**JAX – Benefits**

1. Using Autograd, it can automatically differentiate through loops, ifs, recursion and closures (function + environment).
2. Supports reverse-mode and forward-mode differentiation.
3. Using XLA, it compiles and runs code on accelerators (GPU/TPU).
4. Library calls and custom functions can be just-in-time compiled into XLA-optimised kernels.
5. Three main program transformations:
   1. jit() – speeds up code: just-in-time compilation decorator that optimises sequences of operations together and run at once.
   2. grad() – takes derivatives
   3. vmap() – automatically vectorises/batches

**JAX – Rules and Oddities**

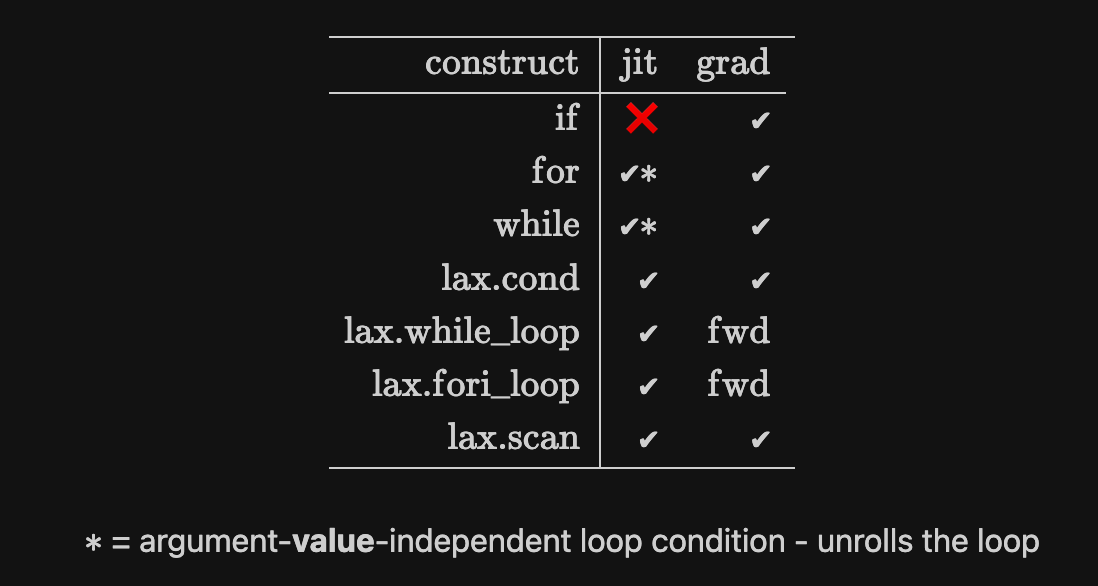
1. **Pure functions only**. No side effects. Same input = same output.
2. Internal stateful objects only.
3. **No in-place modifications** (arrays are immutable).
   1. To update an element, use the indexed update syntax which returns an updated copy: array2 = array1.at[<index>].set(<value>)
   2. The returned array is a copy; the original is unmodified.
   3. Inside jit-compiled code, if the original is not reused, the compiler will optimise the array update to occur in-place.
   4. Can also do such operations as a2 = a1.at[∷2,3:].add(7.)
   5. You can donate an input buffer to hold an output using the donate\_argnums=(1,) jit parameter.
4. **Asynchronous execution**. Use .block\_until\_ready()
5. Jax.numpy is high-level familiar wrapper; jax.lax is lower-level API that is stricter/more powerful.
6. All JAX operations are implemented in terms of operations in XLA.
7. Jax.lax requires same data types for operations (i.e. explicitly promote types).
8. **Jit requires array shapes to be static and known at compile time** (no dynamically sized arrays).
9. Jit (and other transforms) trace a function to determine its effect on inputs of a specific shape/type.
10. Mark as static the variables that can’t/shouldn’t be traced.
    1. From functools import partial; @partial(jit, static\_argnums=(1,))
11. **Statically marked objects are effectively used as a dictionary key** in JIT’s internal compilation cache.
    1. It’s hash equality (obj1==obj2) and object identity (obj1 is obj2) will be assumed to have consistent behaviour.
12. **Control flow statements (if, then) can’t depend on traced values**.
13. **Values and operations can be static or traced**.
14. **Static operations are evaluated at compile-time in Python; traced operations are compiled/evaluated at run-time in XLA.**
    1. Use numpy for static operations; use jax.numpy for traced operations.
15. **Iterators are not recommended in jitted functions** or control-flow primitive.
