楽しい微分可能プログラミング

JAX/NumPyro で遊ぼう v0.5.1

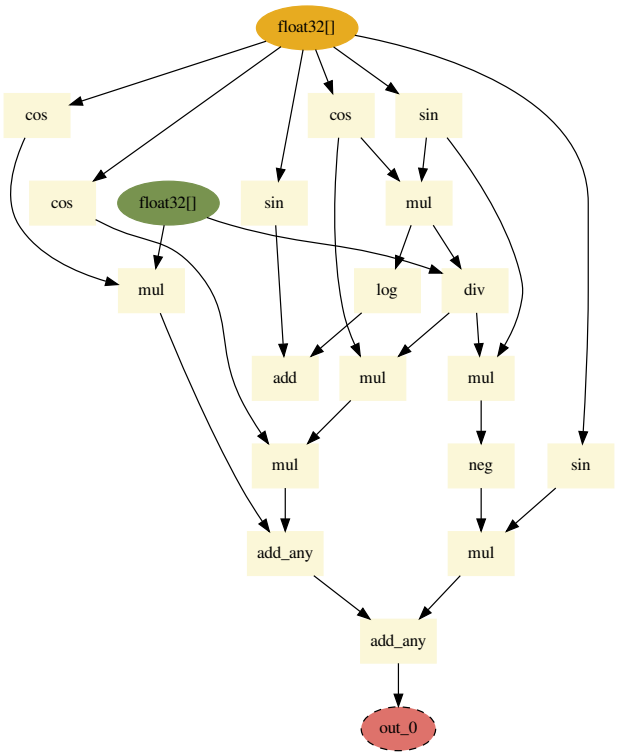


図1　$\partial_x(\log\cos x \sin x + \sin x)$ の計算グラフ

河原創

2025 年 6 月 24 日

目次

第 1 章

イントロダクション

JAX は google が開発中の自動微分と XLA(accelerated Linear Algebra) パッケージです [2]。つまり自動微分＋線形代数パッケージで、なおかつ numpy と同じ形式で書けるようになっています。本稿はこの JAX と PPL として特に numpyro、また最適化ツール JAXopt を使って遊びたいというための文書です。普通であれば「JAX/numpyro で学ぶ機械学習」のようなタイトルになると思うのですが、むしろ私は JAX/numpyro で遊びたいのです。というのも、毎年、年末年始になると時間ができるので新しい言語やパッケージで遊びたくなるのです。2021 年はコロナ下ということもあり JAX で太陽系外惑星のスペクトル推定コードを書いてみようというのをやってみて査読論文にしました。最近続編もだしました。さて、最近は JAX をはじめとする自動微分を実装したプログラミングを微分可能プログラミング (Differentiable Programming) と呼び、新たなパラダイムとみなすことが多くなってきました。そこでえ 2024 年末にタイトルを変更し「微分可能プログラミング」が前面に出るようにしました。これには”The Elements of Differentiable Programming” の登場 [1] が大きかったように思います。随時更新する予定です[*1]。

本稿で用いられているコードはgithubにおいたのでご利用ください。

1.1 自動微分の簡単な紹介

勾配ベースの最適化や HMC-NUTS、変分推定にはモデルを最適化あるいは推定したいパラメタで微分できる必要がある。数値的に微分値を求めるにはおおまかに 4 つの方法がある。一つ目は手で微分 (manual differentiation) をし、結果をコーディングすることである。これはモデルが複雑になってくると破綻しがちであり、またフレキシビリティの観点からもモデルの継続的な改良を妨げる。次に mathematica 等による symbolic differentiation を用いて手で微分する代わりに微分結果を得てコーディングすることが考えられる。これは mathematica 等を用いたことのある方ならばわかると思うが、モデルが複雑になってくると膨大な項数の結果が排出されるので、やはりフレキシビリティの観点からは難点がある。次に、数値微分を行うことが考えられる。数値微分はモデルが複雑になってくるとエラーがたまりやすい。そこで機械学習分野などで用いられているのが JAX でも用いられている自動微分である。コーディングできるモデルは通常、様々な微分の既知である関数（ここでは要素関数と呼ぼう）の加減乗除の組み合わせでできている。そこで、自動微分では、

[*1] 質問・コメント・お問い合わせはdivrot@gmail.comまでお寄せください。なお筆者はソプラノサックスプレイヤー兼野生の天文学の研究者です。

連鎖則が成り立つように各要素関数

$$x :\to f(x) \tag{1.1}$$

を関数のアウトプットとその微分値の情報を出力するように拡張する。また微分の加乗除の演算規則がなりたつように演算を定義する。自動微分の実装法としては JAX では Jacovian Vector Product (JVP) を用いたものが使用されている。しかしここではイントロダクションとしてより説明が容易な、双対数を用いた方式を使用して自動微分を解説しよう。

双対数 (dual number)、$z \in k[\epsilon]/\langle\epsilon^2\rangle$[*2]は $a, b \in \mathbb{R}$ にたいし、

$$z = a + b\epsilon \tag{1.2}$$
$$\epsilon^2 = 0 \tag{1.3}$$

となる数である。複素数は $i^2 = -1$ であったのが $\epsilon^2 = 0$ となったと考えればよい。変数 x の拡張として実部 a に x を、非実部 b に x' を割り当てると、$z = f + f'\epsilon, w = g + g'\epsilon$ に対し、可算・乗数・除算はそれぞれ

$$z + w = (f + g) + (f' + g')\epsilon \tag{1.4}$$
$$zw = fg + (f'g + fg')\epsilon \tag{1.5}$$
$$z/w = \frac{f}{g} + \frac{f'g - fg'}{g^2}\epsilon \tag{1.6}$$

となる。これは実部に関して通常の、また非実部に関して微分の加乗除則と同じであることを示している。次に連鎖則を実現するには関数 $x :\to F(x)$ を

$$x + x'\epsilon :\to F(x) + F'(x)x'\epsilon \tag{1.7}$$

と拡張すればよい。ここで $F(x)$ を元の関数、双対数をもつ拡張された関数を $\hat{F}(x + x'\epsilon)$ と分けて表記する。すると式 (1.7) は

$$\hat{F}(x + x'\epsilon) = F(x) + F'(x)x'\epsilon \tag{1.8}$$

と書ける。さて連鎖則 $G(F(x))$ の拡張は

$$\hat{G}(\hat{F}(x + x'\epsilon)) = \hat{G}(F(x) + F'(x)x'\epsilon) \tag{1.9}$$
$$= G(F(x)) + G'(F(x))F'(x)x'\epsilon \tag{1.10}$$

となり、実部は通常の合成 $(G(F(x)))$ が、非実部は連鎖則 $G'(F(x))F'(x)x' = \frac{dG}{dF}\frac{dF}{dx}x'$ が実現されているのが確認できる。

1.1.1 ミニマム自動微分

というわけで小さな自動微分を Python で実装してみよう。ここで解きたい問題を

$$F(x) = \log(\cos x \sin x) + \sin x \tag{1.11}$$

[*2] 多項式環の商環のことをしめしている。詳しくはたとえば雪江明彦「環と体とガロア理論」などを参照。

の $x=1$ で微分 $F'(1)$ とする。さらにその合成関数

$$\begin{aligned} G(x) &= F(F(x)) \\ &= \log\left(\cos\left(\log\left(\cos x \sin x\right) + \sin x\right) \sin\left(\log\left(\cos x \sin x\right) + \sin x\right)\right) + \sin\left(\log\left(\cos x \sin x\right) + \sin x\right) \end{aligned} \tag{1.12}$$

の $x=1$ での微分 $G'(1)$ も求めたい。式 (1.12) を手で微分するのは大変であるし、symbolic differentiation の結果も長大である。しかし自動微分ならばかなりすっきりしたコードになる。

まず関数 (1.11) に存在する演算は可算と乗算であるので、式 (1.4)、(1.5) に対応した双対数の可乗算を定義する。

```
def mul(x, y):
    a, b = x
    c, d = y
    return a*c, a*d + b*c

def add(x, y):
    a, b = x
    c, d = y
    return a + c, b + d
```

ここに x, y はそれぞれ双対数である。次に関数 (1.11) に存在する要素関数は $\sin x$、$\cos x$、$\log x$ である。式 (1.8) を実装するには、双対数の実部と非実部をペア (a, b) の形式であらわすと、関数 $F(x)$ に対し

$$(x, dx) \to (F(x), F'(x)dx) \tag{1.13}$$

となるように入出力を与えればよいことがわかる。そこで

```
import numpy as np
def cos(x):
    a, b = x
    return np.cos(a), - np.sin(a)*b

def sin(x):
    a, b = x
    return np.sin(a), np.cos(a)*b

def log(x):
    a, b = x
    return np.log(a), b/a
```

これで終わりである。あとは

```
f = lambda x: add(log(mul(cos(x),sin(x))),sin(x))
```

PPL	backend
NumPyro	JAX
Tensorflow Probability	Tensorflow
Pyro	PyTorch
PyMC	Aesara/JAX
BlackJAX *	JAX

表1.1　代表的な PPL とそのバックエンド。∗: BlackJAX は厳密には PPL ではなくサンプラーである。さまざまな PPL 上で利用可能。

```
df = lambda x: f([x,1.0])
df(1.0) #->(0.05324076815279066, -0.37501280285243144)
```

とすれば $(F(1), F'(1))$ が求まる。$G'(x)$ も簡単で、

```
g = lambda x: f(f(x))
dg = lambda x: g([x,1.0])
dg(1.0) #-> (-2.8816056725768977, -7.391555094461485)
```

とするだけである。以上の例からわかるように自動微分の実際の計算には代数計算を用いているため数値微分に比べて誤差の蓄積は少ない。またフレキシブルにコーディングが可能である。

ここではミニマムな自動微分の説明を行った。第3章ではもう少し踏み込んだ説明を行うので興味のある方は参照されたし。

1.2　確率プログラミング言語

確率プログラミング言語 (Probalistic Programming Language; PPL) は確率モデルを扱うためのプログラミング言語のことである。様々なものが存在するが、ほとんどのものはバックエンドとして自動微分パッケージを用いている。表1.1に Python で動く代表的な PPL とそのバックエンドを示す。

本文書では JAX をバックエンドとする NumPyro を用いる。

第 2 章

JAX の自動微分で遊ぶ

JAX [2] は tensorflow や PyTorch 等と同様に自動微分 [7, 3] を利用することができる。またその構文は numpy と形式的にほぼ対応している。また、後の章では JAX を用いた最適化や MCMC などを行う。そのためにまず JAX で様々な関数を作成できるようにしよう。本章ではまず JAX で一通り自動微分を使用してみたから、第3.3章の自動微分のカスタマイズの説明の際に、自動微分の原理の説明を行う構成である。逆に言うと自分で自動微分を定義する必要に迫られない限り原理の理解は必要ないところが JAX の素晴らしいところでもあるといえるかもしれない。

2.1 自動微分

JAX の自動微分で遊んでみよう。

2.1.1 一変数の自動微分

$$f(x) = |x| + \sin x \tag{2.1}$$

を自動微分してみよう。まず jax.numpy をインポートする[*1]。

```
import jax.numpy as jnp
```

jax.numpy の関数を利用して

```
def f(x):
    return jnp.abs(x) + jnp.sin(x)
```

もしくはラムダ関数を利用して

```
f = lambda x:jnp.abs(x) + jnp.sin(x)
```

[*1] jax.numpy は numpy と併用することが多く、jax.numpy を np で numpy を onp とする流派、jax.numpy を jnp で numpy を np とする流派がいるようである。本稿では後者を採用する。

のように関数 $f(x)$ を定義する。

自動微分は grad を用いる。

```
from jax import grad
```

微分を

```
df=grad(f)
```

とすると、例えば、$f'(1.0)$ は

```
df(1.)    # → DeviceArray(1.5403023, dtype=float32)
```

のようにして求まる[*2]。二階微分も同様に

```
df=grad(df)
ddf(1.0) # →  DeviceArray(-0.841471, dtype=float32)
```

のように求まる。

ベクトルマッピング

上では微分の引数として、スカラーを代入していた。以下のようなベクトルを代入するとエラーとなる。

```
xv=jnp.linspace(-10,10,100)
df(xv) # → エラー!
```

これには jax.vmap を用いてマッピングをすることで可能となる。

```
from jax import vmap
dfv=vmap(grad(f))
dfv(xarr) # → jnp.array で返ってくる。
```

これで図が描けるようになった。

```
import matplotlib.pyplot as plt
plt.plot(xarr,f(xarr),".",label="$f(x)=|x|+\sin{x}$")
plt.plot(xarr,dfv(xarr),".",label="f'(x)")
plt.xlabel("x")
plt.legend()
```

[*2] df(1) だとエラーがでる。これは引数に int を取れないためである。

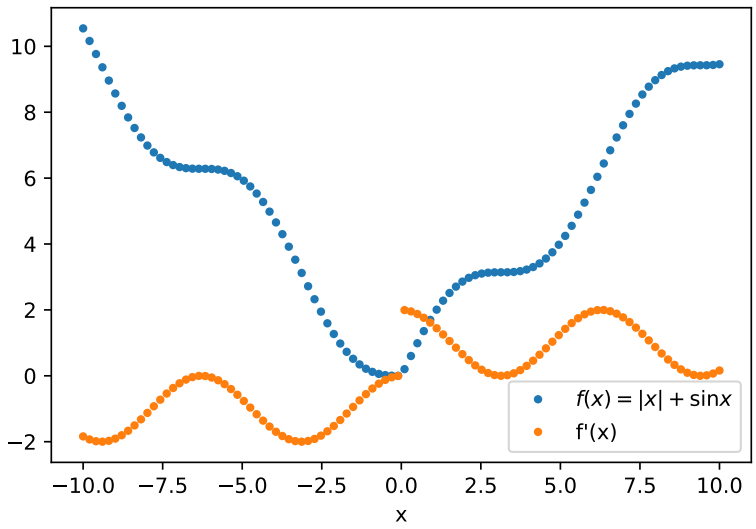


図2.1

2.1.2 多変数の自動微分

次に $\boldsymbol{r} = (x, y, z)^\top$ を変数とした多変数関数

$$g(\boldsymbol{r}) = \sqrt{x^2 + 2y^2 + 3z^2} \tag{2.2}$$

の自動微分を行おう。関数定義は

```
g = lambda x,y,z: jnp.sqrt(x**2+2*y**2+3*z**2)
```

である。偏微分

$$\partial_x g(\boldsymbol{r}) = \frac{\partial g(\boldsymbol{r})}{\partial x} \tag{2.3}$$

として $\boldsymbol{r}_1 = (1, 1, 1)^\top$ での $\partial_x g(\boldsymbol{r}_1)$ は

```
dgdx=grad(g,argnums=0)
dgdx(1.,1.,1.) # → DeviceArray(0.40824828, dtype=float32)
```

ここで argnums が微分する変数を指定している。したがって

$$\partial_y g(\boldsymbol{r}) = \frac{\partial g(\boldsymbol{r})}{\partial y} \tag{2.4}$$

として $\boldsymbol{r}_1 = (1, 1, 1)^\top$ での $\partial_y g(\boldsymbol{r}_1)$ は

```
dgdy=grad(g,argnums=1)
dgdy(1.,1.,1.) # → DeviceArray(0.81649655, dtype=float32)
```

となる。

もしくは

$$\nabla g(\boldsymbol{r}) = \frac{\partial g(\boldsymbol{r})}{\partial \boldsymbol{r}} \tag{2.5}$$

は、

```
dgdr=grad(g,argnums=(0,1,2))
dgdr(1.,1.,1.)
# → (DeviceArray(0.40824828, dtype=float32),
# DeviceArray(0.81649655, dtype=float32),
# DeviceArray(1.2247448, dtype=float32))
```

となる。

マッピング

さて、多変数の場合も、バッチ入力する際、そのまま入力するとエラーとなってしまう。

```
xv=jnp.linspace(-10,10,100)
dgdr(xarr,1.,1.) # → エラー！
```

そこでやはり vmap を利用する。この場合、どの変数をマッピングするか指定する。

```
dgdrv=vmap(grad(g,argnums=(0,1,2)), (0, None,None), 0)
```

(0, None,None) の部分で指定している。

```
dd=dgdrv(xv,1.,1.)
plt.plot(xv,dd[0],label="$\partial_x g(x,1,1)$")
plt.plot(xv,dd[1],label="$\partial_y g(x,1,1)$")
plt.plot(xv,dd[2],label="$\partial_z g(x,1,1)$")
plt.xlabel("x")
plt.legend()
```

で、図2.2を描くことができる。

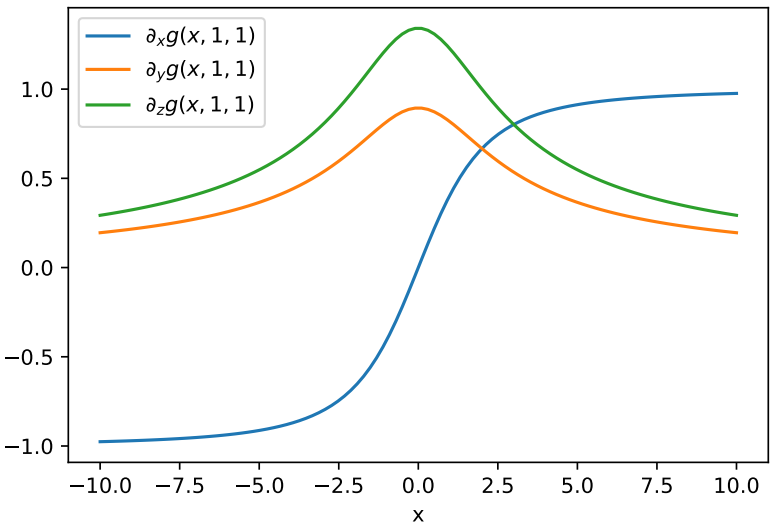


図2.2

2.1.3 入力がベクトルの場合

入力がベクトルの場合も考えよう。例えばとても簡単な

$$f(\boldsymbol{x},\boldsymbol{y}) = \boldsymbol{x}\cdot\boldsymbol{y} \tag{2.6}$$

の場合を考えよう。

```
def f(x,y):
    return jnp.dot(x,y)
```

この場合でも入力がスカラーだったときと同様に自動微分を計算できる。例えば $\partial_{\boldsymbol{x}} f(\boldsymbol{x},\boldsymbol{y})$ は

```
x=jnp.linspace(0,1,11)
y=jnp.linspace(1,2,11)
f(x,y) # → DeviceArray(9.35, dtype=float32)
grad(f,argnums=0)(x,y) # → DeviceArray([1., 1.1,...,2.], dtype=float32)
```

のように計算できる。

vmap を用いれば複数の入力ベクトルに対し、マッピングできる。例えば、5 つの $\boldsymbol{x}$ を vmap で同時に処理してみよう。

```
X=jnp.linspace(0,1,55).reshape(5,11)
vmap(grad(f,argnums=0),(0,None),0)(X,y) # → DeviceArray([[1. ...)
```

ここでも vmap の (0,None) に注意しよう。5 つの $(\boldsymbol{x},\boldsymbol{y})$ に対して同時に処理するには

```
X=jnp.linspace(0,1,55).reshape(5,11)
Y=jnp.linspace(1,2,55).reshape(5,11)
vmap(grad(f,argnums=0),(0,0),0)(X,Y) # → DeviceArray([[1. ...)
```

