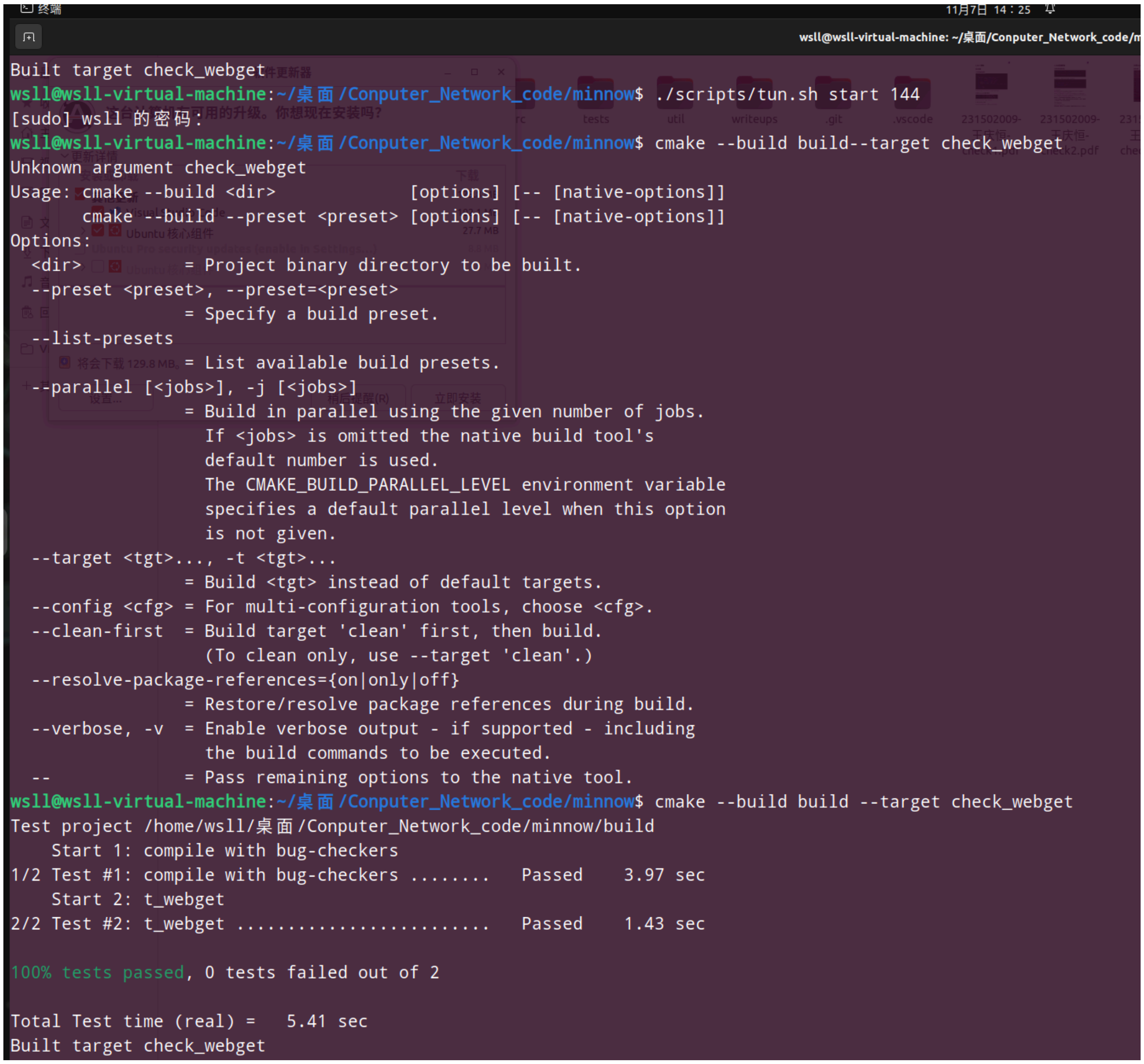
计网lab4

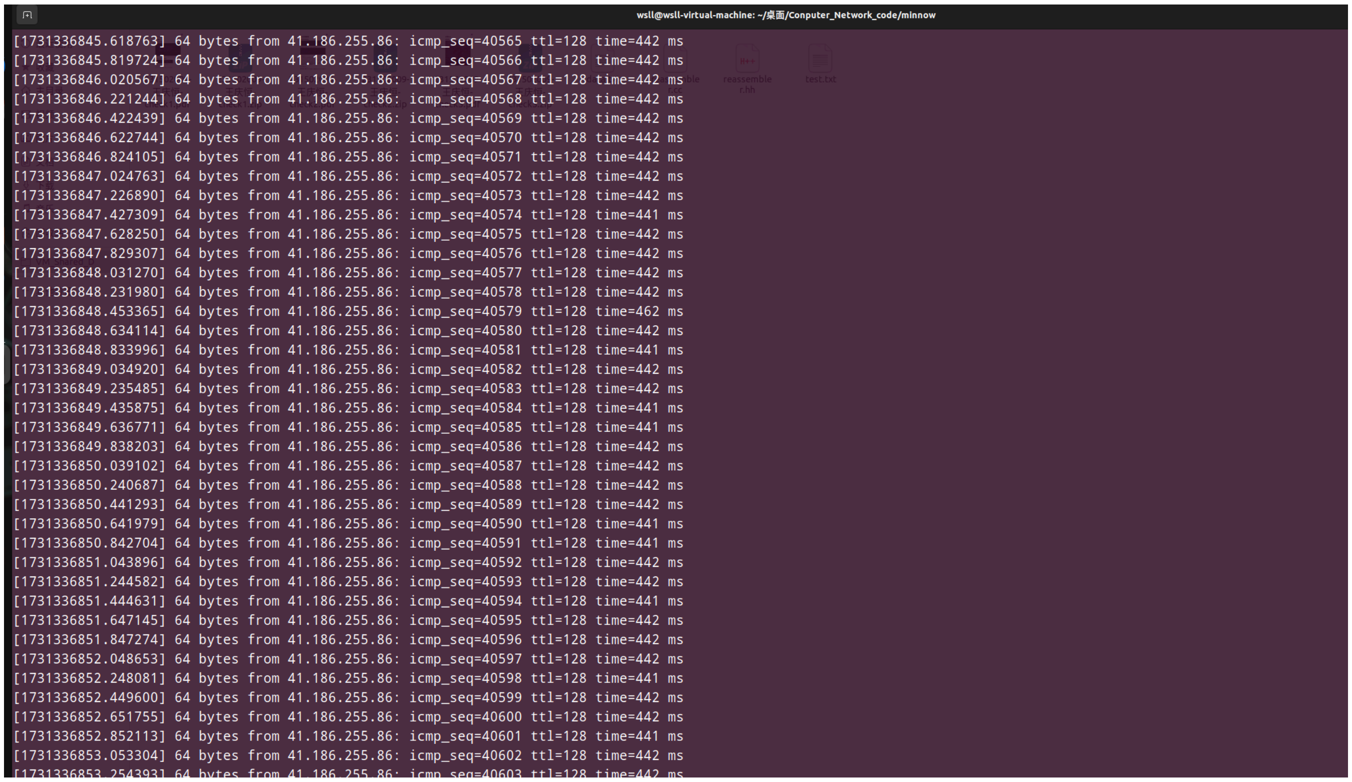
运行webget



```
Built target check_webget
wsll@wsll-virtual-machine:~/桌面/Conputer_Network_code/minnow$ ./scripts/tun.sh start 144
[sudo] wsll 的密码：
wsll@wsll-virtual-machine:~/桌面/Conputer_Network_code/minnow$ cmake --build build--target check_webget
Unknown argument check_webget
Usage: cmake --build <dir>             [options] [-- [native-options]]
       cmake --build --preset <preset> [options] [-- [native-options]]
Options:
  <dir>          = Project binary directory to be built.
  --preset <preset>, --preset=<preset>
                 = Specify a build preset.
  --list-presets
                 = List available build presets.
  --parallel [<jobs>], -j [<jobs>]
                 = Build in parallel using the given number of jobs.
                   If <jobs> is omitted the native build tool's
                   default number is used.
                   The CMAKE_BUILD_PARALLEL_LEVEL environment variable
                   specifies a default parallel level when this option
                   is not given.
  --target <tgt>..., -t <tgt>...
                 = Build <tgt> instead of default targets.
  --config <cfg> = For multi-configuration tools, choose <cfg>.
  --clean-first  = Build target 'clean' first, then build.
                   (To clean only, use --target 'clean'.)
  --resolve-package-references={on|only|off}
                 = Restore/resolve package references during build.
  --verbose, -v  = Enable verbose output - if supported - including
                   the build commands to be executed.
  --             = Pass remaining options to the native tool.
wsll@wsll-virtual-machine:~/桌面/Conputer_Network_code/minnow$ cmake --build build --target check_webget
Test project /home/wsll/桌面/Conputer_Network_code/minnow/build
    Start 1: compile with bug-checkers
1/2 Test #1: compile with bug-checkers ........   Passed    3.97 sec
    Start 2: t_webget
2/2 Test #2: t_webget .........................   Passed    1.43 sec

100% tests passed, 0 tests failed out of 2

Total Test time (real) =   5.41 sec
Built target check_webget
```

