
from bayesreg import make_plot

with open('au-murn-sparse-svi-mvn-samples.pickle', 'rb') as handle:
    svi_mvn_samples = pickle.load(handle)
with open('au-murn-sparse-hmc-samples.pickle', 'rb') as handle:
    hmc_samples = pickle.load(handle)

deltam = np.array([[17.9484, 18.0229],[-3.2248,-3.2182], [132.719,143.852]])

make_plot(svi_mvn_samples, hmc_samples, deltam,
          'au-murn-svi-hmc-delta-sparse', nbinsi = [4,4,6,3])

print(f'''#+attr_org: :width 600
#+caption: Au murnaghan SVI HMC Delta Sparse
[[./au-murn-svi-hmc-delta-sparse.png]]''')
```

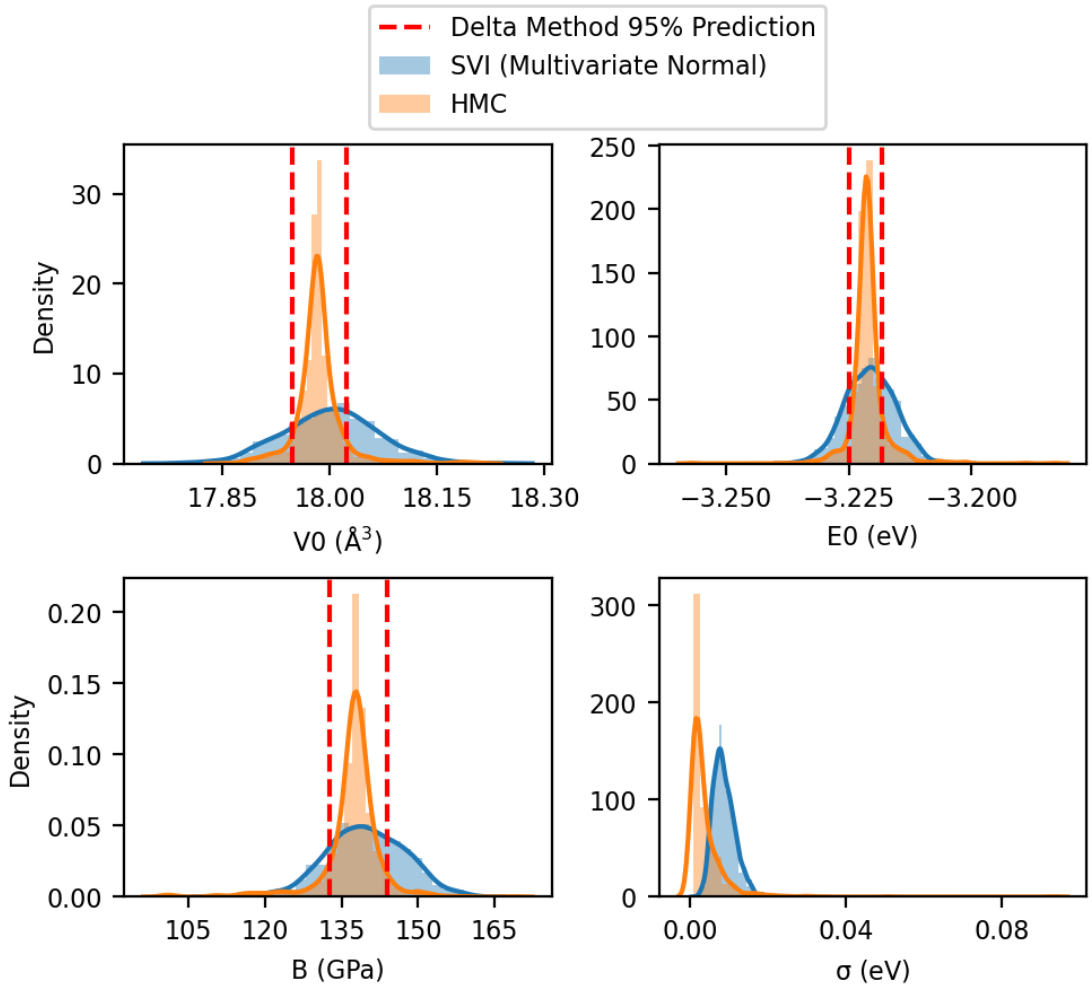

Figure S53: Au murnaghan SVI HMC Delta Sparse