のように、vmap の引数を (0,0) にすれば良いこともわかる。

2.1.4 練習

微分が含まれた関数の表示

経済学で用いられる Utility function $u(x)$ を考えよう。Utility function は上に凸（concave function）であるとして Absolute Risk Aversion (ARA) と Relative Risk Aversion (RRA)

$$\mathrm{ARA}(x) \equiv -\frac{u''(x)}{u'(x)} \tag{2.7}$$

$$\mathrm{RRA}(x) \equiv -\frac{u''(x)}{u'(x)}x \tag{2.8}$$

をプロットしてみよう。以下では、例として $u(x) = \log(1+x)$ としている。

```
#sample utility function
def u(x):
    return jnp.log(1.0+x)
```

vmap を用いてマッピングした後に演算することで jax numpy array 形式を引数として受け付けるようになる。

```
def ARA(xarr):
    den=vmap(grad(grad(u)))
    bun=vmap(grad(u))
    return -den(xarr)/bun(xarr)
```

```
def RRA(xarr):
    den=vmap(grad(grad(u)))
    bun=vmap(grad(u))
    return -den(xarr)/bun(xarr)*xarr
```

プロットは

```
x_=jnp.linspace(0,3,100)
plt.plot(x_,u(x_),label="$u(x)$")
plt.plot(x_,ARA(x_)
,label="ARA$(x) \equiv - u^{\prime\prime}(x)/u^\prime (x)$",ls="dashed")
```

```
plt.plot(x_,RRA(x_)
,label="RRA$(x) \equiv - u^{\prime\prime}(x)/u^\prime (x) x$",ls="dotted")
plt.legend()
plt.xlabel("$x$")
plt.xlim(0.,3.0)
plt.ylim(0,3)
```

のようにすればよい。図2.3に結果を示す。

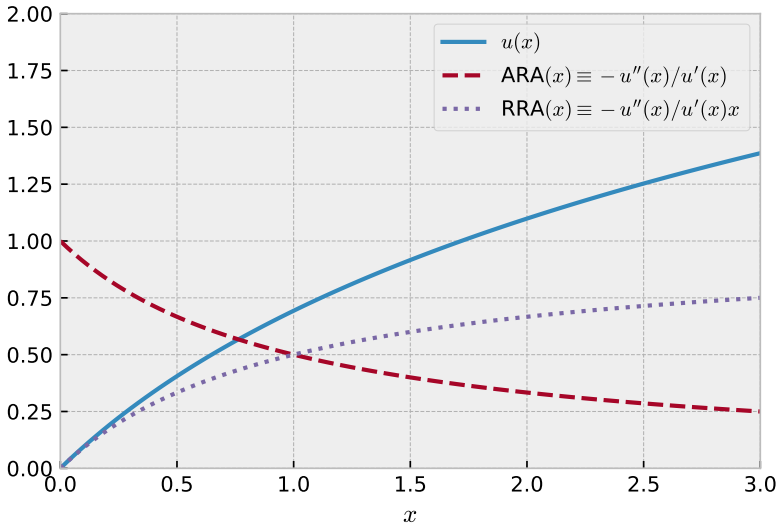


図2.3

コントロールパラメタを入れる

$$u(x, A) = \log(A + x) \tag{2.9}$$

のようにコントロールパラメタ A を入れた場合についても例示しよう。この場合、

```
def u(x,A):
    return jnp.log(A+x)
```

を用いることになる。さてここでは更に ARA や RRA の微分を図示しよう。すなわち

$$\partial_x \mathrm{ARA}(x) = \frac{\partial}{\partial x}\left(-\frac{u''(x)}{u'(x)}\right) \tag{2.10}$$

$$\partial_x \mathrm{RRA}(x) \equiv \frac{\partial}{\partial x}\left(-\frac{u''(x)}{u'(x)}x\right) \tag{2.11}$$

をさまざまな A で図示したい。これには微分があくまで第 0 argument について行われていることに注意し、vmap は最後にこれまた第 0 argument についてマッピングすることに注意すると

```
def dARA(xarr,A):
    den=grad(grad(u,argnums=(0)),argnums=(0))
    bun=grad(u,argnums=(0))
    g=lambda x,A:-den(x,A)/bun(x,A)
    h=vmap(grad(g,argnums=(0)), (0, None), 0)
    return h(xarr,A)
```

および

```
def dRRA(xarr,A):
    den=grad(grad(u,argnums=(0)),argnums=(0))
    bun=grad(u,argnums=(0))
    g=lambda x,A:-den(x,A)/bun(x,A)*x
    h=vmap(grad(g,argnums=(0)), (0, None), 0)
    return h(xarr,A)
```

のように定義すれば良いことがわかる。図2.4は

```
Aarr=[-1.0,0.0,1.0]
fig=plt.figure(figsize=(15,3))
for i,Ain in enumerate(Aarr):
    ax=fig.add_subplot(1,3,i+1)
    x_=jnp.linspace(-Ain+0.5,-Ain+3,100)
    plt.title("A="+str(Ain))
    ax.plot(x_,u(x_,Ain),label="$u(x)$")
    ax.plot(x_,dARA(x_,Ain),label="$\partial_x$ARA$(x)$",ls="dashed")
    ax.plot(x_,dRRA(x_,Ain),label="$\partial_x$RRA$(x)$",ls="dotted")
    plt.legend()
    plt.xlabel("$x$")
```

のように書ける。

2.2 for ループを自動微分する

さてこれまで比較的簡単な関数を自動微分してきた。この程度なら手で（もしくは sympy とか mathematica で）微分できるじゃないかといわれそうである。まあ別に究極的には手でできるのだが、いかにも面倒くさそうな場合として関数が for loop を含む場合[*3]、自動微分を使えると便利そうである。

[*3] 古代の言語 fortran ならば do ループのことである。

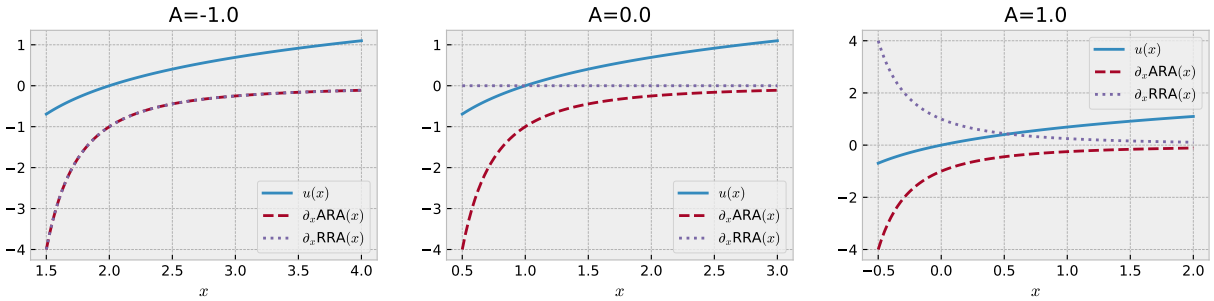


図2.4

jax では fori_loop というのがあるが、version 0.2.6 の時点ではこれを用いると自動微分が通らない。そこで用いるのが jax.lax.scan である。scan と等しい python コードは jax 公式チュートリアルによると

```
def scan(f, init, xs, length=None):
  if xs is None:
    xs = [None] * length
  carry = init
  ys = []
  for x in xs:
    carry, y = f(carry, x)
    ys.append(y)
  return carry, np.stack(ys)
```

である。これは for loop より一般的だが、for loop に使おうと思うとなんかちょっと頭がくらくらする。xs を None として、length を for loop の回数として、ほぼ同じコードなのだが以下のように書き換える。

```
def scan(f, x0, length=n):
  step = [None] * n
  x = x0
  narr = []
  for i in step:
    x, null = f(x, i)
    narr.append(null)
  return x, np.stack(narr)
```

これは、特に使っていない null とか narr を無視すれば、ほとんど以下の for loop と同じである

```
def scan_eq(f,x0,n)
  x=x0
  for i in range(0,n):
    x = f(x)
```

```
    return x
```

というわけで、scan で for loop が実装できることが分かった。

さて、例として

$$f(x) = 1/(1+x) \tag{2.12}$$

を n 回適用させた関数 g(x;n) を考えよう。例えば $n = 3$ の時、

$$g(x;3) = f(f(f(x))) = \frac{1}{1+\frac{1}{1+\frac{1}{1+x}}} \tag{2.13}$$

である。これを微分した関数

$$h(x;n) = \frac{\partial}{\partial x} g(x;n) \tag{2.14}$$

を計算する。

まず scan, grad を import する。

```
from jax.lax import scan
from jax import grad
```

関数 f を定義する。

```
def f(x,null):
    x=1.0/(1.0+x)
    return x,null
```

null は別に意味のない変数だが scan の形式に合わせるために無駄についている。次に scan で for loop 部分を実装する。今は $n = 3$ で試そう。

```
def g(x0):
    x,null=scan(f,x0,None,3)
    return x
```

そして自動微分を定義する。ここでベクトル化した xarr を代入するために vmap しておこう。jit というのは Just-In-Time コンパイラ（実行時コンパイラ）のことで、コンパイルするので高速になる。今回の例では jit をするのとしないので雲泥の差が出るので試されたし。

```
from jax import vmap
from jax import jit
h=jit(vmap(grad(g)))
xarr=jnp.linspace(0,1,100)
harr=h(xarr)
```

さて、答え合わせを行うために sympy でも計算しておこう。

```
import sympy as sp
x = sp.Symbol('x')
def f_sp(x):
    return 1/(1+x)
sp.diff(f_sp(f_sp(f_sp(x))),x)
```

答えが、

$$h(x,3) = -\frac{1}{\left(1+\frac{1}{1+\frac{1}{x+1}}\right)^2\left(1+\frac{1}{x+1}\right)^2(x+1)^2} \tag{2.15}$$

と求まる。つまり比較すべき関数は

```
h_sp=lambda x:-(f_sp(f_sp(f_sp(x))))**2*(f_sp(f_sp(x)))**2*(f_sp(x))**2
```

となる。比較の図は、

```
import matplotlib.pyplot as plt
import seaborn as sns
plt.style.use('bmh')

fig=plt.figure()
ax=fig.add_subplot(211)
plt.plot(xarr,h_sp(xarr)-harr)
plt.ylabel("sympy-jax")
ax=fig.add_subplot(212)
plt.plot(xarr,harr)
plt.xlabel("$x$")
plt.ylabel("$\partial_x f(f(f(x)))$")
plt.show()
```

でかけ、結果は、図2.5である。このように直接関数定義した場合と 6,7 桁くらいの精度で一致しているのがわかる。

2.2.1 練習

scan のインプット配列で微分する

jax.lax.scan の他の使い方を練習してみよう。例えば $\boldsymbol{x} = (x_0, x_1, x_2, x_3)^\top$ に対して関数

$$f(\boldsymbol{x}) = \left(\left(\left(\left(x_0 + \frac{1}{x_0}\right)x_1 + \frac{1}{x_1}\right)x_2 + \frac{1}{x_2}\right)x_3 + \frac{1}{x_3}\right) \tag{2.16}$$

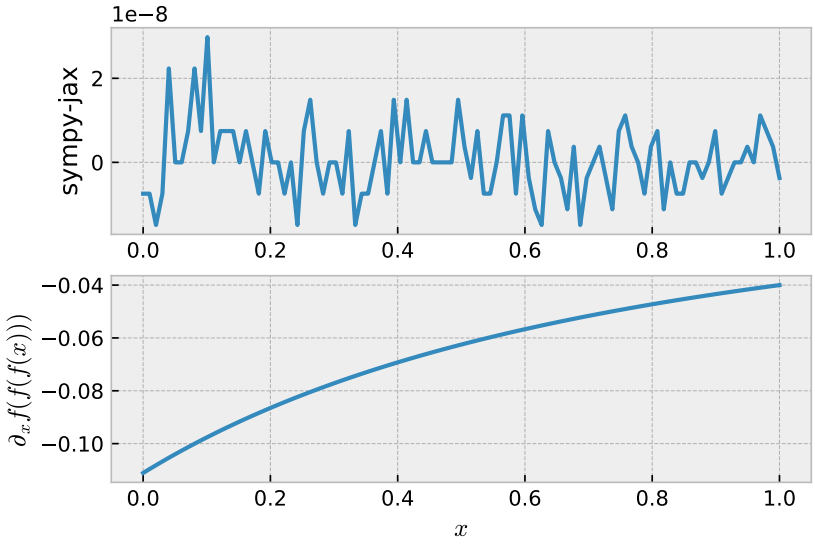


図2.5

を定義する。この関数の微分

$$\boldsymbol{h}(\boldsymbol{x}) \equiv \nabla f(\boldsymbol{x}) = \left(\frac{\partial f(\boldsymbol{x})}{\partial x_0}, \frac{\partial f(\boldsymbol{x})}{\partial x_1}, \frac{\partial f(\boldsymbol{x})}{\partial x_2}, \frac{\partial f(\boldsymbol{x})}{\partial x_3} \right)^\top \tag{2.17}$$

の自動微分を実装してみよう。scan(f, init, xs) の xs を $\boldsymbol{x}$ に用いればよいことが予想される。式 (2.16) は内側のかっこから再帰的に $(\cdot)x_i + 1/x_i$ を計算していけばよいことに気づく。そこで、まず

```
def g(y,x):
    y=y*x+1.0/x
    return y,None
```

とする。そして scan を用いて

```
@jit
def f(xs):
    y0=1.0
    y,null=scan(g,y0,xs)
    return y
```

とすれば再帰的に $y_i = y_{i-1}x_i + 1/x_i$ を計算できることがわかる。この自動微分は単純に

```
h=grad(f)
```

とすればよい。これで、例えば、

```
xs=jnp.array([1.,2.,4.,5.])
h(xs)  # → DeviceArray([ 0.     , 35.     , 22.1875, 18.21  ], dtype=float32)
```

と求まる。この形式はいくらでも要素数を増やせる利点がある。すなわち例えば、

```
h(jnp.array([1.,2.,3.,4.,5.,6.,7.,8.]))
# → DeviceArray([0., 35280., 29493.332, 23135.
# , 18662.559, 15572.979 , 13350.7705, 11682.194 ], dtype=float32)
```

といった感じである。

2.3 JIT とクラス

クラスを定義してクラス内の中においた関数に JIT をおこなえると便利である。この場合は、以下のように functools.partial を用いてデコレータをつけるとよい。

```
from functools import partial
class fcjax(object):
    def __init__(self,c,n):
        self.c=c
        self.n=n

    def f(self,x,null):
        x=self.c/(self.c+x)
        return x,null

    @partial(jit, static_argnums=(0,))
    def g(self,x0):
        x,null=scan(f,x0,None,self.n)
        return x

    @partial(jit, static_argnums=(0,))
    def h(self,x0):
        hh=vmap(grad(self.g))
        return hh(x0)
```

このようにクラスで書くことにより、例えば、関数の中を引数を変えて

```
for i in range(1,4):
    fcc=fcjax(1.0,i)
```

```
    plt.plot(fcc.h(xarr),label="n="+str(i))
plt.legend()
plt.show()
```

のように使うことができる。また、JIT で実行時にコンパイルされてしまうので、一度、g や h を呼び出してしまうと、その後に fcc.n=4 のように更新しても、関数の方の fcc.n は変更されないことに注意が必要である

```
fcc=fcjax(1.0,1)
fcc.n=2
fcc.h(xarr) #この時点でコンパイルされ、n=2で計算される。
fcc.n=3
fcc.h(xarr) #すでにコンパイルされてしまっているのでやはりn=2のままで計算される
```

第 3 章

微分可能プログラミングの基礎

3.1 方向微分と Jacobian Vector Product

方向微分とは、多変数関数において、特定の方向に沿った変化率を評価するものである。与えられた点 $\boldsymbol{\omega}$ において関数の変化率を、その点を基準とし任意のベクトル方向 $\boldsymbol{v}$ に沿って計算する。関数 $f(\omega)$ の $\boldsymbol{v} \in \mathbb{R}^D$ 方向の方向微分は以下で定義される。

$$\partial f(\boldsymbol{\omega})[\boldsymbol{v}] \equiv \lim_{\delta \to 0} \frac{f(\omega + \delta \boldsymbol{v}) - f(\omega)}{\delta} \tag{3.1}$$

$\boldsymbol{\omega}$ がベクトルの場合

$\boldsymbol{\omega}$ がベクトルの時、つまり $\boldsymbol{\omega} \in \mathbb{R}^D$ のとき、$\delta \boldsymbol{v}$ が小さいとすると ($\delta|\boldsymbol{v}| \sim 0$)、Jacobian $\boldsymbol{\partial} f(\boldsymbol{\omega}) \in \mathbb{R}^{M \times D}$ をもちいて、

$$f(\boldsymbol{\omega} + \delta \boldsymbol{v}) \sim f(\boldsymbol{\omega}) + \boldsymbol{\partial} f(\boldsymbol{\omega}) \delta \boldsymbol{v} \tag{3.2}$$

となることから、

$$\partial f(\boldsymbol{\omega})[\boldsymbol{v}] = (\boldsymbol{\partial} f)\boldsymbol{v} = \mathrm{JVP}(\boldsymbol{f}, \boldsymbol{v}) \tag{3.3}$$

となることより、方向微分は Jacobian Vector Product (JVP) とも呼ばれる。

$\boldsymbol{\omega}$ が行列の場合

$\boldsymbol{\omega}$ は行列でも良い。そこで $\boldsymbol{\omega}$ を行列 W であるとする。

1. $f(W) = W\boldsymbol{x}$
   行列による線形変換

$$f(W) = W\boldsymbol{x} \tag{3.4}$$

を考える。このとき、JVP は

$$\partial f(W)[V] \equiv \lim_{\delta \to 0} \frac{(W - \delta V)\boldsymbol{x} - W\boldsymbol{x}}{\delta} = V\boldsymbol{x} \tag{3.5}$$

となる。これは行列 W をベクトル化して式 (3.3) で JVP を計算したものと一致する。これを二次元の場合で例示する。

$$W = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \tag{3.6}$$

のとき、$\boldsymbol{\omega} = \mathrm{Vec}(W) = (a, b, c, d)^T$ と同型変換すると、$f(W) = W\boldsymbol{x}$ は

$$f(\boldsymbol{\omega}) = X\boldsymbol{\omega},\ X \equiv \begin{pmatrix} x & y & 0 & 0 \\ 0 & 0 & x & y \end{pmatrix} \tag{3.7}$$

となる。Jacobian は

$$\boldsymbol{\partial} f(\boldsymbol{\omega}) = \begin{pmatrix} \partial_a(ax+by) & \partial_b(ax+by) & \partial_c(ax+by) & \partial_d(ax+by) \\ \partial_a(cx+dy) & \partial_b(cx+dy) & \partial_c(cx+dy) & \partial_d(cx+dy) \end{pmatrix} = X \tag{3.8}$$

であるので、V をベクトル化したものを $\boldsymbol{v} = \mathrm{vec}(V)$ とおくと、

$$\boldsymbol{\partial} f(\boldsymbol{\omega})\boldsymbol{v} = X\boldsymbol{v} \tag{3.9}$$

となっている。これは $X\boldsymbol{v} = V\boldsymbol{x}$ であることから

$$\partial f(W)[V] = V\boldsymbol{x} \tag{3.10}$$

であることが確認できる。

2. $f(W) = WX$

X も行列の場合を考える。つまり

$$f(W) = WX \tag{3.11}$$

の場合、JVP は同様に

$$\partial f(W)[V] \equiv \lim_{\delta \to 0} \frac{(W - \delta V)X - WX}{\delta} = VX \tag{3.12}$$

