ECON622: Problem Set 2

Jesse Perla

Packages

Add whatever packages you wish here

```
import numpy as np
import matplotlib.pyplot as plt
import scipy
import pandas as pd
import jax
import jax.numpy as jnp
from jax import grad, hessian
from jax import random
import optax
import optimistix
import time
```

Question 1

The trace of the Hessian matrix is useful in a variety of applications in statistics, econometrics, and stochastic processes. It can also be used to regularize a loss function.

For a function $f : \mathbb{R}^N \to \mathbb{R}$, denote the Hessian as $\nabla^2 f(x) \in \mathbb{R}^{N \times N}$.

It can be shown that for some mean zero, unit variance random vectors $v \in \mathbb{R}^N$ with $\mathbb{E}(v) = 0$ and $\mathbb{E}(vv^\top) = I$ the trace of the Hessian fulfills

$$\text{Tr}(\nabla^2 f(x)) = \mathbb{E}\left[v^\top \nabla^2 f(x)\, v\right]$$

Which leads to a random algorithm by sampling M vectors $v_1, \ldots, v_M$ and using the Monte Carlo approximation of the expectation, called the Hutchinson Trace Estimator

$$\mathrm{Tr}(\nabla^2 f(x)) \approx \frac{1}{M}\sum_{m=1}^{M} v_m^\top \nabla^2 f(x)\, v_m$$

Question 1.1

Now, let's take the function $f(x) = \frac{1}{2}x^\top P x$, which is a quadratic form and where we know that $\nabla^2 f(x) = P$.

The following code finds the trace of the Hessian, which is equivalently just the sum of the diagonal of P in this simple function.

```
key = jax.random.key(0)

N = 100  # Dimension of the matrix
A = jax.random.normal(key, (N, N))
# Create a positive-definite matrix P by forming A^T * A
P = jnp.dot(A.T, A)
def f(x):
    return 0.5 * jnp.dot(x.T, jnp.dot(P, x))
x = jax.random.normal(key, (N,))
print(jnp.trace(jax.hessian(f)(x)))
print(jnp.diag(P).sum())
```

```
10240.816
10240.817
```

Now, instead of calculating the whole Hessian, use a Hessian-vector product in JAX and the approximation above with M draws of random vectors to calculate an approximation of the trace of the Hessian. Increase the numbers of M to see what the variance of the estimator is, comparing to the above closed-form solution for this quadratic.

Hint: you will want to do Forward-over-Reverse mode differentiation for this (i.e. the `vjp` gives a pullback function for first derivative, then differentiate that new function. Given that it would then be $\mathbb{R}^N \to \mathbb{R}^N$, it makes sense to use forward mode with a `jvp`)

```
# ADD CODE HERE
```

```
from functools import partial
from jax import jvp, vmap

# Hessian-vector product using forward-over-reverse mode
```

```
def hvp(f, x, v):
    return jvp(lambda s: grad(f)(s), (x,), (v,))[1]

# static_argnums=(0, 3) tells JAX to re-compile
# only if 'f' or 'M' change.
# 'x' and 'key' remain dynamic.
@partial(jax.jit, static_argnums=(0, 3))
def hutchinson_trace_estimator(f, x, key, M=100):
    D = x.shape[0]

    # Rademacher distribution (-1 or +1) is standard for Hutchinson
    v_batch = jax.random.rademacher(key, (M, D)).astype(jnp.float32)

    # Single-sample estimator: v^T H v
    def estimate_one(v):
        hv = hvp(f, x, v)
        return jnp.dot(v, hv)

    # vmap pushes the loop to the compiler
    estimates = vmap(estimate_one)(v_batch)
    return jnp.mean(estimates)

# Test with different values of M
true_trace = jnp.trace(jax.hessian(f)(x))
print(f"True trace: {true_trace:.4f}\n")

key = jax.random.key(42)
for M in [1, 10, 100, 1000]:
    key, subkey = jax.random.split(key)

    # Warm-up call (triggers compilation for this M)
    _ = hutchinson_trace_estimator(
        f, x, subkey, M
    ).block_until_ready()

    # Timed call
    key, subkey = jax.random.split(key)
    start = time.perf_counter()
    est_trace = hutchinson_trace_estimator(
        f, x, subkey, M
    ).block_until_ready()
    elapsed = time.perf_counter() - start
```

```
    error = jnp.abs(est_trace - true_trace)
    print(
        f"M={M:<4}: estimate={est_trace:.4f},"
        f" error={error:.4f},"
        f" time={elapsed*1000:.3f}ms"
    )
```

```
True trace: 10240.8164

M=1   : estimate=10229.5576, error=11.2588, time=0.086ms
M=10  : estimate=10182.5596, error=58.2568, time=0.122ms
M=100 : estimate=10503.5107, error=262.6943, time=0.454ms
M=1000: estimate=10209.1455, error=31.6709, time=1.550ms
```