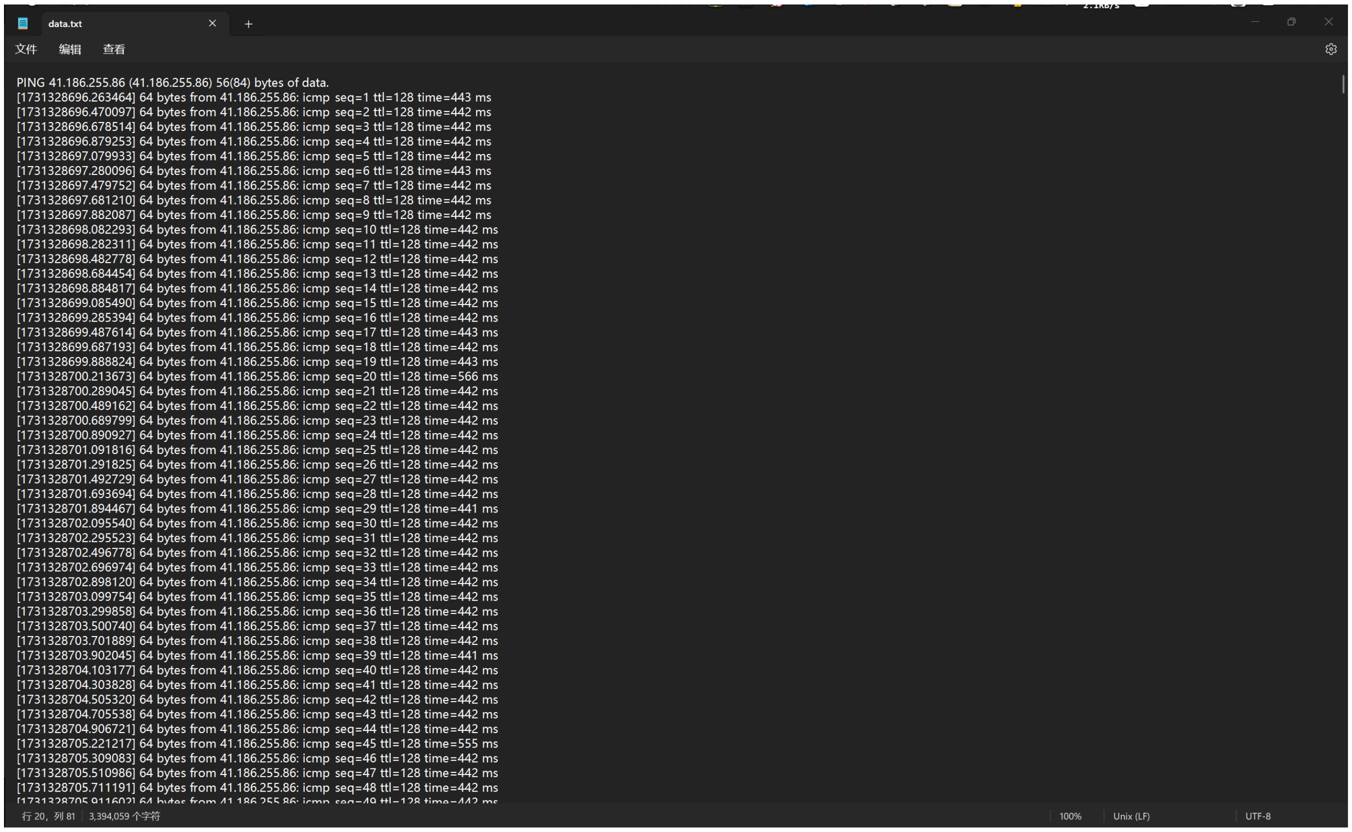
非常正常。

进行测试


wsll@wsll-virtual-machine: ~/桌面/Conputer_Network_code/minnow

```
[1731336845.618763] 64 bytes from 41.186.255.86: icmp_seq=40565 ttl=128 time=442 ms
[1731336845.819724] 64 bytes from 41.186.255.86: icmp_seq=40566 ttl=128 time=442 ms
[1731336846.020567] 64 bytes from 41.186.255.86: icmp_seq=40567 ttl=128 time=442 ms
[1731336846.221244] 64 bytes from 41.186.255.86: icmp_seq=40568 ttl=128 time=442 ms
[1731336846.422439] 64 bytes from 41.186.255.86: icmp_seq=40569 ttl=128 time=442 ms
[1731336846.622744] 64 bytes from 41.186.255.86: icmp_seq=40570 ttl=128 time=442 ms
[1731336846.824105] 64 bytes from 41.186.255.86: icmp_seq=40571 ttl=128 time=442 ms
[1731336847.024763] 64 bytes from 41.186.255.86: icmp_seq=40572 ttl=128 time=442 ms
[1731336847.226890] 64 bytes from 41.186.255.86: icmp_seq=40573 ttl=128 time=442 ms
[1731336847.427309] 64 bytes from 41.186.255.86: icmp_seq=40574 ttl=128 time=441 ms
[1731336847.628250] 64 bytes from 41.186.255.86: icmp_seq=40575 ttl=128 time=442 ms
[1731336847.829307] 64 bytes from 41.186.255.86: icmp_seq=40576 ttl=128 time=442 ms
[1731336848.031270] 64 bytes from 41.186.255.86: icmp_seq=40577 ttl=128 time=442 ms
[1731336848.231980] 64 bytes from 41.186.255.86: icmp_seq=40578 ttl=128 time=442 ms
[1731336848.453365] 64 bytes from 41.186.255.86: icmp_seq=40579 ttl=128 time=462 ms
[1731336848.634114] 64 bytes from 41.186.255.86: icmp_seq=40580 ttl=128 time=442 ms
[1731336848.833996] 64 bytes from 41.186.255.86: icmp_seq=40581 ttl=128 time=441 ms
[1731336849.034920] 64 bytes from 41.186.255.86: icmp_seq=40582 ttl=128 time=442 ms
[1731336849.235485] 64 bytes from 41.186.255.86: icmp_seq=40583 ttl=128 time=442 ms
[1731336849.435875] 64 bytes from 41.186.255.86: icmp_seq=40584 ttl=128 time=441 ms
[1731336849.636771] 64 bytes from 41.186.255.86: icmp_seq=40585 ttl=128 time=441 ms
[1731336849.838203] 64 bytes from 41.186.255.86: icmp_seq=40586 ttl=128 time=442 ms
[1731336850.039102] 64 bytes from 41.186.255.86: icmp_seq=40587 ttl=128 time=442 ms
[1731336850.240687] 64 bytes from 41.186.255.86: icmp_seq=40588 ttl=128 time=442 ms
[1731336850.441293] 64 bytes from 41.186.255.86: icmp_seq=40589 ttl=128 time=442 ms
[1731336850.641979] 64 bytes from 41.186.255.86: icmp_seq=40590 ttl=128 time=441 ms
[1731336850.842704] 64 bytes from 41.186.255.86: icmp_seq=40591 ttl=128 time=441 ms
[1731336851.043896] 64 bytes from 41.186.255.86: icmp_seq=40592 ttl=128 time=442 ms
[1731336851.244582] 64 bytes from 41.186.255.86: icmp_seq=40593 ttl=128 time=442 ms
[1731336851.444631] 64 bytes from 41.186.255.86: icmp_seq=40594 ttl=128 time=441 ms
[1731336851.647145] 64 bytes from 41.186.255.86: icmp_seq=40595 ttl=128 time=442 ms
[1731336851.847274] 64 bytes from 41.186.255.86: icmp_seq=40596 ttl=128 time=442 ms
[1731336852.048653] 64 bytes from 41.186.255.86: icmp_seq=40597 ttl=128 time=442 ms
[1731336852.248081] 64 bytes from 41.186.255.86: icmp_seq=40598 ttl=128 time=441 ms
[1731336852.449600] 64 bytes from 41.186.255.86: icmp_seq=40599 ttl=128 time=442 ms
[1731336852.651755] 64 bytes from 41.186.255.86: icmp_seq=40600 ttl=128 time=442 ms
[1731336852.852113] 64 bytes from 41.186.255.86: icmp_seq=40601 ttl=128 time=441 ms
[1731336853.053304] 64 bytes from 41.186.255.86: icmp_seq=40602 ttl=128 time=442 ms
[1731336853.254393] 64 bytes from 41.186.255.86: icmp_seq=40603 ttl=128 time=442 ms
```