である。

3.1.1 JVP の連鎖則

Jacobian の連鎖則

$$\partial(g \circ f)(\boldsymbol{\omega}) = \partial g(f(\boldsymbol{\omega}))\partial f(\boldsymbol{\omega}) \tag{3.13}$$

であることから

$$\partial(g \circ f)(\boldsymbol{\omega})[\boldsymbol{v}] = \boldsymbol{\partial}(g \circ f)(\boldsymbol{\omega})\boldsymbol{v} = \partial g(f(\boldsymbol{\omega}))\partial f(\boldsymbol{\omega})\boldsymbol{v} = \partial g(f(\boldsymbol{\omega}))[\partial f(\boldsymbol{\omega})[\boldsymbol{v}]] \tag{3.14}$$

となる。つまり $g \circ f$ の方向微分は、図3.1に示すように、g の評価点として $f(\omega)$、方向として f の $\boldsymbol{v}$ 方向の方向微分を用いた方向微分を計算すればよい。このような計算を繰り返すことにより JVP の合成関数の計算を順次的に行うことができる (図3.2)。このように順次的に自動微分を計算していく手法をフォワードモード微分と呼ぶ。

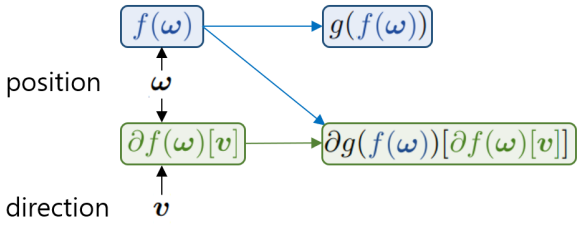


図3.1　$g \circ f$ の方向微分 $\partial g(f(\boldsymbol{\omega}))[\partial f(\boldsymbol{\omega})[\boldsymbol{v}]]$、JVP の計算グラフ。

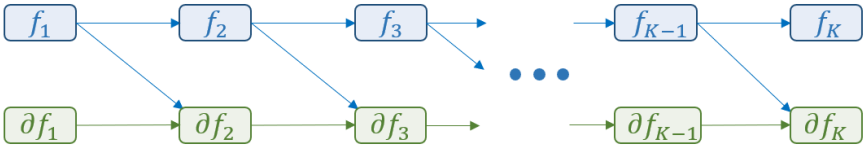


図3.2　フォワードモード微分。

もうすこし具体的に $\boldsymbol{\omega} \in \mathbb{R}^N, \boldsymbol{f} \in \mathbb{R}^M$, として、複数の関数の合成関数である $\boldsymbol{f}(\boldsymbol{\omega})$ を考える。JVP を用いて、Jacobian、

$$J_{\boldsymbol{f},\boldsymbol{\omega}} \equiv \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\omega}} = \begin{pmatrix} \frac{\partial f_1}{\partial \omega_1} & \frac{\partial f_1}{\partial \omega_2} & \cdots & \frac{\partial f_1}{\partial \omega_N} \\ \frac{\partial f_2}{\partial \omega_1} & \frac{\partial f_2}{\partial \omega_2} & \cdots & \frac{\partial f_2}{\partial \omega_N} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_M}{\partial \omega_1} & \frac{\partial f_M}{\partial \omega_2} & \cdots & \frac{\partial f_M}{\partial \omega_N} \end{pmatrix} \in \mathbb{R}^{M \times N} \tag{3.15}$$

を計算したい。$\boldsymbol{f}$ についての JVP、

$$\mathrm{JVP}(\boldsymbol{f}, \boldsymbol{v}) = J_{\boldsymbol{f},\boldsymbol{\omega}} \boldsymbol{v} = \left(\frac{\partial f_1}{\partial \boldsymbol{\omega}} \boldsymbol{v}, \ldots, \frac{\partial f_M}{\partial \boldsymbol{\omega}} \boldsymbol{v} \right)^\top \tag{3.16}$$

が計算できるとすると、$\dfrac{\partial f_i}{\partial \omega_j}$ は、

$$\mathrm{JVP}(\boldsymbol{f}, \boldsymbol{e}_j) = \frac{\partial \boldsymbol{f}}{\partial \omega_j} \tag{3.17}$$

の i 成分を取り出すことで計算できる。ここに $\boldsymbol{e}_j$ は j 番目の単位ベクトル（j 番目の要素だけが 1 で残りが 0 のベクトル）である。そこで $\boldsymbol{f}$ の JVP を、各関数の JVP から合成することを考えよう。

まず $\boldsymbol{f}(\boldsymbol{\omega}) = \boldsymbol{f}_2(\boldsymbol{f}_1(\boldsymbol{\omega}))$ という場合の連鎖がどうなるか考えよう。$\boldsymbol{f}$ の微分は $\boldsymbol{\omega}_1 = \boldsymbol{f}_1$ と書くと、

$$J_{\boldsymbol{f},\boldsymbol{\omega}} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\omega}} = \left(\frac{\partial \boldsymbol{f}_2(\boldsymbol{\omega}_1)}{\partial \boldsymbol{\omega}_1} \right) \left(\frac{\partial \boldsymbol{\omega}_1}{\partial \boldsymbol{\omega}} \right) = \left(\frac{\partial \boldsymbol{f}_2(\boldsymbol{\omega}_1)}{\partial \boldsymbol{\omega}_1} \right) \left(\frac{\partial \boldsymbol{f}_1(\boldsymbol{\omega})}{\partial \boldsymbol{\omega}} \right) \tag{3.18}$$

$$= J_{\boldsymbol{f}_2,\boldsymbol{\omega}_1} J_{\boldsymbol{f}_1,\boldsymbol{\omega}} \tag{3.19}$$

である。つまり一般に $\boldsymbol{f}(\boldsymbol{\omega}) = \boldsymbol{f}_L(\boldsymbol{f}_{L-1}(\cdots \boldsymbol{f}_2(\boldsymbol{f}_1(\boldsymbol{\omega}))$ の場合、

$$J_{\boldsymbol{f},\boldsymbol{\omega}} = J_{\boldsymbol{f}_L,\boldsymbol{\omega}_{L-1}} \cdots J_{\boldsymbol{f}_2,\boldsymbol{\omega}_1} J_{\boldsymbol{f}_1,\boldsymbol{\omega}} \tag{3.20}$$

ということである。

式 (3.20) の右辺に $\boldsymbol{v}$ をかければ

$$J_{\boldsymbol{f},\boldsymbol{\omega}}\boldsymbol{v} = J_{\boldsymbol{f}_L,\boldsymbol{\omega}_{L-1}} \cdots J_{\boldsymbol{f}_2,\boldsymbol{\omega}_1} J_{\boldsymbol{f}_1,\boldsymbol{\omega}}\boldsymbol{v} \tag{3.21}$$

$$= J_{\boldsymbol{f}_L,\boldsymbol{\omega}_{L-1}}(\cdots J_{\boldsymbol{f}_3,\boldsymbol{\omega}_1}(J_{\boldsymbol{f}_2,\boldsymbol{\omega}_1}(J_{\boldsymbol{f}_1,\boldsymbol{\omega}}\boldsymbol{v}))\cdots) \tag{3.22}$$

となるが、これは式 (3.30) のことである。右（小さい L の順）から JVP を適用していけば $J_{\boldsymbol{f},\boldsymbol{\omega}}\boldsymbol{v}$ ひいては $J_{\boldsymbol{f},\boldsymbol{\omega}}$ の各成分が求まることがわかる。フォワードモードに必要なのは $\mathrm{JVP}(\boldsymbol{f}_1,\boldsymbol{v}), \mathrm{JVP}(\boldsymbol{f}_2,\boldsymbol{v}), \cdots, \mathrm{JVP}(\boldsymbol{f}_L,\boldsymbol{v})$ であることが確認された。

3.1.2 ヤコビアンの計算量とメモリ使用量

与えられた方向 $\boldsymbol{v} \in \mathbb{R}^D$ に対して JVP が計算できるとして、ヤコビアン $\boldsymbol{\partial} f \in \mathbb{R}^{M\times D}$ を求めるためには式 (3.3) より、

$$\partial_i f(\boldsymbol{\omega}) = \boldsymbol{\partial} f(\boldsymbol{\omega})[\boldsymbol{e}_i] \in \mathbb{R}^M \tag{3.23}$$

のように各単位ベクトル $\boldsymbol{e}_i \in \mathbb{R}^D$ を $i = 1, 2, \cdots, D$ について計算すればよい。つまり一回のヤコビアンの評価には D 個の JVP を必要とする。関数の合成一回分を 1 レイヤーとすると $f^k \in \mathbb{R}^{D_k}$ から $f^{k-1} \in \mathbb{R}^{D_{k-1}}$ への計算は $D_k \times D_{k-1}$ の計算コストがかかるので、連鎖の各レイヤーで D の数が不変とする ($D_k = D_{k-1} = D$) と、連鎖のレイヤー数を K のとき KD^3 の計算コストがかかる。

3.2 Vector Jacobian Product

JVP、つまり方向微分 $\partial f(\boldsymbol{\omega})[\boldsymbol{v}]$ と双対となる $\boldsymbol{u}$ との内積 $\langle , \rangle \in \mathbb{R}$ を考え、その随伴を Vector Jacobian Product (VJP) と呼ぶ。すなわち

$$\langle \boldsymbol{u}, \partial f(\boldsymbol{\omega})[\boldsymbol{v}] \rangle = \langle \partial f(\boldsymbol{\omega})^*[\boldsymbol{u}], \boldsymbol{v} \rangle \tag{3.24}$$

となる $\partial f(\boldsymbol{\omega})^*[\boldsymbol{u}]$ が VJP である。$\boldsymbol{\omega} \in \mathbb{R}^N$ がベクトルである時、Jacobian を $J \equiv \boldsymbol{\partial} f \in \mathbb{R}^{M\times N}$ と略記することで、上記の関係は

$$\boldsymbol{u}^T(J\boldsymbol{v}) = (J^T\boldsymbol{u})^T\boldsymbol{v} \tag{3.25}$$

となることから、VJP は

$$\partial f(\boldsymbol{\omega})^*[\boldsymbol{u}] = J^T\boldsymbol{u} \tag{3.26}$$

となる。

ベクトルの内積は $\boldsymbol{a}^T\boldsymbol{b} = \mathrm{tr}(\boldsymbol{b}^T\boldsymbol{a})$ のように trace を使ってもあらわされるが、行列の場合の内積も同様に trace を使って定義する。これはフロベニウス内積となる。

$$\langle A, B \rangle = \mathrm{tr}(B^T A) = \sum_{i,j} A_{ij} B_{ij} \tag{3.27}$$

この定義は行列 A, B をベクトル化したものにベクトル内積を適用したものとも同値であることに注意する。

$\boldsymbol{\omega}$ が行列の場合

1. $f(W) = W\boldsymbol{x}$

   $f(W) = W\boldsymbol{x} \in \mathbb{R}^N$ の VJP は $\partial f(W)[V] = V\boldsymbol{x}$ であったので $\boldsymbol{u} \in \mathbb{R}^N$ として、

$$\langle \boldsymbol{u}, V\boldsymbol{x} \rangle = \boldsymbol{u}^T(V\boldsymbol{x}) = \mathrm{tr}((V\boldsymbol{x})^T\boldsymbol{u}) = \mathrm{tr}(\boldsymbol{u}\boldsymbol{x}^T V^T) = \mathrm{tr}(\boldsymbol{x}\boldsymbol{u}^T V) = \langle \boldsymbol{u}\boldsymbol{x}^T, V \rangle \tag{3.28}$$

より VJP は $\partial f(W)^*[\boldsymbol{u}] = \boldsymbol{u}\boldsymbol{x}^T$ となる。ただし $\mathrm{tr}(\boldsymbol{x}^T\boldsymbol{y}) = \boldsymbol{y}^T\boldsymbol{x}$, $\mathrm{tr}(A) = \mathrm{tr}(A^T)$ と trace の巡回性を用いた。

2. $f(W) = WX$

   JVP は $\partial f(W)[V] = VX$ であったので

$$\langle U, VX \rangle = \mathrm{tr}((VX)^T U) = \mathrm{tr}(U\boldsymbol{x}^T V^T) = \mathrm{tr}(XU^T V) = \langle UX^T, V \rangle \tag{3.29}$$

より VJP は $\partial f(W)^*[U] = UX^T$ となる。

3.2.1 VJP の連鎖則

JVP の時と同様に、gradient の連鎖則 (3.13) を用いて

$$\partial(g \circ f)(\boldsymbol{\omega})^*[\boldsymbol{u}] = \boldsymbol{\partial}(g \circ f)(\boldsymbol{\omega})^T\boldsymbol{u} = (\partial g(f(\boldsymbol{\omega}))\partial f(\boldsymbol{\omega}))^T\boldsymbol{u} = \partial f(\boldsymbol{\omega})^T\partial g(f(\boldsymbol{\omega}))^T\boldsymbol{u} \tag{3.30}$$

$$= \partial f(\boldsymbol{\omega})^*[\partial g(f(\boldsymbol{\omega}))^T\boldsymbol{u}] \tag{3.31}$$

となることより、

$$\partial(g \circ f)(\boldsymbol{\omega})^*[\boldsymbol{u}] = \partial f(\boldsymbol{\omega})^*[\partial g(f(\boldsymbol{\omega}))^*[\boldsymbol{u}]] \tag{3.32}$$

を得る。図3.3に VJP の連鎖則の計算グラフを示す。JVP の時とは異なり、f から g へと関数と微分を順次的に計算することはできない。

図3.4に示すように、VJP ではまず関数そのものの合成を行う forward path の計算を行い、それぞれの関数値をメモリに保存しておき、backward path で微分を計算していくときに利用する。このため JVP より多くのメモリを必要とする。

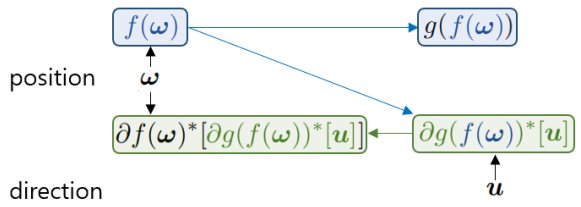


図3.3 $g \circ f$ の VJP の連鎖則の計算グラフ。

またもう少し具体的に $\boldsymbol{\omega}$ がベクトルの場合の VJP を見てみよう。VJP は、

$$\mathrm{VJP}(\boldsymbol{f}, \boldsymbol{u}) = (\boldsymbol{u}^\top J)^\top = \left(\boldsymbol{u} \cdot \frac{\partial \boldsymbol{f}}{\partial \omega_1}, \boldsymbol{u} \cdot \frac{\partial \boldsymbol{f}}{\partial \omega_2}, \ldots, \boldsymbol{u} \cdot \frac{\partial \boldsymbol{f}}{\partial \omega_N} \right)^\top \tag{3.33}$$

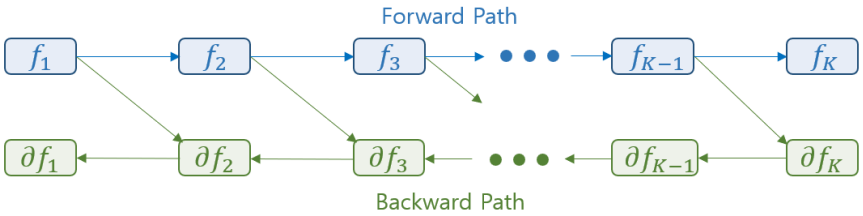


図3.4　リバースモード微分。

と定義される。この場合も同様に $\boldsymbol{u} = \boldsymbol{e}_i$ とおけば、

$$\mathrm{VJP}(\boldsymbol{f}, \boldsymbol{e}_i) = (\boldsymbol{e}_i^\top J_{\boldsymbol{f},\boldsymbol{\omega}})^\top = \left(\frac{\partial f_i}{\partial \omega_1}, \frac{\partial f_i}{\partial \omega_2}, \ldots, \frac{\partial f_i}{\partial \omega_N} \right)^\top = \frac{\partial f_i}{\partial \boldsymbol{\omega}} = \nabla f_i \tag{3.34}$$

が得られるので、Jacobian 各成分が求まることがわかる。VJP を用いる連鎖則の計算法はリバースモードとよばれ、左から

$$\boldsymbol{u}^\top J_{\boldsymbol{f},\boldsymbol{\omega}} = \boldsymbol{u}^\top J_{\boldsymbol{f}_L,\boldsymbol{\omega}_{L-1}} \cdots J_{\boldsymbol{f}_2,\boldsymbol{\omega}_1} J_{\boldsymbol{f}_1,\boldsymbol{\omega}} \tag{3.35}$$

$$= (\cdots((\boldsymbol{u}^\top J_{\boldsymbol{f}_L,\boldsymbol{\omega}_{L-1}}) J_{\boldsymbol{f}_{L-1},\boldsymbol{\omega}_1}) J_{\boldsymbol{f}_{L-2},\boldsymbol{\omega}_1} \cdots) J_{\boldsymbol{f}_1,\boldsymbol{\omega}} \tag{3.36}$$

となり backward differentiation に必要なのは $\mathrm{VJP}(\boldsymbol{f}_L, \boldsymbol{u}), \mathrm{VJP}(\boldsymbol{f}_{L-1}, \boldsymbol{u}), \cdots, \mathrm{VJP}(\boldsymbol{f}_1, \boldsymbol{u})$ ということになる。

3.2.2　JVP と VJP の計算量・メモリ使用量の比較

3.2.3　小まとめ

	JVP (Jacobian-Vector Product)	VJP (Vector-Jacobian Product)
表記	$\partial f(\boldsymbol{\omega})[\boldsymbol{v}]$	$\partial f(\boldsymbol{\omega})^*[\boldsymbol{u}]$
連鎖則	$\partial(g \circ f)(\boldsymbol{\omega})[\boldsymbol{v}] = \partial g(f(\boldsymbol{\omega}))[\partial f(\boldsymbol{\omega})[\boldsymbol{v}]]$	$\partial(g \circ f)(\boldsymbol{\omega})^*[\boldsymbol{u}] = \partial f(\boldsymbol{\omega})^*[\partial g(f(\boldsymbol{\omega}))^*[\boldsymbol{u}]]$
$f(W) = W\boldsymbol{x}$	$\partial f(W)[V] = V\boldsymbol{x}$	$\partial f(W)^*[\boldsymbol{u}] = \boldsymbol{u}\boldsymbol{x}^T$
$f(W) = WX$	$\partial f(W)[V] = VX$	$\partial f(W)^*[U] = UX^T$

表3.1　JVP/VJP まとめ

3.3　JAX の自動微分をカスタマイズする

というわけで JAX に戻って、JVP, VJP を自分で定義してみよう。フォワードモードのカスタム自動微分は、JVP を定義する jax.custom_jvp を使用すれば良い。さて以下の例では

$$g(\boldsymbol{x}) = x_2 \sin x_1 \tag{3.37}$$

の JVP を実装する。みやすさのため $x = x_1$, $y = x_2$ とおいている。

```python
from jax import custom_vjp
@custom_jvp
def g(x, y):
    return jnp.sin(x) * y

@g.defjvp
def g_jvp(primals, tangents):
    x, y = primals
    ux, uy = tangents
    dgdx=y * jnp.cos(x)
    dgdy=jnp.sin(x)
    primal_out = g(x, y)
    tangent_out = dgdx * ux  + dgdy * uy
    return primal_out, tangent_out
```