Key improvements over naive implementation:

- **static_argnums=(0, 3)** makes `f` and `M` static (recompiles only when these change), while `x` and `key` stay dynamic
- **vmap** instead of Python loop (much faster, properly vectorized)
- **Rademacher distribution** (± 1) is the standard choice for Hutchinson (lower variance than Gaussian)
- Variance decreases as $O(1/M)$ (standard error as $O(1/\sqrt{M})$)

Question 1.2 (Bonus)

If you wish, you can play around with radically increase the size of the `N` and change the function itself. One suggestion is to move towards a sparse or even matrix-free $f(x)$ calculation so that the P doesn't itself need to materialize.

```
# ADD CODE HERE
```

```
# For sparse or matrix-free computation,
# we can define f without materializing P

def matrix_free_quadratic(x, A):
    """Compute f(x) = 0.5 * x^T (A^T A) x without forming P = A^T A"""
    Ax = jnp.dot(A, x)
    return 0.5 * jnp.dot(Ax, Ax)

# Test with larger dimension
```

```
N_large = 1000
key = jax.random.key(0)
A_large = jax.random.normal(key, (N_large, N_large))
x_large = jax.random.normal(key, (N_large,))

# Use the same hutchinson_trace_estimator from Q1.1
# Since x is dynamic, we can use different
# x values without recompiling
f_large = lambda x: matrix_free_quadratic(x, A_large)

key = jax.random.key(42)
est_trace = hutchinson_trace_estimator(f_large, x_large, key, M=20)
print(f"Estimated trace for N={N_large}: {est_trace:.2f}")
```

```
Estimated trace for N=1000: 1002486.69
```

This approach avoids materializing the $N \times N$ matrix P, making it feasible for much larger problems.

Question 2

This section gives some hints on how to setup a differentiable likelihood function with implicit functions

Question 2.1

The following code uses scipy to find the equilibrium price and demand for some simple supply and demand functions with embedded parameters

```
from scipy.optimize import root_scalar

# Define the demand function with power c
def demand(P, c_d):
    return 100 - 2 * P**c_d

# Define the supply function with power f
def supply(P, c_s):
    return 5 * 3**(c_s * P)

# Define the function to find the root of, including c and f
```

```python
def equilibrium(P, c_d, c_s):
    return demand(P, c_d) - supply(P, c_s)

# Use root_scalar to find the equilibrium price
def find_equilibrium(c_d, c_s):
    result = root_scalar(
        equilibrium,
        args=(c_d, c_s),
        bracket=[0, 100],
        method='brentq',
    )
    return result.root, demand(result.root, c_d)

# Example usage
c_d = 0.5
c_s = 0.15
equilibrium_price, equilibrium_quantity = find_equilibrium(c_d, c_s)
print(f"Equilibrium Price: {equilibrium_price:.2f}")
print(f"Equilibrium Quantity: {equilibrium_quantity:.2f}")
```

```
Equilibrium Price: 17.65
Equilibrium Quantity: 91.60
```

First, convert this to use JAX and Optimistix for finding the root using `optimistix.root_find()`. Make sure you can jit the whole `find_equilibrium` function

```python
# ADD CODE HERE
```

```python
import optimistix as optx

# Convert functions to JAX - params = [c_d, c_s] as array
def demand_jax(P, params):
    c_d, _ = params
    return 100 - 2 * P**c_d

def supply_jax(P, params):
    _, c_s = params
    # Note: if 3^(c_s*P) overflows, use:
    # 5 * jnp.exp(c_s * P * jnp.log(3.0))
    return 5 * 3.0**(c_s * P)
```

```python
def equilibrium_jax(P, params):
    return demand_jax(P, params) - supply_jax(P, params)

# Use optimistix for root finding
# Takes params as array for easy differentiation later
@jax.jit
def find_equilibrium_jax(params):
    solver = optx.Newton(rtol=1e-6, atol=1e-6)
    solution = optx.root_find(
        equilibrium_jax,
        solver,
        y0=jnp.array(5.0),
        args=params,
        max_steps=50,
        throw=False
    )
    P_eq = solution.value
    Q_eq = demand_jax(P_eq, params)
    return P_eq, Q_eq

# Test
params = jnp.array([0.5, 0.15])  # [c_d, c_s]

# Warm-up (compilation)
_ = find_equilibrium_jax(params)[0].block_until_ready()

# Timed call
start = time.perf_counter()
equilibrium_price, equilibrium_quantity = find_equilibrium_jax(params)
equilibrium_price.block_until_ready()
elapsed = time.perf_counter() - start

print(f"Equilibrium Price: {equilibrium_price:.2f}")
print(f"Equilibrium Quantity: {equilibrium_quantity:.2f}")
print(f"Time (after compilation): {elapsed*1000:.3f}ms")
```

```
Equilibrium Price: 17.65
Equilibrium Quantity: 91.60
Time (after compilation): 0.164ms
```