我们把data.txt转移到Windows中以便操作：



```
PING 41.186.255.86 (41.186.255.86) 56(84) bytes of data.
[1731328696.263464] 64 bytes from 41.186.255.86: icmp seq=1 ttl=128 time=443 ms
[1731328696.470097] 64 bytes from 41.186.255.86: icmp seq=2 ttl=128 time=442 ms
[1731328696.678514] 64 bytes from 41.186.255.86: icmp seq=3 ttl=128 time=442 ms
[1731328696.879253] 64 bytes from 41.186.255.86: icmp seq=4 ttl=128 time=442 ms
[1731328697.079933] 64 bytes from 41.186.255.86: icmp seq=5 ttl=128 time=442 ms
[1731328697.280096] 64 bytes from 41.186.255.86: icmp seq=6 ttl=128 time=443 ms
[1731328697.479752] 64 bytes from 41.186.255.86: icmp seq=7 ttl=128 time=442 ms
[1731328697.681210] 64 bytes from 41.186.255.86: icmp seq=8 ttl=128 time=442 ms
[1731328697.882087] 64 bytes from 41.186.255.86: icmp seq=9 ttl=128 time=442 ms
[1731328698.082293] 64 bytes from 41.186.255.86: icmp seq=10 ttl=128 time=442 ms
[1731328698.282311] 64 bytes from 41.186.255.86: icmp seq=11 ttl=128 time=442 ms
[1731328698.482778] 64 bytes from 41.186.255.86: icmp seq=12 ttl=128 time=442 ms
[1731328698.684454] 64 bytes from 41.186.255.86: icmp seq=13 ttl=128 time=442 ms
[1731328698.884817] 64 bytes from 41.186.255.86: icmp seq=14 ttl=128 time=442 ms
[1731328699.085490] 64 bytes from 41.186.255.86: icmp seq=15 ttl=128 time=442 ms
[1731328699.285394] 64 bytes from 41.186.255.86: icmp seq=16 ttl=128 time=442 ms
[1731328699.487614] 64 bytes from 41.186.255.86: icmp seq=17 ttl=128 time=443 ms
[1731328699.687193] 64 bytes from 41.186.255.86: icmp seq=18 ttl=128 time=442 ms
[1731328699.888824] 64 bytes from 41.186.255.86: icmp seq=19 ttl=128 time=443 ms
[1731328700.213673] 64 bytes from 41.186.255.86: icmp seq=20 ttl=128 time=566 ms
[1731328700.289045] 64 bytes from 41.186.255.86: icmp seq=21 ttl=128 time=442 ms
[1731328700.489162] 64 bytes from 41.186.255.86: icmp seq=22 ttl=128 time=442 ms
[1731328700.689799] 64 bytes from 41.186.255.86: icmp seq=23 ttl=128 time=442 ms
[1731328700.890927] 64 bytes from 41.186.255.86: icmp seq=24 ttl=128 time=442 ms
[1731328701.091816] 64 bytes from 41.186.255.86: icmp seq=25 ttl=128 time=442 ms
[1731328701.291825] 64 bytes from 41.186.255.86: icmp seq=26 ttl=128 time=442 ms
[1731328701.492729] 64 bytes from 41.186.255.86: icmp seq=27 ttl=128 time=442 ms
[1731328701.693694] 64 bytes from 41.186.255.86: icmp seq=28 ttl=128 time=442 ms
[1731328701.894467] 64 bytes from 41.186.255.86: icmp seq=29 ttl=128 time=441 ms
[1731328702.095540] 64 bytes from 41.186.255.86: icmp seq=30 ttl=128 time=442 ms
[1731328702.295523] 64 bytes from 41.186.255.86: icmp seq=31 ttl=128 time=442 ms
[1731328702.496778] 64 bytes from 41.186.255.86: icmp seq=32 ttl=128 time=442 ms
[1731328702.696974] 64 bytes from 41.186.255.86: icmp seq=33 ttl=128 time=442 ms
[1731328702.898120] 64 bytes from 41.186.255.86: icmp seq=34 ttl=128 time=442 ms
[1731328703.099754] 64 bytes from 41.186.255.86: icmp seq=35 ttl=128 time=442 ms
[1731328703.299858] 64 bytes from 41.186.255.86: icmp seq=36 ttl=128 time=442 ms
[1731328703.500740] 64 bytes from 41.186.255.86: icmp seq=37 ttl=128 time=442 ms
[1731328703.701889] 64 bytes from 41.186.255.86: icmp seq=38 ttl=128 time=442 ms
[1731328703.902045] 64 bytes from 41.186.255.86: icmp seq=39 ttl=128 time=441 ms
[1731328704.103177] 64 bytes from 41.186.255.86: icmp seq=40 ttl=128 time=442 ms
[1731328704.303828] 64 bytes from 41.186.255.86: icmp seq=41 ttl=128 time=442 ms
[1731328704.505320] 64 bytes from 41.186.255.86: icmp seq=42 ttl=128 time=442 ms
[1731328704.705538] 64 bytes from 41.186.255.86: icmp seq=43 ttl=128 time=442 ms
[1731328704.906721] 64 bytes from 41.186.255.86: icmp seq=44 ttl=128 time=442 ms
[1731328705.221217] 64 bytes from 41.186.255.86: icmp seq=45 ttl=128 time=555 ms
[1731328705.309083] 64 bytes from 41.186.255.86: icmp seq=46 ttl=128 time=442 ms
[1731328705.510986] 64 bytes from 41.186.255.86: icmp seq=47 ttl=128 time=442 ms
[1731328705.711191] 64 bytes from 41.186.255.86: icmp seq=48 ttl=128 time=442 ms
[1731328705.911602] 64 bytes from 41.186.255.86: icmp seq=49 ttl=128 time=442 ms
```

