STAT243 Lecture 5.8 Efficiency

1 Interpreters and Compilation

1.1 Why Are Interpreted Languages Slow?

- **编译语言（如 C、Fortran）**
  - 源代码在执行前被翻译成 **机器码（machine code）**（0 和 1），
    CPU 可直接执行这些指令。
  - 各种类型检查、内存定位等工作在编译阶段完成，因此运行时无需重复。
- **解释语言（如 Python、R）**
  - 在运行时由 **解释器（interpreter）** 逐行读取和执行。
  - 每次执行时都需：
    1. 检查变量是否存在；
    2. 确认类型是否合法；
    3. 查找对应作用域（scope）中的值与方法。
  - 这导致每次函数调用、运算符执行都需要大量“检查与查找”步骤。

示例：

Python

```python
x = 3
abs(x)   # 计算绝对值
x * 7
x = 'hi'
abs(x)   # 错误：类型不匹配
x * 3    # 字符串重复
```

解释器每次都要重新确认 `x` 的类型与操作的适用性。

1.2 Dynamic Typing and Overhead

Logic

Python 是 **动态类型语言**：变量类型可随时改变。
因此解释器必须在运行时不断验证每个操作是否合法。

示例（极端情况）：

Python

```python
x = np.random.normal(size=10)
for i in range(10):
    if np.random.normal() > 0:
        x = 'hi'
    if np.random.normal() > 0.5:
        del x
    x[i] = np.exp(x[i])
```

解释器必须在每次迭代中检查：

- `x` 是否存在；
- `x` 是否为数组；
- `x` 是否含有数值；
- `x[i]` 是否可修改。

这类动态检查使解释语言较慢。
而 **JIT（Just-In-Time）编译** 的思想即尝试缓存这些类型与检查结果，以加速后续执行。

1.3 CPython 的本质

- **CPython** 是 Python 的标准实现，本质上是一个 **C 编译程序**，用于解释 Python 源码。
- 尽管底层是编译好的 C 代码，但每次执行 Python 指令时仍需大量额外操作（解析、检查、绑定）。
- 类比： 直接用相同语言交流（编译代码） vs. 通过翻译转述（解释器）。

1.4 Built-in Safety Checks

- 许多 Python 函数在调用底层 C 库前都会执行大量检查。
  例如：

  Python

  ```python
  from scipy.linalg import solve_triangular
  solve_triangular(A, b, check_finite=False)
  ```

  若将 `check_finite=False`，性能可提高，但若输入中包含 NaN 或 Inf，则结果不可预期。

> ⚠ **Remark**
>
> 高级库函数（如 SciPy、NumPy）虽调用底层 C/Fortran 实现，
> 但仍会在 Python 层面执行输入检查，这部分往往成为性能瓶颈。

1.5 What Executes Quickly in Python

1. **调用底层编译函数的操作**
   - 如 NumPy 的算术与矩阵运算、Pandas 的矢量化操作。
2. **线性代数（BLAS / LAPACK）调用**
   - 实际运算由高效的 Fortran/C 库完成。
3. **矢量化（Vectorization）**
   - 避免 Python 层的循环，将循环下放至 C 层。

   Python

   ```python
   # 慢：Python 逐元素循环
   for i in range(len(x)): x[i] = np.exp(x[i])
   # 快：C 层矢量化
   x = np.exp(x)
   ```

   在矢量化中，解释器不再对每次迭代做类型与存在性检查。

1.6 Compilation

1.6.1 Overview

- **Compilation（编译）** 是将源码（如 C++、Fortran）转换为机器码的过程。
- 编译后的二进制文件（executable）可被 CPU 直接执行。
- 编译语言具备：
  - 运行速度快；
  - 类型静态、可优化；
  - 但开发周期相对较长。

> **Logic**
>
> 编程语言的发展不断提升抽象层次：
> Machine Code → Assembly → C/Fortran → Python/Julia。
> 抽象度越高，开发效率越高，但执行速度通常越慢。

1.6.2 Python Interpreter

- Python 解释器（CPython）本身是一个 **已编译的 C 程序**。
- 它负责解析并执行 `.py` 文件，但不会生成可直接执行的机器码。

1.7 Just-in-Time (JIT) Compilation

- **AOT (Ahead-of-Time)** 编译：
  代码在执行前被完全编译（如 C、C++）。
- **JIT (Just-in-Time)** 编译：
  代码在运行时被动态编译为机器码，并可缓存结果。

特征：

- **类型推断（type inference）**：推测变量类型以减少动态检查；
- **即时优化（runtime optimization）**；
- **缓存（caching）**：再次调用同一函数时无需重新编译。

示例：

- Julia 语言：大量使用 JIT，首轮运行慢，之后极快；
- Python：通过 `numba` 提供 JIT 编译支持。

Python

```python
from numba import njit

@njit
def f(x):
    return np.exp(x)
```

> **Logic**
>
> `numba.njit` 使用 LLVM 库将 Python + NumPy 代码编译为机器码，
> 运行速度可接近 C。

1.8 Byte Compiling (可选)

- **字节码编译（Byte Compilation）**
  将 Python 源码转换为 **字节码（bytecode）**，以便更快执行。
  - 跳过了解析与语法分析阶段。
  - 但字节码仍需由解释器执行，不是机器码。

生成的 `.pyc` 文件即为字节码文件。
示例：

Python

```python
import py_compile
py_compile.compile('vec.py')
# '__pycache__/vec.cpython-312.pyc'
```

1.8.1 Performance Comparison

Python

```python
import time

def f(vals):
    x = np.zeros(len(vals))
    for i in range(len(vals)):
        x[i] = np.exp(vals[i])
    return x

x = np.random.normal(size=10**6)
# 普通函数
t0 = time.time(); f(x); print(time.time()-t0)  # ~0.75 s
# 矢量化操作
t0 = time.time(); np.exp(x); print(time.time()-t0)  # ~0.013 s
```

字节码化后：

Python

```python
import vec
vec.f(x)  # ~0.73 s
```

> ⚠ **Remark**
>
> `.pyc` 文件仅略微减少解释开销，对 CPU 密集型任务帮助有限。
> 真正显著的性能提升通常来自：
>
> - 使用 **NumPy/Cython/Numba** 的编译扩展；
> - 或直接在 **C/Fortran** 中实现关键计算逻辑。