The key differences from scipy:

- Use `optimistix.root_find()` with `Newton` solver (fast, uses automatic differentiation)
- Bundle parameters as `params = [c_d, c_s]` array for easy differentiation later
- The entire function is jitted for performance

Question 2.2

Now, assume that you get a noisy signal on the price that fulfills that demand system.

$$\hat{p} \sim \mathcal{N}(p, \sigma^2)$$

In that case, the log likelihood for the Gaussian is

$$\log \mathcal{L}(\hat{p} \,|\, c_d, c_s, p) = -\frac{1}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}(\hat{p} - p)^2$$

Or, if p was implicitly defined by the equilibrium conditions as some $p(c_d, c_s)$ from above,

$$\log \mathcal{L}(\hat{p} \,|\, c_d, c_s) = -\frac{1}{2}\log(2\pi\sigma^2) - \frac{1}{2\sigma^2}(\hat{p} - p(c_d, c_s))^2$$

Then for some $\sigma = 0.01$ we can calculate this log likelihood the above as

```
def log_likelihood(p_hat, c_d, c_s, sigma):
    p, x = find_equilibrium(c_d, c_s)
    return (
        -0.5 * np.log(2 * np.pi * sigma**2)
        - 0.5 * (p_hat - p) ** 2 / sigma**2
    )

c_d = 0.5
c_s = 0.15
sigma = 0.01
# get the true value for simulation
p, x = find_equilibrium(c_d, c_s)
# simulate a noisy signal
p_hat = p + np.random.normal(0, sigma)
log_likelihood(p_hat, c_d, c_s, sigma)
```

```
np.float64(3.6639725087467805)
```

Now, take this code for the likelihood and convert it to JAX and jit. Use your function from Question 2.1

```
# ADD CODE HERE
```

```
@jax.jit
def log_likelihood_jax(p_hat, params, sigma):
    p, _ = find_equilibrium_jax(params)
    return (
        -0.5 * jnp.log(2 * jnp.pi * sigma**2)
        - 0.5 * (p_hat - p) ** 2 / sigma**2
    )

# Test
params = jnp.array([0.5, 0.15])  # [c_d, c_s]
sigma = 0.01
key = jax.random.key(0)

# get the true value for simulation
p, _ = find_equilibrium_jax(params)
# simulate a noisy signal
p_hat = p + jax.random.normal(key, ()) * sigma
ll = log_likelihood_jax(p_hat, params, sigma)
print(f"Log likelihood: {ll:.4f}")
```

```
Log likelihood: 2.3698
```

Key changes:

- Replace `np` with `jnp` for all numpy operations
- Use `jax.random.normal()` instead of `np.random.normal()`
- The function is fully jittable including the implicit function solve

Question 2.3

Use the function from the previous part and calculate the gradient with respect to `params` (i.e., `c_d` and `c_s`) using `grad` and JAX.

```
# ADD CODE HERE
```

```python
# Differentiate w.r.t. params (argument 1) using argnums
grad_fn = jax.jit(jax.grad(log_likelihood_jax, argnums=1))

# Test
params = jnp.array([0.5, 0.15])  # [c_d, c_s]
sigma = 0.01
key = jax.random.key(0)

p, _ = find_equilibrium_jax(params)
p_hat = p + jax.random.normal(key, ()) * sigma

# Warm-up (compilation)
_ = grad_fn(p_hat, params, sigma).block_until_ready()

# Timed call
start = time.perf_counter()
grads = grad_fn(p_hat, params, sigma).block_until_ready()
elapsed = time.perf_counter() - start

print(f"Gradient w.r.t. c_d: {grads[0]:.6f}")
print(f"Gradient w.r.t. c_s: {grads[1]:.6f}")
print(f"Time (after compilation): {elapsed*1000:.3f}ms")
```

```
Gradient w.r.t. c_d: -255.215134
Gradient w.r.t. c_s: -18792.078125
Time (after compilation): 0.123ms
```

Since `log_likelihood_jax` already takes `params` as an array, we just use `argnums=1` to differentiate with respect to the second argument. No wrapper functions needed!