行 20，列 81 | 3,394,059 个字符 | 100% | Unix (LF) | UTF-8

写一个python代码，解析操作：
前置操作：

```python
import re
import pandas as pd
import matplotlib.pyplot as plt

# 设置中文字体 如果不设置中文显示会有问题
plt.rcParams['font.sans-serif'] = ['SimHei']  # 使用黑体
plt.rcParams['axes.unicode_minus'] = False    # 解决负号显示问题

# 读取data.txt文件
with open('data.txt', 'r', encoding='utf-8') as f:
    lines = f.readlines()

# 初始化数据列表
seq_nums = []
rtts = []
timestamps = []

# 正则表达式匹配icmp_seq和RTT
pattern = re.compile(r'icmp_seq=(\d+).*time=([\d\.]+) ms')
timestamp_pattern = re.compile(r'\[(\d+\.\d+)\]')

# 解析每一行，提取序列号和RTT
for line in lines:
    line = line.strip()
    if 'icmp_seq' in line:
        seq_match = pattern.search(line)
        time_match = timestamp_pattern.search(line)
        if seq_match and time_match:
            seq_num = int(seq_match.group(1))
            rtt = float(seq_match.group(2))
            timestamp = float(time_match.group(1))
            seq_nums.append(seq_num)
            rtts.append(rtt)
            timestamps.append(timestamp)

# 创建DataFrame
data = pd.DataFrame({
    'seq_num': seq_nums,
    'rtt': rtts,
    'timestamp': timestamps
})

# 找出所有发送的序列号范围
all_seq_nums = set(range(data['seq_num'].min(), data['seq_num'].max() + 1))

# 找出缺失的序列号（即未收到回复的请求）
received_seq_nums = set(data['seq_num'])
lost_seq_nums = sorted(all_seq_nums - received_seq_nums)

# 如果存在缺失的序列号，才创建缺失数据的DataFrame并进行拼接 这里算是个坑吧
if lost_seq_nums:
    # 创建缺失数据的DataFrame
    missing_data = pd.DataFrame({
        'seq_num': lost_seq_nums,
        'rtt': [None] * len(lost_seq_nums),
        'timestamp': [None] * len(lost_seq_nums)
    })

    # 合并数据
    data = pd.concat([data, missing_data], ignore_index=True)

# 按序列号排序
data = data.sort_values('seq_num').reset_index(drop=True)
```

Q1

```
total_requests = data.shape[0]
total_replies = data['rtt'].count()
delivery_rate = total_replies / total_requests

print(f"1. 整体传输成功率: {delivery_rate:.2%}")
```

输出结果：整体传输成功率：99.83%

Q2

```
success_flags = data['rtt'].notnull().astype(int)
max_success_streak = 0
current_streak = 0
for flag in success_flags:
    if flag == 1:
        current_streak += 1
        if current_streak > max_success_streak:
            max_success_streak = current_streak
    else:
        current_streak = 0

print(f"2. 最长连续成功回复的次数: {max_success_streak}")
```

输出结果：最长连续成功回复的次数：11856

Q3

```
failure_flags = data['rtt'].isnull().astype(int)
max_failure_streak = 0
current_streak = 0
for flag in failure_flags:
    if flag == 1:
        current_streak += 1
        if current_streak > max_failure_streak:
            max_failure_streak = current_streak
    else:
        current_streak = 0
print(f"3. 最长连续丢包的次数: {max_failure_streak}")
```

输出结果：最长连续丢包的次数：62

Q4

```
transitions = list(zip(success_flags[:-1], success_flags[1:]))

# 计算P(下一次成功 | 当前成功)
success_to_success = sum(1 for a, b in transitions if a == 1 and b == 1)
total_success = sum(1 for a, b in transitions if a == 1)
success_after_success = success_to_success / total_success if total_success != 0 else 0

# 计算P(下一次成功 | 当前失败)
failure_to_success = sum(1 for a, b in transitions if a == 0 and b == 1)
total_failure = sum(1 for a, b in transitions if a == 0)
success_after_failure = failure_to_success / total_failure if total_failure != 0 else 0

print(f"   在当前请求成功的情况下，下一次请求成功的概率：{success_after_success:.2%}")
print(f"   在当前请求失败的情况下，下一次请求成功的概率：{success_after_failure:.2%}")
print(f"   总体的传输成功率：{delivery_rate:.2%}")
```

输出结果:
在当前请求成功的情况下，下一次请求成功的概率：99.99%
在当前请求失败的情况下，下一次请求成功的概率：7.46%
总体的传输成功率：99.83%

Q5&Q6

```
min_rtt = data['rtt'].min()
print(f"5. 最小RTT：{min_rtt} ms")

max_rtt = data['rtt'].max()
print(f"6. 最大RTT：{max_rtt} ms")
```