16. **Jitted numerical functions can only specialise internal array shapes on argument shapes (not on argument values).**
    1. Inside jitted code/lax.while\_loop/lax.fori\_loop, the size of slices can only be functions of argument shapes, not argument values.
    2. Slice start indices do not have this restriction.
17. Grad() can be used without constraint on regular python control-flow constructs, but jit() has constraints.
18. “To get a view of the python code that is valid for many different argument values, JAX traces it on abstract values that represent sets of possible inputs. There are multiple different levels of abstraction, and different transformations use different abstraction levels.”
    1. Default level of abstraction is ShapedArray: each abstract value represents the set of all array values with a fixed shape and dtype.
    2. Jit can trace on more refined abstract values and traceability constraints can be relaxed.
19. Debugging NaNs requires some fiddling and use of %debug mode.
20. Since XLA rearranges/merges operations for optimal efficiency, **outputs of jitted functions can differ to non-jitted functions due to floating point arithmetic.**
21. For a handful of loop iterations, python is okay, but **if many loop iterations are needed, code should be rewritten to make use of JAX’s structured control flow primitives** (such as lax.scan()) or avoid wrapping the loop with jit.
22. If code makes use of loops due to many arrays with different shapes, it is recommended to make use of jax.numpy.where() to perform computations on padded arrays of fixed shape.
23. Using jit with methods is complicated and requires one of three strategies (website).
24. **JAX array placement properties** are the device (GPU/TPU/CPU) where the data resides and whether the data is committed/sticky to the device.
25. **Abstract tracers carry an abstract value**, e.g. ShapedArray with info about the shape and dtype of an array; **concrete tracers carry ConcreteArray abstract values that include the regular array data.**
    1. Tracer values computed from tracer values are usually tracer values.
    2. Tracer values computed from concrete tracers result in concrete tracers.
    3. A concrete value is either a regular value or a concrete tracer.
26. The **transformations introduce abstract/concrete tracers differently**:
    1. Jit: all abstract except those denoted by static\_argnums.
    2. Vmap, make\_jaxpr: all abstract.
    3. Grad: all concrete, unless arguments are abstract, then abstract.
    4. Cond, while\_loop, fori\_loop, scan: abstract.
27. **To print a traced value at runtime** (for debugging), **use jax.debug.print().**
28. To **use non-JAX code within a transformed JAX function**, use **jax.pure\_callback()**