The key insight: Optimistix implements custom VJP rules based on the **implicit function theorem**, so `jax.grad` can differentiate *through* the equilibrium root-finding step without unrolling the Newton iterations. This is why we can take gradients of a likelihood that internally solves a nonlinear equation.

Question 2.4 (Bonus)

You could try to run maximum likelihood estimation by using a gradient-based optimizer. You can use either Optax (standard for ML optimization) or Optimistix with `optimistix.minimise()`.

If you attempt this:

- Consider starting your optimization at the "pseudo-true" values with the `c_s, c_d, sigma` you used to simulate the data and even start with `p_hat = p`.
- You may find that it is a little too noisy with only the one observation. If so, you could adapt your likelihood to take a vector of $\hat{p}$ instead. The likelihood of IID gaussians is a simple variation on the above.

```
# ADD CODE HERE
```

```
# Generate synthetic data
params_true = jnp.array([0.5, 0.15])  # [c_d, c_s]
sigma = 0.01
n_obs = 50  # Use multiple observations to reduce noise

key = jax.random.key(42)
p_true, _ = find_equilibrium_jax(params_true)
key, subkey = jax.random.split(key)
p_hat_data = p_true + jax.random.normal(subkey, (n_obs,)) * sigma

# Negative log likelihood for multiple observations
# (we minimize this)
@jax.jit
def neg_log_likelihood_multi(params, p_hat_data, sigma):
    p, _ = find_equilibrium_jax(params)
    n = p_hat_data.shape[0]
    ll = (
        -0.5 * n * jnp.log(2 * jnp.pi * sigma**2)
        - 0.5 * jnp.sum((p_hat_data - p) ** 2)
        / sigma**2
    )
    return -ll  # Return negative for minimization

# Option 1: Using Optax (standard optimizer)
print("=== Optax (Adam) ===")
params_init = jnp.array([0.6, 0.18])

optimizer = optax.adam(learning_rate=0.01)
opt_state = optimizer.init(params_init)

# Warm-up (compilation)
nll_grad = jax.value_and_grad(neg_log_likelihood_multi)
_ = nll_grad(
    params_init, p_hat_data, sigma
```

```
)[0].block_until_ready()

# Timed optimization loop
params = params_init
start = time.perf_counter()
for step in range(100):
    loss, grads = nll_grad(
        params, p_hat_data, sigma
    )
    updates, opt_state = optimizer.update(
        grads, opt_state
    )
    params = optax.apply_updates(params, updates)
params[0].block_until_ready()
optax_time = time.perf_counter() - start

print(
    f"True values: c_d={params_true[0]},"
    f" c_s={params_true[1]}"
)
print(
    f"Estimated:   c_d={params[0]:.4f},"
    f" c_s={params[1]:.4f}"
)
print(
    f"Time (100 steps, after compilation):"
    f" {optax_time*1000:.1f}ms"
)

# Option 2: Optimistix minimise (alternative)
print("\n=== Optimistix (BFGS) ===")

@jax.jit
def run_bfgs(params_init, p_hat_data, sigma):
    def loss_fn(params, args):
        p_hat_data, sigma = args
        return neg_log_likelihood_multi(
            params, p_hat_data, sigma
        )

    solver = optx.BFGS(rtol=1e-6, atol=1e-6)
    return optx.minimise(
```

```
        loss_fn,
        solver,
        params_init,
        args=(p_hat_data, sigma),
        max_steps=100,
    )

params_init_opt = jnp.array([0.6, 0.18])

# Warm-up (compilation)
_ = run_bfgs(
    params_init_opt, p_hat_data, sigma
).value.block_until_ready()

# Timed call
start = time.perf_counter()
solution = run_bfgs(
    params_init_opt, p_hat_data, sigma
)
solution.value.block_until_ready()
bfgs_time = time.perf_counter() - start

print(
    f"Estimated:   c_d={solution.value[0]:.4f},"
    f" c_s={solution.value[1]:.4f}"
)
print(
    f"Time (after compilation):"
    f" {bfgs_time*1000:.1f}ms"
)
```

```
=== Optax (Adam) ===
True values: c_d=0.5, c_s=0.15000000596046448
Estimated:   c_d=0.5615, c_s=0.1491
Time (100 steps, after compilation): 561.7ms

=== Optimistix (BFGS) ===
Estimated:   c_d=0.5995, c_s=0.1520
Time (after compilation): 0.3ms
```

This demonstrates MLE with implicit functions:

- **Optax approach**: Standard gradient descent with Adam optimizer, gives fine control over training
- **Optimistix approach**: Uses BFGS quasi-Newton method, often faster for smooth problems
- With 50 observations, the noise is reduced enough to recover parameters accurately
- The implicit function (root finding) is differentiated through automatically