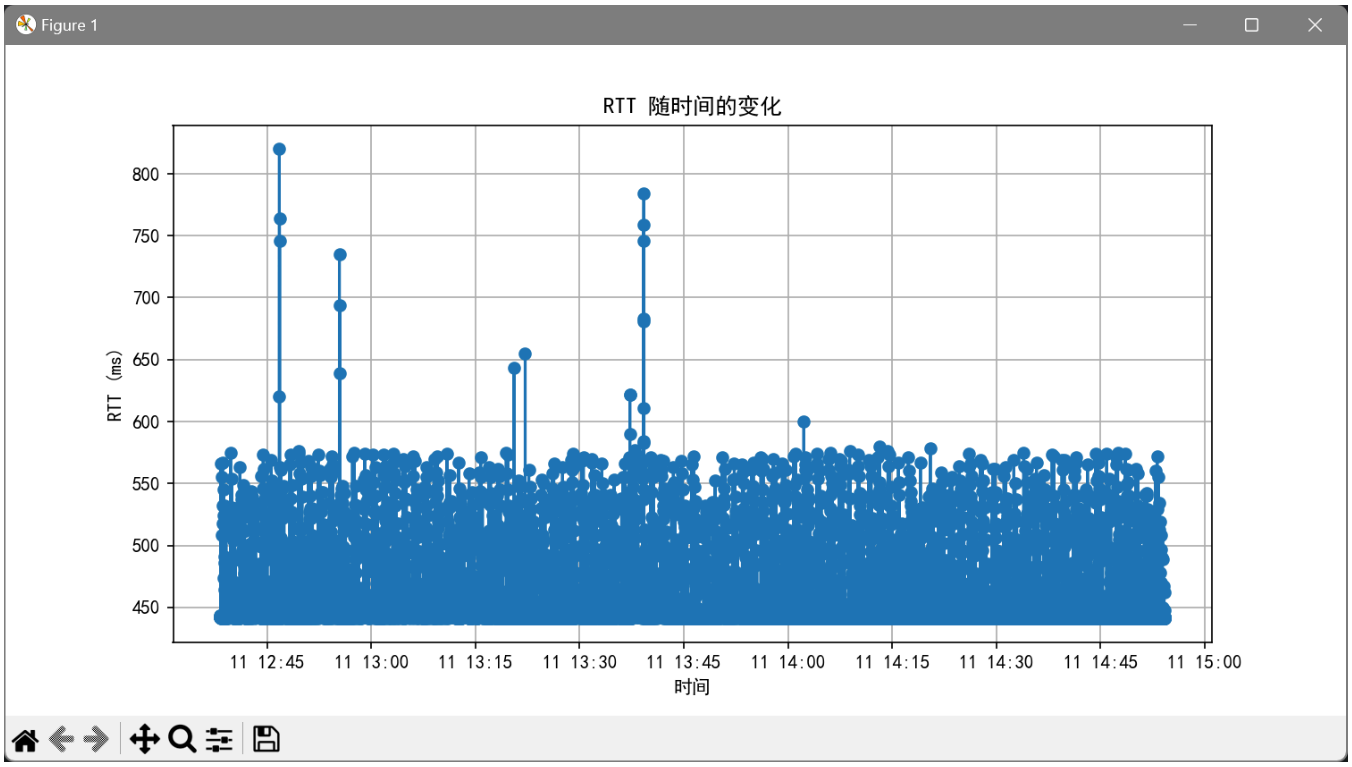
输出结果:
5. 最小RTT：441.0 ms
6. 最大RTT：820.0 ms

Q7&Q8&Q9

由于对绘图库不熟悉，这部分在ai的帮助下进行了学习和编码。

```
# 绘制RTT随时间变化的图形
plt.figure(figsize=(10, 5))
plt.plot(pd.to_datetime(data['timestamp'], unit='s'), data['rtt'], marker='o')
plt.xlabel('时间')
plt.ylabel('RTT (ms)')
plt.title('RTT 随时间的变化')
plt.grid(True)
plt.savefig('rtt_over_time.png')
plt.show()

# 绘制RTT的直方图和累计分布函数
plt.figure(figsize=(10, 5))
plt.hist(data['rtt'].dropna(), bins=30, density=True, alpha=0.6, color='g')
plt.xlabel('RTT (ms)')
plt.ylabel('概率密度')
plt.title('RTT 分布直方图')
plt.grid(True)
plt.savefig('rtt_histogram.png')
plt.show()

plt.figure(figsize=(10, 5))
sorted_rtt = data['rtt'].dropna().sort_values()
cdf = sorted_rtt.rank(method='first') / len(sorted_rtt)
plt.plot(sorted_rtt, cdf)
plt.xlabel('RTT (ms)')
plt.ylabel('累计概率')
plt.title('RTT 累计分布函数')
plt.grid(True)
plt.savefig('rtt_cdf.png')
plt.show()

# 绘制RTT的自相关图
rtt_series = data['rtt'].dropna().reset_index(drop=True)
rtt_n = rtt_series[:-1]
rtt_n_plus_1 = rtt_series[1:]

plt.figure(figsize=(7, 7))
plt.scatter(rtt_n, rtt_n_plus_1)
plt.xlabel('第N次的RTT (ms)')
plt.ylabel('第N+1次的RTT (ms)')
plt.title('RTT的自相关性')
plt.grid(True)
plt.savefig('rtt_correlation.png')
plt.show()

# 计算相关系数
correlation = rtt_n.corr(rtt_n_plus_1)
print(f"9. RTT的相关系数：{correlation:.2f}")
```