図3.5に JVP とインプット、アウトプットの対応を示す。tangent_out = dg1dx * ux + dg1dy * uy とおいているが、これが JVP の微分更新部分になっている。つまり、

$$\mathrm{JVP}(g, \boldsymbol{u}) = J_{\boldsymbol{g},\boldsymbol{x}}\boldsymbol{u} = \frac{\partial g}{\partial \boldsymbol{x}}\boldsymbol{u} = \frac{\partial g(x,y)}{\partial x}u_1 + \frac{\partial g(x,y)}{\partial y}u_2 \tag{3.38}$$

$$= (y\cos x)u_x + (\sin x)u_y \tag{3.39}$$

となっている。

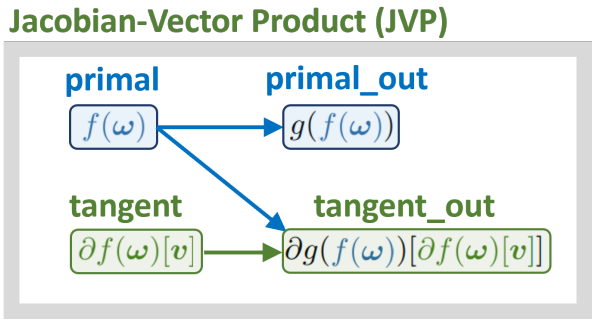


図3.5　JVP の実装。

リバースモードのカスタム自動微分は、VJP を定義する jax.custom_vjp を使用すれば良い。図3.4のようにリバースモードでは、Forward path (fwd) の部分と Backword path (bwd) の部分の二つのメソッドを定義する必要がある。また forward path では、後に backward path で微分を計算するために必要な情報を出力しておく。これを residuals と呼ぶ。この部分は backward path で微分を計算するのに必要ならば、どのような数学的情報でも良いが、メモリ使用量に直接影響するので、最小限にしたほうがよい。

以下では fwd と bwd の実装例を示す。また図3.6に VJP の実装部分のインプット・アウトプットの対応を示す。fwd および bwd は最後に defvjp により VJP として統合して定義される。ちなみにこの例では y と

cos_x を別々に保存しているが、これは掛けた状態で保存したほうがメモリ使用量的には効率が良い。

```python
from jax import custom_vjp

@custom_vjp
def g(x, y):
    return jnp.sin(x) * y

def g_bwd(residuals, u):
    cos_x, sin_x, y = residuals
    return (y * cos_x * u, sin_x * u)

def g_fwd(x, y):
    residuals = (jnp.cos(x), jnp.sin(x), y)
    primal_out = g(x, y)
    return primal_out, residuals

g.defvjp(g_fwd, g_bwd)
```

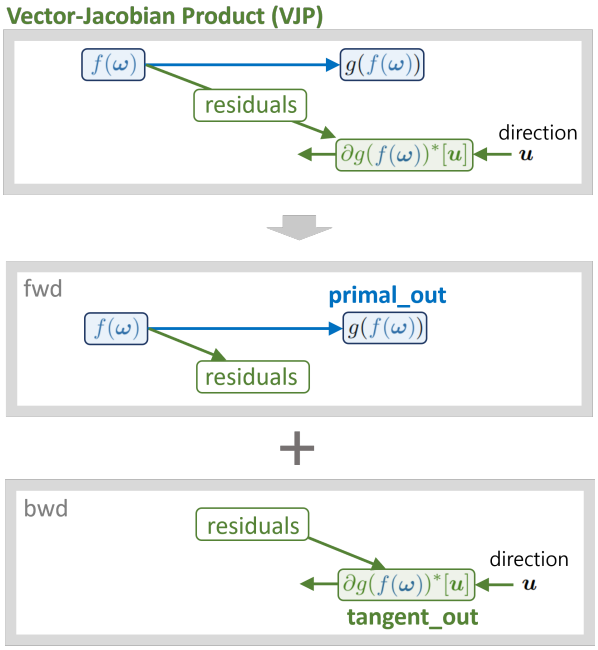


図3.6　VJP の実装。

3.4　Newton-Raphson 法で実装された陰関数の VJP を定義する

一般のプログラム中には陽的な関数ばかりでなく、反復的に解を求める陰関数も使われる。このような場合の微分は、自分で JVP なり VJP を定義する必要がある。ここでは例題として、力学の二体問題中の問題より、eccentricity e と Mean anomaly M から eccentric anomaly E を求める問題を考えよう。とはいえ、力学に興味が無くても問題設定は理解できる。eccentric anomaly は

$$f(E) = E - e\sin E - M = 0 \tag{3.40}$$

を解くことで E が求まる、というだけである。この非線形方程式 $f(E) = 0$ は数値的に求めることが必要である。このための方法の一つとして Newton-Raphson 法がある。Newton-Raphson 法は図3.7にしめすように、まず初期値 E_1 から初めて、$f(E_1)$ に接する直線を求め、その直線と原点との交点を解析的に求め E_2 とする。この手続きを n 回繰り返すという手順である。i 番目の手続きでは接線は

$$y = f'(E_i)(E - E_i) + f(E_i) \tag{3.41}$$

となることから

$$E_{i+1} = E_i - \frac{f(E_i)}{f'(E_i)} \tag{3.42}$$

となる。この手続きを収束条件の判定値を ϵ として $|E_{i+1} - E_i| < \epsilon$ となるまで繰り返し、条件を満たした E_i を近似解とすればよい。

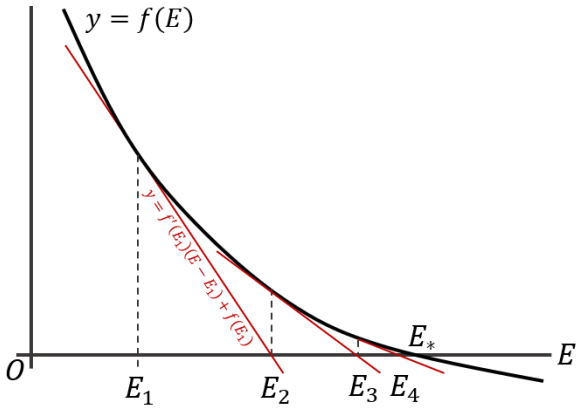


図3.7　Newton-Raphson 法。

さて、いま E は e と M の関数とみなすことができる。すなわち

$$E = E(e, M) \tag{3.43}$$

を上記の Newton-Raphson 法で求める関数を JAX で実装するには、例えば、以下のようにすればよい。ここで e, M 以外にも、Newton-Raphson 法の初期値 E_{ini} も引数に入っている。

```
1 from jax.lax import while_loop
2 import jax.numpy as jnp
```

```
from jax import grad

def f(E, e, M):
    return E - e*jnp.sin(E) - M

dfdE = grad(f, argnums=0)

@custom_vjp
def eccentric_anomaly_newton_raphson(e, M, Eini):
    def cond_fun(carry):
        E_prev, E = carry
        return jnp.abs(E - E_prev) > 1e-6

    def body_fun(carry):
        _, E = carry
        E_new = E - f(E, e, M) / dfdE(E, e, M)
        return E, E_new

    _, E_star = while_loop(cond_fun, body_fun, (jnp.inf, Eini))
    return E_star
```

さて、この関数は while_loop が入っているので reverse-mode で微分が通らない。そこで VJP を自分で定義してやる必要がある。微分は自分で用意する必要があるが、陰関数定理より

$$\frac{\partial E}{\partial e} = -\frac{\partial f/\partial e}{\partial f/\partial E} = \frac{\sin E}{1 - e\cos E} \tag{3.44}$$

$$\frac{\partial E}{\partial M} = -\frac{\partial f/\partial M}{\partial f/\partial E} = \frac{1}{1 - e\cos E} \tag{3.45}$$

である。これより forward path (fwd) では、$\sin E$ と $1 - e\cos E$ を保存しておいて、backward path (bwd) で用いればよいことが分かる。つまり全体として以下のようにすればよい。

```
import jax.numpy as jnp
from jax import grad

def f(E, e, M):
    return E - e*jnp.sin(E) - M

dfdE = grad(f, argnums=0)
dfde = grad(f, argnums=1)
dfdM = grad(f, argnums=2)

```

```
from functools import partial
from jax import custom_vjp
from jax.lax import while_loop

@custom_vjp
def eccentric_anomaly_newton_raphson(e, M, Eini):
    def cond_fun(carry):
        E_prev, E = carry
        return jnp.abs(E - E_prev) > 1e-6

    def body_fun(carry):
        _, E = carry
        E_new = E - f(E, e, M) / dfdE(E, e, M)
        return E, E_new

    _, E_star = while_loop(cond_fun, body_fun, (jnp.inf, Eini))
    return E_star


def eccentric_anomaly_newton_raphson_fwd(e, M, Eini):
    E_star = eccentric_anomaly_newton_raphson(e, M, Eini)
    return E_star, (jnp.sin(E_star), 1.0 - e * jnp.cos(E_star))


def eccentric_anomaly_newton_raphson_bwd(residuals, u):
    sin_E_star, one_minus_cos_E_star = residuals
    return (sin_E_star * u / one_minus_cos_E_star, u / one_minus_cos_E_star, 0.0)


eccentric_anomaly_newton_raphson.defvjp(eccentric_anomaly_newton_raphson_fwd,
                                        eccentric_anomaly_newton_raphson_bwd)
```

$\partial E/\partial M$ が計算できるか確認してみよう。

```
from jax import jit, vmap

dE_de = grad(eccentric_anomaly_newton_raphson, argnums=0)
dE_dM = grad(eccentric_anomaly_newton_raphson, argnums=1)
vmap_eccentric_anomaly = jit(vmap(eccentric_anomaly_newton_raphson, in_axes=(None, 0, 0)))
vmap_dE_de = jit(vmap(dE_de, in_axes=(None, 0, 0)))
```

```
vmap_dE_dM = jit(vmap(dE_dM, in_axes=(None, 0, 0)))

etest = 0.9
M_values = jnp.linspace(0, 2*jnp.pi-0.001, 100)
E_stars = vmap_eccentric_anomaly(etest, M_values, M_values)
dE_stardM = vmap_dE_dM(etest, M_values, M_values)
```

プロットしてみると、ちゃんと微分が計算できている。

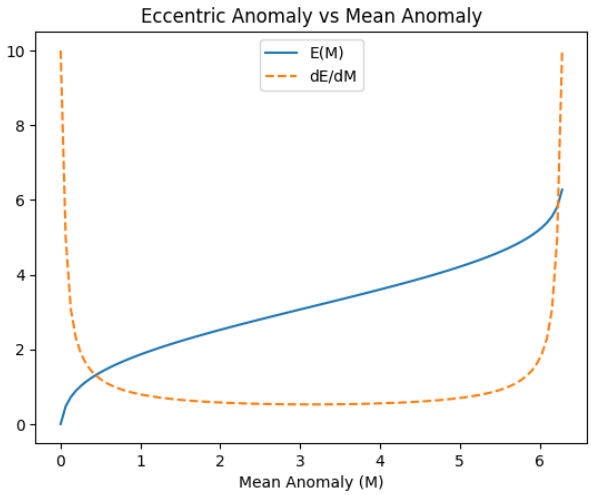


図3.8　$E(M)$ および $\partial E/\partial M$。

第 4 章

JAX プログラミングの様々なテクニック

4.1　XLA の再コンパイルを防ぐ

JAX/XLA は、実行時にコンパイルを行う。このコンパイルは通常、最初の一回のみであるが、関数のハッシュ値が変わると再コンパイルが行われてしまう。これは

- 同じ関数を繰り返し使用する
- 関数実行時間に対してコンパイルの時間が同程度か長い

という二つの条件が満たされると、再コンパイルの時間が実行時間を律速しうる（図4.1）。このような状況が起きているかどうかは、prefetto のようなメモリプロファイラーを用いれば分かる。対策としては jit や scan, vmap の位置を変更するなどがあるが現状手探りで探すしかない。

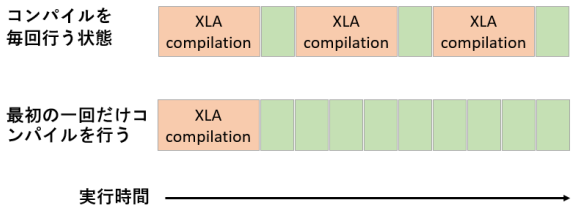


図4.1　コンパイルの実行時間。

4.2　reverse-mode 微分と checkpoint

checkpoint を置くことで、その時点での reverse-mode の前進方向のメモリを開放することができる。ややこしいのでまた後で書く。

4.3　jax.numpy.array の範囲外アサイン

python の numpy.array は未定義のインデックスにアクセスしようとすると IndexError が出される。しかし jax.numpy.array ではこのようなことは起きず、なんらかの値が返されてしまう。直接、範囲外のインデックスを入れると、通常は最終の要素を返すようだが、これも厳密には未定義動作である。一方、jnp.take を用

いると、まったく関係ない値が返される。つまり、JAX ではあらかじめインデックスが有効範囲内にあることをチェックすることが必要である。

例えば

```
import jax.numpy as jnp
a = jnp.array([1,2,3])
a[0],a[1],a[2],a[3],a[4]
```

では

```
(Array(1, dtype=int32),
 Array(2, dtype=int32),
 Array(3, dtype=int32),
 Array(3, dtype=int32),
 Array(3, dtype=int32))
```

が返ってくる（おそらく保証はされていない）。一方、

```
jnp.take(a, 3)
```

では、

```
Array(-2147483648, dtype=int32)
```

のようなとんでもない値が返ってくる！

4.4 jax.numpy.array のインデックス操作

jax.numpy は直接のインデックスアサインはできない。以下のコードでは 5 成分のゼロアレイを用意して、インデックス 1,2 にそれぞれ 10,3 を代入しようとしている。しかし、最後の行はエラーになる。

```
a=jnp.zeros(5)
indx=jnp.array([1,2])
val=jnp.array([10.0,3.0])
a[indx]=val #エラー
```

代わりに jax.numpy.at.set を用いることができる。

```
a=a.at[indx].set(val)
# DeviceArray([ 0., 10.,  3.,  0.,  0.], dtype=float32)
```

とすればよい。

ここですでにある配列のインデックスアサインをして足すには at.add を用いる。

```
a.at[jnp.array([0,1])].add(jnp.array([1.,2.]))
# DeviceArray([ 1., 12.,  3.,  0.,  0.], dtype=float32)
```

注意：jax.opt.index などの関数は廃止されました。

4.5 scan 中のインデクシング

dynamic_slice は jax.array の一部分をスライスするのに用いられる。

```
from jax.lax import scan
from jax.lax import dynamic_slice
import jax.numpy as jnp

N = 30
M = 10
a = jnp.arange(N)

print(dynamic_slice(a, (10,), (M,)))
```

は

```
[10 11 12 13 14 15 16 17 18 19]
```

を返してくる。

この dynamic_slice を用いて、scan の中で slice することができる。ここでは配列 a を順にスライスしたものに c[i] をかけて、append したものを得る例をあげる

```
c = jnp.array([1,3,7])

def scan_slice():

    def f(icarry,factor):
        element = dynamic_slice(a, (icarry,), (M,))
        return icarry+M, element*factor

    _, result = scan(f, 0, c)

    return result
```

この関数を呼ぶと

```
[[  0   1   2   3   4   5   6   7   8   9]
 [ 30  33  36  39  42  45  48  51  54  57]
 [140 147 154 161 168 175 182 189 196 203]]
```

を返す。

第 5 章

JAX で最適化

ちょっと古いです。**JAXOpt** は冷温停止状態で、**Optax** に統合されるようです。今後更新予定。

5.1 jax.experimental

最適化は自動微分の代表的な応用先である。多くの最適化手法はもとを正せば Gradient Descent

$$\boldsymbol{r}^{(k)} = \boldsymbol{r}^{(k-1)} - \gamma \frac{\partial}{\partial \boldsymbol{r}} f(\boldsymbol{r}) \tag{5.1}$$

に似た方法で更新していくからであり、つまり目的関数の微分 $f'(\boldsymbol{r})$ が必要だからである。つまり自動微分が通るように構成しておけば、自分で微分を計算する必要はない。

5.1.1 JAX の optimizer で最適化を楽しむ

JAX には jax.experimental.optimizer という最適化モジュールがある。有名どころの optimizer はここのものを使って手軽に最適化できる。例として Booth function

$$f(x, y) = (x + 2y - 7)^2 + (2x + y - 5)^2 \tag{5.2}$$

を目的関数として、これを最小化する (x, y) を求めてみよう。

```
#Booth function
def booth(x,y):
    f=(x+2.*y-7.)**2 + (2.*x+y-5.)**2
    return f
```

```
# 目的関数
def objective(r):
    f=booth(r[0],r[1])
    return f
```

jax.experimetnal.optimizer は opt_init (初期化), opt_update（アップデート）, get_params（パラメタをゲットする）の 3 セットを用いて最適化を行う。ステップを以下のうように定義する。

```
from jax import value_and_grad
def step(t, opt_state):
    value, grads = value_and_grad(objective)(get_params(opt_state))
    opt_state = opt_update(t, grads, opt_state)
    return value, opt_state
```

value_and_grad(f)(p) は引数 $x = p$ での関数 $f(x)$ の値と微分値 $f'(x)$ を返す。t は step index（整数値）である。微分値と step index、前回の状態 (opt_state) を opt_update に渡すことで、次のステップにアップデートする。初期値を r0=(x0,y0) として、Nstep 回ステップを回して、途中結果を保存するには

```
def doopt(r0,opt_init,get_params,Nstep):
    opt_state = opt_init(r0)
    xtraj=[r0[0]]
    ytraj=[r0[1]]

    for t in range(Nstep):
        value, opt_state = step(t, opt_state)
        p=get_params(opt_state)
        xtraj.append(p[0])
        ytraj.append(p[1])
    return xtraj, ytraj, p
```

のようにすればよい。optimizer としては ADAM と SGD を試す。

```
from jax.experimental import optimizers
#ADAM
opt_init, opt_update, get_params = optimizers.adam(1e0)
r0 = jnp.array([7.5,-7.5]) #初期値
xtraj1, ytraj1, p=doopt(r0,opt_init,opt_update,get_params,100)
```

もしくは

```
#SGD
opt_init, opt_update, get_params = optimizers.sgd(1e-1)
r0 = jnp.array([7.5,-7.5])
xtraj2, ytraj2, p=doopt(r0,opt_init,get_params,100)
```

のような感じである。実行結果を表示しよう。

```
import matplotlib.pyplot as plt
import jax.numpy as jnp
```

```
import seaborn as sns
plt.style.use('bmh')

#grid for contour
Nx=1000;Ny=1000
x=jnp.linspace(-5,15,Nx)
y=jnp.linspace(-10,10,Ny)
g=booth(x[:,None]*jnp.ones((Nx,Ny)),y[None,:]*jnp.ones((Nx,Ny)))

fig=plt.figure(figsize=(6,6))
plt.plot(xtraj1,ytraj1,lw=0.75,c="C1",label="ADAM")
plt.plot(xtraj2,ytraj2,lw=0.75,c="C0",ls="dashed",label="SGD")
plt.contour(x,y,jnp.log(g.T),levels=100,cmap="gray",alpha=0.3)
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
```

結果は、図5.1のようになる。これは ADAM が SGD より悪いというわけではなく step_size を大きめ

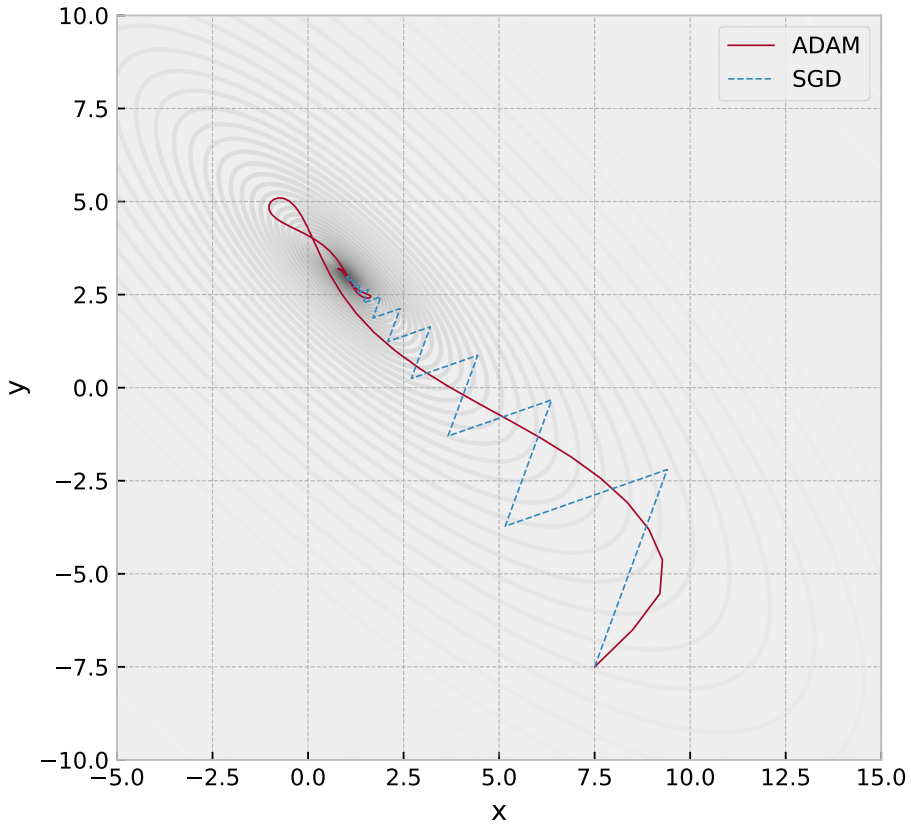


図5.1

に設定したので慣性が効いているのが見えているというわけである。Wikipedia に最適化のテスト関数(https://en.wikipedia.org/wiki/Test_functions_for_optimization)が多数挙げられているので試してみよう。全然、大域最小値に収束しないこと請け合いである。

5.2 JAXopt

JAXoptは JAX を用いた最適化のためのパッケージである。特徴として目を引くのは

- Differentiable: optimization problem solutions can be differentiated with respect to their inputs either implicitly or via autodiff of unrolled algorithm iterations.