```
import numpy as np
from gp import *

V = np.load('v-au.npy')
nrg = np.load('nrg-au.npy')

mu1 = np.load('gp-mu-au-sparse.npy')
var1 = np.load('gp-var-au-sparse.npy')
samples = np.load('gp-samples-au-sparse.npy')
v_test1 = np.linspace(V[0], 21., 1000)

make_gp_posterior(mu1, var1, samples, v_test1, 'au-sparse')

```

```
print(f'''#+attr_org: :width 600
#+caption: Au Gaussian process posterior Sparse
[[./gp-posterior-au-sparse.png]]''')
```

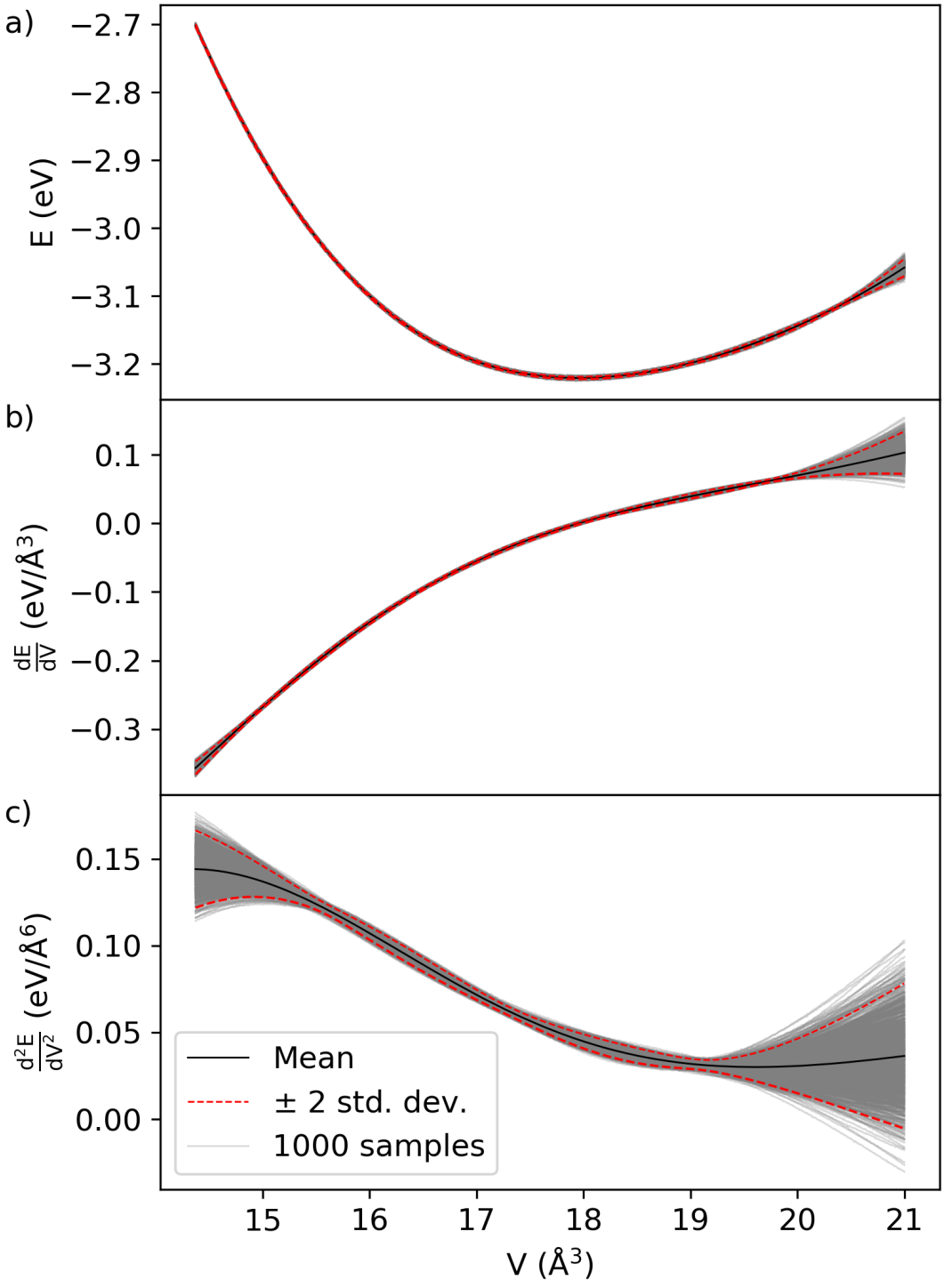

Figure S54: Au Gaussian process posterior Sparse