输出结果：



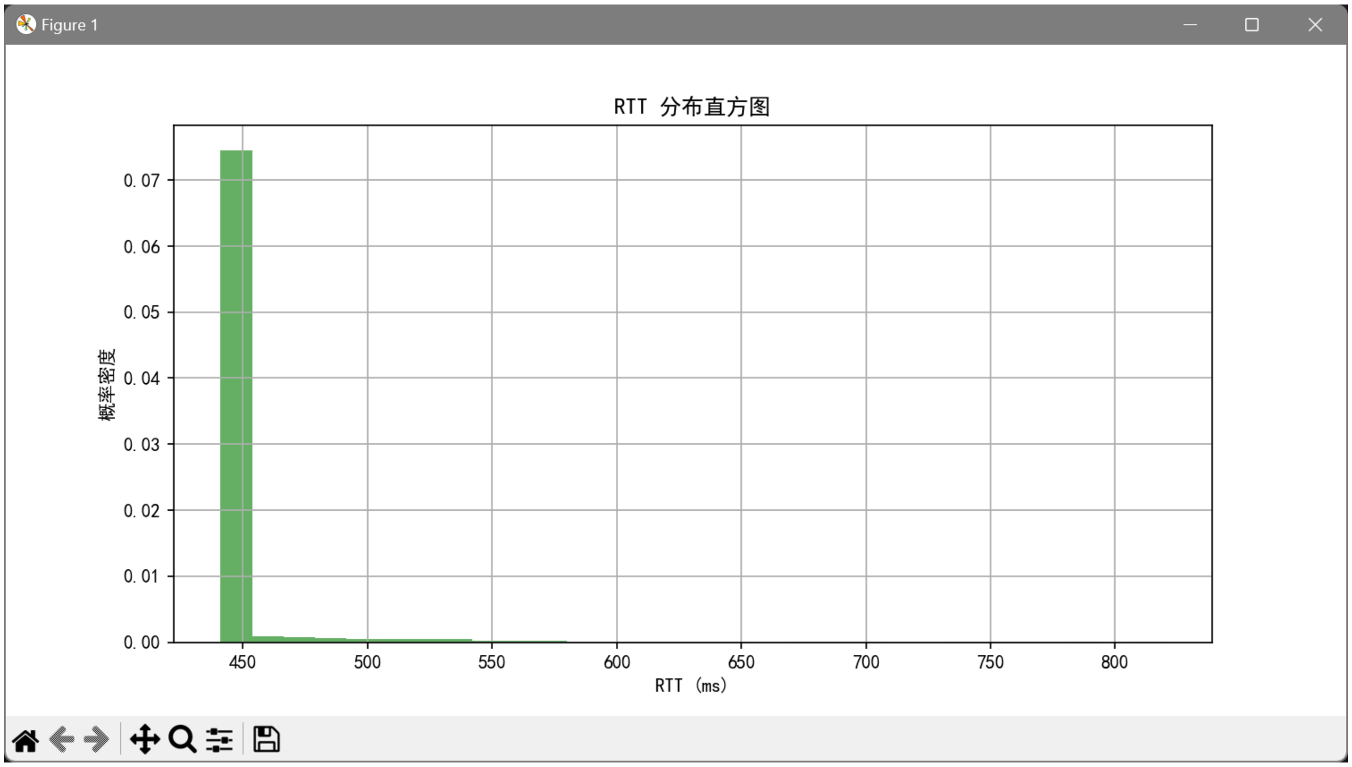


Figure 1
RTT 分布直方图
概率密度
0. 00
0. 01
0. 02
0. 03
0. 04
0. 05
0. 06
0. 07
450
500
550
600
650
700
750
800
RTT (ms)