すなわち、解自体を最適化問題のパラメタで微分できるというところにある。すばらしい。

以下では $f(x;a) = (x - \sin(a))^2$ を x について最適化してみよう。もちろん答えは $x_* = \sin(a) \equiv g(a)$ となるはずである。

```
import jaxopt
def f(x,a):
    return (x-jnp.sin(a))**2
gd = jaxopt.GradientDescent(fun=f, maxiter=500)
```

これだけである。解 $x_* = g(a)$ を以下のように定義すればよい。

```
def g(a):
    res = gd.run(init_params=np.random.normal(0.0,1.0), a=a)
    params, state = res
    return params
```

これで、

```
g(0.1) #-> DeviceArray(0.09983341, dtype=float32)
```

となり解が出る。さらに $g(a)$ を a で微分することができる。

```
from jax import grad
dg=grad(g)
dg(-0.2) #-> DeviceArray(0.9800666, dtype=float32)
```

これは $\frac{\partial}{\partial a} x_{\min} = \frac{\partial}{\partial a} g(a) = \cos(a)$ のことであり、たしかにそうなっている。

第 6 章

pytree/tree-math (TBD)

まだ書いてない

tree_map

第 7 章

Numpyro でマルコフ鎖モンテカルロ・シミュレーションをする

NumPyro の MCMC は HMC-NUTS(Hamiltonian Monte Carlo – No–U–Turn Sampling) が基本である。HMC は勾配情報を用いた高受容率の MCMC であり、フィットするモデルのパラメタ数 D が多い場合にはランダムメトロポリス・ヘイスティング法より効率がよい。理論的には収束までの時間が HMC では $D^{5/4}$、ランダム MH では D^2 に比例するとされている [5]。ここでは HMC の説明はしないが、日本語では「ゼロからできるＭＣＭＣ」(KF 理工学専門書) が筆者のおすすめである。HMC では計算に Leap-frog を用い、モデルのパラメタによる微分が必要である。numpyro [6] を用いて HMC-NUTS をしよう。numpyro は JAX を用いた Probabilistic Programming Language である。つまり JAX の自動微分が通るようにモデルを定義しないとならない。

7.1　曲線フィット

正弦曲線から生成されるデータにガウスノイズが加わったものを考えよう。すなわち

$$y = \sin(x + \phi) + \epsilon \tag{7.1}$$
$$\epsilon \sim \mathcal{N}(0, \sigma) \tag{7.2}$$

のような場合である。ここに $\epsilon \sim p$ というのは確率変数 ϵ が確率密度 p から生成されるということを意味している。また $\mathcal{N}(\mu, \sigma)$ は平均 μ、標準偏差 σ の正規分布のことである。さてデータから位相 ϕ とノイズレベル σ を推定したい。まず模擬データを以下のように生成しておく。

```
import numpy as np
np.random.seed(32)
phase=0.5
sigin=0.3
N=20
x=np.sort(np.random.rand(N))*4*np.pi
y=np.sin(x+phase)+np.random.normal(0,sigin,size=N)
```

このデータ (x,y) は図7.1のような感じとなっている。

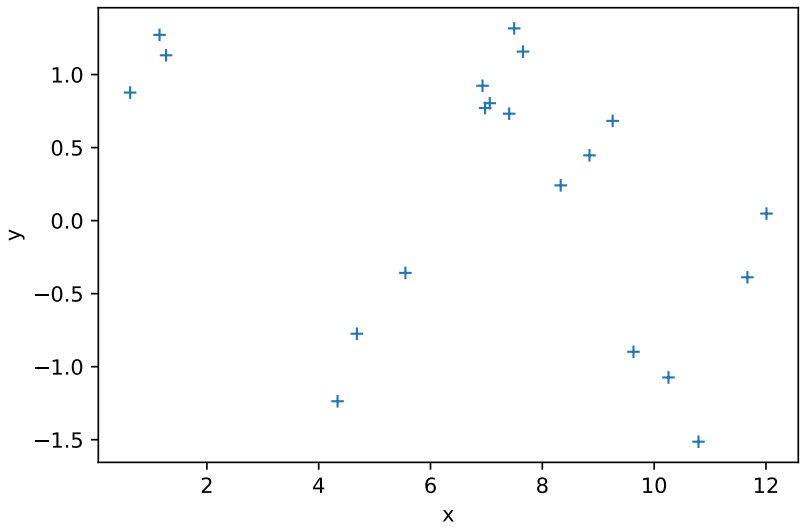


図7.1

さてこのデータに対し確率モデルを設定し HMC-NUTS で事後分布を求めよう。確率モデルとしては式 (7.1,7.2) また事前分布として

$$\phi \sim \mathcal{U}[-\pi, \pi] \tag{7.3}$$
$$\sigma \sim \text{Exponential}(1) \tag{7.4}$$

としたものを用いる。この確率モデルを numpyro を使って以下のように定義する。

```
import jax.numpy as jnp
import numpyro
import numpyro.distributions as dist

def model(x,y):
    phase = numpyro.sample('phase', dist.Uniform(-1.0*jnp.pi, 1.0*jnp.pi))
    sigma = numpyro.sample('sigma', dist.Exponential(1.))
    mu=jnp.sin(x+phase)
    numpyro.sample('y', dist.Normal(mu, sigma), obs=y)
```

この例からわかるように、確率分布は dist で指定されたものを、numpyro の sample 内で定義することにより確率変数として扱われる。最後のデータ y の部分は式 (7.1,7.2) が

$$y \sim \mathcal{N}(\sin(x+\phi), \sigma) \tag{7.5}$$

と書き直されることよりわかる。さてこれを numpyro の HMC-NUTS にかける。

```
from jax import random
```

```
from numpyro.infer import MCMC, NUTS
rng_key = random.PRNGKey(0)
rng_key, rng_key_ = random.split(rng_key)
num_warmup, num_samples = 1000, 2000
```

HMC-NUTS は最初にウォームアップで調整する。ウォームアップのチェーンの数と本番のチェーンの数がそれぞれ num_warmup, num_samples = 1000, 2000 で指定されている。さて MCMC を実行する。

```
kernel = NUTS(model)
mcmc = MCMC(kernel, num_warmup=num_warmup, num_samples=num_samples)
mcmc.run(rng_key_, x=x, y=y)
mcmc.print_summary()
```

最後の print_summary() で以下のような事後分布サンプリングの要約が出力されたと思う。成功だ。

```
mean std median 5.0% 95.0% n_eff r_hat
phase 0.34 0.12 0.34 0.15 0.56 1363.51 1.00
sigma 0.34 0.06 0.33 0.24 0.42 1206.21 1.00
Number of divergences: 0
```

以下では結果のチェックに arviz を用いよう。MCMC は結果の健全性のチェックが重要である。arviz は標準的なチェックが兼ね備えられているので便利である。

```
import arviz
arviz.plot_trace(mcmc, var_names=["phase","sigma"])
```

上では図7.2のように、パラメタの周辺分布（左）とチェーンが表示される。

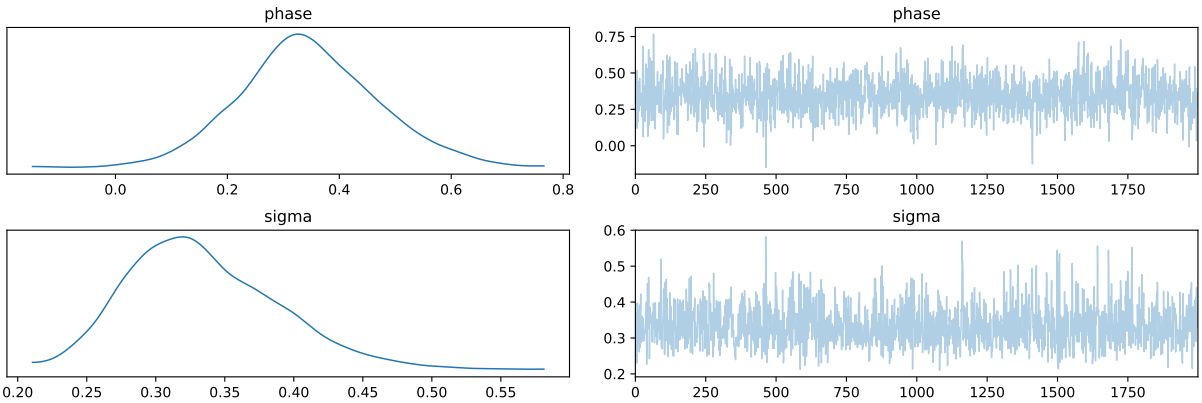


図7.2

いわゆるコーナープロットで事後分布の同時確率密度をチェックしよう[*1]。

[*1] 2020/12 月現在 arviz の develop 版を用いている。

```
refs={};refs["sigma"]=sigin;refs["phase"]=phase
arviz.plot_pair(arviz.from_numpyro(mcmc),kind='kde',
    divergences=False,marginals=True,reference_values=refs,
    reference_values_kwargs={'color':"red", "marker":"o", "markersize":12})
```

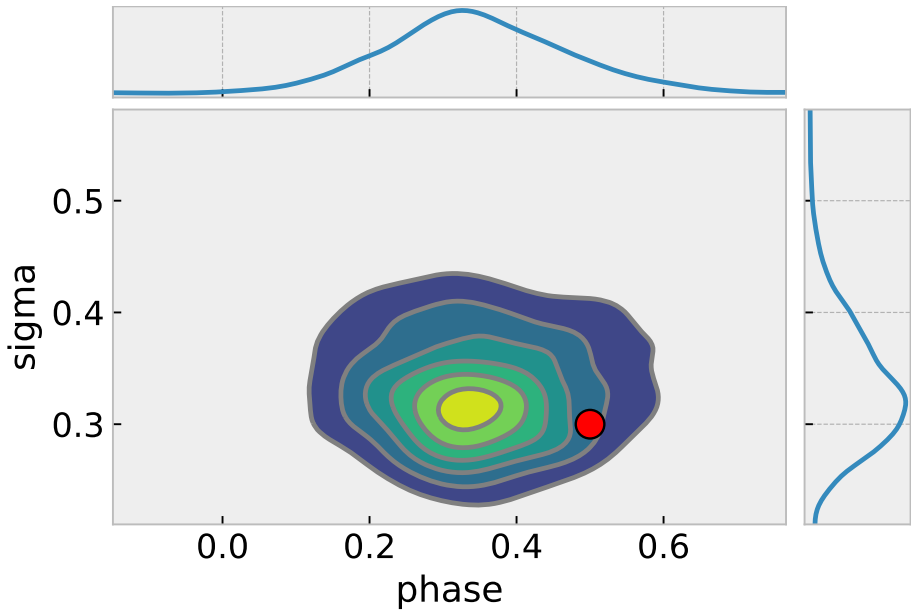


図7.3　赤い点は refs で指定されるインプットの値。

位相 ϕ および σ のサンプリングは

```
posterior_phase = mcmc.get_samples()['phase']
posterior_sigma = mcmc.get_samples()['sigma']
```

のようにして得ることができる。この事後分布サンプリングを用いて、予測分布のサンプリングをつくろう。numpyro.infer の Predictive を用いることで簡単に予測分布をサンプルできる。

```
from numpyro.infer import Predictive
pred = Predictive(model,{'phase':posterior_phase,'sigma':posterior_sigma},
       return_sites=["y"])
x_ = jnp.linspace(0,4*jnp.pi,1000)
predictions = pred(rng_key_,x=x_,y=None)
```

信頼分布として Highest posterior density interval(HPDI) を 90% 区間で計算しよう。

```
from numpyro.diagnostics import hpdi
mean_muy = jnp.mean(predictions["y"], axis=0)
```

```
hpdi_muy = hpdi(predictions["y"], 0.9)
```

最後にデータとともにプロットしよう。もちろんプロットの見た目を良くするだけの seaborn は使わなくても良い。結果は図7.4である。

```
import seaborn as sns
plt.style.use('bmh')

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6, 3))
ax.plot(x,y,"+",color="black")
ax.plot(x_,mean_muy,color="C0")
ax.fill_between(x_, hpdi_muy[0], hpdi_muy[1], alpha=0.3, interpolate=True,color="C0")
plt.xlabel("x")
plt.ylabel("y")
```

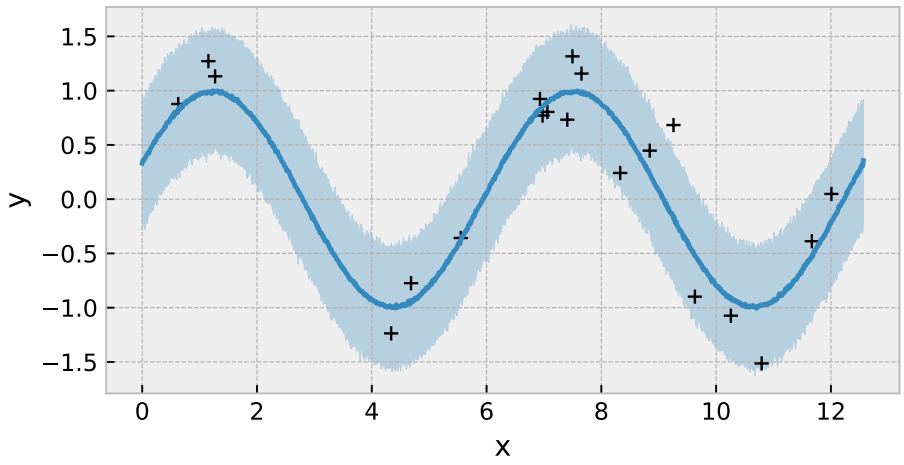


図7.4 。

モデルに対する Prior のサンプリングを確認する

handlers を用いるとモデルからサンプリングできる。ただし、インプットに y は必要ないことにに注意。

```
def modelc(x):
    phase = numpyro.sample('phase', dist.Uniform(-1.0*jnp.pi, 1.0*jnp.pi))
    sigma = numpyro.sample('sigma', dist.Exponential(1.))
    mu=jnp.sin(x+phase)
    d=numpyro.sample('y', dist.Normal(mu, sigma))
```

```
    return d
```

これを handlers.seed を用いてサンプリングできる。

```
from numpyro import handlers
d = handlers.seed(modelc, rng_seed=9)(x)
```

対数尤度からサンプリングする

上の例では model(x,y) の中で numpyro.sample を用いて、データの確率分布を直接指定した。しかし場合によっては対数尤度からサンプリングしたいこともある。この場合、

```
def model(x,y):
    phase = numpyro.sample('phase', dist.Uniform(-1.0*jnp.pi, 1.0*jnp.pi))
    sigma = numpyro.sample('sigma', dist.Exponential(1.))
    mu=jnp.sin(x+phase)
    loglikelihood=-0.5*(y-mu)**2/sigma**2-jnp.log(sigma)
    numpyro.factor("loglike", loglikelihood)
```

のようにして指定することもできる。

7.1.1 複数の種類のデータがある場合

共通のパラメタを変数にもつ２つの種類のデータがある場合を考えよう。例として

$$y_1 = \sin(x+\phi) + \epsilon \tag{7.6}$$
$$y_2 = \cos(x+\phi) + \epsilon \tag{7.7}$$
$$\epsilon \sim \mathcal{N}(0,\sigma) \tag{7.8}$$

という場合を考える。例えば図7.5のようなデータを解析する。

モデルは前節と同様に

$$y_1 \sim \mathcal{N}(\sin(x+\phi),\sigma) \tag{7.9}$$
$$y_2 \sim \mathcal{N}(\cos(x+\phi),\sigma) \tag{7.10}$$

と考えることができるので、numpyro のモデルは以下のように書ける。

```
def model(x1,x2,y1,y2):
    phase = numpyro.sample('phase', dist.Uniform(-1.0*jnp.pi, 1.0*jnp.pi))
    sigma = numpyro.sample('sigma', dist.Exponential(1.))
    mu1=jnp.sin(x1+phase)
    mu2=jnp.cos(x2+phase)
    numpyro.sample('y1', dist.Normal(mu1, sigma), obs=y1)
    numpyro.sample('y2', dist.Normal(mu2, sigma), obs=y2)
```