**JAX – Differences from NumPy**

1. By default, JAX only uses 32-bit dtypes.
   1. Using double-precision numbers (float64) requires setting the configuration variable at startup.
2. Out-of-bounds indexing will either be skipped, or return the first/last value in array.
3. Jnp.nanargmin/jnp.nanargmax return -1 for slices consisting of NaNs.
4. JAX operations only accept ndarray/scalar arguments (no lists, tuples).
   1. Pass a tuple/list by first explicitly converting to array (jnp.array(x))
5. JAX uses a modern Threefry counter-based splittable PRNG.
   1. To get a new pseudorandom number, split the PRNG and get a usable subkey.
   2. key = jax.random.PRNGKey(0)

key, subkey = jax.random.split(key)

jax.random.normal(subkey, shape=(1,))



**JAX functions**

Four structured control-flow primitives that avoid re-compilation, are tracible and avoid un-rolling large loops:

1. lax.cond: (condition, true\_function, false\_function, operand)
2. lax.while\_loop: (condition\_function, body\_function, initial\_value)
3. lax.fori\_loop: (start, stop, body\_function, initial\_value)
4. lax.scan: ()

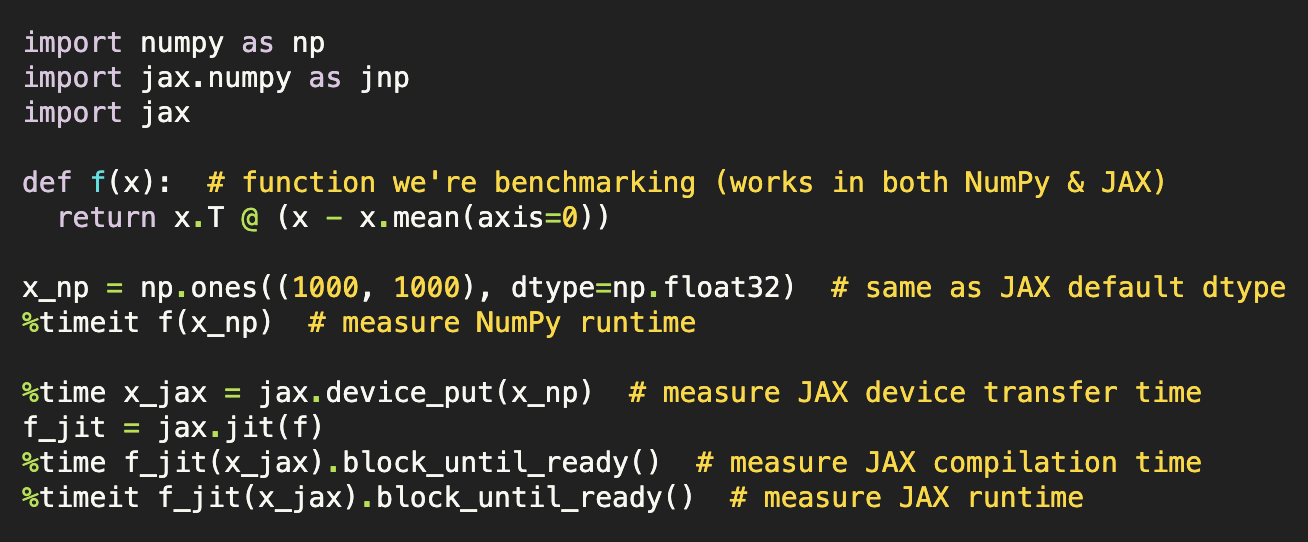
Additional functions:

1. Lax.select(): batched version of cond.
2. Lax.switch(): allows switching between multiple callable choices.
3. Jnp.where(): wrapper of lax.select.
4. Jnp.piecewise(): wrapper of lax.switch (switches on boolean list).
5. Jnp.select(): similar to piecewise, but choices are given as

precomputed arrays rather than functions.

**Useful code examples**

Microbenchmark comparing JAX to NumPy:

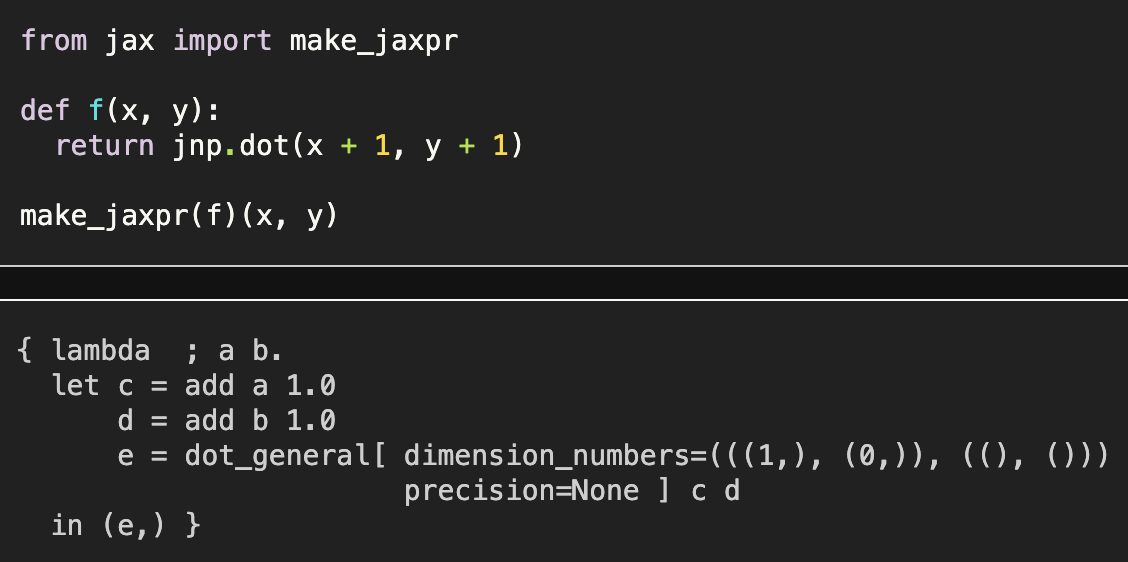


Taking derivatives:

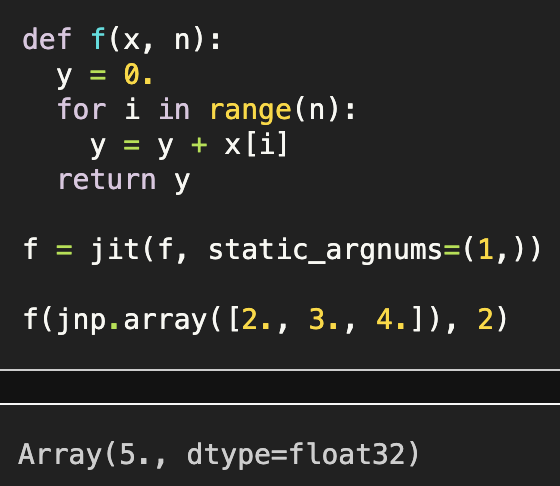
A black screen with white text

Description automatically generated

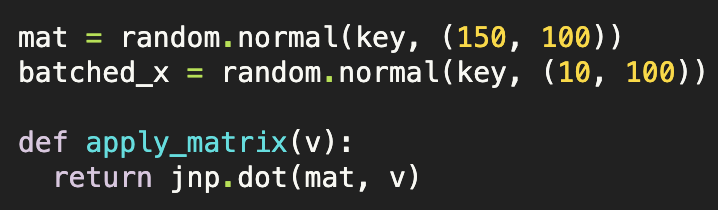
View the extracted sequence of operations (encoded in a JAX expression):

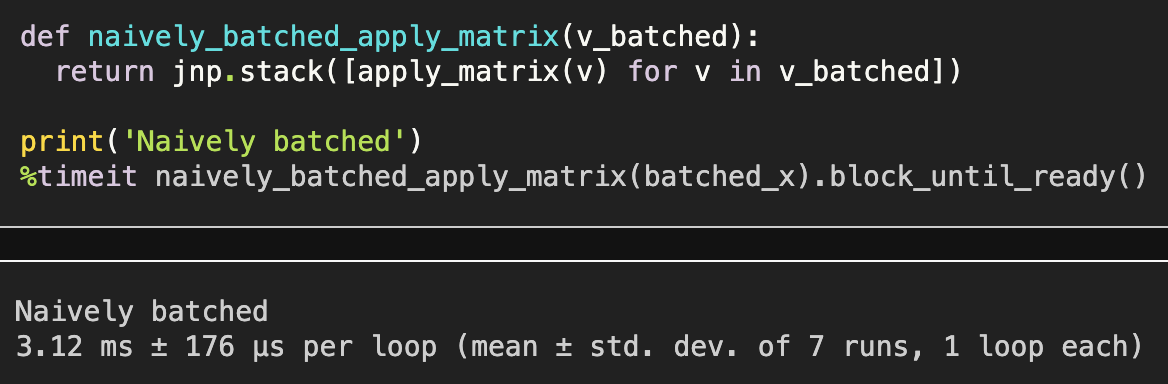


Loop with static argument:



Auto-vectorisation with vmap():







(Using vmap() automatically adds batching support.)