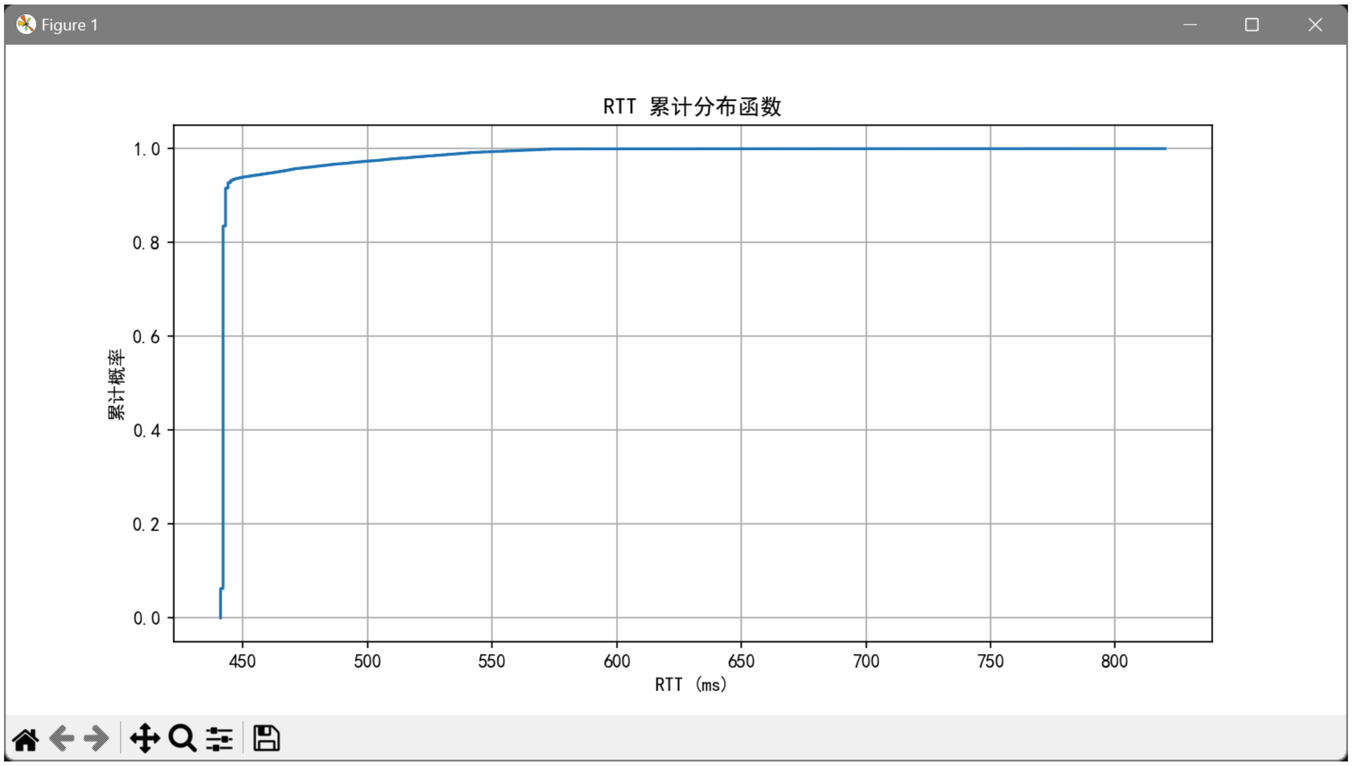
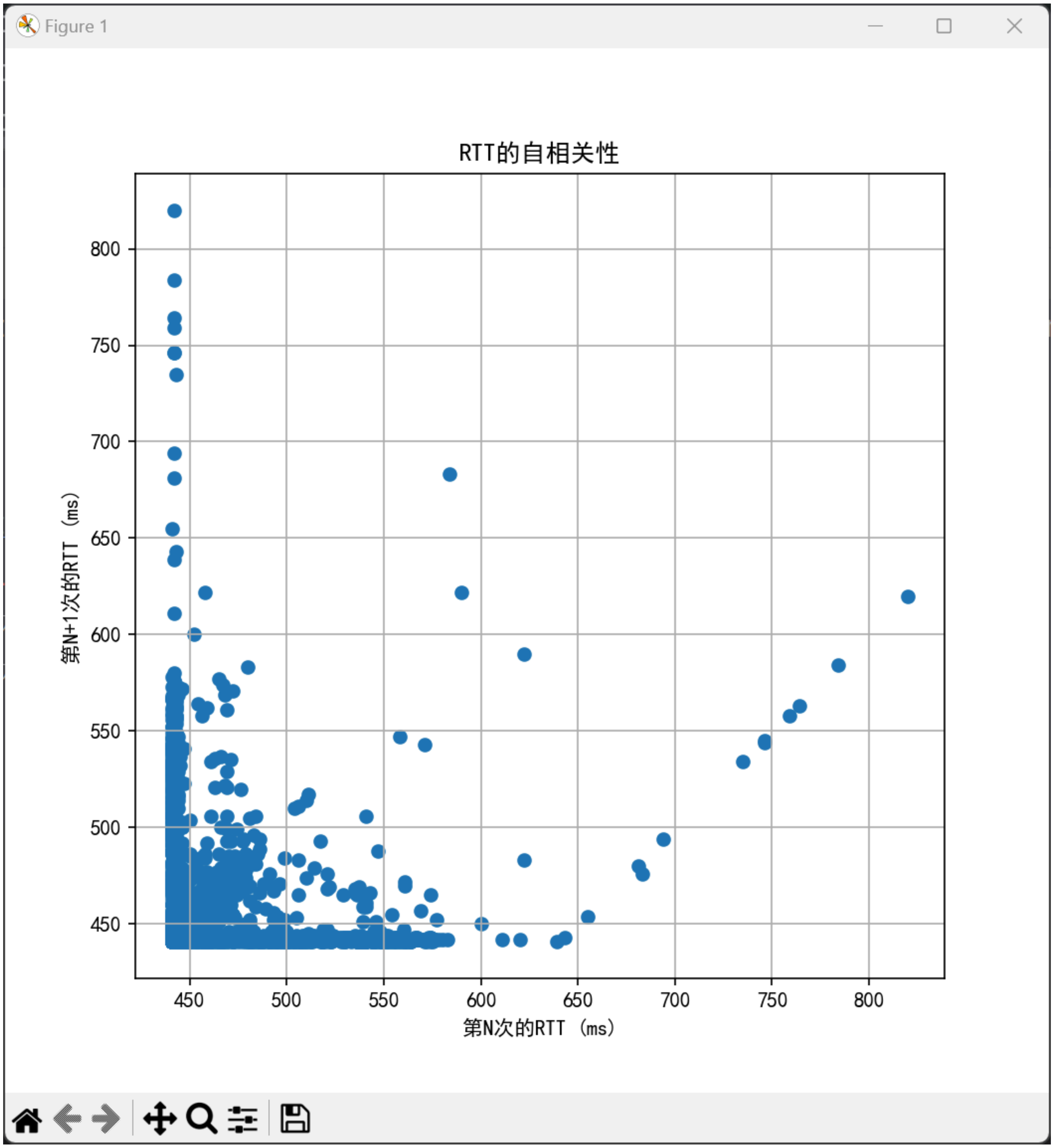


Figure 1
RTT 累计分布函数
累计概率
0. 0
0. 2
0. 4
0. 6
0. 8
1. 0
450
500
550
600
650
700
750
800
RTT (ms)



Q10

由此可见，网络的整体传输效率较高，丢包的概率很低。但是一旦发生丢包，那么接下来的丢包概率还是很高。