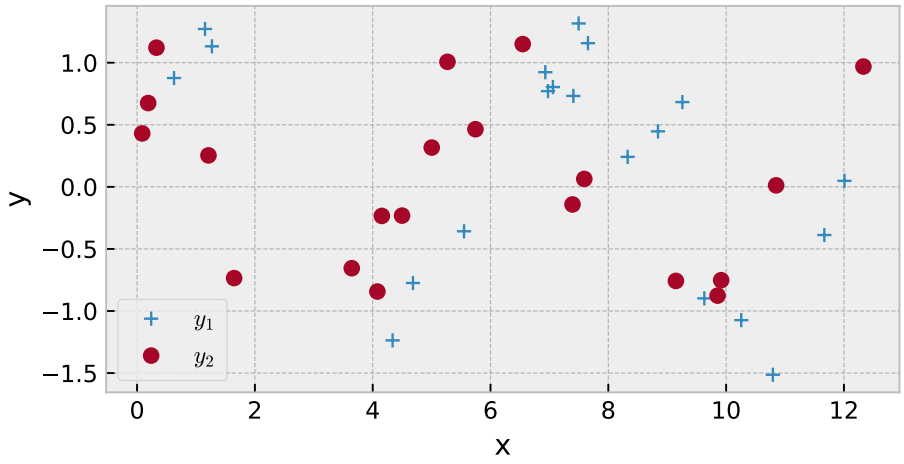


図7.5　。

あとはほとんど前節と同じだが、MCMC 実行時に

```
mcmc.run(rng_key_, x1=x1, x2=x2, y1=y1, y2=y2)
```

のように変数の数を合わせるのと、予測分布を描くときにも以下のように 2 変数にする注意が必要である。

```
pred = Predictive(model,{'phase':posterior_phase,'sigma':posterior_sigma}
       ,return_sites=["y1","y2"])
x1_ = jnp.linspace(0,4*jnp.pi,1000)
x2_ = jnp.linspace(0,4*jnp.pi,1000)
predictions = pred(rng_key_,x1=x1_,x2=x2_,y1=None,y2=None)
```

予測からもそれぞれ平均・HPDI を計算することにより、

```
mean_muy1 = jnp.mean(predictions["y1"], axis=0)
hpdi_muy1 = hpdi(predictions["y1"], 0.9)
mean_muy2 = jnp.mean(predictions["y2"], axis=0)
hpdi_muy2 = hpdi(predictions["y2"], 0.9)
```

最終的に、以下のようなプロットで

```
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6, 3))
ax.plot(x1,y1,"+",color="C0")
ax.plot(x1_,mean_muy1,color="C0")
ax.fill_between(x1_, hpdi_muy1[0], hpdi_muy1[1], alpha=0.3,
                interpolate=True,color="C0")
ax.plot(x2,y2,"o",color="C1")
```

```
ax.plot(x2_,mean_muy2,color="C1")
ax.fill_between(x2_, hpdi_muy2[0], hpdi_muy2[1], alpha=0.3,
                interpolate=True,color="C1")
plt.xlabel("x")
plt.ylabel("y")
```

図7.6のように同時フィットすることができる。

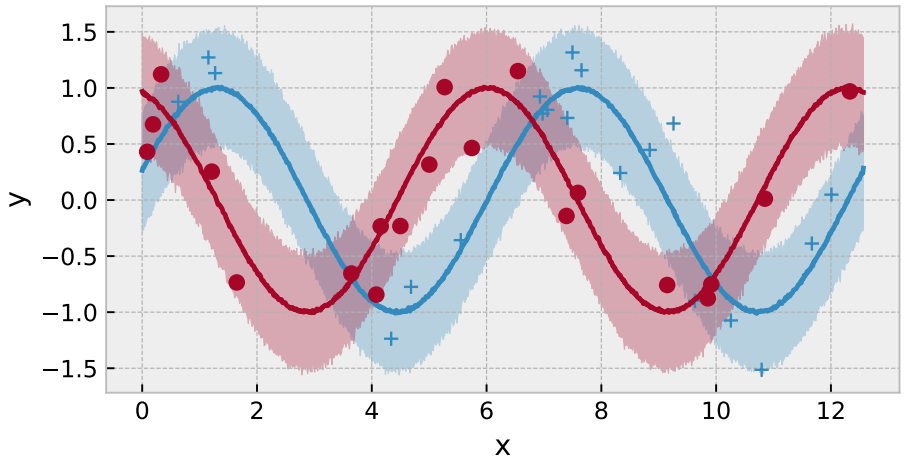


図7.6　。

7.1.2　モデル内の関数がグリッドデータで与えられている時

世の中はモデルは往々にして美しい数学関数だけで記述されるわけではない。複雑なモデルになるほど、実験データからもとまる関係などが入ってくる。このような場合でも補間関数を用いることによって対処できる。一例として、図7.7の上パネルのようなデータ点（黒点）からモデルを構成することを考えよう。

このデータ点 xa, ya を jax.numpy.interp で補完した関数を以下のように定義する。

```
def f(x):
    return jnp.interp(x,xa,ya)
```

すると、$f(x)$ は図7.7の上パネル実線のように、データのある部分は良く補間されているのがわかる。さてこの f(x) に対しても自動微分は通るので

```
from jax import grad,vmap
g=vmap(grad(f))
```

のように定義すれば、微分を計算できる。図7.7のしたパネルはそれを表示したもので、具体的には

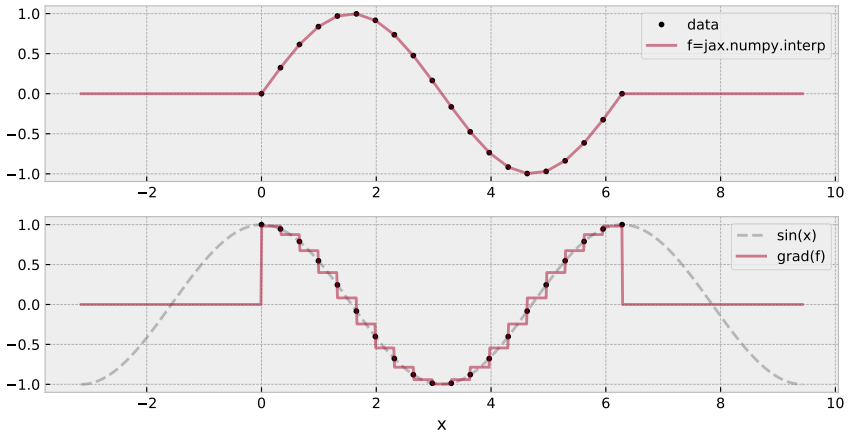


図7.7　。

```
xx=np.linspace(-np.pi,3*np.pi,1000)
plt.plot(xx,np.cos(xx),ls="dashed",label="sin(x)",color="gray",alpha=0.5)
plt.plot(x,np.cos(x),".",color="black")
plt.plot(xx,g(xx),label="grad(f)",color="C1",alpha=0.5)
plt.legend()
plt.xlabel("x")
```

で、作成されている。そう、実は（というほどでもないが）このデータ点は $\sin(x)$ から作成されているのだが、そのことは知らないとして話を進めよう。補間関数の自動微分は、データ点の離散化に対応してカクカクしているものの、おおむね真のモデル $\partial_x \sin x = \cos x$ を再現している。

さて補間関数 $f(x)$ を用いて以下のようにモデルを構成してみよう。

$$y = Af(x-c) + \epsilon \tag{7.11}$$
$$\epsilon \sim \mathcal{N}(0, \sigma) \tag{7.12}$$

上のモデルに $c = 0.3, \sigma = 0.5, A = 10.0$ を用いて作成した模擬データを用いる。

```
xb=np.linspace(0.5,1.7,10)
c=0.3
sigin=0.5
A=10.0
y=A*np.sin(xb+c)+np.random.normal(0,sigin,size=len(xb))
```

したがって HMC-NUTS のためのモデルは以下のようになる。

```
import numpyro
```

```
import numpyro.distributions as dist

def model(x,y):
    c = numpyro.sample('c', dist.Uniform(-0.5, 0.5))
    A = numpyro.sample('A', dist.Uniform(0.1,100))
    sigma = numpyro.sample('sigma', dist.Exponential(1.0))
    mu=A*f(x+c)
    numpyro.sample('y', dist.Normal(mu, sigma), obs=y)
```

これで、HMC-NUTS をいつものように回すと、ちゃんと推定ができる。今回は図7.8に credible interval だけ示そう。

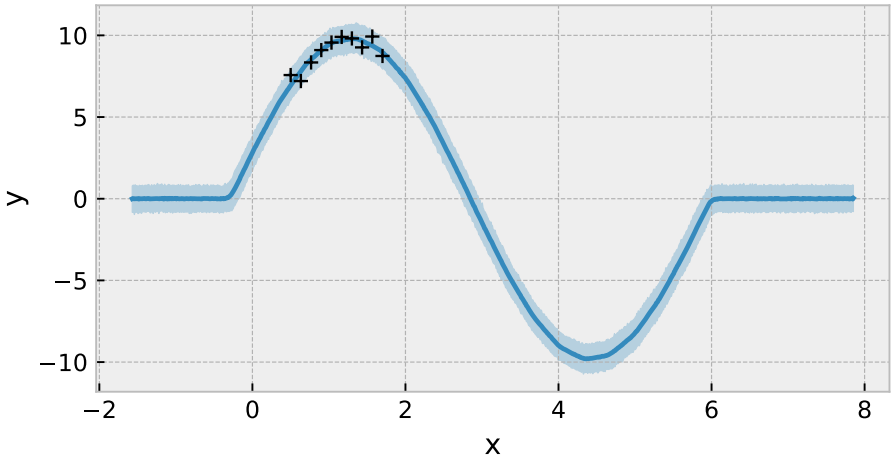


図7.8　。

7.1.3　畳み込みの幅を推定する

次の例は、畳み込みの幅を推定する問題である。これは例えばあるシャープなシグナルが、なんらかの過程でなまされてしまったものが観測されるときなどに有用である。典型的なのは観測装置由来のレスポンスである。このような場合 jax.numpy.convolve を用いることができる。

レスポンスのカーネルとして次のようなガウス関数型を考えよう。なまされる幅は σ =sigk でパラメトライズされるとする。図7.9に、なまされかたの違いを示す。

さてデータとしてはデルタ関数的なピークが何本かたっているもの (yp) を用意して、これをカーネルでなましたものを用いる。HMC 推定としては以下のように構成できる。

```
Nk=100
xkx=jnp.linspace(-0.1,0.1,Nk)

```

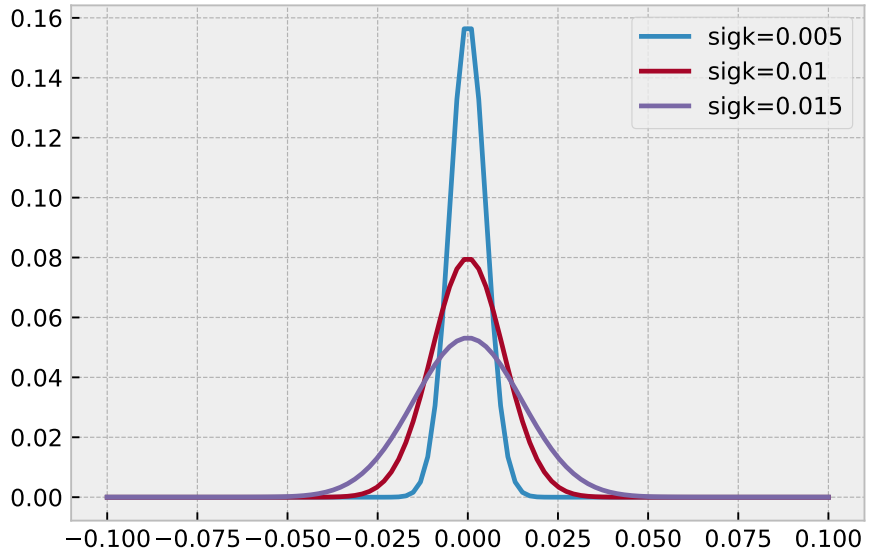


図7.9　。

```
def model(x,y):
    sigk = numpyro.sample('sigk', dist.Exponential(1.))
    sigma = numpyro.sample('sigma', dist.Exponential(1.))
    mu=jnp.convolve(yp,jnp.exp(-xkx**2/(2.*sigk**2)) \
                    /jnp.sqrt(2.*jnp.pi)/sigk*(0.2/Nk),"same")
    numpyro.sample('y', dist.Normal(mu, sigma), obs=y)
```

結果は図7.10の通り。

7.2　カスタマイズした自動微分で numpyro を動かす

Reverse mode

NumPyro の HMC NUTS はデフォルトでは reverse mode を使用する。そこでカスタマイズした自動微分を用いるためには第??章で説明した VJP を定義したものを用いればよい。ここでは $f(x,A) = A\sin x$ の元に、$y_i = f(x_i, A) + \epsilon$、および $\epsilon \sim \mathcal{N}(0,\sigma)$ というモデルで $\boldsymbol{y} = \{y_i\}$ をフィットしよう。VJP の定義は

```
@custom_vjp
def h(x,A):
    return A*jnp.sin(x)

def h_fwd(x, A):
```

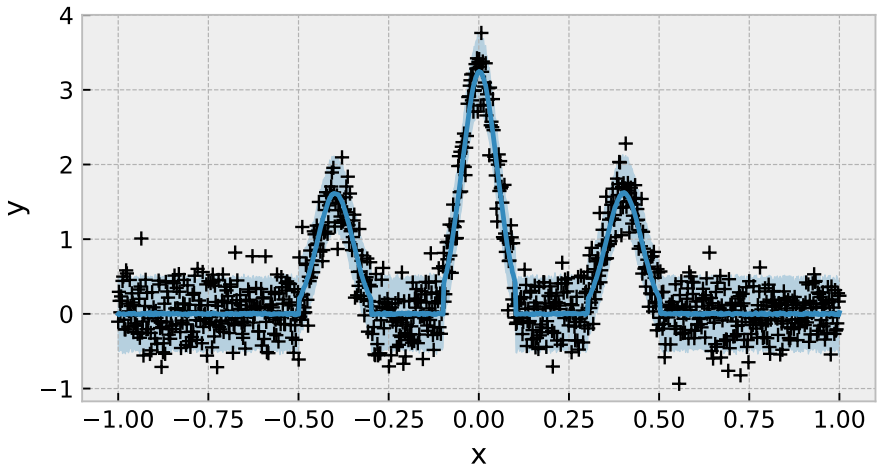


図7.10　。

```
    res = (A*jnp.cos(x), jnp.sin(x))
    return h(x,A), res

def h_bwd(res, u):
    A_cos_x, sin_x = res
    return (A_cos_x * u, sin_x * u)

h.defvjp(h_fwd, h_bwd)
```

となる。モデルは

```
def model(x,y):
    sigma = numpyro.sample('sigma', dist.Exponential(1.))
    x0 = numpyro.sample('x0', dist.Uniform(-1.,1.))
    A = numpyro.sample('A', dist.Exponential(1.))
    hv=vmap(h,(0,None),0)
    mu=hv(x-x0,A)
    numpyro.sample('y', dist.Normal(mu, sigma), obs=y)
```

とすればよい。ここに vmap を用いて $\boldsymbol{x}$ に対してベクトル化していることに注意。HMC-NUTS は

```
rng_key = random.PRNGKey(0)
rng_key, rng_key_ = random.split(rng_key)
num_warmup, num_samples = 1000, 2000
kernel = NUTS(model)
```

```
mcmc = MCMC(kernel, num_warmup, num_samples)
mcmc.run(rng_key_, x=x, y=data)
mcmc.print_summary()
```

とすればよい。

Forward mode

NumPyro 0.5.0 より forward mode での HMC NUTS も動作するようになった。

第 8 章

NumPyro で楽しむガウス過程

8.1　ガウス過程

ガウス過程は、多変数正規分布

$$\boldsymbol{d} \sim \mathcal{N}(\mathbf{0}, \Sigma) \tag{8.1}$$

に従う確率変数 $\boldsymbol{x}$ の確率過程のことである。いま d_i は時系列だとして、この多変数正規分布の共分散行列の各成分が、

$$\Sigma_{ij} = K_{ij}(a, \tau) = ak(|t_i - t_j|; \tau) \tag{8.2}$$

のように、t_i と t_j の差の絶対値の関数として与えられれば、相関長 τ のガウス過程が得られる。カーネル関数 $k(t, \tau)$ としてはいろいろなタイプがありうるが、例えば RBF カーネル

$$k_{\mathrm{RBF}}(t; \tau) = \exp\left(-\frac{t^2}{2\tau^2}\right), \tag{8.3}$$

と Matérn 3/2 カーネル

$$k_{\mathrm{M3/2}}(t; \tau) = \left(1 + \frac{\sqrt{3}t}{\tau}\right) e^{-\sqrt{3}t/\tau}. \tag{8.4}$$

などがある。

とりあえず論より実装で、例を作ってみよう。上記カーネルは

```
import numpy as np
def RBF(t,tau):
    Dt = t - np.array([t]).T
    K=np.exp(-(Dt)**2/2/(tau**2))
    return K

def Matern32(t,tau):
    Dt = t - np.array([t]).T
    fac=np.sqrt(3.0)*np.abs(Dt)/tau
```

```
    K=(1.0+fac)*np.exp(-fac)
    return K
```

のような感じである。さて、scipy.stats.multivariate_normal で多変数正規分布がサンプリングできる[*1]。

```
from scipy.stats import multivariate_normal as smn
np.random.seed(seed=1) #乱数初期値
N = 101
t = np.linspace(0,10,N)
ave = np.zeros(N)
tau = 0.4
a=1.0
ave = jnp.zeros(N)
cov = a*RBF(t,tau) #cov = Matern32(t,tau) でもよい
di = smn(mean=ave ,cov=cov , allow_singular =True).rvs(1).T
```

さて、このように生成されたデータに平均ゼロ、標準偏差 σ のガウスノイズを足しておく。このノイズを便宜的に観測ノイズと呼んでおこう。

```
sigma=0.6
d=di+np.random.normal(0.0,sigma,len(di))
```

図8.1が生成されたデータである。

さてこのデータを元に τ、a、σ を推定してみよう。平均ゼロ、標準偏差 σ のガウスノイズの観測ノイズを足したということは確率モデルとしては、

$$\boldsymbol{d} \sim \mathcal{N}(\mathbf{0}, \Sigma') \tag{8.5}$$
$$\Sigma' = K(a,\tau) + \sigma^2 I \tag{8.6}$$

となる。この Σ' を以下のように実装する。

```
def modelcov(t,tau,a,sigma):
    Dt = t - jnp.array([t]).T
    K=a*jnp.exp(-(Dt)**2/2/(tau**2))+jnp.eye(N)*sigma**2
    return K
```

numpyro のモデルはこの modelcov を用いて

```
import jax.numpy as jnp
import numpyro
import numpyro.distributions as dist
```

[*1] numpy.random や JAX.random、および dist.MultivariateNormal でもあるのだがとある理由 (9.1章参照) によりここでは scipy のものを用いる。

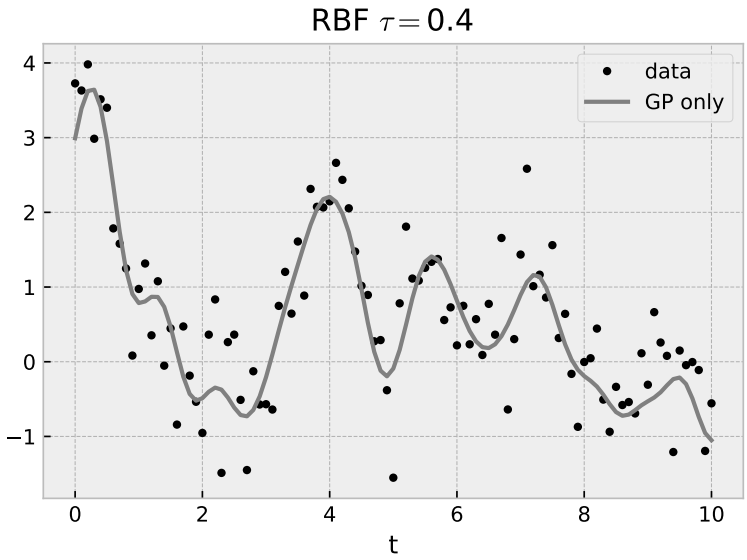


図8.1　実線は σ の観測ノイズなし。

```
4
5  def model(t,y):
6      sigma = numpyro.sample('sigma', dist.Exponential(1.))
7      tau = numpyro.sample('tau', dist.Exponential(1.))
8      a = numpyro.sample('a', dist.Exponential(1.))
9      cov = modelcov(t,tau,a,sigma)
10     numpyro.sample('y', dist.MultivariateNormal(loc=jnp.zeros(N),
11     covariance_matrix=cov), obs=y)
```

となる。ここでは τ と σ の事前分布を Exponential 分布とした。numpyro の HMS-NUTS を以下のように回す。

```
1  from numpyro.infer import MCMC, NUTS
2  from jax import random
3  rng_key = random.PRNGKey(0)
4  rng_key, rng_key_ = random.split(rng_key)
5  num_warmup, num_samples = 1000, 2000
6  # Run NUTS.
7  kernel = NUTS(model)
8  mcmc = MCMC(kernel, num_warmup, num_samples)
9  mcmc.run(rng_key_, t=t, y=d)
```

そして

```
import arviz
refs={};refs["sigma"]=sigma;refs["tau"]=tau;refs["a"]=a
arviz.plot_pair(arviz.from_numpyro(mcmc),kind='kde',
    divergences=False,marginals=True,reference_values=refs,
    reference_values_kwargs={'color':"red", "marker":"o", "markersize":12})
```

のように事後分布を可視化したものが図8.2である。ところで、式 (8.5) という観点のモデル化では、$\boldsymbol{x} = \mathbf{0}$

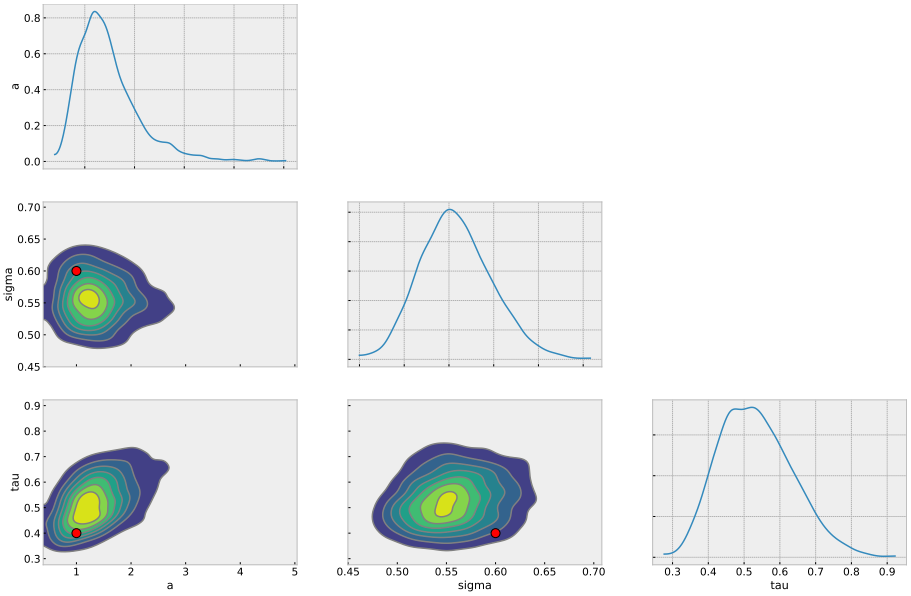


図8.2

がモデル平均である。これは第7章で行ったのと同様に credible interval を計算することでより明確になる。つまり

```
from numpyro.infer import Predictive
from numpyro.diagnostics import hpdi
# 事後分布のサンプリングを取り出す
posterior_a = mcmc.get_samples()['a']
posterior_tau = mcmc.get_samples()['tau']
posterior_sigma = mcmc.get_samples()['sigma']
#予測モデル
pred = Predictive(model,{'a':posterior_a,'tau':posterior_tau,
       'sigma':posterior_sigma},return_sites=["y"])
predictions = pred(rng_key_,t=t,y=None)
mean_muy = jnp.mean(predictions["y"], axis=0)
hpdi_muy = hpdi(predictions["y"], 0.9)
```

で予測サンプリングから hpdi を計算し、

```
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(6, 3))
ax.plot(t,d,".",color="black")
ax.plot(t_,mean_muy,color="C0")
ax.fill_between(t_, hpdi_muy[0], hpdi_muy[1], alpha=0.3, interpolate=True,color="C0")
plt.xlabel("t")
plt.ylabel("y")
```

のようにプロットすると、図8.3となり、直感的にわかる。

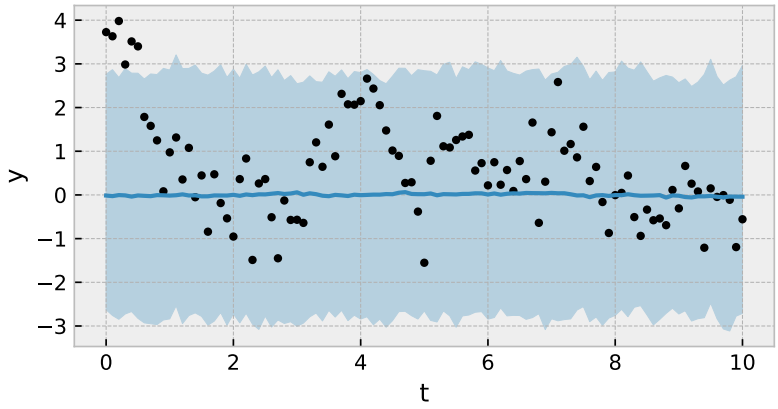


図8.3

モデルパラメタの事前分布としてのガウス過程

上記のガウス過程は、モデルパラメタ $\boldsymbol{m}$ の事前分布（プライア）として

$$p(\boldsymbol{m}) = \mathcal{N}(\boldsymbol{0}, \Sigma) \tag{8.7}$$

と置き、

$$d_i = m_i + \epsilon \tag{8.8}$$
$$\epsilon \sim \mathcal{N}(0, \sigma^2) \tag{8.9}$$

と見ることもできる。ここに ϵ は観測ノイズに対応している。もう少しまどろっこしく書くとモデル $\boldsymbol{g}$ を恒等変換として、

$$\boldsymbol{g}(\boldsymbol{m}) = \boldsymbol{m} \tag{8.10}$$
$$\boldsymbol{d} = \boldsymbol{g}(\boldsymbol{m}) + \boldsymbol{\epsilon} \tag{8.11}$$
$$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \sigma^2 I) \tag{8.12}$$

のように書ける。この場合、尤度関数が

$$p(\boldsymbol{d}|\boldsymbol{m}) = \mathcal{N}(\boldsymbol{d} - \boldsymbol{g}(\boldsymbol{m}), \sigma^2 I) = \mathcal{N}(\boldsymbol{d} - \boldsymbol{m}, \sigma^2 I) \tag{8.13}$$

となる。そして、さらにこの事前分布の中にパラメタがあるという構造となっている。つまり

$$\Sigma_{ij} = ak(|t_i - t_j|;\tau) \tag{8.14}$$

としていて、このパラメタ a,τ をハイパーパラメタという。そしてハイパーパラメタの事前分布 (超事前分布またはハイパープライア) をさらに仮定するということに相当する。

以上のモデルを、直接 numpyro のモデルとして構成し、$\boldsymbol{m}$ の事後分布をサンプリングすることもできるが、ハイパーパラメタを固定した状態では、$\boldsymbol{m}$ の事後分布を解析的に書けることが知られている。ここでガウス過程の計算法を紹介しよう。まず多変数正規分布に多変数正規分布をかけても多変数正規分布である。そこで計算しなくてはならないのは exp のべきの部分のみである。多変数正規分布の「べき」部分は

$$-2\log\mathcal{N}(\boldsymbol{m}|\boldsymbol{\mu},\Sigma) = (\boldsymbol{m}-\boldsymbol{\mu})^\top\Sigma^{-1}(\boldsymbol{m}-\boldsymbol{\mu}) = \boldsymbol{m}^\top\Sigma^{-1}\boldsymbol{m} - 2\boldsymbol{m}^T\Sigma^{-1}\boldsymbol{\mu} + \text{const.} \tag{8.15}$$

となっていることに注意すると、もしある多変数正規分布に従うことがわかっている確率密度分布 $p(\boldsymbol{m})$ が

$$-2\log p(\boldsymbol{m}) = \boldsymbol{m}^\top P\boldsymbol{m} - 2\boldsymbol{m}^\top\boldsymbol{q} + \text{const}, \tag{8.16}$$

のように書けたとすると、式 (8.15) と比べることで、この確率密度分布は

$$p(\boldsymbol{m}) = \mathcal{N}(\boldsymbol{m}|P^{-1}\boldsymbol{q}, P^{-1}). \tag{8.17}$$

であることがわかる。

さて、いま事後分布は

$$p(\boldsymbol{m}|\boldsymbol{d}) \propto p(\boldsymbol{d}|\boldsymbol{m})p(\boldsymbol{m}) = \mathcal{N}(\boldsymbol{d}-\boldsymbol{m},\sigma^2 I)\mathcal{N}(\boldsymbol{0},\Sigma) \tag{8.18}$$

であるので、この「べき」部分を $\boldsymbol{m}$ について展開することで、

$$-2\log\left[\mathcal{N}(\boldsymbol{d}-\boldsymbol{m},\sigma^2 I)\mathcal{N}(\boldsymbol{0},\Sigma)\right] = \boldsymbol{m}^\top(\Sigma^{-1}+\sigma^{-2}I)\boldsymbol{m} - 2\sigma^{-2}\boldsymbol{m}^\top\boldsymbol{d} + \text{const.} \tag{8.19}$$

と書けること、また

$$(\Sigma^{-1}+\sigma^{-2}I)^{-1} = \Sigma(I+\sigma^{-2}\Sigma)^{-1} \tag{8.20}$$

より

$$p(\boldsymbol{m}|\boldsymbol{d}) = \mathcal{N}(\Sigma(\sigma^2 I+\Sigma)^{-1}\boldsymbol{d}, \Sigma(I+\sigma^{-2}\Sigma)^{-1}) \tag{8.21}$$

であることがわかった。

さて、先程、行った HMC-NUTS による τ と σ のサンプリングがこの枠組みで何を意味するか考えよう。これらは今の枠組みではハイパーパラメタとみなせるので、まとめて $\boldsymbol{\theta} = (a,\tau,\sigma)^\top$ をハイパーパラメタとおく。ここからは $\boldsymbol{\theta}$ も確率に入れて考えることにする。さて、今、$\boldsymbol{m}$ について周辺化した尤度

$$p(\boldsymbol{d}|\boldsymbol{\theta}) = \frac{p(\boldsymbol{d}|\boldsymbol{m},\boldsymbol{\theta})p(\boldsymbol{m},\boldsymbol{\theta})}{p(\boldsymbol{m}|\boldsymbol{d},\boldsymbol{\theta})} \tag{8.22}$$

はやはり多変数正規分布となるが、また「べき」部分の $\boldsymbol{d}$ に関する部分だけ考えることにより

$$\begin{aligned}
-2\log p(\boldsymbol{d}|\boldsymbol{\theta}) &= -2\log p(\boldsymbol{d}|\boldsymbol{m},\boldsymbol{\theta}) + 2\log p(\boldsymbol{m}|\boldsymbol{d},\boldsymbol{\theta}) + \text{const} && (8.23)\\
&= -2\log\mathcal{N}(\boldsymbol{d}-\boldsymbol{m};\sigma^2 I) + 2\log\mathcal{N}(\Sigma(\sigma^2 I+\Sigma)^{-1}\boldsymbol{d}, \Sigma(I+\sigma^{-2}\Sigma)^{-1}) && (8.24)\\
&= \boldsymbol{d}^\top(\sigma^2 I+\Sigma)^{-1}\boldsymbol{d} + \text{const.} && (8.25)
\end{aligned}$$

である。よって

$$p(\boldsymbol{d}|\boldsymbol{\theta}) = \mathcal{N}(\mathbf{0}, \Sigma + \sigma^2 I) \tag{8.26}$$

となる。これは $\Sigma = K(\tau)$ とすれば、式 (8.5) と同じである。すなわち、（超）事前分布として $p(\boldsymbol{\theta})$ を与えた時の周辺化事後分布

$$p(\boldsymbol{\theta}|\boldsymbol{d}) \propto p(\boldsymbol{d}|\boldsymbol{\theta})p(\boldsymbol{\theta}) \tag{8.27}$$

を HMC-NUTS でサンプリングしていたことになり、図8.3はハイパーパラメタ τ、σ の周辺化事後分布を示していることがわかった。つまり $k = 0, ..., N_s - 1$ に対して

$$\boldsymbol{\theta}_k^\dagger \sim p(\boldsymbol{\theta}|\boldsymbol{d}) \propto p(\boldsymbol{d}|\boldsymbol{\theta})p(\boldsymbol{\theta}) \tag{8.28}$$

をサンプリングしたことになる。さて、ではこのサンプルと式 (8.21) を用いて、

$$p(\boldsymbol{m}, \boldsymbol{\theta}|\boldsymbol{d}) = p(\boldsymbol{m}|\boldsymbol{\theta}, \boldsymbol{d})p(\theta|\boldsymbol{d}) \tag{8.29}$$

であることから

$$p(\boldsymbol{m}|\boldsymbol{\theta}_k^\dagger, \boldsymbol{d}) = \mathcal{N}(\boldsymbol{\mu}_k, K_k) \tag{8.30}$$

$$\boldsymbol{\mu}_k = K(a_k^\dagger, \tau_k^\dagger)((\sigma_k^\dagger)^2 I + K(a_k^\dagger, \tau_k^\dagger))^{-1}\boldsymbol{d} \tag{8.31}$$

$$K_k = K(a_k^\dagger, \tau_k^\dagger)(I + (\sigma_k^\dagger)^{-2} K(a_k^\dagger, \tau_k^\dagger))^{-1} \tag{8.32}$$

からサンプルした $\boldsymbol{m}_k^\dagger$ と $\boldsymbol{\theta}_k^\dagger$ のセットは

$$(\boldsymbol{m}_k^\dagger, \boldsymbol{\theta}_k^\dagger) \sim p(\boldsymbol{m}, \boldsymbol{\theta}|\boldsymbol{d}) \tag{8.33}$$

とみなせることがわかる[*2]。式 (8.31)、(8.32) を以下のように実装する。

```
def muGP(tau,a,sigma):
    cov = a*RBF(t,tau)
    IKw=sigma**2*np.eye(N)+cov
    A=scipy.linalg.solve(IKw,d,assume_a="pos")
    return cov@A

def covGP(tau,a,sigma):
    cov = a*RBF(t,tau)
    IKw=np.eye(N)+cov/sigma**2
    IKw=scipy.linalg.inv(IKw)
    return cov@IKw
```

のように定義する。次に式 (8.30) からの $\boldsymbol{m}$ のサンプリングを

[*2] ハイパーパラメタの点推定（いわゆる maximum marginal likelihood=maximum evidence）を用いる方法については解説が多くあるが、周辺化事後分布から再サンプリングする議論についてはいまのところ文献が見つからない。ベイズ線形問題＋ガウス過程の枠組みで以前、筆者が議論したことがあるので参考までに挙げておく [4]。

```
import tqdm
Ns=len(posterior_sigma)
np.random.seed(seed=1)
marr=[]
for i in tqdm.tqdm(range(0,Ns)):
    sigmas=float(posterior_sigma[i])
    taus=float(posterior_tau[i])
    a_s=float(posterior_a[i])
    ave=muGP(taus,a_s,sigmas)
    cov=covGP(taus,a_s,sigmas)
    mk = smn(mean=ave ,cov=cov , allow_singular =True).rvs(1).T
    marr.append(mk)
marr=np.array(marr)
```

のように実行し、

```
mean_muy = np.mean(marr, axis=0)
hpdi_muy = hpdi(marr, 0.9)
```

HPDI を計算することで、図8.4の $\boldsymbol{m}$ の credible interval が求まる。ここで注意点としては、これはモデル $\boldsymbol{m}$ の 90% interval であり、観測ノイズ部分を含んだ $\boldsymbol{d}$ の予測ではないという点である。

```
fig=plt.figure(figsize=(6, 3))
ax=fig.add_subplot(111)
ax.plot(t,d,".",color="black")
ax.plot(t, mean_muy,color="C1")
ax.fill_between(t, hpdi_muy[0], hpdi_muy[1], alpha=0.3, interpolate=True,color="C1")
plt.xlabel("t")
plt.ylabel("y")
```

データ点にない位置の予測

さて、一般に $t = t^*$ での予測値はどうなるだろうか？以下では再度ハイパーパラメタを省略して表記する。まず、$\boldsymbol{m}$、$\boldsymbol{m}^*$ がガウス過程に従うとき

$$p(\boldsymbol{m}^*|\boldsymbol{m}) = \mathcal{N}(K_\times^\top K^{-1}\boldsymbol{m}, K_* - K_\times^\top K^{-1} K_\times) \tag{8.34}$$

である (Appendix A)。ここに

$$K_{ij} = ak(|t_i - t_j|;\tau) \tag{8.35}$$
$$(K_\times)_{ij} = ak(|t_i - t_j^*|;\tau) \tag{8.36}$$
$$(K_*)_{ij} = ak(|t_i^* - t_j^*|;\tau) \tag{8.37}$$

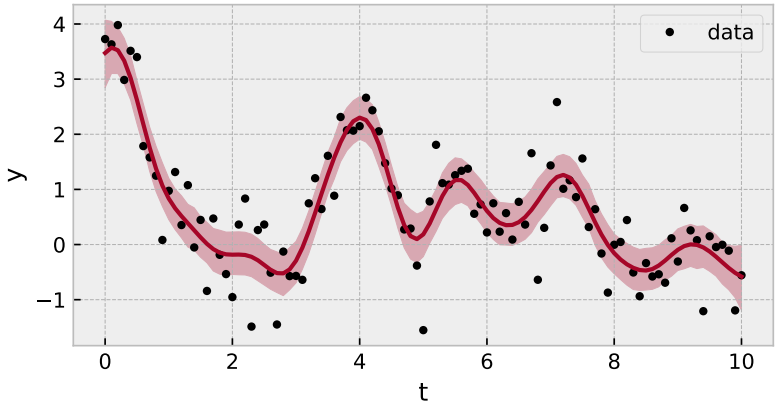


図8.4

同様に

$$p(\boldsymbol{m}^*|\boldsymbol{d}) = \mathcal{N}(K_\times^\top K_\sigma^{-1}\boldsymbol{d}, K_* - K_\times^\top K_\sigma^{-1}K_\times) \tag{8.38}$$

となる。ここに

$$(K_\sigma)_{ij} = ak(|t_i - t_j|;\tau) + \sigma^2\delta_{i,j} \tag{8.39}$$

($\delta_{i,j}$ はクロネッカーデルタ)。ここで $p(\boldsymbol{m}^*|\boldsymbol{d})$ はあくまでモデルパラメタとしてのガウス過程の事後分布なので、観測ノイズの分は考慮されていない。

もし観測ノイズを含んだ予測を行いたいのならば、

$$\boldsymbol{d}^* = \boldsymbol{m}^* + \boldsymbol{\epsilon} \tag{8.40}$$

というモデルに基づき、

$$p(\boldsymbol{d}^*|\boldsymbol{d}) = \mathcal{N}(K_\times^\top K_\sigma^{-1}\boldsymbol{d}, K_{*,\sigma} - K_\times^\top K_\sigma^{-1}K_\times) \tag{8.41}$$

となるだろう。ここに

$$(K_{*,\sigma})_{ij} = ak(|t_i^* - t_j^*|;\tau) + \sigma^2\delta_{i,j} \tag{8.42}$$

である。

というわけで、ハイパーパラメタ込みのサンプリングで式 (8.38,8.41) のサンプリングを行い Credible Interval を求める。

```
def mucovGPx(t,td,tau,a,sigma):
    cov = a*RBF(t,tau) + sigma**2*np.eye(N)
    covx= a*RBFx(t,td,tau)
    # m の推定の場合
    covxx = a*RBF(td,tau)
    # 観測ノイズ込みの推定の場合
```

```
    covxx = a*RBF(td,tau) + sigma**2*np.eye(N)
    IKw=cov
    A=scipy.linalg.solve(IKw,d,assume_a="pos")
    IKw = scipy.linalg.inv(IKw)
    return covx@A, covxx - covx@IKw@covx.T #平均、共分散行列
```

これで同様に HPDI を求めたものが図8.5である。濃い色が $\boldsymbol{m}^*$ の、薄い色が $\boldsymbol{d}^*$ の credible interval である。

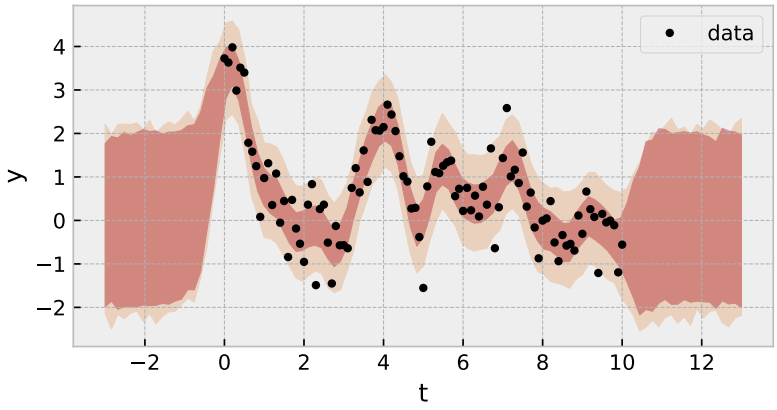


図8.5

8.2　ガウス過程ノイズを含んだモデルフィット

さて、ここまではモデルとしては平均値がゼロのガウス過程により生成されるモデル

$$\boldsymbol{m} \sim \mathcal{N}(\boldsymbol{0}, \Sigma(\boldsymbol{t})) \tag{8.43}$$

を考えていた。ここでは一般に平均値が t の関数となるガウス過程によりデータが生成される場合を考えよう。すなわち

$$\boldsymbol{m} \sim \mathcal{N}(\boldsymbol{f}(\boldsymbol{t}), \Sigma(\boldsymbol{t})) \tag{8.44}$$

の場合を考えよう。ここに $\boldsymbol{f}(\boldsymbol{t})$ は $f(t)$ に対し、$\boldsymbol{t} = (t_0, t_1, \cdot t_{N-1})$ を要素ごとに適用したベクトル版である。ここでは $f(t)$ として

$$f(t) = ke^{-(t-T_0)^2/2s^2} \sin(2\pi t/P) \tag{8.45}$$

というものを考えてみよう。上では簡単に $f(t)$ と書いたが、パラメタを $\boldsymbol{\theta} = (T_0, k, s, P)$ として $f(t; \boldsymbol{\theta})$ という表記も併用する。

式 (8.44) の共分散行列を RBF カーネルとして選択すると、例えば、図8.6のようなデータが生成される。ここで $\tau = 3$ であり $f(t)$ より比較的ゆったりとしたトレンドが乗っているのが見て取れるだろう。このようなモデルを HMC フィットするというのは、シグナル $f(t)$ と相関のあるノイズ＋観測ノイズをモデル化し、$f(t)$ の持つパラメタを推定したいときに対応する。

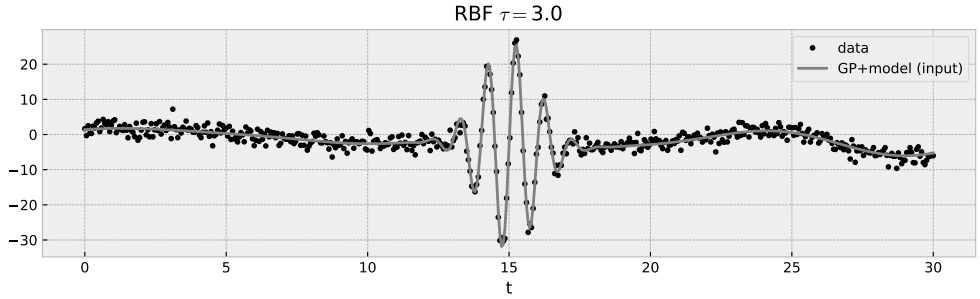


図8.6　実線は σ の観測ノイズなし。

$\boldsymbol{f}(\boldsymbol{t})$ がわかっている場合、観測ノイズを含まない、および、含んだ予測は、それぞれ、

$$p(\boldsymbol{m}^*|\boldsymbol{d}) = \mathcal{N}(\boldsymbol{f}(\boldsymbol{t}^*) + K_\times^\top K_\sigma^{-1}(\boldsymbol{d} - \boldsymbol{f}(\boldsymbol{t})), K_* - K_\times^\top K_\sigma^{-1} K_\times) \tag{8.46}$$

$$p(\boldsymbol{d}^*|\boldsymbol{d}) = \mathcal{N}(\boldsymbol{f}(\boldsymbol{t}^*) + K_\times^\top K_\sigma^{-1}(\boldsymbol{d} - \boldsymbol{f}(\boldsymbol{t})), K_{*,\sigma} - K_\times^\top K_\sigma^{-1} K_\times) \tag{8.47}$$

となる。$\boldsymbol{f}(t)$ の持つパラメタ $\boldsymbol{\theta} = (T_0, k, s, P)$ は HMC でサンプリングされるから、例えば後者なら、サンプリングされた各 $\boldsymbol{\theta}_k^\dagger$ をもちいて

$$\boldsymbol{d}_k^* \sim \mathcal{N}(\boldsymbol{f}(\boldsymbol{t}^*; \boldsymbol{\theta}_k^\dagger) + K_\times^\top K_\sigma^{-1}(\boldsymbol{d} - \boldsymbol{f}(\boldsymbol{t}; \boldsymbol{\theta}_k^\dagger)), K_{*,\sigma} - K_\times^\top K_\sigma^{-1} K_\times) \tag{8.48}$$

をサンプリングすれば、予測のサンプリングができる。というわけで、以下のようにモデル $f(t)$ 込みの GP のフィットを行うことができる。

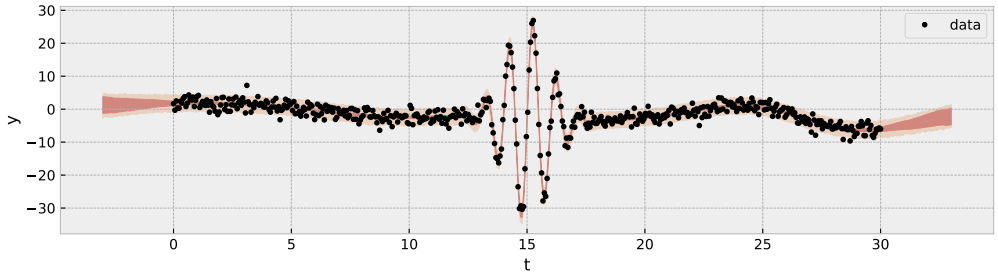


図8.7

第 9 章

NumPyro

NumPyro は PPL なので様々な機能がある。この章では、NumPyro を使っていろいろ遊んでみよう。

9.1 scipy/JAX/NumPyro でサンプリング

scipy/JAX/NumPyro により確率分布からサンプリングをしてみよう。例えば、多変数正規分布の場合

```
from scipy.stats import multivariate_normal as smn
from jax.random import multivariate_normal as jmn

key = random.PRNGKey(4)
N = 1200
t = np.linspace(-5,5,N)
ave = np.zeros(N)
tau = 1.5

tip=0.1
cov = RBF(t,tau)+tip*np.eye(N)

#scipy
ds = smn(mean=ave ,cov=cov , allow_singular =True).rvs(1).T
#jax
dj=jmn(key,ave,cov)
#numpyro
mn=dist.MultivariateNormal(loc=ave, covariance_matrix=cov)
dn = numpyro.sample('a',mn,rng_key=random.PRNGKey(20))
```

こんな感じで、サンプリングできる。

ここで JAX と NumPyro は default では FP32 を用いていることに注意が必要。これは例えば tip=0.1 を小さくしていくと tip=1.e-6 くらいで JAX と NumPyro では値が nan となってしまう。

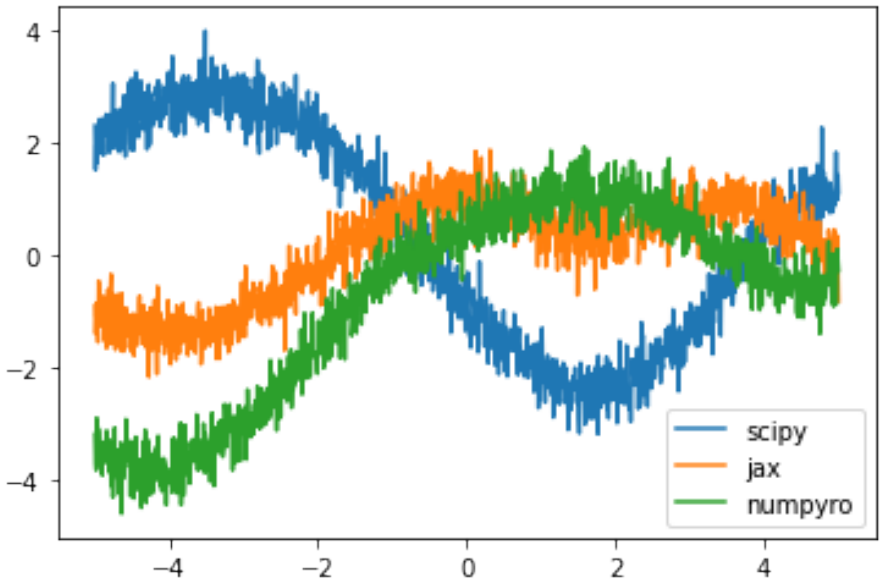


図9.1

```
from jax.config import config
config.update('jax_enable_x64', True)
```

として FP64 にすれば、JAX/NumPyro でも nan とならない。

key の更新

乱数の種を更新していくには jax.random.split を用いる。

```
okey=random.PRNGKey(20) #original key
for i in range(0,10):
    okey,key=random.split(okey)
    dn=numpyro.sample('a',mn,rng_key=key)
```

9.1.1 確率分布の変換

transform

第 10 章

orbax で pytree を保存する

10.1 save/restore

10.1.1 StandardCheckpointer

flax.nnx の state を orbax で保存 (checkpoint) して、restore してみよう。

save は以下のようにする

```
import orbax.checkpoint as ocp
checkpointer = ocp.StandardCheckpointer()
checkpointer.save(ckpt_dir / "state", state)
checkpointer.wait_until_finished()
```

最後の行がないと、バッチ実行の場合、保存中に python が終了してコケるので注意が必要である。これは非同期の save を行っているためである。

restore は以下のようにする

```
from pathlib import Path
ckpt_dir = Path("/home/kawahara/checkpoints")
checkpointer = ocp.StandardCheckpointer()
restored = checkpointer.restore(ckpt_dir / "state")
```

元の object に restore して再利用したい場合、事前に空箱 (abstract_state) を作成する必要がある。

```
from pathlib import Path
ckpt_dir = Path("/home/kawahara/checkpoints")
checkpointer = ocp.StandardCheckpointer()
abstract_model = nnx.eval_shape(
    lambda: EmuMlpDecoder(rngs=nnx.Rngs(0), grid_length=20000)
)
graphdef, abstract_state = nnx.split(abstract_model)
checkpointer = ocp.StandardCheckpointer()
```

```
    state_restored = checkpointer.restore(ckpt_dir / "state", abstract_state)
```

10.1.2 Checkpointer

例えば metadata などを加えて複数のデータを保存する場合、Checkpointer を用いて、save 時に args 引数に複数ファイルの保存の仕方をまとめて (Composite) 与える。ここでは state は pytree なので StandardSave を、metadata は JSON で保存したいので JsonSave を使用している。Checkpointer は非同期でないので wait_until_finished() がない。

```
metadata = {"version": 1}
checkpointer = ocp.Checkpointer(ocp.CompositeCheckpointHandler("state", "metadata"))
checkpointer.save(
        ckpt_dir / "state",
        args=ocp.args.Composite(
            state=ocp.args.StandardSave(state), metadata=ocp.args.JsonSave(metadata)
        ),
)
```

restore 時は以下のようにする

```
checkpointer = ocp.Checkpointer(ocp.CompositeCheckpointHandler("state", "metadata"))
restored = checkpointer.restore(
    ckpt_dir / "state",
    args=ocp.args.Composite(
        state=ocp.args.StandardRestore(abstract_state),
        metadata=ocp.args.JsonRestore(),
    ),
)
state_restored = restored.state
metadata = restored.metadata
```

もちろん metadata だけ最初に読むことも可能である。metadata に abstract_state を定義するために必要な情報を格納しておくことで、 1 ファイルで restore も可能。

```
restored = checkpointer.restore(
    ckpt_dir / "state",
    args=ocp.args.Composite(
        metadata=ocp.args.JsonRestore(),
    ),
)
```

```
print(restored.metadata)
```

また jax.Array や numpy.ndarray を保存するときは ArraySave を用いれば良い

```
array = jnp.ones(100)
checkpointer = ocp.Checkpointer(ocp.CompositeCheckpointHandler("state", "array"))
checkpointer.save(
        ckpt_dir / "state",
        args=ocp.args.Composite(
            state=ocp.args.StandardSave(state), metadata=ocp.args.ArraySave(array)
        ),
)
```

restore も同様。

```
checkpointer = ocp.Checkpointer(ocp.CompositeCheckpointHandler("state", "array"))
restored = checkpointer.restore(
    ckpt_dir / "state",
    args=ocp.args.Composite(
        state=ocp.args.StandardRestore(abstract_state),
        array=ocp.args.ArrayRestore(),
    ),
)
state_restored = restored.state
array_restored = restored.array
```

10.1.3 AsyncCheckpointer

AsyncCheckpointer は非同期でなので、save 前に終了する条件では wait_until_finished() を用いる。

付録 A

Trouble shooting

A.1　JAX 編

JAX の精度が足りてないかもしれない

そんなときは 64bit でチェックだ。

```
from jax.config import config
config.update("jax_enable_x64", True)
```

微分が nan になってる？

そんなときは以下を書いて、どこで nan が発生しているか調べられる。

```
from jax.config import config
config.update("jax_debug_nans", True)
```

付録 B

TIPS

B.1 JAX

B.1.1 jax.value_and_grad の has_aux オプション

jax.value_and_grad に has_aux オプションを True とすると、入力関数の帰り値に定義された補助情報を取得できる。

```python
from jax import value_and_grad
import jax.numpy as jnp

def f(x):
    value = jnp.sum(x**2)
    aux = jnp.sum(jnp.abs(x))
    return value, aux

x = jnp.array([1.0, 2.0, 3.0])
(value, aux), grad = value_and_grad(f, has_aux=True)(x)

print("value:", value)
print("aux:", aux)
print("grad:", grad)
```

出力：

```
value: 14.0
aux: 6.0
grad: [2. 4. 6.]
```

補助情報として accuracy や出力などを出力させると便利である。この事情は nnx.value_and_grad でも同じである。

B.2 Arviz

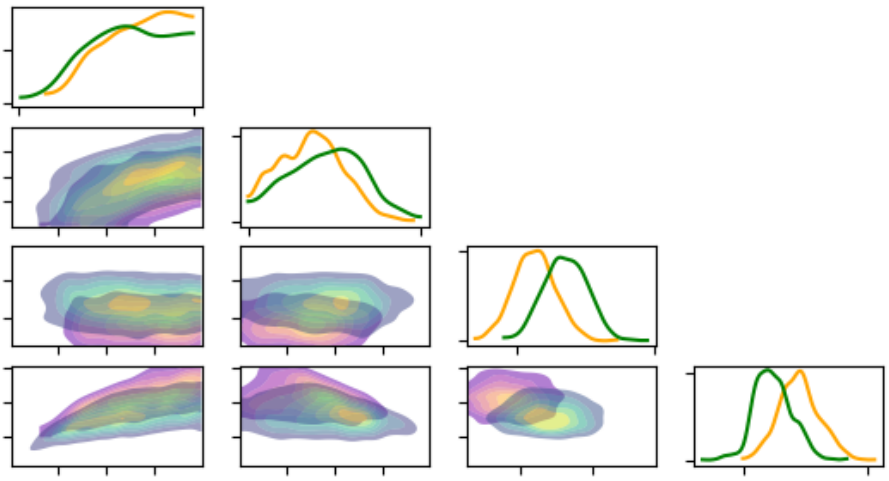
B.2.1 ２つのコーナープロットを重ねて描画する

axes を引き継げばよい。

```
axes=arviz.plot_pair(p,divergences=False,marginals=True,show=False)
axes2=arviz.plot_pair(p2,divergences=False,marginals=True,show=False,ax=axes)
plt.show()
```

色等を区別できるようにするためには、各種 kwarg を使い分ければ良い。例えば図B.1の設定は以下のとおりである。

```
axes=arviz.plot_pair(p,kind='kde',divergences=False,marginals=True,show=False,\
                     kde_kwargs={"contourf_kwargs":{"alpha":0.5,"cmap":"plasma"},\
                     "contour_kwargs":{"alpha":0}},\
                     marginal_kwargs={"color":"orange"})
axes2=arviz.plot_pair(p2,kind='kde',divergences=False,marginals=True,show=False,\
                      ax=axes,\
                      kde_kwargs={"contourf_kwargs":{"alpha":0.5,"cmap":"viridis"},\
                      "contour_kwargs":{"alpha":0}},\
                      marginal_kwargs={"color":"green"})
```



図B.1

付録 C

Appendix

C.1　A. ガウス分布

参考文献

[1] Mathieu Blondel and Vincent Roulet. The Elements of Differentiable Programming. *arXiv e-prints*, page arXiv:2403.14606, March 2024.

[2] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[3] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *arXiv e-prints*, page arXiv:1502.05767, February 2015.

[4] Hajime Kawahara and Kento Masuda. Bayesian Dynamic Mapping of an Exo-Earth from Photometric Variability. *ApJ*, 900(1):48, September 2020.

[5] Radford M. Neal. MCMC using Hamiltonian dynamics. *arXiv e-prints*, page arXiv:1206.1901, June 2012.

[6] Du Phan, Neeraj Pradhan, and Martin Jankowiak. Composable Effects for Flexible and Accelerated Probabilistic Programming in NumPyro. *arXiv e-prints*, page arXiv:1912.11554, December 2019.

[7] Robert Edwin Wengert. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.